82/783 Sc N 82/327 A

Université de Nancy I

U.E.R. Sciences Mathématiques

Centre de Recherche en Informatique de Nancy

BIBLIOTHEQUE-SCIENCES * UNIVERSITE NANCY *

SPECIFICATION ET CONSTRUCTION DE SYSTEMES ORIENTES BASES DE DONNEES: TECHNIQUES ET LANGAGES BASES SUR LE CONCEPT DE TYPE ABSTRAIT

THESE

présentée et soutenue publiquement le **29 octobre 1982**

A L'UNIVERSITE DE NANCY I

pour l'obtention du grade de

DOCTEUR ES-SCIENCES MATHEMATIQUES

par

Jean-Jacques CHABRIER

devant la Commission d'Examen

Président : ..... *J.C. DERNIAME*

Rapporteurs : ..... *P.Y. CUNIN*

J.P. JOUANNAUD

S. MIRANDA

Examinateurs : ..... *J.P. FINANCE*

J. LE BIHAN

C. PAIR

Université de Nancy I

U.E.R. Sciences Mathématiques

Centre de Recherche en Informatique de Nancy

SPECIFICATION ET CONSTRUCTION DE SYSTEMES ORIENTES BASES DE DONNEES: TECHNIQUES ET LANGAGES BASES SUR LE CONCEPT DE TYPE ABSTRAIT

THESE

présentée et soutenue publiquement le **29 octobre 1982**

A L'UNIVERSITE DE NANCY I

pour l'obtention du grade de

DOCTEUR ES-SCIENCES MATHEMATIQUES

par

Jean-Jacques CHABRIER

devant la Commission d'Examen

Président : ..... *J.C. DERNIAME*

Rapporteurs : ..... *P.Y. CUNIN*
J.P. JOUANNAUD
S. MIRANDA

Examinateurs : ..... *J.P. FINANCE*
J. LE BIHAN
C. PAIR

A la mémoire de mes parents

A Jacqueline

A Christophe et Alain

Je tiens à remercier ici les membres du jury:

J.C.DERNIAME Directeur du Centre de Recherche en Informatique de Nancy et Professeur à Nancy I a été depuis mon D.E.A. et jusqu'à cette thèse le guide et le conseiller de toute mon activité de recherche. Je lui suis profondément reconnaissant de la grande confiance qu'il m'a témoignée en me laissant prendre en main l'animation d'un projet de recherche dont il est l'instigateur.

P.Y.CUNIN Professeur à l'Université de Dijon est un spécialiste des langages de programmation. Sa compétence, sa confiance et son amitié me sont précieuses. Je suis très heureux qu'il soit dans ce jury en tant que rapporteur, tâche dont il s'est acquitté avec intérêt, application et diligence.

J.P.JOUANNAUD Professeur à l'Université de Nancy II est un sémanticien qui a su, en quelques années, développer une équipe de recherche dynamique et efficace. Je lui suis très reconnaissant d'avoir bien voulu s'intéresser à mon travail.

S.MIRANDA Chercheur à l'Institut National de Recherche en Informatique et Directeur de Recherche au CERISS de Toulouse est un spécialiste des bases de données. C'est avec gentillesse et une grande compétence qu'il assume aujourd'hui le rôle de rapporteur.

J.P.FINANCE Directeur du Département de Mathématiques Appliquées est un spécialiste de la spécification et de la construction de logiciel. Il m'a, à plusieurs reprises, conseillé sur certains problèmes de spécification, notamment à propos des mécanismes de construction. C'est un grand plaisir pour moi de le voir siéger à ce jury.

J.LE BIHAN Directeur de l'Aide à la Recherche à l'Agence de l'Informatique est un spécialiste des bases de données réparties. Il a toujours apporté un soutien amical et confiant à mon travail, notamment dans le cadre du projet pilote SIRIUS qu'il a dirigé.

C.PAIR Professeur à l'Institut National Polytechnique de Lorraine, actuellement Directeur des Lycées au Ministère de l'Education Nationale, me fait un grand honneur en participant à ce jury. Je lui exprime ma profonde gratitude pour m'avoir fait bénéficier de son exceptionnelle compétence scientifique et pour l'intérêt qu'il a toujours porté aux travaux de notre équipe et pour son soutien permanent.

Je n'oublie pas M.GRIFFITHS Directeur du Laboratoire d'Informatique pour les Sciences de l'Homme qui avait sa place dans ce jury si des calculs malencontreux ne l'en avaient empêché. Ses conseils, ses encouragements m'ont été d'un très grand réconfort dans les moments de doute. Qu'il soit assuré de ma profonde sympathie.

J'associe dans l'expression de ma gratitude:

Jacqueline, ingénieur au CNRS, qui a accepté plusieurs années durant de sacrifier des nuits et des vacances à discuter d'informatique et à mettre au point le système VEGA. La réalisation de cette thèse sous GUTENBERG a mis sa patience à rude épreuve. Sans elle, ce travail n'aurait certainement pas abouti,

les chercheurs qui ont travaillé sur le projet TYP: Patrick HENRY, Régis MINOT, Karol PROCH et Najib TOUNSI qui ont montré une compétence et une détermination qui m'ont beaucoup impressionné,

les chercheurs du projet TYP-R: Claude GODARD, Thiéry GRISON, Rodolphe NASSIF et Najib TOUNSI qui ont contribué à démontrer la faisabilité de nos idées, en réalisant un système de gestion de base de données relationnel,

les chercheurs qui ont collaboré au projet MONT-LOZERE: Nacer BOUDJLIDA, Jacques BOURST et Jacques DURAND. Leur motivation et leur

compétence nous ont permis de réaliser une maquette relativement importante de système d'aide à l'information pour la région de Lozère,

les chercheurs du CRIN qui m'ont aidé durant mes recherches: Jacques DUCLOY Chargé de Mission à l'ANL qui a beaucoup contribué à la diffusion de nos travaux, mes amis de l'IUCAL pour leur patience que j'ai mise à rude épreuve depuis plusieurs années, mes collègues enseignants de l'UER de Mathématiques et Informatique,

et Danièle MARCHAND pour son aide efficace.

Enfin, je remercie ALAIN et CHRISTOPHE ainsi que tous les membres de ma famille pour leur compréhension, leur dévouement et leur appui constant.

SOMMAIRE

INTRODUCTION

PREMIERE PARTIE: ETUDE DE LA PROBLEMATIQUE DE LA SPECIFICATION ET DE LA CONSTRUCTION DES SYSTEMES LOGICIELS D'APPLICATION

Chapitre I. Problématique de la construction des systèmes logiciels d'application

Chapitre II. Approche systèmes de gestion de base de données

Chapitre III. Approche langages de programmation généraux et spécialisés

Chapitre IV. Langages orientés systèmes de base de données

DEUXIEME PARTIE: PRESENTATION DU SYSTEME VEGA D'AIDE A LA SPECIFICATION ALGEBRIQUE DE TYPES ABSTRAITS ET/OU DE PROGRAMMES ET A LA VALIDATION

Chapitre V. Problématique de la spécification de problèmes

Chapitre VI. Présentation de langages (ou systèmes) de spécification expérimentaux

Chapitre VII. Types abstraits algébriques avec erreurs et systèmes de réécriture

Chapitre VIII. Notions de base du calcul des prédicats. Présentation de PROLOG

Chapitre IX. VEGA

TROISIEME PARTIE: PRESENTATION DU SYSTEME TYP DE PROGRAMMATION MODULAIRE

Chapitre X. Concepts de modularité et d'abstraction du système TYP

Chapitre XI. Notion d'application: application référence, application sectorielle, application connexion

Chapitre XII. Le système interactif d'aide à la construction, à la modification et à l'utilisation des A.B.D.

Chapitre XIII. Remarques sur la réalisation du système expérimental TYP

CONCLUSION

ANNEXES

I. Exemples TYP.

II. Exemples VEGA.

INTRODUCTION .

En dépit du nombre considérable de "Systèmes Logiciels d'Applications" qui sont conçus, réalisés et exploités chaque jour dans les entreprises, il existe fort peu de techniques et d'outils logiciels pour les développer de façon cohérente et efficace depuis la spécification jusqu'à la mise en oeuvre et la maintenance.

De prime abord, cela paraît étonnant, d'autant que les diverses branches de l'informatique: Conception de Langages de Programmation, Bases de Données, Intelligence Artificielle (Systèmes Experts), Systèmes d'Exploitation, etc... ont beaucoup évolué. En fait, avec le recul de quelques années, il faut reconnaître que les progrès constants et rapides qui ont été accomplis dans ces branches ont eu pour effets de renforcer le cloisonnement des recherches, et par voie de conséquence, de privilégier une conception de logiciels pour les seuls spécialistes des branches. Il n'est peut être pas inutile de rappeler que pendant de nombreuses années, la particularisation des "univers" des domaines d'application de l'informatique, comme par exemple l'automatisation de la gestion administrative dans les organisations, a constitué une véritable "barrière psychologique" pour bon nombre de chercheurs en informatique. Ce n'est qu'après s'être rendu compte de l'énorme difficulté et souvent de l'impossibilité de rapprocher les concepts, les techniques et les outils logiciels des différentes branches pour résoudre de façon cohérente les problèmes de développement des grands Systèmes Logiciels d'Application que les

premières recherches en génie logiciel ont débuté (KRA80). L'expansion actuelle de la production de logiciels d'application de plus en plus sophistiqués a permis d'une certaine façon de focaliser quelques problèmes qui jusque là avaient été négligés par les chercheurs: la qualité et le coût de développement, notamment. Si les travaux qui sont actuellement menés dans le cadre du génie logiciel sur la nature et la complexité des problèmes de développement de grands Systèmes Logiciels d'Application sont encore balbutiants, il est tout de même symptomatique de constater qu'au niveau des idées, au moins, un certain consensus semble se dégager. Nous sommes même en lieu de penser que les intentions générales qui ont été formulées par S.KRAKOWIAK (KRA80) dans le cadre d'une préétude sur le génie logiciel trouveront un large écho dans un avenir proche. Pour notre part, il nous est apparu dès 1976 (CHA76) que l'on ne pouvait espérer progresser sur les concepts, les mécanismes et les outils d'aide à la spécification et à la production de Systèmes Logiciels d'Application qu'au prix de nouvelles investigations s'appuyant sur les propositions et expérimentations récentes. La démarche qui nous a conduit à faire ce constat puis à entreprendre le projet TYP et ensuite le projet VEGA résulte à la fois:

--du travail de recherche effectué sur les mécanismes et les outils du système de programmation modulaire CIVA avec J.C.DERNIAME (DER74)

--de l'expérience pédagogique acquise avec l'enseignement des systèmes de gestion de bases de données, des langages de programmation COBOL, PL/1, PASCAL, etc..., des systèmes d'informations mais aussi de la méthode déductive conçue par C.PAIR

--des résultats théoriques obtenus dans le domaine de la spécification formelle, de la réécriture mais aussi de l'expérience acquise en programmation logique avec PROLOG (ROU75).

La première partie de cette thèse est une étude de la problématique de la spécification et de la construction de Systèmes Logiciels d'Application. Après un bref rappel de quelques notions sur la qualité d'un logiciel et sur les différentes étapes du développement, nous procédons à une analyse critique des techniques et outils logiciels actuellement utilisés pour concevoir et réaliser des logiciels (Chapitre I). Nous explicitons ensuite, à partir des premières conclusions qui ont été tirées, les idées essentielles ainsi que les buts visés: proposer des concepts, des techniques et des outils logiciels qui fournissent un continuum d'expression depuis la spécification jusqu'à l'exploitation des Systèmes Logiciels d'Application . Nous allons maintenant préciser un peu ce que nous avons appelé à dessein Système Logiciel d'Application. Nous pouvons dire comme S.KRAKOWIAK que c'est "un système intégré de gestion de l'ensemble des informations d'un projet informatique, depuis les spécifications jusqu'aux programmes exécutables" (KRA80). Ce vaste champ d'études nous a naturellement amené à entreprendre d'importantes investigations sur les Systèmes de Gestion de Bases de Données (Chapitre II), sur les langages de programmation généraux et spécialisés (Chapitre III), et sur les langages de programmation orientés bases de données (Chapitre IV). Par rapport au très large éventail de propositions d'études suggérées par KRAKOWIAK, on ne s'est intéressé qu'à certaines des fonctions des Systèmes Logiciels d'Application avec toutefois le souci premier de rapprocher sinon d'unifier des concepts et des techniques des Systèmes de Bases de Données et des langages de programmation modulaires. L'usage d'outils logiciels homogènes et intégrés pose par ailleurs des problèmes de sécurité importants et complexes que nous abordons rapidement.

La deuxième partie est dédiée à la spécification de problèmes et à la validation de spécification par rapport aux intentions initiales à l'aide de maquettes symboliques. Dans l'étude de la problématique de la spécification de problèmes que nous exposons au chapitre V, on a dégagé les principales fonctions que nous semblent devoir remplir une technique de spécification:

--c'est un contrat entre l'utilisateur et le producteur du logiciel,

--c'est un outil efficace d'aide à la conception logique du logiciel,

--c'est une aide à l'implantation,

--enfin, c'est aussi une aide efficace à l'évolution du logiciel.

A travers la brève étude de quelques systèmes expérimentaux d'aide à la spécification de problèmes ou d'aide à la spécification formelle et à la validation de programmes (Chapitre VI), on peut se rendre compte que la plupart de ces langages (ou systèmes) ne remplissent que partiellement les fonctions de spécification précédentes.

Le chapitre VII présente les notions théoriques fondamentales qui sont à la base de la conception du système d'aide à la spécification algébrique VEGA: types abstraits algébriques avec erreurs, systèmes de réécriture notamment. Le chapitre VIII est dévolu à un bref exposé du langage de programmation logique PROLOG (ROU75) que nous avons utilisé pour réaliser le système VEGA. Le système expérimental VEGA qui a été développé sur l'ordinateur IRIS80 de l'Institut Universitaire de Calcul Automatique de Lorraine est un système interactif "multicibles" qui intègre:

--un système d'aide à la spécification algébrique de types

abstraits et/ou de programmes fonctionnels. Le langage formel appelé VEGA supporte deux techniques de spécification ainsi que plusieurs mécanismes puissants d'aide à la spécification;

--et un système d'aide à l'élaboration de maquettes de données symboliques. Le langage de programmation utilisé doit être considéré comme une extension de PROLOG utilisant un environnement de types abstraits et/ou de programmes fonctionnels spécifiés algébriquement.

Les traits de conception du système VEGA sont exposés dans le chapitre IX.

La troisième partie est consacrée à la présentation des concepts, des mécanismes et des outils logiciels qui ont été développés dans le cadre du projet TYP. TYP est un système interactif de programmation modulaire utilisant la notion de type abstrait qui intègre un système de compilation séparée, un système de généricité, des documentalistes et gestionnaires, un éditeur de textes et un système de fabrication de programmes appelé connecteur. Les concepts de modularité et d'abstraction du système TYP (Chapitre X) permettent d'implanter de façon naturelle et progressive les spécifications exprimées en VEGA. Nous avons, de plus, développé des notions d'abstraction (Chapitre XI) qui devraient apporter des aides à la conception et à l'utilisation des Applications Logicielles orientées Bases de Données (Chapitre XII).

Cette thèse a été voulue aussi lisible et complète en elle-même que possible. Les rappels substanciels de concepts, techniques ou langages qui concernent la spécification, les bases de données et les langages (ou systèmes) de programmation peuvent donc être ignorés par les spécialistes de ces domaines.

PREMIERE PARTIE

ETUDE DE LA PROBLEMATIQUE DE LA SPECIFICATION

ET DE LA CONSTRUCTION DES

SYSTEMES LOGICIELS D'APPLICATION

CHAPITRE I. PROBLEMATIQUE DE LA CONSTRUCTION DES SYSTEMES LOGICIELS D'APPLICATION.

I. LES SYSTEMES LOGICIELS D'APPLICATION.

I.1. Qualités d'un logiciel.

I.2. Développement systématique de logiciel.

I.3. Remarques générales.

II. ETUDE D'UN EXEMPLE DE CONSTRUCTION DE LOGICIEL D'APPLICATION.

II.1. Analyse et définition d'un premier énoncé.

II.2. Conception et explicitation de l'énoncé.

II.3. Construction du système logiciel d'application.

II.3.1. Approche système de programmes d'application.

II.3.1.1. Définition des algorithmes du problème.

II.3.1.2. Programmation du système d'application.

II.3.1.3. Remarques.

1.sur la réalisation.

2.sur la structuration du système de programmes.

3.sur l'évolution du système de programmes.

II.3.2. Approche système de base de données d'application.

II.3.2.1. Programmation du système de base de données.

II.3.2.2. Remarques.

1.sur la réalisation.

2.sur la structuration du système de base de données.

3.sur l'évolution du système de base de données.

II.3.3. Approche système logiciel "mixte".

II.3.3.1. Remarques.

1.sur la réalisation.

2.sur la structuration du système "mixte".

III. PREMIERES CONCLUSIONS.

I. LES SYSTEMES LOGICIELS D'APPLICATION.

Avant de donner un synopsis des différentes étapes du développement d'un système logiciel, il nous faut définir ce qu'est la qualité d'un logiciel.

I.1. Qualités d'un logiciel.

C'est l'ensemble des propriétés que l'on souhaiterait retrouver pour les systèmes logiciels produits.

1-- la sureté : le système ne doit pas tomber en panne lorsqu'on l'utilise,

2-- la fidélité : le système doit fournir des résultats conformes aux spécifications,

3-- la facilité d'utilisation : le système doit être facilement utilisable non seulement par l'informaticien qui l'a conçu, mais aussi par la classe des utilisateurs auxquels il est destiné,

4-- l'extensibilité : le système doit pouvoir s'adapter aussi bien à l'évolution des besoins des utilisateurs qu'à la maintenance au sens large,

5-- la performance : le système doit être capable de fournir les résultats escomptés tout en satisfaisant aux contraintes d'espace

mémoire et de temps de réponse imposées,

6-- la sécurité : le système et les informations qu'il gère doivent être protégés contre les utilisations accidentelles ou intentionnelles,

7-- le coût : le coût de développement du système, de son utilisation et de sa maintenance doit être "raisonnable" et justifié.

Il est clair que ces propriétés générales peuvent être appliquées à toutes les sortes de systèmes logiciels de "base" comme les compilateurs, les éditeurs de liens, les systèmes de gestion de bases de données, etc... ou d'applications utilisateurs spécifiques développées à l'aide de ces outils standards.

Il nous importe donc d'apporter des précisions sur ce que nous entendons par application lorsque nous parlons de Système Logiciel d'Applications. Nous avons déjà dit dans l'introduction que nous nous référions au sens général de Projet Informatique mais nous pouvons maintenant préciser avec la perspective du développement "cohérent" d'un ensemble intégré de systèmes logiciels qui concourent à la réalisation du Projet. Un projet informatique, en toute généralité, recouvre toutes les applications qui peuvent être développées sur un système opératoire. En bref, un système logiciel d'applications est un système intégré de logiciels qui fournit un environnement de "qualité" pour les applications d'un projet informatique. Cette idée essentielle, déjà perçue dans le projet CIVA (DER74) nous amène naturellement à souhaiter appliquer les mêmes techniques de développement à tous les logiciels d'applications.

I.2. Développement systématique de logiciel.

Résumons alors les étapes de ce qui est appelé "la discipline de développement systématique" en ingénièrie de logiciel:

1-- l'analyse des besoins et la spécification des problèmes (ABR74) (BRO78) (GOG79) (ABR77) (SMI77) (CHE76) (CHE77) (PAI74) (PAI75) (ROS77): l'étude de cette étape est encore balbutiante. Elle consiste à analyser et à définir des problèmes, des connaissances, des faits et des événements dans un modèle formel (modèle est ici pris au sens de modèle mathématique ou logique). Les propositions de J.P.FINANCE (FIN79) consistent à définir à l'aide d'un langage formel --calcul des prédicats typé-- un énoncé statique et structuré qui soit aussi fidèle que possible --intuitivement-- à l'énoncé initial. La technique de spécification esquissée procède par analyse déductive et description en parallèle de l'énoncé du problème caractérisant la relation entre les données et les résultats, et de l'univers du problème qui comporte la description des ensembles, fonctions et prédicats nécessaires à la compréhension de l'énoncé.

2-- la conception (ABR78) (VLDB78): la démarche générale consiste à passer d'une première spécification "conceptuelle" à un énoncé explicite constructif. Dans la discipline des bases de données , nous dirions d'un schéma conceptuel à un schéma d'accès logique (CAB75).

3-- la construction de schémas logiques de bases de données et de programmes de manipulation à l'aide d'un système de gestion de bases de données et/ou la construction de programmes corrects, efficaces, sûrs, etc... à l'aide de techniques de décomposition (PAR71) (PAR72) et de raffinements (WIR71) (DAH72) (DEN73).

4-- le test: construction de jeux d'essais pour déterminer à la fois la correction et les performances des programmes (MIL71) (POO73).

5-- la vérification formelle de l'adéquation des programmes à la spécification de l'énoncé.

6-- la certification du logiciel qui permet d'établir si le produit possède les qualités de sûreté et de performance annoncées (TSI73).

7-- la définition des environnements et des outils d'aide à la réalisation.

8-- la documentation du logiciel: les spécifications formelles et informelles, les textes des programmes, etc... doivent être gérés (GOO73). Dans l'approche base de données, cette documentation est définie par la technique des dictionnaires de données.

9-- l'adaptabilité: les possibilités de maintenance corrective, adaptative doivent être définies conformément aux contraintes de spécification.

10-- la gestion du développement concerne l'organisation, la budgétisation, la formation du personnel, etc... (MIL71)

La littérature est suffisamment abondante pour que nous ne développions pas plus cette synthèse (NAU68) (SPR73) (BOE76) (ZEL78) (HEN80) (INF77). Notre but en la matière est surtout de contribuer à une certaine clarification du vocabulaire utilisé par différentes branches. Nous citerons une remarque significative d'A.WASSERMAN (WAS78): "..data base management and software development in general share the problem of inconsistent terminology. Different investigations and system developers continue to invent new terms to describe old concepts. This practice, frequently leads to ambiguity, as different

people have different understanding of the same term, to the invention of new jargon, and occasionally to outright confusion. Once again, the need to establish some fundamental concepts, along with a consistent vocabulary for their description is important...".

I.3. Remarques générales.

Clarifier le vocabulaire, rapprocher les techniques de développement de systèmes logiciels d'application ne signifie pas, tout au moins dans notre esprit, proposer un cadre rigide, ce qui serait utopique, mais, pour chaque étape du développement, selon les problèmes abordés, proposer des techniques, des modèles, des langages et des aides logicielles qui faciliteraient et amélioreraient la qualité du logiciel.

D'autre part, la polarisation sur un type d'application de gestion: les traitements administratifs dans les organisations masque ce que nous pensons être le véritable objectif des systèmes d'application orientés gestion, c'est-à-dire l'aide à la décision. Ceci nous semble important pour l'appréhension des problèmes orientés gestion. S'il est vrai que pour les problèmes de traitements administratifs souvent assimilés à la "gestion", il n'existe pas de modèles formels pour les analyser et les décrire, nous pensons que pour beaucoup d'autres problèmes: la gestion du trafic ferroviaire dans une gare, la gestion d'un stock de marchandise, la gestion d'un central téléphonique, etc..., il n'en va pas de même. Il nous semble que l'analyse et la

conception des applications orientées gestion relèvent souvent de plusieurs techniques de modélisation: mathématique, probabiliste, etc... et à ce titre doivent être développées avec des techniques spécifiques. Enfin, d'autres systèmes comme par exemple l'aide au diagnostic de maladies cardiaques sont abordés avec des techniques d'intelligence artificielle (TS079).

II. ETUDE D'UN EXEMPLE DE CONSTRUCTION DE LOGICIEL D'APPLICATION.

Nous nous référons au problème de remboursements d'emprunts bancaires dû à J.R.ABRIAL (ABR77) que nous avons étudié sous plusieurs angles (CHA78).

Enoncé initial :

Une banque prête de l'argent à des clients et on demande de construire un système gérant les remboursements. Un client est dit actif lorsqu'il a emprunté un certain montant d'argent qu'il n'a pas encore fini de rembourser. Chaque mois, chaque client actif effectue un versement qui peut être supérieur, égal ou inférieur (éventuellement nul) à une certaine mensualité. Le client peut donc prendre du retard, payer exactement sa mensualité ou anticiper le remboursement.

Pour chaque client, la mensualité est calculée d'après le montant du prêt et sa durée: elle est égale au quotient du montant par la différence entre la date de fin du prêt et la date de début du prêt.

Chaque client actif possède donc, chaque mois, un certain crédit qui est égal à la somme des versements qu'il a effectués depuis le début du prêt. De même, il possède un débit qui est égal à la somme des

mensualités depuis le début du prêt ou au montant total du prêt si le mois en question est postérieur à la date de fin du prêt. Chaque mois, un certain nombre de clients actifs ont terminé de rembourser leur emprunt. Ils ne sont plus actifs dès le mois suivant. Par ailleurs, chaque mois apparaissent de nouveaux clients (non actifs) qui commenceront à effectuer leur versement dès le mois suivant et seront dès lors considérés comme des clients actifs.

Cet énoncé initial, comme dans beaucoup de problèmes orientés traitements administratifs de gestion, comporte beaucoup de sous-entendus. Par exemple, nous ne savons pas si un client peut souscrire plusieurs emprunts, etc...

Il faut tout de même dire que cet énoncé de problème ne reflète pas la réalité du contenu du cahier des charges des utilisateurs. En particulier, il n'est pas toujours aussi évident de donner un énoncé initial d'un problème de gestion épuré, comme dans cet exemple, des contraintes technologiques.

Après ces remarques préliminaires, nous étudions maintenant le processus de développement de cette application avec les techniques, les langages et les outils logiciels traditionnels.

II.1. Analyse et définition d'un premier énoncé.

Dès cette étape déjà, selon sa "culture", plusieurs approches peuvent être utilisées pour définir "un premier énoncé" consistant -non contradictoire-, "complet", fidèle, etc..

La première approche consiste à "écrire" son propre énoncé en langue naturelle avec plus ou moins de méthode pour essayer de supprimer les contradictions, les imprécisions, etc... Cette façon de procéder est criticable, car on n'est pas à l'abri d'introduire de nouvelles ambiguités dues à la compréhension de la langue.

La deuxième approche possible, proposée par les concepteurs de bases de données (IFIP76), consiste à définir l'univers du problème à l'aide de schémas sémantiques de données (COD70) (COD71) (CHE76) (SMI77) (FOU82). Selon le "degré" de formalisme du modèle de données, certaines propriétés de l'énoncé sont soit négligées, soit définies en langue naturelle.

Enfin, une troisième approche possible est fondée sur l'utilisation de langages formels: théorie des ensembles (ABR78), théorie algébrique (GOG79) (GUT80), ou théorie logique du calcul des prédicats typé (FIN79). Mis à part les formalismes, la différence essentielle entre l'approche d'ABRIAL et les autres porte sur la technique de spécification. Comme le montre J.P.FINANCE (FIN79), il importe de disposer d'un langage formel qui supporte la spécification progressive, structurée, voire même déductive.

Comme le note J.R.ABRIAL, une difficulté importante rencontrée pour la spécification d'énoncés statiques est justement de pouvoir prendre en compte la notion de temps. Nous étudierons plus en détail ces différentes approches en deuxième partie.

II.2. Conception et explicitation de l'énoncé.

Dans l'approche base de données, cette étape permet de représenter le schéma conceptuel dans le schéma logique en prenant en considération les fonctions d'accès définies à partir de l'expression des besoins utilisateurs (BEN79). Plusieurs modèles de données logiques sont utilisés: le modèle standard (CAB75), le modèle d'accès de SENKO (SEN73), etc... Là encore, selon le niveau d'expression sémantique du modèle de données, certaines informations peuvent soit être négligées, soit représentées dans un formalisme qui "déforme" l'information conceptuelle (IFIP76). Pour l'approche par les langages formels de spécification, J.R.ABRIAL (ABR77) propose des règles de transformation systématiques --au nombre de 400-- alors que J.P.FINANCE (FIN79) définit une technique d'explicitation d'énoncés constructifs sur les univers en utilisant les notions de constructeurs et de spécificateurs de types abstraits algébriques (LES79) (REM82). Il montre entre autres la nécessité de disposer de stratégies d'explicitation d'énoncé.

Dans l'approche de GOGUEN, l'utilisation du langage de spécification OBJ (GOG79) permet dès ce niveau d'écrire des programmes (cf deuxième partie).

En l'absence actuellement de méthode, de langage et d'outil logiciel d'aide à la spécification d'énoncé expérimentés, nous nous sommes simplement limités à rappeler les démarches possibles. La documentation à ce stade d'analyse conception est souvent un simple document "papier" (BEN79).

II.3. Construction du système logiciel d'application.

Pour cette troisième étape, selon le choix du logiciel de base effectué, trois sortes de systèmes logiciels d'applications peuvent être construits:

-- un système de programmes d'application à l'aide d'un langage de programmation général COBOL, PL/1, PASCAL, SIMULA, etc..

-- un système de base de données avec un système de gestion de base de données (S.G.B.D.) IMS, CODASYL, SOCRATE, etc..

-- un système logiciel "mixte" qui utilise les possibilités d'un S.G.B.D. et d'un langage hôte du type COBOL ou PL/1 par exemple.

Nous allons étudier un peu plus en détail ces trois approches.

II.3.1. Approche système de programmes d'application.

Pour réaliser ce système, nous sommes amenés à en préciser le contexte. Nous utilisons:

-- une spécification d'énoncé explicite structurée --algorithmes-- décrite avec le principe de la méthode déductive définie par C.PAIR (BEL78),

-- le langage PL/1: une technique de programmation modulaire par procédures multi-points d'entrée est suggérée. Pour chaque procédure, nous utilisons des normes d'interface avec le système: paramètres de procédure, points d'entrée externes à la procédure, variables globales 'statiques' et 'dynamiques' (GAL76),

-- les outils logiciels: macroprocesseur PL/1, compilation séparée, processeurs de gestion de fichiers, procédures systèmes, éditeur de textes,

-- l'environnement et le mode d'exploitation: lot ('batch') ou temps partagé sur système SIRIS 7/8 de l'IRIS 80 avec Pseudo Dispack Privé (P.D.P.).

Nous n'allons pas bien évidemment décrire complètement cette réalisation; seuls certains aspects qui nous semblent importants pour la suite sont abordés.

II.3.1.1. Définition des algorithmes du problème.

Bien que nous ne l'ayons pas mentionné, dans l'étape d'explicitation, il faut introduire les données techniques du problème (ABR77).

Nous les avons définies en fonction de trois événements principaux liés à l'utilisation:

-- l'enregistrement d'un nouveau prêt,

-- le versement d'un client,

-- le traitement mensuel,

et de deux événements secondaires de "routine":

-- l'arrivée d'un nouveau client,

-- la réactivation d'un prêt.

Données techniques :

--Le traitement à la "demande" d'un nouveau prêt: si le client n'existe pas encore, il faut le créer dans le fichier à accès direct CLIENTS qui comprend pour chaque client: le numéro client, le nom, le prénom, l'adresse , la situation familiale et le nombre de prêts. Ensuite, il faut créer un nouveau prêt pour le client dans le fichier à accès direct PRETS constitué pour chaque prêt: du numéro de prêt, de la date de début du prêt, de la date de fin du prêt, du numéro client, du débit, du crédit, du montant du prêt, de l'état du prêt (actif ou non actif), du montant de la mensualité.

Dans le cas où un client a pris du retard pour le remboursement i.e. que le client existe mais son prêt est inactif, il faut "réactiver" le prêt du client à partir du fichier PRETS.

--Le traitement à la "demande" d'un versement s'effectue à partir de deux fichiers à accès direct:

. le fichier PRETS précédemment donné,

. le fichier des versements VERS qui comprend: le numéro de prêt, la date de versement et le montant du versement.

--Enfin, le traitement mensuel qui consiste à mettre à jour le fichier

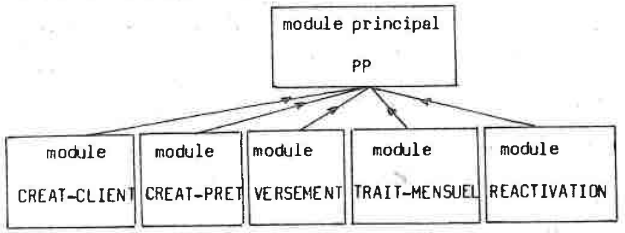
Analyse du module principal d'interface utilisateur

PP:module d'interface utilisateur	PP	
	Init; répéter (Arrêt,Traitement)	Init:lecture initiale d'un événement initialisation événement Arrêt:si dernier événement ZZ=1 Traitement:traite les événements principaux et secondaires
	Init	
ZZ:booléen d'existence d'événement initialisé à 0 EVEN:nature de l'événement	lecture d'un événement(EVEN) si dernier événement alors ZZ=1	
	Arrêt	
	si ZZ=1 alors arrêt-du-programme	
	Traitement	
	si EVEN='création-client' alors CREAT-CLIENT sinonsi EVEN='création-prêt' alors CREAT-PRET sinonsi EVEN='versement' alors VERSEMENT sinonsi EVEN='traitement-mensuel' alors TRAIT-MENSUEL sinonsi EVEN='réactivation' alors REACTIVATION	CREAT-CLIENT:création d'un client dans fichier CLIENTS CREAT-PRET:création d'un prêt dans fichier PRETS VERSEMENT:création d'un versement client dans fichier VERS TRAIT-MENSUEL:mise à jour du fichier PRETS REACTIVATION:activation du prêt d'un client dans fichier PRET pour lequel le remboursement a pris un retard supérieur à 2 mensualités

PRETS à partir des fichiers CLIENTS et VERS.

Nous ne décrivons que l'"arbre d'appel" des modules et l'analyse du module principal PP de l'application.

Arbre d'appel des modules .



II.3.1.2. Programmation du système d'application .

Chaque module est ensuite traduit en une procédure PL/1, mise au point et testée séparément.

Les textes de déclarations de fichiers et de variables d'enregistrement associées dans les modules sont catalogués avec un nom défini par l'utilisateur dans une bibliothèque du PDP. Les noms des textes des déclarations correspondent à des clés de fichiers partitionnés de cette bibliothèque.

Exemple : **texte du fichier PRETS catalogué sous le nom DPRE

```
DCL PRETS FILE RECORD KEYED ENV(ORG(INDEXED),AIAM);
/*fichier à accès direct, en organisation indexée avec méthode
d'accès indexée assistée*/
DCL 1 PRET,
      2 NPRET CHAR(4),
      2 DDP,
```

```
            3 JD DEC FIXED(2),
            3 MD DEC FIXED(2),
            3 AD DEC FIXED(2),
          2 DFP,
            3 JF DEC FIXED(2),
            3 MF DEC FIXED(2),
            3 AF DEC FIXED(2),
          2 NCL CHAR(4),
          2 DEB DEC FIXED(10,2),
          2 CRE DEC FIXED(10,2),
          2 MONTANT DEC FIXED(1),
          2 ACTIVITE DEC FIXED(1),
          2 MENS DEC FIXED(7,2);
    /*variable prêt*/
```

**texte du fichier CLIENTS catalogué sous le nom DCLT

```
    DCL CLIENTS FILE RECORD KEYED DIRECT ENV(ORG(INDEXED),AIAM);
    /*fichier à accès direct, organisation indexée avec méthode d'accès
indexée assistée*/
    DCL 1 CLIENT,
          2 NCLI CHAR(4),
          2 NOMCLI CHAR(15),
          2 PRENCLI CHAR(15),
          2 ADCLI CHAR(15),
          2 SITF CHAR(30),
          2 NBPRT DEC FIXED(2);
     /*variable client*/
```


**texte du fichier versement VERS catalogué sous le nom DVERS

```
    DCL VERS FILE RECORD KEYED ENV(ORG(INDEXED),AIAM);
    /*fichier à accès direct, organisation indexée, méthode d'accès
indexée assistée*/
    DCL 1 VERSEMENT,
          2 NPRET CHAR(5),
          2 DATVER CHAR(4),
          2 MONTANT DEC FIXED(8,2);
    /*variable versement*/
```

Dans chaque procédure, nous remplaçons ensuite ces déclarations par des macro-instructions %INCLUDE <nom-partition> et nous complétons les attributs des autres variables regroupées par nature: paramètres, globales statiques, points d'entrée de procédures externes. Après avoir effectué cette "normalisation", nous cataloguons chacune de ces procédures compilées séparément, en bibiothèque.

Les textes sources des procédures systèmes cartes de commande et des procédures PL/1 sont sauvegardés dans des fichiers de l'utilisateur à l'aide de l'éditeur de textes.

Selon le type d'utilisation de l'application intensif ou non, il faut ensuite:

.soit réaliser un "module de chargement" et le cataloguer sur le PDP de l'utilisateur avec un nom 'gestion-prêt',

.soit rédiger une procédure système de cartes de commande avec une étape d'édition de liens des procédures PL/1.

La dernière étape consiste à tester l'intégration du système de programmes d'application.

II.3.1.3. Remarques .

1-- sur la réalisation : Il est effectivement possible de réaliser un système de programmes d'application structuré à condition que tous les membres de l'équipe qui participent au développement emploient la même discipline de développement (MIL71). Ajoutons que les programmeurs doivent "bien connaître" le langage de commande du système d'exploitation. Enfin, il ne faut pas nier que la mise en oeuvre de PL/1 n'est pas toujours une chose aisée pour qui n'est pas un spécialiste du langage.

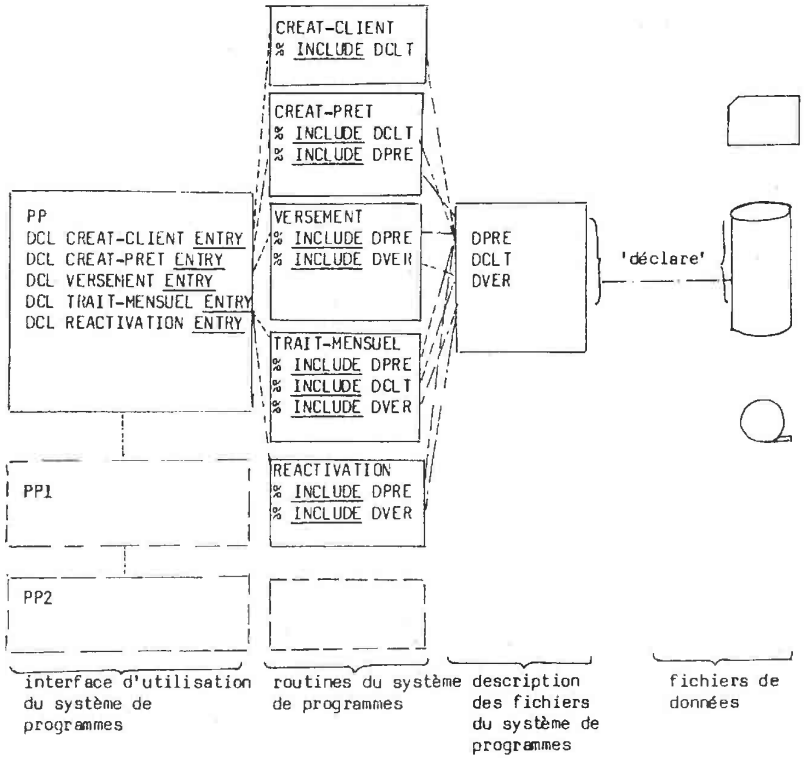
2-- sur la structuration du système de programmes :

Nous allons, à partir de ce schéma, préciser les mécanismes fondamentaux qui permettent de construire des systèmes de programmes d'application structurés avec des langages de programmation généraux du genre PL/1. Il est tout d'abord clair que la technique de programmation suggérée utilise une forme de "décomposition modulaire" (PAR71) et de "raffinement" (WIR71) des traitements et des données. Deux mécanismes sont utilisés : les procédures et la manipulation de textes avec la macro-instruction %INCLUDE du macro-processeur PL/1.

Pour rédiger la procédure principale PP, que l'on peut regarder comme le module d'interface pour l'utilisation du système de programmes, nous n'avons besoin que de déclarer les points d'entrée des routines qui vont être appelées dans le corps de PP. Ces déclarations de routines de type fonction ou procédure définissent les "profils syntaxiques" des opérations externes. Nous avons représenté sur le schéma les relations "utilise routine" correspondant à chacune de ces déclarations du programme principal PP. Ensuite, pour chacune de ces

procédures systèmes de cartes de commande
."lancement" système gestion-prêt .compilation et mise en bibliothèque des procédures PL/1 .mise en bibliothèque des textes sources des déclarations .autres procédures systèmes de service (éventuellement paramétrées): manipulation des fichiers, etc...
fichiers des textes sources des déclarations
.DPRE .DCLT .DVER
fichiers des textes sources des procédures PL/1
.PP .CREAT-CLIENT .CREAT-PRET .VERSEMENT .TRAIT-MENSUEL .REACTIVATION
fichier des textes sources des procédures systèmes cataloguées
...
fichier du module de chargement (éventuellement)
...
fichier de données de l'application
.DATA-DPRE .DATA-DCLT .DATA-DVER

Informations cataloguées sur le PDP de l'utilisateur.



Structuration du système de programmes.

routines, il est possible de rédiger le corps avec le même mécanisme. Ce procédé qui consiste à décomposer un programme en morceaux est le principe de décomposition modulaire des programmes. De plus, la décomposition est appliquée avec description progressive des détails dans chaque procédure. On parle alors du principe de décomposition par raffinements successifs. Dans l'approche gestion des prêts, les routines utilisent des macro-déclarations %INCLUDE DPRE, %INCLUDE DCLT, %INCLUDE DVER. Nous avons représenté sur le schéma les relations "utilise déclarations fichier" correspondant à chaque utilisation de ces textes dans les routines. Ce procédé permet le partage de textes particuliers par les routines de l'application: les déclarations de variables globales (les fichiers permanents de l'application). Ce procédé ne présente qu'un intérêt méthodologique car l'utilisation du macro-processeur ne permet qu'un partage de textes par recopie.

En utilisant la possibilité d'imbriquer des %INCLUDE, il est effectivement possible d'entreprendre une construction de programmes par décomposition et raffinements successifs. Toutefois, la seule abstraction possible est constituée du nom du texte. Après macro-assemblage, on obtient un programme monolithique. Nous ne détaillerons pas plus ce dernier aspect dont on connaît bien tous les inconvénients qu'il engendre (DER74).

En fait, si le mécanisme de procédure de PL/1 permet d'effectuer une construction progressive et structurée des traitements, il n'existe pas d'autre mécanisme que la macro-déclaration pour les données.

3-- sur l'évolution du système de programmes.

Nous distinguons deux possibilités d'évolution:

-- il est effectivement possible, si l'on ne remet pas en cause la conception des routines et des fichiers du système de programmes de l'application, de réaliser rapidement des interfaces d'utilisation pour répondre à de nouveaux besoins. Cette évolution peut nécessiter la construction de nouvelles routines et de nouveaux fichiers; cela ne pose pas trop de problèmes,

-- lorsqu'une routine ou une structure de fichier doit être modifiée: il est nécessaire de recompiler toutes les routines et les procédures de l'interface sensibles à cette modification et éventuellement de reconstituer un nouveau module de chargement.

II.3.2. Approche système de base de données d'application.

La réalisation de ce système est décrite dans le contexte suivant:

* la spécification explicite de l'univers du problème utilise un modèle de données général qui prend en compte les fonctions d'accès logiques,

* le système de gestion de base de données (S.G.B.D.) utilisé est SOCRATE-CII: langage de définition de données (L.D.D.), langage de manipulation de données (L.M.D.), macro-langage SOCRATE,

* les outils logiciels nécessaires: processeurs de gestion de fichiers, procédures systèmes, éditeurs de textes et assembleur,

* l'environnement et le mode d'exploitation: lot, conversationnel ou simulation batch du conversationnel en temps partagé sur système SIRIS 7/8 de l'IRIS 80 avec Pseudo Dispack Privé (P.D.P.).

II.3.2.1. Programmation du système de base de données.

Pour expliciter l'"univers" du problème dans l'approche base de données, plusieurs modèles d'analyse sémantique des données, plus ou moins formalisés, peuvent être utilisés (CHE76) (SMI77) (BRO78) (PIR76) (FLO77) (BEN79). Tous ces modèles dérivent plus ou moins du modèle relationnel de CODD (COD70). Les difficultés rencontrées dans ce processus d'explicitation de "schéma sémantique" relèvent de la complexité de l'"interdépendance" des propriétés des données et des relations. Si la technique de "raffinement" par transformations successives de schémas s'applique, il est difficile d'envisager l'utilisation de la décomposition au sens où on l'entend en méthodologie de la programmation.

Pour aboutir au schéma "logique" de données décrit ci-dessous, nous avons un peu modifié les données techniques de l'approche précédente (II.3.1.1.). Nous utilisons maintenant deux fichiers: un fichier CLT des clients et un fichier PRVT qui regroupe toutes les données des prêts, des prêts actifs et des versements effectués par les clients. D'autre part, il résulte de l'analyse que le numéro du client NCLI est une clé de l'entité CLIENT et que le numéro de prêt NPRET est une clé de l'entité PRET et une clé de l'entité ACTIF.

Enfin, avec les graphes et les schémas "sémantiques", on ne peut pas décrire toutes les propriétés des données (TSO79).

Ce schéma logique, conçu indépendamment d'un S.G.B.D., doit ensuite être décrit avec le langage de définition des données (L.D.D.) choisi, SOCRATE en l'occurrence. Cette "transformation" de schéma peut nécessiter quelques adaptations. Par exemple, en SOCRATE, il n'est pas possible de représenter directement des relations (n,m) avec n,m $\geq$ 1 entre deux objets; on doit introduire des intermédiaires artificiels. La transformation de schémas "sémantiques" pose le problème de la représentation d'un modèle sémantique de données dans un autre.

Nous étudions cette approche avec l'exemple gestion des prêts dont le schéma logique SOCRATE est:

```
$SOUR
   **fichier source du schéma SOUR
   **droits et mots de passe
   (1 2 3 4) CHAB (CHABRIER(TYP))
   DEBUT
   $CLT(32)
      **fichier de données CLT
      ENTITE(30) CLIENT
      DEBUT
         EMPRUNT ANNEAU
         NCLI DE 1000 A 9999 AVEC CLE(10000$HASHNUM)FIN
         NOMCLI MOT(15)
         PRENCLI MOT(15)
         ADCLI MOT(15)
         SITF MOT(30)
         NBPRT DE 0 A 10
      FIN
```



```
$PRVT
   **fichier de données PRVT
   ENTITE(200) PRET
   DEBUT
      EMPRUNTEUR REFERE EMPRUNT DE UN CLIENT
      NPRET DE 1000 A 9999 AVEC CLE(10000$HASHNUM)FIN
      DDP
      DEBUT
         JD DE 1 A 31
         MD DE 1 A 12
         AD DE 70 A 99
      FIN
      DFP
      DEBUT
         JF DE 1 A 31
         MF DE 1 A 12
         AF DE 70 A 99
      FIN
      DUREE
      DEB
      CRE
      MONTANT
      MENS
      ENTITE(60) VERSEMENT
      DEBUT
         DATE
         DEBUT
```

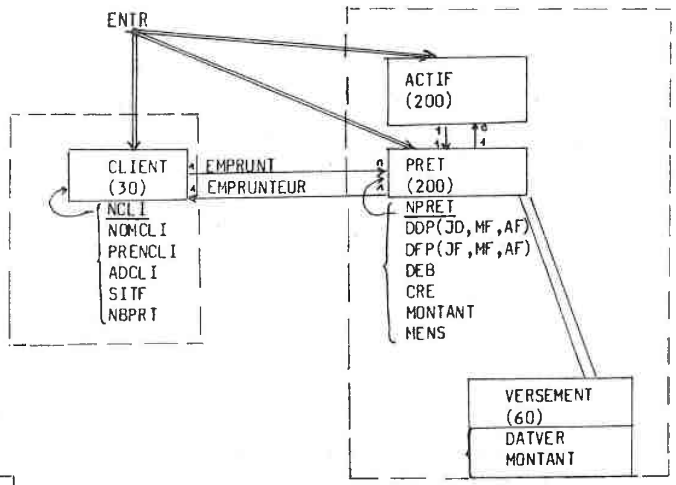
```
            JOUR
            MOIS
            AN
          FIN
          MONTANT
        FIN
      FIN
      ACTIF INVERSE TOUT PRET
  FIN
FIN
```

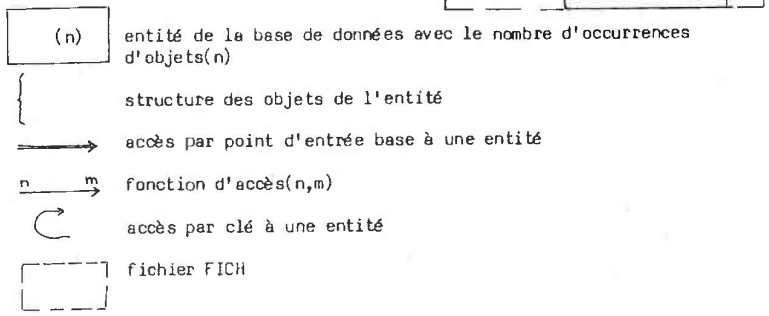
Caractéristiques de cette représentation SOCRATE:

1) <u>représentation des fonctions d'accès</u> :

Les "entrées base" correspondent aux entités de premier niveau de la description CLIENT, PRET, ACTIF. La fonction d'accès EMPRUNT: CLIENT-->PRET(1,n) est ici représentée par la caractéristique EMPRUNT <u>ANNEAU</u> attachée à l'entité EMPRUNT. La fonction d'accès EMPRUNTEUR: PRET-->CLIENT(1,1) est définie par la caractéristique EMPRUNTEUR REFERE EMPRUNT DE UN CLIENT attachée à l'entité CLIENT. La relation PRET-VERSEMENT est représentée par le rattachement hiérarchique de l'entité VERSEMENT à l'entité PRET. Enfin, la relation ACTIF-PRET a conduit à la définition du "fichier inverse" ACTIF INVERSE TOUT PRET.

La définition de caractéristiques clés NCLI de l'entité CLIENT et NPRET de l'entité PRET nécessite la déclaration des programmes d'accès systèmes ou utilisateur. Ici %HASHNUM est un programme de hashcode du système qui utilise un dictionnaire avec 10000 entrées.





<u>Schéma logique de la base de données</u>.

2) représentation du fichier des données : (défini dans la description ci-dessus)

Si cette description de schéma utilise des programmes de hashcode particuliers, il est nécessaire de les écrire en assembleur et de les cataloguer dans le système SOCRATE avant même d'effectuer la compilation du texte. Cette description est ensuite compilée et cataloguée par un processeur du système SOCRATE. Le texte source doit être sauvegardé --selon les versions-- dans des fichiers utilisateurs avec l'éditeur de textes du système SIRIS7/8.

Il faut ensuite traduire les demandes de programmes dans le langage de manipulation de données SOCRATE: création, suppression, mise à jour, interrogation. Ces programmes peuvent alors être mis au point et gérés à l'aide de l'éditeur de textes sous contrôle du système SOCRATE.

II.3.2.2. Remarques .

1-- sur la réalisation .

Il est certain que la plupart des difficultés rencontrées proviennent d'une mauvaise maîtrise de la complexité du problème. Ceci est essentiellement dû à l'approche de conception globale et à l'insuffisance des modèles sémantiques de données actuels. Lorsque le schéma SOCRATE est décrit, on peut alors rédiger les routines de l'application avec le langage de manipulation. Il est aussi possible d'écrire des programmes paramétrés à l'aide du macro-processeur de SOCRATE.

Exemple : Nous définissons l'algorithme de la routine CREAT-PRET

correspondant au macro-programme SOCRATE qui suit:

```
CREAT-PRET(numéro-du-prêt)
si existe un prêt ayant NPRET=numéro-du-prêt alors
   imprimer 'prêt déjà alloué' et sortir de la macro
sinon
   imprimer 'numéro du client' et lire le numéro-du-client
   si le numéro-du-client lu n'existe pas alors
     imprimer 'client inexistant'
     demander un nouveau numéro-du-client
   fsi
fsi
générer une réalisation d'un prêt
affecter le client emprunteur
debit=credit=0
lire les dates de début et de fin du prêt
calculer la durée du prêt et vérifier qu'elle satisfait la
contrainte d'intégrité
calculer la mensualité
incrémenter le nombre de prêts du client
FIN CREAT-PRET
```

Toutes les routines de l'application peuvent être écrites comme des macro-instructions ou encore comme de simples programmes SOCRATE ensuite compilés et catalogués sous contrôle du système.

Exemple :

```
LISTE-DES-PRETS
:EXP
   POUR TOUT CLIENT LISTE FIN
:FDEF
```

Ici, LISTE est encore une routine qui peut être écrite et compilée "séparément":

```
:DEFMAC CREAT-PRET:;
   :EXP
   FAIRE
      SI EXISTE UN PRET AYANT NPRET=:1:
         ALORS
         I '***** PRET DEJA ALLOUE ****'
         SORTIE
         SINON FAIRE
         I '**** NUMERO DU CLIENT ***' M Y1=EXT
         SI EXISTE UN CLIENT X1 AYANT NCLI = Y1
           ALORS SORTIE
           SINON I '*** CLIENT INEXISTANT***' REFAIRE
         FIN
      FIN
      G UN PRET X2
      POUR X2 M EMPRUNTEUR = X1
         M NPRET =:1:
         M CRE = 0
         M DEB = 0
         M MONTANT
         I 'DATE DE DEBUT'
         FAIRE
           LIRE M(0) JD DE DDP M(3) MD DE DDP
```

```
      M(6) AD DE DDP
      FIN
        I 'DATE DE FIN'
        M Y3=MD DE DDP DE X2
      M Y4=AD DE DDP DE X2
      LIRE M(0) JF DE DFP M(3) MF DE DFP
      M(6) AF DE DFP
      FIN
      M Y6=MF DE DFP DE X2
      M Y5=AF DE DFP DE X2
      M Y4= 12 * Y5 - 12 * Y4 + Y6 - Y3
      SI Y4>490
        ALORS I '*****ERREUR DUREE***' REFAIRE
        SINON
        SI Y4<1
          ALORS I '****ERREUR 2 DUREE***' REFAIRE
          SINON
          M DUREE=Y4
        FIN
      FIN
    FIN
      M Y2=MONTANT DE X2
    M Y2=Y2 / Y4
    M MENS = Y2
FIN
M Y7=NBPRT DE X1
M Y7 = Y7 + 1
```



```
    M NBPRT DE X1 = Y7
  FIN
FIN
:FDEF
```

2-- sur la structuration du système de base de données :

Dans l'approche base de données, l'effort de construction le plus important porte sur la définition du schéma, conditionnant ainsi le développement des applications spécifiques. En effet, tant que ce schéma n'est pas complètement défini, il est impossible de commencer la construction des programmes d'utilisation. La "modularisation" de cet univers n'est entreprise qu'à *posteriori* : définition de sous-schémas pour des sous-systèmes de bases de données.

Le développement des sous-systèmes de programmes d'application n'est entrepris qu'après la définition des sous-schémas. La définition d'un sous-schéma ne pose pas de problème particulier. Les routines des applications peuvent être traitées de manière structurée, si l'on fait un usage intensif du macro-générateur de SOCRATE. Notons toutefois que les programmes obtenus ne sont pas très efficaces, peu compréhensibles lorsqu'il y a plusieurs paramètres. En outre, le langage de manipulation ne permet de faire que des "calculs élémentaires".

En tant que langage 'autonome' (self-contained), le langage de manipulation SOCRATE ne supporte pas de mécanisme favorisant la structuration des programmes et donc l'écriture de gros programmes. Précisons que cette dernière remarque s'applique à tous les SGBD actuels.

L'utilisation du langage de manipulation de données offre par contre un certain confort de programmation. En effet, tous les détails liés

aux accès sur mémoire secondaire sont cachés --ouverture, fermeture de fichiers, lecture/écriture--, différenciés selon les organisations et les méthodes d'accès utilisées.

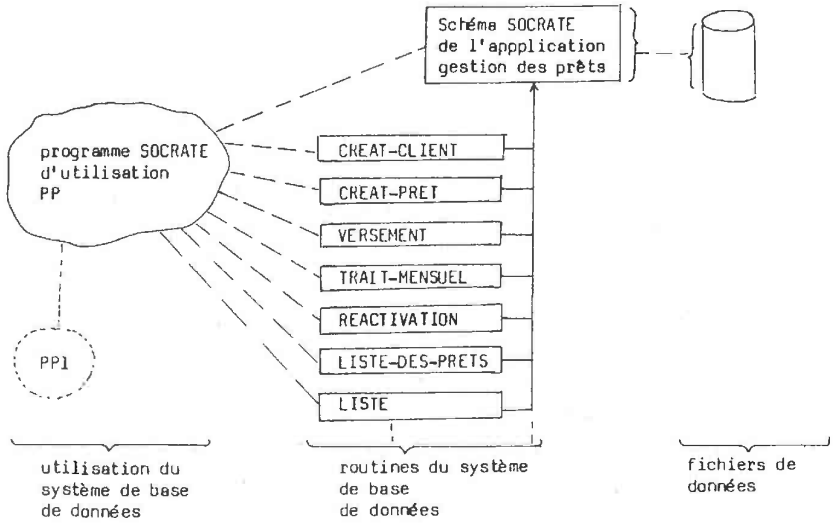
3-- sur l'évolution du système de base de données :

Tous les programmes de routines étant rédigés sur le schéma de la base, il est clair qu'une modification de la structure du schéma va entraîner d'abord la recompilation du schéma puis une réorganisation des données sur mémoire secondaire et la modification des routines des applications.

Le dernier point important concerne l'"indépendance physique" (DAT77). La description (II.3.2.1.) montre qu'en SOCRATE, mais c'est aussi vrai pour beaucoup d'autres SGBD, le schéma dépend de l'organisation physique: déclaration des fichiers, déclaration de méthodes d'accès pour les clés. Seules les organisations et les méthodes d'accès aux fichiers sont cachées.

II.3.3. Approche système logiciel "mixte".

La plupart des SGBD commercialisés offrent des possibilités d'interface en langage hôte: assembleur, Fortran, COBOL, PL/1, etc... Dans cette approche, le développement de systèmes logiciels d'applications supporte à la fois la construction de systèmes de programmes et la construction de systèmes de bases de données. Selon la nature des demandes des utilisateurs, le caractère d'évolution, le mode d'exploitation souhaité, etc... les choix de réalisation du système logiciel seront orientés langage de programmation ou SGBD. Par exemple,



Structuration du système de base de données.

la création, la mise à jour ou même l'édition de volumes importants de données peuvent justifier des traitements 'batch' sous contrôle de l'interface hôte. Par contre, les formulations "ponctuelles" de demandes d'informations rédigées dans le langage de manipulation du SGBD sont plus adaptées à un mode d'exploitation conversationnel. Les programmes écrits en langage autonome peuvent utiliser des routines plus ou moins complexes, paramétrées et préalablement cataloguées dans le système. Enfin, on dispose d'un langage de programmation général pour écrire des programmes scientifiques ou de manipulation de fichiers.

II.3.3.1. Remarques .

1-- sur la réalisation :

Dans l'état actuel des techniques de spécification de problèmes: spécification d'énoncés et d'univers (BRO78) (WEB78), il faut bien avouer que le développement des systèmes logiciels d'applications --programmes et bases de données-- relève essentiellement d'un certain "savoir-faire".

Les différences importantes de concepts manipulés dans les langages de programmation généraux et dans les SGBD, les techniques de construction de systèmes de programmes et de systèmes de bases de données constituent des handicaps énormes pour prétendre aboutir à une construction cohérente de logiciel.

D'autre part, l'hétérogénéité des logiciels de base complique encore la tâche des programmeurs.

Dans ces conditions, il est évidemment difficile de parler de construction de logiciel de qualité.

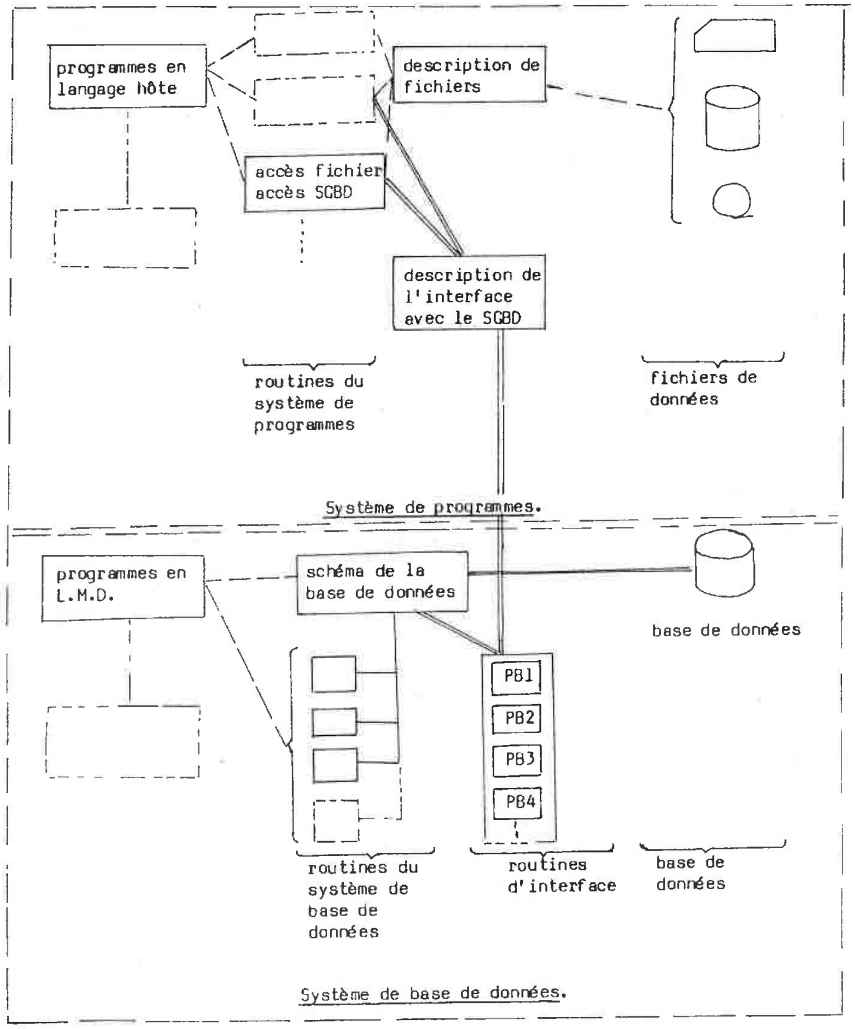
2-- sur la structuration du système mixte :

Nous considérons ici l'interface COBOL du SGBD SOCRATE.

-- Dans le système de programmes , nous avons dû introduire les accès au SGBD ainsi qu'une description de l'interface avec le SGBD. Cette description est définie en zone de travail des programmes COBOL (working-storage section). Elle est commune à toutes les routines qui effectuent des accès à la base de données; c'est pourquoi nous la traitons comme la description des fichiers --instruction COPY de COBOL. Il est aussi nécessaire de déclarer des zones d'échange d'information entre le programme COBOL et le SGBD, ainsi que tous les renseignements nécessaires à l'activation du système SOCRATE: nom du schéma, numéro comptable, numéro de compte, mot de passe, mode d'exploitation, nom des sous-programmes batch du SGBD qui "récupèrent" les informations dans la zone d'échange, des informations système, etc...

Les accès au SGBD SOCRATE effectués dans les routines du système de programmes se font comme avec un système de gestion de fichiers: ouverture, fermeture, appels en mode lecture, écriture ou mise à jour.

-- Dans le système de base de données , nous avons dû définir toutes les routines batch d'interface zone d'échange COBOL-base de données en langage de manipulation de données SOCRATE. Ces routines doivent être compilées avant l'utilisation d'un interface COBOL.



Structuration d'un système mixte.

III. PREMIERES CONCLUSIONS.

L'étude détaillée du développement des systèmes logiciels d'applications que nous venons de présenter montre l'ampleur des problèmes pédagogiques que l'on rencontre dans l'enseignement de ce qui est parfois appelé les "systèmes d'informations". Il est évident que notre participation à différentes activités pédagogiques: enseignement de la programmation par la méthode déductive (BEL78), des systèmes de gestion de bases de données à la Maîtrise d'Informatique Appliquée à la Gestion, en cycle ingénieur CNAM et en DESS Informatique, de recherches orientées système avec J.C.DERNIAME (DER74) a fourni de nombreux éléments de réflexion aussi bien sur les systèmes de programmes, sur les systèmes de bases de données que sur les systèmes de programmation modulaire. Pour parvenir à réduire les difficultés importantes rencontrées lors du développement des systèmes logiciels d'applications que nous venons d'étudier; à savoir:

-- le manque d'orthogonalité des concepts et des techniques utilisés pour concevoir et réaliser des systèmes de programmes et des systèmes de bases de données,

-- l'hétérogénéité des outils logiciels d'aide à la construction de systèmes viables,

il nous a semblé dès 1976 (CHA76) qu'il était nécessaire d'entreprendre de nouvelles investigations sur les propositions et réalisations récentes dans les SGBD, les langages de programmation et les techniques de spécification.

Il nous a donc paru intéressant de présenter une synthèse des idées des différents domaines que nous avons abordés et qui ont plus ou moins

influencé notre travail: approche système de gestion de base de données, approche langages généraux et orientés système, approche langage orienté système de base de données.

CHAPITRE II. APPROCHE SYSTEMES DE GESTION DE BASE DE DONNEES.

I. NOTIONS DE BASE.

II. APPROCHE HIERARCHIQUE.

II.1. Rappels sur les structures d'arbre et de forêt.

II.2. Modèle hiérarchique de données.

II.3. Etude du SGBD IMS.

II.3.1. Structuration d'un système logiciel d'application IMS.

II.3.1.1. Description d'une base de données IMS.

II.3.1.2. Création et manipulation d'une base de données IMS.

II.3.1.3. Evolution.

II.4. Remarques et conclusions.

III. APPROCHE RESEAU.

IV. APPROCHE RELATIONNELLE.

IV.1. Rappels sur le modèle relationnel.

IV.1.1. Dépendance fonctionnelle.

IV.1.2. Clés candidates et clés primaires.

IV.1.3. Algèbre relationnelle.

IV.1.4. Calcul relationnel.

IV.2. Remarques générales.

IV.3. Les SGBD relationnels.

IV.3.1. Langages d'interface utilisateurs.

IV.3.1.1. Le langage SEQUEL.

IV.3.1.2. Le langage QUEL.

IV.3.1.3. QUERY-BY-EXAMPLE.

IV.3.2. Insertion des langages d'interface dans des langages hôtes.

IV.3.2.1. Langage EQUEL du système INGRES.

IV.3.2.2. R.D.I.Langage d'insertion de SEQUEL de SYSTEM-R.

IV.3.3. Implantation des SGBD relationnels.

IV.3.4. Autres propositions.

IV.3.4.1. Le système PIVOINE.

IV.3.4.2. Le système SYNTEX.

IV.3.4.3. La programmation logique.

V. REMARQUES ET CONCLUSIONS.

Dans ce chapitre, nous commençons par définir les principales notions de base du domaine des bases de données. Ensuite, nous étudions quelques traits de conception de plusieurs S.G.B.D. hiérarchiques, réseaux et relationnels.

I. NOTIONS DE BASE.

Pour pouvoir parler de bases de données, il semble important de commencer par définir ce qu'est une donnée, un modèle de données, un modèle de base de données, etc...; ce qui n'est d'ailleurs pas facile.

Le concept de donnée défini par NEUHOLD (NEU74) est: "si une personne arrange de façon intentionnelle un "réel perçu" pour en représenter un autre, alors le premier arrangement est appelé donnée . Le "réel perçu" arrangé est le support utilisé pour mémoriser la donnée". Les langues parlées ou écrites sont des exemples d'arrangements de "réel perçu".

Le concept d'information proposé par NEUHOLD (NEU74) est: "une information est définie comme une nouvelle connaissance". Cette notion fait référence à des définitions explicites de concepts, de règles mais aussi implicites. Ainsi, une information est toujours définie en fonction d'anciennes connaissances. Il est alors évident que l'"interprétation" d'une information par plusieurs individus pose des problèmes de communication d'information: notamment lorsque les références aux anciennes connaissances ne sont pas les mêmes pour tous.

Ce genre de problème a d'ailleurs fait l'objet d'un très large débat

au Congrès IFIP de Freudenstadt (IFIP76).

Une base de données (DAT77) est "une collection de données opérationnelles mémorisées, utilisée par des systèmes d'applications de certaines entreprises ".

Un système de gestion de base de données (S.G.B.D.) (PIR76) est "un ensemble de logiciels qui fournit divers moyens d'accès à une base de données: création, interrogation et modification. Il assure aussi des fonctions particulières permettant: l'intégration et le partage de données ainsi que le maintient de la consistance, de l'intégrité et de la sécurité des données".

Un schéma de base de données (PIR76) est "la définition d'une "structure" de base de données particulière; elle est indépendante des données qui sont mémorisées à un moment donné".

Un sous-schéma est une description 'logique' d'une partie de schéma de base de données.

Dans la littérature sur les bases de données, un modèle de données définit à la fois le schéma de base de données et l'interprétation du schéma fonctionnel (ou relationnel) au sens de la logique formelle (KRE67). C'est un exemple parmi d'autres de terminologie ambigue.

Une base de données est avant tout construite pour être partagée par plusieurs systèmes d'applications utilisateurs. Pour que les bases de données soient viables, il importe que le système de gestion de base de données possède une qualité fondamentale appelée l'indépendance des données . Cette notion pourtant essentielle n'est pas très clairement définie. Citons quelques exemples de définitions de la littérature. J.FRY et E.H.SIBLEY (COM76) distinguent l'indépendance physique des données et l'indépendance logique des données:

.un système satisfait à la propriété d' indépendance physique des données si les programmes d'application ou toute requête "ad hoc" sont "relativement" indépendants de la structure de données physique et des méthodes d'accès,

.l' indépendance logique des données est la possibilité d'effectuer des changements logiques -- au sens structure de données logique des langages de programmation -- sans affecter de façon significative les programmes qui l'utilisent.

Ces définitions ne font qu'adapter ce que D'IMPERIO (IMP69) a appelé dans le domaine de la programmation "data structure independance" et "storage structure independance".

Maintenant, si l'on se réfère à l'ANSI/X3/SPARC (ANS75): "l'indépendance des données est la possibilité d'un système de gestion de base de données de permettre d'une part des références à des données mémorisées, spécialement dans des programmes, d'autre part à leurs descriptions d'être insensibles au changement et aux différents usages dans l'environnement de données: données mémorisées, partagées par d'autres programmes et réorganisées pour améliorer les performances du système".

Il nous semble que la définition de C.J.DATE (DAT77) résume assez bien cette notion fondamentale: "l'indépendance des données correspond à l'insensibilité des programmes d'application au changement de structure de données mémorisées et de stratégies d'accès aux données". Nous allons maintenant préciser trois notions qui, dans la littérature, nous paraissent confuses; il s'agit de la protection, de la sécurité et de l'intégrité des bases de données. En effet, le groupe ANSI (ANS75) distingue:

-- la sécurité ('privacy') qui concerne les réglements, la législation en cas de fraude, etc...

-- la confidentialité: les règles d'accès aux données,

-- la sécurité ('security'): les moyens et les mécanismes à mettre en oeuvre pour que les réglements de sécurité ('privacy') soient effectivement respectés.

En fait, la sécurité des S.G.B.D. diffère de la protection des systèmes d'exploitation sur deux points (TSI73), à savoir:

-- qu'il faut considérer tout l'environnement d'utilisation du S.G.B.D., et même les utilisateurs qui en principe ne concernent pas directement la protection d'un système d'exploitation,

-- la nature et le contenu des données ont un rôle important dans le mécanisme de droits d'accès de la sécurité.

C.J.DATE (DAT77) définit la sécurité comme la protection des données d'une base de données contre les manipulations interdites : impression, modification ou destruction.

L' intégrité concerne la protection des données contre la destruction ou la modification accidentelle par les utilisateurs d'une base de données.

Ces rappels étant faits, nous nous intéressons maintenant aux trois grandes catégories de S.G.B.D.: hiérarchiques, réseaux et relationnels.

II. APPROCHE HIERARCHIQUE.

Pour présenter le modèle hiérarchique de base de données, nous commençons par rappeler brièvement les définitions d'arbre et de forêt. Nous étudions sur l'exemple de SGBD le plus répandu: IMS (Information Management System), les caractéristiques de cette approche. Nous donnons aussi quelques précisions sur les possibilités du langage de manipulation du SGBD SYSTEM 2000.

II.1. Rappels sur les structures d'arbre et de forêt.

Nous nous contentons ici de donner les définitions de E.D.KNUTH Vol 1 (KNU68).

Un arbre est un ensemble fini T d'un ou plusieurs noeuds tel que:

a)il existe un noeud particulier appelé la racine,

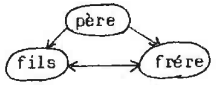
b)les autres noeuds (excepté la racine) sont partitionnés en ensembles disjoints T1,T2,...,Tm ($m \geq 0$), chacun de ces ensembles est à son tour un arbre. Les arbres T1,T2,...,Tm sont appelés les sous-arbres de la racine.

Remarque : L'énoncé récursif de cette description ne signifie évidemment pas que la structure d'arbre est récursive!

Un arbre est dit ordonné lorsque l'ordre relatif des sous-arbres est significatif. Lorsque deux arbres ne différent que par l'ordre des sous-arbres respectifs, on parle alors d'arbre orienté .

Une forêt est un ensemble ordonné de zéro, un ou plusieurs arbres disjoints. Pour "parcourir" un arbre, il convient de rappeler la

terminologie suivante: quand un noeud est situé immédiatement au-dessous d'un autre, on dit que c'est le fils du premier. Le premier est alors le père du second. Les fils d'un même père sont des frères .



fils

Cette relation de filiation crée des niveaux dans l'arbre. Par définition, la racine est au niveau 1. Si un noeud est au niveau j, ses fils sont au niveau j+1.

Une feuille est un noeud qui n'a pas de fils.

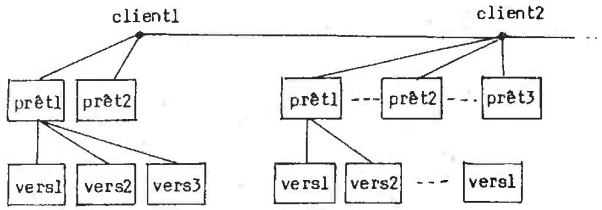
Le degré d'un noeud est le nombre de fils de ce noeud. Une feuille a un degré 0.

On dit que l'ordre de parcours d'un arbre est "longueur d'abord" s'il procède par niveaux et pour chaque niveau de gauche à droite.

II.2. Modèle hiérarchique de données.

Définition : Un modèle hiérarchique de données est une forêt d'arbres orientés.

Illustrons cette caractérisation avec l'exemple de la gestion des prêts bancaires. L'analyse 'sémantique' des données de ce problème peut par exemple conduire à représenter les objets et les relations entre objets sous la forme graphique suivante:



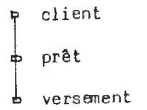
Ce graphe est une forêt d'arbres orientés clienti ($i \geq 1$).

Effectuons maintenant sur ce graphe quelques transformations à l'aide des abstractions suivantes:

-- pour chaque arbre client, négligeons l'ordre associé aux sous-arbres. Nous obtenons une forêt d'arbres clients orientés.

-- ensuite, pour chaque arbre et pour chaque niveau de sous-arbre, réduisons les ensembles d'objets de même "nature" à son objet générique. Par exemple, pour l'ensemble formé de client1, client2,... nous utilisons l'objet générique client.

Dans ces conditions, nous obtenons un arbre 'simplifié' que nous pouvons représenter de la façon suivante:



Cette représentation est souvent appelée enregistrement logique hiérarchique . Dans la terminologie des bases de données, on parle aussi de hiérarchie de segments. Un segment est le nom donné à l'objet "générique".

II.3. Etude du S.G.B.D.IMS (DAT77).

Dans cette étude, nous ne présentons que les principaux traits caractéristiques du système IMS: les langages, la structuration d'un système logiciel d'application -- et plus particulièrement du système de base de données -- , ainsi que les possibilités d'évolution.

II.3.1. Structuration d'un système logiciel d'application IMS.

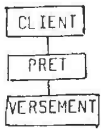
Dans l'architecture du S.G.B.D. IMS, la structuration d'un système de base de données est illustrée par la figure ci-après.

II.3.1.1. Description d'une base de données IMS .

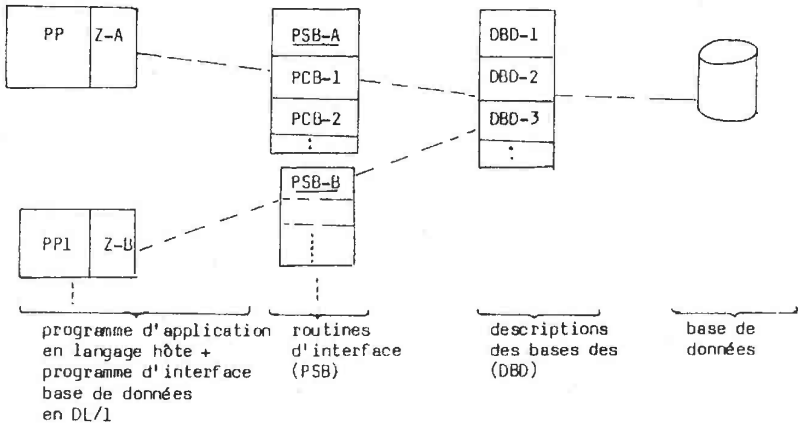
Dans les exemples que nous présentons, nous utilisons l'application des prêts bancaires du chapitre I (II.2.).

Pour le programmeur d'application, la "structure" de la base de données est perçue au travers de l'enregistrement logique constitué de segments hiérarchisés.

Exemple :



Chaque segment est décomposé en plusieurs zones de données dont une est une clé . Les seuls types possibles sont alphabétique et décimal



Structuration d'un SBD IMS.

(condensé et étendu).

Une description de base de données (DBD) définit les noms, les tailles, les dispositions des éléments de la base et l'organisation physique.

Exemple :

```
DBD NAME=GESTPRET,ACCESS=HISAM
SEGM NAME=CLIENT,BYTES=81,FREQ=30,PARENT=0
FIELD NAME=(NCLI,SEQ),BYTES=6,START=1,TYPE=C
FIELD NAME=NOMCLI,BYTES=15,START=1,TYPE=C
.
.
SEGM NAME=PRET,PARENT=CLIENT,BYTES=...
.
SEGM NAME=VERSEMENT,PARENT=PRET,BYTES=..
```

Remarques :

. La zone NCLI est définie comme zone clé à l'aide du paramètre SEQ

. le type des caractères est indiqué par le paramètre TYPE=C (caractère)

. le paramètre FREQ indique le nombre d'enregistrements estimés dans la base de données

. BYTES indique la longueur en octets

. le paramètre ACCESS = HISAM définit la méthode d'accès utilisée.

Maintenant, pour pouvoir accéder à la base de données à partir d'un programme d'application rédigé en PL/1, en COBOL, etc.., il faut définir une description logique de la base dans le PCB (Program Communication Block).

Le PSB (Program Specification Block) définit l'ensemble des PCB d'un

utilisateur. Les PCB ne sont pas partageables.

Exemple de PCB (simplifié) :

```
PCB TYPE=DB,DBNAME=GESTPRET,KEYLEN=12
SENSEG NAME=CLIENT,PROCOPT=GU
SENSEG NAME=PRET,PARENT=CLIENT,PROCOPT=G
SENSEG NAME=VERSEMENT,PARENT=PRET,PROCOPT=G
.
.
```

Remarques :

.SENSEG définit les segments sensibles au programme d'application,

.KEYLEN est une zone de travail pour le calcul des clés lors d'un accès à la base,

.PROCOPT (processing options) déclare les opérations permises GET (G), GET UNIQUE (GU).

La phase de création de la base de données ne peut démarrer qu'après génération (à l'aide d'une routine IMS) puis chargement, en librairie, des DBD et PSB.

II.3.1.2. Création et manipulation d'une base de données IMS .

Pour créer la base de données IMS, on procède de la façon suivante: un programme (PL/1) lit les segments sur supports externes (cartes, bandes, etc..), dans la séquence croissante des clés des racines et dans l'ordre hiérarchique à l'intérieur des enregistrements logiques; il faut ensuite les charger avec DL/1 (Data Language/1). Ceci n'est possible que si les segments du PSB sont sensibles et si l'option de

PROCOPT est 'load'.

Pour manipuler la base de données à partir d'un programme d'application PL/1, il faut écrire une procédure DLIPLI. Cette procédure définit tous les paramètres d'accès nécessaires: zones d'échanges, arguments de recherche de segment, des variables de travail, etc...

Les opérations de manipulation de la base IMS sont: GET UNIQUE NEXT, GET NEXT, GET NEXT WITH PARENT, GET HOLD, INSERT, DELETE, REPLACE.

Exemple :

```
CALL DLIPLI(FOUR,GU,PTR-PCB,ZE-CLIENT,ZE-PRET)
```

Remarques :

.FOUR indique le nombre de paramètres,

.GU définit l'opération,

.PTR-PCB est un pointeur sur le PCB,

.ZE-CLIENT et ZE-PRET sont les zones d'échange déclarées dans le programme DLIPLI.

II.3.1.3. Evolution .

L'évolution des systèmes de base de données (dans la version IMS-2 tout au moins) peut être prise en compte à trois niveaux DBD, PSB et programmes d'application:

-- DBD: l'administrateur peut en effet définir une nouvelle description de base de données "virtuelle" à partir de deux ou plusieurs autres DBD à l'aide des pointeurs "logiques" de filiation. On peut aussi modifier des paramètres de certaines des méthodes d'accès

IMS (HISAM et HSAM),

-- PSB: on peut ajouter de nouveaux PSB et pour chaque PSB définir de nouveaux PCB. L'évolution des PCB porte sur les segments "sensibles" et les opérations autorisées,

-- enfin, on peut construire de nouveaux programmes d'application. Dans ce cas, on doit définir de nouveaux PSB (ou simplement des PCB).

II.4. Remarques et conclusions.

Le modèle hiérarchique de données est "attractif" parce que conceptuellement simple .

En contre partie, le langage de manipulation de données pose quelques problèmes d'intégrité. En effet, la suppression d'un enregistrement entraîne la suppression de tous les descendants (DAT77).

Le SGBD SYSTEM 2000 montre qu'il est possible de développer dans le cadre de l'approche hiérarchique des langages de manipulation de plus haut niveau.

Ce S.G.B.D. traite les segments comme des ensembles de valeurs.

Les opérations de manipulation de SYSTEM 2000 utilisent un mécanisme de sélection basé sur la qualification "logique" hiérarchique.

III. APPROCHE RESEAU.

Dans la littérature, l'approche réseau de référence est sans aucun doute CODASYL (COD71). Rappelons que l'objectif initial du groupe DBTG-CODASYL était de fournir des mécanismes et des outils qui permettent de traiter des données intégrées avec le langage de programmation COBOL.

En CODASYL, il n'existe pas comme en SOCRATE (langage de requêtes) de véritable langage autonome ('self-contained'). Nous ne parlerons pas plus de ce système, ayant déjà discuté en détail des principales fonctionnalités du système SOCRATE. Le lecteur intéressé peut trouver des informations détaillées sur cette approche dans (COD71) ainsi que dans (COM76) (DAT77) (NIJ74).

Nous avons vu que le système SOCRATE supportait un interface hôte, COBOL en l'occurrence.

Résumons les principaux inconvénients du système CODASYL:

-- l'utilisation d'un langage de manipulation inséré dans un langage procédural, COBOL,

-- la nécessité de définir de nombreux "détails physiques" dans les schémas et sous-schémas.

IV. APPROCHE RELATIONNELLE.

Les travaux d'E.F.CODD (COD70,71) sont à l'origine de nombreuses propositions et réalisations de prototypes de SGBD relationnels. Les premiers prototypes RDMS (STE74), SEQUEL (AST75), etc.. ont tout d'abord permis de montrer la faisabilité de systèmes supportant des langages de très haut niveau non procéduraux, c'est-à-dire fondés sur l'algèbre ou sur le calcul relationnel.

Avant d'exposer les principaux traits de conception des systèmes prototypes les plus récents INGRES (STO76), SYSTEM R (AST76), nous commençons par rappeler quelques notions fondamentales du modèle relationnel de CODD.

IV.1. Rappels sur le modèle relationnel.

Dans ce paragraphe, lorsque nous parlons de relation, c'est dans le sens mathématique.

Etant donnés les ensembles $D_1,D_2,..,D_n$ (pas nécessairement distincts), une relation R sur $D_1,D_2,..,D_n$ est un ensemble de n-uples $(d_1,d_2,...,d_n)$ avec $d_i \in D_i$ pour $1 \leq i \leq n$. En d'autres termes, R est un sous-ensemble du produit cartésien $D_1xD_2x...xD_n$. On dit alors que la relation R est de degré n et que D_i est le ième domaine de R pour $1 \leq i \leq n$.

Une base de données relationnelle BDR est une collection finie variable dans le temps de relations définies sur une liste finie de domaines $D_1,D_2, ...,D_n$.

Remarques :

.les domaines $D_1,D_2,...,D_n$ peuvent être simples ou composés,

.l'ensemble des tuples d'une relation R, à un instant donné t est appelé instance de R,

.les domaines d'une relation ne sont pas nécessairement distincts. Pour les distinguer, on introduit des sélecteurs appelés attributs .

Une relation R est dite normalisée ou en première forme normale si aucun de ses attributs n'est un attribut composé.

Remarques :

Le mécanisme de normalisation des relations (DEL80) présente un intérêt double: il fournit une aide à la conception de base de données et un moyen de faire abstraction des détails d'implantation: pointeurs, hash-codes, etc...

Soit $A=(A_1,A_2,..,A_n)$ la liste des attributs associés aux domaines respectifs $D_1,D_2,..D_n$ d'une relation R, $r.A=(r.A_1,r.A_2,..,r.A_n)$ où $r \in R$ et $\bar{A}$ l'ensemble des attributs de R qui ne sont pas contenus dans A.

On dit que deux domaines simples sont union-compatibles s'ils sont tous les deux de même "nature". Ainsi, deux relations sont union-compatibles si les domaines composés , sous-ensembles de R et S sont union-compatibles.

IV.1.1. Dépendance fonctionnelle.

Un attribut A_1 de la relation R est fonctionnellement dépendant d'un attribut A_2 de R si, à tout instant, à chaque valeur de A_1, il n'est possible d'associer qu'au plus une valeur de A_2 dans R, ce que l'on note $R.A_1 \rightarrow R.A_2$. Lorsque deux attributs ne sont pas en dépendance fonctionnelle, on écrit $R.A_1 \not\rightarrow R.A_2$. C.DELOBEL (DEL80) a caractérisé les différentes propriétés de la dépendance fonctionnelle.

IV.1.2. Clés candidates et clés primaires.

Nous définissons une clé candidate K d'une relation R $(A_1, A_2, .., A_n)$ comme une collection d'attributs avec les propriétés suivantes:

Propriété 1 : l' identification unique :

Pour chaque tuple r d'une relation R, la valeur de l'attribut K détermine de manière unique le tuple r.

$R.K \rightarrow R.(A_1, A_2, .., A_n)$

Propriété 2 : la non-redondance :

Aucun attribut de K ne peut être supprimé sans détruire la propriété 1

Propriété 3 : Chaque attribut de K est fonctionnellement dépendant de chaque attribut de R.

Propriété 4 : la collection des attributs de R qui constituent la clé candidate K doit satisfaire à:

-- chaque sous-ensemble propre S_1 de K est fonctionnellement indépendant de tous les autres sous-ensembles propres S_2 de K lorsque

$S_1 \not\subseteq S_2$ et $S_2 \not\subseteq S_1$,

-- aucun attribut de R ne peut être ajouté dans K sans détruire l'indépendance fonctionnelle de K.

Une des clés candidates est alors sélectionnée comme clé primaire de la relation R. Cette clé primaire sert d'identification unique des tuples de la relation R. Cette clé peut aussi apparaître comme clé d'autres relations, on l'appelle alors clé étrangère ("foreign key").

Remarque : Aucune des valeurs des attributs d'une clé primaire ne doit être indéfini.

IV.1.3. Algèbre relationnelle.

Opérations de la théorie des ensembles : les opérations d'union ($\cup$), d'intersection ($\cap$) et de différence(-) doivent s'appliquer à des relations union-compatibles. Pour le produit cartésien, les relations peuvent être quelconques.

Opérations relationnelles :

. le produit cartésien relationnel est défini par:

$R \otimes S = \{(\widehat{rs}) : r \in R \wedge s \in S\}$ de degré n+m

. la projection par:

$R[A] = \{r.A : r \in R\}$ de degré card(A)

. la jointure ("join"): Etant données deux relations R et S, R.A et S.B deux listes d'attributs comparables par rapport aux opérateurs de relations θ :=, $\neq$, <, $\leq$, $\geq$, >. On définit alors le θ-join de R et S sur les attributs A et B par

$R(A \theta B)S = \{(\widehat{rs}) : r \in R \wedge s \in S \wedge (r.A \theta r.B)\}$

Pour supprimer la redondance qui apparait lorsqu'on utilise l'opération d'égalité, E.F.CODD a introduit le natural-join défini par:

$$R(A*B)S=\left\{\widehat{rs}.\bar{B}: r \in R \wedge s \in S \wedge (r.A=r.B)\right\}$$

Exemples : Considérons les relations:

$R(A_1,A_2,A_3)$			$S(B_1,B_2)$	
a_1	1	1	2	b_1
a_1	2	1	3	b_2
a_3	1	2		
a_4	2	5		

Alors:

$R(A_2=B_1)S$

A_1	A_2	A_3	B_1	B_2
a_1	2	1	2	b_1
a_4	2	5	2	b_2

et

$R(A_2*B_1)S$

A_1	A_2	A_3	B_2
a_1	2	1	b_1
a_4	2	5	b_2

. la division : Etant données deux relations R et S, les listes d'attributs A et B respectivement de R et S telles que les domaines composés définis par A et par B soient union-compatibles, alors l'opération de division est définie par:

$$R\left[A\%B\right]=R\left[\bar{A}\right]-((R\left[\bar{A}\right]\otimes S\left[\bar{B}\right])-R)\left[\bar{A}\right]$$

. la restriction : Etant donnée une relation R et deux attributs A et B comparables relativement à l'opérateur Θ, alors la restriction est définie par:

$$R\left[A\Theta B\right]=\left\{r \in R \wedge (r.A\Theta r.B)\right\}$$

Remarque 1: les opérations de l'algèbre relationnelle présentent quelques inconvénients que nous résumons brièvement ci-dessous (PIR76):

.la manipulation d'objets simples ne peut se décrire que sous forme de relations,

.l'introduction de fonctions standard: MIN, MAX, etc.. ne peut être faite que sous forme de "mapping" sur des relations.

Remarque 2: les opérations de l'algèbre relationnelle offrent un moyen de spécification des problèmes des utilisateurs en termes de relations sur un "univers" de relations normalisées.

IV.1.4. Calcul relationnel.

Les remarques 1 et 2 précédentes justifient le développement du calcul relationnel. En effet, il semble tout aussi important de pouvoir spécifier un problème en termes de propriétés des objets i.e sous forme d'expressions logiques (FIN79).

La formulation d'expressions dans le calcul relationnel utilise:

-les constantes: $a_1,a_2,\cdots$

-les noms d'attributs: $d_1,d_2,\cdots$

-les variables de tuples: $r_1,r_2,\cdots$

-les prédicats: $p_1,p_2,\cdots$

-les symboles d'opérateurs de comparaison: $=,>,<,\leq,\geq,\neq$

-les connecteurs logiques: $\exists,\forall,\vee,\wedge,\neg$

-les délimiteurs: $[,],(,)$

De plus, il existe une bijection entre l'ensemble des prédicats $p_1,p_2,..,p_n$ et l'ensemble des relations $R_1,R_2,..,R_n$ du modèle relationnel, p_j caractérisant l'appartenance de tuples à R_j (j=1,..,n).

Les termes du calcul relationnel sont uniquement formés:

-- de " range term " qui ont la forme $p_j r$ où p_j est un prédicat et r une variable de tuple, le domaine de r étant la relation R_j,

-- ou encore, de composants de tuple r.d, r étant une variable de tuple et d un attribut identifiant la composante de ce tuple.

Soit les composants de tuple e et f, θ un symbole d'opérateur de comparaison et a une constante, on appelle alors join term les termes $e\theta f$ et $e\theta a$.

Nous pouvons maintenant résumer les propriétés des formules bien formées (WFF) du calcul relationnel qui sont les suivantes:

1) un "range term" et un "join term" sont des WFF,

2)si e est une WFF, alors $\neg e$ est aussi une WFF,

3)si e,f sont des WFF, alors $e \vee f$ et $e \wedge f$ sont des WFF,

4)si e est une WFF dans laquelle r apparaît comme une variable libre, alors $\exists r(e)$ et $\forall r(e)$ sont des WFF,

5)il n'existe pas d'autre WFF.

Pour définir proprement les propriétés du domaine cible (range term) des variables de tuple, E.F.CODD a introduit le concept de "range separability" qui restreint les formules bien formées.

Exemple de formule bien formée "range separable":

$p_6 r_2 \wedge (r_2.d_1 = a_1)$ de variable libre r_1

Exemple de formule bien formée non "range separable":

$p_8 r_1 \vee (r_1.d_3 = a_3)$ de variable libre r_1

Ces formules bien formées "range separable" permettent de spécifier des expressions logiques caractérisant les propriétés des objets sélectionnés dans une base de données. Ainsi, une alphaexpression simple a la forme suivante:

$$\{(t_1, t_2, .., t_n) : w\}$$

où w est une formule bien formée range separable et $t_1, t_2, .., t_n$ sont soit des variables de tuples, soit des composantes de tuples telles que l'ensemble des variables de tuples apparaissant dans $t_1, .., t_n$ est précisément l'ensemble des variables libres de w.

La liste $(t_1, t_2, .., t_n)$ est appelée liste cible (target list) et w l'expression de qualification .

Le résultat de l'évaluation d'une alphaexpression simple est une relation obtenue par la projection sur la liste cible du sous-ensemble de $R_1 \otimes R_2 \otimes \ldots \otimes R_n$ qui satisfait à l'expression de qualification et où $R_1, .., R_n$ sont les cibles des variables libres de tuples de w.

Une alphaexpression générale se caractérise donc par:

--toute alphaexpression simple est une alphaexpression,

--si $\{t : w_1\}$ et $\{t : w_2\}$ sont des alphaexpressions, alors

$\{t : (w_1 \vee w_2)\}$

$\{t : (w_1 \wedge \neg w_2)\}$

$\{t : (w_1 \wedge w_2)\}$ sont des alphaexpressions,

--il n'existe pas d'autre expression qui soit une alphaexpression.

Pour terminer ces rappels, E.F.CODD a aussi montré dans ses travaux (COD70) la complétude de l'algèbre relationnelle qu'il avait définie.

IV.2. Remarques générales.

Les deux théories que nous venons de présenter sont complémentaires. Une spécification d'énoncé de problèmes peut se décrire sous forme de séquence d'opérations relationnelles et d'expressions logiques.

D'autre part, bien que les S.G.B.D. relationnels récents n'utilisent que des relations en première forme normale, il importe de préciser que la mise à jour, la suppression et l'insertion de données exigent quelques précautions si l'on ne veut pas détruire la cohérence des bases. Il existe plusieurs représentations de relations en troisième forme normale. Nous citerons la définition originale proposée par BOYCE et CODD (DAT77): "une relation R est en troisième forme normale si elle est en première forme normale et, pour chaque collection d'attributs A de R, si tout attribut qui n'est pas dans C est fonctionnellement dépendant de C, alors tous les attributs de R sont fonctionnellement dépendants de C".

Intuitivement, cette définition signifie que chaque relation ne doit représenter qu'un "concept", sinon la relation devra être éclatée en plus petites relations. Une étude détaillée de ce type de propriétés des relations est exposée dans la thèse d'A.FLORY (FLO77). Actuellement, certains prolongements sur les fonctions multivaluées sont proposées (DEL80) (DEE77).

IV.3. Les S.G.B.D. relationnels.

Dans cette partie, nous présentons les concepts et les mécanismes de SGBD relationnels les plus récents. Nous nous sommes essentiellement intéressés aux langages -- interrogation, définition, manipulation des données --, à l'insertion de ces langages ("embedding") dans des langages hôtes de programmation et aux mécanismes de protection des données. Les promotteurs de prototypes de S.G.B.D. relationnels visent en effet de multiples objectifs très largement développés dans la littérature (COD71) (PIR74) (STO76) (ZLO75) (BOY74) (BOY75) (PIR76) (COM76) (AST76) (KIM79), etc... et que nous résumons brièvement ci-dessous.

Les S.G.B.D. relationnels doivent fournir:

a) des interfaces de haut niveau, non procéduraux, pour plusieurs classes d'utilisateurs: langage d'interrogation, langage de manipulation, langage de définition et outils adaptés aux contrôles des données (sécurité, intégrité des données),

b) des structures de fichiers et différentes méthodes d'organisation sophistiquées qui permettent des stockages de données et des accès efficaces,

c) des mécanismes d'optimisation de stratégies de décomposition de requêtes utilisateurs,

d) des mécanismes de construction de "fenêtres" sur les bases de données. Ces fenêtres définissent des vues dynamiques ('views') ou statiques ('snapshots') sur la base de données,

e) un mécanisme de description des contraintes d'intégrité utilisateur qui soit orthogonal avec les langages de a),

f) des contrôles de synchronisation sophistiqués pour la mise à jour concurrentielle de données partagées par les différents utilisateurs des bases de données,

g) des mécanismes de contrôle d'accès sélectifs par autorisation, révocation, modification des privilèges d'accès d'un utilisateur de base de données à d'autres utilisateurs, qui soient orthogonaux avec a),

h) des mécanismes de récupération ('recovery') en cas de défaillance logicielle ou matérielle.

IV.3.1. Langages d'interface utilisateurs.

Jusqu'à présent, c'est certainement l'aspect langage d'interrogation qui a été le plus développé dans le domaine des bases de données relationnelles. Il y a des raisons historiques à cela. En effet, les premiers prototypes directement issus de la théorie de E.F.CODD sur le modèle relationnel, citons SQUARE (BOY75) orienté calcul relationnel et PRTV (NOT72) basé sur l'algèbre relationnelle, ont eu pour objectif essentiel de montrer la faisabilité de telles approches. Il est vrai que par rapport aux autres approches de S.G.B.D., les nouvelles propositions présentent des différences importantes. L'utilisation des langages orientés calcul des prédicats ou algèbre relationnelle permettent de dégager totalement l'utilisateur de détails d'implantation. En fait, ces S.G.B.D. réalisent une véritable indépendance des données. Les incidences sont fondamentales. Il devient ainsi possible d'entreprendre indépendamment des langages de définition

et d'interrogation, la réalisation des choix de stockage, de méthodes d'organisation des données et d'optimisations de stratégies de décomposition de requêtes.

Evidemment, ces prototypes de S.G.B.D. étaient peu efficaces et surtout utilisaient des langages d'interrogation comportant de nombreuses sophistications mathématiques. Comme le montre A.PIROTTE (PIR76), l'utilisation des quantificateurs dans les expressions de ALPHA (COD71) ou l'introduction des variables dans SQUARE posent de sérieux problèmes de compréhension aux utilisateurs qui ne sont pas habitués à manipuler des langages formels. Le trait de langage le plus intéressant de SQUARE, tout au moins celui qui a le plus influencé les développements ultérieurs QUEL (HEL75) et SEQUEL (BOY74) est certainement le concept de "mapping". Détaillons ce concept dans le cadre du langage d'interrogation SQUARE.

Considérons la relation employé EMP qui comporte les domaines NOEMP, NOM, SAL, MAN, NODPT:

EMP(NOEMP,NOM,SAL,MAN,NODPT)

et supposons que l'on formule la requête suivante:

"trouver les noms des employés du département 30".

La requête SQUARE s'écrit:

```
    EMP      ('30')
NOM  NODEPT
```

Le mapping se définit par la donnée du nom de la relation (EMP), le nom du domaine source (NODPT), le nom du domaine cible (NOM) et l'argument, ici ('30'). Le résultat du mapping est l'ensemble (éventuellement réduit à un élément) des valeurs de la colonne cible du domaine NOM de la relation EMP dont les valeurs de la colonne source du domaine NODPT sont 30.

Toutes les fonctions standard COUNT, AVG, MAX, MIN, sont définies à l'aide de concept de mapping. On doit toutefois distinguer syntaxiquement les mappings sur les ensembles et les multi-ensembles. Par exemple, pour calculer la moyenne des salaires des employés du département jouet en SQUARE, le caractère ' indique que le résultat du mapping est un multi-ensemble:

```
AVG(   EMP'    ('30'))
    SAL    DEPT
```

Nous présentons maintenant quelques traits caractéristiques des langages d'interface utilisateurs dérivés de ALPHA et SQUARE.

IV.3.1.1. Le langage SEQUEL (BOY74).

SEQUEL (Structured English QUEry Language) est un langage d'interrogation relationnel qui utilise une syntaxe basée sur des blocs structurés de mots clés anglais. SEQUEL est un langage orienté mapping (COM76). Le prototype SYSTEM R développé par IBM (AST76) utilise une version de SEQUEL appelée SQL aussi bien pour l'interrogation, la modification, la suppression, l'insertion des bases de données relationnelles que pour définir des relations, des 'vues', des contrôles d'intégrité et des contrôles d'accès aux données.

Pour présenter le langage d'interrogation de SEQUEL, considérons la relation:

EMP(NOEMP,NOM,SAL,MAN,NODPT) définie précédemment et la relation DEPT(NODPT,NUMDPT,ADRESSE). En SEQUEL, ces relations sont manipulées comme des tableaux à double entrée.

a) le mapping simple :

Reprenons la question formulée précédemment:

Q1:trouver les noms des employés du département 30.

```
SEQUEL: SELECT NOM
        FROM EMP
        WHERE NODPT=30
```

Cette requête illustre la définition du mapping que nous avons défini pour SQUARE.

Q2:trouver tous les employés qui ont un salaire de 5000"

```
SEQUEL: SELECT EMP
        WHERE SAL=5000
```

Cette requête correspond à l'opération de sélection de l'algèbre relationnelle.

Q3:lister les noms d'employés.

```
SEQUEL: SELECT NOM
        FROM EMP
```

C'est l'opération de projection de l'algèbre relationnelle. Plusieurs fonctions standard sont disponibles en SEQUEL: SUM, COUNT, MAX, MIN et AVG.

Q4:calculer la moyenne des salaires des employés du département 30.

```
SEQUEL: SELECT AVG (SAL)
        FROM EMP
        WHERE NODPT=30
```

Notons que par rapport à SQUARE, il n'existe plus de distinction syntaxique pour le mapping.

SEQUEL permet aussi d'utiliser les opérations ensemblistes: union ($\cup$), intersection ($\cap$), différence (/) entre plusieurs mappings.

Q5:lister les numéros de départements qui se trouvent à Nancy et qui

ont des employés qui gagnent moins de 3000F.

SEQUEL: SELECT NODPT
FROM DEPT
WHERE ADRESSE='NANCY'
∩
SELECT NODPT
FROM EMP
WHERE SAL<3000

Q6:lister les noms des employés qui travaillent pour les départements de NANCY.

SEQUEL: SELECT NOM
FROM EMP
WHERE NODPT IN
SELECT NODPT
FROM DEPT
WHERE ADRESSE='NANCY'

Cette requête illustre la composition des mappings.

Enfin, pour des requêtes plus complexes, SEQUEL fournit comme en SQUARE des constructions qui permettent à l'utilisateur de ne pas avoir à définir des variables.

Q7:lister les numéros de département dont la moyenne des salaires est supérieure à 100000.

SEQUEL: SELECT NODPT
FROM EMP GROUP BY NODPT
WHERE AVG(SAL)>100000

Q8:lister pour chaque département le numéro de département, son nom et la moyenne des salaires des employés.

SEQUEL: SELECT NODPT,NOMDPT,VAR1
FROM VAR2 IN DEPT
COMPUTE VAR1=
SELECT AVG(SAL)
FROM EMP
WHERE NODPT=VAR2.NODPT

Ce dernier exemple montre que pour formuler des questions un peu plus élaborées, l'utilisateur doit introduire des variables; ce qui pose des problèmes de compréhension (REI77).

IV.3.1.2. Le langage QUEL (HEL75).

QUEL (QUEry Language) développé comme langage d'interface du système relationnel INGRES (HEL75) possède beaucoup de similarités avec SEQUEL.

En QUEL, il est nécessaire de déclarer pour chaque requête le domaine des variables utilisées dans des commandes:

RANGE OF <liste-de-variables> IS nom-de-relation

Ces variables désignent précisément des tuples de la relation. Une commande a la forme syntaxique suivante:

<command><nom-du-résultat(liste-cible)>
WHERE qualification

Comme pour SEQUEL, le langage d'interface utilisateur QUEL est utilisé aussi bien pour l'interrogation (RETRIEVE), l'adjonction (APPEND), la modification (REPLACE) que la suppression (DELETE) de tuples dans les relations.

Dans les cas RETRIEVE et APPEND, le nom-résultat est alors le nom de

la relation traitée. La liste cible est de la forme <domaine-de-la-relation>=<fonction>. Illustrons sur les exemples suivants quelques traits caractéristiques de QUEL.

Q1:trouver les noms des employés du département 30.

EQUEL: RANGE OF E IS EMP
RETREIVE INTO W(LENOM=E.NOM)
WHERE E.NODPT=30

Ici, E est une variable de tuple qui opère sur la relation EMP. La commande RETREIVE définit une nouvelle relation W avec un seul domaine LENOM. Chaque valeur de cette nouvelle relation correspond à une valeur E.NOM de tuple de la relation EMP qui satisfait à la qualification E.NODPT=30. Si le nom de la relation résultat W est omis, alors les valeurs sont affichées au terminal de l'utiliisateur.

Q2:lister les départements dont la moyenne des salaires est supérieure à la moyenne des salaires de l'entreprise, ces moyennes ne concernant que les employés dont le salaire est supérieur à 5000F.

QUEL: RANGE OF E IS EMP
RETRIEVE INTO HIGHPAY(E.NODPT)
WHERE
AVG (E.SAL BY E.NODPT WHERE E.SAL>5000)
> AVG (E.SAL WHERE E.SAL>5000)

Ici, AVG (E.SAL BY E.NODPT WHERE E.SAL>5000) est défini comme une fonction d'aggrégat ('aggregate') de AVG (E.SAL WHERE E.SAL>5000 AND E.NODPT='valeur') qui fournit une valeur pour chaque E.NODPT.

Comme en SEQUEL, certaines requêtes peuvent poser quelques problèmes de compréhension. Par exemple:

Q3:pour chaque département, lister la moyenne des salaires des employés qui gagnent plus que la moyenne des salaires de leur département.

QUEL: RANGE OF E IS EMP
RANGE OF F IS EMP
RETRIEVE (COMP= AVG (E.SAL BY E.NODPT
WHERE E.SAL
> AVG (F.SAL WHERE F.NODPT=E.NODPT)),
BY NODPT=E.NODPT

La variable E de la clause BY a par définition une portée globale alors que la variable E de E.SAL n'a qu'une portée locale à l'aggrégation. Il devient difficile dans ces conditions d'écrire des requêtes compréhensibles et à fortiori correctes.

IV.3.1.3. QUERY-BY-EXAMPLE (ZL075).

QUERY-BY-EXAMPLE fournit un langage d'interrogation graphique de haut niveau, non procédural. Cet outil logiciel développé par l'équipe de SYSTEM R est particulièrement attrayant.

Illustrons sur un exemple comment s'utilise ce système. Pour lister le nom des employés du département 30, on demande d'afficher un "modèle" de table que l'utilisateur complète comme suit:

EMP	NOM	NODPT	

Ensuite, on entre le prédicat de sa requête "NODPT=30" puis on spécifie les renseignements P(PRINT) et le nombre maximum d'éléments désirés.

EMP	NOM	NODPT
	P.50	=.30

Ici, l'utilisateur demande que 50 noms d'employés du département 30 soient affichés.

IV.3.2. Insertion des langages d'interface dans des langages hôtes.

IV.3.2.1. Le langage EQUEL du système INGRES (HEL75).

EQUEL (Embedded QUEL) est un langage de programmation qui insère QUEL dans le langage de programmation général "C" développé sur le système d'exploitation UNIX.

EQUEL est un langage qui permet d'utiliser à la fois QUEL et "C". Les C-programmes traitent individuellement chaque tuple sélectionné par les qualifications d'une commande RETRIEVE de QUEL. Pour illustrer la syntaxe d'EQUEL, nous considérons un C-programme qui lit des noms d'employés et qui imprime leur salaire.

```
EQUEL: main ( )
       {
    ## char ENOM 20 ;
    ## int SAL
       while( READ (ENOM))
       {
    ## RANGE OF E IS EMP
    ## RETRIEVE (SAL=E.SAL)
    ## WHERE E.NOM=ENOM
```



```
    ##  {
        PRINT ("le salaire de",ENOM,"est",SAL);
    ##  }
       }
    }
```

Notons d'abord que le langage est à structure de bloc où "{" et "}" correspondent à "DEBUT" et "FIN" d'Algol 60. Les instructions "C" sont aussi des instructions EQUEL. Les instructions QUEL sont des instructions EQUEL lorsqu'elles sont préfixées par le symbole "##". Les C-variables peuvent alors être utilisées dans des instructions QUEL comme noms de relations de domaines d'éléments cibles ou encore comme valeurs de domaines. Dans ce cas, les C-variables doivent aussi être préfixées par le symbole "##".

Ainsi, pour le C-programme décrit ci-dessus, la C-variable ENOM agit comme une valeur de domaine pour l'instruction RETRIEVE de QUEL et pour chaque tuple qualifié.

Précisons encore que les autres commandes de QUEL: REPLACE (mise à jour des tuples), APPEND (adjonction de nouveaux tuples) et DELETE (suppression de tuples) fournissent les commandes complémentaires de ce qui est parfois appelé le langage de manipulation de base de données (L.M.D.). Ce L.M.D. est évidemment inséré dans EQUEL.

Exemple d'utilisation de la commande REPLACE en QUEL.

Q:on veut augmenter de 20% le salaire de "Karol" qui travaille au département "informatique". On écrira en QUEL:

```
RANGE OF E IS EMP
RANGE OF D IS DEPT
 REPLACE E(SAL=1.2*E.SAL)
```

WHERE E.NOM="Karol" AND

E.NODPT=D.NODPT AND D.NOMDPT='informatique'

Ici, E.SAL est remplacé par 1.2*E.SAL pour chaque tuple E qui satisfait à la qualification.

Enfin, une restriction de QUEL permet de spécifier des contrôles d'intégrité des droits d'accès ainsi que des vues. Détaillons ces trois aspects sur des exemples.

Vues sur les relations

RANGE OF E IS EMP

DEFINE VEMP(E.NOM,E.SAL,E.NODPT)

WHERE E.NODPT=30

La syntaxe de la commande DEFINE est la même que celle de RETRIEVE; les fonctions d'aggrégat ne sont toutefois pas autorisées. La visibilité sur la relation de base est restreinte aux domaines NOM, SAL et NODPT. De plus, la vue VEMP ne contient que des tuples de la relation de base EMP qui satisfont à la propriété E.NODPT=30. Ajoutons que DEFINE est une commande du SGBD INGRES qui permet de cataloguer une vue.

Contrôles d'intégrité

Les contraintes d'intégrité peuvent être "statiques": elles caractérisent tous les états de la base; ou "dynamiques": elles caractérisent alors les états de transition valides. En outre, les assertions d'intégrité peuvent être définies aussi bien sur des tuples que sur des ensembles de tuples (WEB78).

L'exemple qui suit montre comment on peut définir des contraintes d'intégrité. Supposons que tous les employés de la relation EMP ne puissent qu'appartenir au département 30. On écrira:

RANGE OF E IS EMP

INTEGRITY CONSTRAINT IS E.NODPT=30

Une commande d'INTEGRITY est une commande du S.G.B.D. INGRES qui catalogue la contrainte d'intégrité.

Protection, sécurité

Rappelons aussi que la protection-sécurité est un mécanisme du S.G.B.D. qui vérifie les droits d'accès aux données partagées de la base. Supposons que l'on veuille s'assurer que les seules personnes habilitées à modifier les salaires des employés sont les managers de ces employés, on écrira:

RANGE OF E IS EMP

PROTECT EMP FOR ALL (E.SAL;E.NOM)

WHERE E.MAN=*

Ici, * est le nom de l'utilisateur défini au "login". La commande PROTECT est aussi une commande INGRES qui catalogue l'assertion dans un fichier du système.

Plusieurs autres commandes INGRES ont été élaborées pour créer ou détruire des bases de données, créer ou détruire des relations, effectuer des copies de relations, des modifications d'organisation, etc... ainsi que des mécanismes de contrôle de concurrence d'accès qu'il serait trop long de développer.

IV.3.2.2. RDI, langage d'insertion de SEQUEL de SYSTEM R.

Le RDI (Relational Data Interface) est le principal interface externe de SYSTEM R. Dans cette partie, nous n'illustrons que les

aspects interface en langage hôte de SEQUEL, les principes de langage de manipulation, de contrôle d'intégrité, de définition de vue sont assez similaires. SYSTEM R propose un mécanisme de contrôles de droits d'accès beaucoup plus élaboré que INGRES mais qui nécessiterait de longs développements (GRI76). L'interface de SEQUEL RDI utilise un concept clé appelé curseur ('cursor'). Un curseur est un nom donné à un ensemble de tuples appelé ensemble actif ('active set') utilisé pour maintenir la position d'un tuple dans un ensemble. Une opération FETCH permet de retrouver les tuples un à un.

En fait, le RDI est constitué d'un ensemble d'opérateurs SEQUEL: FETCH, BIND, etc..(AST76) que l'on utilise dans un environnement PL/I par des appels de subroutines.

programme d'interface

Supposons que l'on veuille lister les noms et les salaires des employés du département 30.

```
RDI: CALL BIND ('X',ADDR(X));
     CALL BIND ('Y',ADDR(Y));
     CALL SEQUEL (C1,' SELECT NOM:X,SAL:Y
                FROM EMP
                WHERE NODPT=30');
     CALL FETCH (C1);
```

Ce programme montre que les adresses systèmes des variables X et Y doivent d'abord être définies par des appels de BIND.

L'appel de SEQUEL a ensuite pour effet d'associer un curseur C1 à l'ensemble des tuples qui répondent à la question et ensuite de positionner celui-ci sur le premier tuple trouvé. Les tuples sont extraits un à un de l'ensemble actif à chaque appel de FETCH. Il existe

plusieurs opérateurs de manipulation des curseurs: FETCH-HOLD pour traiter la concurrence d'accès aux données partagées, RELEASE, OPEN, CLOSE.

IV.3.3. Implantation des S.G.B.D. relationnels.

Pour les prototypes de S.G.B.D. relationnels que nous venons de présenter, plusieurs autres aspects importants concernant: la traduction des interfaces, la décomposition des requêtes avec optimisation des chemins d'accès, la concurrence d'accès aux données partagées, etc... sont fortement liés aux systèmes d'exploitation utilisés. Tous ces SGBD ont été réalisés de façon modulaire: INGRES sur PDP11 45 avec UNIX et SYSTEM R sur IBM 370 VM.

IV.3.4. Autres propositions.

Dans les paragraphes précédents, nous avons tenu à présenter des approches qui ont permis de valider, au prix de très gros efforts de développement, un grand nombre d'idées nouvelles. Quelques autres réalisations, certes moins ambitieuses, nous ont paru riches d'enseignements. Nous en exposons maintenant quelques-unes.

IV.3.4.1. Le système PIVOINE (CRE75).

Le système expérimental PIVOINE propose des concepts originaux dans le domaine des bases de données. Il permet de bien traiter l'indépendance des données et surtout de réaliser un mécanisme d'optimisation automatique de stratégies d'accès sur des structures de données complexes. Le modèle de donnée de PIVOINE est fondé sur les notions formelles de données (valeurs ou repères) et de relations orientées entre ces données. La décomposition du système en plusieurs niveaux: LOGIQUE, INTERNE (physique) et PHYLOG (physico-logique) fournit un cadre général (PHYLOG) pour réaliser l'indépendance des données entre les programmes de requêtes exprimés dans un langage déclaratif sur la structure LOGIQUE et l'implantation des structures physiques dans INTERNE.

IV.3.4.2. Le système SYNTEX (DEM75).

SYNTEX-2 est un S.G.B.D. relationnel réalisé au CERT-DERI de Toulouse. Le langage d'interface non procédural est orienté calcul des prédicats avec les notions de " variables individuelles et de tuples " introduites par A.PIROTTE dans FQL (PIR74). Les constituants de base du langage sont:

-- des valeurs (" atomes ") ou désignation de valeurs (" molécules")

-- des relations à plusieurs arguments définies

.sur des atomes

.sur des molécules,

.ou par des "algorithmes",

-- des informations élémentaires qui sont des assertions.

Autrement dit, un univers se définit avec des termes appelés atomes et molécules, des axiomes appelés relations et des assertions qui décrivent la "connaissance" de la validité de certaines relations à n arguments pour certains n-uples de termes. L'interrogation résulte de l'évaluation d'expressions "fermées" (sans variable) qui fournit une réponse oui/non aux questions posées et d'expressions "ouvertes" (n variables libres, $n \geq 1$) qui délivrent des n-uplets d'atomes pour lesquels l'expression a la valeur vrai. Dans ce cas, l'évaluateur utilise pour déterminer ces n-uplets des procédures de recherche de données explicites stockées dans la base et des procédures de déduction sur les axiomes de l'univers. La mise à jour peut porter sur les termes, les axiomes et les assertions. Notons que pour certaines questions, il est nécessaire d'introduire des quantificateurs ($\exists$) et ($\forall$). D'autre part, ne serait-ce que pour effectuer des opérations de calcul élémentaires, il faut utiliser un interface hôte qui est COBOL!

Plusieurs travaux du domaine de l'intelligence artificielle (KEL78) (MIN73) utilisent cette approche pour des systèmes "question-réponse" déductifs.

Comme nous l'avons déjà exprimé dans une remarque sur le calcul relationnel, il nous semble que si cette approche fournit un outil intéressant pour manipuler des propriétés d'objets dans des cas simples (PIR74), elle échoue si l'on veut manipuler des ensembles, par exemple avec des opérations de l'algèbre relationnelle.

IV.3.4.3. La programmation logique.

L'idée de K.A.KOWALSKI (KOW74) a été d'employer la logique des prédicats à la fois comme langage de programmation et comme langage de résolution de problèmes par décomposition de problèmes en sous-problèmes. Par rapport au langage SYNTEX précédent, il s'agit d'une "interprétation procédurale" de la logique des prédicats du 1er ordre en forme clausale (clauses de Horn pour simplifier) telle que:

+B-A1-A2-...-An

(qu'il faut lire "B si A1 et A2 et...et An")

qui fournit une technique puissante de résolution de problème par décomposition de problème de la forme B en une suite de sous-problèmes A1,A2,..,An.

Une clause de Horn est ainsi interprêtée comme une "déclaration de procédure d'entête B" dont le corps est constitué d'une suite d'appels de procédures A1,A2,..,An. Les "appels de procédures" ne correspondent pas à l'acception des langages de programmation classiques en ce sens qu'il n'y a pas de distinction entre paramètres entrées et résultats.

Dans ce contexte, on parle parfois de <u>programmation logique</u> (KOW76). Cette approche nous a paru intéressante car elle fournit des "concepts relativement simples" pour construire des programmes logiques par "raffinements successifs": les déclarations de procédures spécifiant pour B la partie logique de l'énoncé et les A1,A2,..,An la stratégie de résolution du proclème.

Le langage PROLOG (ROU75) que nous présentons dans la deuxième partie, fournit d'autres possibilités intéressantes comme la gestion des entrées/sorties, la création ou suppression de clauses et de symboles, le contrôle de stratégie de résolution ("/"), le traitement de caractères et d'entiers, etc.. En PROLOG, une requête se décrit par

une clause de la forme:

-B1-B2-..-Bn où les (Bi,$1 \leq i \leq n$) sont des appels de procédures des seules parties logiques des clauses qui ont été déclarées. Il est intéressant de relever les similitudes, comme le montre M.H.VAN EMDEN (EMD79), entre les descriptions de requêtes sous forme de clauses de réfutation et la présentation sous forme de tableaux de QUERY-BY-EXAMPLE.

V. <u>REMARQUES ET CONCLUSIONS</u>.

Les efforts importants que nous avons investis dans l'étude des propositions et réalisations de SGBD nous incitent d'abord à constater que nombre de débats sur les modèles de bases de données, où l'on passe beaucoup de temps à comparer son modèle aux autres pour savoir quel est le meilleur (FLO77), pourraient devenir plus constructifs dans un cadre d'ingénierie de logiciel. Car, si on se place dans ce cadre, il faut reconnaître que le modèle relationnel de données et les langages non procéduraux d'interrogation de INGRES et de SYSTEM R, par exemple, sont assez séduisants.

La réalisation effective de l'indépendance des données, de langages d'interrogation compréhensibles pour un grand nombre de requêtes, ainsi que la possibilité de concevoir et de réaliser séparément des optimisations sur les méthodes d'organisation de fichiers et de stratégies de recherche de chemins d'accès pour la décomposition des requêtes, la possibilité d'ajouter et de supprimer des relations, de définir des vues, des contrôles d'intégrité, de sécurité et de

concurrence d'accès aux données partagées dans des langages orthogonaux avec le langage d'interrogation, nous semblent constituer un catalogue de qualités suffisamment éloquent.

Les "faiblesses" de ces SGBD se situent au niveau de l'insertion des langages d'interface dans des langages hôtes pour la manipulation de données. Il nous semble en effet que les langages d'interrogation et de manipulation de bases de données relationnelles, utilisent des concepts et un niveau de programmation non procédural incompatible avec les langages de type "C" pour INGRES ou PL/1 pour SYSTEM R. Les propositions du projet TYP visent précisément à dégager des mécanismes et des outils orthogonaux qui diminueraient ces inconvénients.

CHAPITRE III. APPROCHE LANGAGES DE PROGRAMMATION GENERAUX ET SPECIALISES.

I. RAPPELS DE QUELQUES NOTIONS DE BASE SUR LES LANGAGES ET LA PROGRAMMATION.
I.1. La programmation structurée.
I.2. La notion de type de données.
I.3. La notion de module fonctionnel.
I.4. La protection dans les langages.
I.4.1. Contrôles d'accès.
I.5. L'évolutivité.
II. QUELQUES LANGAGES EXISTANTS.
II.1. PL/1.
II.2. Le langage PASCAL.
II.3. Le langage SIMULA.
II.4. LIS.
II.5. Remarques.
III. QUELQUES APPROCHES DE LANGAGES RECENTS.
III.1. EUCLID.
III.2. ALPHARD.
III.2.1. La notion de form.
III.2.2. Générateurs et itérations.
III.2.3. Protection et contrôle d'accès.
III.3. CLU.
III.3.1. La notion de cluster.
III.3.2. La liaison de l'interface abstrait à l'implantation.
III.3.3. Les itérateurs.
III.4. ADA.
III.4.1. La notion de package.
III.4.2. Les packages génériques.

IV. CONCLUSIONS.

Dans ce chapitre, nous nous intéressons à quelques langages de programmation existants ou de conception récente qui fournissent certains des mécanismes que nous énumérons ci-dessous:

-- description modulaire des types de données et des actions,

-- spécification abstraite modulaire de l'implémentation de types de données: profils syntaxiques des opérations abstraites associées et description de leur comportement en langue naturelle ou dans un langage formel. Nous portons un intérêt particulier à l'indépendance de cette spécification vis à vis de l'implémentation,

-- protection (confinement, confidentialité) par contrôle de privilèges d'accès aux objets,

-- extensibilité et adaptabilité (techniques de paramétrisation, multireprésentation), mais aussi d'évolutivité.

Pour beaucoup de langages de programmation prototypes, nous n'avons que des spécifications sommaires. Nous n'exposons donc ici que ceux qui nous ont permis de tirer quelques enseignements. Quelques études de synthèses peuvent apporter des informations plus détaillées sur certains traits de langages, que nous n'abordons pas dans notre présentation (SIG76) (SIG77) (GRO79) (DER79).

En préambule de cette étude, nous rappelons sommairement quelques notions de base du domaine de la méthodologie de la programmation que nous utilisons par la suite.

I. RAPPELS DE QUELQUES NOTIONS DE BASE SUR LES LANGAGES ET LA PROGRAMMATION.

Bien avant de construire des programmes "structurés", on pratiquait ce qui était appelé la "programmation modulaire" (DER74) (PAR72), "modulaire" signifiait alors: "construit avec des unités standardisées ou d'une dimension physique qui favorise la flexibilité" (DEN73).

Pour résumer rapidement les buts visés par la programmation modulaire, nous paraphrasons DENNIS (DEN73) en disant:

.qu'elle devait "permettre au programmeur d'acquérir l'intime conviction que le module programme qu'il a écrit est correct, et ceci indépendamment du contexte d'utilisation",

.et qu'elle "facilitait l'intégration des différents modules --programmes-- écrits sous des autorités différentes".

Nous avons retenu cette définition générale de la programmation modulaire car elle nous paraît assez bien introduire les motivations qui ont, par la suite, conduit au développement de la programmation structurée. En effet, sous l'influence de DAHL (DAH72), DIJKSTRA (DIJ72), HOARE (HOA72), PARNAS (PAR72) et WIRTH (WIR71), les techniques de conception de programmes ("top-down", "bottom-up", machines virtuelles), la conception de langages de programmation qui supportent une méthodologie (pas de "GOTO"), l'axiomatisation des langages pour effectuer des preuves de correction des programmes constituent autant d'angles d'attaque de l'amélioration du premier objectif de la programmation modulaire énoncé par DENNIS. Quant à BAKER (BAK72), MILLS (MIL71), JACKSON (JAC75), ils se sont principalement occupés des techniques de développement de projets informatiques par "cellules de

programmation". Ce dernier aspect recouvre le deuxième objectif de la programmation modulaire.

I.1. La programmation structurée.

D'après HOARE (INF77), la programmation structurée est "la tâche qui consiste à organiser la pensée de façon telle que l'on aboutisse dans un délai raisonnable à l'expression compréhensible et correcte d'un programme".

Nous ne développons pas plus ici cette définition générale que nous avons extraite d'une étude de la programmation structurée dans (INF77). Nous ne détaillons pas les techniques de construction de programmes qui sont maintenant bien connues: la décomposition modulaire et le raffinement successif auxquelles nous ferons parfois référence.

I.2. La notion de type de données.

Il est bien difficile, compte tenu des nombreuses interprétations existantes, de formuler une définition de la notion de type de données.

Un type de données peut se définir par:

1-- la syntaxe : c'est alors une information associée à une variable au moment de sa déclaration,

2-- l'espace des valeurs : c'est simplement un ensemble de valeurs possibles (HOA72) (WEG74),

3-- le comportement : un type est un ensemble de valeurs et un

ensemble d'opérations applicables aux éléments de cet espace (NAU63),

4-- la représentation : un type est déterminé par la façon dont il a été représenté en terme de types plus primitifs,

5-- la représentation + le comportement : un type est alors défini par la représentation et l'ensemble des opérations qui en définissent le comportement (DAH70) (LIS74).

Illustrons rapidement les difficultés de ce genre de définition. Si nous prenons, par exemple, pour référence la définition d'un type de données telle qu'elle est proposée par WIRTH (Jensen et Wirth 1975 page 12): "un type de données définit l'ensemble des valeurs que peut prendre une variable". Il est alors tout à fait clair qu'avec les définitions suivantes, les types sont différents:

array [integer] of integer

et array [integer] of boolean

Par contre, est-il aussi évident de distinguer les types:

array [1..10] of integer

et array [1..11] of integer

PASCAL fait cette distinction alors que par exemple PL/1 ne la fait pas.

Cette discussion n'est pas seulement académique. Il nous semble tout de même important de savoir ce qu'est un type de données si l'on veut pouvoir étudier les mécanismes de contrôle de types de données.

I.3. La notion de module fonctionnel.

Le concept d'"encapsulement" d'une ressource et des opérations qui la partagent, introduit par PARNAS (PAR72), a beaucoup évolué depuis. On parle maintenant de modules fonctionnels (GOO78). Dans les langages modernes, un module fonctionnel est une unité de programme fermée définie par son comportement (DER79).

On distingue cinq catégories de modules fonctionnels:

1-- les modules qui fournissent une (des) fonction(s) pour la(les)quelle(s) il n'existe pas de dépendance entre deux ou plusieurs appels; par exemple, les "packages" de fonctions mathématiques,

2-- les modules qui fournissent une (des) fonction(s) pour la(les)quelle(s) il existe des dépendances entre les appels successifs, comme par exemple un générateur de nombres au hasard,

3-- les modules qui fournissent un ensemble d'opérations (procédures et fonctions) partageant une ressource commune. Le résultat des fonctions dépend des appels précédents des autres opérations, par exemple les modules de PARNAS (PAR72),

4-- les modules qui fournissent un ensemble d'opérations pour une classe d'objets; autrement dit, la classe d'objets utilise la même description de module; par exemple, les classes de SIMULA (DAH70),

5-- les modules qui fournissent une description d'un ensemble de déclarations: variables, types et actions; comme par exemple les plex en LIS (ICH75).

Dans la suite de l'exposé, nous nous intéressons principalement aux modules des catégories 3, 4 et 5. Notons que la définition des modules

de catégorie 4 est une autre façon de caractériser les types de données 5. Cette classification définit en fait des mécanismes nécessaires pour décomposer un programme en opérations et objets abstraits (on dit aussi logiques).

Les opérations abstraites, que B.LISKOV qualifie d'abstractions fonctionnelles (LIS74), peuvent ensuite être raffinées en définissant des procédures ou des fonctions dont les corps peuvent aussi être décomposés. Dans ce processus de construction de programmes, il importe de pouvoir aussi décomposer les structures de données complexes en objets abstraits dont le comportement logique soit compréhensible. Les objets abstraits que l'on appelle types abstraits de données peuvent par la suite être raffinés en définissant des modules de catégorie 4 qui, à leur tour, peuvent être décomposés. De la même façon que le mécanisme de procédure permet de cacher les variables locales et les détails de l'algorithme implémenté, les modules de catégorie 4 doivent cacher la structure interne des objets et les algorithmes des opérations applicables à ces objets.

Les procédures et les fonctions de catégorie 1 et 2 et les modules de catégorie 4 représentent des mécanismes de base de la programmtion structurée: les définitions des abstractions que l'on appelle interface abstrait étant constitués:

.pour les opérations: par le nom (et éventuellement les paramètres données et/ou résultats) de procédures et de fonctions,

.pour les objets: par le nom (avec éventuellement des paramètres) d'un module de catégorie 4 avec la liste des opérations applicables à ces objets.

Cette technique et ces mécanismes d'aide à la construction de

programmes structurés nécessitent que les interfaces abstraits (opérations et types) dépendent le "moins possible" des choix d'implémentation. C'est ce que ROSS appelle la propriété de référence uniforme (ROS70).

Le concept d'interface abstrait tel que nous venons de le préciser pose immédiatement le problème de l'expression des propriétés des abstractions. Pour comprendre, communiquer ou effectuer des preuves formelles, il devient nécessaire d'expliciter la sémantique de cette abstraction soit en langue naturelle, soit dans un langage formel. On parle alors de la spécification de l'interface abstrait ou plus simplement de spécification.

I.4. La protection dans les langages.

La notion de protection pour les systèmes de programmes est actuellement utilisée avec la même signification que pour les S.G.B.D.; seuls les moyens mis en oeuvre pour assurer l'intégrité et la sécurité des informations changent.

Dans ces rappels, nous nous limitons à l'exposé des mécanismes qui réalisent une protection par contrôle des droits d'accès aux objets décrits par des modules de catégorie 3 et 4. Plusieurs raisons peuvent justifier des contrôles d'accès aux informations dans un environnement de système logiciel d'application:

1-- les opérations qui manipulent un objet peuvent ne pas avoir les

mêmes droits d'accès vis à vis de l'objet,

2-- les utilisateurs ou classes d'utilisateurs n'ont pas obligatoirement les mêmes droits sur les objets ou classes d'objets,

3-- le contexte d'utilisation d'un programme peut conduire au contrôle des droits d'accès de ce programme vis à vis des objets.

La mise en oeuvre des mécanismes de protection est basé sur deux techniques:

-- une dynamique, qui conduit à des contrôles à l'exécution des programmes,

-- et l'autre statique, où les contrôles d'accès aux objets sont effectués à la compilation ou au plus tard à l'édition de liens (JON76).

Il est évident que la technique statique fournit une meilleure sécurité. Elle permet de "refouler" très tôt un programme avant même qu'il ait commencé à modifier l'environnement, s'il ne possède pas tous les pouvoirs requis. C'est aussi une technique efficace qui évite la propagation des erreurs.

I.4.1. Contrôles d'accès.

Nous nous intéressons maintenant aux mécanismes d'extension et de restriction des droits d'accès aux objets entre deux activations de procédures.

Extension des droits .

Ce mécanisme étudié par LAMPSON (LAM69) et FERRIE (FER75) consiste à modifier les droits d'une procédure appelée dont le droit est supérieur

à celui de la procédure appelante, pour que la procédure appelée puisse s'exécuter.

Nous ne détaillons pas les techniques dynamiques de mise en oeuvre de ce mécanisme qui ont été essentiellement réalisées pour les systèmes d'exploitation MULTICS (SCH72),PLESSEY 250 (ENG74) et HYDRA (WUL74). Une étude complète de ces techniques dynamiques de protection est exposée dans (MOR73).

Les techniques statiques de protection sont fondées sur l'utilisation de modules de catégorie 3 (PAR72) et de modules de catégorie 4 (types abstraits de données) (LIS74). La technique de contrôle statique d'accès à la ressource encapsulée dans un module de PARNAS repose sur la vérification par le compilateur des seules opérations associées visibles. On dit parfois que les opérations associées définissent des capacités d'accès à l'objet (LAM73).

Restriction des droits .

Ce mécanisme étudié par LAMPSON (LAM77) et JONES (JON76) consiste à contrôler l'effet que peut avoir une procédure appelée sur les objets de la procédure appelante. La technique de mise en oeuvre de ce mécanisme de restriction des droits procède en deux étapes:

.définition de l'ensemble des objets accessibles: soit dynamiquement comme dans MULTICS, PLESSEY et HYDRA en associant à chaque domaine un ensemble de capacités, soit statiquement en interdisant l'utilisation de variables globales et en imposant de déclarer les listes d'objets importés et exportés,

.restriction des droits sur un objet.

L'ensemble des objets accessibles étant défini, il suffit maintenant de contrôler que les accès aux objets passés en paramètre d'une

procédure appelée sont les seuls accès autorisés par la procédure appelante. Ce contrôle peut être dynamique (WUL74) ou statique (JON76) (BAN78) (DAR78).

Le confinement des programmes .

Le problème du confinement concerne la méfiance d'un utilisateur vis à vis d'un programme qu'il n'a pas écrit. L'utilisateur voudrait être sûr que le programme qui manipule ses données ne conserve pas d'informations confidentielles qui pourraient être divulguées ultérieurement.

Le confinement total d'un programme peut être contrôlé statiquement en vérifiant qu'il n'existe pas de communication avec l'extérieur. Autrement dit, les procédures du programme n'utilisent pas de variables globales et il n'y a pas d'opération d'entrée/sortie. Le confinement total impose des contraintes très fortes. Pratiquement, on se contente d'un confinement partiel qui permet par exemple la lecture de variables globales (MIN76).

La confidentialité .

La mise en oeuvre de la confidentialité doit permettre de contrôler qu'un programme manipulant des données confidentielles n'a pas la possibilité de les communiquer à l'extérieur du programme. On veut s'assurer qu'un programme qui calcule une moyenne de salaires ne peut pas délivrer le salaire d'un employé (JON75) (MIN76).

I.5. L'évolutivité.

L'évolutivité d'un système de programmes recouvre les notions d'extensibilité des langages de programmation et d'adaptabilité des programmes. Examinons rapidement ces deux notions:

L'extensibilité des langages de programmation .

ALGOL 68 (ALG68) a poussé très loin l'étude de cette notion d'extensibilité. Précisons qu'en ALGOL 68 un type s'appelle un mode. Ainsi, à partir d'un certain nombre de modes prédéfinis: int , real , char , bool ,etc... définis par une représentation et des opérations qui leurs sont applicables, l'utilisateur a la possibilité d'étendre cet ensemble à l'aide du mécanisme de constructeur de modes. Il existe cinq constructeurs: proc, row , struct , union et ref . Les nouveaux modes ainsi construits héritent des opérations caractéristiques du constructeur (ou de la composition des constructeurs). Conceptuellement, les modes construits possèdent les mêmes attributs que les modes prédéfinis: une représentation, une notation et un ensemble de fonctions d'accès. Les nouveaux modes utilisés dans les déclarations de variables, en paramètres de procédures sont contrôlés statiquement par le compilateur de la "même façon" que les modes prédéfinis.

De même, l'utilisateur a la possibilité d'étendre le "registre" des opérations du langage: affectation, if then else , etc... à l'aide du mécanisme de procédure. ALGOL 68 fournit pour cela des mécanismes puissants comme les routines, la surcharge ('overloading') d'opérations, etc.. Des études très détaillées d'ALGOL 68 sont exposées dans (ALG68).

Remarques :

1-- Il importe de noter que l'extensibilité en ALGOL 68 permet:

.de définir de nouveaux modes, mais certainement pas de nouveaux constructeurs de modes,

.et de nouvelles opérations utilisateurs mais pas de nouvelles structures de contrôle (if then else)

2-- Malgré la généralité des mécanismes d'extension, des contrôles statiques de modes fournits par ALGOL 68, ce langage n'offre pas de facilités d'abstraction sur les modes.

Un autre trait caractéristique de l'extensibilité concerne la paramétrisation des opérations abstraites et des types abstraits par des types (JAC78) (LIS74).

L'adaptabilité des systèmes de programmes .

Pour qu'un système de programmes soit viable, il est essentiel qu'il possède cette qualité parfois qualifiée de maintenance (DER79) (SWA76). D'ailleurs SWANSON distingue trois sortes de maintenance:

-- la maintenance perfective qui débute avant même que le programme soit complètement rédigé pour rendre compte des changements de spécification, des erreurs ou inefficacités qui sont détectées. Pour que ces modifications soient maîtrisables, il importe que les mécanismes de construction des programmes supportent l'encapsulement des choix de réalisation locaux, c'est-à-dire soient des modules fonctionnels. L'amélioration de l'efficacité des programmes peut par exemple nécessiter la substitution de l'implantation d'une abstraction par une autre, conformément aux spécifications de l'interface, voire même conduire à des multireprésentations de types abstraits à l'exécution,

-- la maintenance de récupération ('recovery') mise en oeuvre en cas

de défaillance du système d'exploitation. Elle consiste à reconfigurer l'environnement utilisateur dans un état cohérent sans perte sensible d'information,

-- la maintenance adaptative correspond au changement de support, au transcodage de fichiers de données.

II. QUELQUES LANGAGES EXISTANTS.

Nous présentons maintenant quelques langages de programmation existants qui fournissent à des degrés divers des facilités de définition de modules de catégories 3, 4 ou 5.

II.1. PL/1.

Lors de notre étude de la problématique de la construction de systèmes de programmes, nous avons implicitement laissé entendre que PL/1 ne fournissait pas de facilités de définition de types utilisateurs autrement que par les constructeurs de base. Ceci n'est pas tout à fait exact, comme l'illustre l'exemple suivant de définition du type pile

```
pile: PROC
        DCL 1 une-pile BASED(sep),
            2 der-elt-pile PTR,
            2 valent CHAR(1);
        DCL sep PTR ;
```

```
        DCL wp PTR ;
        DCL acc-descripteur-pile(CHAR(32), PTR ) RETURNS ( PTR );
        RETURN
creer-pile: ENTRY (noml-pile,init);
        DCL noml-pile CHAR(32),init CHAR(1);
        DCL 1 descripteur-pile BASED (sp),
           2 der-descripteur-pile PTR ,
           2 nom CHAR(32),
           2 point-sommet PTR ;
        DCL sp PTR;
        DCL som-descripteur-pile PTR STATIC INIT (NIL);
        wp=acc-descripteur-pile(noml-pile,som-descripteur-pile);
        IF wp ¬=NIL THEN SIGNAL error;
                              /*la pile existe déjà*/
        ALLOCATE descripteur-pile;
        sp-->der-descripteur-pile=som-descripteur-pile;
        som-descripteur-pile=sp;
        sp-->nom=noml-pile;
        ALLOCATE une-pile;
        sep-->value=init;
        sep-->der-elt-pile=NIL;
        sp-->point-sommet=sep;
        RETURN;
empiler: ENTRY (evaleur,nompile2);
        DCL evaleur CHAR(1),nompile2 CHAR(32);
        wp=acc-descripteur-pile(nompile2,som-descripteur-pile);
        IF wp=NIL THEN SIGNAL error;
```



```
                              /*la pile n'existe pas*/
        ALLOCATE une-pile;
        sep-->valeur=evaleur;
        sep-->der-elt-pile=wp-->point-sommet;
        wp-->point-sommet=sep;
        RETURN
depiler: ENTRY (nom3-pile);
        DCL nom3-pile CHAR(32);
        wp=acc-descripteur-pile(nom3-pile,som-descripteur-pile);
        IF wp=NIL THEN SIGNAL error;
        sep=wp-->point-sommet;
        wp-->point-sommet=der-elt-pile;
        FREE une-pile
sommet: ENTRY (nom4-pile) RETURNS (CHAR(1));
        DCL nom4-pile CHAR(32);
        DCL val-som CHAR(1);
        wp=acc-descripteur-pile(nom4-pile,som-descripteur-pile);
        IF wp=NIL THEN SIGNAL error;
        wp=wp-->point-sommet;
        IF wp=NIL THEN SIGNAL error;
        val-som=wp-->valeur;
        RETURN (val-som)
END pile;
acc-descripteur-pile: PROC (anom,q) RETURNS ( PTR );
        DCL anom CHAR(32);
        DCL (q,mp) PTR ;
        DCL echec BIT INIT ('1' B );
```

```
mp=q;
DO WHILE (echec ¢ mp  =NIL);
   IF mp-->descripteur-pile-nom=anom THEN echec='0' B ;
   ELSE mp=mp-->der-descripteur-pile;
END ;
RETURN (mp)
END acc-descripteur-pile;
```

Cet exemple nous semble intéressant car il illustre la puissance des concepts de procédure et de pointeur que l'on retrouve d'une façon plus ou moins implicite dans beaucoup d'autres langages de programmation. Ici, grâce à la construction d'une liste de descripteurs de piles, que l'on utilise pour garder les piles instanciées ainsi que les éléments de chaque pile, on peut manipuler effectivement plusieurs piles de caractères.

On peut par exemple, dans un programme PL/1, créer les piles suivantes:

```
CALL creer-pile('ma-pile','£');
CALL creer-pile('ta-pile','?');
```

et ensuite appliquer les opérations définies comme points d'entrée (empiler, dépiler, sommet) de la procédure pile. Par exemple, les appels suivants:

```
CALL empiler('ma-pile','1');
CALL empiler('ma-pile','5');
```

ont pour effet d'empiler dams ma-pile les caractères '1' puis '5'. Il est intéressant de remarquer que cette description de "pile abstraite" est une forme de définition d'un module fonctionnel de catégorie 4 ou encore d'un type de données défini par une représentation et par le

comportement des opérations associées. Maintenant, il faut aussi remarquer que ces abstractions de types de données ne sont pas manipulées comme les types prédéfinis: pas de déclaration de variables et surtout pas de contrôles de types possibles par le compilateur.

Le fait que PL/1, développé depuis 1960 par IBM, soit maintenant disponible sur de nombreuses installations, a conduit par exemple ZELKOWITZ (ZEL78) à étudier des extensions du compilateur PL/1 pour implanter des types abstraits. Ces propositions consistent à "envelopper" le mécanisme de pointeurs décrit précédemment à l'aide du concept d'environnement de PL/1. Cette notion utilisée pour décrire les attributs d'une variable fichier est étendue aux types utilisateurs. En reprenant l'exemple de pile précédent, cette extension permet de déclarer une variable ma-pile de type PILE avec une liste de droits associée à la variable: empiler, dépiler qui représentent les points d'entrée de la procédure pile:

```
DCL ma-pile ENVIRONMENT (pile empiler, dépiler)
```

Comme le suggère ZELKOWITZ, si le compilateur dont on dispose n'est pas aussi facilement modifiable que celui de l'université de Maryland, on peut toujours envisager l'utilisation d'un préprocesseur!

II.2. Le langage PASCAL.

Lorsque N.WIRTH a conçu PASCAL en 1969, son objectif était de fournir un langage adapté à l'enseignement de la construction systématique de programmes, qui puisse être implanté comme un système de programmation fiable et efficace. Ce langage fournit des facilités de structuration des données: structures, fichiers, ensembles, pointeurs. La sémantique de PASCAL a été définie axiomatiquement (HOA72) pour faciliter la vérification des programmes. Les seules possibilités d'abstraction de PASCAL concernent les fonctions et les procédures. Pour illustrer les traits de ce langage, nous décrivons l'exemple simple de la pile.

```
pile= record corps: array [1..100] of char ;
   /*définition du type pile*/
        sommet: integer ;
procédure empiler(y: char , var x:pile);
begin
        x.sommet:=x.sommet+1;
        if x.sommet>100 then error;
        x.corps[x.sommet]:=y
end ;
procédure dépiler( var x:pile);
begin
        if x.sommet<1 then error;
        x.sommet:=x.sommet-1
end ;
function elem-sommet(x:pile): char ;
```



```
begin
        if x.sommet>100 then error;
        if x.sommet<1 then elem-sommet:='£';
           else elem-sommet:=x.corps[x.sommet]
end ;
procedure init(y: char ; var x:pile);
begin
        x.corps[1]:=y;
        x.sommet:=1
end ;
```

La déclaration d'une variable de type pile se fait au début d'un programme

```
var ma-pile,ta-pile:pile
```

Ici, la définition du type pile est une forme de module de catégorie 5. Le mécanisme de sélection de la structure record utilise le préfixage des sélecteurs par le nom de la variable

ma-pile.sommet et ma-pile.corps[i] avec $1 \leq i \leq 100$

Cette description, contrairement à celle que nous avons donnée pour PL/1, est à la fois compréhensible et facile à écrire. On peut toutefois émettre quelques réserves -- la définition du type pile n'est pas liée syntaxiquement aux "routines" associées pour manipuler les variables déclarées du type pile. Conceptuellement, c'est gênant.

Par contre, ce qui est plus grave, c'est que la représentation d'une variable de type pile puisse être modifiée par d'autres opérations que les routines "associées" au type pile. Enfin, il n'est pas possible d'utiliser le même nom d'opérateur pour deux types de données différents.

Remarque :

Comme le note HABERMANN (HAB73), il est vrai que le mécanisme de définition de type de PASCAL est ambigu. Il n'existe pas de véritable distinction entre le concept de type et le concept de structure de données: "un type définit un domaine pour les objets déclarés de ce type et détermine les opérations qui peuvent être appliquées à ces objets" alors que "une structure définit une règle de connexion des objets dans des unités plus larges, mais les opérations ne portent pas sur la structure, elles s'expriment en termes d'éléments individuels de la structure", ce que nous résumons en disant simplement que PASCAL ne fournit pas de mécanisme d'abstraction de types de données.

II.3. Le langage SIMULA (DAH70).

Développé en 1967 par O.J.DAHL, D.MUHRARRG et K.HYGAARD, SIMULA est un langage conçu principalement pour faire de la simulation. Nous ne nous intéressons, ici, qu'aux traits généraux de ce langage. Il fournit des possibilités puissantes d'abstraction par le mécanisme de classes. On peut concaténer des classes, ce qui permet de construire des hiérarchies de structures de données. Comme ALGOL 60, SIMULA utilise la structure de blocs. Les classes sont des blocs dans lesquels les variables rémanentes sont déclarées au plus haut niveau. Plusieurs instances d'une classe peuvent exister simultanément à l'exécution. Il est aussi possible de passer les paramètres par référence, par valeur ou par nom.

Pour illustrer les possibilités d'abstraction de SIMULA 67,

considérons la déclaration de la classe pile suivante:

```
CLASS pile(n); integer (n);
begin char array corps[1:n];
   integer sommet;
   procedure empiler(elt); char elt;
   begin;
      sommet:=sommet+1;
      if sommet>n then erreur
            else corps[sommet]:=elt;
      end empiler;
   procedure depiler;
   begin ;
      if sommet<1 then erreur
            else sommet:=sommet-1;
   end depiler;
   char procedure elt-sommet;
   begin ;
      if sommet>100 then erreur;
      if sommet<1 then elt-sommet:='£'
            else elt-sommet:=corps[sommet];
   end elt-sommet;
   procedure init(y); char (y);
   begin ;
      sommet:=1;
      corps[1]:=y;
   end init;
   sommet:=0;
```

end pile

Ici, n représente la taille de la pile.

La représentation de la pile de taille n comporte:

-- la variable corps: tableau de taille n d'éléments de type char,

-- et la variable sommet matérialisant l'index sommet de la pile dans le tableau. Cette variable est initialisée à 0.

Les routines de l'abstraction pile sont: empiler, depiler, elt-sommet et init. Pour instancier un objet, on dit en SIMULA générer un objet, on commence par déclarer une variable pointeur ma-pile

ref (pile) ma-pile

Un objet est alors créé à l'aide de l'opération new

ma-pile:- new pile(100)

L'opérateur new a pour effet de créer un objet de la classe pile et de délivrer une référence à cet objet. L'opérateur ':-' affecte ensuite cette référence à la variable référence ma-pile. Les routines associées à la classe pile: empiler, depiler, elt-sommet, init sont les opérations de manipulation de l'objet ma-pile. Toutefois, en SIMULA, il est aussi possible d'accéder directement à la représentation de l'objet de nom ma-pile en préfixant les variables locales de la classe pile par ce nom. Cette possibilité, longtemps reprochée à SIMULA, est maintenant contrôlée par les attributs "hidden" et "protected".

Comme le montre J.F.DUFOURD (DUF80), l'utilisation du mécanisme de sous-classes et de l'attribut virtual fournit des possibilités supplémentaires d'abstraction de données. B.MEYER a aussi suggéré, lors de sa présentation de SIMULA aux journées GROPLAN de Cargèse (GRO79), de prolonger ces facilités en introduisant des déclarations de pointeurs pour construire des types génériques (JAC78). En fait, il

nous semble, à l'analyse de ces quelques propositions, que si le mécanisme de pointeurs permet effectivement l'élaboration de nouvelles abstractions, leur sophistication est telle que nous doutons de leur utilité pratique. En ce sens, nous adhérons totalement aux remarques 5-1, 5-2 et 5-3 page 378 de B.MEYER dans (GRO79).

Enfin, notons qu'en SIMULA, il n'existe pas de structure de contrôle d'itération de haut niveau et que les possibilités des fichiers sont assez restreintes.

II.4. LIS (ICH75).

LIS (Langage d'Implantation de Systèmes) conçu en 1972 par J.D.ICHBIAH, J.P.RISSEN, J.C.HELIARD peut être caractérisé comme suit:

-- C'est un langage de haut niveau du type PASCAL ou ALGOL 68 qui fournit des facilités de structuration des données et des actions, et des structures de contrôle de programmes de haut niveau.

-- La compilation séparée d'unités segments programmes et segments données permet des modifications locales dans un programme sans avoir de gros changements dans le reste du programme. Le compilateur effectue des contrôles de types et vérifie en plus la compatibilité des unités compilées séparément.

-- La notion d'implémantation de LIS permet aussi au programmeur d'agir sur le compilateur pour éventuellement modifier des choix de représentation de données.

Pour illustrer la syntaxe de LIS, nous reprenons le même exemple de type pile:

```
data segment main ;
   erreur: constant integer == integer @ last ;
   max: constant integer == 100
type pile-d-entier=
   plex
      taille: constant (1..max)
      rep: private array (1..taille) of integer ;
      sp: private (0..max)
      empiler: action (element: integer )
      depiler: action
      sommet: integer action ;
      vide: action
   end ;
end ;
 use data main;
program segment main
   program pile-d-entier.empiler;
   begin
      if sp/=taille then
      sp:=sp+1;
      rep(sp):=element;
      end ;
   end ;
   program pile-d-entier.depiler;
      begin
      if sp/=(rep. first -1) then
      sp:=sp-1;
```



```
      end ;
   end ;
   program pile-d-entier.sommet;
   begin
      if sp/=(rep. first -1) then
      return rep(sp);
      else
      return erreur;
   end ;
   program pile-d-entier.vide;
   begin
      sp:=rep. first -1;
   end ;
end ;
```

Ici, la définition d'une pile-d-entier est décrite dans deux unités compilées séparément: data segment et program segment. L'unité data-segment main représente l'interface abstrait du type pile-d-entier.

Dans l'unité data-segment main, l'interface de la pile-d-entier est décrite avec le constructeur de type plex . Cette définition du type pile-d-entier comportant une liste de déclarations: de constantes (taille), de variables (rep, sp) et d'actions (empiler, depiler, sommet, vide) est un module de catégorie 5. L'attribut constant de la variable taille permet de vérifier qu'après l'affectation initiale (1..max), cette valeur n'est plus modifiée. L'attribut private des variables rep et sp définit la protection de la représentation du type pile-d-entier contre tout accès direct. Par exemple, si l'on a déclaré

un objet ma-pile de taille 100 dans un data-segment:

```
ma-pile:pile-d-entier == (taille == 100);
```

il est interdit d'écrire l'instruction suivante qui modifie directement la représentation:

```
ma-pile.rep(3):= 101
```

Seules les opérations déclarées comme des actions sont autorisées comme:

```
ma-pile.empiler(1)
```

Ce langage orienté système fournit par le biais de ce mécanisme de plex avec protection d'accès direct à certains attributs, un outil plus puissant que le système de programmation CIVA (DER74). Le système de segmentation inspiré de SUE (HOR77) permet de limiter les recompilations d'unités en cas de modification. Cependant, le choix d'ICHBIAH de lier la déclaration de la représentation à l'interface ne nous semble pas satisfaisant. Conceptuellement, le comportement d'une pile d'entiers implémentée par un tableau ou une liste doit être le même. Par conséquent, la dépendance de l'abstraction vis à vis de la structure interne des objets est regrettable même si on peut justifier ce choix par la sécurité de l'implémentation (pas de pointeurs). En outre, toute modification de la structure interne implique une recompilation de toutes les unités qui déclarent des variables du type; la réservation des places étant faite à la compilation. Nous reviendrons sur cet aspect à propos d'ADA (ADA80) qui reprend ce même concept.

Enfin, LIS ne fournit pas de structures de contrôle d'itération de très haut niveau et les possibilités d'utilisation des fichiers sont très restreintes.

II.5. Remarques.

Dans cet exposé rapide de langages de programmation existants, nous nous sommes limités à ceux que nous avons pu expérimenter sur l'IRIS 80 de l'IUCAL de Nancy. Il est bien évident que nous ne leur attribuons pas plus de qualités qu'aux autres langages existants comme ALGOL 68 (ALG68), EL/1 (WEG74), SETL (SCH75). En résumé, parmi les langages existants, seuls SIMULA et LIS fournissent des facilités d'abstraction de types de données. La possibilité en LIS de retarder le codage des opérations associées à l'interface constitue une amélioration sensible par rapport à SIMULA.

III. QUELQUES APPROCHES DE LANGAGES RECENTS.

Dans cette partie, nous nous intéressons principalement à deux catégories de langages: les langages orientés vérification EUCLID, ALPHARD, et orientés système CLU, ADA.

III.1. EUCLID (LAM77).

EUCLID conçu en 1977 par B,W.LAMPSON, J.J.HORNING, R.L.LONDON, J.G.MITCHELL et G.J.POPEK est un langage de programmation orienté vers la vérification de programmes systèmes. Basé sur PASCAL, EUCLID fournit des mécanismes pour contrôler la portée et la visibilité des identificateurs utilisés dans des structures de blocs.

Un module EUCLID peut être un module fonctionnel de catégorie 4 où la représentation des objets abstraits et l'implantation des opérations associées peuvent être cachées. Les objets, opérations et types définis à l'intérieur d'un module doivent être explicitement exportés pour être utilisés à l'extérieur (exports). De même, les valeurs de variables déclarées à l'extérieur d'un module doivent être explicitement importées (imports). En EUCLID, le mécanisme de module offre deux possibilités de définition de type abstrait de données. Reprenons le type pile d'entiers:

```
type pile (taille:unsignedint) = module
   exports (empiler, depiler, sommet)
   var pile-d-entier : array 1..taille of signedint
   var sp : 0..taille:=0
procedure empiler(X:signedint)=
   imports ( var pile-d-entier, var sp,taille)
   begin
   procedure overflow=.... end overflow
   if sp = taille then overflow
   else sp:=sp+1
      pile-d-entier(sp):=x
```



```
   endif
   end empiler
procedure depiler=
   imports ( var pile-d-entier, var sp)
   begin
   procedure underflow=... end underflow
   if sp = 0 then underflow
      else sp = sp-1
   endif
   end depiler
function sommet returns signedint=
   imports ( var pile-d-entier, var sp)
   return pile-d-entier(sp)
   end sommet
```

Nous ne détaillons pas la signification des instructions dont la syntaxe est tout à fait similaire à PASCAL.

La déclaration d'une pile ma-pile et son utilisation dans un programme s'écrira:

```
   var ma-pile: pile(100)
   var element: signedint
   .
   ma-pile.empiler(3);
   element:=ma-pile.sommet;
```

En EUCLID, les fonctions sont sans effet de bord, ce qui interdit par exemple de récupérer la valeur dépilée par l'opérateur dépiler.

La deuxième manière de définir le type pile est la suivante:

```
type pile = module
```

```
exports (pile-d-entier,empiler,depiler,sommet)
type pile-d-entier (taille:unsignedint)= record
   var sp:0..taille := 0
   var rep: array 1..taille of signedint
end pile-d-entier
procedure empiler( var vpile:pile-d-entier ( parameter),
                X:signedint)=
   begin
   procedure overflow=... end overflow
   if vpile.sp := vpile.taille then overflow
   else vpile.sp := vpile.sp + 1
      vpile.sep(vpile.sp) := X
   endif
   end empiler
   .
   .
end pile
```

Ici pour manipuler l'objet ma-pile de type pile de taille 100, on écrit:

```
(1) var mpile:pile
/*instanciation d'un exemplaire de module pile*/
(2) var ma-pile:mpile.pile-d-entier(100)
/*instanciation d'un objet*/
var element:signedint
.
mpile.empiler(ma-pile,3)
```

Notons que l'attribut ' parameter ' signifie simplement que la

désignation du type est différée jusqu'à l'instanciation de la ligne (2). Il n'est pas possible de définir de types paramétrés en EUCLID.

Contrairement à d'autres langages que nous exposons par la suite (ALPHARD, CLU), EUCLID ne fournit pas de mécanisme adéquat pour séparer l'interface de l'implantation; sinon recopier les textes des interfaces dans chaque module supposé implantant la même abstraction.

Enfin, les opérations définies sur un type de données utilisateur ne peuvent être que des routines et non de nouveaux opérateurs infixés ou préfixés.

Les structures de contrôle sont tout à fait similaires à celles de PASCAL. Pour vérifier des programmes EUCLID, la technique des assertions est la même que celle déjà utilisée pour PASCAL.

III.2. ALPHARD (WUL75).

Conçu en 1975 par W.F.WULF, R.L LONDON, M.SHAW, le langage de programmation ALPHARD fournit des mécanismes d'abstraction qui favorisent la construction de programmes structurés et leur vérification formelle.

III.2.1. La notion de form.

Le mécanisme de base utilisé pour décrire l'encapsulement de l'abstraction de type de données dans une unité syntaxique est la form. La définition d'une form comporte quatre niveaux que nous illustrons sur l'exemple classique de l'abstraction pile d'entiers:

```
form ipile(n:integer)
beginform
specifications
   requires n>0
   let ipile=<--x_i---> where x_i is integer;
   invariant 0 ≤ long(ipile) ≤ n;
   initially ipile=nullseq;
   function
   empiler(s:ipile,x:integer) pre 0 ≤ long(s)<n
                post s=s^x,
   depiler(s:ipile) pre 0<long(s) ≤ n
                post s=leader(s'),
   sommet(s:ipile) returns x:integer
      pre 0<long(s) ≤ n
      post x=last(s'),
   vide(s:ipile) returns b:boolean
                post b=(s=nullseq);
representation
   unique v:vector(integer,1,n),sp:integer init sp<-0;
   rep (v;sp) = seq(v,1,sp);
   invariant 0 ≤ sp ≤ n;
```



```
states
   mt when sp=0,
   normal when 0<sp<n,
   plein when sp=n,
   erreur otherwise;
implementation
   body empiler out (s.sp=s.sp'+1 ∧ s.v=∝(s.v',s.sp,x))=
      mt,normal::(s.sp<-s.sp+1;s.v[s.sp]<-x);
      otherwise ::FAIL;
   body depiler out (s.sp=s.sp'-1)=
      normal,plein::s.sp<-s.sp-1;
      otherwise ::FAIL;
   body sommet out (x=s.v[s.sp])=
      normal,plein::x<-s.v[s.sp];
   body vide out (b=(sp=0))=
      normal,plein::b<-false
      mt::b<-true;
      otherwise ::FAIL
endform ;
```

Explicitons cette définition de form ipile, par niveaux de conception:

-- le premier niveau (form) comporte le nom de l'abstraction ipile avec en paramètre, la profondeur de la pile d'entiers. En fait, ipile est un générateur de pile. Une déclaration dans un programme local ma-pile:ipile(100); a pour effet de créer une instance d'objet liée par la déclaration à l'identificateur ma-pile,

-- le deuxième niveau (specification) décrit les profils

syntaxiques des opérations associées à la classe d'objets abstraits: nom de l'opération, paramètres données, paramètres résultats. Ensuite, pour permettre les vérifications on définit le domaine abstrait, la valeur initiale de chaque type abstrait et les pre post conditions de chaque opération. Les propriétés de ipile sont exprimées en termes de séquence de longueur finie,

-- le troisième niveau (representation) définit la structure de données qui sera utilisée à chaque instanciation de la form . On déclare que cette représentation doit être unique à chaque instanciation. Pour le partage, il faut spécifier common . La clause rep spécifie la fonction de représentation des objets concrets en termes d'objets abstraits. Enfin, on définit un ensemble d'états caractéristiques des objets,

-- le quatrième niveau (implementation): on définit chaque opération associée sur la représentation concrète, en explicitant les pre post conditions concrètes.

Le niveau specification de la form ipile est la seule partie visible de l'extérieur. Cette description en plusieurs niveaux présente l'important avantage de permettre une preuve séparée de l'implantation d'une abstraction et du programme abstrait qui l'utilise (WUL78). Nous ne détaillons pas ici la vérification de la cohérence entre les spécifications abstraites et concrètes basée sur la méthode des assertions (FLO67).

III.2.2. Générateurs et itérations.

Un générateur est en fait une form (ou un module) qui fournit des fonctions spéciales et traite une séquence de liaisons de la variable de contrôle d'une itération. Deux types d'itérations sont fournies:

--l'itération for qui permet de parcourir toute la structure de donnée de la form définie comme générateur gen.

En ALPHARD, la syntaxe de cette instruction est:

for x:gen(y) [while β (x)] do ST(x,y,z)

où β(x) est une expression conditionnelle,

ST(x,y,z) est le corps de la boucle,

x est l'instanciation du générateur gen,

y est l'ensemble correspondant à l'instanciation des paramètres du générateur,

z est une variable libre,

[ ] signifie facultatif.

Pour illustrer cette instruction, considérons le générateur simplifié suivant:

```
form upto (b_i,b_s:integer) extends k:integer=
beginform
specification
   inherits < allbut <- >;
   function
      mt(u:upto) returns (b:boolean)
      next(u:upto) returns (b:boolean)
   .
   .
```

implementation

body ¢mt=(u.k<-u.b_i;b<-u.b_i ≤ u.b_s);

body ¢next=(u.k<-u.k+1;b<-u.k ≤ u.b_s);

endform

Notons que cette form étend les entiers, mais pour s'assurer que l'utilisateur ne change pas la variable de contrôle dans la boucle, l'instruction d'affectation est interdite < allbut <- >.

Pour décrire la somme des n premiers nombres entiers, on écrit:

somme<-0; for c:upto(1,n) do somme<-somme + c

Pour cette séquence, l'expansion du compilateur s'écrit avec la sémantique ALPHARD:

somme<-0;

begin

local c :upto(1,n);

c<-c.b_i; /* ¢init*/

while c ≤ c.b_s do (somme<-somme+c;c<-c+1); /* ¢next*/

end

-- l'itération first permet de rechercher dans une séquence de valeurs la première qui satisfait à une condition (x). La syntaxe de cette instruction est:

first x:gen(y) suchthat β(x) then S_1(x,y,z) else S_2(y,z)

où S_1,S_2 sont des instructions,

x est l'instanciation du générateur.

Pour illustrer cette syntaxe, considérons la spécification de la fonction eq-vect qui compare deux vecteurs de types et d'ensembles d'index arbitraires:

function eq-vect(A,B:vect(?t<£>,?b_i,?b_s)) returns (eq:boolean)=

first c : upto(b_i,b_s)

suchthat A(c)≠B(c)

then eq<- false

else eq<- true

end ;

La notation < > est une extension de la notion classique de contrôle de type. Explicitons la signification de cette itération first par l'expansion que le compilateur pourrait produire (à quelques optimisations près):

begin label λ;

begin local c:upto(b_i,b_s);

c<-c.b_i; /* ¢ init*/

if A c ≠B c then eq<-false; goto

else c<-c+1 /*¢ next*/

end ;

eq<-true;

λ. end

L'identificateur λ ci-dessus est généré par le compilateur. L'analyse des expressions décrites ci-dessus montrent qu'à la compilation des itérations for et first , une séquence upto(b_i,b_s) est instanciée puis liée à la variable de contrôle de boucle c. Après l'initialisation ¢init, l'itération dans les deux cas utilise la même stratégie de parcours: celle d'une séquence avec ¢next. En ALPHARD, comme nous le voyons, les générateurs sont implantés par des séquences, dans l'unique but de faciliter les preuves.

III.2.3. Protection et contrôle d'accès.

ALPHARD fournit des facilités de protection basées sur la notion de capacités . Dans l'exemple précédent de la fonction de comparaison de deux vecteurs, nous avons utilisé avec le paramètre formel, la notation syntaxique <≠>. Cette notation définit une restriction sur le paramètre effectif qui est vérifiée à la compilation. En fait, les spécifications d'une form définissent un ensemble d'accès aux objets du type définis par la form. la notation < > représente à la fois les droits d'accès requis sur le paramètre et nécessaires pour le corps de la fonction. Dans un programme, lorsque le paramètre effectif est lié au paramètre formel, alors le paramètre formel représente le nom d'un pouvoir appelé aussi capacité. Ce mécanisme de protection est étudié de façon très complète dans (JON76).

III.3. CLU (LIS74).

Développé depuis 1974 au M.I.T. par B.LISKOV et S.N.ZILLES, CLU est un système de programmation conçu pour supporter des mécanismes d'abstractions fonctionnelles, de types de données et de structures de contrôle des itérations.

Le mécanisme de base d'abstraction de types de données est le cluster : c'est un module de catégorie 4. Etudions, toujours sur l'exemple de la pile d'entiers les principaux traits de cette abstraction.

III.3.1. Notion de cluster.

```
pile: cluster (element-type: type )
   is creater,empiler,depiler,sommet,êtêtre,vide;
   rep (type-param: type )=(tp: integer;
                            e-type: type ;
                            vpile: array[1..] of type-param);
create: operation () returns ( cvt )
   s: rep (element-type);
   s.tp:=0;
   s.e-type:=element-type;
   return (s);
   end
empiler: operation (s: cvt ,v:s.e-type);
   s.tp:=s.tp+1;
   s.vpile[s.tp]:=v;
   return ;
   end
depiler: operation (s: cvt ) returns (s.e-type);
   if s.tp=0 then error ;
   s.tp:=s.tp-1;
   return (s.vpile[s.tp+1]);
   end
sommet: operation (s: cvt ) returns (s.e-type);
   if s.tp=0 then error ;
   return (s.vpile[s.tp]);
   end
```

```
étêter: operation (s: cvt );
   if s.tp:=s.tp-1;
   s.tp:=s.tp-1;
   return
   end
vide: operation (s: cvt ) returns ( boolean );
   return s.tp=0;
   end
end pile
```

Dans cette définition de cluster pile, on distingue quatre parties:

-- l'interface abstrait :

```
pile: cluster (element-type : type )
   is xcreate,empiler,depiler,sommet,étêter,vide
```

Cette abstraction est paramétrée par le type des éléments (element-type): c'est un générateur de type pile. Cette description sommaire de l'interface qui comporte donc: le nom de l'abstraction (pile), la liste des paramètres (element-type) et la liste des opérations associées, est la seule partie visible de l'extérieur. Le système de programmation CLU permet de cataloguer dans une bibliothèque des interfaces les renseignements de l'interface nécessaires aux contrôles de types ultérieurs. Pour être déclarée dans un programme abstrait, il faut déclarer une variable:

```
ma-pile:pile(entier)
```

Cette déclaration, contrairement à ALPHARD, ne produit pas d'instanciation d'objet. En fait, cette déclaration, bien que cela ne soit jamais dit explicitement dans les articles, est une "déclaration cachée" de pointeur qui s'écrirait en SIMULA (ref (pile(entier):pile).

-- la représentation :

La description rep (type-param:type) définit un nouveau type seulement accessible de l'intérieur du cluster. Cette définition de type utilise les constructeurs de PASCAL: record, array (avec index entiers). La représentation peut aussi être paramétrée: c'est ce qu'indique le paramètre (type-param).

-- l'opération de création : définie par le mot réservé create.

La création d'un objet ne devient effective qu'à l'appel de l'opération create :ma-pile:=pile$create()

L'activation de cette opération réserve de la place pour la représentation et délivre une référence à cet emplacement. Cette référence est ensuite affectée à la variable "référence" ma-pile.

-- les opérations associées du cluster pile:

Ces opérations sont des procédures et des fonctions classiques qui peuvent accéder à la représentation si elles possèdent un paramète rep.

On peut faire deux remarques sur la définition de ces opérations:

. elles possèdent toutes au moins un paramètre de type cvt (ou rep). Autrement dit, le cluster définit un modèle de capacités pour tous les objets abstraits du type pile.

. ' cvt ' spécifie la coercition implicite du type abstrait pile en rep, au moment du passage de paramètre. C'est ce qui fait de CLU un langage fortement typé.

III.3.2. La liaison de l'interface abstrait à l'implantation.

Le système de programmation CLU comporte une bibliothèque offrant quelques facilités pour développer des programmes de manière incrémentale. Cette bibliothèque permet en effet de cataloguer des unités de description d'abstractions et des modules d'implantation, séparément.

Ces unités de description d'abstractions contiennent toutes les informations nécessaires sur les abstractions de type de données, de contrôles et fonctionnelles (LIS77) pour permettre des contrôles de type. Un module peut par exemple être compilé avant un interface abstrait, ce qui permet d'ailleurs de définir plusieurs implémentations pour une même abstraction.

La correspondance d'une implantation à une abstraction est décrite avec un langage de connexion. Cependant, la compilation d'un module nécessite qu'une liste de noms externes soit fournie avec le code source pour faire des vérifications de conformité. En d'autres termes, si CLU permet effectivement de compiler séparément des unités de description d'abstraction et d'implémentation, le processus de compilation impose une liaison permanente d'un module d'implantation à un interface abstrait.

III.3.3. Les itérateurs.

CLU fournit un mécanisme d'abstraction (iterator) de la stratégie d'accès à des objets traités dans les itérations (for). Cette facilité permet de décomposer une itération en une partie algorithme d'accès aux objets et en une autre partie traitement d'objets. Un itérateur (iter) est une opération particulière qui peut être définie dans un cluster.

La syntaxe d'une itération CLU est définie par:

```
for déclarations in invocation d'itérateur do
    corps
end ;
```

L'itération et l'itérateur opèrent comme des coroutines. L'itérateur fournit un objet à la fois consommé par l'itération qui redonne le contrôle à l'itération. Pour illustrer la définition et l'utilisation de ce mécanisme, considérons l'exemple simple de la manipulation d'une chaîne de caractères. On définit l'itérateur de la façon suivante:

```
chaine-car= iter (s: string ) yields ( char );
   index: int :=1;
   limit: int :=string$size(s);
   while index<=limit do
      yield string$fetch(s,index);
      index:=index+1;
   end;
end chaine-car
```

L'itération rédigée dans un programme abstrait serait:

```
s: string ; count: int :=0;
```

```
for c: char in chaine-car(s) do
   if char-co-numeric(c)
      then count:=count+1;
   end
end
```

Précisons que dans l'itérateur, l'opération size(s) délivre la taille de la chaîne de caractères s et que fetch(s,n) délivre le nième caractère de la chaîne s. On voit que l'itérateur produit les caractères de la chaîne de caractères s dans l'ordre où ils apparaissent.

Cet itérateur est activé par l'itération for qui lui passe l'argument s.

Lorsqu'une instruction yield est rencontrée, l'itérateur fournit un caractère à l'itération qui le consomme après affectation à la variable locale c de l'itération. L'itérateur reprend ensuite le contrôle après l'instruction yield avec le dernier contexte d'activation. L'itération s'arrête lorsque l'itérateur ne produit plus d'objet, c'est-à-dire à l'exécution d'un return (implicite ou explicite).

D'autres exemples très détaillés sont exposés dans (LIS77).

III.4. ADA (ADA80).

ADA est un langage de programmation conçu en 1978 pour le DOD avec pour objectif de faciliter le développement d'applications numériques, de systèmes logiciels d'applications dans un environnement temps réel.

Dans cette partie, nous nous intéressons uniquement aux packages et aux packages génériques qui fournissent les mécanismes d'abstraction de données d'ADA.

III.4.1. La notion de package.

Pour illustrer cette notion de package, décrivons l'abstraction pile-d-entiers:

```
package ALL-ABOUT-STACKS is
   restricted type pile is private ;
   procedure empiler(e: in integer ;s: in out pile);
   procedure dépiler(e: out integer ;s: in out pile);
   private
      type pile is
      record
         sp: integer range 0..1000:=0
         rep: array(1..1000) of integer;
      end record ;
   end ;
   package body ALL-ABOUT-STACKS is
   procedure empiler(e: in integer ;s: in out pile) is
```

```
      begin
         sp:=sp+1;
         rep(sp):=e;
      end
      procedure dépiler(e: in integer ;s: in out pile) is
      begin
         e:=rep(sp);
         sp:=sp-1;
      end
   end
```

Cet exemple montre que par rapport à LIS (ICH75), ADA n'apporte pas beaucoup d'innovations concernant l'abstraction de types de données. L'attribut restricted du type pile spécifie l'interdiction d'utiliser les opérations d'égalité et d'affectation sur les variables déclarées du type pile. L'attribut private a, à peu de choses près, la même signification qu'en LIS. ADA établit une distinction entre l'interface logique du package constitué de toute la partie visible du package et l'interface physique qui englobe les parties visibles et privées du package.

L'interface logique correspond à ce que nous avons appelé jusque là l'interface abstrait de l'abstraction. La partie physique contient toutes les informations nécessaires au compilateur pour créer une instance d'objet du type privé dans l'environnement de l'utilisateur. L'utilisation d'un package dans un programme serait:

```
   declare
      use ALL-ABOUT-STACKS;
      ma-pile:pile;
   begin
      empiler(ma-pile,3)
      .
      .
   end
```


Sur cette notion de package, nous formulons les mêmes remarques que pour LIS, en ce qui concerne la partie private.

III.4.2. Les packages génériques.

Le package défini précédemment peut être paramétré. L'unité de description s'appelle un package générique.

```
generic (size: integer ; type element)
package pile is
   procedure empiler( in elem: in element);
   procedure dépiler( out elem: out elememt);
   end pile
```

Une instanciation de package générique s'écrit:

```
package pile-d-entiers is new pile(100,integer)
```

En ADA, un package générique est une macro-définition qui ne peut donc être instanciée que par macro-génération. Les contrôles syntaxiques de types n'interviennent qu'au moment de la compilation de l'expansion obtenue par instanciation du package générique. Ce mécanisme n'est pas très satisfaisant, car les erreurs syntaxiques en particulier, ne sont détectées qu'à la compilation de l'instanciation,

ce qui complique beaucoup la recherche des erreurs.

IV. CONCLUSIONS.

Dans ce chapitre, nous avons essentiellement mis l'accent sur l'étude des mécanismes d'abstraction, de protection et d'évolutivité fournis par des langages de programmation anciens et de conception récente. Le but visé dans cette présentation a été plus de faire le point de l'"état de l'art" sur ces mécanismes qu'un panorama plus ou moins exhaustif. On peut en effet trouver dans la littérature des exposés sur d'autres langages dont nous n'avons pas parlé ici (GRO79) (HAN79). On a vu que dans les langages existants, seuls SIMULA et LIS fournissaient des mécanismes d'abstraction de données, de protection mais cependant pas de possibilités concernant l'évolutivité (représentation unique des abstractions, pas de paramétrisation des unités d'abstraction).

Maintenant, en ce qui concerne les nouvelles propositions, nous distinguons trois orientations:

-- les nouveaux langages du genre EUCLID ou ADA qui proposent des extensions de langages de programmation existants PASCAL et LIS pour en faire des systèmes de programmation adaptés à la multiprogrammation ou au temps réel. Citons aussi MODULA (WIR76).

-- les langages de type CLU ou MESA (GES77) influencés par des langages existants SIMULA et PASCAL; ils introduisent de nouveaux mécanismes d'abstraction: l'abstraction de contrôle d'itération, la paramétrisation.

-- les langages conçus pour la vérification tels que ALPHARD, IOTA (NAK78).

Enfin, ajoutons que notre collaboration avec P.Y.CUNIN a permis de discuter et d'approfondir quelques propositions du langage prototype MEFIA (CUN79). Nous nous sommes plus particulièrement intéressés aux notions de propriétés et de relations de MEFIA qui permettent de caractériser les valeurs prises par un ou plusieurs objets. La définition de propriétés évite en effet au programmeur de déclarer de nouveaux types. On peut rapprocher l'utilisation de ces propriétés des définitions de types de données restreintes (BUC77) et de sous-types d'ADA (ADA80).

CHAPITRE IV. LANGAGES ORIENTES SYSTEMES DE BASE DE DONNEES.

I. L'APPROCHE EXTENSIUN DE LANGAGES EXISTANTS.

I.1. PASCAL-R.

I.1.1. Instructions élémentaires.

I.1.2. Constructeur de relation élémentaire.

I.1.3. Instruction d'itération foreach.

I.1.4. Généralisation des constructeurs de relation.

I.2. ASTRAL.

I.3. PLAIN.

I.3.1. Définition des données.

I.3.2. Accès aux données.

I.3.3. Opérations de manipulation de tuples.

I.3.4. Opérations de manipulation de relations.

I.3.5. Remarques.

II. CONCEPTION DE NOUVEAUX LANGAGES DE PROGRAMMATION.

II.1. RIGEL.

II.1.1. Les générateurs et les itérations.

II.1.2. Définition de "schémas externes" d'une base de données relationnelle.

II.1.3. Le mécanisme de vue.

II.1.4. Remarques.

II.2. TAXIS.

III. REMARQUES GENERALES ET CONCLUSIONS.

Dans ce chapitre, nous présentons quelques unes des récentes propositions de langages de programmation conçus pour développer des applications bases de données. Au chapitre III., nous avons montré que si les S.G.B.D. fournissaient des langages d'interrogation et de manipulation de bases de données relationnelles très intéressants, il n'en allait pas de même de leur insertion dans des langages hôtes d'interface avec les systèmes de gestion (STO77).

Pour réduire ces difficultés, de récents travaux de recherche proposent deux types d'approches:

-- l'extension de langages de programmation existants (PASCAL, EUCLID): de nouveaux constructeurs de types et de nouvelles structures de contrôle de haut niveau sont proposés,

-- et la conception de nouveaux langages de programmation dont les caractéristiques permettent de répondre à ces nouvelles préoccupations.

I. L'APPROCHE EXTENSION DE LANGAGES EXISTANTS.

Dans cette partie, nous ne présentons que les caractéristiques des nouvelles constructions introduites dans les langages existants exposés au chapitre IV.

I.1. PASCAL-R (SCH77).

Les extensions de PASCAL développées à l'Université de Hambourg par J.W.SCHMIDT comportent:

-- un constructeur de type de base appelé relation ; un élément d'une relation est représenté par un record PASCAL,

-- et de nouvelles constructions:

.soit élémentaires pour l'insertion, la suppression, la modification et l'affectation de tuples de relations,

.soit de haut niveau pour la manipulation d'ensembles, de tuples.

Examinons maintenant ces nouvelles constructions à partir d'un exemple.

Considérons un employé (d'une société) comme étant caractérisé par son numéro d'employé, sa qualification, son nom et son département:

PASCAL-R: type employé = record nempl,qual: integer ,nom: string ,ndpt: integer end;

L'ensemble des employés d'une société est défini par:

PASCAL-R: type relemp = relation <nempl> of employé;

Ici, nempl désigne la clé primaire de la relation relemp. On déclare une variable employés de type relemp de la manière suivante:

var employés: relempl;

I.1.1. Instructions élémentaires.

PASCAL-R fournit quatre opérations élémentaires dont la syntaxe est:

$$x \left\{ \begin{array}{l} :=\text{(affectation)} \\ :+\text{(insertion)} \\ :-\text{(suppression)} \\ :*\text{(remplacement)} \end{array} \right\} y$$

où x,y sont des variables relation de même type, la variable y n'étant jamais altérée par les instructions. Précisons rapidement chacune de ces opérations:

-- l' affectation (:=) a une sémantique classique (transfert par copie)

-- l' insertion (:+) ajoute (par copie) tous les éléments de y qui ne figurent pas dans x, c'est-à-dire dont les clés sont différentes

-- la suppression (:-) enlève de x tous les éléments de même clé que les éléments de y

-- le remplacement (:*) substitue dans x tous les éléments de même clé qu'un élément de y par cet élément de y.

I.1.2. Constructeur de relation élémentaire.

Ce constructeur noté [..] fabrique un type de relation à partir d'une variable record PASCAL. Si on déclare eltempl: employé, le constructeur élémentaire [..] permet d'écrire employés:+[eltempl].

I.1.3. Instruction d'itération foreach.

La structure syntaxique de cette instruction est la suivante:

```
foreach <variable de contrôle record de type T>
    in <variable relation du type T>
    do <instruction>
```

Illustrons cette instruction sur l'exemple des employés.

```
PASCAL-R: recherch:=[ ] ; /*initialisation*/
  foreach comp in employés do
    if comp.qual=2 then recherch:+[comp];
```

La variable de contrôle comp est locale à la boucle. Dans la portée de cette itération, ni la clé de comp, ni la relation employés ne doivent être modifiées. L'imbrication de ces itérations est autorisée.

I.1.4. Généralisation des constructeurs de relation.

La structure syntaxique de cette instruction est définie de la façon suivante:

```
[each <variable de contrôle record de type T>
 in <variable relation du type T>: <expression logique>]
```

avec dans l'expression logique des quantificateurs universels (all) et existentiels (some).

Illustrons cette instruction sur l'exemple des employés que l'on complète avec les déclarations suivantes:

```
PASCAL-R: type departement = record ndpt: integer ,nomdpt,adresse:
                  string end
          departements = relation <ndpt> of departement;
        var nancéiens, embauchés : employés
        begin
        nancéiens:= [each epl in employés: some dpt in departements
          ((epl.ndpt=dpt.ndpt) and (dpt.adresse='nancy')) ;
        embauchés:= each epl in employés: all dpt in departements
          (epl.ndpt≠dpt.ndpt)];
        end
```

Il est enfin possible d'utiliser un constructeur généralisé dont la syntaxe est la suivante:

```
[each (liste-d-éléments)
  for liste-de-variables-de-contrôle-de-types T1,T2,..,Tn
  in liste-de-variables-relations-des-types T1,T2,..,Tn:
  expression-logique]
```

Remarques : Par rapport à d'autres langages de programmation de haut niveau, par exemple SETL (SCH75), PASCAL-R fournit deux nouvelles constructions intéressantes:

-- la transformation par le constructeur [..] de variables PASCAL de type record en variables de tuples. L'intérêt d'un tel mécanisme est aussi montré par A.PIROTTE (PIR76). Ce constructeur permet de résoudre de manière syntaxique la liaison d'une variable de l'environnement de programmation aux objets de la base de données. Nous avons vu qu'avec SYSTEM-R (chapitre II), il était nécessaire d'établir explicitement cette liaison avec une instruction BIND et ensuite de contrôler les accès aux variables partagées par la base de données et l'environnement de programmation.

-- des expressions du calcul des prédicats pour caractériser les propriétés des objets de la base de données. Sur ce point, nous réitérons les remarques que nous avons formulées à propos de SQUARE: les quantificateurs conduisent souvent à des expressions difficiles à comprendre pour des informaticiens non spécialistes de la logique des prédicats.

Enfin, PASCAL-R n'offre qu'une possibilité d'abstraction de type de données dépendante de la représentation.

I.2. ASTRAL (RIN79).

ASTRAL ("A structured and unified approach to data base management and manipulation") développé à l'université de Trondheim par T.AMBLE, K.BRATBERGENSEN et O.RINES est un langage de programmation basé aussi sur PASCAL. Les extensions proposées concernent:

-- un mécanisme de définition hiérarchique des données,

-- et des constructions de langages de haut niveau orientées algèbre relationnelle.

Le constructeur de relation du langage de programmation ASTRAL est ici appelé table:

table of type-record;

spécification-des-attributs unique?

description-des-routines-locales?

Illustrons cette description syntaxique sur l'exemple suivant:

```
ASTRAL: type employés = table of
                    record nom:names;
                         sal:5000..20000;
                         mgr:names;
                         dno:1..100
                    end ;
                    unique nom;
                    end ;
```

Ici, le record PASCAL ne doit pas comporter de pointeur, ni de partie variable. La clé primaire de la table employés nom est définie par l'attribut unique . Avec une table, il est possible d'utiliser les

routines MAX, MIN, AVG, etc.. Dans ce cas, elles doivent obligatoirement être définies dans la partie description-routines-locales de la table. En ASTRAL, la définition du schéma d'une base de données est une description hiérarchique de modules appelés model :

```
ASTRAL: model société;
        type noms= string (20);
             dpt= string (20);
             dnos=1..100;
             salaires=5000..20000;
             emprec= record
                          nom:noms;
                          sal:salaires;
                          mgr:noms;
                          dno:dnos
                     end ;
             emptype= table of emprec;
                     unique nom;
                     end ;
        var employés:emptype;
        on access employés do
        case opération of
             insert:(corps-de-l'opération)
             update:(corps-de-l'opération)
             delete:(corps-de-l'opération)
        end
        export employés to paye;
        end
```


Un model est une unité de description modulaire qui comporte des déclarations de types, de variables et la définition des routines de contrôle d'accès aux variables table . Les types définis dans un model sont implicitement exportables, en particulier dans des programmes d'application. Seules les variables table sont contrôlées avec le mécanisme classique d' export / import . Les contraintes d'intégrité applicables aux variables tables sont définies dans les blocs on access ... end . Ainsi, à chaque appel d'une opération insert, update ou delete sur la table employés, le corps de l'opération correspondante du bloc on access est exécuté.

Le langage de manipulation de bases de données ASTRAL est basé sur l'instruction d'affectation (:=) et les expressions de tables. Par exemple, la formulation de la question suivante:

Q1: "rechercher les employés de la société qui ont un salaire supérieur à 5000F" s'exprime en ASTRAL par:

```
result[ false ] :=employés[sal>5000]
```

Chaque tuple de la "relation" employés qui satisfait le prédicat sal>5000 est recopié dans la relation result. Si le prédicat de la relation result avait été true, il n'y aurait alors pas eu de recopie. Il est possible, en ASTRAL, d'utiliser quelques constructions de prédicat un peu plus compliquées. Par exemple:

Q2: "rechercher les employés qui gagnent plus que leur manager"

```
ASTRAL: result:=employés[manager in employés:
                sal>manager.sal and
                mgr=manager.nom];
```

Remarques : Les mécanismes d'extensions développés dans le langage de

programmation ASTRAL sont fortement influencés par SIMULA. La hiérarchisation des model est tout à fait similaire aux classes hiérarchisées de SIMULA.

I.3. PLAIN (WAS79).

PLAIN (Programming LAnguage for INtersection") est développé par A.I.WASSERMANN à l'université de Californie de BERKELEY dans le cadre du projet USE. Ce langage a été conçu pour produire des systèmes logiciels d'applications de façon interactive. Un des objectifs visés avec PLAIN est l'intégration des concepts des S.G.B.D. relationnels dans un langage de programmation basé sur PASCAL.

Les principales facilités fournies par ce langage de programmation sont:

-- le type de base relation et le mécanisme de construction de variables tuples à partir de variables record PASCAL,

-- un mécanisme d'accès associatif à des tuples de relations et des opérations élémentaires d'adjonction (:+), de suppression (:-), de modification (:=) de tuples dans des relations,

-- des opérations relationnelles de sélection (where), de projection (==>), de join (join) et des opérations ensemblistes d'union (*), d'intersection (+) et de différence (-),

-- et un mécanisme d'abstraction de données par encapsulement (avec liste d'importation et contrôle de visibilité), de déclarations de types, de variables et de descriptions de procédures appelé program.

I.3.1. Définition des données.

Tous les objets utilisés dans un programme PLAIN doivent être déclarés d'un des types de base: integer , float , boolean , char , types scalaires, array , record , set , relation , file , string et pointeur (marking).

La déclaration de variables relations dans des unités spécifiques permet de définir une base de données. En reprenant l'exemple des employés d'une société, on écrit:

```
var employés: relation [key nempl] of
        nempl: integer ;
        qual:1..50;
        nom:char 20 ;
        nodpt:1..20
end employés
```

où nempl déclaré key définit la clé primaire de la relation employés. Cependant, dans une unité de programmation modulaire appelée program, il est possible de déclarer des variables relations pour lesquelles la portée des identificateurs suit la règle classique de bloc de PASCAL et des variables relations spécifiquement temporaires dont le type est appelé marking . Un "marking" ('pointeur caché') ne peut être créé que par une opération d'affectation d'expression relationnelle. Ce mécanisme n'est en fait introduit que pour décomposer des expressions relationnelles complexes en une suite d'expressions plus compréhensibles. On peut par exemple déclarer:

```
var nancéien : marking of employés   ou simplement
    temp: marking
```

Dans ce dernier cas, la structure du marking est inférée par les opérations qui la créent. Un marking n'a pas de clé.

I.3.2. Accès aux données.

PLAIN fournit un mécanisme d'accès associatif dans une relation appelé designator . Un tuple ainsi défini peut ensuite être référencé (%). Par exemple, on définit le tuple employés[10] de la relation employés à l'aide du numéro d'employé (10) qui est la clé associée. Maintenant, en écrivant: employés%:= employés[10], on définit une référence (employés%) au tuple (employés 10).

Ces mécanismes permettemt l'accès et la modification des attributs des tuples des relations. Sur l'exemple d'instruction suivante:

```
if employés[10].nom='mon nom'
then employés[10].qual:=20
endif
```

on voit que l'accès à l'attribut nom s'effectue par simple préfixage de l'attribut nom par le "désignateur" de tuple employés[10]. La modification de la valeur de la qualification pour le tuple de clé 10 de la relation employés s'obtient par l'affectation (:=) de la nouvelle valeur (20).

I.3.3. Opérations de manipulation de tuples.

Comme pour PASCAL-R, la notation [..] permet de construire une variable de tuple à partir d'une variable déclarée de type record PASCAL. Nous retrouvons les opérations d'insertion (:+) et de suppression (:-) de tuples de PASCAL-R que nous ne détaillons donc plus ici. Une instruction d'itération foreach permet de parcourir tous les tuples d'une relation ou d'un ensemble. Considérons l'itération suivante:

```
PLAIN: temp:= [ ] /*initialisation du marking*/
    foreach e in employés
    loop
        if e.qual=10
        then temp:+[<e.nom,e.nodpt>]
        endif
    repeat
```

Ce programme construit une relation temporaire temp par ajouts successifs de couples (nom,nodpt) des tuples de la relation employés dont l'attribut qual est égal à 10.

I.3.4. Opérations de manipulation de relations.

Nous présentons brièvement les opérations de l'algèbre relationnelle: la sélection (where), la projection (==>), le join (join) et les opérations d'affectation d'expressions relationnelles à des variables de type marking ou relation. Deux opérateurs d'affectation sont possibles: l'opérateur standard (:=) et l'opérateur (<-) pour obtenir les doubles dans des variables de type marking uniquement.

Par exemple, pour lister les noms des employés du département 10, on peut écrire:

soit temp:=employés where nodpt=10 ==> (nom)

ou temp<- employés where nodpt=10 ==> (nom)

Dans le premier cas, tous les noms recopiés dans la relation temporaire temp sont distincts alors que dans le deuxième cas, on peut avoir plusieurs fois le même nom.

Les expressions relationnelles de PLAIN ne peuvent qu'être des expressions simples formées d'une sélection, d'une sélection-projection ou encore d'un join. Le programmeur est donc obligé de décomposer les requêtes relationnelles en une suite d'instructions d'affectation simples.

I.3.5. Remarques.

Pour développer des applications bases de données, comme PASCAL-R, le langage de programmation PLAIN fournit deux niveaux de constructions:

-- des opérations élémentaires de mise à jour: insertion, suppression et modification de tuples de relations. Une instruction d'itération (foreach) utilisée avec les mécanismes de désignation/référence de tuples dans des relations offre la possibilité de manipuler les bases de données tuple par tuple,

-- des opérations relationnelles .

Ici, il est intéressant de rapprocher les mécanismes de PASCAL-R et de PLAIN. En PASCAL-R, les constructions de haut niveau sont définies comme des extensions des constructions élémentaires à des expressions du calcul relationnel. PLAIN fournit une autre approche basée sur l'utilisation disciplinée des opérations de l'algèbre relationnelle.

II. CONCEPTION DE NOUVEAUX LANGAGES DE PROGRAMMATION.

De récents travaux de recherche dont nous exposons maintenant quelques résultats, proposent de nouveaux langages de programmation conçus spécifiquement pour le développement d'applications base de données. Les traits caractéristiques de ces langages de programmation sont définis avec le souci évident d'améliorer la qualité des interfaces programmes d'application et systèmes de gestion de bases de données. Nous avons en effet vu que les extensions sont traitées soit par un préprocesseur (ALL76), soit par l'appel de subroutines (CHA76), ou encore par la modification de compilateurs existants (SCH77).

II.1. RIGEL (ROW79).

En cours d'implantation à l'université de BERKELEY (ROW80), ce langage de programmation expérimental a été conçu pour supporter le développement d'applications bases de données relationnelles. Ne disposant pas d'informations détaillées sur la réalisation de RIGEL, nous nous contentons de résumer quelques constructions de langage particulièrement attrayantes:

-- des instructions d'itération de haut niveau pour la manipulation des relations de la base de données ainsi que des "vues",

-- la définition de schéma de bases de données relationnelles avec un concept de module proche de MODULA,

-- la définition de "vues".

Comme dans les langages précédemment présentés, RIGEL supporte les types de données prédéfinis tuples et relations. Nous n'y revenons donc pas.

II.1.1. Les générateurs et les itérations.

En RIGEL, les itérateurs (EAR73) sur les relations sont des constructions fondamentales pour définir les instructions d'itération, les "fonctions d'aggrégat" (par exemple, calcul d'une moyenne, d'un maximum) ou encore les relations temporaires. Ces itérateurs appelés "expressions liées" (bind expressions) constituent des expressions d'interrogation relationnelle de la base de données. Une "expression liée" comporte une variable d'itération locale à l'expression et un "générateur" qui fournit une séquence de valeurs successivement assignées à la variable d'itération. Pour définir d'autres générateurs, il faut spécifier des routines procédures appelées generator . Nous allons voir maintenant que selon le type d'action appliquée aux tuples fournis par les générateurs, différentes instructions d'itération doivent être utilisées: la sélection (itération for) ou la mise à jour (itération update).

-- Instruction d'itération "for" .

Illustrons l'utilisation de cette instruction d'itération sur l'exemple de la relation employés:

Q: "lister les noms des employés du département numéro 10"

```
RIGEL: for e in employés where e.nodpt=10
          do print (e.nom);
       end
```

Cette instruction extrait de la relation employés une séquence de tuples filtrés par la condition définie dans la clause where, et ensuite imprime pour chacun d'eux la valeur sélectée par l'attribut de tuple nom. Les instructions de modification de tuples sont interdites dans le bloc do -- end . Dans cette itération for , "e in employés" représente l'expression liée à la variable d'itération locale et employés le générateur de tuples associé à la relation employés.

Remarque : Les concepteurs de RIGEL envisagent de généraliser les expressions liées à des expressions qui comportent plusieurs générateurs.

-- Instruction d'itération "update" .

Si le bloc do end de l'itération comporte une instruction de suppression (delete) ou de modification (replace) de tuple, il faut utiliser une itération update au lieu de for . Par exemple:

Q: "supprimer les employés dont la qualification est égale à 5"

```
RIGEL: update e in employés where e.qual=5
       do delete e;
       end
```

ou encore:

Q: "modifier la qualification de tous les employés par la routine newqual"

```
RIGEL: update e in employés
       do replace e by e<qual:newqual(e.qual,e.nodpt>;
       end
```

Notons que la procédure qui calcule à partir d'une qualification et d'un numéro de département donné une nouvelle qualification d'employé est spécifiée par ailleurs.

Les instructions de modification (replace), de suppression (delete) ou comme nous le signalons un peu plus loin d'adjonction (append) de tuples dans une relation ne modifient pas directement la base de données des employés. Pour l'itération update, toutes les modifications de tuples sont en fait copiées sur un fichier de travail. La mise à jour effective de la relation de base est différée jusqu'à la terminaison de l'itération. Ce procédé de mise à jour différé, fort bien adapté à l'implantation sur système UNIX, est appliqué pour des considérations d'intégrité et de sécurité de la base: modification d'un inx secondaire, de modification non fonctionnelle -- remplacement d'une valeur par plusieurs autres -- et aussi de 'recovery' en cas de "crash" du système (STO76). En réalité, RIGEL n'assure pas la fonctionnalité des modifications.

-- Instruction "append" .

Contrairement aux instructions élémentaires de modification de tuples précédentes replace et delete , l'instruction d'adjonction de tuple append ne peut figurer dans une itération. Ainsi, la requête:

Q: "ajouter le tuple (15,5,'paul',10) à la relation employés" s'écrit:

```
RIGEL: append
       <nempl:15,qual:5,nom:'paul',nodpt:10>
       to employés
```

Remarque : Comme nous le montrons dans le paragraphe suivant, les instructions élémentaires de modification peuvent aussi figurer dans le corps de routines procédures abstraites. Cette facilité nécessite cependant l'introduction d'une nouvelle construction de langage: les types référence de tuples ; autrement dit, des pointeurs typés, de type tuple. Par exemple, la spécification:

```
updatequal: procedure (e: tuple employés' type )=
   begin
      replace e by e<qual'newqual(e.qual,e.nodpt)
   end
```

permettrait de réécrire l'itération de modification de la qualification des employés de la façon suivante:

```
   update e in employés
   do updatequal(e)
   end
```

Remarque : Ce dernier exemple nous incite à penser que le contrôle de la nature des opérations utilisées dans le corps des itérations for n'est effectué qu'à l'exécution.

II.1.2. Définition de "schémas externes" d'une base de données relationnelle.

En RIGEL, l'interface entre un programme utilisateur et une base de données relationnelle est spécifié par un module de conception proche des modules de MODULA. La spécification d'un module RIGEL comporte trois parties:

-- l'interface abstrait -- public -- où figurent des déclarations d'objets exportables (visibles de l'extérieur du module): relations, profils syntaxiques de procédures qui représentent des opérations abstraites sur les relations exportées,

-- l'implantation -- private -- des abstractions spécifiées dans la section public . Les objets locaux à cette partie ne sont accessibles

de l'extérieur que par les procédures exportées,

-- et une partie initialisation du module, locale au module. Notons qu'en MODULA, les modules doivent être explicitement initialisés par une routine. Explicitons maintenant cette notion de module RIGEL sur l'exemple de la base des employés.

```
RIGEL: emp-schema: module =
    public
    employés: relation
                  nempl#:integer;
                  qual:qualtype;
                  nom:idnomtype;
                  nodpt:idnotype;
                  key nompl;
              end
    delempl: procedure (q:qualtype);
        .
        .
    private
    delempl: procedure =
        begin update e in employés where e.qual=q
            do delete e;
            end
    end;
    .
    .
    initialization
    end emp-schema;
```

Notons de suite qu'un module RIGEL ne comporte pas de liste d'importation. Autrement dit, tous les types de données définis dans l'environnement de compilation du module RIGEL emp-schema sont implicitement importables dans la définition du module. La liaison entre les variables relation et la base de données nécessite l'utilisation d'un langage externe de connexion que nous ne détaillerons pas. Quelques informations sont données dans (ROW79). En RIGEL, il n'existe pas de mécanisme de définition de types abstraits de données. La notion de module, telle qu'elle introduite dans le langage est un moyen d'abstraction de données et plus précisément d'abstraction de bases de données relationnelles.

Pour que les relations et opérations abstraites spécifiées dans la partie public de modules soient visibles dans un programme d'application, il faut que tous les modules qui contiennent des objets abstraits utiles soient explicitement importés dans le programme. Il est évident qu'avec ce mécanisme d'importation, le programmeur est obligé d'importer plus de choses qu'il n'en a besoin. Le résultat n'est pas toujours adapté à sa propre vision de la base de données. Pour réduire ces inconvénients, et par là même améliorer la communicabilité, la protection et l'indépendance des données de la base, un nouveau type de relation appelé view a été introduit dans le langage.

II.1.3. Le mécanisme de vue.

Une vue (CHA76) en RIGEL est définie par un module qui contient dans

-- la partie public la déclaration de relations abstraites (view) avec éventuellement des opérations abstraites sur ces relations abstraites,

-- et, dans la partie private , la spécification des mappings des relations abstraites sur les relations de base avec les corps des opérations abstraites. Illustrons ce mécanisme en spécifiant la vue dpt-vue sur la relation de base précédemment définie employés:

```
RIGEL: dpt-vue : module
    public
      /*déclaration de la relation abstraite dept-10*/
    dept-10: view
        nempl#:integer;
        nodpt:idnotype;
        qual:qualtype;
    end
    totempl: procedure (q:qualtype):integer
    .
    .
    private
      /*spécification du mapping dela vue*/
       dept-10: view =<nempl#:e.nempl#,nodpt:e.nodpt, qualie.qual>:
            e in employés where e.nodpt=10;
     totempl: procedure
       begin
```

```
return count (x in employés where x.qual=q);
end
```

end dpt-vue

Notons que la relation abstraite dept-10 déclarée dans la partie public du module définit la partie visible de ce que les spécialistes des bases de données relationnelles appellent communément une vue (CHA76). La spécification du mapping de la vue est décrite séparément dans la partie private . Ce mapping qui définit la sémantique de la construction de la relation abstraite dept-10 à partir de la relation de base employés est donc caché à l'utilisateur. Dans l'exemple, la vue dept-10 est la projection sur les "colonnes" nempl, nodpt et qual du sous-ensemble de "rangées" de la relation de base employés, obtenu par sélection sur le numéro de département égal à 10.

Les opérations abstraites spécifiées dans la partie public sont implantées comme on le voit avec la procédure totempl, sur la relation de base employés. Ces constructions: relation abstraite et mapping fournissent une solution élégante aux problèmes soulevés par l'importation directe des relations de base mais introduisent par ailleurs tous les problèmes bien connus (DAT77) de mise à jour des vues. Pour ne pas interdire totalement la mise à jour des vues, les concepteurs de RIGEL proposent une solution intermédiaire qui consiste à n'autoriser que les mises à jour spécifiées dans des opérations abstraites du module.

Des constructions ont été définies pour implanter des vues de manière performante. Le lecteur intéressé pourra trouver les détails dans (ROW79).

II.1.4. Remarques.

Nous avons cru bon d'exposer en détail quelques constructions de RIGEL qui, à notre connaissance, est un des premiers langages de programmation à avoir été spécialement conçu pour supporter le développement d'applications bases de données relationnelles (STO77).

Un prototype RIGEL réalisé sur machine VAX/11/780 sous système UNIX devait être opérationnel au cours du premier semestre 80. Cette version interprétée exploite les méthodes d'accès du S.G.B.D. INGRES --développé aussi à BERKELEY -- pour stocker et extraire les données. L.A.ROWE et K.A.SHOENS envisagent dans une seconde étape d'implanter un compilateur avec compilation séparée des modules ainsi qu'un "debugger" de langage source. Ils étudient également le renforcement des contrôles sémantiques de certaines constructions du langage.

Nous conclurons rapidement sur ce langage, car nous discuterons à plusieurs reprises dans la suite des divers choix de conception qui ont été faits. L'originalité de RIGEL est d'être un langage de programmation de très haut niveau fournissant un environnement adapté au développement d'applications bases de données relationnelles . RIGEL ne supporte pas de mécanisme de définition de types abstraits de

données.

II.2. TAXIS (MYL79).

Développé à l'université de TORONTO par J.MYLOPOULOS, P.BERNSTEIN et H.WONG, l'originalité du langage de programmation TAXIS résulte du rapprochement des concepts de classes hiérarchisées de SIMULA (DAH70) et de réseau sémantique (LEV79).

Nous ne discuterons pas des caractéristiques de ce langage très récent. Signalons simplement qu'il vise à offrir des facilités de gestion de base de données relationnelles, de spécification des contraintes d'intégrité et de traitement d'exceptions.

III. REMARQUES GENERALES ET CONCLUSIONS.

Il y a une dizaine d'années, les concepteurs des premiers S.G.B.D. --hiérarchiques et réseaux-- furent déjà confrontés à quelques uns des problèmes dont nous venons de discuter dans ce chapitre. Par exemple, le système SOCRATE a été conçu comme un S.G.B.D. devant fournir des possibilités d'interface avec des langages de programmation externes existants --COBOL, FORTRAN--, alors que le système CODASYL est une extension de COBOL qui supporte des accès à une base de données réseau. La recrudescence de l'intérêt porté à la conception de langages de programmation de haut niveau qui supportent le développement d'applications bases de données (STO77) est dû aux nouvelles possibilités offertes à la fois par les S.G.B.D. relationnels et les langages de programmation de haut niveau. C'est d'ailleurs la diversité des investigations qui nous a amené à faire une présentation quasi exhaustive des récentes propositions. On a pu remarquer que le constructeur de type relation est défini dans tous les langages dont nous avons discuté, situant de facto ces approches dans le cadre des bases de données relationnelles.

Il nous semble que la notion de propriétés du langage prototype MEFIA (CUN79) pourrait permettre d'améliorer la définition des contraintes d'intégrité et leur contrôle. En effet, même dans un langage fortement typé comme PASCAL-R, beaucoup de contraintes d'intégrité ne peuvent pas être conservées lors des mises à jour des relations. C'est une des raisons qui justifient la définition des mécanismes d'abstraction de données en PLAIN, RIGEL (ROW79) et ASTRAL

(RIN79). Certaines opérations de modification qui nécessitent la vérification de contraintes d'intégrité peuvent être spécifiées comme opérations abstraites des modules d'interface. Il semble, mais nous ne disposons pas sur ce point d'informations suffisamment détaillées, que cela pose quelques problèmes de contrôles de la sémantique de certaines instructions, par exemple, pour les itérations de RIGEL (ROW79).

Notons encore que RIGEL, PLAIN et ASTRAL envisagent tous les trois la conception d'un système de construction modulaire des interfaces avec la base de données, mais ne proposent pas (encore) d'intégrer la construction modulaire de programmes utilisateurs. En fait, les concepteurs de ces langages de programmation ne considèrent que les applications bases de données relationnelles dont les caractéristiques générales seraient:

-- une description de schémas de bases de données complexe,

-- et un grand nombre de transactions simples et volatiles.

Un autre aspect important, qui n'est abordé pour l'instant que par RIGEL concerne l'adaptabilité des interfaces, la protection et la confidentialité des données. Il est bien évident que ces récentes investigations sont encore balbutiantes et que beaucoup d'études sont encore à entreprendre.

DEUXIEME PARTIE

PRESENTATION DU SYSTEME VEGA

D'AIDE A LA SPECIFICATION ALGEBRIQUE

DE TYPES ABSTRAITS ET/OU PROGRAMMES

ET A LA VALIDATION

CHAPITRE V. PROBLEMATIQUE DE LA SPECIFICATION DE PROBLEMES.

I. ROLES DE LA SPECIFICATION FORMELLE DANS LE DEVELOPPEMENT DES SYSTEMES LOGICIELS.

I.1. La spécification est un contrat.

I.2. La spécification est un outil efficace d'aide à la conception logique.

I.3. La spécification est une aide à l'implantation.

I.4. La spécification est une aide à l'évolution.

II. QUELQUES CONSIDERATIONS GENERALES SUR LES SPECIFICATIONS INITIALES.

INTRODUCTION.

La première phase du développement d'un projet informatique est en général celle de l'"analyse-conception" des intentions initiales (ou besoins) des utilisateurs, consignées dans ce que l'on a coutume d'appeler le cahier des charges du projet. Ce début de résolution vise à transformer une expression "informelle", souvent ambigue, contradictoire ou incomplète des besoins, en une représentation "conceptuelle" satisfaisante. Ce travail difficile et ingrat exige la plupart du temps, de procéder par approximations successives: analyse du cahier des charges et enquêtes auprès des utilisateurs. La fin de l'itération est une nouvelle formulation des problèmes qui peut prendre des formes très diverses: dictionnaires de données ou bases documentaires, diagrammes et parfois même descriptions "papier" plus formelles. Pour réaliser cette tâche, les analystes disposent d'un certain nombre d'aides bien connues: méthode de JACKSON (JAC75), méthode des machines abstraites (GAL76), HIPO (MYE75), SADT (RDS77), SREM (ALF77), etc.., et/ou lorsque les problèmes sont plus orientés "données", des techniques fondées sur le processus de normalisation du modèle de données relationnel défini par E.F.CODD: Le lecteur intéressé trouvera dans les références précédentes et dans (HOW82) des informations détaillées sur toutes ces techniques.

En ce qui concerne notre travail, nous retiendrons seulement qu'elles soulèvent toutes (à des degrés divers, cela s'entend) les mêmes types de problèmes: de contrôle du processus d'analyse-conception, de compréhension et donc de communication des résultats de l'analyse, de validation de la fidélité des représentations conceptuelles aux intentions initiales des utilisateurs, de vérification de la cohérence et de la complétude des représentations décrites, d'évolution, etc...

Il est maintenant admis, tout au moins dans le domaine du génie logiciel (ADI82) (BOE82) (WAS82), que cette première étape de résolution constitue une étape clé du développement. En effet, de la façon dont on l'aborde dépendent la qualité du logiciel produit et, ce qui nous semble tout aussi important, la possibilité de modifier les objectifs du projet: fabrication de maquettes, de prototypes plutôt qu'un système opérationnel (WAS82).

Dans le projet TYP, la spécification de problèmes orientés données a été depuis le début une de nos préoccupations essentielles (CHA76). Précisons tout de suite que si notre travail repose sur des résultats théoriques récents, notre but est cependant essentiellement pragmatique. Pour nous, il s'agit de dégager des concepts, des techniques et des outils d'aide à la spécification formelle de problèmes qui puissent s'intégrer de façon cohérente et harmonieuse dans un atelier de conception et de production de "gros" logiciel (KRA81). Il est bien évident que pour atteindre cet objectif, une grande rigueur théorique est absolument nécessaire.

Des premières investigations sur la spécification, deux approches se sont très vite dégagées: celle des types abstraits dans les langages de programmation algorithmiques, présentée dans la première partie de ce travail et les techniques de spécification formelle (ABR78) (PAI79) (LIS75) (GUT78) (GOG79) (LES79) (FIN79). A ce moment là, la théorie des types abstraits était balbutiante. Il nous paru bien difficile, à partir de ces premiers résultats, de pouvoir en évaluer toute la portée pratique. Notre expérience du système de programmation modulaire CIVA (DER74) et ces premières constatations nous ont tout naturellement conduit à engager la conception du système TYP présenté en troisième partie tout en poursuivant les investigations sur les techniques formelles. Ce n'est donc qu'après avoir longuement étudié les différents travaux sur la spécification formelle, les langages orientés résolution de problèmes comme PROLOG (ROU75), et réalisé le projet TYP que des mécanismes et des outils d'aide à la spécification formelle ont pu être progressivement dégagés et ensuite réalisés.

Avant de préciser ce que nous pensons être le rôle d'une technique de spécification formelle dans le processus de développement d'un projet informatique, il n'est peut être pas inutile de rappeler ici que certains informaticiens dits "de gestion" ont consacré beaucoup d'efforts depuis une dizaine d'années à ce problème que l'on appelle maintenant la spécification (HOW82) (INF76) (DUF80) (FRE76). Il n'y a pas de doute que la production intensive de systèmes logiciels de plus en plus sophistiqués mais aussi de plus en plus coûteux a permis de révéler l'insuffisance des techniques et des outils utilisés jusque là (BOE82). Cette prise de conscience apparaît d'ailleurs dans de nombreuses branches de l'informatique (PIN81) (WAS82): conception de langage de programmation, bases de données (BRO82) et intelligence artificielle (LAU82). Ce mouvement devrait encore s'amplifier dans les années à venir car, que constate-t-on aujourd'hui?

. de nombreux systèmes logiciels, malgré des coûts de production et de maintenance très élevés (BOE82), ne satisfont pourtant pas (ou mal) toutes les intentions initiales des utilisateurs,

. le travail d'analyse-conception du cahier des charges relève de pratiques beaucoup trop "archaiques" (DUF80),

. les erreurs de conception dues à des spécifications insuffisantes sont fréquentes et très coûteuses car elles ne peuvent être détectées qu'en phase de test, donc beaucoup trop tard,

.en l'abscence de spécifications acceptables, il est impossible de pouvoir certifier les logiciels produits,

.enfin, mais cette liste n'a pas la prétention d'être exhaustive (BOE82), que les problèmes d'évolution, notamment lorsqu'il s'agit des

spécifications, ont tendance à remettre en cause la viabilité des systèmes logiciels déjà fabriqués.

I. RÔLES DE LA SPECIFICATION FORMELLE DANS LE DEVELOPPEMENT DES SYSTEMES LOGICIELS.

Quelques études (ZEL78) (BOE82) s'appuyant sur de nombreuses analyses de cas de développement de systèmes logiciels fournissent des indications parfois surprenantes tant sur la répartition du travail effectué aux différentes phases du développement que sur les techniques et les outils utilisés, ou même sur la nature des difficultés rencontrées. Bien que la spécification, d'après ces études, ne représente que 10% de l'effort total de développement, son impact dans la fabrication de systèmes logiciels de qualité est un fait maintenant admis dans le domaine du génie logiciel.

Une spécification ou plus précisément un système de spécifications formelles doit remplir quatre fonctions essentielles.

I.1. La spécification est un contrat.

Une spécification est un contrat entre le client et le fabricant de logiciel. Cette spécification ne peut être une référence indiscutable que si les deux parties ont effectivement la possibilité d'en élaborer le contenu en étroite collaboration. Pratiquement, cela signifie qu'il doit être possible d'une part de vérifier que la spécification formelle est cohérente et aussi complète que possible, d'autre part de vérifier ou tout au moins d'acquérir l'"intime conviction" que cette même spécification décrit fidèlement toutes les intentions initiales du client.

I.2. La spécification est un outil efficace d'aide à la conception logique.

Les langages (ou systèmes) de spécification formelle permettent en général de définir une première spécification abstraite -représentation conceptuelle- d'un problème qui peut ensuite être progressivement raffinée en introduisant de plus en plus de détails de conception. Le fait que la plupart des propositions de systèmes de spécification: Z (ABR78) (FIN79), HDM (LEV79), FDM (EGG80), AFFIRM (MUS80), SPRAC (FOI82), GYPSY (GOO78), FORMEL (HUE82) REVE (LES82) aient comme objectif premier d'automatiser la vérification de la correction (et/ou de la sécurité) des systèmes logiciels, a sans aucun doute eu un effet déterminant sur la conception des langages de spécification. Comme J.P.FINANCE (FIN79), nous pensons même que certaines de ces techniques

de spécification formelle de programmes ne sont pas suffisantes pour entreprendre la conception d'une première spécification de problème acceptable. Par certains traits de conception: "surcharge", notations infixées, préfixées, suffixées et/ou mixtes des opérations, et/ou l'explicitation des cas d'exceptions et d'erreurs, et/ou la paramétrisation des types de données, etc.., mais aussi et surtout, par la possibilité de réaliser très facilement l'exécution symbolique de plans de tests, des systèmes comme OBJ (GOG79) ou encore LPG (BER82), OASIS (BAR82), nous semblent déjà apporter une aide efficace à la conception logique d'une première spécification de problème relativement intéressante.

I.3. La spécification est une aide à l'implantation.

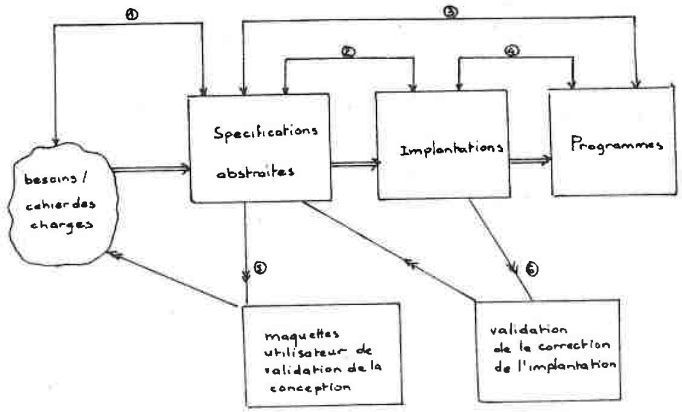
Une des principales caractéristiques d'une spécification (formelle) abstraite est d'être indépendante de tout choix d'implantation, à fortiori de tout choix de langage de programmation et de machine. Une implantation, dans cette approche formelle, est simplement une étape de raffinement d'une spécification abstraite, où l'on précise certains choix de conception logique. Précisons que la définition de ces choix n'utilise que des spécifications abstraites de l'environnement de spécification déjà construit. Ainsi, si l'on décide de "descendre" la spécification au niveau des concepts d'un langage de programmation, ce qui n'est pas obligatoire (WAS82), alors la spécification formelle modulaire et progressive devient, en plus d'une aide conceptuelle au codage, un outil d'aide à la vérification de sa correction.

I.4. La spécification est une aide à l'évolution.

Les études auxquelles nous nous sommes déjà référé plus haut indiquent que le travail de maintenance réalisé sur les systèmes logiciels actuels représente 67% de l'effort total de développement. Ce chiffre peut toujours être discuté. Nous le citons tout de même car il montre bien quelle est la préoccupation majeure des constructeurs de "gros" logiciels. Nous croyons que l'utilisation de systèmes de spécification formelle supportant des techniques de décomposition modulaire et de raffinements progressifs, et d'une base de spécifications correctes (MUS80) (PRO82) devrait permettre une meilleure maîtrise de l'évolution des systèmes logiciels: prototypes et systèmes opérationnels. Il est évident que cela ne peut s'envisager qu'avec des environnements de programmation appropriés, ce qui n'est pas le cas aujourd'hui.

La présentation sommaire de quelques aspects de ce que nous pensons être des fonctions essentielles d'un système de spécifications formelles appelle une première remarque importante. Nous sommes très conscients d'une part que certains choix de conception avancés sont discutables, d'autre part que nos propositions pourraient certainement être complétées. En effet nous ne traitons pas certains problèmes comme la documentation et le suivi de projet, le parallélisme et la synchronisation, etc... Il ne faut cependant pas oublier qu'actuellement nous ne disposons d'aucune expérience significative sur le développement de systèmes logiciels à partir de techniques de spécification formelle. Un des objectifs de notre travail est donc de

proposer des mécanismes et des outils qui nous serviront précisément à montrer la faisabilité de certaines de ces idées. Nous décrivons dans le schéma ci-dessous les relations entre les différentes étapes de la spécification.



1.conception logique

2.conception des implantations des spécifications abstraites

3.et 4.codage éventuel des spécifications dans un langage de programmation algorithmique

5.validation, par les utilisateurs, de la conception logique des entités définies par le spécifieur

6.validation de la correction des implantations des spécifications abstraites.

II. QUELQUES CONSIDERATIONS GENERALES SUR LES SPECIFICATIONS INITIALES.

Dans la pratique, la caractérisation des intentions initiales des utilisateurs se présente souvent comme l'amalgame d'un grand nombre de détails, ce qui conduit évidemment à des cahiers des charges d'une importante complexité. Si, comme nous l'avons déjà dit plus haut, de plus en plus de méthodes d'analyse préconisent des approches dérivées des principes de décomposition et des raffinements successifs, ces aides conceptuelles perdent beaucoup de leur efficacité du seul fait qu'il n'existe pas, à notre connaissance, d'outil (opérationnel) de spécification supportant ces méthodes, pour décrire et "gérer" de façon automatisée les résultats de cette analyse.

Cette considération préliminaire a une certaine importance, car elle élimine "ipso facto" de notre champ d'investigations beaucoup d'outils de programmation traditionnels: systèmes de gestion de bases de données, langages de programmation classiques (BASIC, FORTRAN, COBOL et PL1). D'autres langages (ou systèmes) de programmation modernes comme SIMULA (DAH70),CLU (LIS74), ADA (ADA80), TYP (CHA82) (DER79) peuvent effectivement servir de langages de spécification; toutefois, comme beaucoup d'autres (ABR78) (FIN79) (PAI74), nous pensons qu'un langage de spécifications initiales doit en plus être essentiellement déclaratif (c'est-à-dire indépendant de tout commencement de résolution) et statique (c'est-à-dire indépendant des notions dynamiques des langages de programmation: variables et séquencement des instructions).

Les seuls langages qui possèdent ces deux dernières propriétés sont des langages formels avec des sémantiques définies dans un certain nombre de théories formelles:

-- la théorie des ensembles (SCH75) (PAI79) (LAU82) etc..

-- le calcul des prédicats (COD70) (NAK78) (FIN79) (REI77) (GAL78)

-- les théories algébriques (GUT80) (ZIL80) (GOG78) (LES79) (MUS80) (BER82) etc...

L'étude que nous avons pu faire de ces propositions nous inspire plusieurs réflexions:

a) Une première d'ordre général: les recherches sur les langages (ou systèmes) de spécification formelle sont encore balbutiantes sur beaucoup d'aspects. Bien que les théories formelles utilisées soient bien connues, paradoxalement, la sémantique formelle des langages définis sur ces théories n'est pas, à quelques exceptions près, aussi "claire" que leurs auteurs veulent bien le dire (BID81) (GOG78).

Un certain nombre de traits de conception de langages formels orientés types abstraits demandent encore des approfondissements théoriques: exceptions ou erreurs, types abstraits paramétrés avec erreurs et/ou des propriétés sur les paramètres, sous-types abstraits ou encore choix de la forme des axiomes, etc..., mais aussi, d'être plus largement validés avec plus de réalisations de systèmes de spécification. Les quelques expérimantations de ces systèmes: DTVS-AFFIRM (GER79), OBJO-OBJT (GOG79), etc... deviennent pratiquement caduques dès que nous parvenons à en prendre connaissance, du fait de l'évolution très rapide de ces systèmes expérimentaux. Dans ces

conditions, il nous paraît bien difficile de définir des critères d'évaluation de ces techniques de spécification formelle qui ne soient pas, en définitive, essentiellement théoriques. Nous pensons, en effet, que les critères proposés par B.LISKOV (LIS77) dans "an introduction to formal specification of data abstraction": compréhensibilité, large domaine d'application ou même extensibilité, concernent plus les systèmes "en devenir" que les systèmes expérimentaux actuels.

Maintenant, si nous résumons les critiques les plus courantes de certains spécialistes de ces systèmes, on note:

- un "manque de crédibilité" (GER79): les spécifications à fournir au système de preuve sont souvent plus "obscures" que les programmes à prouver (ACM82 p38),
- un "certain ésotérisme" (CAP78): seuls quelques initiés sont capables de se servir de ces systèmes,
- une "faible portée" (ACM82): les systèmes de spécification et de preuve ne permettent de manipuler qu'une centaine de lignes de code,
- etc...

Ces quelques critiques ressemblent d'ailleurs beaucoup à celles que certains spécialistes des systèmes de gestion de bases de données ont déjà pu formuler à l'encontre des premières expérimentations du modèle de données relationnel de E.F.CODD (COD70) (COD71). En fait, beaucoup de maquettes et de prototypes (KIM79) de SGBD relationnels ont du être réalisés et expérimentés avant que les SGBD les plus populaires comme INGRES (STO76), SYSTEM-R (AST76), QUERY-BY-EXAMPLE (ZLO75) et quelques autres ne parviennent à acquérir leurs "lettres de noblesse". Ce

parallèle SGBD relationnels-systèmes de spécification formelle ne peut que conforter notre point de vue sur l'évolution des langages (ou systèmes) de spécification. En effet, la plupart des recherches sur ce que nous appellerons "le confort de l'utilisateur" de SGBD relationnels comme: la compréhensibilité des langages d'interrogation et de mise à jour des bases de données (PIR76), l'adaptabilité des contraintes de sécurité et d'intégrité des données, l'extensibilité des schémas n'ont véritablement abouti qu'avec la conception de cette génération de SGBD, c'est-à-dire après de nombreuses expérimentations (au moins 7 versions pour le seul système INGRES!).

b) La deuxième réflexion concerne les différents cadres d'application de ces techniques formelles. Nous avons déjà eu l'occasion de faire remarquer que de nombreux domaines de l'informatique: langages de programmation, bases de données, intelligence artificielle se référaient de plus en plus souvent aux mêmes concepts et principes généraux de la méthodologie de la programmation (SIG76) (PIN81). Cet attrait est en grande partie du au fait qu'ils peuvent s'exprimer dans plusieurs cadres formels, et en particulier dans ceux des différents domaines de l'informatique (calcul des prédicats, théories algèbriques).

Actuellement, beaucoup de recherches s'intéressent aux techniques de spécification formelle:

.en compilation (GAU80) (PAI80)

.en spécification de problèmes généraux (ABR78) (GUT80) (GOG78) (FIN79) (BER82) (DER82) (ACM82)

.en spécification de systèmes de bases de données (BRO82) (REI77)

(WEB78) (GAL78) (ACM82)

. en intelligence artificielle (systèmes experts) (CAU82) (TSO79) (GAL78).

De cette récente évolution semblent déjà se dégager quelques caractéristiques d'une nouvelle approche des problèmes informatiques: les systèmes experts (LAU82). Des recherches très prometteuses, mais aussi très ambitieuses commencent déjà à envisager la conception d'une nouvelle architecture de système qui intègrerait la plupart des concepts, des techniques et des outils de plusieurs disciplines avancées de l'informatique: intelligence artificielle, bases de données relationnelles, systèmes de programmation avec types abstraits (KAH82). Nous pensons que les systèmes de spécification formelle devraient déjà pouvoir être considérés comme des systèmes experts et pas uniquement, comme c'est si souvent le cas aujourd'hui, comme des systèmes interactifs de preuve de programmes PASCAL (ACM82).

C'est en fait cette idée essentielle qui nous a conduit à entreprendre la conception du système VEGA. Précisons que c'est aussi de cette idée que débattent, à l'heure actuelle, certains spécialistes de la spécification.

Examinons par exemple quelques propositions de la spécification de problèmes généraux. Une première approche (ABR78) (FIN79) vise essentiellement à définir un énoncé initial formel et structuré des problèmes que l'on essaie ensuite de transformer progressivement et de façon aussi systématique que possible (voire automatiquement) en un

programme correct et efficace. Si ces techniques semblent très bien se prêter à la construction de programmes corrects, la preuve se faisant au fur et à mesure du développement, on peut toutefois s'interroger sur les possibilités de vérification de la fidélité des spécifications initiales, et ensuite des programmes, aux intentions initiales. Or, il est maintenant de plus en plus admis (ADI82) (KRA81) qu'une des étapes critiques de la conception des systèmes logiciels est précisément celle de la conception de spécifications "initiales" acceptables. Elle est critique pour deux raisons, au moins: la première, parce qu'elle exige un véritable travail de création; la seconde, parce que le travail de spécification (analyse-conception) ne peut se faire qu'en étroite collaboration avec les utilisateurs, ce qui pose, avec les langages de spécification formelle actuels, d'énormes problèmes de communication et de compréhension.

C'est ce type de préoccupations qui a conduit GOGUEN (GOG79), LEVI et SIROVITCH (LEV77), à concevoir des langages de spécification algébrique "exécutables", GUTTAG (GUT80) à interfacer des spécifications algébriques par des "routines" définies avec des "plus faibles préconditions" à la DIJKSTRA.

En fait, ces dernières propositions préfigurent déjà des futures évolutions des systèmes de spécification formelle. En effet, il apparaît de plus en plus (ADI82) qu'un système de spécification formel doit pouvoir supporter plusieurs types d'aides à la vérification des spécifications formelles:

- des aides aux spécifieurs: prouveurs de théorèmes (KNU70) (MUS80)

(ACM82), possibilités d'interprétation symbolique de plans de tests, de "debugging" permettant de valider ou plutôt d'invalider en cas d'échec des spécifications algébriques,

- des aides aux utilisateurs: certaines, éventuellement identiques à celles fournies aux spécifieurs: interprétation symbolique de plans de tests et "debugging"; mais aussi d'autres facilités comme l'élaboration de maquettes avec des langages logiques non déterministes. Nous croyons, en effet, que ce type de système logique fournit une aide logicielle beaucoup plus adaptée à une manipulation directe par les utilisateurs que les systèmes algébriques.

Après ces considérations sur les techniques, on est naturellement amené à s'interroger sur leur applicabilité aux domaines "traditionnels" de l'informatique. C'est un problème très important, mais qui demanderait pour être traité des approfondissements qui vont bien au delà de notre cadre de travail. Il nous semble toutefois que certains spécialistes de la spécification de problèmes "généraux", s'appuyant sur le principe de la "généralité des techniques formelles", ne prennent pas suffisamment en compte la spécificité des domaines d'application où ils veulent les introduire.

Considérons, par exemple, le cas de la paie dans les entreprises ou les administrations. Nous croyons, mais cet avis exige encore d'autres investigations pour être infirmé ou confirmé, que les exigences de précision, de minimalité des techniques de spécification, deviennent très vite une contrainte difficile à assumer (pour l'instant), dès qu'il s'agit d'analyser des textes comme ceux de la législation du

travail. Si, comme nous le croyons, le but recherché est de produire des programmes certifiés corrects, adaptables, au moindre coût, aux différents contextes utilisateurs, alors il nous semble plus confortable et tout aussi sûr d'écrire directement des programmes PASCAL ou ADA qui pourraient ensuite être prouvés avec des systèmes du type AFFIRM ou GYPSY (ACM82) (WAS82).

Au fond, nous avons le sentiment que les techniques formelles n'améliorent pas et parfois même compliquent la conception des applications informatiques à "faibles fonctionnalités". Pour d'autres problèmes plus orientés bases de données, ou bases de connaissances ou même systèmes d'information, plusieurs voies de recherche sont actuellement étudiées. En plus de l'approche base de données relationnelle déjà discutée dans la première partie de notre travail, il existe trois types de propositions: les techniques de spécification algébrique (WEB78) (GOG79) (CHA82), les techniques utilisant le calcul des prédicats (REI77) (GAL78) (FDM82), enfin des techniques plus axées sur la représentation de connaissances comme les "frame" (MIN75) (TSU79) (LAU82).

L'existence de toutes ces approches montre que ce domaine d'application est aussi un domaine de recherche en pleine évolution. La première expérimentation significative du système VEGA (CHA82) effectuée dans le cadre du projet MONT-LOZERE (CHA82) nous laisse penser que l'intégration cohérente de plusieurs de ces techniques pourrait finalement aboutir à une approche "système expert" (MIC79).

Enfin les techniques formelles semblent pouvoir s'appliquer à la spécification et à la vérification de logiciels comme les éditeurs de textes (GUT80) (DAV80), de parties de systèmes d'exploitation: sécurité (ACM82) et protocoles de communication (BER82).

CHAPITRE VI. PRESENTATION DE LANGAGES (OU SYSTEMES) DE SPECIFICATION EXPERIMENTAUX.

I. SYSTEMES D'INFORMATION ET SYSTEMES DE BASES DE DONNEES.

I.1. Le projet ISDOS/PSL-PSA.

I.2. Le projet UDS2-SPECLE.

I.3. Le système SDC/INAJO.

I.4. Le système BETA.

II. SYSTEMES DE SPECIFICATION DE PROBLEMES "GENERAUX".

II.1. OBJ-O.

II.2. AFFIRM.

II.3. LPG.

III. CONCLUSIONS.

IV. OBJECTIFS ET HYPOTHESES DE TRAVAIL.

Nous nous proposons, dans cette partie, de présenter quelques traits de conception caractéristiques de systèmes de spécification expérimentaux se rapportant à deux grands domaines d'applications: les systèmes d'information et systèmes de bases de données, et la spécification de problèmes "généraux". Notre but n'est pas de faire un exposé exhaustif, car il existe beaucoup trop de propositions, mais d'essayer de montrer, à travers des systèmes plus ou moins représentatifs, les différentes façons d'aborder les fonctions de la spécification. Précisons que les systèmes dont nous discuterons dans la suite, ne sont sans doute pas meilleurs que d'autres, mais certainement mieux documentés!

I. SYSTEMES D'INFORMATION ET SYSTEMES DE BASES DE DONNEES.

En général, une spécification de problème base de données est une spécification d'univers, ou plus précisément d'une classe d'univers (PAI74). La spécification des énoncés (ou besoins) des utilisateurs est implicitement limitée par l'ensemble des expressions bien formées de la classe d'univers. Dans une spécification de système d'information (orienté base de données), l'analyse des problèmes conduit à expliciter un certain nombre d'énoncés à partir de l'univers déjà spécifié.

Cette distinction (ou même cette subtilité) n'est pas toujours facile à faire, surtout lorsque les systèmes de bases de données sont en plus des systèmes déductifs (GAL78).

D'un point de vue plus pratique, "un système d'information dans les organisations est un système conçu pour organiser, mémoriser, retrouver ou modifier des informations, selon les besoins propres de l'organisation" (TEI75).

I.1. Le projet ISDOS/PSL-PSA (TEI75).

Ce projet très ambitieux, développé à l'université de Michigan par TEICHROEW et al a pour objectif de proposer des concepts d'analyse simples, de fournir des outils d'aide à la conception et surtout au suivi de la documentation des projets informatiques.

En ISDOS, la spécification d'un problème comporte trois parties:

-- l'environnement du problème : cette section comporte une définition de tous les concepts de l'organisation auxquels il est fait plus ou moins explicitement référence dans le problème,

-- le système cible : c'est le système construit de façon incrémentale pour résoudre le problème,

-- le projet regroupe tout ce qui concerne la définition du problème, la documentation, la conception, l'implantation et l'installation du système d'information sur un ordinateur.

La technique de spécification du projet ISDOS consiste à définir de façon progressive et aussi systématique que possible une documentation "acceptable" par les utilisateurs de l'organisation. Chaque pas d'analyse-conception du problème se traduit systématiquement par l'enregistrement d'une description. De cette façon, il est clair que toutes les parties intéressées au projet: utilisateurs, concepteurs, administrateurs ont toutes facilités pour consulter, vérifier, modifier l'état en cours du projet.

Le système documentaire permet d'éditer plusieurs types de documents: des documents en langue naturelle, des diagrammes, des glossaires, des références croisées, etc..

Le système de spécification ISDOS comporte:

-- un langage de spécification appelé PSL (Problem Statement Language) utilisé dans les trois parties de la spécification d'un problème: environnement, système cible et projet,

-- un "traducteur" de PSL appelé PSA (Problem Statement Analyzer) chargé d'analyser les instructions PSL et ensuite de stocker le résultat dans une base de données,

-- une base de données qui sert uniquement à la mémorisation des résultats générés par PSA,

-- un générateur de rapports utilisé pour sélectionner, analyser et formatter les informations contenues dans la base de données,

-- un langage de commande pour manipuler de façon interactive la base de données: mise à jour, production de rapports, etc..

Il importe de préciser que PSL/PSA ne génère pas de code exécutable par une machine. Bien que le projet ISDOS soit de plus en plus référencé dans une certaine littérature, nous pensons qu'il est utile de montrer sur des exemples tirés de (TEI75) quelques traits du langage PSL. Nous n'entrerons pas dans une description détaillée de ces exemples extraits d'une spécification de problème "classique" de paie.

Les exemples 1 et 2 montrent l'utilisation de deux concepts de PSL: "PROCESS" et "ENTITY" dans la spécification initiale de la paie. L'exemple 3 précise le sous-problème "hourly-employee-processing" de l'exemple 1. Pour faciliter la lecture de ces exemples, nous donnons un sommaire des instructions PSL.

Un dernier aspect particulièrement intéressant de PSL/PSA concerne les contrôles de consistance et de complétude de la spécification. Toute entrée dans le système de spécifications PSL (adjonction, modification ou suppression) produit après analyse par PSA un contrôle automatique de cohérence avant stockage dans la base de données du projet.

Une spécification de problème est "complète" (cette notion n'a rien à voir avec la "complétude" des théories logiques) lorsque les informations stockées dans la base de données du projet permettent à tous les utilisateurs de l'organisation d'avoir une vue complète de leur problème; autrement dit lorsque ceux-ci n'éprouvent plus le besoin d'aller consulter d'autres sources d'information pour interpréter la spécification du problème. PSL/PSA apporte quelques aides pour

PROCESS INPUT/OUTPUT

[illegible] (COPY FILE)

FILE INPUT OUTPUT DESCRIPTION PROCEDURE MOVER-PAGE NOINDEX PRINT NOPUNCH

1* hourly-employee-processing

This process performs those actions needed to interpret time cards to produce a pay "item" for each hourly employee.

1. compute gross pay from time card.
2. compute tax from gross pay.
3. subtract tax from gross pay to obtain net pay.
4. update hourly employee record accordingly.
5. update department record accordingly.
6. generate paycheck.

Note: if status code specifies that the employee did not work that week, no processing will be done for that employee record.

*** INPUTS ***

1 time-card	RECEIVED
2 tax-rate	USED
3 hourly-employee-record	USED

*** OUTPUTS ***

1 pay-statement	GENERATED
2 error-listing	GENERATED
3 hourly-employee-report	GENERATED
4 error-listing	DERIVED
5 pay-statement	DERIVED
6 hourly-employee-report	DERIVED

Example 3

-40-

TABLE 6 PSL Statement Summary [1,2]

Section Type / Object Name in Statement	REAL WORLD ENTITY	INPUT	OUTPUT	SET	ENTITY	RELATION	GROUP	ELEMENT	PROCESS
REAL WORLD ENTITY	PART OF SUBPARTS ARE	GENERATES	RECEIVES	RESPONSIBLE FOR					
INPUT	GENERATED BY	PART OF SUBPARTS ARE	TO DERIVE[3]	CONTAINED IN TO DERIVE/UPDATE[3]	TO DERIVE/UPDATE[3]		CONSISTS OF TO DERIVE/UPDATE[3]	CONSISTS OF TO DERIVE/UPDATE[3]	RECEIVED BY USED BY
OUTPUT	RECEIVED BY	USING[3]	PART OF SUBPARTS ARE	CONTAINED IN USING[3]	USING[3]		CONSISTS OF USING[3]	CONSISTS OF USING[3]	GENERATED BY DERIVED BY
SET	RESPONSIBLE-ONE IS	CONSISTS OF USING[3]	CONSISTS OF TO DERIVE[3]	SUBSET OF SUBSETS ARE TO DERIVE/UPDATE USING[3]	CONSISTS OF TO DERIVE/UPDATE[3] USING[3]		TO DERIVE/UPDATE[3] USING[3] SUBSETTING-CRITERIA	TO DERIVE/UPDATE[3] USING[3] SUBSETTING-CRITERIA	USED BY UPDATED BY DERIVED BY
ENTITY		USING[3]	TO DERIVE[3]	CONTAINED IN USING[3] TO DERIVE/UPDATE[3]	USING[3] TO DERIVE/UPDATE[3] RELATED TO	VIA[3]	CONSISTS OF USING[3] TO DERIVE/UPDATE[3] IDENTIFIED BY	CONSISTS OF USING[3] TO DERIVE/UPDATE[3] IDENTIFIED BY	DERIVED BY UPDATED BY USED BY
RELATION					BETWEEN		ASSOCIATED-DATA	ASSOCIATED-DATA	MAINTAINED BY
GROUP		USING[3] CONTAINED IN	TO DERIVE[3] CONTAINED IN	USING[3] TO DERIVE/UPDATE[3] SUBSETTING-CRITERION	USING[3] TO DERIVE/UPDATE[3] IDENTIFIER CONTAINED IN	ASSOCIATED WITH	USING[3] TO DERIVE/UPDATE[3] CONSISTS OF CONTAINED IN	USING[3] TO DERIVE/UPDATE[3] CONSISTS OF	DERIVED BY UPDATED BY USED BY
ELEMENT		USING[3] CONTAINED IN	TO DERIVE[3] CONTAINED IN	USING[3] TO DERIVE/UPDATE[3] SUBSETTING-CRITERION	USING[3] TO DERIVE/UPDATE[3] IDENTIFIES CONTAINED IN	ASSOCIATED WITH	USING[3] TO DERIVE/UPDATE[3] CONTAINED IN	USING[3] TO DERIVE/UPDATE[3]	DERIVED BY UPDATED BY USED BY
PROCESS		USING[3] RECEIVES	DERIVES TO DELETE[3] GENERATES	DERIVES FILL UPDATES USING[3] TO DERIVE/UPDATE[3]	DERIVES USES UPDATES USING[3] TO DERIVE/UPDATE[3]	MAINTAINS	DERIVES USES UPDATES USING[3] TO DERIVE/UPDATE[3]	DERIVES USES UPDATES USING[3] TO DERIVE/UPDATE[3]	PART OF SUBPARTS UTILIZES UTILIZED BY PRECEDENCE[3]
EVENT									ON INCEPTION ON TERMINATION TRIGGERS
CONDITION									
INTERVAL									
SYSTEM PARAMETER									

[1] The statements in the body of the table may be used in a Problem Statement Section of the type designated by the row. The statement may name objects of the type designated by the row in which the statement appears.

[2] In addition the following (defined in the DEFINE SECTION) apply to almost every Section: ATTRIBUTES ARE, DESCRIPTION, KEYWORDS, RESPONSIBLE-PERSON PERSONS, SECURITY, SYNONYM, SYSTEM

[3] [illegible]

FORMATTED PROBLEM STATEMENT

PARAMETERS FOR: PPS

NAME=payroll-processing NOINDEX PRINT NOPUNCH SMARG=5 NMARG=20 AMARG=10 BMARG=25 RMMARG=70
CMARG=1 HMARG=40 DESG ONE-PER-LINE DEFINE COMMENT NONEW-PAGE NONEW-LINE

```
 1 PROCESS                                   payroll-processing;
 2     SYNONYMS ARE:  payproc,
 3                    p1;
 4     DESCRIPTION;
 5           This process represents the highest level process
 6           in the target system.  it accepts and processes
 7           all inputs and produces all outputs.;
 8     SEE-MEMO:      goals-memo,
 9                    process-memo;
10     SUBPARTS ARE:  new-employee-processing,
11                    terminating-emp-processing,
12                    hourly-employee-processing,
13                    salaried-employee-processing,
14                    shared-routines;
15     RECEIVES:      employee-information;
16     GENERATES:     paysystem-outputs;
17     DERIVES:       paysystem-outputs;
18     UPDATES:       payroll-master-information;
19     USES:          employee-information,
20                    payroll-master-information;
21     RESPONSIBLE-PROBLEM-DEFINER IS:
22                    michael-j-bastarache;
23
24 EOF EOF EOF EOF EOF
```

Example 1

PSA VERSION 2.0.6 PSA-EXAMPLE AUG 1, 1975 20:33:42 PAGE

FORMATTED PROBLEM STATEMENT

PARAMETERS FOR: PPS

FILE NOINDEX PRINT NOPUNCH SMARG=5 NMARG=20 AMARG=10 BMARG=25 RMMARG=70 CMARG=1 HMARG=40 DESG
ONE-PER-LINE DEFINE COMMENT NONEW-PAGE NONEW-LINE

```
 1 ENTITY                                    department-record;
 2     SYNONYMS ARE:  dept-rec;
 3     DESCRIPTION;
 4           This record holds all current data relevent about each
 5           department;
 6     CONTAINED IN:  department-file,
 7                    payroll-master-information;
 8     CONSISTS OF:
 9                         department,
10           2             supervisor,
11                         number-of-employees,
12                         total-budget,
13                         remaining-funds;
14     /* LEFT */ RELATED TO:
15          hourly-employee-record
16           VIA           dept-hourly-emp-relation;
17     /* LEFT */ RELATED TO:
18          salaried-employee-record
19           VIA           dept-salaried-emp-relation;
20     IDENTIFIED BY: department;
21     OCCURRENCES:   no-of-departments;
22
23 ENTITY                                    hourly-employee-record;
24     SYNONYMS ARE:  h-emp-rec;
25     DESCRIPTION;
26           This record holds all current data about each hourly employee;
27     CONTAINED IN:  hourly-employee-file ,
28                    payroll-master-information;
29     CONSISTS OF:
30                         employee-name,
31                         employee-identification-number,
32                         social-security-number,
33                         pay-grade-code,
34                         address,
35                         phone ,
36                         [illegible]
```

déterminer certains types d'incomplétude.

En résumé, le projet ISDOS intègre toutes les fonctions d'un système de spécification avec une composante essentielle en plus: la documentation du suivi de projet.

I.2. Le système UDS2-SPECLE (BIG79).

UDS2 (The Unified Design Specification System) est un système prototype développé par Boeing Computer Service Company pour améliorer la conception et la production de très gros systèmes logiciels. Le principal objectif visé est donc de fournir un ensemble de techniques et d'outils d'aide à la conception et qui s'adaptent facilement à un environnement de production de logiciel. Actuellement, UDS2 est essentiellement composé d'un langage de spécification noyau appelé SPECLE, d'un système type base de données utilisé pour analyser et contrôler les spécifications SPECLE et produire automatiquement la documentation qui est ensuite gérée par le système. Des études sont actuellement menées pour essayer d'étendre UDS2 avec PSL/PSA. Le langage SPECLE est un langage déclaratif, modulaire qui supporte une technique de conception structurée, hiérarchique. UDS2 utilise, comme dans le projet CIVA (DER74) la possibilité d'étendre les instructions du langage de spécification de base avec le méta-langage SPECLE.

En UDS2, la conception initiale de la spécification d'un problème s'effectue par décomposition et raffinements progressifs des descriptions rédigées en langage naturel. Le processus se poursuit en permettant d'introduire des instructions d'un langage de programmation (FORTRAN ou COBOL), des spécifications intermédiaires en SPECLE et/ou en langage naturel.

En bref, UDS2 est une approche fondamentalement basée sur les concepts, les techniques et les outils de la programmation modulaire (ou structurée) CIVA (DER74). L'originalité de ce système nous paraît essentiellement résider dans la production automatique de la documentation associée à la spécification. Toutefois, il n'est pas possible, comme en ISDOS, de bien séparer la documentation propre à l'environnement de spécification initiale du problème, de celle qui à différentes étapes de la conception comporte des références à des concepts de langages de programmation. De ce fait, les membres de l'organisation n'ont plus les mêmes facilités, comparativement à ISDOS, pour contrôler le processus de conception.

La plupart des travaux sur les systèmes d'information (INF76) (BOD81), REMORA (FOU82), GALION (FLO75), LAPAGE (CAV79), etc... proposent des approches qui vont dans le sens des idées que nous venons de mettre en exergue dans la présentation d'ISDOS et d'UDS2.

I.3. Le système SDC/INAJO (BER81).

System Development Corporation (SDC, Santa Monica, Californie) travaille depuis quatre ou cinq ans sur un important projet de système de spécification et de vérification de systèmes orientés bases de données. Un des objectifs de SDC est de fournir un ensemble cohérent de techniques et d'outils d'aide: à la spécification de problèmes, à la vérification de la spécification, à la spécification de la conception de programmes et à la vérification de leur implantation.

L'environnement de SDC comporte un langage de spécification formelle appelé INAJO, un prouveur de théorèmes interactif et un générateur de vérification de conditions pour un sous-ensemble de MODULA (WIR76).

Le langage de spécification INAJO est un langage déclaratif, basé sur le calcul des prédicats du 1er ordre, avec en plus, plusieurs traits des langages de programmation de haut niveau, comme par exemple des types de données: booléen, entier, types énumérés, structurés. Au demeurant, INAJO est un langage statique.

Une spécification SDC est une description INAJO relativement complexe qui regroupe une spécification fonctionnelle de la conception avec une spécification de divers critères de correction: invariants, "criterions", "constraints". Le traitement d'une telle spécification génère un certain nombre de conditions affirmant que la spécification fonctionnelle satisfait bien les critères de correction spécifiés. Toutes ces conditions doivent ensuite être prouvées à l'aide du

prouveur de théorèmes. On démontre ainsi que ces formules logiques sont des théorèmes.

La technique de spécification par raffinements successifs qui consiste à inclure de plus en plus de détails de moins en moins abstraits, supportée par INAJO fournit une spécification hiérarchique. Le premier niveau de spécification est une représentation fonctionnelle d'une machine d'état abstraite avec des définitions de critères de correction appelés "criterions" et "constraints". Les "criterions" spécifient les conditions qui doivent être vérifiées par tous les états de la machine abstraite alors que les "constraints" sont des conditions associées aux changements d'état. La machine d'état abstraite peut se caractériser comme un ensemble de variables d'état et un ensemble de transformations.

Les spécifications d'implantation ne sont définies qu'au plus bas niveau des spécifications SDC. Elles comportent un ensemble de déclarations, de structures de données des programmes, de noms de modules-programmes, d'entêtes de procédures et/ou de fonctions et produisent toutes les connexions entre les entités des spécifications abstraites et les éléments (variables, constantes, paramètres et types) du programme. A noter que SDC génère automatiquement toutes les assertions d'entrée et de sortie des procédures.

La dernière étape est une étape de vérification qui nécessite l'utilisation d'un générateur de vérification de conditions (VCG). Le VCG dépend évidemment du langage de programmation cible qui est utilisé, ici MODULA (WIR76). Les programmes codés selon des normes très précises sont alors soumis au VCG qui génère les conditions affirmant que le code satisfait les assertions de sortie s'il satisfait aussi les assertions d'entrée. Il ne reste plus qu'à prouver chacune de ces conditions à l'aide du prouveur du système.

Cette brève présentation de SDC montre que ce système expérimental, en cours de développement est très novateur dans le domaine des bases de données. Utilisé comme système de spécification et de vérification de l'intégrité d'une base de données, SDC nous semble présenter plusieurs avantages intéressants:

-- une technique de spécification précise, structurée, indépendante des modèles de données conventionnels (hiérarchiques, réseaux ou relationnels) et surtout d'une grande généralité pour exprimer les contraintes d'intégrité des bases de données,

-- des spécifications en INAJO peuvent aussi bien servir d'aide conceptuelle pour la manipulation des bases de données, pour des utilisateurs "experts" (toutefois), et d'aide conceptuelle à l'implantation de la base de données dans un environnement de programmation (modulaire) ou avec un système de gestion de base de données.

Enfin, si la base de données est implantée conformément aux spécifications abstraites et s'il n'y a pas de problème de panne d'ordinateur (!), alors on peut avoir l'assurance que la base de données satisfait bien les contraintes d'intégrité voulues.

I.4. Le système BETA (BRO78).

Le système de spécification et de vérification de bases de données BETA développé à l'université de TORONTO par M.L.BRODIE est un des outils supporté par une méthode de conception et d'implantation de bases de données appelée ACM/PCM (BRO82).

Le langage de spécification BETA est un langage assertionnel qui intègre des concepts des langages de programmation de haut niveau: les types de données au sens PASCAL, des concepts d'intelligence artificielle: aggrégation, généralisation (SMI77), association, etc..., ainsi que des notions du modèle de données de E.F.CODD (COD71) comme les opérations de l'algèbre relationnelle. BETA est utilisé comme un outil d'aide à la spécification et à la vérification de schémas de bases de données. Axiomatisé avec la technique de HOARE (HOA72), (LIV70), BETA permet de vérifier un certain nombre de propriétés (formules logiques, opérations relationnelles ou assertions-formules logiques quantifiées) des spécifications. M.L.BRODIE décrit dans (BRO78) un méta-algorithme de vérification de spécifications de bases de données. Il comporte trois étapes que nous résumons brièvement:

-- une "analyse textuelle" des spécifications: analyse syntaxique avec contrôles statiques des instructions BETA qui génèrent un certain nombre d'assertions,

-- cette analyse se poursuit pour les assertions et formules logiques de la spécification mais ici une vérification est effectuée,

-- enfin, le système vérifie que les assertions générées par la première étape satisfont les propriétés de la spécification analysées dans la deuxième étape.

BETA est sans aucun doute un langage de spécification beaucoup plus compréhensible syntaxiquement qu'INAJO. On peut toutefois regretter que ce langage ne supporte pas de technique de structuration, car on s'aperçoit, en analysant les exemples de spécification publiés (BRO78) (BRO82), d'abord qu'il n'est pas toujours très facile d'établir des liens logiques entre les différentes descriptions et ensuite que le procédé de vérification devient extrémement complexe, d'autant qu'il faut éventuellement compléter les assertions générées par le système!

D'autres recherches du domaine des bases de données déductives utilisent des concepts (types de données) des langages formels (sous-ensemble du calcul des prédicats du ler ordre) et des techniques (modularité, parfois abstraction) proches de ceux de la spécification (REI81) (GAL78) (WAN77). Ces travaux visent tous le même but: concevoir un langage de programmtion de très haut niveau qui ait à la fois les caractéristiques d'un langage de spécification formel, d'un langage d'utilisation et de mise à jour (PAI76). Les langages basés sur le formalisme des clauses de Horn, comme PROLOG (ROU75) (WAR77) semblent particulièrement bien adaptés à ce type d'approche (GAL78) (REI81).

Certains essaient même de montrer, à partir d'exemples tirés de l'approche de spécification algébrique (DAV80), qu'un langage de type PROLOG est un outil de spécification tout à fait comparable aux outils algébriques (GUT80) et même certainement plus approprié pour tester la conception logique des problèmes.

II. SYSTEMES DE SPECIFICATION DE PROBLEMES "GENERAUX".

Un grand nombre de propositions de spécification dont le principal domaine d'intérêt est la spécification et la vérification de programmes se retrouvent plus ou moins classées dans ce thème: HDM (ACM82), GYPSY (ACM82), OASIS (BAR82), SPRAC (FOI82), etc...

Pratiquement, tous les langages de programmation de très haut niveau (applicatifs ou fonctionnels), à "très large domaine d'application" sont très souvent assimilés à des langages de spécification de problèmes généraux: LUCID (ASH77), MEDEE (FIN79), ALICE (LAU76), SETL (SCH75), Z (ABR78), etc...

Plutôt que de faire une étude, même partielle, de ces différents travaux, nous nous sommes limités à trois types de propositions qui concernent plus directement notre travail: OBJ, AFFIRM et LPG.

II.1. OBJ-0 (GOG79).

OBJ-0 est un langage expérimental de spécification algébrique développé à UCLA par J.TARDO (GOG79) selon les idées de J.GOGUEN (GOG78).

OBJ-0 est un langage de spécification formelle qui supporte les techniques de décomposition et de raffinements successifs de spécifications algèbriques. C'est aussi un langage de programmation abstraite avec des caractéristiques syntaxiques et sémantiques très intéressantes, aussi bien pour écrire que pour tester ou mettre au point des spécifications complexes.

La technique de spécification supporté par OBJ conduit à décrire des unités modulaires appelées "objets" (OBJ). Le fait qu'en OBJ il soit possible de définir des expressions avec des opérateurs préfixés, infixés, postfixés ou plus généralement "distributed-fix", d'utiliser la surcharge d'opérateurs, ainsi qu'une syntaxe homogène pour définir les cas d'exception, etc... explique suffisamment pourquoi les spécifications en OBJ paraissent souvent claires, expressives et "complètes" (au sens où elles rendent aussi compte des cas d'exception).

Une fois décrites, les spécifications peuvent être testées, mises au point en procédant à l'évaluation symbolique de plans de tests sur leur modèle sémantique associé qui, dans l'approche algébrique de OBJ, est toujours l'algèbre initiale. Le choix de l'algèbre initiale comme

modèle d'une spécification est ici fondamental. En effet, la théorie montre que ce modèle existe toujours (ADJ78). Par conséquent, pour s'assurer qu'une spécification est correcte, il suffit de vérifier qu'elle est consistante avec l'algèbre initiale correspondante. Maintenant, pour exécuter les plans de tests, autrement dit, pour évaluer des expressions de termes, OBJ utilise des mécanismes de réécriture de termes. Les équations décrites dans les spécifications étant implantées comme des règles de réécriture, la technique d'évaluation consiste à réduire ces expressions de termes en appliquant les règles de réécriture selon une stratégie "au plus profond d'abord et de la gauche vers la droite" (HUL80). Cette technique est très voisine de celle qui est habituellement employée pour évaluer des expressions LISP: l'appel par valeur. Ajoutons qu'OBJ fournit une stratégie d'évaluation optionnelle pour des équations qui contiennent des opérateurs commutatifs.

Nous allons maintenant décrire un exemple de spécification simple (une pile d'entiers avec erreurs!) pour illustrer quelques-uns des traits que nous venons d'exposer très brièvement.

```
OBJ STACK-OF-INT
SORTS STACK/INT BOOL
OK-OPS
   PUSH:INT STACK -> STACK
   POP_:STACK -> STACK
   TOP_:STACK -> INT
   BOTTOM: -> STACK
   EMPTY?_:STACK -> BOOL
```



```
ERR-OPS
   UNDERFLOW: -> STACK
   TOPL: -> INT
VARS
   I:INT
   S:STACK
OK-EQNS
   (POP PUSH(I,S) = S)
   (TOP PUSH(I,S) = I)
   (EMPTY? BOTTOM = T)
   (EMPTY? PUSH(I,S) = F)
ERR-EQNS
   (TOP BOTTOM = TOPL)
   (POP BOTTOM = UNDERFLOW)
JBO
```

L'environnement de base de OBJ-0 ne comporte que trois objets prédéfinis: INT (pour les entiers), BOOL (pour les booléens) et ID (pour les identificateurs) avec les opérations traditionnelles pour ces types d'objets mais sans équation. La spécification d'un objet débute par le mot-clé OBJ suivi du nom de l'objet, ici, STACK-OF-INT et se termine par le mot-clé JBO. La description proprement dite comporte deux parties: la première appelée signature déclare les sortes (SORTS), les opérateurs (OK-OPS, ERR-OPS) , la seconde qui définit la sémantique de l'objet regroupe des déclarations de variables (VARS) utilisées dans les OK-EQNS et les ERR-EQNS.

Souvent, dans les langages de spécification algébrique: AFFIRM (MUS80), OASIS (BAR82), etc.. les seules notations syntaxiques qu'il soit possible d'utiliser sont celles des mathématiques, à savoir une notation préfixée et parenthésée pour les opérateurs. Si cette notation existe aussi en OBJ-0 (PUSH), beaucoup d'autres formes syntaxiques, plus expressives, peuvent être utilisées (POP, TOP, EMPTY?).

Une expression soumise à évaluation se présente sous la forme suivante:

RUN PUSH(3,POP PUSH(2,BOTTOM)) NUR

et la réponse de OBJ est:

AS STACK:PUSH(3,BOTTOM)

OBJ-0 indique systématiquement le type de la valeur ou de l'expression en forme normale obtenue après réduction. Si maintenant on demande d'évaluer l'expression suivante:

RUN PUSH(TOP BOTTOM,POP PUSH(3,PUSH(2,BOTTOM))) NUR

la réponse est alors:

AS STACK:>>ERROR>> PUSH(TOPL,PUSH(2,BOTTOM))

Ici, le résultat est un terme erreur du type STACK; ce qu'indique le message >>ERROR>> précédant le terme PUSH(TOPL,PUSH(2,BOTTOM)). Notons que la technique d'évaluation utilisée, essaie de réduire au maximum toutes les sous-expressions qui composent le terme. C'est le cas de la sous-expression POP PUSH(3,PUSH(2,BOTTOM)) qui a pu être réduite en un "OK-terme" PUSH(2,BOTTOM) malgré la détection d'une erreur sur la sous-expression TOP BOTTOM.

Dans cet exposé de synthèse, nous n'entrerons pas dans plus de détails et nous n'aborderons pas non plus d'autres traits tout aussi intéressants d'OBJ-0: la coercion de types, les opérations cachées ou encore les expressions conditionnelles. Plusieurs de ces points seront discutés lorsque nous présenterons nos propositions.

Signalons encore que J.GOGUEN a publié un article théorique sur la paramétrisation, les sous-types avec erreurs (GOG78), mais qu'à notre connaissance, ces idées n'ont pas encore été incorporées dans le langage OBJ.

Actuellement, OBJ-0 a été expérimenté dans plusieurs domaines d'application, avec un certain succès: la spécification d'un système de manipulation de fichiers à clés, la spécification d'un système de base de données concernant la gestion d'un aéroport. L'article de J.GOGUEN (GOG79) fait un résumé de tous les enseignements qui ont ainsi pu être tirés de cette série d'expérimentations. Le lecteur intéressé peut s'y reporter pour obtenir plus d'informations.

II.2. AFFIRM (MUS80).

Développé par D.MUSSER dans le cadre d'un projet de recherche de l'USC Information Sciences Institute, AFFIRM est un système interactif de spécification et de vérification qui fait suite à deux autres systèmes: Xivus System et Data Type Verification System DTVS. Une des caractéristiques d'AFFIRM est de pouvoir accepter en entrée des descriptions d'abstraction de données et de programmes écrites soit avec un langage de spécification formelle (algébrique), soit encore avec un langage de programmation, une version dérivée de PASCAL-EUCLID, en l'occurrence.

En AFFIRM, la technique de spécification d'un problème consiste à définir une première spécification abstraite regroupant des descriptions algébriques de types de données et des propriétés (axiomes d'induction), puis on développe un raffinement progressif de chacune de ces abstractions de données. Finalement, on obtient une spécification formelle structurée.

Dans cette approche, la spécification d'un type abstrait de données passe systématiquement par une vérification de la correction: consistance et suffisante-complétude . En effet, un type abstrait de données ne peut être entré dans un contexte de spécifications gérées par une base de données, que si le spécifieur a pu, auparavant, vérifier sa correction à l'aide du prouveur de théorèmes du système.

Le plus bas niveau de spécification, parfois qualifié de définitionnel (ACM82), revient en fait à définir une spécification "constructive" (FIN79). On obtient de cette façon une définition explicite de chacune des opérations du type abstrait de données. Le spécifieur doit ensuite vérifier à l'aide du prouveur de théorèmes que chacune de ces définitions satisfait les assertions précédemment établies par une base de données.

Le passage de ce niveau de spécification définitionnelle au programme proprement dit, se fait de façon manuelle. Le principe du codage d'un type abstrait en PASCAL-AFFIRM consiste essentiellement à implanter les constructeurs par des procédures et les fonctions d'observation par des fonctions PASCAL. Diverses conditions de vérification peuvent aussi être insérées dans le code du programme: pré et/ou post-conditions, invariants de boucles, WHILE, FOR, REPEAT ainsi que des clauses d'assertion pour les instructions GOTO et RETURN.

La vérification de la correction du programme produit nécessite l'utilisation du générateur de vérification de conditions du système.

Ce générateur de vérification de conditions comporte un ensemble de règles axiomatiques qui définissent la sémantique du langage en termes de transformation de prédicats. Les règles de substitution utilisées sont les règles de preuve d'EUCLID.

Nous ne présenterons pas le prouveur de théorèmes qui pourtant est un des plus importants composants du système AFFIRM. Une discussion détaillée de cet outil nous ferait très largement sortir de notre cadre de travail. Nous dirons seulement que ce prouveur de théorèmes est un système interactif de déduction naturelle qui opère sans stratégie de preuve particulière sur des expressions du calcul des prédicats du ler ordre. Cet outil est utilisé pour prouver des propriétés des types abstraits de données et les conditions de vérification des programmes. Cependant, il semble que seuls des "experts confirmés" soient capables de manipuler ce type d'outil (GER79). Un des aspects les plus critiques semble toutefois se situer au niveau du contrôle de l'expansion des résultats intermédiaires du prouveur.

Parmi les différentes expérimentations qui ont été menées à ISI à l'aide du système AFFIRM d'aide à la spécification algébrique de types abstraits de données et à la vérification de leurs propriétés, citons:

.DELTA (GER79): cette application AFFIRM s'intéresse à la spécification de la manipulation de messages militaires et à la vérification de l'ordre des mises à jour des messages,

.plusieurs travaux sur la spécification et la vérification de protocoles de communication (BER82).

Notons que la plupart des chercheurs qui ont participé à ces expérimentations aboutissent souvent aux mêmes conclusions:

."l'utilisation des méthodes algébriques pour des applications non triviales est difficilement concevable sans un outil logiciel facilitant leur mise en oeuvre",

.le langage de spécification algébrique devrait permettre d'exprimer des spécifications plus expressives et plus structurées,

.la possibilité de définir des types paramétrés ne pourrait qu'améliorer le confort d'utilisation du système.

Nous ajouterons encore deux remarques concernant le codage et la preuve des programmes PASCAL-AFFIRM. D'abord, la correspondance entre les concepts de la spécification définitionnelle et les concepts du programme n'étant pas toujours explicite, c'est une source d'erreur importante. De plus, comme il n'existe pas de compilateur pour cette version de PASCAL, les programmes vérifiés doivent être recodés dans une autre version ou un autre langage, ce qui enlève malheureusement beaucoup de crédibilité à la vérification.

II.3. LPG (BER81).

LPG (Langage pour la Programmation Générique) développé à l'IMAG par D.BERT est un langage applicatif de spécification et de programmation. Cet outil est conçu pour être, par la suite, intégré dans un système expérimental de spécification et de programmation, l'objectif visé étant de tester l'applicabilité des notions de spécification algébrique de types abstraits et de généricité des programmes.

Les principaux traits caractéristiques de LPG sont les types abstraits génériques et les propriétés. Un type abstrait générique, dans cette approche, est un type algébrique paramétré par d'autres types appelés types formels. Ces types formels sont caractérisés par un ensemble minimum de fonctions que devront posséder les types effectifs au moment de l'instanciation du type générique. Cet ensemble de fonctions appelé propriété sera spécifié par des axiomes équationnels.

LPG supporte, de la même manière, la spécification de fonctions génériques. Dans ce cas, les fonctions qui caractérisent les types formels sont des fonctions formelles qui pourront être utilisées dans le corps des fonctions génériques.

Notons que LPG autorise la surcharge des fonctions et des opérateurs.

Enfin, un dernier aspect intéressant a trait à la définition des exceptions. La spécification de cas d'exception conduit à définir des noms d'exception (éventuellement paramétrés) en les déclarant comme des fonctions à résultat dans "exc" , et à "préconditionner" un peu à la manière d'ADA (ADA80) les axiomes équationnels des expressions du type quand nom-d-exception alors fq ou si alors => nom-d-exception fsi .

Ce langage, en cours d'implantation, possède plusieurs traits de conception originaux qui pourraient fournir des aides à la spécification algébrique de types abstraits mais par ailleurs risquent, nous semble-t-il, d'augmenter la complexité de la spécification,

notamment pour tout ce qui concerne le raffinement des propriétés. En fait, seule l'expérimentation permettra de savoir si ces notions facilitent le développement de spécifications algébriques, pour des applications non triviales, cela s'entend.

III. CONCLUSIONS.

Nous avons essayé de montrer dans cette brève et incomplète étude, que s'il existait des systèmes expérimentaux intéressants d'aide à la spécification de problèmes ou d'aide à la spécification formelle et la la vérification de programmes, très peu étaient adaptés à la spécification de problèmes, dans le sens où ils ne satisfont que partiellement les différentes fonctions de spécification qui ont été caractérisées au chapitre V.

En fait, cette présentation reflète très mal l'important travail d'investigation que nous avons pu faire sur des études, des propositions de langages, des réalisations de systèmes expérimentaux et que nous n'avons fait que citer en référence. C'est par exemple le cas de MEDEE (BEL78) MAIDAY (GUY82) qui fait l'objet de recherches intensives au sein même du CRIN.

Dans cet exposé de synthèse, nous nous sommes surtout appliqué à extraire de plusieurs travaux récents, les formalismes, les techniques et les outils qui nous paraissaient le mieux correspondre à notre approche du problème de la spécification.

Le lecteur intéressé par des synthèses moins orientées que la nôtre, pourra trouver des présentations d'autres approches dans (FIN79) (ACM82).

IV. OBJECTIFS ET HYPOTHESES DE TRAVAIL.

Etudier et dégager des concepts, des techniques et des caractéristiques d'un langage qui répondent le mieux possible aux besoins en logiciel d'aide à la spécification et à la validation d'une classe donnée de problèmes; ensuite concevoir et implanter les mécanismes et les outils qui les mettent en oeuvre et enfin, procéder à une validation systématique des choix de conception effectués par l'expérimentation est une démarche typique du domaine du génie logiciel que nous avons tenu à préserver dans la présentation de cette partie de notre travail. C'est d'ailleurs cette même démarche qui nous a aussi servi de guide pour charpenter la présentation de notre projet.

Les objectifs visés sont étroitement liés à ceux que nous nous sommes fixés pour l'ensemble du projet; à savoir, fournir des concepts, des techniques et des outils d'aide à la spécification et à la validation de la conceptions qui s'intègrent de façon cohérente à un atelier de conception et de production de logiciels fiables.

Les idées de base qui ont ici présidé à la conception du système interactif d'aide à la spécification algébrique de types abstraits et de programmes et à la validation des propriétés que nous avons appelé VEGA, sont très voisines de celles qui ont été développées pour le système interactif de programmation modulaire présenté dans la troisième partie; à savoir, le principe d'abstraction, le concept de type abstrait de données et les mécanismes de modularité.

VEGA est un système interactif "multicibles" qui comporte deux systèmes:

-- un système d'aide à la spécification algébrique de types abstraits et de programmes et d'aide à la validation de leurs propriétés, le langage formel étant appelé VEGA,

-- et un système d'aide à la programmation logique de maquettes défini à partir du langage PROLOG (ROU75).

Cette dualité nous semble pouvoir apporter quelques aides à la spécification et à la programmation particulièrement intéressantes. En effet, le système VEGA fournit un langage algébrique de spécification et de programmation: VEGA et une extension de PROLOG, le langage de programmation logique utilisant un environnement de types abstraits de données et/ou de programmes spécifiés algébriquement.

Cette dernière possibilité qui, dans notre approche, permet de définir des interfaces logiques des systèmes algébriques, et de façon plus générale, des maquettes de tests.

Pourquoi avons nous choisi PROLOG comme langage cible?

En fait, lorsqu'il y a un peu plus de deux ans, nous nous sommes préoccupés de trouver un langage orienté intelligence artificielle disponible sur un IRIS-80 V10, nous n'avions que le choix entre un V-LISP et un PROLOG. Les essais de V-LISP n'étant pas très convaincants, nous avons donc décidé d'exercer notre sagacité sur le PROLOG de Marseille (ROU75) qui, malgré ses insuffisances, possède un certain nombre de possibilités dont nous avons pu apprécier l'intérêt ou même l'utilité en réalisant le système VEGA. Actuellement, il existe d'autres versions de PROLOG plus opérationnelles que celle de 75: PROLOG II (COL82), etc...

Pratiquement tous les langages (ou systèmes) expérimentaux de spécification formelle dont nous avons pu avoir connaissance ne supportent qu'une approche de spécification. La seule exception pour ainsi dire est le système AFFIRM que nous avons présenté en VI.2.2. Nous croyons que c'est tout à fait révélateur de l'usage actuel des systèmes de spécification qui, répétons le, sont essentiellement conçus pour spécifier et vérifier des programmes et non des problèmes au sens où nous l'entendons. En ce sens, nous reprenons à notre compte le point de vue de J.P.FINANCE (FIN79 p31,73): il nous semble évident qu'une spécification formelle ne possédera les qualités que l'on en attend: fidélité, compréhensibilité, etc... que si la technique de spécification employée est "suffisamment appropriée". C'est une question qui a déjà été très longuement débattue par les spécialistes de la conception des bases de données (FRE76). Tout le monde semble maintenant convenir qu'il n'y a pas de modèle universel dont on puisse affirmer qu'il est supérieur à tous les autres, cela dans un très large

domaine d'application. Par contre, si l'on restreint le champ des applications à une classe de problèmes et que par ailleurs, on ait à choisir entre plusieurs techniques de modélisation, la question de savoir s'il existe une technique plus appropriée qu'une autre devient primordiale.

Lorsque nous avons conçu le langage de spécification formelle VEGA, nous avons eu le souci de tenir le plus possible compte de ce type de réflexion. En effet, VEGA supporte deux des principales techniques de spécification formelle du domaine de la méthodologie de la programmation (LIS77).

- Avec la première technique, le modèle de la spécification est implicite et la caractérisation des opérations est définie par des axiomes équationnels. C'est donc une approche algébrique au sens de (GOG78).

- Avec la deuxième technique, la spécification s'effectue sur un modèle abstrait (ensemble, séquence, etc...). C'est une approche de spécification à la ALPHARD (WUL75). Cependant, en VEGA, les propriétés des opérations sont caractérisées par des axiomes équationnels alors que dans (WUL75) (FIN79), cette caractérisation est généralement définie à partir d'expressions du calcul des prédicats du 1er ordre.

Ces choix de conception sont fondamentaux. D'abord, parce que nous pensons que ces deux techniques de spécification formelle sont souvent complémentaires, notamment pour la classe de problèmes de notre domaine d'intérêt et qu'en plus, les descriptions des spécifications obtenues à l'aide de ces deux techniques n'utilisent qu'un seul formalisme, ce qui

présente beaucoup d'avantages. Maintenant, pour que les spécifications algébriques obtenues à l'aide de ces techniques encapsulent effectivement toute la sémantique des concepts des problèmes, il est essentiel que l'on puisse aussi spécifier le comportement des opérations dans des états exceptionnels. Autrement dit, les spécifications algébriques VEGA devront systématiquement inclure les erreurs abstraites des problèmes.

Il importe encore de préciser que VEGA a essentiellement été conçu pour permettre de faire des validations de spécification par tests . Quelques traits syntaxiques de VEGA apportent quelques aides à l'écriture de spécifications "suffisamment complètes", mais actuellement, nous n'avons pas d'outil d'aide à la vérification des propriétés des spécifications comme dans AFFIRM (MUS80), SDC (BER81).

Avant de présenter VEGA, il nous semble indispensable d'exposer l'essentiel des définitions et résultats théoriques concernant la théorie des types abstraits algébriques avec erreurs, les systèmes de réécriture, et ensuite la théorie logique du langage PROLOG dont nous décrivons les principaux traits de conception.

CHAPITRE VII. TYPES ABSTRAITS ALGEBRIQUES AVEC ERREURS ET SYSTEMES DE REECRITURE.

INTRODUCTION.

I. ALGEBRES HETEROGENES.

I.1. Σ-algèbres.

I.2. Σ-homomorphismes, algèbres initiales.

I.3. Equations.

I.4. Exemples.

I.5. Correction des spécifications.

I.5.1. Algèbres canoniques.

I.5.2. Consistance et complétude des spécifications algèbriques.

I.6. Spécifications algébriques avec "erreurs".

I.6.1. Présentation du problème des erreurs sur un exemple.

I.6.2. Spécification avec cas d'"exceptions et d'échecs".

I.6.3. Spécification avec propagation des erreurs.

I.6.4. Les algèbres avec erreurs.

I.7. Spécification des types abstraits paramétrés.

I.8. Implantation des types abstraits.

I.8.1. Implantation abstraite des spécifications avec erreurs.

I.8.2. Implantation abstraite des spécifications selon GUTTAG.

I.8.3. Implantation directe des spécifications abstraites.

II. LES SYSTEMES DE REECRITURE.

II.1. Rappels de notions de base.

II.2. Systèmes de réécriture d'expressions.

II.3. Propriétés des systèmes de réécriture.

II.4. Principe de définition.

II.5. Notion de surréduction.

INTRODUCTION.

Le but de cette section est de présenter l'essentiel des définitions et résultats théoriques qui ont une incidence sur la sémantique déclarative ou opérationnelle (LIV78) des langages formels du système VEGA. Nous commençons par développer au chapitre VII le concept de type abstrait algébrique avec erreur ainsi que des notions de base sur les systèmes de réécriture. Pour cette étude, nous nous sommes permis de faire quelques emprunts aux travaux de ADJ (ADJ78), GHM (GHM78), (GAU80) (HUL80) (KK82). Le chapitre VIII est dévolu à quelques rappels de notions de logique et de résolution (ROB65). Nous terminons par une description rapide du langage de programmation logique PROLOG (ROU75) (KUW77) et de la mise en oeuvre de sa méthode de résolution. Cette section a également pour objet de fixer les notations et de préciser la terminologie employée dans la suite de cette thèse. Pour les aspects théoriques, nous ne donnons aucune démonstration. Le lecteur intéressé par ces détails techniques devra donc se reporter aux références citées ci-dessus.

L'abstraction et la modularité sont deux concepts importants de ce chapitre. Dans la première partie de notre travail, nous avons vu qu'ils étaient aujourd'hui supportés par la plupart des langages de programmation modernes. On a pu aussi se rendre compte que ces concepts étaient souvent utilisés avec des acceptions différentes, et de ce fait créaient une certaine confusion dans les esprits. Il importe donc d'essayer de donner une idée intuitive de la signification du terme abstraction dans une approche de spécification algébrique.

Supposons donc que l'on ait demandé à un programmeur de définir une pile avec plusieurs des langages que nous avons exposés dans la première partie et qu'on lui demande ensuite d'expliquer les différentes versions de programmes. Il est alors clair que ses explications vont dépendre des possibilités d'abstraction de données supportées par ces langages. On peut présumer (sans trop se tromper) qu'elles seraient les suivantes:

(a)"ma pile est représentée par un tableau et un index",

(b)"ma pile est réalisée par le module pile avec une représentation par tableau et index",

(c)"ma pile est définie par la liste d'opérations: empiler, dépiler, etc...",

(d)"ma pile est définie par une équation au domaine".

Ce dernier cas nous a été suggéré par J.P.JOUANNAUD.

Sans entrer dans une nouvelle discussion sur les langages de programmation, nous pouvons tout de même dire que tous les langages supportant l'abstraction procédurale: ALGOL68, PASCAL conduiraient au cas (a). Le cas (b) correspondrait à l'utilisation de langages de type SIMULA ou CLU et le cas (c), moins courant que les autres se présenterait pour des langages comme ADA, TYP, etc...

Dans la situation (b), l'abstraction de données revient à cacher les détails de réalisation décrits dans un module, cependant l'objet abstrait (la pile) est lié à une seule réalisation. Les langages qui conduisent à la situation (c) fournissent un mécanisme d'abstraction de données plus élaboré. En effet, la description de cette abstraction de données appelée spécification (PAR72) est dans ce cas totalement séparée du module qui définit la réalisation. De ce fait, il est possible de concevoir plusieurs versions de modules de réalisation pour une même abstraction; la seule contrainte imposée est qu'ils doivent tous être "conformes" à la spécification. La conformité est uniquement une caractéristique syntaxique. Avec ces langages, la communication entre modules ne se fait que par l'intermédiaire des spécifications plus souvent appelées interfaces (abstraites). Ainsi, une interface définit une forme de caractérisation du comportement externe de tous les modules qui implantent une même abstraction de données. Si maintenant, cette interface comporte en plus une description formelle des propriétés des opérations, il est clair que l'utilisateur va pouvoir disposer d'une interface précise faisant abstraction des détails d'implantation et qui devrait même permettre de prouver la correction des modules. On perçoit que ce concept d'abstraction amène

en fait à distinguer deux choses: d'une part, le schéma des interfaces abstraites, d'autre part, l'interprétation qui définit la liaison des symboles d'objets et d'opérations respectivement aux représentations et algorithmes des modules d'implantation. Il importe donc de bien préciser quelles devront être les contraintes sur les interprétations des schémas si l'on veut qu'ensuite elles fournissent une version de module d'implantation correcte. Bref, il apparaît donc qu'une interprétation est une algèbre et ce que l'on cherche à définir est en fait une spécification d'une classe d'algèbres qui interprète correctement les schémas de programmes (modules d'implantation).

Maintenant, il nous faut encore préciser un certain nombre de points concernant les erreurs.

En programmation (programmes d'application, conception de langages, etc...), mais aussi dans beaucoup d'autres domaines de l'informatique, les erreurs (ou exceptions) ont longtemps été considérées comme des problèmes secondaires dont on pouvait reporter l'examen à l'implantation (messages d'erreur des compilateurs PL/1, PASCAL, etc..), à l'étape de tests ou même (très souvent) à l'exécution (!). Les coûts de maintenance des logiciels, l'apparition de plus en plus fréquente de systèmes logiciels importants devant fonctionner en "mode dégradé" (systèmes répartis), semblent avoir fait prendre conscience de l'importance réelle de ce type de problème. D'ailleurs plusieurs langages de programmation modernes CLU (LIS74), ADA (ADA80), etc.. fournissent aujourd'hui quelques mécanismes pour détecter et traiter les erreurs. Souhaitons que ces nouvelles caractéristiques de langage

favorisent une approche beaucoup plus systématique des erreurs, notamment au niveau de la conception. En effet, il nous paraît essentiel que les erreurs soient prises en compte dès le début de la conception. Il faudrait encore que l'on puisse caractériser des erreurs (ou messages d'erreur) indépendamment des langages de programmation ou des machines. Il apparaît donc essentiel que l'on admette qu'une erreur peut être une notion abstraite. Dans cette éventualité, les spécifications doivent aussi inclure les erreurs abstraites. C'est en ce sens que nous avons l'intention de présenter le cadre algébrique des types abstraits de données.

I. ALGEBRES HETEROGENES.

I.1. Σ-algèbres.

De manière informelle, une structure de données est une algèbre (LES79) constituée d'un ensemble d'objets et d'une collection d'opérations sur ces objets. Plus formellement, la collection d'objets est formée d'une famille d'ensembles disjoints appelés domaines (carriers). Chaque domaine est alors identifié par un nom de sorte (sort) s. On note $\mathcal{S}$ l'ensemble des noms de sortes de l'algèbre. De la même manière, chaque opération est identifiée par un nom d'opérateur σ. Chaque opération est définie comme une application de zéro ou plusieurs domaines dans un quelconque autre domaine de la structure de données. Chaque q-uplet $s_1 \cdot\cdot s_q$ de noms de sortes correspondants aux domaines d'une opération associée à un opérateur σ est appelé la source

de :

$$source(\sigma)=s_1 s_2 \cdots s_q.$$

Le nom de sorte correspondant au codomaine de l'opération associée à est appelée le but de σ :

$$but(\sigma)=s$$

Le q+1-uplet $s_1.s_2 \cdots s_q$ est appelé le profil de σ, que l'on note de la manière suivante:

$$profil(\sigma)=s_1.s_2 \cdots s_q \dashrightarrow s$$

Avec cette notation, q est appelé l' arité de σ. Un opérateur d'arité 0 est appelé constante. Ainsi chaque nom d'opérateur est typé par son profil.

On note Σ l'ensemble de tous les noms d'opérateurs et $\Sigma_{w,s}$ l'ensemble de tous les noms d'opérateurs de source w et de but s. Σ est l'union de tous les $\Sigma_{w,s}$.

Définition :Une signature sur $\mathcal{S}$ est une famille $\{\Sigma_{w,s}\}$ d'ensembles de symboles. Par abus de langage, la signature est encore désignée par Σ.

Une Σ-algèbre A est définie par:

-- une famille $\mathcal{S}$-indéxée $\{A_s\}_{s\in\mathcal{S}}$ d'ensembles appelée supports de s,

-- et, pour tout symbole d'opérateur $\sigma\in\Sigma_{sl\ldots sq,s}$ une fonction $\sigma_A : A_{sl}*\ldots*A_{sq} \dashrightarrow A_s$.

Des Σ-algèbres ayant des propriétés communes peuvent être réunies dans des classes. On note Alg_Σ la classe des Σ-algèbres (algèbres de signature Σ).

Exemple 1: Considérons la classe des Σ-algèbres constituées des sortes Booléens, Entiers et Ensembles avec les opérations CREER un Ensemble, INSERER un Entier dans un Ensemble, ENLEVER un Entier d'un Ensemble et DANS qui teste si un Entier est dans un Ensemble.

Dans cette classe: $\mathcal{S}=\{Bool,Int,Set\}$,

$$\Sigma_{\lambda,Set}=\{CREER\},$$

$$\Sigma_{Set\ Int,Set}=\{INSERER,ENLEVER\},$$

$$\text{et } \Sigma_{Set\ Int,Bool}=\{DANS\},$$

il existe plusieurs Σ-algèbres différentes. Par exemple:

a)la Σ-algèbre SET1 définie par:

.la famille $\mathcal{S}$-indexée:

$$Bool_{SET1}=\{T,F\}$$

$$Int_{SET1}=\{\ldots -2,-1,0,1,2,\ldots\}$$

$$Set_{SET1}=\{\{\},\{0\},\{1,2\},\ldots\}$$

Les ensembles des supports $Bool_{SET1}$, Int_{SET1}, Set_{SET1} sont définis soit par énumération des éléments, soit en utilisant un procédé de construction.

.et pour chaque symbole d'opérateur, CREER, INSERER, ENLEVER, DANS, les fonctions:

$$CREER_{SET1} : \dashrightarrow Set_{SET1}$$

$$INSERER_{SET1}, ENLEVER_{SET1} :$$

$$Set_{SET1}*Int_{SET1} \dashrightarrow Set_{SET1}$$

$DANS_{SET1}$: Set_{SET1}*Int_{SET1} --> $Bool_{SET1}$

définies par tabulation comme par exemple

$DANS_{SET1} = \{ (\{\},0),F),((\{0\},0),T),.... \}$

b)la Σ-algèbre SET2 sera définie par:

.la famille $\Im$-indexée:

$Bool_{SET2} = \{F\}$

$Int_{SET2} = \{0\}$

$Set_{SET2} = \{\{\}\}$

.et, pour chaque symbole d'opérateur, les fonctions $CREER_{SET2}$, $INSERER_{SET2}$, $ENLEVER_{SET2}$,$DANS_{SET2}$ définies comme précédemment par tabulation.

Ici, $DANS_{SET2} = \{ (\{\},0),F)\}$

c)ou encore la Σ-algèbre SET3 définie par:

.la famille $\Im$-indexée:

$Bool_{SET3} = \{1,0\}$

$Int_{SET3} = \{ -min,...,-1,0,1,...,max\}$

Set_{SET3} = Tableaux d'Entiers à borne variable, l'index indiquant la cardinalité courante de l'ensemble d'entiers.

De la même façon que ci-dessus, pour chaque opérateur, les fonctions $CREER_{SET3}$, $INSERER_{SET3}$, etc.. peuvent aussi être définies par tabulation:

$DANS_{SET3}$ = { ((Tabvide,0),F),((Ajout(Tabvide,0),0),T), ((Ajout(Tabvide,1),0),f), ...}

Ce dernier exemple est particulièrement intéressant. En effet, cette Σ-algèbre caractérise une représentation d'ensemble d'Entiers, en utilisant des structures de données standard des langages de programmation : Booléen, Entier, Tableau, etc..

I.2. Σ-homomorphismes, algèbres initiales.

Nous avons rappelé en introduction du chapitre III, que certains langages de programmation modulaire: ADA, TYP et autres permettaient de définir plusieurs modules d'implantation pour une même abstraction. Chaque version de module d'implantation décrit un choix de représentation et/ou d'algorithmes associés aux opérations de l'abstraction. L'utilisation d'une telle technique d'abstraction de données implique toutefois que l'on vérifie que chaque module ainsi défini a un comportement externe correct. Si c'est le cas, alors au niveau de l'utilisation de l'interface abstrait, on ne fait plus aucune distinction entre les différentes versions de modules "similaires". La vérification pratique de la correction des modules consiste simplement à effectuer des contrôles de conformité syntaxique des entêtes d'opérations et des tests spécifiques. Formellement, le comportement externe des opérations est une caractérisation abstraite des propriétés des algèbres. Il est donc judicieux d'utiliser ces propriétés pour déterminer précisément les classes d'algèbres similaires.

Dans les langages de programmation cités ci-dessus, les interfaces abstraits peuvent inclure des descriptions de comportements externes d'opérations: ce sont des commentaires qui servent par ailleurs de documentation.

Le formalisme algébrique permet d'obtenir une caractérisation précise des "algèbres similaires".

Soient A et B deux Σ-algèbres.

Définition :Un Σ-homomorphisme h:A --> B est une famille $\mathcal{S}$-indexée de fonctions $\{ h_s : A_s \text{-->} B_s \}_{s \in \mathcal{S}}$

telle que pour tout $\sigma \in \Sigma_{s1,\ldots,sq}$ et

pour tout $(a_1,\ldots,a_q) \in A_{s1} * A_{s2} * \ldots * A_{sq}$ alors

$$h_s(\sigma_A(a_1,\ldots,a_q)) = \sigma_B(h_{s1}(a_1),\ldots, h_{sq}(a_q)).$$

Exemple 2: Reprenons les Σ-algèbres SET1 et SET3 de l'exemple 1.

Un Σ-homomorphisme h de SET1 dans SET3 est défini par:

$$h{:}Bool_{SET1} \text{ --> } Bool_{SET3}$$

$$h{:}Int_{SET1} \text{ --> } Int_{SET3}$$

$$h{:}Set_{SET1} \text{ --> } Set_{SET3}$$

avec $h(DANS_{SET1}(s,i)) = DANS_{SET3}(h(s),h(i))$

Il est clair que le Σ-homomorphisme h préserve toutes les opérations de la Σ-algèbre SET1. Maintenant, le comportement externe des opérations de deux Σ-algèbres sera identique s'il existe un Σ-homomorphisme bijectif entre elles.

Un Σ-homomorphisme bijectif est aussi appelé Σ-isomorphisme. Ainsi, deux Σ-algèbres représentent la même abstraction si et seulement si elles sont Σ-isomorphes. Un "type abstrait de données" est alors une classe de Σ-algèbres isomorphes.

Pour que l'on parvienne à présenter un type abstrait avec un langage de spécification, il est nécessaire d'obtenir une caractérisation encore plus abstraite que la précédente. Plus précisément, il nous faut caractériser un élément représentatif de la classe des Σ-algèbres isomorphes.

Définition :Une Σ-catégorie $\mathcal{C}$ est une "classe" (catégorie) dont les objets sont des Σ-algèbres et les flèches tous les Σ-homomorphismes entre ces Σ-algèbres.

La Σ-catégorie particulière constituée de toutes les Σ-algèbres sera notée ALG_Σ.

Une Σ-algèbre A d'une Σ-catégorie $\mathcal{C}$ est dite initiale dans $\mathcal{C}$ si et seulement si pour toute Σ-algèbre B de $\mathcal{C}$ il existe un Σ-homomorphisme unique h de $\mathcal{C}$ où h: A-->B. L'intérêt d'une telle définition est de donner une caractérisation abstraite de la Σ-algèbre A.

Proposition : La Σ-catégorie ALG_Σ possède un objet (classe) initial(e) que l'on note T_Σ. On dit parfois que T_Σ est une Σ-algèbre initiale.

Proposition : Si A et B sont deux classes initiales d'une Σ-catégorie $\mathcal{C}$ alors A et B sont isomorphes dans $\mathcal{C}$ et

si B est une classe isomorphe dans $\mathcal{C}$ à une classe A initiale de $\mathcal{C}$ alors B est aussi une classe initiale de $\mathcal{C}$.

On voit que pour spécifier des abstractions de données, il suffit, en fait, de présenter des objets adéquats de la Σ-catégorie .

La classe T_Σ de $\mathcal{C}$ apparaît comme étant l'algèbre des expressions algébriques bien formées sur Σ. Ces expressions appelées encore Σ-termes sont composées de symboles d'opérateurs, de constantes (opérations 0-aires) et de variables de sortes appropriées; il faut en plus que les opérations soient conformes aux profils des opérateurs correspondants de Σ.

Soit $X=\{X_s\}_{s\in S}$ une famille S-indexée d'ensemble X_s dont les éléments sont appelés variables de sorte s . Nous supposons que les variables de domaines différents sont distinctes et qu'aucune variable n'est un nom d'opérateur de Σ. Soit $\Sigma(X)$ la signature obtenue en ajoutant à Σ les X_s, considérés comme ensembles de constantes de sorte s. Alors, $T_\Sigma(X)$ est une $\Sigma(X)$-algèbre. De plus, $T_\Sigma(X)$ est initiale dans la $\Sigma(X)$-catégorie $Alg_{\Sigma(X)}$.

Les éléments des supports de T_Σ sont appelés termes fermés, ou encore mots. Notons que $T_\Sigma(X)$ a les mêmes supports que $T_\Sigma(X)$ mais des symboles d'opérateurs dans Σ.

Définition : $T_\Sigma(X)$ est appelé Σ-algèbre libre engendrée par X.

Proposition : Soit A une Σ-algèbre et ass:X-->A une A-assignation ou A-interprétation (i.e. ass est une famille S-indexée de fonctions $ass_s:X_s \dashrightarrow a_s$). Alors ass peut être prolongée de façon unique en un Σ-homomorphisme ass : $T_\Sigma(X)$-->A tel que pour tout $s\in S$ et $x\in X_s$ alors $ass_s(x)=ass_s(x)$.

Cette proposition formalise une notion très utile en pratique: l'évaluation des expressions algébriques. En effet, ass est le procédé qui évalue les termes de $T_\Sigma(X)$ (éléments qui comportent des variables de X) à partir des valeurs des variables délivrées par l'assignation ass.

I.3. Equations.

Définition :Une Σ-équation est une paire de Σ-termes $G,D \in T_{\Sigma(X),s}$ où $s\in S$. Les équations sont notées (s(G=D)).

Ainsi, une équation est formée de deux termes composés de symboles fonctionnels et de variables (implicitement universellement quantifiées) et ces deux termes sont de même sorte. Pour éviter toute ambiguité avec la surcharge des opérateurs, nous donnons systématiquement la sorte des termes. Nous utilisons sans rappel la notion d'occurrence de variable dans un terme et V(t) désigne l'ensemble des variables qui ont une occurrence non nulle dans le terme t (cf II.1.).

Soit A une Σ-algèbre et (s(G=D)) une équation. A est un modèle de (s(G=D)) ou encore A valide la Σ-équation (s(G=D)) si et seulement si pour toute assignation ass:V(G) U V(D)-->A, on a : $ass_s(G) = ass_s(D)$ i.e. si et seulement si G et D dénotent le même élément du support A_s quelle que soit l'interprétation de leurs variables comme éléments de A. Si E est un ensemble de Σ-équations alors une Σ-algèbre A valide E si et seulement si A valide chaque équation de E.

Une algèbre qui valide E est appelée (Σ,E)-algèbre, et la classe des (Σ,E)-algèbres munie de tous les Σ-homomorphismes entre elles est notée $Alg_{\Sigma,E}$. $Alg_{\Sigma,E}$ possède aussi une classe initiale notée $T_{\Sigma,E}$.

Définition :Une présentation (ou spécification) équationnelle est un

triplet $P=(\mathcal{S},\Sigma,E)$ où Σ est la signature et E un ensemble de Σ-équations appelées les <u>axiomes de la présentation</u> (ou spécification).

<u>Théorème</u>: Pour toute spécification $P=(\mathcal{S},\Sigma,E)$ il existe une classe $T_{\Sigma,E}$ (i.e.une classe initiale dont les éléments sont des classes d'équivalence de Σ-termes fermés).

Nous donnons maintenant quelques résultats concernant la construction de $T_{\Sigma,E}$.

<u>Définition</u> :

Une <u>Σ-congruence</u> $\equiv$ sur une Σ-algèbre A est une famille $\mathcal{S}$-indexée de relations d'équivalence $\equiv_s$ sur A_s compatible avec la signature, i.e. on a:

pour tout $\sigma \in \Sigma_{s1\ldots sq,s}$, pour tout a_i,b_i A_{si} $(1 \leq i \leq n)$,

si $a_i \equiv_{si} b_i$ pour $1 \leq i \leq n$ alors

$\sigma_A(a_1,\ldots,a_n) \equiv_s \sigma_A(b_1,\ldots,b_n)$.

<u>Proposition</u> : Soit A une Σ-algèbre et $\equiv$ une Σ-congruence. Pour tout s dans $\mathcal{S}$ on définit:

-- $(A/\equiv)_s = A_s/\equiv_s$ (l'ensemble des $\equiv_s$-classes d'équivalence de A_s),

-- pour a A_s on note $[a]_s$ ou $[a]$ sa $\equiv_s$-classe d'équivalence,

-- les opérations de Σ sur $A/\equiv$ de la façon suivante:

-- si $\sigma \in \Sigma_{\lambda,s}$ alors $\sigma_{A/\equiv} = [\sigma_A]_s$

-- si $\sigma \in \Sigma_{s1\ldots sn,s}$ et $[a_i]_{si} \in A_{si}/\equiv_{si}$ pour $1 \leq i \leq n$,

alors $\sigma_{A/\equiv}([a_1]_{s1},\ldots,[a_n]_{sn}) = [\sigma_A(a_1,\ldots,a_n)]_s$.

Dans ces conditions, $A/\equiv$ est une Σ-algèbre appelée <u>algèbre quotient</u> de A par $\equiv$ ($\sigma_{A/\equiv}$ est bien définie car $\equiv$ est une congruence).

<u>Proposition</u> : Soit A une Σ-algèbre et $R \subseteq A*A$ une relation binaire sur A (i.e. une famille $\mathcal{S}$-indexée de relations R_s sur A_s). Alors il existe

une plus petite Σ-congruence sur A contenant R qui est appelée la <u>Σ-congruence engendrée par R</u> .

<u>Proposition</u> : Soit A une Σ-algèbre. Alors la relation $=_A$ définie sur $T_\Sigma(X)$ par $(s(G=_A D))$ est une Σ-congruence sur $T_\Sigma(X)$ si et seulement si A valide la Σ-équation $(s(G=D))$ (à condition qu'aucun support de A_s ne soit vide).

<u>Théorie équationnelle</u> : Soit E un ensemble d'équations. La théorie équationnelle définie par E est la plus petite Σ-congruence sur $T_\Sigma(X)$ engendrée par E et notée $\equiv_E$. Ainsi $\equiv_E$ est l'intersection de toutes les relations de congruence qui contiennent l'ensemble des relations d'équivalence déduites des équations de E par substitutions et remplacements de termes égaux.

Alors l'algèbre quotient $T_\Sigma / \equiv_E$ initiale dans $ALG_{\Sigma,E}$ sera notée $T_{\Sigma,E}$.

Intuitivement, ce résultat veut dire que les objets d'une spécification équationnelle sont des classes d'équivalence de termes, où la relation d'équivalence est choisie pour satisfaire E et pas plus.

<u>Théorème</u> : Soit Σ une signature, E un ensemble de Σ-équations. Alors $Alg_{\Sigma,E}$ valide $(s(G=D))$ si et seulement si $(s(G \equiv_E D))$.

I.4. Exemples.

a)Considérons la Σ-algèbre Nat constituée d'une sorte Entiers Naturels et des opérations 0(constante), SUCCesseur d'un entier. Cette algèbre:

$\mathcal{S} = \{Nat\}$

$\Sigma_{\lambda,Nat} = 0$

$\Sigma_{Nat,Nat} = \{SUCC\}$

est présentée de la manière suivante:

SPECIF Nat

OPNS 0: λ-->Nat

SUCC: -->Nat

AXS pas d'équation

FIN

b)Maintenant, donnons la spécification d'un type abstrait algébrique Horaire (Hor):

SPECIF Hor/Hor Nat

OPNS 0: λ-->Nat

SUCC: Nat-->Nat

0^* : λ-->Hor

HOR-SUIV: Hor-->Hor

DEC-HOR: Hor Nat-->Hor

AXS 1:Hor(HOR-SUIV(0^*)=0^*)

2:Hor(DEC-HOR(horaire,0)=horaire)

3:Hor(DEC-HOR(horaire,SUCC(n))=

HOR-SUIV(DEC-HOR(horaire,n)))

FIN

Ici, horaire est une variable de sorte Hor et n une variable de sorte Nat.

La Σ-algèbre Horl de spécification Hor est définie par:

-- les ensembles des supports Nat_{Horl} , Hor_{Horl} tels que

$Nat_{Horl} = N$ (ensemble des entiers naturels)

$Hor_{Horl} = \{ (n:m)/n,m \in N$ et $0 \leq n \leq 23,\ 0 \leq m \leq 59 \}$

-- les fonctions 0_{Horl} , $SUCC_{Horl}$, 0^*_{Horl} , $HOR\text{-}SUIV_{Horl}$:

0_{Horl} : -->Nat_{Horl}

$SUCC_{Horl}$: Nat_{Horl} -->Nat_{Horl}

0^*_{Horl} : -->Hor_{Horl}

$HOR\text{-}SUIV_{Horl}$:Hor_{Horl} -->Hor_{Horl}

définies de la manière suivante:

0^*_{Horl} = (0:0) où $0 \in N$

$0_{Horl} = 0 \in N$

$SUCC_{Horl}$ (n)=n+1 où $n,m \in N$

$HOR\text{-}SUIV_{Horl}$(n:m) = si $m \leq 58$ alors (n:m+1)

sinon si $n \leq 22$ alors (n+1:0) fsi

et validant l'équation (1) de la spécification Hor

-- et la fonction $DEC\text{-}HOR_{Horl}$:

$DEC\text{-}HOR_{Horl}$: Hor_{Horl} *Nat_{Horl} -->Hor_{Horl}

définie à partir des équations (2) et (3).

Ici, la Σ-algèbre telle que $\mathcal{S} = \{N\}$ et $\Sigma_{\lambda,N} = \{0\}$, $\Sigma_{N,N} = \{SUCC\}$ est isomorphe à l'algèbre de termes T_{Σ} de la spécification $SPECIF_{Nat}$ dont les ensembles du spport sont les classes d'équivalence: 0 et $SUCC^n(0)$ pour n>0 (constructibles dans $SPECIF_{Nat}$).

La Σ-algèbre Horl (modèle algébrique d'un Horaire) est isomorphe au quotient de la Σ-algèbre $T_{\Sigma,E}$ de spécification algébrique Hor. En

effet, la Σ-algèbre de termes T_{Σ} , formée de toutes les expressions bien formées de symboles appartenant à la signature Σ est initiale dans la classe Alg_{Σ} . En plus, la Σ-algèbre quotient $T_{\Sigma} / \equiv_E$ est initiale dans la classe $Alg_{\Sigma,E}$.

Comme la Σ-algèbre Horl valide les équations de la spécification Hor, il existe un Σ-homomorphisme unique h de $T_{\Sigma,E}$ vers Horl. Donc le type abstrait de spécification Hor est une Σ-algèbre isomorphe à $T_{\Sigma,E}$ et initiale dans $Alg_{\Sigma,E}$. Parfois, le type abstrait est défini sur le quotient de $T_{\Sigma,E}$. En effet, cette algèbre conserve l'essentiel des propriétés et de plus tous ses termes peuvent être obtenus à partir des Σ-opérations appliquées aux constantes du type abstrait.

I.5. Correction des spécifications.

I.5.1. Algèbres canoniques.

Définition :Une spécification algébrique de type abstrait de données $P=(\mathcal{S},\Sigma,E)$ est correcte si $T_{\Sigma,E}$ est isomorphe à un modèle mathématique standard.

Lorsque le type abstrait spécifié a un standard mathématique (entiers, ensembles, etc...), il est effectivement possible de faire une preuve rigoureuse de la correction de la spécification: en prouvant que la spécification est correcte et ensuite que le modèle mathématique est isomorphe à $T_{\Sigma,E}$. Une autre méthode (ADJ78) consiste à traiter des classes d'équivalence canoniques de termes. Ici, il s'agit d'obtenir

une Σ-algèbre $\mathcal{C}$ dont les supports sont constitués de termes canoniques et ensuite, de démontrer que $\mathcal{C}$ est isomorphe au modèle mathématique. Si $P=(\mathcal{S},\Sigma,E)$ est une spécification algébrique, il existe toujours une Σ-algèbre A dont les ensembles des supports des sortes sont les Σ-termes. Il suffit donc de choisir un terme représentatif t^* dans chaque classe d'équivalence $[t]$ de $T_{\Sigma,E}$ et de définir ensuite les opérations par

$\sigma_A(t_1^*,\ldots,t_n^*)=(\sigma(t_1,\ldots,t_n))^*$.

Si le choix du représentant de la classe est en principe sans importance; en fait, la structure particulière de certains termes peut faciliter les preuves.

Définition :Une Σ-algèbre $\mathcal{C}$ est dite canonique si tous les supports sont constitués de termes fermés (i.e. $\forall s\in\mathcal{S}\quad \mathcal{C}_s \subset T_{\Sigma,s}$) et si $\sigma(t_1\ldots t_n)\in\mathcal{C}$ alors

pour tout $i\leq n, t_i\in\mathcal{C}$ et $\sigma_{\mathcal{C}}(t_1,\ldots,t_n)=\sigma(t_1,\ldots,t_n)$.

Théorème : Soit $P=(\mathcal{S},\Sigma,E)$ une spécification algébrique , alors il existe une Σ-algèbre $\mathcal{C}$ canonique initiale dans la classe $Alg_{\Sigma,E}$ (cette algèbre est donc isomorphe à $T_{\Sigma,E}$).

Théorème : Soit $P=(\mathcal{S},\Sigma,E)$ une spécification et $\mathcal{C}$ une Σ-algèbre canonique. Alors $\mathcal{C}$ est isomorphe à $T_{\Sigma,E}$ (et donc initiale) si et seulement si

(1) $\mathcal{C}$ valide E, et

(2) pour tout opérateur $\sigma\in\Sigma_{s1,\ldots,sn,s}$ et pour tout terme $t_i\in\mathcal{C}_{si}$ $(1\leq i\leq n)$, on a $\sigma(t_1,\ldots,t_n)\equiv_E\sigma_{\mathcal{C}}(t_1,\ldots,t_n)$.

La dernière condition de ce théorème revient justement à dire que les termes de l'algèbre $\mathcal{C}$ "représentent" les $\equiv_E$-classes d'équivalence. Ce théorème fournit ainsi un critère général pour vérifier qu'une algèbre

canonique donnée est initiale. Le corollaire que nous rappelons ci-dessous est aussi très utile dans le cas d'une approche de spécification structurée.

Corollaire : Soit P1=($\mathcal{S}$,Σ1,E1) et P2=($\mathcal{S}$,Σ2,E2) deux spécifications algébriques telles que : $\Sigma 1 \subseteq \Sigma 2$ et $E1 \subseteq E2$ et A1 une Σ-algèbre canonique de termes. Alors une Σ-algèbre A2 telle que: &

(1) pour tout $s \in S$, $A2_s = A1_s$,et

(2) pour tout $\sigma \in \Sigma 1$, $\sigma_{A2} = \sigma_{A1}$

est une Σ-algèbre initiale et la spécification P2 est un enrichissement de P1 si

(1) A2 valide E2-E1, et

(2) pour tout $\sigma \in (\Sigma 2_{s1\ldots sn,s} - \Sigma 1_{s1\ldots sn,s})$ et

pour tout terme $t_i \in A1_{si}$ $(1 \leq i \leq n)$ on a

$\sigma(t_1,\ldots,t_n) \neq_{E2} \sigma_{A2}(t_1,\ldots,t_n)$.

Ces résultats sont très utiles pour montrer la correction d'un certain nombre de spécifications comme les Booléens, les Entiers Naturels, les Séquences, etc... Maintenant, lorsque le type abstrait de données n'a pas de modèle mathématique standard, cas d'une pile, d'une table des symboles, et de nombreux types abstraits propres aux applications informatisées, il existe d'autres façons de montrer la correction de la spécification, par exemple en montrant directement la consistance et la complétude.

I.5.2. Consistance et complètude des spécifications algébriques.

Intuitivement, un système de spécification algébrique est consistant si toute expression algébrique bien formée se réduit à un terme unique, quel que soit l'ordre d'application des axiomes équationnels. Elle est complète si la procédure de décision termine en un nombre fini d'étapes.

En général, les problèmes de consistance et de complétude des spécifications algébriques sont indécidables. GUTTAG et HORNING (GH78) ont défini une propriété de complétude moins forte: la "complétude suffisante" et des critères qui permettent d'en faire la preuve. Nous ne ferons qu'énoncer les résultats importants exposés en détail dans (GH78) (GAU80).

Soit P=($\mathcal{S}$,Σ,E) une spécification algébrique de type abstrait de donnée définie de la façon suivante:

-- $\mathcal{S}$ est un ensemble de sortes avec un élément distingué d appelé "type d'intérêt",

-- Σ est un ensemble d'opérations (noms d'opérateurs et fonctionnalités) partitionné en deux sous-ensembles G et O. L'ensemble G appelé ensemble des opérations internes est constitué des opérations de Σ à codomaine d. L'ensemble O appelé ensemble des opérations externes (ou d'observation) est constitué des opérations de Σ dont le codomaine est différent de d.

-- E est un ensemble d'axiomes équationnels (éventuellement conditionnés). & Soit $F=\mathcal{S}-\{d\}$ l'ensemble des sortes "externes". Notons

T_F l'ensemble des termes de sorte $s \in \mathcal{S}$ où n'apparaissent pas les opérations de Σ. Il s'agit des termes des types externes (ou prédéfinis).

Définition :On dit que E est une abstraction suffisamment complète de la spécification $P=(\{d\} \cup F, G \cup O, E)$ si et seulement si, pour tout terme de la forme $f(t_1,\ldots,t_n)$ tel que $f \in O$, il existe un théorème, démontrable avec E, par des réécritures de la forme $f(t_1,\ldots,t_n)=u$ où $u \in T_F$.

Et, E est consistant si pour tout $f(t_1,\ldots,t_n)$, u est unique.

Il a été montré que la complétude suffisante d'un ensemble d'axiomes est en général indécidable. cependant, si l'on impose quelques restrictions au système d'axiomes équationnels, alors il existe des conditions suffisantes pour vérifier cette propriété syntexiquement. Les restrictions portent sur la structure et le degré d'imbrication des termes. Une axiomatisation qui satisfait ces contraintes est dite axiomatisation sûre (GAU80). Soit $P=(\mathcal{S}, G \cup O \cup \{f\}, E)$ une spécification où $f: d*s_1*\ldots*s_n \rightarrow d$. Alors on dit que f est convertible à G, si pour toute expression de la forme $f(x,y^*)$, où y^* est une liste de variables libres typées, il existe un théorème démontrable avec E, de la forme $f(x,y^*)=z$ où z ne contient pas d'occurrence de f.

Les constantes du "type d'intérêt" d sont les opérations de G U O qui n'ont pas d'opérande de sorte d mais dont le codomaine est de sorte d. Il existe au moins une constante dans G U O.

Théorème : Soit $P=(\mathcal{S}, G+O, E)$ une spécification algébrique. L'ensemble des axiomes équationnels E est suffisamment complet s'il existe une

partition de G en deux ensembles C (les constructeurs) et R (les extensions), les constantes appartenant toutes à C et,

-- pour tout $c \in C$, $o \in O$ il existe un axiome $o(c(x^*),y^*)=z^*$ vérifiant les propriétés d'une axiomatisation sûre (décroissance, imbrication)

-- il est possible d'ordonner les éléments de O: $o_1,\ldots,o_n$ de manière à ce que o_1 soit convertible à C, o_2 soit convertible à $C+\{o_1\}$, etc...

Ce théorème présente un intérêt pratique important. D'abord parce qu'on dispose ainsi d'un critère de suffisante complétude applicable à la plupart des spécifications algébriques "informatiques". De plus, en l'utilisant avec l'algorithme de KNUTH et BENDIX (KB70) (LES79) on a alors un outil efficace pour contrôler la consistance des spécifications des utilisateurs (MUS80).

Ensuite, il fournit une technique d'aide à la construction de spécifications suffisamment complètes . En effet, il suffit de:

-- distinguer dans la signature de la spécification algébrique, les opérations de construction (C), les extensions (R) et les opérations d'observation (O),

-- écrire les axiomes équationnels qui caractérisent les extensions R,

-- écrire les axiomes équationnels qui caractérisent l'effet des opérations d'observation sur les constructeurs de C. Chacun de ces axiomes doit avoir une partie gauche de la forme $o(c(x^*),y^*)$ et une partie droite qui vérifie les propriétés d'une axiomatisation sûre pour obtenir une spécification suffisamment complète.

Précisons encore que si l'on interprête les axiomes équationnels comme des règles de réécriture, alors les propriétés de consistance et de complétude suffisante d'une spécification algébrique s'interprètent en terme de propriétés des systèmes de réécriture. Or ces systèmes (HUE80) (MUS80) fournissent des procédures de décision pour tester ou compléter un grand nombre de spécifications. Nous exposerons rapidement quelques résultats sur les systèmes de réécriture en VII.2.

I.6. Spécifications algébriques avec "erreurs".

Pour certains domaines de l'informatique: conception de langages de programmation (ADA80), conception de systèmes de gestion de bases de données (DAT77) (WAS79), conception de systèmes experts (LAU82), la complexité des problèmes des logiciels mais aussi le volume important des données traitées ont conduit depuis plusieurs années à introduire dans ces langages (ou systèmes) spécialisés des mécanismes adéquats de protection, de sécurité et donc de détection et de traitement d'erreurs. En ce qui concerne la spécification, le problème a encore à peine été étudié (GOG78) (GUT80). En fait, l'introduction des erreurs dans le cadre formel algébrique pose quelques problèmes théoriques sérieux (algèbres partielles, algèbres continues, ...).

Avant d'exposer les deux principales approches du problème de la spécification des cas d'erreurs, nous allons montrer sur un exemple très simple la difficulté d'aborder le problème. Nous présentons très brièvement la technique de GUTTAG (GUT80) pour ensuite développer

l'approche des algèbres d'erreurs de GOGUEN (GOG78).

I.6.1. Présentation du problème des erreurs sur un exemple.

Considérons une spécification algébrique Pile[Element] telle qu'elle a été proposée par GUTTAG (GUT76):

type Pile[Element]

opérations

pilevide()-->Pile;

empiler(Pile,Element)-->Pile;

dépiler(Pile)-->Pile U {undef};

sommet(Pile)-->Element U {undef};

estvide(Pile)-->Booleen;

axiomes

vars p∈Pile, e∈Element;

(1)estvide(pilevide)=vrai;

(2)estvide(empiler(p,e))=faux;

(3)dépiler(pilevide)=undef;

(4)dépiler(empiler(p,e))=p;

(5)sommet(pilevide)=undef;

(6)sommet(empiler(p,e))=e;

fintype Pile

Cette spécification est apparemment suffisamment complète et consistante d'après le théorème énoncé en VII.1.5.2. Elle nous semble pourtant particulièrement ambigue. En effet, examinons par exemple l'axiome (3) caractérisant l'opération d'extension dépiler.

Dans ce cas, comment doit on interpréter le symbole undef? S'il représente une valeur, alors quel est son type? On est donc tout naturellement amené à introduire dans chaque type une valeur spécifique d'erreur: $undef_p$ pour Pile, $undef_e$ pour Element et $undef_b$ pour Booleen. Mais alors on ne connaît pas le comportement des opérations du type sur ces nouvelles valeurs. La spécification est incomplète. Une façon de la compléter consiste à définir de nouveaux axiomes qui caractérisent la "propagation des erreurs":

(7)empiler($undef_p$,e)=$undef_p$;

(8)empiler(p,$undef_e$)=$undef_p$;

(9)dépiler($undef_p$)=$undef_p$;

(10)estvide($undef_p$)=$undef_b$;

(11) sommet($undef_p$)=$undef_e$;

Nous supposons bien sûr que le symbole undef est remplacé dans les axiomes (3) et (5) respectivement par les symboles $undef_p$ et $undef_e$. La spécification semble maintenant suffisamment complète. Malheureusement, elle n'est plus consistante. Pour le montrer, considérons l'équation suivante:

dépiler(empiler(pilevide,sommet(pilevide))=pilevide.

Elle semblerait à priori correcte, d'après l'axiome (4). Or appliquons à la partie gauche les axiomes suivants:

(5):dépiler(empiler(pilevide,$undef_e$)

(8):dépiler($undef_p$)

(9):$undef_p$

soit $\underline{undef_p = pilevide}$

Or d'après l'axiome (4), pour toute pile p et tout élément e, on a:

p=dépiler(empiler(p,e))

d'après (8)et(9) p=dépiler(empiler(p,$undef_e$))=dépiler($undef_p$)

soit $\underline{p=undef_p}$

comme d'après (2)

faux=estvide(empiler(p,e))=estvide(empiler($undef_p$,e))

d'après (7) =estvide($undef_p$)

d'après (10) =$undef_b$

$\underline{faux=undef_b}$

et comme d'après (1) vrai=estvide(pilevide)=estvide($undef_p$)

d'après (10) =$undef_b$

$\underline{vrai=undef_b}$

Alors faux=$undef_b$=vrai.

Pour lever ce problème d'inconsistance, il faudrait revoir toute l'axiomatisation. Cet exemple simple montre que la spécification des cas d'erreurs doit être menée avec beaucoup de précautions si l'on ne veut pas aller à l'encontre du but recherché: fournir une spécification la plus complète possible mais cohérente. Nous allons aborder maintenant, mais très brièvement, l'approche proposée par GUTTAG.

I.6.2. Spécification avec cas d'"exceptions et d'échecs". (GUT80)

Dans l'approche définie par GUTTAG (GUT80), il nous semble que les notions d'exceptions et d'échecs répondent plus au souci d'adapter les techniques des spécifications algébriques à la preuve de correction des programmes écrits dans des langages de programmation modernes CLU (LIS74), ADA (ADA79), qu'à celui de fournir des aides à la spécification de problèmes. Cette technique consiste à définir les cas exceptionnels par des préconditions qui restreignent l'applicabilité des axiomes et par des prédicats d'échecs qui caractérisent de manière abstraite les "anomalies" des implantations des types abstraits.

Avec cette technique, une spécification de pile bornée se décrirait comme ci-dessous:

type Pile[Element]

opérations

pilevide()-->Pile;

empiler(Pile,Element)-->Pile;

dépiler(Pile)-->Pile;

sommet(Pile)-->Element;

estvide(Pile)-->Booleen;

taille(Pile)-->Entier;

axiomes

Vars p ∈ Pile, e ∈ Element;

(1)'estvide(pilevide)=vrai;

(2)'estvide(empiler(p,e))=faux;

(3)'dépiler(empiler(p,e))=p;

(4)'sommet(empiler(p,e))=e;

(5)'taille(pilevide)=0;

(6)'taille(empiler(p,e))=taille(p)+1;

restrictions

(7)' pre (dépiler,p)=¬estvide(p);

(8)' pre (sommet,p)=¬estvide(p);

(9)'taille(p) ≥ 50 ==> echec (empiler,p,e);

fin

Ici, les prédicats (7)' et (8)' définissent les préconditions à satisfaire pour pouvoir appliquer respectivement les opérations dépiler et sommet. Le programmeur a donc à charge de vérifier les préconditions et éventuellement de définir les traitements correspondants. La restriction (9)' spécifie que toute implantation de l'opération empiler devra comporter un test sur le nombre des éléments mais aussi un traitement caché en cas d'échec. C'est donc l'implémenteur qui dans ce cas aura à charge de définir le test et le traitement d'échec.

Cette technique pose des problèmes théoriques qui concernent notamment les prédicats associés au type d'intérêt. Le lecteur intéressé trouvera une étude détaillée de ces problèmes dans (KAP80) (REM82).

I.6.3. Spécification avec propagation des erreurs.

Si l'on examine la spécification décrite en VII.1.6.1., on s'aperçoit que l'inconsistance de l'axiomatisation provient uniquement de l'existence des équations suivantes:

dépiler(empiler(p,e))=p et dépiler($undef_p$)=$undef_p$

qui s'appliquent toutes les deux à l'expression:

dépiler(empiler(p,$undef_p$))

Ceci montre qu'on peut éliminer l'inconsistance, en restreignant de manière judicieuse l'application des équations. Cela peut se faire en remplaçant les équations précédentes par des équations conditionnelles qui devront naturellement caractériser tous les comportements possibles des opérations. Ainsi, pour définir les axiomes conditionnels de l'opération dépiler, il faut distinguer trois sortes de comportements:

-- dépiler n'a aucun argument en erreur et délivre une valeur du type; ce qui s'exprime par:

" si p≠$undef_p$ et e≠ $undef_e$ alors dépiler(empiler(p,e))=p"

-- dépiler a comme argument une constante du type mais délivre une valeur erreur. C'est le cas de:

dépiler(pilevide)=$undef_p$

-- et, tous les cas où dépiler a un ou plusieurs arguments erronés et délivre donc une valeur erreur ("propagation des erreurs")

" si p≠$undef_p$ alors dépiler(empiler(p,$undef_e$))=$undef_p$", etc...

En fait, les conditions n'ont été introduites que pour déterminer si les arguments sont en erreur ou non. C'est ce qui a amené GOGUEN, THATCHER et WAGNER à introduire pour chaque type, une opération "ok" et une opération "ifok", extension du si alors sinon sur des booléens avec erreurs. Les spécifications définies avec cette approche sont excessivement complexes. En effet, pour l'exemple de la pile il faudrait encore écrire:

.des opérations:

okpile(Pile)-->Booleen;

okelt(Element)-->Booleen;

ifoke(Pile,Element,Element)-->Booleen;

ifokp(Pile,Element,Pile)-->Booleen;

.des équations sur les opérations "ok" et "ifok":

okpile($undef_p$)=faux;

okpile(pilevide)=vrai;

.

.

ifokp(p_1,e,p_2)= si (okpile(p_1)∧ okelt(e),p_2,$undef_p$)

ifoke(p_1,e_1,e_2)= si (okpile(p_1)∧ okelt(e_1),e_2,$undef_e$)

.des équations spécifiques:

dépiler(pilevide)=$undef_p$

ifokp(p,n,dépiler(empiler(p,n))= ifokp(p,n,p)

.

.

.et des équations de propagation des erreurs:

empiler($undef_p$,n)=$undef_p$

empiler(p,$undef_e$)=$undef_p$

dépiler($undef_p$)=$undef_p$

.

.

Par la suite, COGUEN (COG78) a développé un raffinement technique des algèbres hétérogènes: les algèbres avec erreurs ("error algebras") qui permettent d'obtenir des spécifications algébriques avec erreurs beaucoup plus simples. En effet, avec ce formalisme, on a juste à expliciter les équations d'erreurs spécifiques décrites précédemment. Les "ok" et "ifok" opérations, les "ok" et "ifok" équations ainsi que les équations de propagation d'erreurs sont ici implicites.

I.6.4. Les algèbres avec erreurs. (COG78)

Nous présentons maintenant les notions essentielles concernant les algèbres avec erreurs: signatures, homomorphismes, présentation, algèbres erreurs quotients, etc...

Si Σ et Σ' sont deux signatures de même ensemble de sortes $\mathcal{S}$ alors $\Sigma \cup \Sigma'$ est encore une signature d'ensemble de sortes $\mathcal{S}$ où $(\Sigma \cup \Sigma')_{w,s} = \Sigma_{w,s} \cup \Sigma'_{w,s}$.

Définition :Une signature avec erreurs est un triplet $(\mathcal{S}, \Sigma^{OK}, \Sigma^{ER})$ où Σ^{OK} et Σ^{ER} sont deux signatures disjointes sur le même ensemble de sortes $\mathcal{S}$. Les opérations de Σ^{OK} sont appelées OK-opérations et celles de Σ^{ER} ER-opérations.

Définition :Une $(\Sigma^{OK}, \Sigma^{ER})$-algèbre avec erreurs A est définie par:

-- une famille $\mathcal{S}$-indexée $\{A_s\}_{s \in \mathcal{S}}$ où chaque support A_s est partitionné en A_s^{OK} et A_s^{ER}. Les éléments de A_s^{OK} sont appelés les OK-éléments et ceux de A_s^{ER} les erreurs ("messages d'erreurs")

-- et,(a)pour tout symbole d'opérateur $\sigma \in \Sigma^{OK}_{s1,..,sq,s}$

il existe une fonction $\sigma_A : A_{s1} * ... * A_{sq} \rightarrow A_s$ telle que

pour tout $a_j \in A_{sj}$ $(1 \leq j \leq n)$

si $a_j \in A_{sj}^{ER}$ alors $\sigma_A(a_1,..,a_j,..,a_q) \in A_s^{ER}$

(b)pour tout symbole d'opérateur $\varphi \in \Sigma^{ER}_{s1,...,sq,s}$

pour tout $(a_1,..,a_i,..,a_q) \in A_{s1} * .. * A_{si} * .. * A_{sq}$

alors $\varphi_A(a_1,..,a_i,..,a_q) \in A_s^{ER}$.

Cette définition donne une caractérisation simple de la "propagation des erreurs".

(a)Si une OK-opération a un (ou plusieurs) argument(s) en erreur(s), alors son résultat est une erreur.

(b)Les ER-opérations ne délivrent que des erreurs.

Notons que l'algèbre A est constituée de toutes les $(\Sigma^{OK} \cup \Sigma^{ER})$-expressions.

Soit $R=(\mathcal{S}, \Sigma^{OK}, \Sigma^{ER})$ une signature avec erreurs, A et B deux R-algèbres avec erreurs. Alors un R-homomorphisme avec erreurs est un $(\Sigma^{OK} \cup \Sigma^{ER})$-homomorphisme composé d'une famille $\mathcal{S}$-indexée de fonctions $\{h_s : A_s \rightarrow B_s\}_{s \in \mathcal{S}}$ telle que

pour tout $s \in \mathcal{S}$: $h_s(A_s^{ER}) \subseteq B_s^{ER}$.

Ainsi, un R-homomorphisme avec erreurs est un $(\Sigma^{OK} \cup \Sigma^{ER})$-homomorphisme qui préserve les erreurs.

Le $(\Sigma^{OK}, \Sigma^{ER})$-catégorie des algèbres avec erreurs de signature $R=(\mathcal{S}, \Sigma^{OK}, \Sigma^{ER})$ est notée $EAlg_R$.

La $(\Sigma^{OK},\Sigma^{ER})$-catégorie $EAlg_R$ possède une classe initiale notée T_R ayant pour

.supports et opérations exactement ceux de $T_{\Sigma^{OK}\cup\Sigma^{ER}}$

.et,pour ensemble d'OK-éléments les termes de $T_{\Sigma^{OK}}$ et eux uniquement.

Il importe encore de savoir définir des classes représentatives de T_R si l'on veut pouvoir spécifier des algèbres avec erreurs comme des algèbres initiales de ces classes.

Définition :Une présentation (ou spécification) avec erreurs est un 5-uplet $P=(\mathcal{S},\Sigma^{OK},\Sigma^{ER},E^{OK},E^{ER})$ où $(\mathcal{S},\Sigma^{OK},\Sigma^{ER})$ est une signature avec erreurs, E^{OK} est un ensemble de Σ^{OK}-équations et E^{ER} est un ensemble de $(\Sigma^{OK}\cup\Sigma^{ER})$-équations.

On dira alors que la présentation avec erreurs P est validée par une $(\Sigma^{OK},\Sigma^{ER})$-algèbre avec erreurs notée A si et seulement si, pour toute équation $(s(G=D))$ de $E^{OK}\cup E^{ER}$ où $s\in\mathcal{S}$ et $G,D\in T_{\Sigma OK(X)\cup\Sigma ER(X),s}$ et toute interprétation (assignation) des variables ayant une occurrence dans l'équation, alors le résultat de l'évaluation g=d est

(a) si $g\in A^{OK}$ et $d\in A^{OK}$ alors $(g=d)\in A^{OK}$

(b) si $g\in A^{ER}$ ou $d\in A^{ER}$ alors $(g=d)\in A^{ER}$

L'algèbre avec erreurs A est appelée P-algèbre. La classe des P-algèbres munie de tous les $(\Sigma^{OK}\cup\Sigma^{ER})$-homomorphismes sur elles est notée $EAlg_p$.

Pratiquement, ce qui nous intéresse, c'est de pouvoir exhiber une classe d'algèbres de $EAlg_p$ dont un représentant serait l'algèbre initiale T_p.

Le mécanisme de construction consiste à interpréter les

$(E^{OK}\cup E^{ER})$-équations comme des règles de réécriture. Nous ne détaillons pas ces notions qui ont déjà été discutées en I.3. et I.5.2.

Maintenant, dans tout ce qui suit, nous supposons que les présentations avec erreurs $P=(\mathcal{S},\Sigma^{OK},\Sigma^{ER},E^{OK},E^{ER})$ vérifient la propriété suivante:

(*) à chaque équation de E^{ER} correspond au moins une ER-opération de Σ^{ER}.

Soit A une $(\mathcal{S},\Sigma^{OK},\Sigma^{ER})$-algèbre avec erreurs et $\equiv$ une $(\Sigma^{OK}\cup\Sigma^{ER})$-congruence définie sur A. Pour tout s dans $\mathcal{S}$ on définit:

-- $(A/\equiv)_s=(A_s/\equiv_s)$ (ensemble des $\equiv_s$-classes d'équivalence de A_s),

-- pour tout $a\in A_s$ on note $[a]_s$ sa $\equiv_s$-classe d'équivalence,

-- les opérations de $(\Sigma^{OK}\cup\Sigma^{ER})$sur $A/\equiv$ de la façon suivante:

$\sigma_{A/\equiv}([a])=[\sigma_A(a)]$ où une $\equiv$-classe d'équivalence $[a]$ est une classe erreur si et seulement si il existe un ER-élément b dans A^{ER} tel que $a\equiv b$ et $[a]$ est une classe-ok si et seulement si elle ne contient que des OK-éléments de A^{OK}. Dans ces conditions, $A/\equiv$ est une $(\mathcal{S},\Sigma^{OK},\Sigma^{ER})$-algèbre avec erreurs appelée $(\Sigma^{OK},\Sigma^{ER})$-algèbre avec erreur quotient de A par $\equiv$ et les fonctions $h:A\dashrightarrow A/\equiv$ des $(\mathcal{S},\Sigma^{OK},\Sigma^{ER})$-homomorphismes avec erreurs.

Donnons nous une $(\mathcal{S},\Sigma^{OK},\Sigma^{ER})$-algèbre avec erreurs Q qui va nous servir à montrer que la P-algèbre avec erreurs initiale T_P que l'on cherche à construire est en fait un représentant de l'algèbre avec erreurs quotient de T_Q par $\equiv_P$.

De plus, pour pouvoir définir les ER-éléments de T_Q il faut encore

introduire un prédicat ERR sur $T_{\sum^{OK} \cup \sum^{ER}}$. Ce prédicat est défini par induction sur les termes de $T_{\sum^{OK} \cup \sum^{ER}}$, sachant que pour tout $\sigma \in (\sum^{OK} \cup \sum^{ER})$, ERR($\sigma$) a la valeur vrai si et seulement si $\sigma \in \sum^{ER}$ ou alors il existe une équation de $E^{OK} \cup E^{ER}$ utilisant l'opération σ. Ainsi, soit $t \in T_{\sum^{OK} \cup \sum^{ER}}$, $\sigma \in (\sum^{OK} \cup \sum^{ER})$ et $t=\sigma(t_1,\ldots,t_q)$. Alors ERR(t) (supposé définie sur t) prend la valeur vrai si et seulement si:

-- $\sigma \in \sum^{ER}$

-- ou $ERR(t_i)$ pour au moins un entier i tel que $1 \leq i \leq q$

-- ou encore s'il existe un élément a_i tel que $a_i \equiv_{E^{OK}} t_i$ et $1 \leq i \leq q$ pour lequel on peut appliquer une équation de $E^{OK} \cup E^{ER}$ à l'occurrence i de $\sigma(a_1,..,a_i,..,a_q)$. Ici, $\equiv_{E^{OK}}$ est la $\sum$-congruence engendrée par E^{OK} sur $T_{\sum^{OK}}$ et $t_i \in T_{\sum^{OK}}$.

Théorème :Soit P une spécification avec erreurs vérifiant la condition (*) précédente. Alors, l'algèbre $T_P = T_Q / \equiv_P$ est initiale dans la classe des P-algèbres vérifiant la propriété suivante: si g est l'unique Q-homomorphisme de T_Q vers T_P, alors le résultat de l'évaluation de g(t) doit satisfaire la condition suivante:

ERR(t) est vrai si et seulement si $g(t) \in T_{\sum^{OK}}$.

Ce théorème a une importance pratique essentielle. En effet, la construction de T_P à partir de $\equiv_P$ est une justification théorique du procédé de réécriture "du plus interne au plus externe en commençant par les règles d'erreurs" que nous avons développé dans le système VEGA.

I.7. Spécification des types abstraits paramétrés.

Dans le domaine du génie logiciel, il est maintenant tout à fait admis que la paramétrisation appelée aussi généricité (BER79) est une technique d'aide à la conception et à la programmation essentielle, surtout lorsqu'il s'agit de développer des applications logicielles importantes. De plus, dans beaucoup d'applications informatiques, le nombre de "types" de structures de données utiles est assez limité: produits cartésiens, séquences, "ensembles" (ensembles sans répétition), multi-ensembles (ensembles avec répétition). Il convient d'affiner un peu ce concept de "voisinage" car, pour certaines applications informatiques comme par exemple les bases de données, on s'aperçoit que la "distance" entre les différentes instances d'un même "type" de structure de données peut devenir relativement importante. S'il est vrai que la paramétrisation s'avère déjà fort utile pour construire des spécifications syntaxiquement voisines comme par exemple des ensembles d'entiers et des ensembles de réels, etc.., elle le devient encore plus lorsqu'il s'agit de définir des ensembles où les paramètres doivent vérifier des contraintes (propriétés sur les valeurs, ou propriétés structurelles) ou encore restreindre la liste des opérations, procéder à leur renommage, etc...

En fait, l'intérêt de la paramétrisation n'est pas seulement d'apporter une aide efficace à la manipulation des textes, c'est aussi dans une approche formelle de pouvoir obtenir automatiquement des spécifications correctes. Par construction, une instance d'une spécification paramétrée correcte est aussi correcte si l'on a pu

vérifier que les valeurs des paramètres effectifs satisfont les contraintes associées aux paramètres formels respectifs. En toute rigueur, il faudrait en plus s'assurer que l'instanciation ne modifie pas le type: problème de persistance.

Toutes les notions que nous venons d'introduire de façon informelle font l'objet de recherches pour l'instant essentiellement théoriques (ADJ78). Une présentation de ces travaux, même sommaire, nous ferait sortir très largement du cadre de cette thèse. Nous terminerons donc ce paragraphe par un simple rappel de la définition d'une spécification paramétrée (ADJ78).

Définition :Une spécification algébrique paramétrée est une paire PSPEC=(SPEC,SPEC1) où SPEC est une spécification des paramètres incluse dans la spécification SPEC1.

Les instanciations des paramètres formels de la spécification SPEC qui définissent la spécification effective SPEC2 sont des "morphismes de spécification" de SPEC vers SPEC2. Ainsi, une instance de la spécification paramétrée PSPEC correspond "en gros" à une spécification SPEC1 dans laquelle la partie SPEC a été renommée. Ce renommage est effectué à partir des spécifications utilisées comme paramètres effectifs.

Le lecteur intéressé par ces approches théoriques de la spécification paramétrée peut se reporter aux références citées ci-dessus ainsi qu'à la thèse de (BER79).

I.8. Implantation des types abstraits.

En introduction de cette partie (VII.1.), nous avons indiqué que le processus de conception des applications pouvait amener à spécifier des implantations abstraites de types abstraits de données. Ce cas se présente lorsqu'il s'agit d'implanter un type abstrait (non primitif) à partir d'autres types abstraits de l'environnement de spécification. Nous présenterons rapidement deux techniques d'implantation: celle de GOGUEN (GOG78) (ADJ78) puis celle de ZILLES et GUTTAG (ZIL79).

Une autre façon de procéder consiste à faire la synthèse automatique de l'implantation du type abstrait (GAU80). Nous ne présenterons dans ce paragraphe que la méthode que GUTTAG (GUT78) a appelée "implantation directe". Le lecteur intéressé par les autres solutions possibles trouvera des informations détaillées dans (GAU78).

I.8.1. Implantation abstraite des spécifications avec erreurs.(GOG78)

La technique d'implantation abstraite que nous allons présenter maintenant est une généralisation de (ADJ76). Soit $R=(\mathcal{S},\Sigma^{OK},\Sigma^{ER})$ une signature avec erreurs. Soit $u=u_1..u_i..u_n$ et $w=w_1..w_i..w_n$ tels que $u_i,w_i \in \mathcal{S}$ pour tout i, $1 \leq i \leq n$. On note $T_R(u,w)$ l'ensemble de tous les $(\Sigma^{OK} \cup \Sigma^{ER})$-termes t de la forme $t=(t_1,..,t_i,..,t_n)$ où t_i est de sorte w_i et t_i possède des occurrences de variables dans $\{X_1,..,X_i,..,X_n\}$ où X_i est de sorte u_i , pour tout i, $1 \leq i \leq n$.

Soient $R_1=(\mathcal{S}_1,\Sigma_1^{OK},\Sigma_1^{ER})$ et $R_2=(\mathcal{S}_2,\Sigma_2^{OK},\Sigma_2^{ER})$ deux signatures avec erreurs. Alors un dériveur avec erreurs d de R_1 vers R_2 est composé:

-- d'une fonction $f:\mathcal{S}_1 \rightarrow \mathcal{S}_2$

-- et d'une famille de fonctions $\{d_{u,v}\}_{u,v\in \mathcal{S}^*}$ où

$d_{u,v} : (\Sigma_1^{OK} \cup \Sigma_1^{ER})_{u,v} \rightarrow T_{R2}(f(u),f(v))$ telle que si $\alpha \in \Sigma_1^{ER}$ alors $d_{u,v}(\alpha)$ est un ER-élément de T_{R2}.

Soit B une R_2-algèbre avec erreurs. Alors l'algèbre avec erreurs dR_2 "d-dérivée" de R_2 est la R_1-algèbre avec erreurs où pour chaque opération σ de $\Sigma_1^{OK} \cup \Sigma_1^{ER}$ alors σ_{dB} définit l'opérateur dérivé $(d(\sigma))_B$ des R_2-termes $d(\sigma)$. Maintenant, soit $P=(\mathcal{S}_1,\Sigma_1^{OK},\Sigma_1^{ER},E_1^{OK},E_1^{ER})$ une spécification avec erreurs. Alors, une implantation abstraite avec erreurs de P est un triplet $(B,d,\equiv)$ où B est une R_2-algèbre, d est un dériveur avec erreurs de $R_1=(\mathcal{S}_1,\Sigma_1^{OK},\Sigma_1^{ER})$ vers R_2 et $\equiv$ est une R_1-congruence sur dB où T_P est isomorphe à un sous-ensemble de la R_1-algèbre avec erreurs $dB/\equiv$.

Exemple : L'exemple d'implantation abstraite que nous présentons reprend les structures de données Set d'entiers et Tableau de pointeurs dont nous avons déjà discuté en VII.1.1. Nous commençons par décrire la spécification de Set d'entiers avec une opération erreur de nom VIDE, ensuite nous présentons la spécification d'une combinaison Index, Tableau d'entiers appelée Tabind

```
SPECIF Set/Int,Bool
OKOPNS
    CREER:-->Set
    INSERER:Set Int-->Set
    ENLEVER:Set Int-->Set
    DANS:Set Int-->Bool
EROPNS
    VIDE:-->Set
VARS e∈Set; i,j∈ Int
OKAXS
    1:INSERER(INSERER(e,i),j)= SI EQ(i,j) ALORS INSERER(e,j)
            SINON INSERER(INSERER(e,j),i) FSI
    2:ENLEVER(INSERER(e,i),j)= SI EQ(i,j) ALORS ENLEVER(e,j)
            SINON INSERER(ENLEVER(e,j),i) FSI
    3:DANS(CREER,i)=faux
    4:DANS(INSERER(e,i),j)= SI EQ(i,j) ALORS vrai SINON DANS(e,j)
            FSI
ERAXS
    1:ENLEVER(CREER)=VIDE
    2:ENLEVER(VIDE)=VIDE
END
```



```
SPECIF Tabind/Int,Bool
OKOPNS
    TABVIDE:-->Tabind
    AJOUT:Tabind Int Int-->Tabind
    VAL:Tabind Int-->Int
    EST-DANS:Int Tabind Int-->Bool
    INDEX:Int Tabind Int-->Int
VARS
    t∈Tabind; i,j,n,m∈Int
OKAXS
    1:VAL(AJOUT(t,i,m),j)= SI EQ(i,j) ALORS m SINON VAL(t,j) FSI
```

2:AJOUT(AJOUT(t,i,m),j,n)= SI PPQ(j,i) ALORS

AJOUT(AJOUT(t,j,n),i,m) FSI

3:EST-DANS(n,AJOUT(t,j,m),i)= SI EQ(n,m) ET PPQ(j,i)

ALORS VRAI SINON EST-DANS(n,t,i) FSI

4:INDEX(n,AJOUT(t,j,m),i)= SI EQ(n,m) ET PPQ(j,i) ALORS i

SINON INDEX(n,t,i) FSI

END

La spécification Tabind utilise deux prédicats EQ (égalité) et PPQ (plus petit que). Nous supposerons dans la suite que Int a été défini avec une opération erreur NEG telle que DECREM(n)=n-1, DECREM(0)=NEG et DECREM(NEG)=NEG. Nous utilisons aussi une opération classique INCR: INCR(n)=n+1.

Notons $R_{Set}=(\mathcal{S}_{Set}, \Sigma^{OK}_{Set}, \Sigma^{ER}_{Set})$ et $R_{Tabind}=(\mathcal{S}_{Tabind}, \Sigma^{OK}_{Tabind}, \Sigma^{ER}_{Tabind})$ les signatures avec erreurs des spécifications avec erreurs respectives Set et Tabind. Notons $\mathcal{S}_{Set}=\{Set,Int,Bool\}$ et $\mathcal{S}_{Tabind}=\{Tabind,Int,Bool\}$ les ensembles de sortes respectivement de Set et Tabind et, par abus de langage, $Set=(\mathcal{S}_{Set}, \Sigma^{OK}_{Set}, \Sigma^{ER}_{Set}, E^{OK}_{Set}, E^{ER}_{Set})$ la spécification avec erreurs Set. Une implantation abstraite avec erreurs de Set est donc la donnée d'un triplet (B,d, $\equiv$) où

.B est la R_{Tabind}-algèbre,

.d est un dériveur avec erreurs de R_{Set} vers R_{Tabind} auquel on associe une fonction f de Set dans Tabind telle que f(Set)=<Tabind,Int> avec pour tout $s\in\mathcal{S}_{Set}-\{Set\}$ on ait f(s)=s

.et pour chaque opérateur σ de $\Sigma^{OK}_{Set} \cup \Sigma^{ER}_{Set}$ les opérations dérivées correspondantes d(σ):

d(CREER)=<TABVIDE,0>

d(INSERER)(<t,i>,n)= SI EST-DANS(n,t,i) ALORS <t,i>

SINON <AJOUT(t,INCR(i),n),INCR(i)> FSI

d(ENLEVER(<t,i>,n)= SI EST-DANS(n,t,i)

ALORS <AJOUT(t,INDEX(n,t,i),VAL(t,i)),DECREM(i)>

SINON <t,i> FSI

d(DANS)(<t,i>,n)=DANS(n,t,i)

d(VIDE)=<TABVIDE,NEG>

. $\equiv$ est une R_{Set}-congruence sur dB, où dB est une R_{Set}-algèbre définie sur Tabind avec les opérations (d(CREER), d(INSERER), d(ENLEVER), d(DANS), d(VIDE)). Notons A_{Set} cette R_{Set}-algèbre. Maintenant, si l'on montre qu'il existe un homomorphisme surjectif h de Tabind sur Set tel que $A_{Set}/\equiv_h$ soit isomorphe à T_{Set} alors $A_{Set}/\equiv_h$ sera une implantation abstraite de Set avec Tabind.

Si l'on définit l'homomorphisme h de la manière suivante:

pour tout t$\in$Tabind: h(<t,0>)=CREER

pour tout t$\in$Tabind et n$\neq$0:

h(<t,n>)= INSERER(INSERER(...(INSERER(CREER,VAL(t,1),VAL(t,2),...)..).

Alors toute forme normale de Set sera représentée par un h(<t,n>).

Il faut aussi définir la relation de congruence $\equiv_h$ sur Tabind:

$<t_1,n_1> \equiv_h <t_2,n_2>$ si et seulement si $h(<t_1,n_1>) \equiv h(<t_2,n_2>)$.

Il reste encore à préciser l'équivalence des erreurs sur l'implantation ainsi définie. Prenons donc la relation de congruence $\equiv_h$ générée par l'équation d'erreur suivante:

<TABVIDE,NEG>=<t,NEG>

Avec cette dernière équation, l'implantation abstraite est maintenant complètement définie. Pour prouver la correction de cette implantation, nous devrions en toute rigueur, montrer que h ainsi défini est un homomorphisme surjectif. Nous ne le ferons pas ici, car cette démonstration est relativement longue à établir. Elle ne présente toutefois pas de difficulté.

I.8.2. Implantation abstraite des spécifications selon GUTTAG.(GUT76)

Cette approche a déjà été discutée dans de nombreux articles et thèses (GUT78) (GAU80) (LES79) (BER79) (FIN79), etc... Aussi, nous ne ferons qu'un bref rappel de cette technique de définition et de preuve de la correction des implantations abstraites de types abstraits (sans erreurs) à partir d'autres types.

La spécification d'une implantation abstraite est relativement simple à décrire. Elle comporte deux parties:

-- une représentation qui déclare une fonction d'abstraction de la structure "concrète" vers la structure abstraite (le type abstrait implanté),

-- et un programme constitué d'équations qui caractérisent le comportement des opérations du type abstrait implanté sur la structure concrète. Il faut ensuite vérifier que cette implantation est correcte. Cela se fait par étapes.

(a) On définit un invariant de représentation de l'implantation qu'il faut ensuite prouver par induction.

(b) On définit l' interprétation de l'égalité sur l'implantation.

(c) On établit ensuite des conditions de vérification en appliquant à chaque axiome du type abstrait implanté et à chaque axiome définissant l'égalité (réflexivité, symétrie, transitivité, substitution) une substitution de la représentation qui utilise l'invariant de représentation.

(d) Il faut enfin procéder à la preuve de chaque condition de vérification.

Maintenant, si l'on compare cette technique avec celle de GOGUEN présentée en VII.1.8.1., cette dernière nous semble plus rigoureuse et surtout plus satisfaisante en ce qui concerne la spécification et l'implantation des erreurs. Nous renvoyons le lecteur intéressé par l'implantation et la correction de spécifications avec restrictions et préconditions aux thèses de (KAP80) et (REM82). Ce qui différencie surtout ces deux approches est la façon dont l'une ou l'autre aborde le problème de la correction d'une spécification.

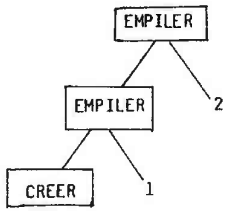
Nous avons vu dans les paragraphes précédents que la technique de GOGUEN est essentiellement basée sur la notion d' algèbre initiale . Dans ce cas, la théorie équationnelle définie par les équations d'une spécification de signature Σ est la plus petite Σ-congruence sur T_Σ engendrée par les équations de la spécification. C'est ce qui fait que $T_{\Sigma,E}$ est initiale et unique à un isomorphisme près. Avec cette approche, si l'on ne peut pas déduire de l'ensemble des équations d'une spécification que deux termes sont équivalents alors on considère qu'ils sont distincts.

La technique de GUTTAG est fondée sur la notion d' algèbre terminale (LES79). La relation de congruence utilisée déduite de l'interprétation de l'égalité n'est donc pas la plus petite congruence sur T_Σ engendrée par les équations. Deux termes sont équivalentss si l'on ne peut pas déduire de l'ensemble des équations qu'ils ne le sont pas. L'avantage de cette deuxième solution est de laisser une plus grande liberté de choix à l'implanteur (LES79) (FIN79).

I.8.3. Implantation directe des spécifications abstraites. (GUT78)(LES79)

Introduite par GUTTAG (GUT78), cette technique d'implantation directe consiste "en gros" à interpréter les parties gauches et droites des équations, et de façon plus large toute expression bien formée des types abstraits comme des arbres . Si la spécification algébrique est canonique, c'est-à-dire si les classes d'équivalence de termes de même sorte ont un représentant seulement constitué de constructeurs et si ce représentant est unique, alors on peut associer à chaque terme d'une sorte une représentation en arbre de sa forme canonique (LES79). Notons $\underline{t}$ l'ensemble des arbres associés à une sorte d'intérêt t d'un type abstrait. Alors, une implantation directe de t est une interprétation libre ou interprétation de HERBRAND de t obtenue en associant à chaque opérateur σ du type abstrait tel que $\sigma : t_1 \ldots t_q \rightarrow t_i$ et $t_i \neq t$ (i.e. σ n'est pas un constructeur) une opération $\underline{\sigma}$ de $\underline{t}_1 * \ldots * \underline{t}_q$ dans $\underline{t}_i$ vérifiant les équations de la spécification.

Exemple: Le terme EMPILER(EMPILER(CREER,1),2) de Set d'Entiers se représente sous la forme de l'arbre suivant:



Intuitivement, le concept d'implantation directe vise à interpréter les équations d'une spécification algébrique comme des opérations qui produisent et manipulent des arbres. Plus précisément, si l'on oriente les équations, alors l'ensemble des équations de la spécification constitue un système de réécriture. Nous présenterons cette notion dans le prochain paragraphe.

Cette technique d'implantation des spécifications algébriques présente de nombreux avantages. Elle fournit une aide à la construction de spécifications correctes. En effet, la vérification de la correction peut être faite sans avoir à préciser d'implantation abstraite du type abstrait. C'est une technique qui supporte une véritable méthode de spécification structurée par raffinements successifs. MUSSER (MUS80) l'utilise aussi pour prouver des propriétés des types abstraits.

II. LES SYSTEMES DE REECRITURE (KK82) (HUL80).

Cette section est dévolue à la présentation des notions fondamentales qui sont à la base de la réalisation du système de spécification algébrique avec erreurs VEGA (compilateur de spécifications algébriques avec erreurs, implantation programmée de types paramétrés avec erreurs, interprétation d'exécution). Nous commençons par un bref rappel de notions de base qui sont utilisées dans les définitions de la réécriture et de la surréduction que nous exposons ensuite. Les définitions et résultats de cette section ont été empruntés à un certain nombre de spécialistes de la réécriture: G.HUET et D.OPPEN (HO80), HUET (HUE80), C. et H. KIRCHNER (KK82) (LES79). Aucune démonstration n'est rappelée. Nous renvoyons donc le lecteur intéressé aux références citées ci-dessus.

Ajoutons encore que cette section utilise les définitions et résultats sur les algèbres hétérogènes et les algèbres avec erreurs qui ont été exposées en VII.1. Dans toute la suite, nous supposons donnés un ensemble $\mathcal{S}$ de noms de domaines, une signature Σ, une famille X de variables, T_Σ la Σ-algèbre des termes fermés et $T_\Sigma(X)$ la Σ-algèbre libre engendrée par X.

Soit $\langle \mathcal{S}, \Sigma, E \rangle$ une spécification algébrique. Pour ne pas alourdir les notations, nous considérons que cette spécification est uni-sorte; les définitions et les résultats décrits dans cette section se généralisant sans difficulté aux spécifications multi-sortes.

La signature Σ est l'ensemble des noms d'opérateurs avec les arités respectives. Σ est partitionné en deux ensembles: Σ^C l'ensemble des constructeurs et Σ^D l'ensemble des opérateurs définis. On suppose en plus que Σ^C comporte au moins deux constructeurs.

II.1. Rappels de notions de base.

Soit X un ensemble énumérable de symboles appelés variables. Une expression (ou terme) se définit par induction comme étant soit un symbole de variable v, soit un symbole d'opérateur f et une séquence d'expressions $e_1,\ldots,e_n$. Le symbole f est appelé opérateur dominant de l'expression et $e_1,\ldots,e_n$ les arguments. Cette expression sera notée $f(e_1,\ldots,e_n)$. Une expression f() sans argument est appelée constante.

Remarque: En fait, une expression (ou terme) est une notation linéaire d'un arbre étiqueté. Les sous-expressions d'une expression sont soit l'expression elle même, soit les sous-expressions des arguments de l'expression.

Les expressions formées de variables ou de constantes n'ont pas d'autre sous-expression qu'elles-mêmes.

L'ensemble des occurrences d'une expression e, noté D(e) est défini par une séquence d'entiers telle que:

(1) λ(mot vide) $\in$ D(e)

(2) si ij $\in$ D(e) alors i $\in$ D(e)

(3) si i $\in$ D(e) alors ij $\in$ D(e) pour tout j tel que $1 \leq j \leq$ arité(e(i))

et il n'y en a pas d'autre.

Exemple: Les occurrences des sous-expressions de l'expression

EMPILER(EMPILER(VIDE,i),j) sont:

λ EMPILER(EMPILER(VIDE,i),j)

1 EMPILER(VIDE,i)

2 j

11 VIDE

12 i

On note O(e) l'ensemble des occurrences de non variable d'une expression e.

On note e/i la sous-expression de e à l'occurrence i.

On note e $\left[i \leftarrow e_1\right]$ l'expression obtenue à partir de e en remplaçant e/i par e_1. Deux expressions sont identiques si:

(1) elles sont constituées de variables identiques ou si

(2) les symboles d'opérateurs et les séquences d'arguments associées sont identiques.

L'identité de deux expressions e_1 et e_2 est notée $e_1=e_2$.

Soit $\mathcal{T}$ l'ensemble des expressions définies sur Σ et X (notation simplifiée de X_{set}).

Définition : Une substitution est une application Θ de X dans $\mathcal{T}$ telle que $\Theta(x)=x$ pour tout x dans X sauf sur un sous-ensemble fini.

On note $\Theta = \left[e_1/v_1,\ldots,e_n/v_n\right]$ cette application où toutes les variables v_i pour $i \leq i \leq n$ sont distinctes et telle que l'on ait $\Theta(v_1)=e_1,\ldots,\Theta(v_n)=e_n$ et $\Theta(v)=v$ sinon.

Cette application se prolonge immédiatement (de façon unique) à $\mathcal{T}$ et de plus

$\Theta(f(e_1,\ldots,e_n))=f(\Theta(e_1),\ldots,\Theta(e_n))$.

Définition :Un filtrage d'une expression e_2 par une expression e_1 est

une substitution Θ telle que

$$\Theta(e_1)=e_2.$$

On dit aussi que e_2 est filtrée (ou est "matchée") par e_1.

Exemple: L'expression DEPILER(EMPILER(VIDE,i)) est filtrée par DEPILER(EMPILER(p,x)) par la substitution $\Theta = \left[VIDE/p, i/x\right]$.

Définition :Un unificateur de deux expressions e_1 et e_2 est une substitution Θ telle que

$$\Theta(e_1)=\Theta(e_2).$$

On dit aussi que les expressions e_1 et e_2 sont unifiables par Θ.

Exemple: Considérons les expressions e_1=EMPILER(p,SOMMET(q)) et e_2=EMPILER(DEPILER(t),i) et les substitutions:

$\Theta_1 = \left[DEPILER(SOMMET(e_2))/p,\ SOMMET(e_2)/t, SOMMET(q)/i\right]$

$\Theta_2 = \left[DEPILER(t)/p, SOMMET(q)/i\right]$.

Alors $\Theta_1(e_1)=\Theta_1(e_2)$= EMPILER(DEPILER(SOMMET(e_2)),SOMMET(q)) et $\Theta_2(e_1)=\Theta_2(e_2)$= EMPILER(DEPILER(t),SOMMET(q)) sont deux unificateurs des expressions e_1 et e_2.

Définition :Un plus grand unificateur de deux expressions e_1 et e_2 est un unificateur Θ tel que pour tout autre unificateur Θ_1 de e_1 et e_2 il existe une substitution Θ_2 telle que

$$\Theta_1=\Theta_o\Theta_2.$$

Ainsi, dans l'exemple précédent, Θ_2 est le plus grand unificateur de e_1 et e_2. L'algorithme d'unification de ROBINSON (ROB65) fournit un plus grand unificateur s'il en existe et sinon signale que les expressions ne sont pas unifiables.

II.2. Systèmes de réécriture d'expressions.

Définition : Une règle de réécriture d'expression est un couple d'expressions (G,D) tel que l'ensemble des variables de G soit inclus dans l'ensemble des variables de D. Un système de réécriture d'expressions est un ensemble fini de règles de réécriture.

Soit un système de réécriture R. Soient deux expressions e_1 et e_2, alors on dit que e_1 se réduit (se récrit) en e_2 à l'occurrence i de e_1 si et seulement si il existe:

-- une règle $e_1 \rightarrow e_2$ de R

-- une occurrence i dans $D(e_1)$

-- une substitution θ telle que

$$e_1/i = \theta(e_1) \text{ et } e_2 = e_1\left[i \leftarrow \theta(e_2)\right]$$

et l'on note e_1 se réduit en e_2 par $e_1 \xrightarrow{}_R e_2$ ou simplement $\overrightarrow{R}$.

De plus, $\overset{+}{\rightarrow}_R$ désigne la fermeture transitive de R,

$\overset{*}{\rightarrow}_R$ sa fermeture réflexive et transitive et

$\overset{=}{R}$ sa fermeture réflexive, symétrique et transitive.

Exemple: Considérons le système de réécriture associé à une pile d'entiers sans erreurs:

R= { ...,DEPILER(VIDE)-->VIDE, DEPILER(EMPILER(p,i))-->p,... }.

Alors l'expression DEPILER(EMPILER(EMPILER(VIDE,1),2)) se réduit à l'occurrence λ en EMPILER(VIDE,1).

Définition :Une expression e_1 est dite sous forme normale ou irréductible si et seulement si il n'existe pas d'expression e_2 telle que $e_1 \xrightarrow{}_R e_2$; autrement dit, si e_1 ne peut plus être récrit dans R.

Soient deux expressions e_1 et e_2. Si e_2 est irréductible et si $e_1 \overset{*}{\rightarrow}_R e_2$, on dit alors que e_2 est une forme normale de e_1 que l'on note $FN(e_1)$. Une substitution θ est dite normalisée si et seulement si pour toute variable x dans X, $\theta(x)$ est sous forme normale.

Remarques:

1-- une expression ne peut avoir une forme normale que si le processus de réécriture termine,

2-- une expression peut aussi avoir plusieurs formes normales. C'est le cas par exemple pour les ensembles:

AJOUTER(AJOUTER(e,1),2)=AJOUTER(AJOUTER(e,2),1).

II.3. Propriétés des systèmes de réécriture.

Définition : Un système de réécriture R est noethérien si et seulement si pour toute expression, il n'existe pas de dérivation infinie $e_0 \rightarrow e_1 \rightarrow \ldots$

.. utilisant R. On dit aussi que R possède la propriété de terminaison finie. Si le système de réécriture R est noethérien, alors toute expression possède au moins une forme normale (LES79).

Définition :Un système de réécriture est confluent si et seulement si pour toutes expressions e, e_1 et e_2 telles que $e \overset{*}{\rightarrow}_R e_1$ et $e \overset{*}{\rightarrow}_R e_2$, il existe une expression e_3 telle que

$$e_1 \overset{*}{\rightarrow}_R e_3 \text{ et } e_2 \overset{*}{\rightarrow}_R e_3.$$

On dit aussi que R possède la propriété de terminaison unique.

Remarque : Si le système de réécriture est confluent, alors chaque expression se récrit en au plus une expression irréductible, ou forme normale.

Définition :Un système de réécriture R est localement confluent si et

seulement si pour toutes expressions e, e_1 et e_2 telles que $e \xrightarrow[R]{} e_1$ et $e \xrightarrow[R]{} e_2$, il existe une expression e_3 telle que

$$e_1 \xrightarrow[R]{*} e_3 \text{ et } e_2 \xrightarrow{*} e_3,$$

et un système de réécriture noethérien est confluent si et seulement si il est localement confluent.

Définition :Un système de réécriture canonique (ou complet) est un système à la fois noethérien et confluent.

Et si R est un système complet alors toute expression a une forme normale et une seule appelée forme canonique ; de plus, $e_1 =_R e_2$ si et seulement si $FN(e_1)=FN(e_2)$.

II.4. Principe de définition.

G.HUET et D.OPPEN (HO80) ont développé une théorie équationnelle où les propriétés des équations d'un système de spécification algébrique $(\mathcal{S}, \Sigma, E)$ s'expriment en terme de propriétés de règles de réécriture.

Soit T_Σ l'ensemble des expressions sans variables définies sur Σ, C_Σ l'ensemble des expressions sans variables définies sur Σ^C c'est-à-dire qui ne sont composées que de constructeurs. Alors:

PRINCIPE DE DEFINITION:

Soit E un ensemble de Σ-équations. Soit $=_E$ la Σ-congruence engendrée sur $T_\Sigma(X)$. Alors on dit que E définit Σ^D sur Σ^C si et seulement si pour tout m dans T_Σ, il existe dans C_Σ un n unique tel que $m =_E n$.

Ce principe peut aussi être exprimé avec les deux conditions suivantes:

(1) pour tout m dans T_Σ, il existe un n dans C_Σ tel que $m =_E n$

(2) pour tout m et n dans C_Σ, on a $m =_E n$ si et seulement si m=n.

Si E satisfait le principe de définition alors C_Σ est isomorphe à l'algèbre initiale $T_{\Sigma,E}$. Nous énonçons maintenant deux lemmes qui fournissent des critères de décision du principe de définition.

Lemme 1: Soit E un ensemble de Σ-équations définissant un système de réécriture noethérien tel que pour toute expression de la forme $f(m_1,\ldots,m_n)$ avec f dans Σ^D et $m_1,\ldots,m_n$ dans C_Σ, $f(m_1,\ldots,m_n)$ est réductible, alors E satisfait la condition (1).

Lemme 2: Soit E un ensemble de Σ-équations définissant un système de réécriture canonique où chaque partie gauche est de la forme $f(m_1,\ldots,m_n)$ avec f dans Σ^D, alors E a la propriété (2).

Le principe de définition et les deux lemmes que nous venons de rappeler ont une importance pratique fondamentale. La syntaxe et la compilation du langage de spécification VEGA conçu sur ces bases peuvent ainsi être justifiées théoriquement. Le dernier lemme montre que le principe de définition est préservé par extension si et seulement si l'extension est valide dans le modèle initial $T_{\Sigma,E}$.

Lemme 3: Soit $(\mathcal{S}, \Sigma_1, E_1)$ et $(\mathcal{S}, \Sigma_2, E_2)$ deux spécifications algébriques telles que $\Sigma_1 \subseteq \Sigma_2$ et $E_1 \subseteq E_2$. Si E_1 satisfait la condition (1) du principe de définition et si E_2 est telle que la relation $=_{E1}$ est incluse dans $=_{E2}$, alors E_2 a la propriété (2) si et seulement si:

(a) E_1 satisfait (2) et

(b) chaque équation de E_2 est valide dans $T_{\Sigma,E}$.

II.5. Notion de surréduction.

Soit $(\mathcal{S},\Sigma,E)$ une spécification algébrique. Résoudre une équation $e_1 =_E e_2$ dans la théorie équationnelle engendrée par les Σ-équations E, c'est trouver une substitution θ normalisée telle que $\theta(e_1) =_E \theta(e_2)$ ou encore $FN(\theta(e_1)) = FN(\theta(e_2))$.

Si une telle substitution θ existe, c'est que θ unifie les deux expressions e_1 et e_2 et dans ce cas, θ s'obtient par l'unification classique ou alors l'une des deux expressions $\theta(e_1)$ ou $\theta(e_2)$ est réductible et θ est une surréduction.

Définition :Soit e_1 une expression, $g_k \xrightarrow{}_R d_k$ une règle k de réécriture de R et W_k la réunion des ensembles disjoints de variables de e_1 et de g_k.

On dit que e_1 se surréduit en e_2 à l'occurrence i de $\theta(e_1)$ avec la règlekde R et la substitution θ si et seulement si

$$t/i =_E \theta(g_k) \text{ et } e_2 = \theta(e_1\,[\,i \leftarrow\!- d_k\,]).$$

Cette substitution θ est appelée surréduction et on la note $e_1 \xrightarrow[[i,k,\theta]]{} e_2$.

Remarque : Toute expression réductible par une règle de R est aussi surréductible avec cette même règle. Cependant, pour la surréduction, la substitution s'applique à la fois aux variables de l'expression et aux variables renommées de la règle alors que pour la réécriture la substitution ne s'applique qu'aux seules variables de la règle.

Exemple : Considérons deux règles du système de réécriture canonique associé à un type pile d'entiers:

1: CARD(VIDE)=0

2: CARD(EMPILER(p,i))=SUCC(CARD(p))

et une expression CARD(p).

Alors, $CARD(p) \xrightarrow[[0,1,\theta_1]]{} 0$ avec $\theta_1 = [\,p \text{-->} VIDE\,]$,

$CARD(p) \xrightarrow[[0,2,\theta_2]]{} SUCC(CARD(p_1))$

avec $\theta_2 = [\,p \text{-->} EMPILER(p_1, i_1)\,]$

représentent deux surréductions possibles de l'expression CARD(p).

Définition :Une surdérivation est une suite de surréductions.

On dit qu'une surdérivation suit une stratégie de l'intérieur vers l'extérieur si à chaque étape de réduction $e_i \xrightarrow[[i,k,\theta]]{} e_{i+1}$ la substitution θ telle que $e_{i+1} = e_i[\,i \leftarrow\!- \theta(d_{ki})\,]$ est normalisée.

L'interpréteur d'exécution (évaluation des expressions) du système VEGA utilise la surdérivation. De façon générale, l'ensemble des définitions et résultats de cette section constitue l'essentiel des bases théoriques du système VEGA que nous présentons au chapitre IX.

CHAPITRE VIII. NOTIONS DE BASE DU CALCUL DES PREDICATS, PRESENTATION DE PROLOG.

I. NOTIONS DE BASE DU CALCUL DES PREDICATS DU PREMIER ORDRE.

II. METHODE DE RESOLUTION DE ROBINSON.

III. PROLOG.

Nous nous plaçons désormais dans le cadre du calcul des prédicats du premier ordre. Nous commençons d'abord par rappeler brièvement quelques notions de base du calcul des prédicats (I.), ensuite nous exposons la méthode de résolution de ROBINSON (II.) utilisée par le langage PROLOG que nous présentons en III.

I. NOTIONS DE BASE DU CALCUL DES PREDICATS DU PREMIER ORDRE.

Nous ne rappelons ici que sommairement les notions de logique mathématique utilisées dans la suite de ce travail. On trouvera des exposés plus complets dans les ouvrages de logique (KRE67) mais aussi dans les articles et thèses de (PAI74) (FIN79).

Nous supposons connues les notions élémentaires du calcul des prédicats que sont les définitions de l'alphabet, des axiomes logiques, des règles d'inférence (modus ponens et introduction du quantificateur existentiel), les formules bien formées du langage ou la définition d'une interprétation des formules bien formées.

Définition :On appelle modèle M d'un ensemble de formules bien formées E toute interprétation qui satisfait E, ce que l'on note M ⊨=E. Une formule vraie dans toutes les interprétations est dite valide ; ce que l'on note ⊨=e.

Définition :Une formule e est conséquence logique de E si e est vraie dans tous les modèles de E, ce que l'on note E ⊨=e. C'est la théorie du modèle qui détermine E ⊨=e.

Définition :Une formule e est un théorème si elle est dérivable à partir des axiomes logiques par application un nombre fini de fois, des règles d'inférence (modus ponens, introduction du quantificateur existentiel) ce que l'on note ⊢-e.

Définition :Une formule e est conséquence de E si elle est dérivable à partir des axiomes logiques et des formules de E par application, un nombre fini de fois, des règles d'inférence, ce que l'on note E (prémisses) ⊢-e (conclusion). La théorie qui détermine E ⊢-e est la théorie de la preuve .

De toutes ces définitions, il résulte que:

-- les théorèmes sont des formules valides, ce que l'on note si ⊢-e alors |=e, et on dit que le calcul des prédicats est une théorie valide

-- de plus, si une formule est valide alors c'est un théorème de la théorie, ce que l'on note si |=e alors ⊢-e, et on dit que le calcul des prédicats est une théorie complète.

Le calcul des prédicats permet de formaliser toute théorie du premier ordre. Par exemple, les systèmes formels (PAI74) (REM74) sont des théories du premier ordre avec égalité et "typées". En effet, ce sont des théories constituées du calcul des prédicats auquel on a ajouté un prédicat d'égalité, des axiomes caractérisant l'égalité formelle , des axiomes caractérisant l'égalité de termes compatibles (de même profils), des axiomes de substitution et des axiomes propres. Ce sont les axiomes propres (ou axiomes de définition) qui permettent de distinguer une théorie d'une autre.

Avant d'exposer une méthode de mise en oeuvre de cette théorie, c'est-à-dire de démonstration de théorèmes, nous devons encore présenter la forme standard sous laquelle ces formules sont manipulées. D'après le théorème de SKOLEM (CHA73), toute formule bien formée et fermée (c'est-à-dire sans variable libre) du calcul des prédicats du premier ordre peut s'écrire sous forme clausale.

Définition : Une clause est une formule qui a la forme générale suivante:

$$B_1 \vee B_2 \vee \ldots \vee B_m \leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

(où <-- désigne l'implication logique) que l'on écrit souvent sous la forme:

$$+B_1+B_2\ldots+B_m-A_1-A_2\ldots-A_n$$

où les A_i et B_i sont des formules atomiques.

Définition :Une formule est dite sous forme clausale si elle s'écrit sous la forme:

$C_1 \wedge C_2 \wedge \ldots \wedge C_p$ où les C_i sont des clauses.

Notons que cette formule est implicitement quantifiée universellement.

Par la suite, nous n'utiliserons que des clauses ayant l'une des formes suivantes:

$+B-A_1-A_2\ldots-A_n$

ou $-A_1-A_2\ldots-A_p$

ou $+B$

appelées clauses de HORN .

Exemple : La formule définissant la concaténation des listes sous forme de clauses de HORN s'écrit:

(1) +CONC(nil,x,x)

(2) +CONC(cons(x,y),z,cons(x,u))-CONC(y,z,u)

où nil est un symbole 0-aire et cons un symbole fonctionnel d'arité 2.

L'énoncé du problème de concaténation de deux listes cons(1,nil) et cons(2,cons(3,nil)) s'exprime avec la clause de HORN suivante:

-CONC(cons(1,nil),cons(2,cons(3,nil)),u).

S'il est vrai qu'un certain nombre de problèmes de logique s'expriment facilement en termes de clauses de HORN (PER80), il faut aussi préciser que ces clauses imposent quelques contraintes. Mis à part le fait que l'on ne puisse pas exprimer de propriétés sur les clauses elles-mêmes, mais ceci est aussi vrai pour toute formule du calcul des prédicats du premier ordre, on ne peut pas écrire que "A<--(B<--C)" ou encore que "A si et seulement si B$\vee$C", etc...

II. METHODE DE RESOLUTION DE ROBINSON (CHI73).

Rappelons qu'une formule e est valide si et seulement si non e est insatisfiable (ou inconsistante). Aussi, pour prouver la validité d'une formule e sous forme clausale $e_1 \wedge e_2 \wedge \ldots \wedge e_n$, il faut prouver l'insatisfaisabilité de non $e_1 \vee$ non $e_2 \vee \ldots \vee$ non e_n.

D'après le théorème de HERBRAND, une formule sous forme clausale est insatisfiable si et seulement si il existe une conjonction finie d'instances de non variables de ses clauses qui est insatisfiable .

Définition : On appelle instance de non variable d'une clause C toute clause déduite de C en substituant chacune de ses variables par un terme de l'univers de HERBRAND engendré par C .

Avec ce théorème, nous disposons donc d'une méthode de démonstration constructive de l'insatisfaisabilité d'une formule sous forme clausale, toutefois sa mise en oeuvre s'avère peu efficace. ROBINSON (ROB65) a amélioré cette méthode en introduisant une règle d'inférence appelée principe de résolution qui génère à partir d'une clause des ensembles de clauses comprenant ces mêmes clauses et des clauses appelées résolvants.

Avec la méthode de ROBIN,SON une formule e sous forme clausale est insatisfiable si et seulement si la clause vide notée $\square$ peut être déduite par l'application du principe de résolution. On dit alors que l'on réfute e.

Exemple : Prouvons par réfutation que U est une conséquence logique de l'ensemble de clauses suivant:

(1) +S-P (2) +U-S (3) +P (4) +U

On commence par réfuter la clause +U, ce qui engendre l'ensemble des

clauses

(1) +S-P (2) +U-S (3) +P (4)' -U

Puis de (1) et (3) on déduit le résolvant (5) +S

de (2) et (5) on déduit le résolvant (6) +U

et de (6) et (4)' on déduit le résolvant (7) □ .

La clause □ est insatisfiable puisqu'elle ne contient aucun terme non variable et donc ne peut être vérifiée dans aucune interprétation.

Si la formule à démontrer est valide, la démonstration termine, mais il se peut qu'elle ne termine pas dans les limites de temps de calcul fixées . Dans ce cas, on ne peut malheureusement rien conclure quant à la validité de la formule.

Cette méthode de démonstration semi-décidable n'est intéressante que dans la mesure où le processus de résolution termine le plus souvent possible dans des limites de temps "raisonnables". Pour améliorer cette méthode de résolution on peut d'abord limiter l'espace de recherche de toutes les clauses dérivables par résolution c'est-à-dire utiliser une stratégie de résolution efficace, mais aussi supprimer la génération de clauses inutiles, en utilisant des heuristiques de génération de l'espace de dérivation.

Dans l'exemple ci-dessus, nous avons utilisé une stratégie de résolution dite linéaire . Avec cette stratégie tout à fait classique, on commence par réfuter la clause à démontrer puis on cherche à unifier, à chaque étape de dérivation, le dernier résolvant obtenu avec une partie gauche des autres clauses. Le processus s'arrête avec la dérivation de la clause □ . Cette méthode de résolution est dite descendante.

On peut aussi utiliser une stratégie dite ascendante . Dans ce cas, on part d'une clause +U (assertion du théorème à démontrer) et ensuite on cherche des substitutions qui unifient à la fois les parties droites d'une clause et certaines parties gauches des autres clauses. Le processus s'arrête avec la clause vide.

Cette dernière stratégie utilise une fonction de sélection du littéral à résoudre, qui est du type "dernier entré premier sorti". Nous ne discutons pas des heuristiques de génération d'espace de recherche délimité par une stratégie de résolution. Le lecteur intéressé trouvera des informations détaillées dans les ouvrages de CHIN-LIANG CHANG et RICHARD CHAR TUNGLEE (CHI73) et de R.KOWALSKI (KOW79).

III. PROLOG. (ROU75) (BAR82)

L'objectif de cette section est de présenter quelques traits du langage PROLOG que nous avons utilisé pour réaliser le système VEGA. Il s'agit de la version développée à Marseille par A.COLMERAUER et son équipe (ROU75). Le lecteur intéressé par les nouvelles versions de ce langage pourra se reporter aux références citées au chapitre VI. ainsi qu'à (WAR77) (LAU82).

Syntaxiquement, un programme PROLOG se présente sous la forme d'une séquence de clauses de Horn, avec en plus quelques caractères spéciaux de contrôle que nous précisons dans la suite.

PROLOG utilise mais aussi permet d'écrire des clauses plus générales que les clauses de Horn classiques:

-- Plusieurs prédicats prédéfinis appelés prédicats évaluables sont utilisés par le traducteur. Ces prédicats permettent d'effectuer des opérations spéciales qu'il serait difficile voire même impossible de définir en termes de clauses de Horn:

-- entrée/sortie de caractères, de termes, de clauses,

-- opérations arithmétiques et opérations sur les caractères,

-- modification dynamique d'un programme par ajout ou suppression de clauses,

-- accès dynamique à la syntaxe des unités,

-- accès et modification de l'état de contrôle des programmes,

-- etc...

Ajoutons que les clauses, à la différence des clauses définies par un programme ne seront pas détruites par le mécanisme de retour.

-- Le traducteur appelé superviseur admet des clauses dont les arguments peuvent être:

-- des prédicats: +P(*X)-*X

-- des termes fonctionnels: +F(.(*X,*Y),*Z)

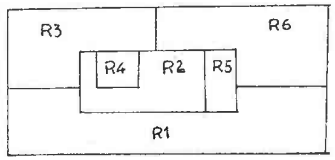
-- des expressions; mais alors tous les opérateurs utilisés doivent être définis par le programme, à l'aide d'une clause spéciale appelée AJOP:

```
-AJOP(".",8,"X^(X^X)")
+E(*X.*Y,*Z)
```

Une demande d'exécution de programme (soumission à l'interpréteur) se fait en écrivant une liste de prédicats précédés du symbole "-", et se terminant par le caractère spécial "]".

Exemple :(tiré de J.CHABIER (CHA82))

Nous décrivons en PROLOG le coloriage avec quatre couleurs des six zones (R1,R2,R3,R4,R5,R6) de la figure ci-dessous, en imposant que deux zones contigues n'aient pas la même couleur.



** On définit des voisinages de couleurs**.

```
+ESTVOIS(BLEU,JAUNE).
+ESTVOIS(BLEU,ROUGE).
```

```
        +ESTVOIS(BLEU,VERT).
        +ESTVOIS(JAUNE,ROUGE).
        +ESTVOIS(JAUNE,VERT).
        +ESTVOIS(ROUGE,VERT).
** On donne la règle décrivant la figure**.
        +FIG(*R1,*R2,*R3,*R4,*R5,*R6)
          -ESTVOIS(*R1,*R2)-ESTVOIS(*R1,*R3)
          -ESTVOIS(*R1,*R5)-ESTVOIS(*R1,*R6)
          -ESTVOIS(*R2,*R3)-ESTVOIS(*R2,*R4)
          -ESTVOIS(*R2,*R5)-ESTVOIS(*R2,*R6)
          -ESTVOIS(*R3,*R4)-ESTVOIS(*R3,*R6)
          *ESTVOIS(*R5,*R6).
** Maintenant, on peut connaître toutes les possibilités de colorier
cette figure en écrivant**.
        -FIG(*U1,*U2,*U3,*U4,*U5,*U6)
          -SORM("R1")-SORT(*U1)-LIGNE
          -SORM("R2")-SORT(*U2)-LIGNE
          -SORM("R3")-SORT(*U3)-LIGNE
          -SORM("R4")-SORT(*U4)-LIGNE
          -SORM("R5")-SORT(*U5)-LIGNE
          -SORM("R6")-SORT(*U6)-LIGNE
```

Les prédicats SORM, SORT et LIGNE sont des prédicats évaluables.

En PROLOG, les arguments des prédicats peuvent être indifféremment des données ou des résultats. PROLOG est un langage non déterministe .

Exemple : Reprenons l'exemple de la concaténation de deux listes décrit en I. et remplaçons le symbole fonctionnel cons par l'opérateur ".". Soit:

```
+CONC(NIL,*L,*L).
+CONC(*X.*L1,*L2,*X.*L3)-CONC(*L1,*L2,*L3).
```

Ce programme permet de calculer la liste résultat de la concaténation de deux listes:

-CONC(A.B.NIL,C.NIL,*R) détermine *R comme étant la liste A.B.C.NIL

ou, par exemple, de calculer la liste *C qui, concaténée à la liste A.B.NIL, produit la liste A.B.C.NIL

-CONC(A.B.NIL,*R,A.B.C.NIL) détermine *R comme étant la liste C.NIL.

Notons qu'ici, le calcul des résultats s'effectue par approximations successives.

Les trois mécanismes de base de l'interpréteur PROLOG sont:

-- le filtrage qui utilise un algorithme d'unification opérant sur des représentations canoniques de termes appelées "listes peignées",

-- la résolution linéaire qui met en oeuvre la stratégie dite de LUSH résolution (KOW79), la sélection des littéraux à unifier se faisant selon l'ordre d'écriture des clauses,

-- et une heuristique de génération en profondeur de l'arbre de dérivation (KOW79).

Nous allons montrer, à partir d'exemples simples, de quelle manière opèrent les algorithmes d'unification (1) et de résolution (2).

(1) L' unification . En PROLOG, l'unification de deux termes t_1 et t_2 consiste à trouver la substitution minimale θ telle que

$\theta(t_1)=\theta(t_2)$.

On se donne le prédicat +EGAL(*X,*X). Alors:

** l'unification de deux symboles de constantes ne réussit que si les deux symboles sont identiques

-EGAL(a,a) réussit

-EGAL(a,b) échoue

** l'unification d'une variable et d'une constante

-EGAL(a,*X1) réussit avec la substitution $\theta = [*X1 \text{-->} a]$

** l'unification de deux variables libres

-EGAL(*X,*X) réussit et

-EGAL(*X,*X1) réussit avec la substitution $\theta = [*X \text{-->} *X2, *X1 \text{-->} *X2]$

** l'unification de deux termes comportant des opérateurs

-EGAL(a.b.c,*X1.*X2) réussit avec la substitution $\theta = [*X1 \text{-->} a, *X2 \text{-->} b.c]$

** l'unification d'un terme avec un terme fonctionnel

-EGAL(*X,f(*X1)) réussit avec la substitution $\theta = [*X \text{-->} f(*X1)]$

-EGAL(*X1,f(*X1)) produit une infinité de substitutions $\theta = [*X1 \text{-->} f(*X1)]$

PROLOG n'a pas d'"occurs check" qui permette de contrôler ce type de boucle à l'unification.

(2) La résolution . La stratégie de résolution de PROLOG explore systématiquement tous les choix possibles. On peut toutefois modifier l'état du contrôle en éliminant certains points de retour en utilisant les prédicats:

/(slash) qui supprime tous les choix possibles pour tous les

noeuds accumulés entre l'appel de la clause contenant ce prédicat et l'appel de ce prédicat

/(<littéral sous forme de terme>) qui supprime tous les choix pour tous les noeuds accumulés depuis la plus récente occurrence de ce littéral ancêtre, y compris celui-ci

IMPASSE qui produit un échec.

Considérons en plus du prédicat d'égalité, les prédicats évaluables d'impression suivants: LIGNE qui effectue un saut de ligne et SORT(*X) qui imprime le terme *X.

Nous laissons le soin au lecteur d'analyser les exécutions suivantes (BAR78):

+EGAL(*1,*1).

+N(0).

+N(1).

+N(2).

-N(*1)-SORT(*1)-LIGNE] produit le résultat

0

1

2

-N(*1)-N(*2)-SORT(*1)-SORT(*2)-LIGNE] produit le résultat

00

01

02

10

11

12

```
20
21
22
-N(*1)-SORT(*1)-N(*2)-SORT(*2)-LIGNE] produit le résultat
00
1
2
10
1
2
20
1
2
-N(*1)-/-N(*2)-SORT(*1)-SORT(*2)-LIGNE] produit le résultat
00
01
02
-N(*1)-EGAL(*1,1)-/-N(*2)-SORT(*1)-SORT(*2)-LIGNE ] produit
10
11
12
-N(*1)-EGAL(*1,1)-N(*2)-/-SORT(*1)-SORT(*2)] produit
10
-N(*1)-/-SORT(*1)]  produit
0
-N(*1)-SORT(*1)-/ ] produit
0
```


Ces différents mécanismes font de PROLOG un langage très adapté au traitement de données symboliques comme la compréhension des langues naturelles, la résolution de problèmes, la formation de plans en robotique, les bases de données déductives, les systèmes experts en recherche pharmaceutique, etc...(GAL77) (LAU82). Certains pensent même que PROLOG est un outil plus approprié que LISP pour tout ce qui concerne la représentation et la manipulation des connaissances (WAR77). Il est vrai que les nouveaux systèmes PROLOG: IC-PROLOG, T-PROLOG, M-PROLOG, PROLOG II, etc... sont devenus fiables et souvent très efficaces.

Par contre, nous avons dû faire preuve d'une grande sagacité pour développer VEGA avec la version FORTRAN de P.ROUSSEL car ce PROLOG expérimental exige beaucoup de mémoire; les messages d'erreur du type "la pile est pleine" ou "débordement de pile" sont tout à fait insuffisants. En outre, ce PROLOG ne fait aucun contrôle de "boucle", ce qui est très génant avec ce genre de programmation.

CHAPITRE IX. VEGA.

I. INTRODUCTION.

II. PRESENTATION DES COMPOSANTS DU SYSTEME VEGA.

II.1. Le langage de spécification algébrique VEGA.

II.1.1. Spécification des SYSTEMes avec la technique axiomatique.

II.1.2. Spécification des SYSTEMes avec la technique du modèle abstrait.

II.2. L'environnement de spécification initial du système VEGA.

II.2.1. Le SYSTEMe BOOL.

II.2.2. Le SYSTEMe CHAR.

II.2.3. Le SYSTEMe INT.

II.2.4. Le SYSTEMe STRING.

II.2.5. SYSTEMes TUPLE.

II.2.6. SYSTEMes SET.

II.2.7. SYSTEMes SEQ.

II.2.8. Remarques.

II.3. Instructions de traitement de données symboliques.

II.3.1. Instruction RUN.

II.3.2. Instruction MRUN.

II.3.2.1. L'instruction EACH.

II.3.2.2. L'instruction FIRST.

II.3.2.3. Remarques.

III. MECANISMES DE CONSTRUCTION DE SPECIFICATIONS.

III.1. Dérivation de sous-types abstraits avec erreurs.

III.1.1. Clause RESTRICTO.

III.1.2. Clause INVARIANT.

III.1.3. Remarques.

III.2. Extensions de spécifications avec coercition et surcharge des

opérateurs.

III.3. Restriction des spécifications avec renommage des opérations.

III.4. Combinaison de spécifications.

III.5. Un exemple de spécification de système de base de données, de construction et de manipulation d'une maquette de données symboliques.

III.5.1. Spécification de l'application.

III.5.2. Elaboration d'une maquette de système de base de données.

III.5.3. Manipulation de la maquette.

IV. IMPLANTATION DE VEGA.

IV.1. Le compilateur.

IV.2. L'interpréteur RUN.

IV.3. L'interpréteur MRUN.

IV.4. Le bibliothécaire.

IV.5. L'éditeur de textes.

IV.6. L'environnement du système VEGA.

I. INTRODUCTION.

VEGA est un système expérimental d'aide à la spécification algébrique de types abstraits de données et de programmes et à la validation de leurs propriétés: correction et fidélité aux intentions initiales. C'est aussi un système interactif de programmation modulaire qui comporte des constructions de langage pour fabriquer des maquettes symboliques de bases de données et de bases de connaissances.

Le langage de spécification algébrique VEGA est un langage formel qui a la particularité de supporter deux techniques de spécification formelle:

-- une technique axiomatique (LIS77) qui permet de caractériser les propriétés des types abstraits de données et/ou des programmes par des axiomes équationnels. Les "structures de données profondes" des types de données des spécifications sont ici implicitement définies par des opérations particulières appelées constructeurs,

-- une technique "modèle abstrait" (LIS75) de type ALPHARD (WUL76) ou pré/post (FIN79) mais avec un cadre formel algébrique. En VEGA, la fonction d'abstraction du modèle abstrait vers le type abstrait et les propriétés des opérations sont définies avec des équations (de définition et de propagation des erreurs) conditionnelles. Avec cette technique, les propriétés des types abstraits de données sont définies sur des structures de données explicites (modèles abstraits, tuples,

ensembles, séquences, etc...).

Le langage formel VEGA supporte aussi plusieurs mécanismes de construction de spécifications : la "dérivation" de sous-types abstraits de données, l'extension de spécifications avec coercition et surcharge des opérateurs, la restriction de spécifications avec renommage des opérateurs, la "combinaison" dans une même spécification de plusieurs spécifications de types abstraits.

VEGA est aussi un langage de programmation fonctionnel fortement typé qui fournit des facilités pour tester et mettre au point des spécifications algébriques.

VEGA est un système "multicibles" . En effet, les spécifications algébriques de types abstraits de données et de programmes VEGA, une fois compilées, peuvent très facilement être interfacées par le langage hôte PROLOG. Cette technique originale est particulièrement intéressante car de cette façon, on peut utiliser le mécanisme d'inférence de PROLOG pour définir et résoudre des problèmes sur des (classes) d'univers (PAI74) qui ont été spécifiés et validés séparément à l'aide du système autonome VEGA. Cette facilité fait de VEGA un système de programmation logique supportant des spécifications algébriques de types abstraits de données et de programmes .

Notre participation au projet MONT-LOZERE (Ministère du Tourisme, ADI, Service Loisirs-Accueil de Lozère, SYSECA, équipe CHORUS de l'INRIA, Société BERTIN et CRIN) a permis d'expérimenter cette technique sur une application relativement sophistiquée: une maquette de système expert d'aide à l'information touristique et à la réservation de gîtes ruraux en Lozère. Cette maquette conçue et réalisée avec la collaboration de Jacqueline CHABRIER, Nacer BOUJLIDA, Jacques DURAND et Jacques BOURST a fait l'objet d'une démonstration en Juin 82 au Service Loisirs-Accueil de Mende. C'est à la suite de cette expérimentation que nous avons décidé d'introduire dans VEGA des constructions de langage plus adaptées à la manipulation de données symboliques de type tuples, ensembles, séquences, etc...

La section II. de ce chapitre est dévolue à la présentation des différents composants du système autonome VEGA: le langage de spécification algébrique avec erreurs, l'environnement initial du système et les principaux traits du langage de manipulation de données symboliques. Nous exposons ensuite en section III., les mécanismes de construction de spécifications supportées par le langage formel, en donnant de nombreux exemples. Un exemple de spécification et de manipulation de base de données symboliques est aussi décrit. En section IV., nous discutons de quelques aspects particuliers de l'implantation du système.

II. PRESENTATION DES COMPOSANTS DU SYSTEME VEGA.

II.1. Le langage de spécification algébrique VEGA.

Une spécification VEGA de type abstrait de données et/ou de programme est une unité de texte appelée SYSTEM , décrite avec le schéma syntaxique 1: technique axiomatique, ou avec le schéma syntaxique 2: technique modèle abstrait.

Schéma 1	Schéma 2
SYSTEM (<nom-de-la-spécification>)/	SYSTEM (<nom-de-la-spécification>)/
SORTS (<noms-des-sortes -définies>)/	SORTS (<nom-de-la-sorte -dèfinie>)/
IMPORTS (<noms-des-sortes -externes>)/	MODEL (<nom-du-modèle -abstrait>)/
BUILD /	OKOPNS /
co liste des opérateurs constructeurs oc	co liste des opérateurs définis oc
OKOPNS /	EROPNS /
co liste des opérateurs définis oc	co liste des opérateurs erreurs oc
EROPNS /	AFUNC /
co liste des opérateurs erreurs oc	co liste d'axiomes définissant la fonction d'abstraction oc
OKAXS /	OKSEMS /
co liste des axiomes de définition oc	co liste des axiomes de définition oc
ERAXS /	ERSEMS /
co liste des axiomes de propagation d'erreurs oc	co liste des axiomes d'erreurs oc
END (<nom-de-la-spécification>)/	END (<nom-de-la-spécification>)/

Dans les paragraphes qui suivent, nous précisons la syntaxe et la sémantique de ces deux formes de SYSTEMe. Les différents composants de chaque schéma syntaxique sont illustrés avec des exemples que l'on a l'habitude de trouver dans la plupart des articles et revues sur la spécification. Ce n'est qu'après avoir présenté le langage VEGA que nous détaillons un exemple de spécification et de manipulation de base de données: une gestion (partielle) de prêts de livres à des chercheurs. Cette application a aussi été traitée avec le système de programmation modulaire TYP (troisième partie).

II.1.1. Spécification des SYSTEMes avec la technique axiomatique.

Les deux exemples de spécification décrits ci-dessous permettent d'illustrer simplement différents aspects du schéma syntaxique 1. L'exemple 1 est une spécification avec erreurs d'un type abstrait de données pile d'entiers. L'exemple 2 est une spécification avec erreurs du PGCD de deux nombres entiers (les listings de ces spécifications sont donnés en Annexe). Pour ce type de SYSTEMe, nous avons défini une syntaxe qui ressemble à celle du langage OBJ-0 développé par J.GOGDO (TAR79).

Exemple 1 : Spécification avec erreurs du type abstrait de données pile d'entiers.

```
SYSTEM (PILINT)/
   SORTS (PILINT)/ IMPORTS (INT;BOOL)/
BUILD /
   PILINT(VIDE <- PILINT)/
   PILINT(EMPILER(PILINT,INT) <- PILINT)/
OKOPNS /
   PILINT(DEPILER(PILINT) <- PILINT)/
   PILINT(SOMMET(PILINT) <- INT)/
   PILINT(ESTVIDE(PILINT) <- BOOL)/
EROPNS /
   PILINT(SOMERR <- INT)/
   PILINT(PILERR <- PILINT)/
OKAXS /
   PILINT(DEPILER(EMPILER(*P,*I)) = *P)/
   INT(SOMMET(EMPILER(*P,*I)) = *I)/
   BOOL(ESTVIDE(VIDE) = TRUE)/
   BOOL(ESTVIDE(EMPILER(*P,*I)) = FALSE)/
ERAXS /
   PILINT(DEPILER(VIDE) = PILERR)/
   INT(SOMMET(VIDE) = SOMERR)/
END (PILINT)/
```


Exemple 2 : Spécification avec erreurs du PGCD de deux nombres entiers.

```
SYSTEM (PGCD)/
   IMPORTS (INT;BOOL)/
OKOPNS /
   PGCD(PGCD(INT,INT) <- INT)/
EROPNS /
   PGCD(ARGNEG <- INT)/
OKAXS /
   INT(PGCD(*I,*I) = *I)/
   INT(PGCD(*I,0) = *I)/
   INT(PGCD(0,*I) = *I)/
   INT(PGCD(*I,*J) = IT(INT(*J<*I)∧INT(*J>0),
                        PGCD(*I-*J<*J)))/
   INT(PGCD(*I,*J) = IT(INT(*I<*J)∧INT(*I>0),
                        PGCD(*I,*J-*I)))/
ERAXS /
   INT(PGCD(*I,*J) = IT(INT(*I<0) OR INT(*J<0),ARGNEG))/
END (PGCD)/
```

La présentation d'un SYSTEMe selon l'approche axiomatique comporte un entête de SYSTEMe, une signature, des équations et une fin de SYSTEMe.

L'entête est formé du mot-clé SYSTEM suivi, entre parenthèses, d'un identificateur qui définit de façon unique (dans un contexte de spécification donné), son nom.

La signature est constituée de plusieurs paragraphes dans lesquels sont respectivement déclarés les sortes (SORTS / IMPORTS), les

profils des constructeurs (BUILD), les profils des opérateurs définis (OKOPNS) et les profils des opérateurs erreurs (EROPNS).

Les noms de sortes d'un SYSTEMe sont introduits par le mot-clé SORTS avec, entre parenthèses, la liste des noms de sortes définies, suivi du mot-clé IMPORTS avec, entre parenthèses, la liste des noms de sortes externes .

(Exemple 1) SORTS (PILINT)/ IMPORTS (INT;BOOL)/

Si un SYSTEMe n'a qu'une sorte définie, le nom de cette sorte peut être le même que celui du SYSTEMe. Lorsqu'un SYSTEMe a une sorte définie (au moins une), on doit alors décrire la liste des constructeurs (au moins deux) dans un paragraphe introduit par le mot-clé BUILD . Le profil d'un constructeur comporte un nom d'opérateur, puis entre parenthèses, la liste des sortes de ses arguments, le symbole "<-" suivi du nom de la sorte définie. Une déclaration de constructeur est alors un profil d'opérateur introduit par un "type" (nom de sorte ou de système) de façon à éviter des ambiguités en cas de surcharge de l'opérateur. Ajoutons que le nom de la sorte définie doit aussi figurer dans la liste des arguments des constructeurs.

(Exemple 1) PILINT(EMPILER(PILINT,INT) <- PILINT)/

Si un SYSTEMe n'a pas de sorte définie, il n'y a évidemment pas de paragraphe BUILD . C'est le cas de l'exemple 2.

Les opérateurs définis d'un SYSTEMe sont déclarés dans un paragraphe introduit par le mot-clé OKOPNS . Les déclarations des profils de ces opérateurs sont décrites comme dans le paragraphe BUILD. Précisons que ce paragraphe est optionnel.

(Exemple 1) PILINT(DEPILER(PILINT) <- PILINT)/

Les opérateurs erreurs d'un SYSTEMe sont déclarés dans un paragraphe introduit par le mot-clé EROPNS . Ici, les opérateurs définissent des constantes qui servent par ailleurs à générer des messages d'erreurs significatifs. Ce paragraphe est aussi optionnel.

(Exemple 1) PILINT(SOMERR <- INT)/

La partie définition des axiomes équationnels d'un SYSTEMe comporte deux paragraphes optionnels, introduits respectivement par les mots-clés OKAXS pour les équations de définition et ERAXS pour les équations de propagation des erreurs.

Les équations de définition sont:

.soit des équations inconditionnelles de la forme:

(Exemple 1) PILINT(DEPILER(EMPILER(*P,*I)) = *P)/

.soit des équations conditionnelles, c'est-à-dire des équations dont la partie droite est un "si-alors" noté II ou un "si-alors-sinon" noté ITE.

(Exemple 2) INT(PGCD(*I,*J)= II (INT(*J<*I)ᐧINT(*I>0),

PGCD(*I-*J,*J)))/

où les expressions de part et d'autre du symbole "=" sont de même sorte. Pour éviter des ambiguités de surcharge d'opérateur, chaque équation est préfixée par la sorte des expressions. Les variables formelles, identificateurs précédés d'une astérisque "*", qui figurent dans chaque équation sont implicitement déclarées. La sorte de ces variables est déterminée à partir de leur occurrence en tant qu'arguments d'opérations en partie gauche des équations. Les sortes des variables de chaque équation sont définies conformément aux profils

syntaxiques des opérations. En outre, les équations de définition ne doivent en aucun cas utiliser des opérateurs erreurs (EROPNS).

Les équations de propagation d'erreurs sont, comme précédemment, des équations inconditionnelles

(Exemple 1) PILINT(SOMMET(VIDE)=SOMERR)/

ou des équations conditionnelles de la forme IT ou ITE :

(Exemple 2) INT(PGCD(*I,*J)= IT (INT(*I<0) OR INT(*J<0),ARGNEG))/

Dans les équations de propagation d'erreurs, la partie droite des équations ne doit être formée que d'expressions erreurs.

Enfin, la description d'un SYSTEMe se termine par le mot-clé END suivi, entre parenthèses, du nom du SYSTEMe.

Nous donnons en Annexe deux autres exemples de spécification: le système FILE et le système FACT (factorielle).

Pour préciser la sémantique formelle d'un SYSTEMe, nous allons considérer l'exemple 1.

Notons PILINT1 l'algèbre avec erreurs $(\mathcal{S}_{PILINT1}, \Sigma^{OK}_{PILINT1}, \Sigma^{ER}_{PILINT1}, E^{OK}_{PILINT1}, E^{ER}_{PILINT1})$ spéecifié par le SYSTEMe PILINT. Pour simplifier les notations, nous avons assimilé les constructeurs à des opérations OK. Alors,

$--\mathcal{S}_{PILINT1} = \{ PILINT^{OK \cup ER}_{PILINT1}, BOOL^{OK \cup ER}_{PILINT1}, INT^{OK \cup ER}_{PILINT1} \}$

$--\Sigma^{OK}_{\lambda, PILINT^{OK}_{PILINT1}} = \{ VIDE_{PILINT1} \}$,

$\Sigma^{OK}_{PILINT^{OK}_{PILINT1}\ INT^{OK}_{PILINT1}, PILINT^{OK}_{PILINT1}} = \{ EMPILER_{PILINT1} \}$

$\Sigma^{OK}_{PILINT^{OK}_{PILINT1}, PILINT^{OK}_{PILINT1}} = \{ DEPILER_{PILINT1} \}$,

$\Sigma^{OK}_{PILINT^{OK}_{PILINT1}, INT^{OK}_{PILINT1}} = \{ SOMMET_{PILINT1} \}$,

$\Sigma^{OK}_{PILINT^{OK}_{PILINT1}, BOOL^{OK}_{PILINT1}} = \{ ESTVIDE_{PILINT1} \}$,

$--\Sigma^{ER}_{\lambda, INT^{ER}_{PILINT1}} = \{ SOMERR_{PILINT1} \}$,

$\Sigma^{ER}_{\lambda, PILINT^{ER}_{PILINT1}} = \{ PILERR_{PILINT1} \}$.

Il est alors clair que $BOOL^{OK}_{PILINT1} = \{ TRUE, FALSE \}$ et $INT^{OK}_{PILINT1} = Z$ avec les opérations habituelles (+,-,*,/) et $PILINT^{OK}_{PILINT1} = \{ VIDE, EMPILER(VIDE,i_1), \ldots, EMPILER(EMPILER(\ldots(VIDE,i_1),\ldots,i_n) \}$ où $i_1,\ldots,i_n$ sont des éléments de $INT^{OK}_{PILINT1}$. Les éléments de $PILINT^{OK}_{PILINT1}$ sont en fait les termes canoniques engendrés par les constructeurs VIDE et EMPILER.

Les ensembles des supports $BOOL^{ER}_{PILINT1}$, $INT^{ER}_{PILINT1}$ et $PILINT^{ER}_{PILINT1}$ sont définis de la manière suivante:

$--BOOL^{ER}_{PILINT1} = \{ ESTVIDE_{PILINT1}(I)$ tels que $I \in PILINT^{ER}_{PILINT1} \}$,

$--INT^{ER}_{PILINT1} = \{ SOMERR_{PILINT1} \} \cup \{ SOMMET_{PILINT1}(EMPILER_{PILINT1}(P,I))$ tels que $I \in INT^{ER}_{PILINT1}$ et $P \in PILINT^{OK}_{PILINT1} \}$

$\cup \{ SOMMET_{PILINT1}(EMPILER_{PILINT1}(P,I))$ tels que $P \in PILINT^{ER}_{PILINT1}$ et $I \in INT^{OK}_{PILINT1} \}$

$\cup \{ SOMMET_{PILINT1}(EMPILER_{PILINT1}(P,I))$ tels que $P \in PILINT^{ER}_{PILINT1}$ et $I \in INT^{ER}_{PILINT1} \}$,

$--PILINT^{ER}_{PILINT1} = \{ PILERR_{PILINT1} \} \cup$

$\{ DEPILER_{PILINT1}(DEPILER_{PILINT1}(\ldots (DEPILER_{PILINT1}(P))\ldots)$ tels que $P \in PILINT^{ER}_{PILINT1} \} \cup$

$\{ EMPILER_{PILINT1}(P,I)$ tels que $P \in PILINT^{ER}_{PILINT1}$ et $I \in INT^{OK}_{PILINT1} \} \cup$

$\{ EMPILER_{PILINT1}(P,I)$ tels que $I \in INT^{ER}_{PILINT1}$ et $P \in PILINT^{OK}_{PILINT1} \} \cup$

$\{ EMPILER_{PILINT1}(P,I)$ tels que $P \in PILINT^{ER}_{PILINT1}$ et $I \in INT^{ER}_{PILINT1} \}$.

Ainsi définie, l'algèbre avec erreurs PILINT1 est d'après le théorème énoncé au chapitre VII. isomorphe à l'algèbre $T_{\Sigma_{PILINT}^{BUILD} \cup \Sigma_{PILINT}^{OK} \cup \Sigma_{PILINT}^{ER} / \equiv}$ PILINT initiale dans la classe des PILINT-algèbres vérifiant la propriété de propagation des erreurs.

Ainsi, le type abstrait spécifié par le SYSTEMe PILINT est la Σ-algèbre T_{PILINT} vérifiant la propriété de propagation des erreurs.

II.1.2. Spécification des SYSTEMes avec la technique du modèle abstrait.

Avant d'aborder de façon plus précise la syntaxe et la sémantique de ce type de SYSTEMe, nous commençons par décrire un exemple de spécification de Pile d'entiers sur le modèle abstrait Séquence d'entiers, illustrant ansi le schéma syntaxique 2. Les opérations de Séquence d'entiers ADG, DELG, CAR, EMPTY qui figurent dans le paragraphe OKSEMS signifient respectivement: ajouter à gauche un entier, enlever à gauche un entier, accès au premier élément à gauche, est vide. L'opération REP, appelée fonction d'abstraction de Séquence d'entiers vers Pile d'entiers sera précisée dans la suite.

Exemple:

```
SYSTEM (SPINT)/
   SORTS (SPINT)/ MODEL (SEQ(INT))/
OKOPNS /
   SPINT(VIDE <- SPINT)/
   SPINT(EMPILER(SPINT,INT) <- SPINT)/
   SPINT(DEPILER(SPINT) <- SPINT)/
   SPINT(SOMMET(SPINT) <- INT)/
   SPINT(ESTVIDE(SPINT) <- BOOL)/
EROPNS /
   SPINT(SOMERR <- INT)/
   SPINT(PILERR <- SPINT)/
AFUNC /
   SPINT( REP (NIL) = VIDE)/
   SPINT( REP (*I.*S) = EMPILER( REP (*S),*I))/
OKSEMS /
   SPINT(VIDE = REP (NIL))/
   SPINT(EMPILER( REP (*P),*I) = REP (ADG(*P,*I)))/
   SPINT(DEPILER( REP (*P)) = REP (DELG(*P)))/
   INT(SOMMET( REP (*P)) = CAR(*P))/
   BOOL(EMPTY( REP (*P)) = EMPTY(*P))/
ERSEMS /
   SPINT(DEPILER(VIDE) = PILERR)/
   INT(SOMMET(VIDE)=SOMERR)/
END (SPINT)/
```


On voit que cette seconde technique de spécification revient à spécifier les opérations du SYSTEMe PILINT en terme d'opérations associées du modèle abstrait SEQ(INT). Plus précisément, si l'on note $P=(\mathcal{S}_{PILINT}, \Sigma^{OK}_{PILINT}, \Sigma^{ER}_{PILINT})$ la signature du SYSTEMe PILINT, alors le type abstrait spécifié par le SYSTEMe PILINT est isomorphe à un sous-ensemble de la P-algèbre avec erreurs définie sur SEQ(INT) avec les opérations "renommées" de P. L'opération REP représente un homomorphisme surjectif de SEQ(INT) vers PILINT, justifiant notre présentation.

La présentation d'un SYSTEMe selon l'approche modèle abstrait comporte un entête de SYSTEMe, une signature, des équations et une fin

de SYSTEMe. L'entête est constitué du mot-clé SYSTEM suivi, entre parenthèses, d'un identificateur qui définit de façon unique (dans un contexte de spécification donné) son nom. La signature comporte trois paragraphes dans lesquels sont respectivement décrits la sorte définie (SORTS) et le modèle abstrait (MODEL), les profils des opérations du type abstrait (OKOPNS) et les profils des opérations erreurs (EROPNS). Dans ce type SYSTEMe, on ne distingue pas les constructeurs des autres opérations.

Les noms de sortes d'un SYSTEMe sont introduits par le mot-clé SORTS avec, entre parenthèses, le nom de la sorte définie , suivi du mot-clé MODEL avec, entre parenthèses, le nom de sorte (ou de la construction de sorte) associé au modèle abstrait.

(Exemple) SORTS (SPINT)/ MODEL (SEQ(INT))/

Nous montrerons en III qu'il est possible d'attacher des propriétés sur l'argument INT (restriction de l'ensemble des valeurs) et/ou sur SEQ(INT) en définissant un invariant. Les paragraphes OKOPNS et EROPNS se décrivent de la même manière que dans le premier type de SYSTEMe en II.1.1.

La partie définition des axiomes équationnels comporte trois paragraphes optionnels, introduits respectivement par les mots-clés AFUNC pour les équations de la fonction d'abstraction, OKSEMS pour les équations de définition de la représentation et ERSEMS pour les équations de propagation des erreurs.

Les équations inconditionnelles du paragraphe AFUNC associent deux à deux les constructeurs du type abstrait spécifié et du modèle abstrait.

(Exemple) SPINT(REP (NIL)=VIDE)/
SPINT(REP (*I.*S)=EMPILER(*S,*I))/

Nous préfixons ici aussi les équations de AFUNC pour éviter des ambiguités de surcharge d'opérateurs. Notons aussi que le nom de la fonction d'abstraction est un nom réservé du langage de spécification. Les équations qui caractérisent la représentation (ou l'implantation) sont:

.soit des équations inconditionnelles de la forme:

(Exemple) SPINT(EMPILER(REP (*P),*I)= REP (ADG(*P,*I)))/
BOOL(EMPTY(REP (*P))=EMPTY(*P))/

.soit des équations conditionnelles, c'est-à-dire des équations dont la partie droite comporte un "si-alors" noté II ou un "si-alors-sinon" noté ITE .

Comme pour l'autre type de SYSTEMe, il est possible d'imbriquer des II et des ITE . Il est clair que ces formes d'équations fournissent une spécification pré/post des opérations du type abstrait (FIN79). En outre, si la sorte associée au modèle abstrait possède une propriété (restriction et/ou invariant), alors chaque équation de AFUNC et de OKSEMS devra aussi vérifier cette propriété.

Les variables formelles qui figurent dans chaque équation sont déclarées de manière implicite, comme pour le premier type de SYSTEMe. De plus, les équations de AFUNC et de OKSEMS ne doivent pas utiliser d'opérateurs erreurs (EROPNS).

Les équations de propagation d'erreurs sont définies comme dans le

premier type de SYSTEMe. Précisons que la partie droite de ces équations ne peut être formée que d'expressions erreurs abstraites. La description du SYSTEMe se termine par le mot-clé END suivi, entre parenthèses, du nom du SYSTEMe.

Maintenant, considérons deux signatures avec erreurs $R_1=(\mathcal{S}_1,\Sigma_1^{OK},\Sigma_1^{ER})$ et $R_2=(\mathcal{S}_2,\Sigma_2^{OK},\Sigma_2^{ER})$.
Les définitions que nous donnons généralisent celles de (REM82).

Définition : On appelle morphisme de signature avec erreurs une application $\rho=(\rho_{\mathcal{S}},\rho_{\Sigma})$ (non nécessairement injective) de $(\mathcal{S}_1,\Sigma_1^{OK},\Sigma_1^{ER})$ dans $(\mathcal{S}_2,T_{\Sigma_2^{OK}\cup\Sigma_2^{ER}}(X))$ telle que pour tout opérateur σ de $\Sigma_1^{OK}\cup\Sigma_1^{ER}$, de profil $\sigma:s_1,..,s_n<-s$, il y ait des variables $x_1,\ldots,x_n$ de sortes respectives $\rho_{\mathcal{S}}(s_1),..,\rho_{\mathcal{S}}(s_n)$ tel que si σ appartient à Σ_1^{OK} alors $\rho_{\Sigma}(\sigma)$ soit un terme à résultat de sorte $\rho_{\mathcal{S}}(s)$ sur les variables $x_1,..,x_n$ et si σ appartient à Σ_1^{ER} alors $\rho_{\Sigma}(\sigma)$ soit un élément erreur de T_{R2}.

Soit ρ un morphisme de la signature avec erreurs R_1 dans R_2. Alors ρ définit une fonction de renommage de l'ensemble des R_1-algèbres dans celui des R_2-algèbres de la manière suivante:

Définition :Soit A_1 une R_1-algèbre, l' algébre renommée $A_1=A_2\cdot\rho$ est définie par

$(A_1)_s=(A_2)_{\rho(s)}$ pour s dans $\mathcal{S}_1$,

$\sigma_{A1}=(\rho(\sigma))_{A2}$ pour σ dans $\Sigma_1^{OK}\cup\Sigma_1^{ER}$, où $\rho(\sigma)_{A1}$ détermine l'application de $(A_1)_{\rho(s1..sn)}$ dans $(A_1)_{\rho(s)}$ qui définit $\rho(\sigma)$.

Soit ρ un morphisme de la signature R_1 dans R_2. Alors ρ se prolonge naturellement aux termes et l'on note encore ρ l'unique morphisme de T_{R1} dans l'algèbre renommée T_{R2}.

Définition : Soit $P=(\mathcal{S}_1,\Sigma_1^{OK},\Sigma_1^{ER},E_1^{OK},E_1^{ER})$ une spécification avec erreurs. Alors une implantation abstraite avec erreurs de P est définie par:

- une R_2-algèbre (modèle abstrait),
- un morphisme de signature aux erreurs ρ de $R_1=(\mathcal{S}_1,\Sigma_1^{OK},\Sigma_1^{ER})$ vers $R_2=(\mathcal{S}_2,\Sigma_2^{OK},\Sigma_2^{ER})$,
- et une R_1-congruence $\equiv$, sur la R_2-algèbre renommée $A_2\cdot\rho$ où T_P est isomorphe à un sous-ensemble de la R_1-algèbre avec erreurs $A_2\cdot\rho/\equiv$

Soit ρ un morphisme de la signature R_1 dans R_2 et A_2 une R_2-algèbre. On appelle support de ρ dans A_2 la sous-algèbre finiment engendrée par $\Sigma_1^{OK}\cup\Sigma_1^{ER}$ dans l'algèbre renommée $A_2\cdot\rho$. Lorsque A_2 est T_{R2}, le support T_{R1} est encore appelé le domaine de ρ. Alors, la spécification des axiomes équationnels AFUNC , OKSEMS et ERSEMS d'un SYSTÉMe S consiste à définir une fonction d'abstraction REP de $T_{\Sigma_2^{OK}}$ vers $T_{\Sigma_1^{OK}}$, par récurrence sur les constructeurs du type d'intérêt du modèle abstrait, ceci dans le paragraphe AFUNC , puis à spécifier dans OKSEMS les axiomes d'implantation des opérateurs σ du paragraphe OKOPNS de S de la manière suivante:

$s(\sigma(\text{REP}(x))=\text{REP}(\rho(\sigma)(x)))$

où s est la sorte des expressions de part et d'autre du symbole "=".

On doit avoir aussi:

$\text{REP}([\rho(t)]_{T_{\Sigma_2^{OK}}})=[t]_{T_{\Sigma_1^{OK}}}$.

Les équations de propagation des erreurs ERSEMS doivent être spécifiées de façon à ce que les parties droites des équations ne comportent que des expressions d'erreurs abstraites.

En outre, si le modèle abstrait possède des propriétés U (restriction et/ou invariant), on doit en plus assurer que:

U(x)==>U(ρ(G)(x))=vrai,

et pour chaque opération de S:

U(x)==>s(G(REP(x))= REP(ρ(G)(x)))

Une discussion plus détaillée de la preuve d'implantation est discutée dans (REM82).

II.2. L'environnement de spécification initial du système VEGA.

Le système VEGA a été avant tout conçu pour fournir des aides à la spécification algébrique de systèmes orientés bases de données et à leur validation. Nous avons donc développé un environnement de spécification initial en fonction de cet objectif. Les SYSTEMes prédéfinis de VEGA sont composés de SYSTEMes standard : booléens (BOOL), caractères (CHAR), chaînes de caractères (STRING) , entiers relatifs (INT) et de systèmes paramétrés surspécifiés : produits cartésiens labellés (TUPLE) , ensembles (SET) et séquences (SEQ) . Les SYSTEMes qui spécifient des types abstraits avec erreurs ont été programmés directement en PROLOG, selon des protocoles conformes à ce que le traducteur du langage de spécification VEGA génère. Ces SYSTEMes ont la particularité de comporter des opérations préfixées, infixées, et pour certains d'entre eux, des "iota-opérations", c'est-à-dire des opérations de la forme x dans E:P(x) où x est la seule variable libre dans P(x). Ces opérations présentent un grand intérêt pour la manipulation symbolique des systèmes de bases de données.

II.2.1. Le SYSTEMe BOOL.

```
SYSTEM (BOOL)/
   SORTS (BOOL)/
BUILD /
   BOOL(TRUE <- BOOL)/
   BOOL(FALSE <- BOOL)/
OKOPNS /
   BOOL(BOOL ↑ BOOL <- BOOL)/
   BOOL('(BOOL) <- BOOL)/
   BOOL(BOOL OR BOOL <- BOOL)/
   BOOL(BOOL IMP BOOL <- BOOL)/
   BOOL(BOOL = BOOL <- BOOL)/
OKAXS /
   BOOL(*B1 ↑ *B2 = ITE (*B1,*B2,FALSE))/
   BOOL(*B1 OR *B2 = ITE (*B1,TRUE,*B2))/
   BOOL('(*B1) = ITE (*B1,FALSE,TRUE))/
   BOOL(*B1 IMP *B2 = ITE (*B1,*B2,TRUE))/
   BOOL(*B1=*B1 = TRUE)/
   BOOL(*B1=*B2 = ITE (*B1,*B2,'(*B2)))/
END (BOOL)/
```

où les opérateurs logiques notés ↑ , OR , ' , IMP , = désignent respectivement le "et", le "ou", la "negation", l'"implication" et l'"équivalence" logique.

Notons que l'opérateur "=" sur les booléens désigne l'opérateur d'équivalence logique sur le domaine des valeurs de vérité, et aussi l'égalité logique sur les booléens. En fait, on interprétera toujours

le symbole "=" comme l'égalité du type abstrait défini. Précisons encore que le SYSTEMe BOOL spécifie en fait un type abstrait BOOL avec erreurs. L'évaluation d'une expression de la forme A ↑ TRUE, A étant un caractère quelconque, produit un message d'erreur "A ↑ TRUE". Quelques exemples sont donnés en Annexe.

II.2.2. Le SYSTEMe CHAR.

Ce système comporte une opération de création de caractère et une opération d'égalité. Pour des raisons d'implantation, la dénotation des 128 caractères ASCII est la suivante:$0,$1,...$9,$A,...$Z,$%,...etc...

II.2.3. Le SYSTEMe INT.

Ce système possède toutes les opérations standard des entiers relatifs: l'addition (+), la soustraction (-), la multiplication (@), la division entière (%), une opération valeur absolue (ABS) et des opérateurs de relation: égalité (=), inférieur (<) et supérieur (>). Un message d'erreur est produit en cas de division par zéro.

```
SYSTEM (INT)/
   SORTS (INT)/ IMPORTS (BOOL)/
OKOPNS /
   INT(INT + INT <- INT)/
   INT(INT - INT <- INT)/
   INT(INT @ INT <- INT)/
   INT(INT % INT <- INT)/
   INT(ABS(INT) <- INT)/
   INT(INT = INT <- BOOL)/
   INT(INT < INT <- BOOL)/
   INT(INT > INT <- BOOL)/
EROPNS /
   INT(INT % 0 <- INT)/
OKAXS /
   .
   .
ERAXS /
   INT(*I % *J = IT (INT(*J=0),*I % 0))/
END (INT)/
```


II.2.4. Le SYSTEMe STRING.

Pour ce type de SYSTEMe, nous avons implanté les opérations de manipulation de chaînes de caractéres de PL/1. NIL représente la chaîne vide, ADD ajoute un caractère à la chaîne, "+" concatène deux chaînes de caractères, SUBSTR(s,i,j) délivre la sous-chaîne de longueur j à partir du ième caractère de la chaîne s, INDEX(s,t) délivre la position de la première occurrence de la sous-chaîne t de la chaîne s, EMPTY teste si la chaîne est vide et LENGTH délivre la longueur de la chaîne. Les constantes chaînes de caractères sont notées "ABC".

```
SYSTEM (STRING)/
   SORTS (STRING)/ IMPORTS (CHAR;BOOL;INT)/
```

```
BUILD /
   STRING(NIL <- STRING)/
   STRING(ADD(STRING,CHAR) <- STRING)/
OKOPNS /
   STRING(STRING + STRING <- STRING)/
   STRING(SUBSTR(STRING,INT,INT) <- STRING)/
   STRING(INDEX(STRING,STRING) <- INT)/
   STRING(LENGTH(STRING) <- INT)/
   STRING(EMPTY(STRING) <- BOOL)/
   STRING(STRING = STRING <- BOOL)/
EROPNS /
   STRING(BADINDEX1 <- INT)/
   STRING(BADINDEX2 <- INT)/
   STRING(BADPARAM1 <- INT)/
   STRING(BADPARAM2 <- INT)/
OKAXS /
   BOOL(EMPTY(NIL) = TRUE)/
   BOOL(EMPTY(ADD(*S,*C)) = FALSE)/
   INT(LENGTH(NIL) = 0)/
   INT(LENGTH(ADD(*S,*C)) = INT(STRING(LENGTH(*S))+1))/
   STRING(*S + NIL = *S)/
   STRING(*S + ADD(*U,*C) = ADD(*S+*U,*C))/
   STRING(SUBSTR(ADD(*S,*C),*I,*J) = ITE (INT(*J=0),NIL,
                ITE (INT(*J=STRING(LENGTH(*S))-*I+2),
                ADD(SUBSTR(*S,*I,*J-1),*C),
                SUBSTR(*S,*I,*J))))/
   INT(INDEX(*S,NIL) = INT(STRING(LENGTH(*S))+1))/
```



```
   INT(INDEX(NIL,ADD(*U,*C)) = 0)/
   INT(INDEX(ADD(*S,*C),ADD(*U,*D)) =
                ITE ('(INT(STRING(INDEX(*S,ADD(*U,*D)))=0),
                STRING(INDEX(*S,ADD(*U,*D))),
                ITE (CHAR(*C=*D) ¶
                STRING(*U=SUBSTR(*S,INT(STRING(LENGTH(*S))
                -STRING(LENGTH(*U))+1),STRING(LENGTH(*U)))),
                INT(STRING(LENGTH(*S))-STRING(LENGTH(*U)+1)), 0)))/
   BOOL(NIL = NIL = TRUE)/
   BOOL(ADD(*S,*C)=ADD(*U,*C) = STRING(*S=*U))/
   BOOL(ADD(*S,*C1)=ADD(*U,*C2) = FALSE)/
ERAXS /
   INT(SUBSTR(*S,*I,*J) =
      IT (INT(*I>STRING(LENGTH(*S))),BADPARAM1))/
   INT(SUBSTR(*S,*I,*J) =
      IT (INT(*I+*J>STRING(LENGTH(*S))),BADPARAM2))/
   INT(INDEX(*S,*U) = IT ('(STRING(*S)),BADINDEX1))/
   INT(INDEX(*S,*U) = IT ('(STRING(*U)),BADINDEX2))/
```

II.2.5. SYSTEMes TUPLE.

Les TUPLEs sont des produits cartésiens labellés. Nous utilisons la notation

TUPLE $(\langle s_1:S_1;\ldots;s_i:S_i;\ldots;s_n:S_n\rangle)$ pour désigner l'ensemble des n-uplets tels que pour tout i compris entre 1 et n, la ième composante de sélecteur s_i est un élément de sorte S_i. Les éléments d'un tuple

T= TUPLE (<s_1:S_1;..;s_n:S_n>) sont notés (x_1;...;x_n) avec x_i dans S_i pour tout i compris entre 1 et n. L'opération de projection sur la ième composante est notée proj(s_i). Si x est un n-uplet de sorte T alors la ième projection de x: proj(s_i)(x) sera notée T(s_i(x)), pour tout i compris entre 1 et n. Deux n-uplets (x_1;...;x_n) et (y_1;...;y_n) de type TUPLE (<s_1:S_1;...;s_n:S_n>) sont égaux si et seulement si x_i=y_i pour tout i compris entre 1 et n. Il faut, pour cela, que toutes les sortes S_i qui figurent dans le TUPLE possèdent une opération d'égalité.

SYSTEM (TUPLE (<s_1:S_1;...;s_n:S_n>))/

SORTS (T)/ IMPORTS (S_1;...;S_n;BOOL;INT)/

BUILD /

T((S_1;...;S_n) <- T)/

OKOPNS /

T(s_i(T) <- S_i)/ (pour tout i compris entre 1 et n)

T(T = T <- BOOL)/

T(SIZE(T) <- INT)/

EROPNS /

T($ERROR_i$ <- S_i)/ (pour tout i compris entre 1 et n)

OKAXS /

S_i(s_i(x_1;..;x_n) = x_i)/ (pour tout i compris entre 1 et n)

BOOL((x_1;..;x_n)=(y_1;..;y_n) =

ITE (S_1(x_1=y_1)^..^S_n(x_n=y_n),TRUE,FALSE))/

INT(SIZE(x_1;..;x_n) = ITE (INT(0<n)),n,0)/

ERAXS /

S_i(s_i(x_1;..;x_n) = IT (S_i(x_i)=TRUE,x_i,$ERROR_i$))/ (pour tout i compris entre 1 et n)

END (TUPLE (<s_1:S_1;..;s_n:S_n>))/

Remarque : Pour le moment, nous n'avons pas défini de systèmes TUPLEs récursifs; c'est-à-dire des SYSTEMes de la forme suivante:

T= TUPLE (<s_1:S_1;..;s_i:T;..;s_n:S_n>)

où la sorte définie T a une occurrence (i) dans les sortes qui composent le tuple.

II.2.6. SYSTEMes SET.

On note SET (S) le SYSTEMe qui définit l'ensemble des parties finies de sorte S. Cette construction n'est valide que si la sorte S possède une opération d'égalité "=". L'égalité est en effet l'opération qui permet de décider si un élément x appartient ou n'appartient pas à un ensemble X de SET (S). Ce SYSTEMe comporte deux constructeurs: la constante ensemble vide notée "∅" et l'opération ajouter un élément dans un ensemble notée "ADD". Nous avons les opérations classiques sur les ensembles: appartenance d'un élément notée "IN", la réunion notée "+", l'intersection notée "^", la différence notée "-", l'égalité notée "=", l'inclusion INCL, le cardinal CARD et l'opération de test EMPTY. Les opérations tout "A!", existe "E!" et sélecter "S!" sont traitées comme des familles d'opérations paramétrées, où les paramètres sont les variables liées d'une lambda expression définissant un prédicat. Les opérations "A!" et "E!" peuvent en plus être imbriquées. Enfin, une opération YIELD délivre les éléments d'un ensemble un à un. Pour ce SYSTEMe nous ne donnerons que la signature; l'ensemble des axiomes qui caractérisent ces opérations peut se déduire, par exemple, des axiomes de ZERMELO-FRAENKEL (KRI72).

SYSTEM (SET(S))/

SORTS (SET)/ IMPORTS (S;BOOL;INT)/

BUILD /

SET(∅ <- SET)/

SET(ADD(SET,S) <- SET)/

**ADD est utilisée sous forme infixée avec la notation '#' **

OKOPNS /

SET(DEL(SET,S) <- SET)/

SET(S IN SET <- BOOL)/

SET(SET + SET <- SET)/

SET(SET * SET <- SET)/

SET(SET - SET <- SET)/

SET(EMPTY(SET) <- BOOL)/

SET(INCL(SET,SET) <- BOOL)/

SET(SET = SET <- BOOL)/

SET(CARD(SET) <- INT)/

SET(AIS IN SET : P(S) <- BOOL)/

SET(EIS IN SET : P(S) <- BOOL)/

SET(SIS IN SET : P(S) <- SET)/

SET(YIELD(S IN SET) <- S)/

OKAXS /

.

.

END (SET(S))/

Précisons que les ensembles de SET(S) sont notés $(x_1 \neq x_2 \neq \ldots \neq x_n \neq \emptyset)$ où x_i dans S pour tout i compris entre 1 et n, sinon (∅).

II.2.7. SYSTEMes SEQ.

On note SEQ (S) le SYSTEMe qui définit l'ensemble des séquences finies de sorte S . Ce SYSTEMe utilise deux constructeurs: la constante séquence vide notée "NIL" et l'opération ajouter un élément à droite dans une séquence notée "ADD". Les autres opérations sont: ajouter un élément à gauche "ADG", supprimer un élément à droite "DELD", supprimer un élément à gauche "DELG", enlever le premier élément d'une séquence qui vérifie une propriété donnée "DELS", enlever tous les éléments d'une séquence qui vérifient une propriété donnée "DELM", longueur d'une séquence "LENGTH", concaténation de deux séquences "+", égalité de deux séquences "=", premier élément d'une séquence "CAR" et queue de séquence "CDR", dernier élément d'une séquence "LAST", test si séquence vide "EMPTY", extraction d'une sous-séquence de longueur donnée à partir d'un index donné "SUBSTR", index d'une sous-séquence donnée d'une séquence "INDEX", inverse d'une séquence "INVERSE", test d'inclusion d'une séquence dans une autre "INCL", test si un élément appartient à une séquence donnée "IN". En outre, comme pour les SYSTEMes SET, nous avons des familles d'opérations paramétrées tout "A!", existe "E!" et sélecter "S!", où les paramètres snt les variables liées d'une lambda expression qui définit un prédicat. Nous avons aussi une extraction YIELD qui délivre les éléments un à un. Nous ne donnerons que la signature de ce SYSTEMe décrite ci-dessous.

SYSTEM (SEQ(S))/

SORTS (SEQ)/ IMPORTS (S;BOOL;INT)/

BUILD /

SEQ(NIL <- SEQ)/

```
    SEQ(ADD(SEQ,S) <- SEQ)/
OKOPNS /
    SEQ(ADG(SEQ,S) <- SEQ)/
    SEQ(DELG(SEQ) <- SEQ)/
    SEQ(DELD(SEQ) <- SEQ)/
    SEQ(DELS(S IN SEQ : Q(S)) <- SEQ)/
    SEQ(DELM(S IN SEQ : Q(S)) <- SEQ)/
    SEQ(LENGTH(SEQ) <- INT)/
    SEQ(SEQ + SEQ <- SEQ)/
    SEQ(SEQ = SEQ <- BOOL)/
    SEQ(CAR(SEQ) <- S)/
    SEQ(CDR(SEQ) <- SEQ)/
    SEQ(LAST(SEQ) <- S)/
    SEQ(EMPTY(SEQ) <- BOOL)/
    SEQ(SUBSTR(SEQ,INT,INT) <- SEQ)/
    SEQ(INDEX(SEQ,SEQ) <- INT)/
    SEQ(INVERSE(SEQ) <- SEQ)/
    SEQ(INCL(SEQ,SEQ) <- BOOL)/
    SEQ(AIS IN SEQ : P(S) <- BOOL)/
    SEQ(EIS IN SEQ : P(S) <- BOOL)/
    SEQ(SIS IN SEQ : P(S) <- SEQ)/
    SEQ(YIELD(S IN SEQ) <- SEQ)/
OKAXS /
    .
    .
END (SEQ(S))/
```

Les séquences finies de SEQ(S) sont notées $(x_n.....x_2.x_1.NIL)$ avec

x_i dans S pour tout i compris entre 1 et n, sinon NIL.

II.2.8. Remarques.

Remarque 1: Tous les SYSTEMes précédents spécifient des types abstraits avec erreurs : BOOL, CHAR, INT, STRING et des types abstraits paramétrés avec erreurs : TUPLE, SET, SEQ. Ils ont été implantés directement en PROLOG, conformément au principe de construction des algèbres avec erreurs énoncé au chapitre VII . Cette technique d'implantation comporte un mécanisme de contrôle de sortie des messages d'erreurs: "messages d'avertissement" qui signalent systématiquement les expressions erreurs et messages d'erreurs finaux, produits en fin d'évaluation des expressions erreurs. La production de ces différents messages fournit une aide particulièrement efficace lors de la mise au point des SYSTEMes.

Remarque 2: Pour contrôler que les spécifications algébriques construites sont fidèles aux intentions initiales des utilisateurs, nous proposons de fabriquer des maquettes de test. L'expérience nous a montré que si l'on introduisait une opération d'assignation dans chacun des systèmes de l'environnement initial, cela facilitait la création et la manipulation des maquettes de systèmes de bases de données symboliques. C'est la raison essentielle qui nous a amené à le faire effectivement. L'exemple de maquette de système de base de données décrit en Annexe illustre l'utilisation de ce genre d'opération.

II.3. Instructions de traitement de données symboliques.

II.3.1. Instruction RUN.

Pour évaluer une expression, VEGA fournit une instruction programme dont la syntaxe est:

RUN (<expression>)?

Exemple : RUN (PILINT(DEPILER(EMPILER(EMPILER(PVIDE,1),2))))?

Le résultat du traitement se présente alors sous la forme:

PILINT:EMPILER(PVIDE,1)

L'expression résultat que l'on obtient est systématiquement préfixée par sa sorte.

Pour évaluer une expression avec une instruction RUN, VEGA utilise les équations des SYSTEMes comme des règles de réécriture orientées de la gauche vers la droite (cf II.2.chapitre VII.). Ainsi le traitement d'un RUN consiste à réduire l'expression qui suit entre parenthèses, en une expression équivalente irréductible dite forme canonique (si elle existe) qui est alors l'unique représentant de l'ensemble des termes de la classe d'équivalence. (cf II.3.chapitre VII.)

Dans l'exemple précédent, l'expression obtenue après évaluation: EMPILER(PVIDE,1) est une forme canonique du SYSTEMe PILINT.

Le procédé d'évaluation d'une expression utilise une stratégie de réduction de l'intérieur vers l'extérieur. Pour réduire une expression (ou sous-expression), c'est-à-dire pouvoir lui appliquer une règle de réécriture, il faut que tous les arguments de ses opérations soient déjà réduits. La réduction consiste alors à appliquer des règles de

réécriture aux sous-expressions les plus internes d'abord, jusqu'à l'obtention d'une sous-expression irréductible. Précisons que le procédé de réduction commence par appliquer les règles d'erreurs avant les règles OK, conformément au principe énoncé en I.6.chapitre VII.. Pour illustrer ce procédé d'évaluation, nous allons considérer une expression erreur définie à partir du SYSTEMe PGCD:

PGCD(2+PGCD(-2,3))

L'évaluation de cette expression se décompose comme suit: (d'après les axiomes du SYSTEMe PGCD)

PGCD(2+PGCD(-2,3))

PGCD(2+PGCD(-2,3))

PGCD(2+ARGNEG)

INT:2+ARGNEG

Nous donnons en Annexe plusieurs exemples d'évaluation d'expressions PGCD.

II.3.2. Instruction MRUN.

VEGA fournit aussi des constructions de langage adaptées à la manipulation de maquettes de systèmes de bases de données symboliques. La syntaxe d'un programme est

MRUN (<instruction>[;<suite-d-instructions>]*) où les instructions peuvent être des RUN définis comme en II.3.1., des instructions d'impression: PRINI (<expression>), de saut de ligne LINE et surtout des instructions de très haut niveau EACH et FIRSI qui augmentent les possibilités d'expressions sur les ensembles, les séquences, les

tuples.

II.3.2.1. L'instruction EACH .

Le but de cette instruction est de délivrer un par un tous les objets d'une collection abstraite (ensemble, séquence, ou SYSTEMes ayant des ensembles, des séquences pour MODEL) qui satisfont une propriété. La syntaxe est:

EACH (<variable(s) contrôlée(s) de sorte s> IN <variable collection sur sorte s> { :(<expression logique>) })

Montrons sur un exemple que nous détaillons par ailleurs (annexe) comment s'utilise cette instruction. Soit EL un ensemble fini de livres de sorte LIVRES défini par un SYSTEMe LIVRES. Un livre de EL est défini par un TUPLE LIVRE comportant un numéro de livre, un nom d'auteur, un sujet de sorte STRING et un titre. Pour lister tous les livres de sujet INFO de la collection EL on écrit:

MRUN (EACH (*L IN LIVRES(EL): STRING(LIVRE(SUJET(*L))=INFO)))?

On peut aussi formuler cette requête de la façon suivante:

MRUN (EACH ((*NL;*NOM;*SUJ;*TIT) IN LIVRES(EL):STRING(*SUJ=INFO)))?

Ici, (*NL;*NOM;*SUJ;*TIT) est un terme variable de sorte LIVRE.

II.3.2.2. L'instruction FIRST.

Cette instruction délivre un (ou le premier) objet d'une collction abstraite qui satisfait une propriété. La syntaxe est:

FIRST (<variable(s) contrôlée(s) de sorte s> IN <variable collection sur sorte s> { :(<expression logique>) })

En reprenant l'exemple précédent, on peut lister, cette fois, un (ou le premier) livre de EL qui traite d'INFO en écrivant:

MRUN (FIRST (*L IN LIVRES(EL): STRING(LIVRE(SUJET(*L))=INFO)))?

II.3.2.3. Remarques.

Remarque 1: Les instructions EACH et FIRST peuvent être combinées avec des instructions RUN, LINE, PRINT. Des exemples sont décrits en Annexe.

Remarque 2: L'interprétation d'un programme MRUN utilise le mécanisme d'unification de PROLOG sur la suite des instructions. Les instructions RUN et les expressions logiques sont évaluées, quant à elles, en utilisant le même procédé d'unification mais pour réduire les expressions.

Remarque 3: Ces constructions de langage, orientées bases de données, ressemblent à celles que nous avons présentées en première partie de notre travail, notamment à PASCAL relationnel (SCH77).

III. MECANISMES DE CONSTRUCTION DE SPECIFICATIONS.

III.1. Dérivation de sous-types abstraits avec erreurs.

La plupart des dérivations de sous-types abstraits se définissent à partir d'un modèle abstrait dont on a restreint l'ensemble des valeurs. En VEGA, cette restriction s'exprime sous la forme de clauses RESTRICTO et d' INVARIANTs abstraits.

III.1.1. Clause RESTRICTO.

Ce type de clause vise à restreindre l'ensemble des valeurs d'une sorte (modèle abstrait élémentaire ou paramétré) à une liste de valeurs énumérées, ou à un intervalle de valeurs (si la sorte possède une opération "<"), ou à un ensemble de valeurs satisfaisant une propriété. La syntaxe est la suivante:

RESTRICTO (<variable de sorte s>: <constante de sorte s>[.<constante de sorte s>]*)

ou RESTRICTO (<variable de sorte s>: <constante de sorte s> IO <constante de sorte s>)

Cette clause peut s'appliquer à un modèle abstrait élémentaire (CHAR, BOOL, STRING, INT), à l'argument de l'instance de modèle abstrait paramétré, à l'instance elle-même ou les deux à la fois.

. Cas d'un modèle abstrait élémentaire .

Exemple 1: SYSTEM (ENTENUM)/ SORTS (ENTENUM)/

MODEL (INT RESTRICTO (*I:1.5.15.20))/

Ce système définit ce qu'on a coutume d'appeler en programmation un type énuméré.

Exemple 2: SYSTEM (ENTINT)/ SORTS (ENTINT)/

MODEL (INT RESTRICTO (*I:10 IO 50))/

Le système INT possède une opération inférieur (<).

. Cas d'un modèle abstrait paramétré .

Nous allons montrer comment s'utilise cette clause sur les exemples simples suivants:

Exemple 3: SYSTEM (ENS1)/ SORTS (ENS1)/

MODEL (SET(INT RESTRICTO (*I:1.5.15.20)))/

Exemple 4: SYSTEM (ENS2)/ SORTS (ENS2)/

MODEL (SET(INT) RESTRICTO (*I:(2.5).(3.4.6)))/

Exemple 5: SYSTEM (ENS3)/ SORTS (ENS3)/

MODEL (SET(INT RESTRICTO (*I:5 IO 20)), RESTRICTO (*J:(12.15).(13.14.16)))/

Notons que la clause RESTRICTO ne s'applique pas aux sortes à l'intérieur d'un TUPLE. Dans ce cas, il faut définir un SYSTEMe pour chacune des sortes qui doit être restreinte.

III.1.2. Clause INVARIANT.

Un INVARIANT abstrait caractérise l'ensemble des valeurs d'un modèle abstrait, sa syntaxe a la forme suivante:

INVAR (nom-d-invariant(<variable de sorte s>) <-

nom-de-sorte(<variable de sorte s>):<condition>,

<message-d-erreur>)

[↑ nom-de-sorte(<variable de sorte s>):<condition>,
<message-d-erreur>)]*)

Les conditions peuvent être des restrictions simples: valeurs énumérées, intervalle de valeurs ou encore des expressions logiques utilisant des quantificateurs universels et/ou existentiels. Il est important de noter qu'un message d'erreur est associé à chaque propriété de l'invariant. Les propriétés sont indépendantes les unes des autres de sorte que l'évaluation d'un INVARIANT déclenche autant de messages d'erreurs qu'il y a de conditions non vérifiées. Ceci constitue une aide supplémentaire à la mise au point des SYSTEMes mais aussi des maquettes de systèmes de bases de données symboliques. Nous allons montrer sur un exemple comment se déclare un invariant:

Exemple 6:

SYSTEM (ENS4)/ SORTS (ENS4)/ MODEL (SET(INT) INVARIANT IENS4)/

INVAR (IENS4(*E)<-ENS4(*E : INT(ENS4(CARD(*E))<50),ERRLONG))/

La maquette de système de base de données que nous décrivons dans la suite fournit d'autres exemples.

III.1.3. Remarques.

Dans la version actuelle du système VEGA, la technique de spécification axiomatique des SYSTEMes ne supporte pas les mécanismes de restriction précédents. Il est évidemment possible de dériver un sous-type d'un type abstrait spécifié axiomatiquement en utilisant ce dernier comme modèle abstrait d'un nouveau SYSTEMe. On peut alors lui appliquer les restrictions RESTRICTO et INVAR décrits dans les

paragraphes précédents.

III.2. Extensions de spécifications avec coercition et surcharge des opérateurs.

Nous avons vu dans la première partie de notre travail que la plupart des langages de programmation modernes ADA, CLU, etc.. supportaient le mécanisme de surcharge des opérateurs. L'opérateur "+" qui figure dans presque tous les SYSTEMes de l'environnement initial de spécification: BOOL, INT, STRING, SET, SEQ est un exemple d'opérateur surchargé. Les SYSTEMes définis par les utilisateurs supportent aussi ce mécanisme. Par contre la coercition est un mécanisme qui, à notre connaissance, n'a été introduit que dans deux langages ALGOL 68 (ALG75) et OBJ-0 (GOG79). La coercition consiste à "forcer" une sorte s_1 à être une sous-sorte d'une autre sorte s_2. En VEGA, cela se traduit par une déclaration d'OKOPNS:

Exemple : $s_2(s_1)<-s_2$ où s_1 est une sous-sorte de la sorte s_2.

La coercition s'accompagne bien sûr de la surcharge des opérateurs de la sorte s_1. Il faudra donc prendre quelques précautions lorsqu'on définira de nouvelles opérations sur la sorte s_2. Un SYSTEMe $P_1=(\mathcal{F}_1,\Sigma_1,E_1)$ est une extension d'un SYSTEMe $P_2=(\mathcal{F}_2,\Sigma_2^{BUILD},\Sigma_2^{OK},\Sigma_2^{ER},E_2^{OK},E_2^{ER})$ si $\mathcal{F}_1,\Sigma_1^{BUILD},\Sigma_1^{OK},\Sigma_1^{ER},E_1^{OK},E_1^{ER}$ sont respectivement inclus dans $\mathcal{F}_2,\Sigma_2^{BUILD},\Sigma_2^{OK},\Sigma_2^{ER},F_2^{OK},F_2^{ER}$

Un enrichissement de SYSTEMe est une extension ayant même ensemble de sortes (cf I.5.chapitre VII.). Montrons maintenant sur un exemple de quelle façon la coercition et la surcharge des opérateurs s'utilisent

pour définir une extension d'un SYSTEMe.

Nous présentons un exemple qui a fait l'objet de vives controverses (MAJ77) (VEL79). Il s'agit de la fameuse pile avec lecture interne. Rappelons brièvement le problème. On se propose de spécifier algébriquement une telle pile comme une extension d'une pile classique, dont les opérations sont: pilevide (PVIDE), empiler (EMP), dépiler (DEP), estvide (VIDE). L'extension PLI qui spécifie la pile avec lecture interne comporte une opération pour lire à l'intérieur d'une pile (VAL), à une hauteur précisée par un index (INDSOM). L'index peut descendre (DESC) ou remonter (REM). On dispose aussi d'une opération (DECR).

Le lecteur attentif remarquera que nous avons introduit une opération auxiliaire DESC dans PILC. On aurait aussi pu la définir dans PLI, mais comme nous n'avons pas implanté de mécanisme d"opérations cachées", nous l'avons placée dans PILC. L'extension PLI est consistante et suffisamment complète vis à vis de PILC (REM82).

III.3. Restriction des spécifications avec renommage des opérations.

ce mécanisme découle de la technique de spécification sur modèle abstrait. En effet, toutes les opérations du SYSTEMe qui est défini sont des opérations du modèle abstrait, à un renommage près. D'ailleurs, si nous avons surspécifié les SYSTEMes paramétrés de l'environnement initial de VEGA, c'est précisément dans le but d'obtenir, par restriction, d'autres structures de données classiques. Par exemple, les SYSTEMes LIST sont définis comme des restrictions du SYSTEMe SEQuence.

Exemple :

```
SYSTEM (LIST)/
   SORTS (LIST)/ MODEL (SEQ(INT))/
OKOPNS /
   LIST(NIL <- LIST)/
   LIST(CONS(LIST,INT) <- LIST)/
   LIST(CAR(LIST) <- INT)/
   LIST(CDR(LIST) <- LIST)/
   LIST(INVERSE(LIST) <- LIST)/
   LIST(EMPTY(LIST) <- BOOL)/
EROPNS /
   LIST(CARERR <- INT)/
   LIST(CDRERR <- LIST)/
AFUNC /
   LIST(REP(NIL) = NIL)/
   LIST(REP(*E.*F) = CONS(REP(*F),*E))/
OKSEMS /
```

```
    LIST(NIL = REP(NIL))/
    LIST(CONS(REP(*L),*I) = REP(ADG(*L,*I)))/
    INT(CAR(REP(*L)) = CAR(*L))/
    LIST(CDR(REP(*L)) = REP(CDR(*L)))/
    LIST(INVERSE(REP(*L)) = REP(INVERSE(*L)))/
    BOOL(EMPTY(REP(*L)) = EMPTY(*L))/
ERSEMS /
    INT(CAR(NIL) = CARERR)/
    LIST(CDR(NIL) = CDRERR)/
END (LIST)/
```

On aurait aussi pu spécifier un sous-type abstrait de LISTe d'entiers, en restreignant la sorte paramètre INT (RESTRICTO) et/ou la sorte SEQ(INT) elle-même (INVARIANT). Nous donnons en Annexe des exemples d'évaluation d'expressions LIST.

III.4. Combinaison de spécifications.

Il arrive parfois que l'on veuille spécifier plusieurs SYSTEMes en même temps, soit parce qu'il existe des relations (opérateurs) entre deux ou plusieurs sortes, soit encore, parce que l'utilisateur désire, tout simplement, les "encapsuler" dans une seule unité de spécification. La spécification d'une combinaison exige seulement que l'on regroupe les axiomes se rapportant à un sous-SYSTEMe. Pour illustrer ce mécanisme, nous présentons une combinaison (MPITA) de PILE et de TABLE d'entiers. La pile ne comporte ici que deux opérations

constructeurs (PVIDE et EMP). La table d'entiers TAB utilise deux constructeurs: la constante table vide (TVIDE) et l'opération ranger une valeur à une position donnée (RANGER) et des opérations de lecture d'une valeur (LIRE), de suppression (ENLEVER) et de test (EXISTE).

```
SYSTEM (MPITA)/
    SORTS (TAB;PILC)/ IMPORTS (INT;BOOL)/
BUILD /
    PILC(PVIDE <- PILC)/
    PILC(EMP(PILC,INT) <- PILC)/
    TAB(TVIDE <- TAB)/
    TAB(RANGER(TAB,INT,INT) <- TAB)/
OKOPNS /
    TAB(LIRE(TAB,INT) <- INT)/
    TAB(ENLEVER(TAB,INT) <- TAB)/
    TAB(EXISTE(TAB,INT) <- BOOL)/
EROPNS /
    TAB(VALERR <- INT)/
OKAXS (TAB)/
    INT(LIRE(RANGER(*T,*CL1,*V),*CL2)= ITE (INT(*CL1=*CL2),*V,
                    TAB(LIRE(*T,*CL2))))/
    TAB(ENLEVER(TVIDE,*CL1)=TVIDE)/
    TAB(ENLEVER(RANGER(*T,*CL1,*V),*CL2)=
                    ITE (INT(*CL1=*CL2),ENLEVER(*T,*CL2),
                    RANGER(ENLEVER(*T,*CL2),*CL1,*V)))/
    BOOL(EXISTE(TVIDE,*CL1)=FALSE)/
    BOOL(EXISTE(RANGER(*T,*CL1,*V),*CL2)= ITE (INT(*CL1=*CL2),TRUE,
                    TAB(EXISTE(*T,*CL2))))/
```

```
ERAXS (TAB)/
   INT(LIRE(TVIDE,*I)=VALERR)/
END (MPITA)/
```

Exemples d'évaluation:

```
RUN (TAB(LIRE(RANGER(RANGER(TVIDE,1,10),2,20),2)))?
                    INT:20
RUN (TAB(ENLEVER(RANGER(RANGER(RANGER(TVIDE,1,10), 2,20),3,30),2)))?
                    TAB:RANGER(RANGER(TVIDE,1,10),3,30)
RUN (TAB(EXISTE(RANGER(RANGER(TVIDE,1,10),2,20),3)))?
                    BOOL:FALSE
```

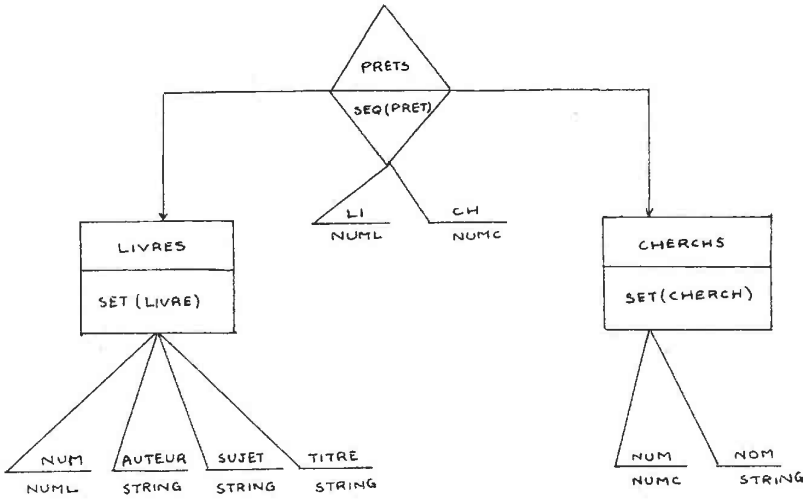
III.5. Un exemple de spécification algébrique de système de base de données, de construction et de manipulation d'une maquette de données symboliques.

L'application de gestion de prêts de livres à des chercheurs que nous présentons maintenant, a simplement pour but de montrer la façon de spécifier, puis de construire et de manipuler une maquette de système de base de données symboliques. Dans la troisième partie de notre travail, cette application (plus détaillée) a été programmée en TYP.

Le schéma ci-dessous décrit relativement bien l'univers du problème. On a deux entités livres (LIVRES) et chercheurs (CHERCHS) et une association prêts de livres à des chercheurs (PRETS).

où

représente une opération de sélection sur un objet LIVRE de



. l'entité LIVRES. Ce sélecteur de nom NUM opère sur LIVRE et délivre un résultat de sorte NUML.

III.5.1. Spécification de l'application.

Pour spécifier cette application, nous allons procéder par raffinements successifs. On commence donc par spécifier les entités LIVRES, CHERCHS et l'association PRETS de la façon suivante:

```
SYSTEM (LIVRES)/
   SORTS (LIVRES)/ MODEL (SET(LIVRE))/
END (LIVRES)/
SYSTEM (CHERCHS)/
   SORTS (CHERCHS)/ MODEL (SET(CHERCH))/
END (CHERCHS)/
SYSTEM (PRETS)/
   SORTS (PRETS)/ MODEL (SEQ(PRET))/
END (PRETS)/
```

Cette spécification introduit trois nouveaux SYSTEMes LIVRE, CHERCH et PRET. Ce sont des produits cartésiens labellés spécifiés par:

```
SYSTEM (LIVRE)/
   SORTS (LIVRE)/
   MODEL ( TUPLE (<NUM:NUML;AUTEUR:STRING;SUJET:STRING;
      TITRE:STRING>))/
END (LIVRE)/
SYSTEM (CHERCH)/
   SORTS (CHERCH)/
```

```
    MODEL ( TUPLE (<NUM:NUMC;NOM:STRING>))/
  END (CHERCH)/
```

Le SYSTEMe PRET doit être spécifié de façon à satisfaire la contrainte:
"un livre ne peut être prêté que s'il n'a pas déjà été emprunté par un chercheur".

```
  SYSTEM (PRET)/
    SORTS (PRET)/
    MODEL ( TUPLE (<LI:NUML;CH:NUMC>) INVARIANT IPRET)/
    INVAR (IPRET(*NL;*NC) <-PRET((*NL;*NC):
                PRETS(A!(*NL;*N) IN EP: (NUMC(*N=*NC))),ERRPRET))/
  END (PRET)/
```

Cette spécification introduit des sortes NUML (numéro de livre) et NUMC (numéro de chercheur), une sorte de base STRING, et une variable EP que l'on précisera après avoir défini les nouvelles sortes:

```
  SYSTEM (NUML)/
    SORTS (NUML)/
    MODEL (INT RESTRICTO (*I:0 IO 60))/
  END (NUML)/
  SYSTEM (NUMC)/
    SORTS (NUMC)/
    MODEL (INT RESTRICTO (*I:50 IO 100))/
  END (NUMC)/
```

La spécification algébrique terminée et testée, l'utilisateur doit pouvoir facilement s'assurer que le résultat obtenu est fidèle à l'énoncé initial du problème, ou encore répond à tous les besoins initialement exprimés. VEGA fournit quelques facilités pour élaborer et

ensuite pour tester des maquettes sur des jeux de données symboliques.

III.5.2. Elaboration d'une maquette de système de base de données.

Il suffit de construire des objets EL, EC et EP de sortes respectives LIVRES, CHERCHS et PRETS. Ce sont des objets dynamiques (variables) fortement typés qui permettent de conserver des valeurs symboliques typées.

Illustrons la construction de ces objets à partir du cas de EP (séquence de prets). On commence par déclarer une variable EP de sorte PRETS, initialisée à vide "NIL"

```
  RUN (PRETS(EP<-NIL))?
```

Ensuite, on effectue une suite de modifications de EP de la façon suivante:

```
  RUN (PRETS(EP<-ADD(EP,(1;52))))?
  RUN (PRETS(EP<-ADD(EP,(7;56))))?
```

La modification de EP entraîne un certain nombre de "contrôles de types". Un premier contrôle est fait sur la valeur que l'on cherche à insérer dans EP. Cette valeur doit être de sorte PRET sinon VEGA déclenche un message d'erreur. En cas de succès, cette valeur ne peut être insérée dans EP que si la contrainte spécifiée par l'invariant abstrait IPRET de PRET est satisfaite. Un échec va produire un message d'erreur spécifique, ici: >>ERROR>>ERRPRET.

On procède de la même façon pour EL et EC (cf listing).

Une fois la maquette construite, l'utilisateur va pouvoir faire des manipulations en utilisant le langage d'interrogation VEGA.

III.5.3. Manipulation de la maquette.

Cette manipulation s'effectue pratiquement toujours avec des MRUN. Nous allons décrire quelques exemples qui figurent sur le listing donné en Annexe.

Requête 1: "Lister les prêts" s'écrit:

```
MRUN ( EACH (*P IN PRETS(EP));
       PRINT (PRET(*P));
       LINE )?
```

et produit le résultat suivant:

```
PRET:1;52
PRET:7;56
```

Requête 2: "Lister les sujets des livres empruntés" s'écrit:

```
MRUN ( EACH (*NL;*NC) IN PRETS(EP));
       FIRST ((*NL;*A;*S;*T) IN LIVRES(EL));
       PRINT (*S);
       LINE )?
```

et produit le résultat suivant:

```
INFO
INFO
```

Requête 3: "Lister chaque chercheur avec les numéros des livres qu'il a empruntés"

```
MRUN ( EACH ((*NC;*NO) IN CHERCHS(EC));
       PRINT (*NO); LINE ;
       EACH ((*NL;*NC) IN PRETS(EP));
       PRINT (*NL);
       LINE )?
```


produit les résultats suivants:

```
JEAN
1
2
5
CLAUDE
.
```

Notons que l'utilisateur a toujours la possibilité de modifier les entités EL, EC et EP entre deux interrogations. Il peut aussi construire d'autres jeux de données (EL1, EC1, EP1),... pour tester des aspects particuliers de la spécification de l'application.

L'exemple d'application que nous venons de présenter est élémentaire. Nous pouvons tout de même préciser que VEGA s'est avéré un outil tout à fait approprié pour construire la maquette de système expert du projet MONT-LOZERE (ADI82). En bref, VEGA semble assez bien répondre aux objectifs que nous nous étions fixés.

IV. IMPLANTATION DE VEGA.

Le système VEGA a été développé en PROLOG (ROU75) sur l'IRIS 80 V10 de l'Institut Universitaire de Calcul Automatique de Lorraine (IUCAL). Il comporte un compilateur de SYSTEMes, un interpréteur (évaluateur) d'expressions (RUN), un interpréteur du langage de manipulation de données symboliques (MRUN), un bibliothécaire et l'éditeur de textes SIRIS 8.

IV.1. Le compilateur.

Le compilateur se compose de plusieurs modules. Citons:

-- l'analyseur non déterministe des SYSTEMes qui se décompose comme suit:

.l'analyseur d'expressions de l'entête et de la fin des SYSTEMes,

.l'analyseur d'expressions du paragraphe BUILD,

.l'analyseur d'expressions du paragraphe OKOPNS,

.l'analyseur d'expressions du paragraphe EROPNS,

.l'analyseur d'expressions du paragraphe AFUNC,

.l'analyseur d'expressions INVAR,

.l'analyseur d'expressions du paragraphe OKAXS,

.l'analyseur d'expressions du paragraphe ERAXS,

.l'analyseur d'expressions du paragraphe OKSEMS,

.l'analyseur d'expressions du paragraphe ERSEMS,

-- le module de génération PROLOG qui se décompose aussi en plusieurs autres modules de génération: entête et fin de SYSTEMes, BUILD, ..., ERSEMS.

Un exposé, même sommaire, de cette réalisation nous obligerait à entrer dans beaucoup trop de détails de programmation en PROLOG. Le lecteur intéressé pourra toutefois trouver des informations dans (CHA82). Précisons que l'implantation des SYSTEMes de l'environnement initial de spécification est organisée de façon à permettre le partage des textes pour chaque instance de SYSTEMes paramétrés, mais aussi pour chaque variable de SYSTEMe standard ou utilisateur.

IV.2. L'interpréteur RUN.

L'interpréteur (ou évaluateur) d'expressions est initialisé lorsque le mot-clé RUN est rencontré. L'évaluation de l'expression, entre parenthèses, commence lorsque le système rencontre le caractère "?". Si une expression est incorrecte, pour l'instant, l'interpréteur se contente de lister l'expression (sans message).

IV.3. L'interpréteur MRUN.

Comme précédemment, l'interpréteur du langage de manipulation est initialisé lorsque le mot-clé MRUN est rencontré. L'interprétation de l'expression, entre parenthèses, débute lorsque le système rencontre le caractère "?". Si une "instruction" du programme est incorrecte, l'interpréteur s'arrête sur l'instruction et l'imprime, mais sans message d'erreur, pour l'instant.

IV.4. Le bibliothécaire.

Tous les modules objets résultant de la compilation des SYSTEMes sont sauvegardés dans un fichier partitionné géré par PROLOG. Cette sauvegarde est à la charge de l'utilisateur. La compilation d'un SYSTEMe de nom X produit un module objet que l'utilisateur peut sauvegarder, à l'aide de la directive SAVE / de VEGA, avec un nom qu'il devra préciser avant chaque compilation.

IV.5. L'éditeur de textes.

L'utilisateur a la possibilité, s'il le désire, de mettre au point ses textes sources à l'aide de l'éditeur de textes sous SIRIS 8. Une directive INPUT / de VEGA lance la lecture du fichier qui a été créé sous l'éditeur de textes et effectue ensuite l'enchainement des traitements: compilation, interprétation RUN et/ou MRUN. Pour obtenir un listing du fichier source, il suffit de placer une directive LIST / en début du fichier source.

IV.6. L'environnement du système VEGA.

L'environnement d'un utilisateur est construit de façon incrémentale par rapport au système de base VEGA, en utilisant la directive SAVE /. VEGA peut supporter plusieurs environnements utilisateurs. L'initialisation d'un environnement est immédiate; il suffit que l'utilisateur se connecte au système de nom VEGA , de la façon suivante:

```
!VEGA
            /*procédure cataloguée SIRIS 8*/
SYSTEME? VEGA
    /*système VEGA*/
SOURCE? ESSAI
    /*nom du fichier source créé sous l'éditeur de textes*/
SAUVE? MODULE
    /*nom du fichier sauvegardé*/
```

Des exemples de sessions VEGA sont décrites en Annexe.

TROISIEME PARTIE

PRESENTATION DU SYSTEME TYP

DE PROGRAMMATION MODULAIRE

CHAPITRE X. CONCEPTS DE MODULARITE ET D'ABSTRACTION DU SYSTEME TYP.

I. DESCRIPTION MODULAIRE DES TRAITEMENTS: MODULES ET MACHINES.

II. DESCRIPTION DES TYPES DE DONNEES: TYPES ET CAPSULES.

III. CONSTRUCTION MODULAIRE DES BASES DE DONNEES.

III.1. Les machines et les systèmes de base de données.

III.2. La conservation des données permanentes dans le système.

III.3. Gestion des systèmes de bases de données.

III.4. La maintenance des systèmes de bases de données.

IV. DESCRIPTION MODULAIRE DES CONTRAINTES D'INTEGRITE ET DE SECURITE.

V. LA MANIPULATION DES DONNEES.

V.1. Pour les non-programmeurs.

V.2. Pour les programmeurs "terminaux".

V.3. Pour les programmeurs "experts".

V.4. Pour les programmeurs "système".

VI. META-LANGAGE: DEFINITION D'UNITES GENERIQUES.

VI.1. La notion d'unité générique.

VI.2. Ecriture des unités génériques.

VI.3. Utilisation des unités génériques.

INTRODUCTION

Pour démarrer le projet TYP, nous partions, comme nous avons déjà eu l'occasion de le dire, d'un ensemble de considérations générales sur les problèmes posés par le développement des applications informatiques nécessitant une gestion de données plus ou moins sophistiquée (CHA76).

Une de nos premières préoccupations fut alors de déterminer une conduite "réaliste" du projet. Dans notre esprit, cela signifiait: essayer de dégager progressivement les problèmes de développement posés, les caractériser le mieux possible pour pouvoir ensuite y apporter des solutions satisfaisantes; cette recherche de solutions impliquant naturellement des investigations sur les propositions plus ou moins récentes de bases de données et de langages de programmation.

Dans la première partie consacrée à la problèmatique de la construction des applications, nous avons montré qu'en pratique, actuellement, la seule façon de concevoir une gestion élaborée des données de programmes COBOL, PL1, était d'utiliser des interfaces de S.G.B.D. ou alors une approche de type CODASYL présentée au chapitre II. Nous avons montré que cela posait des problèmes de conception, de construction et de maintenance. Pour réduire ce genre de difficultés, de récents travaux proposent d'insérer dans des langages de programmation généraux ou spécifiques les concepts des bases de données relationnelles (constructeurs de tuples, de relations et constructions de langage pour la manipulation des données, etc..). Mais le problème

de la conception et de la réalisation de "grosses" applications informatique demeure. D'où l'idée naturelle et bien connue de décomposer (PAR71) l'application (ou une partie) en plusieurs parties plus faciles à maîtriser et d'appliquer des raffinements successifs (WIR71). On est alors ramené à deux problèmes:

-- décrire la conception et la réalisation de ces parties,

-- ensuite assembler ces différentes parties pour fabriquer des programmes de manipulation exécutables.

Les qualités du modèle de données relationnel, que nous avons rappelées au chapitre II., sont maintenant bien connues. On peut toutefois se demander si les tuples et les relations de ce modèle sont des notions suffisantes pour représenter tous les concepts rencontrés dans les applications bases de données. D'ailleurs, dans la communauté scientifique des bases de données, ne se pose-t-on pas le même genre de questions à propos des possibilités offertes par les différents modèles de données: hiérarchique, réseau et relationnel?

Dans le projet TYP, nous estimons qu'au lieu de privilégier ou même d'imposer à priori des choix de structures de données, il est préférable de fournir des outils appropriés pour construire, réaliser et manipuler celles qui paraissent les plus naturelles. Nous pensons que cette idée importante devrait ouvrir de nouveaux horizons pour la conception, la réalisation et l'utilisation des bases de données.

Nous avons vu que pour développer des applications bases de données, traditionnellement, on commence par concevoir et réaliser globalement une base de données, en essayant de prendre en compte tous les besoins potentiels des futurs utilisateurs. Ensuite, pour chaque application particulière, on conçoit et on réalise une vue externe de la base de données globale; les programmes de manipulation étant écrits à partir des schémas externes.

Cette approche nous a semblé poser deux problèmes importants:

— la conception globale des bases de données constitue une tâche d'une grande complexité,

— les programmeurs d'application doivent attendre que la base de données globale soit complètement construite pour commencer à développer les applications particulières, ce qui contribue à un allongement conséquent des délais de production du système logiciel. Pour réduire ces difficultés, nous proposons des concepts, des mécanismes et des outils qui permettent d'entreprendre une conception progressive de la base de données (par décomposition et raffinements successifs); la construction des programmes d'application s'effectuant en parallèle. Les "gros programmes" d'application sont bien sûr conçus et réalisés selon les mêmes principes généraux.

En programmation, l'idée de décomposer un programme en petites unités indépendantes appelées procédures ou modules n'est pas nouvelle (NAT79) (DIJ71) (WIR71) (PAR71). Le projet de recherche CIVA (DER74) auquel nous avons participé (CHA73) développait des idées qui allaient aussi dans ce sens. Toutefois, comme nous l'avons rappelé au chapitre

I., le concept de modularité a beaucoup évolué au cours de ces dernières années. De notre point de vue, il est nécessaire de définir une notion de module qui permette de décrire les traitements aussi bien que les systèmes de données (bases de données). Le principe des modules multi-procédures de PARNAS apporte sans aucun doute des éléments de réponse. Toutefois, ce genre de description modulaire exige que l'on précise beaucoup de détails: structures de données et algorithmes définissant les procédures. Pour simplifier la conception, faciliter l'utilisation, mais aussi pouvoir décrire des choix d'implantation qui soient relativement indépendants des programmes d'utilisation, nous avons défini une unité de spécification abstraite d'un module à la PARNAS, complètement séparée de ce dernier. Cette unité de spécification abstraite comporte essentiellement des déclarations de profils d'opérations: nature, nom, types des paramètres données et résultats. La spécificité des bases de données (utilisation d'objets construits, de types de structures de données d'objets construits) nous a aussi amené à définir un mécanisme de construction de types approprié. Nous avons retenu une notion de module voisine de celle de CLU. Il nous a cependant paru important de pouvoir séparer les spécifications abstraites des types des descriptions de choix d'implantation.

On a pu entrevoir précédemment que la construction d'une base de données comportait un ensemble de spécifications abstraites et de modules d'implantation. Un programme d'application peut être un "petit" programme de manipulation de base de données ou encore un "gros programme". Dans ce dernier cas, celui-ci sera constitué d'un ensemble

de spécifications abstraites et de modules d'implantation.

Une application base de données se définit comme un ensemble d'unités conçues et réalisées pour un domaine d'application relevant de l'intérêt commun de plusieurs utilisateurs. Dans cette approche, la notion de programme d'application autonome, telle qu'on la trouve dans les récentes propositions de langages de manipulation insérés dans des langages de programmation standard, disparaît au profit de celle de "chaîne de programmes" construite à partir des différentes unités de l'application. Une même unité peut appartenir à plusieurs programmes d'application. Dans notre projet, la mise en oeuvre de cette idée a nécessité le développement d'un outil spécifique de fabrication de programmes.

Nos propositions ici visent à réduire les difficultés que l'on rencontre pour construire des applications bases de données. Les mécanismes d'abstraction et de modularité supportés par le langage doivent favoriser une construction progressive des bases de données, par décomposition et raffinements successifs.

Lorsque nous parlons d'applications bases de données, nous pensons aux diverses applications informatiques des entreprises qui ont été développées pour automatiser des traitements d'information que nous qualifierons par le terme de "centres d'intérêts": ventes, stocks, production, etc... Chacune de ces applications devient pour les différentes catégories d'utilisateurs qui sont amenées à les utiliser, un véritable centre d'intérêt commun. Pour toutes les raisons que l'on

imagine facilement: sécurité et intégrité des données, etc..., il importe de pouvoir construire pour chaque classe d'utilisateurs, des contextes d'utilisation appropriés. Cette caractéristique du projet TYP relève aussi des concepts de modularité et d'abstraction. Elle permet d'unifier les notions de vues et de copie instantanée ("snapshot") des bases de données relationnelles avec celles de contextes de compilation et d'exécution des systèmes de programmation.

Un dernier point important concerne les outils logiciels utilisés pour la conception, la construction, l'utilisation, la documentation et la maintenance des applications bases de données. Au chapitre IV., nous avons donné un aperçu des difficultés posées par les logiciels de base actuels. Nos propositions abordent ici quelques idées du génie logiciel qui commencent seulement à émerger: système de compilation séparée, généricité, etc... Il s'agit de fournir un environnement de programmation modulaire qui intègre de façon cohérente tous les outils nécessaires: système de compilation séparée du langage noyau, système de compilation séparée des unités paramétrées (génériques), gestionnaires, documentalistes, système de connexion et éditeur de textes. Pour faciliter l'utilisation interactive de tous ces outils logiciels, nous avons aussi conçu un interface intégré au système qui permette d'assister l'utilisateur à chaque étape de son travail.

Au chapitre X., nous présentons les concepts de modules et d'abstraction qui permettent de développer, d'utiliser et de maintenir des applications bases de données. Le chapitre XI présente la notion d'application base de données. En XII, nous exposons rapidement le principe du moniteur d'interaction du système TYP. Dans le chapitre

XIII, nous discutons brièvement de la réalisation du prototype TYP.

La plupart des idées exposées dans cette troisième partie sont le fruit d'un travail d'équipe qui a déjà donné lieu à plusieurs rapports ou publications (CHA76) (MIN79) (PRO79) (TOU79) (HEN80) (CHA82) (CHA82) (TOU82).

Pour permettre la construction, la gestion, l'utilisation et la maintenance des applications bases de données, nous proposons un langage de programmation modulaire "orienté système" constitué de plusieurs niveaux de langages, structurés à la fois verticalement (CHE74) et horizontalement (DIJ78) (WIR71). Le choix d'un langage structuré multicibles est fondamental. Il pourrait, à priori, sembler en contradiction avec le but indiqué dans l'introduction:"... une formulation unique ...", nous allons voir qu'il n'en est rien.

Nous avons conçu un langage qui fournit des moyens d'expressions adaptés à chaque fonction d'utilisation du système: conception, construction, etc.., et en même temps qui assure une continuité d'expression au travers des diverses formulations utilisées. Notre choix n'est donc que la concrétisation au niveau du langage de programmation modulaire de cette recherche d'intégration cohérente des différentes possibilités de formulation. Bien entendu, selon les objectifs de réalisation que l'on vise: transportabilité du système, efficacité, etc.., plusieurs conceptions de langages sont envisageables (NEB74) (BRI76) (CHI78) (DIJ78). Nous reparlerons de cet aspect dans la suite.

Une partie importante de notre travail a été ici de dégager des concepts, des mécanismes de modularité et d'abstraction qui permettent d'unifier les concepts des bases de données avec ceux des langages de programmation avancée.

Dans les paragraphes qui suivent, nous précisons les concepts de modularité et d'abstraction du langage de programmtion appelé ATM.

I. DESCRIPTION MODULAIRE DES TRAITEMENTS: MODULES ET MACHINES.

Dans ce paragraphe, nous ne nous intéressons qu'à la définition des traitements, dans le sens où ce terme est employé en programmation. En programmation, l'expérience acquise avec la réalisation de systèmes logiciels importants (systèmes d'exploitation,etc..) a permis de montrer, depuis longtemps, les avantages d'une décomposition modulaire des traitements.

Les langages de programmation actuels supportent des notions de modules de plusieurs catégories, ainsi que nous l'avons rappelé au chapitre III. (DER79). Pour assurer la cohérence des choix de conception des modules, il faut:

— réduire le plus possible le nombre des concepts, quite à faire des compromis ou des adaptations pour des raisons évidentes de compatibilité entre les interfaces et la représentation des informations échangées entre modules

— que ces "modules" servent de support à une description systématique des traitements. Les modules de PARNAS (PAR72) apportent quelques éléments de réponse. Ce sont des modules fonctionnels qui regroupent dans une même unité de description les choix de réalisation de plusieurs opérations partageant des données communes, locales et

cachées. A l'exécution d'un programme, il ne peut exister qu'un seul exemplaire de ce type de module; sa durée de vie est fixée statiquement.

Dans un processus de conception "descendant" des traitements, il peut être utile de retarder certains choix de réalisation; autrement dit, de faire abstraction de détails de réalisation qui n'ont pas d'intérêt à ce niveau de conception. Il n'existe pas de notion de module pour décrire cela. On a besoin d'une unité de spécification modulaire qui soit totalement séparée de l'unité décrivant les détails de réalisation.

Nous avons donc défini un concept de "module" que l'on appelle machine : nous utilisons dans cette présentation la notation de Backus généralisée pour décrire la syntaxe.

machine < nom de machine >;
    < liste d'opérations abstraites >;
    ≠ <commentaire>
end

Une machine est une unité de description modulaire abstraite constituée d'une liste d'opérations abstraites. Ces opérations peuvent être des procédures et/ou des fonctions. La description d'une opération abstraite comporte: sa nature, son nom, la liste des types des paramètres données ainsi que la liste des types des paramètres résultats. Les fonctions sont des opérations sans "effet de bord".

Pour pouvoir utiliser ou implanter convenablement une machine, il faut d'autres informations "sémantiques" décrivant l'effet et/ou le comportement de l'ensemble des opérations. Dans l'état actuel du projet, ce genre d'information est donné sous forme de commentaires.

Une machine est une unité de compilation indépendante dont le texte source (y compris les commentaires), est stocké dans une bibliothèque particulière appelée bibliothécaire. Dès qu'une machine est compilée, elle est connue du système. On peut alors l'utiliser, en faisant appel à ses opérations, et cela indépendamment de tout choix d'implantation. Ce mécanisme d'abstraction est fondamental surtout si l'on veut que le langage de programmation puisse supporter une technique de décomposition modulaire et de raffinements successifs de tous les constituants d'une application base de données.

Dans le système TYP, une machine standard (appelée MAIN) définit un des points d'entrée possibles pour les programmes d'application. Les machines prédéfinies (systèmes dépendantes): lecteur (READER), écrivain (WRITER), etc... fournissent toutes les opérations abstraites d'entrées/sorties et de gestion des données sur supports secondaires souhaitables.

Nous appelons module une unité décrivant la réalisation d'une machine. Un module est défini de la façon suivante:

module < nom de module > for < nom de machine >;
    [resource < description de la structure de données
        interne > end]
    < suite de descriptions des réalisations des

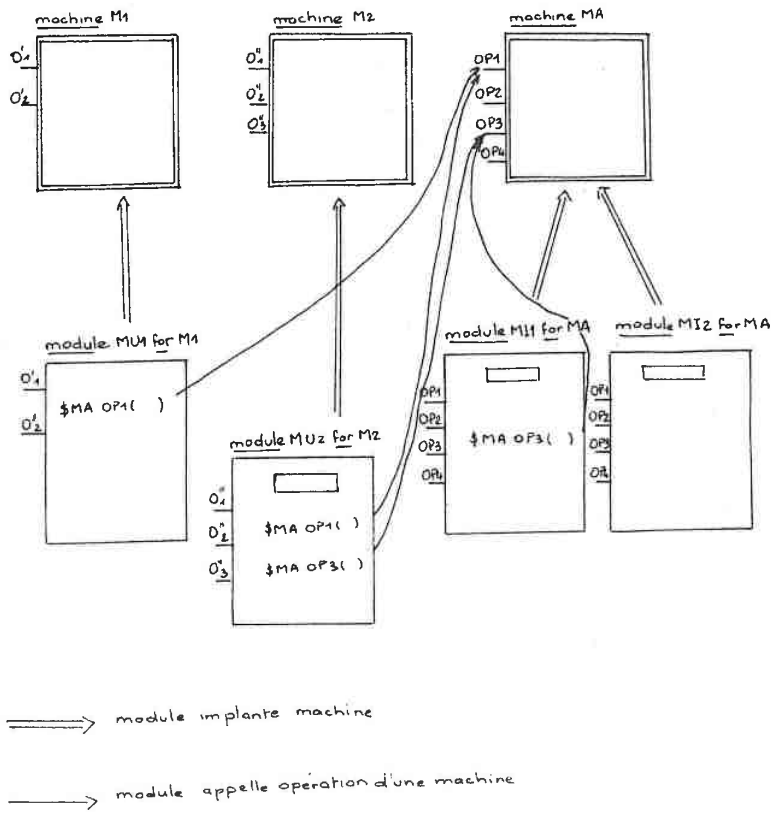
opérations abstraites >;

end

Notre définition de module ressemble à celle d'un module de PARNAS. Il importe toutefois de noter une distinction importante à la fois pour la conception des traitements et la cohérence des différents concepts de module du projet. Un module ATM est la "concrétisation"" dans une unité séparée des choix de réalisation d'une unité de spécification abstraite, ce qui n'est pas le cas pour un module de PARNAS.

La liaison entre une machine et un module qui l'implante est déclarée dans l'en-tête du module. Une partie resource facultative, déclare tous les objets partagés par les opérations du module. Ces objets locaux ne sont accessibles qu'à partir des opérations du module. A chaque opération abstraite de la machine correspond un algorithme d'implantation. Ce texte comporte des déclarations facultatives de variables locales, des instructions de type PASCAL ainsi que des appels d'opérations de machine (ou de type). Précisons que les opérations sont récursives. Un module est une unité de compilation séparée.

Dans un "environnement" donné, il est possible de définir plusieurs modules d'implantation d'une même machine. Toutefois, comme le système ne supporte pas de multi-représentation à l'exécution, la liaison entre une machine et un des modules d'implantation doit être figée au plus tard à l'édition de liens. Un langage spécifique dit de connexion (HEN80) sur lequel nous reviendrons en détail dans la suite, permet de définir la liaison.



Ce mécanisme fournit une aide efficace à la mise au point et à la maintenance des programmes. En effet, il permet de passer rapidement et facilement d'un choix de réalisation à un autre, entre deux exécutions d'un "même" programme.

La figure ci-contre schématise les relations entre les modules d'utilisation MU1, MU2, la machine MA et les modules d'implantation MI1 et MI2.

La machine MA déclare des opérations abstraites OP1, OP2, etc... qui sont appelées par un ou plusieurs modules d'utilisation MU1, MU2, etc... ou même parfois par les modules d'implantation MI1, MI2, etc... de MA. Lorsqu'on compile ces modules, le traducteur utilise le profil des opérations abstraites associées à chacun des appels pour faire tous les contrôles de types nécessaires.

La machine MA définit un interface abstrait des modules d'implantation MI1, MI2, etc..., de sorte que les modules d'utilisation MU1, MU2, etc... ne peuvent communiquer avec MI1 (ou MI2, etc..) qu'en faisant appel aux opérations de MA. La machine MA permet donc de cacher mais aussi de protéger les ressources gérées par son module d'implantation MI1 (ou MI2, etc..). Tous les modules d'utilisation MU1, MU2, etc.. partagent, à travers MA, la même ressource et les mêmes textes d'opérations d'accès à celle-ci.

II. DESCRIPTION DES TYPES DE DONNEES: TYPES ET CAPSULES.

L'étude détaillée des modèles de données a montré que les langages des bases de données n'utilisent pas de concept de type de données, tout au moins avec la signification habituelle des langages de programmation typés (même la plus large); citons par exemple MC LEOD (SIG76): "In classical data models the data space is partioned into "types" and "instances". Types correspond to the schema on the logical structure of the database and instances correspond to data, the population of the types" (!)

En fait, ce n'est que depuis l'apparition des langages de manipulation de bases de données relationnelles insérés dans des langages de programmation typés comme PASCAL (SMI76) que l'on a véritablement commencé à se poser le problème de l'introduction des types de données dans les bases de données (SIG80).

Dans le projet TYP, notre travail a d'abord consisté à entreprendre d'importantes investigations tant dans le domaine des langages de programmation que dans celui de la spécification de problèmes pour essayer de dégager des concepts de types de données qui permettent de développer de manière cohérente des applications bases de données.

L'étude des langages de programmation nous a amené à nous intéresser à l'abstraction de types de données supportée par certains nouveaux langages. Les modules de CLU (LIS74) nous ont particulièrement intéressé.

En effet, avec ce genre de module , un type de données est défini comme un ensemble d'objets et un ensemble d'opérations. La représentation des objets est cachée à l'intérieur du module et les opérations déclarées externes sont les seules opérations applicables aux objets. Par rapport aux mécanismes "classiques" de construction de types, un module de CLU est une forme d'abstraction de la représentation des objets des types de données. C'est la raison pour laquelle on parle de type abstrait de données. Nous ne reviendrons pas ici sur les modules de CLU qui ont été présentés en III.3.

Quand on conçoit des types de données ou quand on les met en oeuvre dans les programmes d'application, les choix de réalisation décrits dans les modules de CLU (représentation des objets et algorithmes des opérations abstraites) ne sont à priori d'aucun intérêt. Nous précisons bien à priori, car en l'abscence de spécifications simples, compréhensibles du comportement externe de ces opérations, les utilisateurs sont dans l'obligation d'aller consulter le contenu de ces modules. Les langages de programmation avancés ne fournissent pas de mécanisme de description modulaire permettant de séparer totalement la spécification abstraite des types de données de la description des choix de réalisation sous forme de module à la CLU.

En ADA, nous avons vu que l'interface abstrait d'un type de données était aussi lié à une représentation cachée des objets: par contre, les algorithmes sont décrits dans une unité séparée.

Nous avons donc défini un autre concept de "module" appelé type:

```
type < nom de type >;
     < liste d'opérations abstraites >;
     ≠< commentaires >
end
```

Cette unité de spécification modulaire permet de définir un type abstrait de données indépendamment de tout choix de réalisation: représentation des objets et algorithmes associés aux opérations abstraites. Elle comporte une spécification syntaxique des opérations applicables à tout objet déclaré de ce type, et une spécification sémantique des opérations, sous forme de commentaires. Les opérations d'un type sont de deux sortes: procédures et fonctions. Pour renforcer les contrôles statiques de types (à la compilation) nous avons différencié les procédures, compte tenu de leur effet sur l'argument associé au type : lecture, écriture ou modification.

La description d'une opération comporte donc sa machine, son nom, la liste des types paramétrés données et la liste des types paramétrés résultats. Le type que l'on spécifie est un paramètre implicite appelé "paramètre principal" de l'opération. Nous avons distingué quatre natures d'opérations:

-- la construction notée BUILD : Cette opération construit un objet, avec éventuellement des données passées en paramètre. A la fin de l'exécution de cette opération, l'argument associe au paramètre principal désigné l'objet construit.

-- la modification notée MODIF : Elle a pour effet de modifier l'objet désigné par l'argument principal de l'opération, ou encore de

fabriquer un nouvel objet.

-- la consultation notée CONSULT : C'est une opération de lecture. Elle délivre un ou plusieurs résultats.

-- et une opération de lecture notée FUNCTION : C'est une opération de consultation qui délivre un et un seul résultat.

Ajoutons que la distinction qui est faite dans la description des paramètres, entre paramètres données et résultats, permet d'effectuer des contrôles statiques de validité de la correspondance arguments-paramètres, à l'appel d'une opération. La spécification sémantique d'un type, décrite sous forme de commentaires, doit fournir des informations précises et compréhensibles. Il nous semble qu'un bon compromis consiste à combiner des définitions en langue naturelle avec des spécifications formelles, aussi proches que possible de VEGA, comme le montre l'exemple de Pile d'Eléments décrite ci-dessous:

```
TYPE Pilelt;
   ≠IMPORTS(Elt;Bool);
≠OKOPNS;
BUILD pilevide;
   ≠effet: construction d'un objet Pilelt: pilevide
MODIF empiler(Elt);
   ≠effet: ajoute l'élément de type Elt au sommet de la pile
   ≠désignée par l'argument principal
MODIF dépiler --> Pilelt,Bool;
   ≠effet: si pilevide
   ≠       alors pilerr et Bool=true
   ≠       sinon enlève l'élément au sommet de la pile
   ≠             désignée par l'argument principal et
```

```
    ¢              délivre un nouvel objet Pilelt avec un
    ¢Booléen=false
  CONSULT sommet --> Elt,Bool;
    ¢effet: si pilevide
    ¢       alors somerr et Bool=true
    ¢       sinon délivre l'élément au sommet de la pile
    ¢              désignée par l'argument principal et un
    ¢Booléen=false
  FUNCTION estvide RETURNS Bool;
    ¢effet: si pilevide alors true sinon false
  ¢ EROPNS
    ¢ ERROR : pilerr <- Pilelt;
    ¢ ERROR : somerr <- Elt;
  ¢ OKAXS
    ¢p: Pilelt,i:Elt;
    ¢1:dépiler(empiler(p,i))=p;
    ¢2:sommet(empiler(p,i))=i;
    ¢3:estvide(pilevide)=true;
    ¢4:estvide(empiler(p,i))=false;
  ¢ ERAXS
    ¢1:dépiler(pilevide)=pilerr;
    ¢2:sommet(pilevide)=somerr;
  END
```

Précisons que les lignes de commentaires débutent par le caractère "¢".

Un type T est une unité de compilation indépendante. A la compilation de l'unité T, le système range le texte source

(commentaires compris) dans un bibliothécaire. Si, au moment de la compilation de T, T utilise (ou importe) des noms de types non spécifiés, T est compilé et le système enregistre le (ou les) nom(s) des autres comme étant en attente de spécification. Un type qui n'utilise pas de nom de type en attente de spécification est appelé type défini . Seuls les types définis peuvent être "mis en oeuvre" dans les programmes. Cela se fait en déclarant des variables de ces types et, si t est une variable déclarée de type défini T, en appliquant à t des opérations abstraites de T. Le compilateur vérifie, entre autres choses, que les différentes opérations appliquées à t sont licites, c'est-à-dire qu'elles figurent effectivement dans la liste des opérations de T.

Exemple de type prédéfini (description partielle):

```
  TYPE FIND;
    ¢fichier indexé
  BUILD CRE(TEXT,INTEGER,INTEGER,TEXT,INTEGER,INTEGER);
    ¢création d'un objet fichier indexé
    ¢TEXT:étiquette opérationnelle
    ¢INTEGER:longueur d'enregistrement
    ¢INTEGER:longueur de bloc
    ¢TEXT:nom interne du fichier
    ¢INTEGER:longueur de la clé
    ¢INTEGER:position de la clé dans l'enregistrement
  MODIF OPENG;
    ¢ouverture du fichier en lecture
  .
  .
```

```
CONSULT GET(INTEGER) --> TEXT,BOOLEAN;
   ¢lecture d'un enregistrement(TEXT) de clé(INTEGER)
   ¢si fin de fichier ou anomalie alors BOOLEAN=faux
   ¢                              sinon BOOLEAN=vrai
.
.
MODIF KILL;
   ¢suppression de l'objet fichier indexé
END
```

Ici, nous pourrions décrire le comportement des opérations du type FIND avec des traces de PARNAS (PAR77) (DER79). Les types prédéfinis sont communs à tous les environnements des utilisateurs et constituent un ensemble de structures élémentaires permettant de définir et de réaliser de nouveaux types ATM.

Exemple de type défini par l'utilisateur :

```
TYPE CHERCH;
   ¢type chercheur
BUILD CREER(INTEGER,TEXT);
   ¢création d'un objet chercheur
   ¢INTEGER:numéro du chercheur
   ¢TEXT:nom du chercheur
FUNCTION CLE RETURNS INTEGER;
   ¢fonction qui délivre le numéro du chercheur
FUNCTION NOM RETURNS TEXT;
   ¢fonction qui délivre le nom du chercheur
CONSULT PRINT;
   ¢impression d'un chercheur
```



```
FUNCTION EQ(CHERCH) RETURNS BOOLEAN;
   ¢égalité de deux chercheurs
END
```

Exemple d'utilisation d'un type :

On utilise une notation infixée pour distinguer parmi les paramètres celui qui est du type d'intérêt.

```
.
.
VAR
   C:CHERCH;
END
   ¢déclaration d'une variable C de type CHERCH
BEGIN
.
   C.CREER(10,"JEAN");
      ¢création d'un objet chercheur
      ¢dans la variable C et initialisation
      ¢de celle-ci avec le numéro 10 et le nom "JEAN"
.
   C.PRINT;
      ¢impression des caractéristiques du chercheur
      ¢désigné par C
END
```

La notation introduite permet en fait de réaliser des contrôles de type plus fins qu'en CLU (LIS75) ou même qu'en ADA (ADA79), sans que la lisibilité des programmes en soit pour autant diminuée. Comme en CLU, une déclaration de variable n'a pas d'effet. Pour construire

effectivement un objet, il faut appliquer une opération BUILD (LIS75).

Maintenant, un programme ne sera exécutable que si tous les types qu'il met en oeuvre possèdent au moins une implantation. Les choix de réalisation (implantation) d'un type se décrivent dans des unités appelées capsules.

Une capsule est un module fonctionnel de la forme suivante:

capsule <nom-de-la-capsule> for <nom-de-type>

rep <description du type concret> end ;

<suite de descriptions de réalisation d'opérations abstraites>

end

Une capsule est une unité de compilation séparée qui ressemble à peu de choses près à un module CLU. Cette unité permet de préciser et de localiser les choix d'implantation d'un type: la structure concrète des objets et les algorithmes réalisant les opérations abstraites qui ont été spécifiées dans l'unité type.

Ainsi, la spécification d'un type est totalement indépendante des choix d'implantation décrits dans une capsule. Le texte d'une capsule comporte son nom et le nom du type qu'elle réalise, suivis de la description de la représentation des objets (rep .. end) et, pour chaque opération abstraite du type, d'un entête et d'un corps d'opération qui la réalise.

Toutes les variables déclarées d'un certain type partagent le même texte de capsule. De ce fait, nous avons dû adopter quelques conventions d'écriture pour désigner les objets concrets à l'intérieur des capsules. Si t est une variable déclarée de type T, alors l'appel d'une opération de T sur t a pour effet de convertir (de façon implicite) t dans le type concret c'est à dire rep . Par "abus" de

notation, rep peut être utilisé, à l'intérieur de la capsule, pour désigner l'objet converti.

Du fait de la séparation des spécifications (types) et des définitions concrètes (capsules), un utilisateur peut concevoir, réaliser et compiler séparément plusieurs capsules pour un même type. Dans ce cas, elles appartiennent toutes à l'environnement de cet utilisateur et sont toutes gérées par le bibliothécaire du système. Cependant, comme à l'exécution, le système ne supporte, pour un type donné, qu'une implantation, il faudra fixer, avant une exécution, quelle capsule doit lui être associée. Un ensemble de directives d'un système de fabrication dynamique de programmes appelé connecteur, intégré au système TYP, permet de décrire cette "connexion". En CLU, cette connexion nécessite l'utilisation d'un langage de commande externe.

Exemple de capsule :

```
CAPSULE CH1 FOR CHERCH;
   ≠réalisation du type CHERCH avec la capsule CH1
REP
   NO:INTEGER;
   ≠numéro du chercheur
   N:TEXT;
   ≠nom du chercheur
END
   ≠représentation des objets du type CHERCH
BUILD CREER(I:INTEGER,T:TEXT);
   BEGIN NO:=I;N:=T; END
FUNCTION CLE RETURNS INTEGER;
```

```
   BEGIN RETURN NO; END
FUNCTION NOM RETURNS TEXT;
   BEGIN RETURN N; END
CONSULT PRINT;
   VAR T1,T2:TEXT;
   END
   BEGIN
      T2:='';
      T2.AJUST(14);
      /initialisation à vide d'une zone text de longueur 14
      T1:=$CONV TEXTINT(NO);
      /conversion de l'entier NO en text
      T2.SUBD(T1,1,4);
      /affectation de T1 à la zone text(1,4) de T2
      T2.SUBG(N,5,14);
      /affectation de N à la zone text(5,14) de T2
      T:=T2;
      /conversion de la représentation d'un objet CHERCH en TEXT
      $WRITER WRITE(T);
      /impression du texte
   END
FUNCTION EQ(C:CHERCH) RETURNS BOOLEAN;
   BEGIN
      IF NO=C.CLE THEN RETURN TRUE;
      ENDIF;
   END
END
```


Précisons que la structure concrète des objets du type CHERCH, localisée dans la partie REP -- END de la capsule CH1, n'est visible qu'à l'intérieur de cette capsule. La seule possibilité d'accéder, de l'extérieur, à cette représentation, est de faire appel aux opérations abstraites du type. Le mécanisme de (type-capsule) permet ainsi de cacher et de protéger la représentation des objets et par-là, de garantir leur intégrité au cours des différentes manipulations.

Notons encore que le concepteur dispose de beaucoup de facilités pour entreprendre différents choix d'implantation, retenir l'un d'eux ou encore modifier celui-ci entre deux exécutions de programme.

Cette localisation des choix de réalisation d'un type dans une capsule parfaitement étanche et cette possibilité de remplacer une capsule par une autre (à condition qu'elle reste conforme aux spécifications du type), sans qu'il soit nécessaire de changer un seul mot dans les modules qui utilisent le type, sont des caractéristiques essentielles d'ATM permettant une grande évolutivité du logiciel.

Concrètement, la "mise en oeuvre" des types s'effectue en déclarant des variables dans les différentes unités de réalisation: modules et capsules. Par exemple, le module U1 utilise plusieurs déclarations de variables de type TI1: dans la partie resource -- end , en paramètre de l'opération O2 mais aussi comme variable locale de l'opération O1. Pour pouvoir compiler U1, il faut que tous les types utilisés soient connus du système (définis); ceci pour être en mesure de faire tous les contrôles de types nécessaires. Il en va de même pour la capsule C1 qui implante TI1. L'évaluation, à l'exécution, d'une déclaration de variable déclenche l'appel d'une opération interne (create) de la capsule associée au type qui crée un objet en mémoire. Tous les objets

d'un type ont la même représentation mémoire; leur durée de vie dépend de la localisation des déclarations: resource, paramètre ou variable locale d'une opération, d'un module ou d'une capsule (MIN79).

Le mécanisme de type que nous venons de présenter réalise non seulement l'indépendance des programmes d'utilisation vis à vis des réalisations mais encore une protection efficace des accès aux données. En fait, les opérations d'un type déterminent des droits d'accès aux objets (HEN80).

Nous ne détaillerons pas plus le langage ATM. Nous renvoyons les lecteurs intéressés aux thèses de 3ème cycle de Régis MINOT (MIN79), de Patrick HENRY (HEN80), ainsi qu'aux rapports de DEA de Karol PROCH (PRO80) et de Najib TOUNSI (TOU80).

III. CONSTRUCTION MODULAIRE DES BASES DE DONNEES.

Le concept de module d'ATM fournit un moyen particulièrement adapté pour définir un système de base de données (WEB78) (ROW79) puisqu'il permet de conserver des données et d'en définir les fonctions d'accès. La seule possibilité d'accéder de l'extérieur d'un module à sa ressource "données" est d'appeler une des opérations de la machine qu'il implante; cette sorte de machine est alors appelée machine base de données . La totale séparation entre les machines bases de données et les systèmes de base de données associés réalise une réelle "indépendance des données" (DAT77).

III.1. Les machines et les systèmes de base de données.

Dans le contexte base de données où nous nous plaçons, une abstraction d'un module de gestion de données ATM: système de base de données , apparaît comme une sorte d'objet abstrait dont les caractéristiques: nom, ensemble d'opérations, comportement de celles-ci sont spécifiées par une machine. Le nom de cet objet qui est aussi le nom de la machine (base de données) a une portée primale (BAN77) dans l'environnement de l'utilisateur. En ce sens, une machine base de données définit un objet global qui peut être partagé par plusieurs programmes d'utilisation. On retrouve, de cette façon, une des propriétés caractéristiques des objets relations, schémas des systèmes de gestion de base de données.

Pour illustrer ces notions, nous allons décrire la machine CHERCHS et le système de base de données CHE de l'application bibliothèque:

```
MACHINE CHERCHS;
   ¢machine base de données chercheurs
   PROC CREER;
      ¢initialise la base des chercheurs
   PROC ACCHERNO(INTEGER) --> CHERCH,BOOLEAN;
      ¢accès à un chercheur de numéro INTEGER
      ¢ si le chercheur de numéro INTEGER n'existe pas
            ¢ alors BOOLEAN=faux
            ¢ sinon BOOLEAN=vrai
END
```

Le type CHERCH a été spécifié en II.

Nous ne donnons ici qu'une description partielle du module CHE

définissant le système de base de données associé à la machine base de données CHERCHS:

```
MODULE CHE FOR CHERCHS;
   ¢système de base de données CHE
   RESOURCE
      ¢structure concrète de l'objet abstrait CHERCHS:fichier indexé
      NF:FIND;
   END
   PROC CREER;
   .
   PROC ENLEVER(I:INTEGER) --> SUCCES:BOOLEAN;
      VAR
         ¢déclaration des variables locales à la procédure
         E:CHERCH,B:BOOLEAN;
      END
      BEGIN
         ¢appel de l'opération ACCHERNO de la machine CHERCHS
          CHERCHS ACCHERNO(I) --> E,B;
         IF B THEN NF.DELETE ENDIF;
         SUCCES:=B;
      END
      .
END
```


III.2. La conservation des données permanentes dans le système.

La technique de construction modulaire des systèmes de bases de données que nous proposons consiste à concevoir et à implanter de façon modulaire les différentes possibilités d'accès aux données stockées sur mémoires secondaires depuis des machines bases de données.

Jusqu'ici, nous nous sommes surtout intéressé à la gestion des données en mémoire centrale. Lorsqu'on aborde la conservation des objets sur mémoire secondaire, on se trouve confronté à deux types de problèmes. D'abord, comment stocker et gérer les données sur mémoires secondaires et ensuite, comment définir de façon cohérente les liaisons entre les programmes en mémoire centrale et les données stockées sur disque.

Pour résoudre ces problèmes, nous avons envisagé plusieurs approches possibles:

-- privilégier des ressources permanentes dans l'espace virtuel de travail du système TYP,

-- utiliser ou encore développer un système de gestion de données capable de fournir des méthodes d'accès physiques adaptées au contexte de bases de données,

-- ou, plus simplement, utiliser le système de gestion de fichiers SIRIS 8 de l'IRIS 80.

Après réflexion, il nous a semblé qu'interfacer le système de gestion de fichiers SIRIS 8 fournissait une solution à la fois élégante et orthogonale avec les autres choix de conception du système TYP. Il est évident que si nous avions disposé d'un système de gestion de données plus élaboré, nous aurions pu l'interfacer de la même façon.

Autrement dit, notre choix n'exclut pas la deuxième solution.

Par contre, si l'on privilégie une ressource permanente, cela conduit à introduire des machines et des modules permanents, c'est-à-dire des machines et des modules qui permettent de "figer" des liaisons entre une abstraction de données et des fichiers de données sur disque. Le projet CIVA auquel nous avons participé proposait déjà ce genre d'idée. Nous renvoyons le lecteur intéressé par cette approche à la thèse de J.C.DERNIAME (DER73).

Un exemple d'interface du système de gestion de fichiers SIRIS 8 réalisé par J.CHABRIER (CHA82) a été donné précédemment: type fichier indexé FIND.

III.3. Gestion des systèmes de base de données.

Dans ce paragraphe, on s'intéresse uniquement à la mise en oeuvre des unités compilées qui composent des systèmes de bases de données. Plusieurs solutions sont, là encore, envisageables. Une première approche consiste à "éditer" tous les systèmes de base de données d'un environnement utilisateur, pour en faire un "monolithe" résident en mémoire centrale. Dans ce cas, les programmes utilisateurs sont eux édités et chargés à la demande. Cette solution ne nous satisfait pas pour deux raisons, au moins:

-- d'abord parce que le "monolithe" peut être un programme résident très volumineux. Or, il nous a paru que l'exécution d'un grand nombre de requêtes n'exploitaient qu'une très faible partie de ce programme. Cela rejoint un des reproches que nous avons déjà eu l'occasion de formuler à l'encontre des SGBD traditionnels.

-- ensuite, parce qu'il est très difficile et coûteux de prendre en compte l'évolution des contraintes de sécurité et d'intégrité (MEN80).

Une autre solution consiste à éditer les programmes utilisateurs avec des systèmes de base de données déjà édités ou préédités, puis à faire un chargement en mémoire centrale suivi de l'exécution. Les inconvénients sont du même ordre que dans l'approche précédente (MIN79).

Nous avons préféré une solution "dynamique", ce qui nous a amené à développer un outil spécifique appelé connecteur (HEN80). Cet outil réalise une édition de liens dynamique des seules unités nécessaires à l'exécution d'un programme d'application. Cette édition que nous appelons connexion intervient à chaque premier appel d'une opération d'un type ou d'une machine. A ce moment là, le connecteur dispose de toutes les informations utiles pour décider s'il doit ou non satisfaire la demande de connexion. Un langage de connexion, il serait plus juste de parler de directives, permet de définir pour chaque exécution les liens entre des noms de machines et de types et respectivement les noms de modules et de capsules d'implantation que l'on a choisi d'utiliser.

III.4. La maintenance des systèmes de bases de données.

La maintenance de ce genre de logiciel peut demander plus ou moins de travail selon qu'on modifie des textes de capsules et/ou de modules, des types et/ou des machines ou que l'on change (au plus bas niveau d'abstraction) par exemple, des méthodes d'accès de fichiers. Dans ce dernier cas, l'administrateur doit souvent procéder à des conversions de fichiers.

IV. DESCRIPTION MODULAIRE DES CONTRAINTES D'INTEGRITE ET DE SECURITE.

Dans un environnement base de données, un des objets les plus sensibles est certainement la base de données. Les seuls agents qui opèrent directement sur elle sont les systèmes de base de données et le système d'exploitation. Bien que les problèmes liés au système d'exploitation soient essentiels, nous ne les avons pas abordés dans le cadre de ce travail. Nous nous sommes limités aux contrôles des accès des systèmes de base de données. Il importe de pouvoir contrôler tous les accès possibles à la base: lectures, écritures et modifications. Cela peut vouloir dire: interdire, autoriser avec ou sans restrictions.

Les machines base de données fournissent un premier niveau de contrôle global des opérations d'accès à la base. En effet, à partir du moment où une opération de lecture, d'écriture ou de modification est disponible, elle permet d'accéder quasiment sans restriction à toute une collection de données. C'est précisément ce que l'on voudrait éviter ou contrôler. Pour cela, il faut un autre mécanisme. On doit

contrôler qu'une opération de lecture d'une machine base de données ne délivre que certaines valeurs de la base, ou qu'une opération d'écriture ou de modification n'introduit pas d'incohérence. On parle alors de contrôle de contraintes d'intégrité de la base.

Un SGBD est conçu pour être manipulé par des utilisateurs (ou programmes). Il importe donc de pouvoir masquer des machines base de données ou certaines de leurs opérations avec ou sans restriction pour certains utilisateurs ou groupes d'utilisateurs. Pour réaliser le contrôle des contraintes de sécurité, un autre mécanisme doit être conçu. Dans les SGBD traditionnels, ces contraintes sont déclarées dans des sous-schémas.

Les contraintes de sécurité et d'intégrité des machines base de données déterminent des conditions d'"utilisation" des machines et/ou de certaines de leurs opérations. Les contrôles globaux sont statiques. Par contre, les contrôles portant sur les valeurs sont essentiellement dynamiques. Nous proposons de les décrire séparément dans des machines spécifiques appelées machines de contrôle . Leur mise en oeuvre s'effectue au moyen du connecteur. Le mécanisme de modification d'appel d'opérations des machines base de données du connecteur ressemble à la technique de modification dynamique de requêtes développé dans INGRES (HENBO).

En résumé, pour définir les contraintes de sécurité et d'intégrité, nous essayons d'appliquer les principes de base que sont l'abstraction et la modularité. Ainsi, les contraintes sont conçues le plus tard possible et décrites dans des unités complètement séparées des machines base de données .

V. LA MANIPULATION DES DONNEES.

Les applications base de données peuvent comporter des "petits" programmes de manipulation de données, mais aussi des "gros" programmes de traitement. Selon la nature des besoins et le profil des utilisateurs du système TYP: non programmeurs, programmeurs "terminaux", programmeurs experts, ou programmeurs "système", TYP fournit plusieurs niveaux de services correspondants à des abstractions de langage.

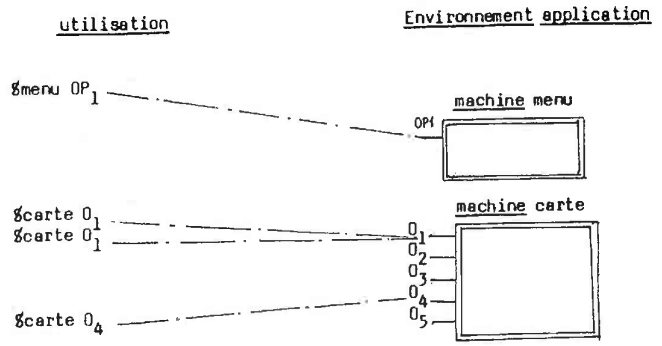
V.1. Pour les non-programmeurs.

Cette catégorie d'utilisateurs n'emploie que des applications préprogrammées sous forme de "menus": ce sont des machines qui ne comportent qu'une opération sans paramètre, ou encore "à la carte": les machines possèdent alors plusieurs opérations sans paramètre. La figure ci contre illustre ce type d'utilisation.

Dans ce contexte, le simple fait d'"appuyer" sur le "bouton" d'une machine a pour effet de connecter l'utilisateur à une application préprogrammée.

L'automate d'interaction du système TYP (cf Chapitre XII) a été conçu à partir de cette idée.

Les environnements de cette catégorie d'utilisateurs ne comportent que des machines "presse-boutons". Pour permettre d'illustrer le concept de machine "à la carte", nous pouvons faire le parallèle avec l'utilisation d'un téléviseur. En effet, l'utilisateur d'un tel poste,



qui n'est pas un électronicien confirmé, ne manipule que les boutons de commande de l'appareil. Il ignore tout de la partie électronique qui est cachée à l'intérieur du meuble. Il en est de même des machines "presse-boutons". Tout le logiciel modulaire de réalisation est caché dans la machine.

V.2. Pour les programmeurs "terminaux".

Les utilisateurs de cette catégorie veulent pouvoir écrire de petits programmes de manipulation de bases de données, sans pour autant être obligés de connaître les programmes d'applications. Les langages de manipulation de données des SGBD ont précisément été conçus pour cette catégorie d'utilisateurs. TYP fournit des constructions de langages de haut niveau s'appuyant sur des mécanismes d'itérateurs abstraits et d'itérations.

Un itérateur abstrait est une opération abstraite d'une nouvelle nature qui peut être spécifiée pour un type ou pour une machine base de données. Les itérateurs sont des opérations particulières qui ne peuvent être appelées que dans des itérations. L'utilisation d'itérateurs dans les itérations permet de faire totalement abstraction, à la fois de la représentation interne des objets et de la stratégie d'accès employée. Les programmes de manipulation de données qui utilisent de telles itérations sont aussi indépendants des choix faits dans les modules (resp. capsules) implantant les itérateurs des machines (resp. types). La "data independance" (DAT78) permettra de réaliser une implantation efficace et une véritable maintenance des

S.B.D.

Pour pouvoir contrôler statiquement (à la compilation) la manipulation des objets à l'intérieur du corps des itérations, nous distinguons trois natures d'itérateurs:

. les " selector " délivrent des objets uniquement en consultation,

. les " updator " en modification,

. et les " deletor " en suppression. L'écriture de ces itérations n'exige pas un gros effort de programmation. Donnons un exemple:

```
machine CH1;
selector consulter yields CHERCH
selector informaticien yields CHERCH
updator changer yields CHERCH
.
end
```

Q1:"lister tous les chercheurs de la base"

```
module transaction1 for user1;
        ≠ programme de manipulation
proc requete;
        ≠ requete qui liste tous les chercheurs de la base
consulter all e in CH1
        ≠ début itération de consultation sur la machine CH1
        ≠ e est une variable de contrôle locale de l'itération
        ≠ déclarée implicitement de type CHERCH
for e.print;
        ≠ impression du chercheur
end ;
end requête
```



```
        ≠ fin de l'itération
end transaction1
```

L'itération est une sélection qui appelle l'itérateur abstrait consulter de la machine CH1.

Q2:"augmenter de 10% le salaire du chercheur 10"

```
module transaction1 for user1;
proc requete;
    changer one e in CH1 where e.numéro=10
    by msal( e.salaire*1.10)
    else print message("employé inexistant");
end requete;
end transaction1
```

L'itération est une modification qui appelle l'itérateur abstrait changer de la machine CH1, pour appliquer au chercheur sélectionné (one e in CH1 where e.numéro=10), s'il existe, la modification de 10% de son salaire actuel (e.msal (e.salaire*1.10))

On constate que les mécanismes d'itérateur-itération permettent d'écrire relativement facilement des requêtes de manipulation de données. Notons que ces transactions sont ici compilées avec un certain nombre de contrôles de types alors que dans la plupart des SGBD-INGRES, RIGEL, etc..., elles seraient interprètées.

Aussi l'environnement de cet utilisateur pourra comporter des machines "presse-boutons", des machines base de données appropriées c'est-à-dire avec des opérations sans paramètre et des itérateurs abstraits, des types et des machines de service pour les entrées/sorties console.

V.3. Pour les programmeurs "experts".

Ces utilisateurs sont en réalité les programmeurs qui réalisent les applications bases de données.

Pour décrire les implantations des abstractions (machines et types), les programmeurs disposent de toute la puissance du langage de programmation ATM: des constructions de base avec des instructions de type PASCAL, et les facilités que nous venons de présenter pour les autres utilisateurs.

En résumé, nous proposons de fournir aux différentes catégories d'utilisateurs du système TYP, les constructions de langage qui paraissent le mieux répondre à leurs besoins. Toutes les couches de langage que nous venons de présenter n'utilisent qu'un seul traducteur.

V.4. Pour les programmeurs système.

Pour conserver, dans le langage de programmation noyau du système TYP, d'entrée-sortie console, les types fichiers, etc... Tous les interfaces du système sont décrits avec les concepts du langage de programmation. Ainsi, pour le programmeur "expert", il n'y a aucune différence entre ces interfaces (prédéfinis) et les machines ou types qu'il peut construire. Par contre, les implantations des interfaces "système dépendants" sont décrites dans un autre langage, en l'occurrence le langage assembleur de l'ordinateur IRIS 80. Pour que les modules et capsules rédigés en langage de bas niveau restent compatibles avec les autres, nous avons adopté des règles de

programmation. Seuls des programmeurs "système" sont en mesure d'écrire ces unités. Précisons qu'une fois réalisées, ces unités seront incorporées par l'administrateur , lors de la phase d'initialisation de l'environnement des applications et que les implantations seront complètement cachées aux utilisateurs.

VI. META-LANGAGE: DEFINITION D'UNITES GENERIQUES.

VI.1. La notion d'unité générique (PRO79).

Très vite, la construction d'applications bases de données permet de faire deux constatations:

-- les concepteurs-réalisateurs doivent rédiger à la main un grand nombre d'unités: machines, types, modules et capsules.

-- beaucoup de textes produits ne diffèrent que par "peu de chose", essentiellement par des noms de types ou de machines. Il nous a donc paru important de pouvoir à la fois faciliter et améliorer cette production de textes, en essayant de l'automatiser le plus possible. Nous avons choisi une solution relativement simple qui consiste à définir des unités génériques (ADA80). Autrement dit, le méta-langage que nous proposons est une couche de langage externe au langage de programmation ATM.

Une unité générique est une méta-description d'unités ATM, paramétrée par des types et/ou des machines avec les opérations requises pour la spécification ou l'implantation, et/ou des constantes de type pour lesquelles une notation est définie.

VI.2. Ecriture des unités génériques.

Nous distinguons quatre définitions d'unités génériques: les types (typegen), les capsules (capsgen), les machines (machgen) et les modules (modgen).

La spécification des paramètres des unités génériques n'est décrite que dans les méta-descriptions abstraites types (typegen) et machines (machgen). Ces paramètres ne peuvent pas eux-mêmes être paramétrés.

En outre, contrairement à CIVA (DER74) ou ADA (ADA80), les unités génériques devront obligatoirement être définies et utilisées à l'extérieur des unités de base du langage de programmation. Cet outil est essentiellement conçu pour faciliter la manipulation de textes ATM.

La description des unités génériques est très proche des unités de base. Donnons en un exemple avec la description du type ensemble borné paramétré par le type élément:

```
typegen ensemble;
   ¢spécification du type générique ensemble
   ¢d'objets de type élément
   ¢borné par N
param
   type élément( function eq(élément) returns Boolean),
   const integer N
      ¢la spécification algébrique ci-dessous met
      ¢en évidence la nécessité d'une opération
      ¢eq pour le type paramètre puisque cette
      ¢opération est utilisée pour définir les
      ¢axiomes du type
```



```
   end
opérations
   build vide;
      ¢construction d'un objet ensemble
   modif ajouter(élément);
      ¢ajoute une valeur de type élément dans l'ensemble
   modif retirer(élément);
      ¢ enlève de l'ensemble la valeur donnée en paramètre
   function estdans(élément) returns boolean;
      ¢teste si la valeur appartient à l'ensemble
   function card returns integer;
      ¢délivre le nombre de valeurs de l'ensemble
sémantique
   ¢dcl:e:ensemble élément ,x,y:élément,n:integer;
   ¢card(vide(e))=0;
   ¢card(ajouter(e,x))= si non(estdans(e,x)) et
         ¢card(e)<n alors card(e)+1
         ¢ sinon card(e) finsi
   ¢estdans(vide(e),x)= faux ;
   ¢estdans(ajouter(e,x),y)= si eq(x,y) alors vrai
         ¢ sinon estdans(e,y) finsi
   ¢retirer(vide(e),x)=vide(e);
   ¢retirer(ajouter(e,x),y)= si eq(x,y) alors
         ¢retirer(e,y)
         ¢ sinon ajouter(retirer(e,y),x) finsi
end
```

Toutes les unités génériques sont traduites séparément et là encore,

la spécification sémantique n'est qu'un commentaire. Toutefois, on constate qu'ici, la spécification formelle fournit des indications sur les opérations requises sur les types: dans l'exemple, l'opération eq sur le type élément en est l'illustration.

La description d'une capsule générique est identique à une capsule simple au mot-clé capsgen près. Donnons maintenant la description (partielle) d'une capsule générique ensg qui implante le type générique ensemble élément, N:integer

```
capsgen ensg for ensemble;
   rep
      ≠représentation des ensembles
   tab: array (N) of élément;
   nbre:integer;
   end
build vide;
   begin
      nbre:=0;
      ≠initialisation de la représentation
   end
modif ajouter(e:élément);
   var
      i:integer,trouvé:boolean;
   end
   begin
      i:=1;trouvé:= false ;
      while (i=nbre and not trouvé);
      .
```



```
   end
   .
end
```

Une fois compilés, les types et les capsules génériques sont gérées par le bibliothécaire des unités génériques. A la compilation d'une capsule générique, on vérifie que les opérations formelles appliquées aux objets des types paramètres du type générique sont bien celles qui ont été spécifiées. Ces contrôles statiques, similaires au contrôle de "liste d'importation" (CUN79) renforcent la fiabilité du logiciel.

VI.3. Utilisation des unités génériques.

La production d'unités de description de base est réalisée par instanciation d'unités génériques. Un langage simple, constitué de directives define a été défini. A la traduction de ces directives, des contrôles sont effectués sur la correspondance argument-paramètre formel, permettant de mieux localiser les erreurs, comparativement au mécanisme de généricité d'ADA (ADA80). Par exemple, pour instancier un ensemble de 100 chercheurs à partir du type générique ensemble élément,N:integer , nous écrivons d'abord la définition du type enscher:

```
define type enscherch=ensemble with cherch(eq), 100 end
```

puis la définition d'une capsule associée:

```
define capsule ensc=ensg of ensemble with cherch(eq),100 end for enscherch;
```

En résumé, cet outil peut être regardé comme un mécanisme

d'abstraction de "certains détails" des unités de base; essentiellement des types, des constantes et des opérations de machines.

CHAPITRE XI. NOTION D'APPLICATION: APPLICATION REFERENCE, APPLICATION SECTORIELLE, APPLICATION CONNEXION.

I. APPLICATION BASE DE DONNEES.

II. SCHEMAS DE L'ORGANISATION D'A.B.D.

III. APPLICATION REFERENCE.

IV. APPLICATIONS SECTORIELLES.

V. CONTEXTE GLOBAL D'UNE BASE DE DONNEES D'UNE APPLICATION D'A.B.D.

VI. APPLICATION CONNEXION.

VII. EXECUTION D'UN PROGRAMME D'APPLICATION.

I. APPLICATION BASE DE DONNEES.

Une application base de données (ABD) est un système logiciel particulier comportant:

-- une description modulaire de la spécification et de la réalisation à la fois des informations et des actions

-- une gestion intégrée des données et des programmes

-- une utilisation adaptée à plusieurs besoins et à différents profils de programmation

-- une exploitation interactive.

En introduction de la troisième partie, nous avons déjà indiqué que les ABD nous semblaient correspondre au type d'application informatique que les entreprises développent au niveau de ce que l'on appelle parfois un département. En effet, les départements exploitent généralement de très grosses applications: production, commerciale,... Les outils logiciels d'aide à la construction d'ABD sont les SGBD classiques pour la gestion des données et les langages de programmation: COBOL, Pl/1, etc.. pour les traitements.

Un département est constitué de services. Il est bien connu (DUF79) que l'organisation du traitement de l'information dans les départements dépend étroitement de celle des services eux-mêmes. Nous n'avons pas l'intention d'aborder ici le problème de l'organisation du traitement de l'information dans l'entreprise. Nous voulons simplement faire remarquer que les données et les procédures de traitement exploitées par l'ensemble des utilisateurs sont (ou ne sont pas) partagées avec d'autres utilisateurs et que ces questions complexes relèvent essentiellement de l'analyse du traitement de l'information de l'entreprise. D'un point de vue informatique, une ABD regroupe toutes les constructions nécessaires à l'exploitation des données et des

traitements, partagées ou non, conformément à la réglementation et aux besoins d'utilisation. Il peut arriver que des groupes d'utilisateurs, appartenant à différents services d'un département, gèrent leurs propres applications en concomitance avec une aplication de référence. Le problème est alors de concevoir une organisation viable.

Il nous semble qu'il faut prendre en compte trois aspects essentiels:

-- les mécanismes de partage d'objets et de textes: machines et types

-- la protection et l'adaptabilité du langage en développant des mécanismes de masquage sélectif pour accéder aux objets et aux textes partagés

-- l'évolutivité.

Certains de ces aspects, comme le partage d'objets, ne sont pas tout à fait nouveaux (ABR74). Pour être capable de maîtriser l'ensemble des problèmes posés, nous nous appuyons sur une organisation hiérarchisée. Cette organisation ne représente pas obligatoirement l'organisation physique d'un département.

Dans une ABD, il existe un niveau de construction qui regroupe toutes les informations et les actions concernant directement ou indirectement l'ensemble des utilisateurs de l'application développée dans le département; nous l'appelons application de référence . Cette application utilise deux autres applications particuliéres:

-- l' application système qui définit toutes les machines et tous les types système-dépendants: entrée-sortie console, gestion de fichiers, etc...

-- et l' application généricité qui fournit les constructions de

textes paramétrés. C'est ensuite à partir de l'application référence que sont développées progressivement les applications bases de données manipulées par les utilisateurs. Nous les appelons applications sectorielles .

II. SCHEMAS DE L'ORGANISATION D'A.B.D.

Voir schémas ci après

III. APPLICATION REFERENCE.

Dans des applications informatiques intégrées telles que nous les envisageons, les utilisateurs partagent souvent un fond commun de données, de traitements opérant sur ces données et des utilitaires. C'est en ce sens que nous parlons d'application référence. Cette application importante sera construite progressivement à partir de l'expression de nouveaux besoins "globaux" ou encore de l'analyse de certains besoins sectoriels susceptibles d'intéresser d'autres utilisateurs. Elle sera constituée de la spécification de l'implantation du système de base de données référence (SBDR), des types données, des capsules associées, de machines de manipulation de données, de machines utilitaires, de machines de contrôle et d'un catalogue de connexion. Toutes les machines et tous les types système-dépendants utiles à la construction de l'ABD seront définis dans l'application système.

Une application référence est formée de:

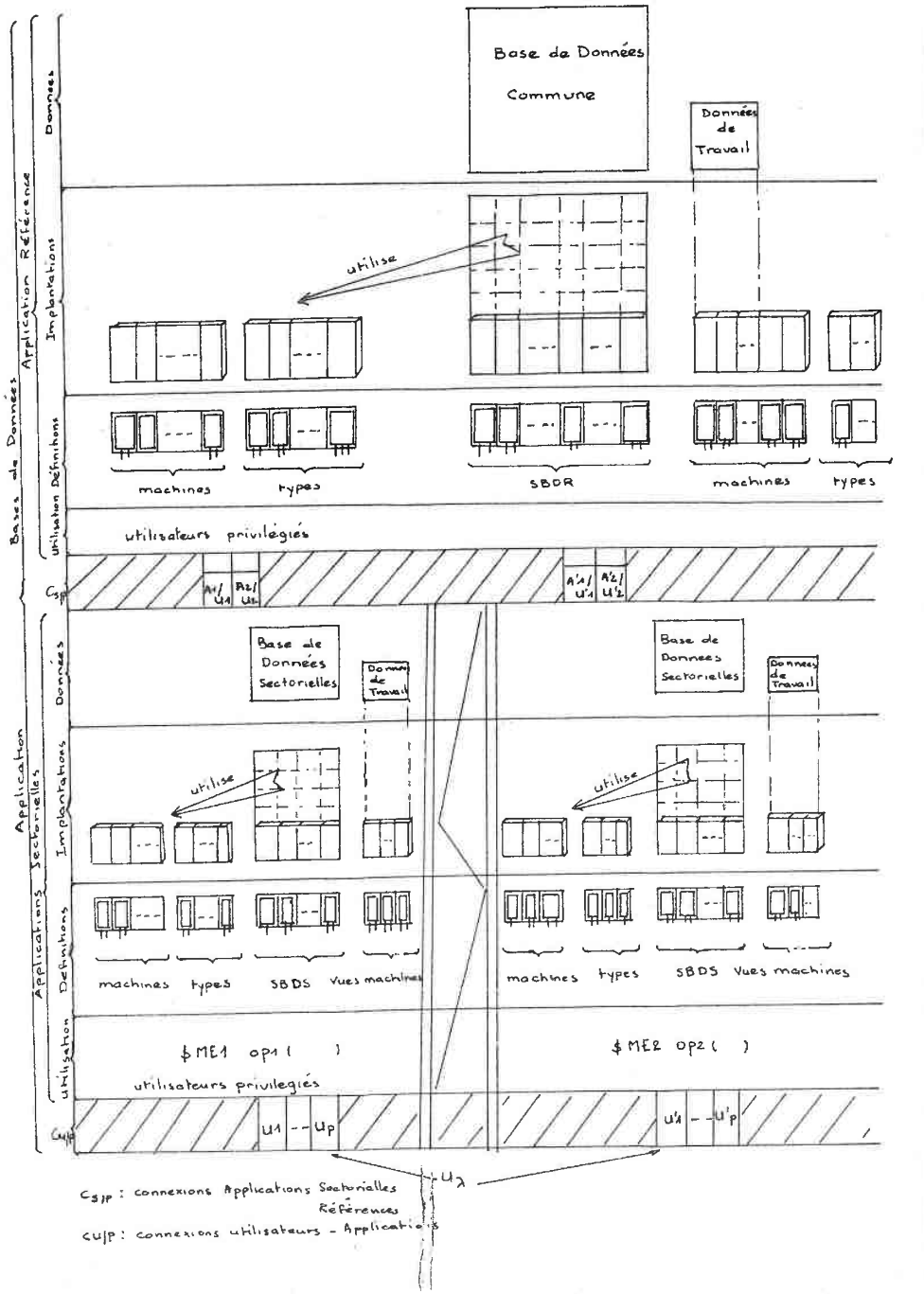
-- un ensemble de bibliothèques d'unités de spécification et d'implantation sous la forme de textes source et de textes compilés,

-- un catalogue de directives de connexion comportant les renseignements relatifs à la gestion des machines bases de données (noms de machines et modules de base de données, noms de machines et modules de conversion, date de connexion, etc...),

-- une base de données (commune) .

les utilisateurs autorisés à travailler sur l'application référence peuvent interroger le documentaliste qui gère les bibliothèques de textes source. Les autres bibliothèques sont exploitées à la fois par le compilateur et le connecteur qui construit des programmes d'application exécutables. Pour des raisons évidentes de protection mais aussi d'adaptation de l'application référence aux problèmes et aux profils spécifiques des utilisateurs, il est indispensable de fournir des mécanismes qui permettent à la fois de masquer sélectivement certaines spécifications et de définir d'autres unités d'intérêt sectoriel.

La notion d'application sectorielle que nous présentons maintenant a été introduite pour répondre à ce type de problème.



IV. APPLICATIONS SECTORIELLES.

Le but visé est finalement de construire des "contextes d'utilisation" qui répondent aux besoins des différentes catégories d'utilisateurs d'une ABD (adéquation des spécifications aux problèmes posés, facilité d'utilisation) et qui supportent aussi les protections prévues par l'administrateur.

En ce sens, une application sectorielle recouvre naturellement plusieurs types de construction. La première, bien connue du domaine des bases de données relationnelles, est la notion de "vue". Rappelons qu'en ce qui nous concerne, une vue est une abstraction d'une machine de base de données. Une aplication sectorielle sera donc formée d'un ensemble de spécifications de machines particulières. En effet, ces machines ('vues') définissent un masquage sélectif d'autres machines appartenant à d'autres applications (références ou sectorielles). Ce que l'on appelle dans les bases de données le "mapping" de la vue est décrit ici par des machines de contrôle appropriées. La manipulation de bases de données amène aussi certains utilisateurs à vouloir gérer des copies instantanées (partielles) de la base, ce que les spécialistes des bases de données appellent des "snapshots". Pour que les applications sectorielles supportent ces constructions, il faut pouvoir définir, dans ce contexte d'utilisation, de nouvelles unités machines et modules et parfois même de nouveaux types de données. Nous jugeons important de pouvoir prolonger cette possibilité en permettant de construire des "sous-applications" d'une application référence, appelées applications sectorielles.

Pour préciser cette notion d'application sectorielle, donnons

quelques définitions. Soit $\mathcal{A}$ l'ensemble des applications (applications système, référence ou sectorielles) d'une ABD et notons RA la relation "est fils de" définie sur l'arbre ordonné sous-jacent à l'ABD. L'application système, notée A_{sys} est une application particulière de $\mathcal{A}$ telle que $RA(A_{sys})=\emptyset$. L'application référence notée A_{ref} est aussi une aplication particulière de $\mathcal{A}$ telle que $A_{ref} \neq A_{sys}$ et A_{ref} RA A_{sys}. Les applications sectorielles a de $\mathcal{A}$ sont des applications telles que $a \neq A_{ref}$ et $a \neq A_{sys}$. On notera A_{sec} l'ensemble des applications sectorielles de $\mathcal{A}$.

Elles sont caractérisées par:

pour tout a de A_{sec}: $RA(a) \neq \emptyset$ et si a RA A_{sys} alors il existe i>1 tel que $a\ RA^{i-1}\ A_{ref}$. Considérons une application A de $\mathcal{A}$ et notons TY,MA,MO,CA les ensembles respectifs de types, de machines, de modules et de capsules d'une ABD donnée. Une unité U (type, machine,module ou capsule) est dite intérieure à l'application A si elle est déclarée dans A. On note TY(A), MA(A), MO(A) et CA(A) les ensembles respectifs de types, machines, modules et capsules intérieures à l'application A. Notons $RT_y \subset CA(A) \times TY(A)$ la relation capsule réalise type et $RM_a \subset MO(A) \times MA(A)$ la relation module réalise machine. Lorsqu'une opération O est déclarée dans une unité U (type, machine, module ou capsule), on notera cette relation U D O.

Ainsi, une occurrence d'utilisation d'une opération O d'un type ou d'une machine est une opération légale (dans A) s'il existe U intérieure à A telle que U D O.

Un type (resp. une machine) U est dit déterminé dans A si U est intérieur à A et s'il existe c dans CA(A) tel que c RT_y U (resp. il existe mo dans MO(A) tel que mo RM_a U).

Un type (resp. une machine) U sera dit importé dans une application A_i (de niveau i) s'il existe $0<j<i$ tel que $A_i\ RA^{j-1}\ A_j$ et U déterminé dans A_j et pour tout O : U D O dans A_j => U D O dans A_i.

L'ensemble des unités types et machines intérieures à une application A définit le contexte d'utilisation de A.

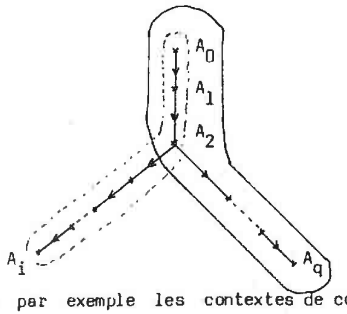
Maintenant, certaines machines du contexte d'utilisation de A ne sont pas déterminées dans A_i, ce sont les machines qui réalisent l'abstraction sélective de machines du contexte d'utilisation "ascendant". Pour être exploitées dans un programme d'application, elles requièrent l'utilisation de machines déterminées dans des contextes d'utilisation "ascendants". Soit A_i une application de niveau i de $\mathcal{A}$, nous appellerons contexte de connexion de A_i, noté C_{Ai} l'ensemble des applications de $\mathcal{A}$ telles que:

$$C_{Ai} = \bigcup_{0 \leq q \leq i} RA^q(A_i)$$

On constate donc qu'une application sectorielle, dans le cas le plus général représente à la fois des abstractions sélectives et des enrichissements d'"ascendants" (référence ou sectorielle).

La construction progressive des applications sectorielles d'une ABD devrait normalement conduire le concepteur d'une application à "réutiliser" des spécifications intérieures à d'autres applications du contexte de connexion. Pour permettre cela, nous proposons un mécanisme de partage d'objets (machines bases de données) et de textes (machines et types) entre les applications du contexte de connexion de l'application en cours de construction. Dans ce contexte là, toutes les unités types, machines, capsules et modules doivent être distinctes et en nombre limité. Le partage de textes (types et machines) est réalisé soit par copie de textes ce que nous appelons importation soit par

abstraction sélective de machine: il faut alors définir des machines de contrôle appropriées. Une importation de texte ne peut se faire qu'à partir de l'application où l'unité est effectivement déterminée. Maintenant, il peut aussi arriver que dans deux contextes de connexion distincts d'une même ABD, deux unités différentes possèdent le même nom:



Considérons par exemple les contextes de connexion respectifs de $A_i(A_0,.,A_2,.,A_i)$ et $A_q(A_0,.,A_2,.,A_q)$. Ces deux contextes de connexion distincts ont cependant une partie commune (A_0,A_1,A_2). Alors, sous réserve que tous les identificateurs d'unités, de chacun de ces contextes soient différents, il est possible que des applications qui n'appartiennent pas à la partie commune utilisent les mêmes identificateurs pour désigner des unités différentes. En fait, le contexte de connexion d'une application d'ABD détermine le domaine de validité des identificateurs d'unités intérieures à chacune des applications du contexte. Cette caractérisation complète l'étude des conditions de validité des identificateurs et des règles de visibilité que nous avons développée pour la définition des unités de spécification et d'implantation d'une application d'ABD.

D'autres mécanismes tels que les sous-types et les types dérivés d'ADA (ADA80) sont envisageables. Ils justifieraient à eux seuls une autre étude importante.

En résumé, on remarquera que nous avons essayé de proposer des concepts et des mécanismes qui permettent d'appliquer, ici aussi, les principes généraux de construction de logiciels: la décomposition modulaire et les raffinements successifs. La spécificité de l'environnement base de données nous a amené à définir de nouveaux concepts: les applications ainsi que des mécanismes de partage d'objets et de textes que nous allons préciser un peu plus maintenant.

V. CONTEXTE GLOBAL D'UNE BASE DE DONNEES D'UNE APPLICATION D'ABD.

Nous avons vu qu'une machine base de données intérieure à une application d'ABD définissait une abstraction d'une base de données. Seuls les utilisateurs autorisés à travailler dans le contexte d'utilisation de l'application ont le droit de manipuler la base commune ou sectorielle par des appels d'opérations de la machine base de données. Afin d'assurer une protection efficace de cette base mais aussi de faciliter l'adaptation du contexte d'utilisation aux différentes catégories d'utilisateurs, il convient de concevoir des applications sectorielles spécifiques à partir de l'application qui contient la machine base de données. En effet, les descriptions des applications sectorielles sont totalement séparées de la description de l'application qui définit la machine base de données. Les liaisons entre les applications seront établies à la connexion, par l'intermédiaire des machines de contrôle appropriées. On constate donc que le mécanisme d'application sectorielle permet de réaliser un

masquage complet, ou partiel, de la machine base de données à protéger. En outre, le contexte d'utilisation des applications sectorielles est totalement indépendant des choix d'implantation de la machine base de données. En résumé, le mécanisme d'application sectorielle réalise un haut niveau d'indépendance de la base de données vis à vis des programmes d'applications utilisateurs.

Nous appelons contexte global d'une base de données (caractérisée par une machine base de données intérieure à une application A référence ou sectorielle) l'ensemble des applications de l'ABD formé de A et des aplications sectorielles qui requièrent la machine base de données de A. Notons que le contexte global d'une base de données peut potentiellement englober toutes les applications sectorielles dérivées de A.

Le contexte global "effectif" est élaboré progressivement en définissant des abstractions sélectives et/ou des importations d'une machine base de données de A dans les applications sectorielles du contexte. Si ce contexte se réduit à la seule application A, cela signifie que c'est une base de données locale. Par contre, s'il comporte d'autres applications sectorielles fournissant des accés à la base de données de A, il est alors évident qu'une modification réalisée à partir d'une quelconque application du contexte global de A sera "propagée" à toutes les applications qui fournissent aussi des accès à cette base. En résumé, on constate que le mécanisme d'application (sectorielle) permet une description "très fine" du partage et de la protection des bases de données pour les applications intérieures d'une ABD.

VI. APPLICATION CONNEXION.

Nous avons déjà présenté le principe de la connexion des unités intérieures à une application. Pour qu'un programme d'une application soit exécutable, il convient de réaliser toutes les connexions appropriées des unités du contexte de connexion de l'application. Plus précisément, il faut d'abord établir les connexions à l'intérieur de chacune des applications du contexte de connexion puis réaliser les "connexions" des applications entre elles. La connexion des applications nous a amené à définir un mécanisme particulier utilisant des machines de contrôle intérieures aux applications. Ce mécanisme défini dans (HEN80) consiste à décrire des modifications (statiques) de liaisons entre unités du contexte de connexion. Une application connexion est une unité de description modulaire de directives de connexion définies pour les unités du contexte de connexion d'une application d'ABD. Les différents choix de connexion sont particulièrement importants aussi bien pour l'exploitation que pour la protection des bases de données de l'application. Il semble souhaitable que seul le responsable de l'ABD ait le droit de définir de telles applications. Cela veut aussi dire qu'un utilisateur ne pourra travailler effectivement sur une application que si l'administrateur lui a, au préalable, défini une application connexion. Pour faciliter le travail des utilisateurs, ces applications connexion pourront être cataloguées à la demande dans le système. Dans ces conditions, pour qu'un utilisateur ait le droit de "lancer" des exécutions sur une application, il devra fournir le nom d'une application connexion.

Pour une ABD, le nom (unique) d'application connexion sert en

quelque sorte de numéro de compte utilisateur. Finalement, la demande d'exécution sera acceptée si le connecteur parvient à authentifier ce nom à partir de renseignements connus du système. Maintenant, pour décrire, mettre à jour et cataloguer les différentes applications connexion des utilisateurs d'une ABD, nous avons dû développer un langage spécifique. Nous proposons ici un langage extrèmement simple qui ne comporte que des opérations élémentaires: les opérations de connexion d'unités, de modification des liaisons (HEN80) et des opérations relatives au cataloguage dans une bibliothèque appropriée: cataloguer une application de nom donné, remplacer le texte d'une application cataloguée par celui d'une autre application, lister une ou toutes les applications cataloguées, etc.. Une application connexion décrit dans un ordre quelconque la liste des directives de connexion d'une application d'ABD. Notons enfin, qu'il nous semble aussi souhaitable que certaines catégories d'utilisateurs, aient la possibilité de décrire des connexions relatives à des définitions intérieures à l'application sur laquelle ils sont amenés à travailler. Ces connexions ne pourront être que "temporaires", c'est-à-dire valides que pour une demande d'exécution. En introduisant cette facilité, nous pensons précisément aux utilisateurs qui formulent des requêtes "ponctuelles" de manipulation des bases de données d'une ABD. Pour eux, il serait fastidieux de devoir demander systématiquement à l'administrateur de modifier leur application connexion, alors que la modification à introduire est locale, temporaire et de plus, simple à décrire.

VII. EXECUTION D'UN PROGRAMME D'APPLICATION.

Nous avons exposé en III. le principe de l'exécution d'un programme d'application. Si une application requiert d'autres applications, le procédé d'exécution reste le même, mais dans le contexte de connexion de l'application concernée. En bref, nous avons étendu l'ensemble des connexions relatives à une application au contexte de connexion de l'application, lorsque celle-ci requiert d'autres applications pour pouvoir exécuter un programme. Dans le contexte de connexion d'une application d'ABD, la description des connexions, du fait de la définition des modifications de liaison, ne fait plus explicitement référence aux applications du contexte.

CHAPITRE XII. LE SYSTEME INTERACTIF D'AIDE A LA CONSTRUCTION, A LA MODIFICATION ET A L'UTILISATION DES A.B.D.

I. SCHEMA DE L'AUTOMATE D'ETATS DES COMMANDES.

II. REMARQUES.

Pour pouvoir véritablement entreprendre le développement interactif et progressif de systèmes logiciels aussi importantts que les ABD, il importe que les constructeurs aient à leur disposition des outils logiciels appropriés. Nous avons ainsi conçu un interface interactif des processeurs intégrés au système TYP: système de compilation, système de généricité, système de conexion, éditeur de textes, documentalistes divers permettant de fournir de manière interactive les commandes utiles. Pour faciliter encore le travail des constructeurs, c'est-à-dire leur éviter de penser à la fois à l'enchaînement des commandes et aux problèmes posés par la construction ou la modification des applications systèmes, référence, sectorielle, connexion ou même utilisation, il nous a paru souhaitable de programmer le pilotage et une documentation à la demande des commandes de l'interface. La réalisation de cet interface s'appuie sur un automate d'états décrit ci après.

I. SCHEMA DE L'AUTOMATE D'ETATS DES COMMANDES.

A tous les stades du travail, l'utilisateur a la possibilité de demander des explications sur les commandes à fournir au système, en tapant un caractère "?". D'autre part, pour remonter d'un niveau, il suffit de taper un retour chariot . Une commande " end " permet de sortir du connecteur.

II. REMARQUES.

Actuellement, pour soumettre des travaux à un système d'exploitation: compilation, éditions de liens, etc... les programmeurs doivent écrire des directives ou des instructions (cartes de commande) dans un langage (de commande) proche de la machine. Pour éviter ce type de programmation, nous avons repris l'idée qui consiste à définir un interface de programmation "à la carte" pour non spécialistes. Cette technique permet:

-- de cacher à l'utilisateur les cartes de commande paramétrées qui sont nécessaires à l'appel des processeurs intégrés au système TYP: système de compilation,etc...

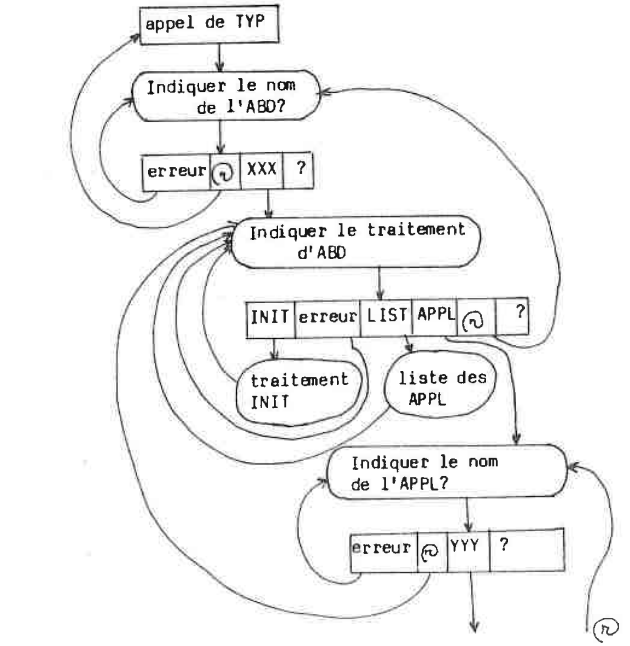
-- de piloter (ou de guider) l'utilisateur dans le choix des options à fournir au système interactif selon qu'il veut construire, modifier, lister des unités ou utiliser une application d'ADD. Pour préciser ses choix, l'utilisateur est conduit "pas à pas". Il lui suffit de répondre par des mots-clés aux questions posées par le système interactif. Si à un moment donné, l'utilisateur veut obtenir des explications sur les

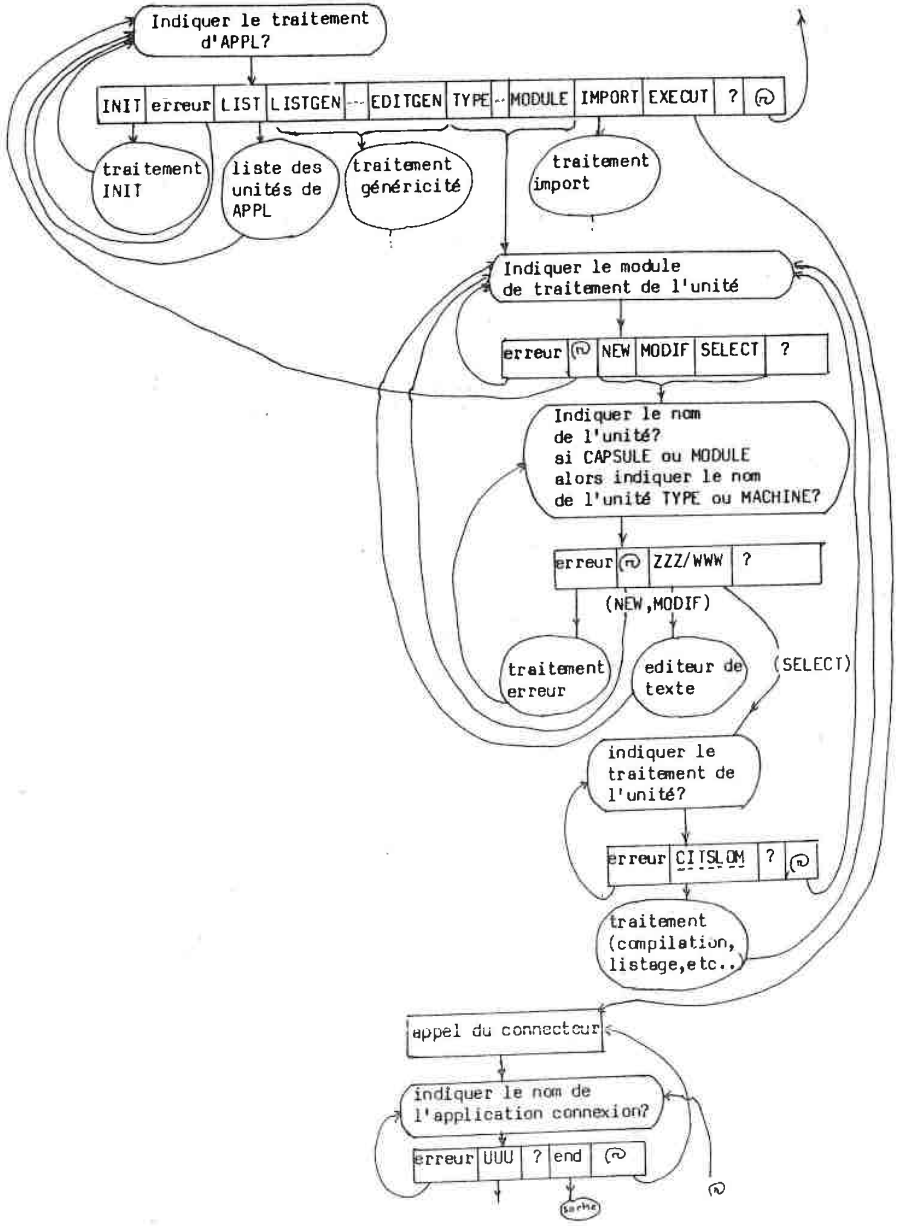
possibilités de réponse à une question, il lui suffit de frapper un caractère "?". Si la réponse fournie est erronée, le système repose la même question. Pour "remonter" dans l'automate d'états, il suffit de frapper un retour chariot.

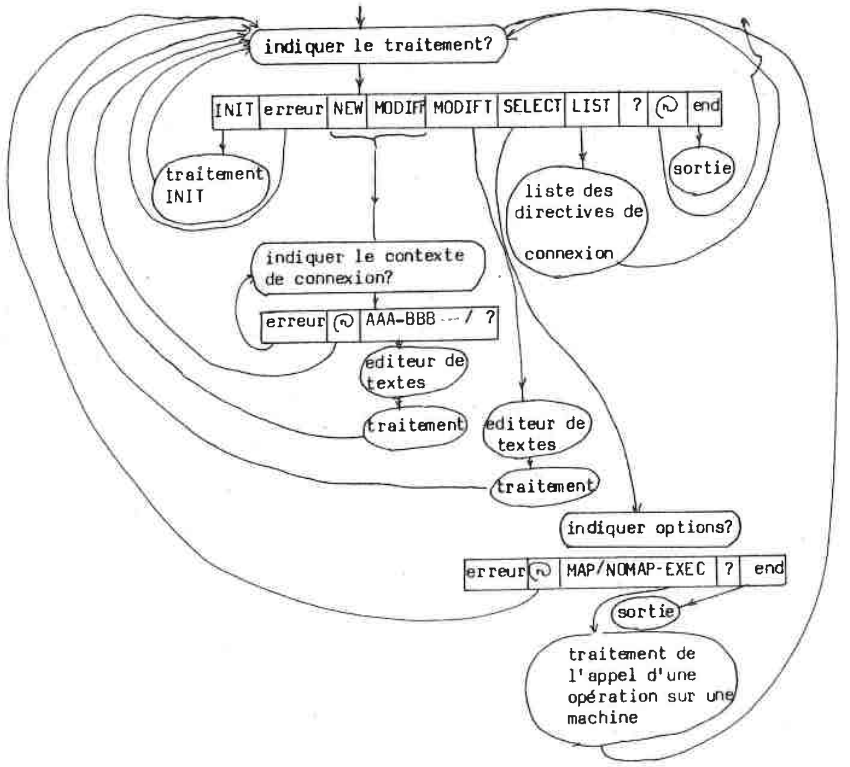
Sur le schéma précédent, on constate que l'utilisateur peut très facilement faire appel aux documentalistes d'une ABD. Ainsi, selon le niveau d'utilisation de l'option "LIST", il est possible d'obtenir: la liste des noms d'applications d'ABD, la liste des noms des types, de capsules, de machines et de modules d'une application particulière, la liste des directives de connexion pour une application connexion. Pour les unités génériques, il faut utiliser l'option "LISTGEN". Pour lister complètement le texte source (commentaires compris) d'une unité donnée d'une application d'ABD, il suffit d'utiliser les options "SELECT" puis "LT" (listage terminal).

Notons encore que pour construire ou modifier des unités, le programmeur dispose de toutes les facilités d'un éditeur de textes classique, en l'occurrence celui de SIRIS 8.

Schéma de l'automate d'états des commandes.




Indiquer le traitement d'APPL?
INIT
erreur
LIST
LISTGEN
EDITGEN
TYPE
MODULE
IMPORT
EXECUT
?
traitement INIT
liste des unités de APPL
traitement généricité
traitement import
Indiquer le module de traitement de l'unité
erreur
NEW
MODIF
SELECT
?
Indiquer le nom de l'unité?
si CAPSULE ou MODULE
alors indiquer le nom
de l'unité TYPE ou MACHINE?
erreur
ZZZ/WWW
?
(NEW,MODIF)
(SELECT)
traitement erreur
editeur de texte
indiquer le traitement de l'unité?
erreur
CITSLOM
?
traitement (compilation, listage,etc..)
appel du connecteur
indiquer le nom de l'application connexion?
erreur
UUU
?
end



Légende:

commande fournie par l'utilisateur

message émis par le système

traitement effectué par le système TYP ou l'utilisateur (sous l'éditeur de textes)

CHAPITRE XIII. REMARQUES SUR LA REALISATION DU SYSTEME EXPERIMENTAL TYP.

Le système intégré et interactif de programmation modulaire TYP est opérationnel sur le système SIRIS 8 V10 de l'IRIS 80 de l'Institut Universitaire de Calcul Automatique de Lorraine. Il comporte:

-- un système de compilation séparée réalisé par Régis MINOT (MIN79),

-- un système de généricité réalisé par Karol PROCH (PRO79),

-- des systèmes documentalistes et gestionnaires (MIN79) (PRO79),

-- un connecteur, un éditeur de textes et un système d'interaction (HEN80).

Le lecteur intéressé par des informations détaillées sur la réalisation du système TYP peut se rapporter aux rapports et thèses de troisième cycle r éférencés ci-dessus.

La première version du système réalisée par Régis MINOT utilise un éditeur de liens statique. Il nous est apparu à l'analyse de cette version que les deux étapes de l'édition de liens statique: résolution et définition des références externes, puis édition et translation des unités d'implantation compilées séparément pouvaient être évitées. La première étape peut être réalisée dès la compilation; quant à la seconde, il suffit d'adresser les externes par l'intermédiaire d'un descripteur. Ce descripteur est une table avec autant d'entrées que d'externes (noms d'opérations) connus du système. Enfin, si l'on dispose d'un adressage basé, ce qui est notre cas, le code généré pour des modules et des capsules devient indépendant de l'adresse d'implantation en mémoire. De cette façon, le changement d'un module (ou d'une capsule) est ramené à un simple transfert de code d'un support secondaire en mémoire centrale. C'est à partir de ces idées

qu'a été réalisé le connecteur du système TYP (HEN80).

CONCLUSION .

bilan et perspectives :

Notre objectif dans ce travail est de fournir des concepts et des outils logiciels intégrés qui fournissent un continuum d'expression depuis la spécification des problèmes (orientés bases de données) jusqu'à l'exploitation et la maintenance des programmes.

Pour ce faire, nous utilisons un principe fondamental: l'abstraction et un mécanisme (ou une technique) puissant(e): la modularité avec la notion de type abstrait de données.

Pour discuter de la réalisation de cet objectif, nous commençons par regarder ce qu'apporte ou non le système VEGA, puis ensuite le système TYP avant de conclure.

Le système VEGA :

L'originalité de VEGA est de fournir des aides et un environnement adaptés à la spécification et à la validation de problèmes orientés bases de données. Le langage formel de VEGA supporte deux techniques de spécification basées l'une sur les modèles abstraits, l'autre sur les constructeurs. Un environnement de spécifications initiales fournit une

base de spécifications paramétrées demodèles abstraits: produits cartésiens labellés, ensembles, séquences, etc... particulièrement adaptés à la modélisation des univers de problèmes orientés bases de données. En outre, les mécanismes d'aide à la spécification de VEGA: coercition et surcharge d'opérateurs, enrichissement, restriction permettent d'effectuer une spécification compréhensible et progressive (structurée). Pour pouvoir tester si les spécifications répondaient aux intentions initiales des utilisateurs, il nous a semblé intéressant de pouvoir utiliser un langage d'interface pour produire facilement et rapidement des maquettes de bases de données symboliques. Sur ce point, la réalisationde la maquette de système expert d'aide à l'information touristique du projet MONT-LOZERE a constitué un chantier expérimental particulièrement fructueux.

Maintenant, il reste que l'approche de spécification que nous avons prise: les spécifications avec erreurs, ne permet pas dans l'état actuel des recherches d'en vérifier la correction. C'est un problème ouvert qui fait l'objet de recherches théoriques intensives.

Le système TYP :

Les choix de conception qui ont été faits dans TYP: langage ADA, généricité, système de fabrication de programmes montrent que nous avons eu le souci de conserver au niveau de l'implantation mais aussi de l'exploitation et de la maintenance des systèmes d'application bases de données les mêmes notions d'abstraction et de modularité. En particulier, le fait de pouvoir remplacer une unité de réalisation par une autre entre deux exécutions de programmes ou encore de fabriquer

dynamiquement des programmes sont autant d'aspects originaux et particulièrement adaptés à l'approche bases de données.

Le domaine que nous avons contribué à explorer est très vaste. On peut penser que le développement de systèmes intégrés d'applications logicielles plus particulièrement de systèmes experts devrait mobiliser encore bien des efforts. Nous commençons d'ores et déjà à nous y intéresser.

ANNEXES

ANNEXE 1

TYP

Environnement de programmation

```
!

 typ
STATION?



       T Y P   A   V O T R E   S E R V I C E

***** R E A L I S A T I O N ***************************************
*                                                                 *
*******************************************************************
*                 *             *                     *           *
*  COMPILATEUR ATM * GENERICITE  * OBJETS CONSERVES * CONNECTEUR  *
*                 *             *                     *           *
*******************************************************************
*                 *             *                     *           *
* HENRY P.    9-79 * PROCH K.    * CHABRIER J.         * HENRY P.    *
* MINOT R.    3-79 *       1-80 *           10-79     *       9-79  *
* TOUNSI N.   1-80 *             *                     *           *
*                 *             *                     *           *
*******************************************************************

SI RIEN N'EST SPECIFIE
A TOUTES LES QUESTIONS REPONDEZ PAR "OUI" OU PAR "NON"
REPONDEZ "?" POUR OBTENIR DES EXPLICATIONS
UN RETOUR CHARIOT PERMET DE REMONTER D'UN NIVEAU.

INDIQUEZ L'ENVIRONNEMENT(DE 0 A 3 CAR.) : jc


INDIQUEZ LE TRAITEMENT PAR
"TYPE","MACHINE","MODULE","CAPSULE",
"LIST","INIT","EXECUT",
"UNITGEN","GENERER","LISTGEN","EDITGEN" :list




*** LISTE DES TYPES ***

TYPE              CAPSULES . . . . . .
INTEGER
REAL
CHAR
BOOLEAN
TEXT              MONTEXT
FSEQ              FSEQ
DATE              DATE1     DATE2
LIVRE             LIVRE
CHERCH            CHERCH
PRET              PRET
DEMANDE           DEMANDE
FIND              FIND
```

*** LISTE DES MACHINES ***

MACHINE	MODULES					
READER	LIRE					
WRITER	ECRIRE					
CONV	CONV					
MAIN	ESSFIND	ESSFIND1	ESSFSEQ	ESSFIND2	BIB2	ESS
TIME	TIME					
LIVRES	LIVRES	LIVRES1				
CHERCHS	CHERCHS	CHERCHS1				
PRETS	PRETS					
DEMANDES	DEMANDES					
BIB	BIB					
MAIN1	LOG					

. Types standards

f/jc:t:01/
```
00001*0,$1
00001 TYPE INTEGER;
00002   ### *** STANDARD *** ###
00003 END
EOF HIT
*
```

f/jc:t:02/
```
00001*0,$1
00001 TYPE REAL;
00002 END
EOF HIT
*
```

f/jc:t:03/
```
00001*0,$1
00001 TYPE CHAR;
00002     ### STANDARD  ###
00003 END
EOF HIT
*
```

f/jc:t:04/
```
00001*0,$1
00001 TYPE BOOLEAN;
00002      ### STANDARD ###
00003 END
```

. Types prédéfinis

f/jc:t:05/
```
00001*0,$1
00001 TYPE TEXT;
00002           # CHAINES DE CARACTERES DE LONGUEUR VARIABLE
00003           # LA LONGUEUR EST A DETERMINER PAR LA CAPSULE
00004           # LE COMPILATEUR NE CONNAIT DES CONSTANTES DE
00005           # TYPE TEXTE QUE SI LEUR LONGUEUR EST INFERIEURE
00006           # A 63
00007 MODIF CONCHAR(CHAR);
00008           #CONCATENATION A DROITE D'UN CARACTERE
00009           # EN CAS D'ANOMALIE : TRONCATURE
00010 FUNCTION GETCHAR(INTEGER) RETURNS CHAR;
00011           # RENVOIE LE CARACTERE DE RANG I DANS LA CHAINE
00012           # EN CAS D'ERREUR RENVOIE LE CARACTERE '@'
00013 MODIF PUTCHAR(INTEGER,CHAR);
00014           # AFFECTE LE CARACTERE C DANS LA CHAINE
00015           # A L'ENDROIT I
00016 FUNCTION LENGTH RETURNS INTEGER;
00017           # RENVOIE LE NOMBRE DE CARACTERES CONTENUS DANS
00018           # LA CHAINE
00019 MODIF CONTEXT(TEXT);
00020           # CONCATENATION A DROITE D'UNE CHAINE DE CARACTERES
00021           # EN CAS D'ANOMALIE TRONCATURE
00022 FUNCTION SUBSTR(INTEGER,INTEGER) RETURNS TEXT;
00023           # RENVOIE LA SOUS CHAINE DELIMITEE PAR I ET J
00024           # EN CAS D'ANOMALIE LA CHAINE RENVOYEE EST
00025           # SOIT NULLE (I<0 OU J<I)
00026           # SOIT DE LONGUEUR L TELLE QUE 0<=L<=J-I
00027 FUNCTION EQ(TEXT) RETURNS BOOLEAN;
00028           # @T1 EQ(T2) EST VRAI SI T1=T2 , FAUX SINON
00029 FUNCTION GT(TEXT) RETURNS BOOLEAN;
00030           # @T1 GT(T2) EST VRAI SI T1>T2 , FAUX SINON
00031 MODIF SUBG(TEXT,INTEGER,INTEGER);
00032           # @T1 SUBG(T2,I,J)
00033           # SI LENGTH(T1) <= J ALORS AJUSTER LA LONGUEUR A J
00034           # EN COMPLETANT AVEC DES BLANCS ;
00035           # AFFECTER DANS T1 A PARTIR DU IEME CARACTERE LA
00036           # CHAINE T2 CADREE A GAUCHE ET EVENTUELLEMENT
00037           # TRONQUEE A DROITE
00038 MODIF SUBD(TEXT,INTEGER,INTEGER);
00039           # IDENTIQUE A SUBG SAUF
00040           # CADRAGE A DROITE ET TRONCATURE A GAUCHE
00041 MODIF AJUST(INTEGER);
00042           # SERT A AJUSTER LA LONGUEUR D'UNE CHAINE DE CARACTERES
00043           # SOIT EN TRONQUANT
00044           # SOIT EN COMPLETANT AVEC DES BLANCS
00045           # SI I<0 ALORS I=0
00046 FUNCTION REPEAT(TEXT,INTEGER) RETURNS TEXT;
00047           # RENVOIE UNE CHAINE DE CARACTERES OBTENUE EN
00048           # EN CONCATENANT INTEGER FOIS LA CHAINE TEXT
00049 FUNCTION INDEX(TEXT) RETURNS INTEGER;
00050           # RENVOIE LE RANG DE TEXT DANS LA CHAINE
00051           # 0 SI PAS TROUVE
00052 END;
EOF HIT
*
```

```
 f/jc:t:06/
00001*0,$1
00001   TYPE FSEQ;
00002       # FICHIER SEQUENTIEL #
00003     BUILD CRE(TEXT,INTEGER,INTEGER,TEXT);
00004       # CREATION D UN FICHIER SEQUENTIEL #
00005       # TEXT = ETIQUETTE OPERATIONNELLE #
00006       # INTEGER = LONGUEUR DE L ENREGISTREMENT #
00007       # INTEGER = LONGUEUR DU BLOC #
00008       # TEXT = NOM DU FICHIER #
00009     MODIF OPENG;
00010       # OUVERTURE DU FICHIER EN LECTURE #
00011     MODIF OPENP;
00012       # OUVERTURE DU FICHIER EN ECRITURE #
00013     MODIF OPENU;
00014       # OUVERTURE DU FICHIER EN MISE A JOUR #
00015     MODIF CLBEGIN;
00016       # FERMETURE DU FICHIER ET POSITIONNEMENT AU DEBUT #
00017     MODIF CLEND;
00018       # FERMETURE DU FICHIER ET POSITIONNEMENT EN FIN #
00019     CONSULT GET -> TEXT,BOOLEAN;
00020       # LECTURE D UN ENREGISTREMENT DE TYPE TEXT #
00021       # BOOLEAN = FAUX SI FIN DE FICHIER #
00022     MODIF PUT(TEXT);
00023       # ECRITURE D UN ENREGISTREMENT DE TYPE TEXT #
00024     MODIF REPLACE(TEXT);
00025       # REMPLACEMENT DU DERNIER ENREGISTREMENT LU #
00026     MODIF DELETE;
00027       # EFFACEMENT DU DERNIER ENREGISTREMENT LU #
00028     MODIF KILL;
00029       # DESTRUCTION DU FICHIER #
00030   END
EOF HIT
*
```



```
 f/jc:t:0c/
00001*0,$1
00001 TYPE FIND;
00002       # FICHIER SEQUENTIEL INDEXE #
00003   BUILD CRE(TEXT,INTEGER,INTEGER,TEXT,INTEGER,INTEGER);
00004       # CREATION D UN FICHIER SEQUENTIEL INDEXE #
00005       # TEXT = ETIQUETTE OPERATIONNELLE #
00006       # INTEGER = LONGUEUR DE L ENREGISTREMENT #
00007       # INTEGER = LONGUEUR DU BLOC #
00008       # TEXT = NOM DU FICHIER #
00009       # INTEGER = LONGUEUR DE LA CLE #
00010       # INTEGER = POSITION DE LA CLE #
00011   MODIF OPENG;
00012       # OUVERTURE DU FICHIER EN LECTURE #
00013   MODIF OPENP;
00014       # OUVERTURE DU FICHIER EN ECRITURE #
00015   MODIF OPENU;
00016       # OUVERTURE DU FICHIER EN MISE A JOUR #
00017   MODIF CLBEGIN;
00018       # FERMETURE DU FICHIER ET POSITIONNEMENT AU DEBUT #
00019   MODIF CLEND;
00020       # FERMETURE DU FICHIER ET POSITIONNEMENT EN FIN #
00021   CONSULT GET -> TEXT,BOOLEAN;
00022       # LECTURE D UN ENREGISTREMENT DE TYPE TEXT #
00023       # BOOLEAN = FAUX SI FIN DE FICHIER #
00024   CONSULT GETN(TEXT) -> TEXT,BOOLEAN;
00025       # LECTURE D UN ENREGISTREMENT DE CLE DONNEE #
00026       # BOOLEAN = FAUX SI LA CLE N EXISTE PAS #
00027   MODIF PUT(TEXT) -> BOOLEAN;
00028       # ECRITURE D UN ENREGISTREMENT DE TYPE TEXT #
00029       # BOOLEAN = FAUX #SI LA CLE EST DECLASSEE #
00030   MODIF PUTN(TEXT) -> BOOLEAN;
00031       # ECRITURE D UN ENREGISTREMENT DE NOUVELLE CLE #
00032       # BOOLEAN = FAUX SI LA CLE EXISTE DEJA #
00033   MODIF DELETE;
00034       # EFFACEMENT DU DERNIER ENREGISTREMENT LU #
00035   MODIF KILL;
00036       # DESTRUCTION DU FICHIER #
00037 END
EOF HIT
*
```

. Types construits

```
 f/jc:t:07/
 00001*0,$1
 00001 TYPE DATE;
 00002   CONSULT OUT ->TEXT;
 00003       # SORTIR UNE DATE #
 00004   BUILD IN(TEXT);
 00005       # ENTRER UNE DATE #
 00006   FUNCTION GT(DATE) RETURNS BOOLEAN;
 00007       # @D1 GT(D2) -> B; #
 00008       # B = VRAI SI D1 EST POSTERIEURE A D2 #
 00009   FUNCTION JOUR RETURNS INTEGER;
 00010       # ACCES AU JOUR DE LA DATE #
 00011   FUNCTION MOIS RETURNS INTEGER;
 00012       # ACCES AU MOIS DE LA DATE #
 00013   FUNCTION ANNEE RETURNS INTEGER;
 00014       # ACCES A L ANNEE DE LA DATE #
 00015   BUILD CREDATJ;
 00016       # @D CREDATJ; D EST LA DATE DU JOUR #
 00017   BUILD CREDAT(INTEGER);
 00018       # @D CREDAT(20); D EST LA DATE DU JOUR #
 00019       # AUGMENTEE DE 20 JOURS #
 00020 END
 EOF HIT
 *
```

```
 f/jc:t:08/
00001*0,$1
00001 TYPE LIVRE;
00002   BUILD CREER(INTEGER,TEXT,TEXT,TEXT);
00003     # CREATION D UN LIVRE #
00004   FUNCTION CLE RETURNS INTEGER;
00005     # ACCES A LA CLE (NUMERO) DU LIVRE #
00006   FUNCTION SUJET RETURNS TEXT;
00007     # ACCES AU SUJET DU LIVRE #
00008   FUNCTION TITRE RETURNS TEXT;
00009     # ACCES AU TITRE DU LIVRE #
00010   FUNCTION AUTEUR RETURNS TEXT;
00011     # ACCES A L AUTEUR DU LIVRE #
00012   CONSULT OUT -> TEXT;
00013     # SORTIR UN LIVRE #
00014   BUILD IN(TEXT);
00015     # ENTRER UN LIVRE #
00016 END
EOF HIT
*
```



```
 f/jc:t:09/
00001*0,$1
00001 TYPE CHERCH;
00002   BUILD CREER(INTEGER,TEXT);
00003     # CREATION D UN CHERCHEUR #
00004   FUNCTION CLE RETURNS INTEGER;
00005     # ACCES A LA CLE (NUMERO) DU CHERCHEUR #
00006   FUNCTION NOM RETURNS TEXT;
00007     # ACCES AU NOM DU CHERCHEUR #
00008   CONSULT OUT -> TEXT;
00009     # SORTIR UN CHERCHEUR #
00010   BUILD IN(TEXT);
00011     # ENTRER UN CHERCHEUR #
00012   FUNCTION EQ(CHERCH) RETURNS BOOLEAN;
00013 END
 f/jc:t:0a/
00001*0,$1
00001 TYPE PRET;
00002   BUILD CREER(LIVRE,CHERCH,DATE,DATE);
00003     # CREATION D UN PRET #
00004   FUNCTION LIVRE RETURNS LIVRE;
00005     # ACCES AU LIVRE DU PRET #
00006   FUNCTION CHERCH RETURNS CHERCH;
00007     # ACCES AU CHERCHEUR DU PRET #
00008   FUNCTION DATEPRE RETURNS DATE;
00009     # ACCES A LA DATE DU PRET #
00010   FUNCTION DATEDU RETURNS DATE;
00011     # ACCES A LA DATE DE RESTITUTION DU PRET #
00012   CONSULT OUT ->TEXT;
00013     # SORTIR UN PRET #
00014   BUILD IN(TEXT);
00015     # ENTRER UN PRET #
00016   FUNCTION NUMERO RETURNS INTEGER;
00017     # ACCES A LA CLE DU LIVRE DU PRET #
00018 END
EOF HIT
*
```

```
 f/jc:t:0b/
00001*0,$1
00001 TYPE DEMANDE;
00002   BUILD CREER(LIVRE,CHERCH,DATE);
00003     # CREATION D UNE DEMANDE #
00004   FUNCTION LIVRE RETURNS LIVRE;
00005     # ACCES AU LIVRE DE LA DEMANDE #
00006   FUNCTION CHERCH RETURNS CHERCH;
00007     # ACCES AU CHERCHEUR DE LA DEMANDE #
00008   FUNCTION DATEDEM RETURNS DATE;
00009     # ACCES A LA DATE DE LA DEMANDE #
00010   CONSULT OUT -> TEXT;
00011     # SORTIR UNE DEMANDE #
00012   BUILD IN(TEXT);
00013     # ENTRER UNE DEMANDE #
00014   FUNCTION NUMERO RETURNS INTEGER;
00015     # ACCES A LA CLE DU LIVRE DE LA DEMANDE #
00016 END
EOF HIT
*
```

. capsules des types prédéfinis


```
 f/jc:t:05:montext/
00001*0,$1
00001 CAPSULE MONTEXT FOR TEXT;
00002     REP
00003       CPT:      INTEGER;
00004       CHAINE:   ARRAY(136) OF CHAR;
00005     END
00006
00007 MODIF CONCHAR(A:CHAR);
00008     BEGIN
00009       IF CPT>=136
00010       THEN RETURN;
00011       ENDIF;
00012     CPT:=CPT+1;
00013     CHAINE(CPT):=A;
00014     END
00015
00016 FUNCTION GETCHAR(I:INTEGER) RETURNS CHAR;
00017     BEGIN
00018       IF CPT<I
00019       THEN RETURN "@";
00020       ELSE RETURN CHAINE(I);
00021       ENDIF;
00022     END
00023
00024 MODIF PUTCHAR(I:INTEGER,C:CHAR);
00025     BEGIN
00026       IF ((I>CPT) OR (I<0))
00027       THEN RETURN;
00028       ELSE CHAINE(I):=C;
00029       ENDIF;
00030     END
00031
00032 FUNCTION LENGTH RETURNS INTEGER;
00033     BEGIN
00034       RETURN CPT;
00035     END
00036
00037 MODIF CONTEXT(T:TEXT);
00038     VAR
00039       C:CHAR;
00040       J,K:INTEGER;
00041     END
00042     BEGIN
00043       J:=1;K:=T.CPT;
00044       WHILE (CPT<136) AND (J<=K);
00045         CPT:=CPT+1;
00046         CHAINE(CPT):= T.CHAINE(J);
00047         J:=J+1;
00048       ENDWHILE;
00049     END
00050
```



```
00051 FUNCTION SUBSTR(I:INTEGER,J:INTEGER) RETURNS TEXT;
00052     VAR
00053       T:TEXT;
00054       C:CHAR;K:INTEGER;
00055     END
00056     BEGIN
00057     T:="";
00058     FOR K:=I BY 1 WHILE K<=J;
00059        C:=CHAINE(K);
00060       @ T CONCHAR(C);
00061     ENDFOR;
00062     RETURN T;
00063     END
00064
00065 FUNCTION EQ(T:TEXT) RETURNS BOOLEAN;
00066     VAR
00067       C:CHAR;K:INTEGER;
00068     END
00069     BEGIN
00070       IF CPT <> T.CPT
00071       THEN RETURN FALSE;
00072       ENDIF;
00073       FOR K:=1 BY 1 WHILE ((K<=CPT) AND (CHAINE(K) = T.CHAINE(K)));
00074       K:=K;
00075       ENDFOR;
00076       IF K=CPT+1
00077       THEN RETURN TRUE;
00078       ELSE RETURN FALSE;
00079       ENDIF;
00080     END
00081
00082 FUNCTION GT(T:TEXT) RETURNS BOOLEAN;
00083     VAR
00084       J,K:INTEGER;
00085     END
00086     BEGIN
00087       IF CPT<T.CPT
00088       THEN FOR K:=1 BY 1 WHILE ((K<=CPT) AND (CHAINE(K)<T.CHAINE(K)));
00089       K:=K;
00090       ENDFOR;
00091           IF K=CPT+1
00092           THEN FOR K:=K BY 1 WHILE ((K<T.CPT)AND(T.CHAINE(K)>" "));
00093           K:=K;
00094           ENDFOR;
00095             IF K=T.CPT+1 THEN RETURN TRUE; ENDIF;
00096         ELSE RETURN FALSE;
00097         ENDIF;
00098       ELSE FOR K:=1 BY 1 WHILE((K<=T.CPT) AND (CHAINE(K)<T.CHAINE(K)));
00099       K:=K;
00100       ENDFOR;
00101           IF K=T.CPT+1
00102            THEN FOR K:=K BY 1 WHILE ((K<CPT)AND(CHAINE(K)<" "));
00103           K:=K;
00104             ENDFOR;
00105                IF K=CPT+1 THEN RETURN TRUE; ENDIF;
00106            ELSE RETURN FALSE;
00107           ENDIF;
00108       ENDIF;
00109     END
00110
00111 MODIF SUBG(T:TEXT,I:INTEGER,J:INTEGER);
00112     VAR
00113       M,N,K:INTEGER;
00114     END
```

```
00116       N:=1;
00117       M:=T.CPT;
00118       FOR K:=I BY 1 WHILE ((K<=J) AND (K<=CPT) AND (N<=M));
00119         CHAINE(K):=T.CHAINE(N);
00120       N:=N+1;ENDFOR;
00121       IF K<>J+1
00122       THEN FOR M:=K BY 1 WHILE ((K<=J) AND (K<=CPT));
00123              CHAINE(K):=" ";
00124            ENDFOR;
00125       ENDIF;
00126     END
00127
00128 MODIF SUBD(T:TEXT,I:INTEGER,J:INTEGER);
00129     VAR
00130       K,N,M:INTEGER;
00131     END
00132     BEGIN
00133       N:=T.CPT;
00134       IF CPT<
00135       THEN M:=CPT;
00136       ELSE M:=J;ENDIF;
00137       FOR K:=M BY -1 WHILE ((K>=I) AND (K>0) AND (N>0));
00138         CHAINE(K):=T.CHAINE(N);
00139       N:=N-1;ENDFOR;
00140       IF K<>I-1
00141       THEN FOR M:=K BY -1 WHILE ((K>=I) AND (K>0));
00142         CHAINE(K):=" ";
00143         ENDFOR;
00144       ENDIF;
00145     END
00146
00147 MODIF AJUST(I:INTEGER);
00148     VAR
00149       M:INTEGER;
00150     END
00151     BEGIN
00152       CPT:=I;
00153       FOR M:=CPT+1 BY 1 WHILE M<=I
00154             CHAINE(M):=" ";
00155           ENDFOR;
00156     END
00157
00158 FUNCTION REPEAT(T:TEXT,I:INTEGER) RETURNS TEXT ;
00159     VAR
00160       T1:TEXT;
00161       I1:INTEGER;
00162     END
00163     BEGIN
00164       T1:="";
00165       FOR I1:=1 BY 1 WHILE I1<=I;
00166         @T1 CONTEXT(T);
00167       ENDFOR;
00168     RETURN T1; END
00169
00170 FUNCTION INDEX(T:TEXT) RETURNS INTEGER;
00171     VAR
00172       I1,I2,I3:INTEGER;
00173     END
00174     BEGIN
00175       I2:=0;
00176       FOR I1:=1 BY 1 WHILE I1<=CPT-T.CPT+1;
00177         I2:=1;
00178         FOR I3:=I1 BY 1 WHILE((I2<=T.CPT) AND (CHAINE(I3)=T.CHAINE(I2)));
00179           I2:=I2+1;
00180         ENDFOR;
00181         IF I2=T.CPT+1 THEN RETURN I1;ENDIF;
00182       ENDFOR;
00183       RETURN 0;
```



```
        f/jc:t:06:fseq/
00001*0,$1
00001           CAPSULE   FSEQ      FOR       FSEQ;
00002           SOCW
00003 B1        EQU       X'4000'
00004 B5        EQU       X'14000'
00005 B7        EQU       X'1C000'
00006 LI        S:SIN,2   X'22'
00007 CI        S:SIN,2   X'21'
00008 MI        S:SIN,2   X'23'
00009 AI        S:SIN,2   X'20'
00010 LIAI      S:SIN,2   X'01'
00011 BGE       S:SIN,1   X'681'
00012 B         S:SIN,1   X'680'
00013 BG        S:SIN,1   X'6?2'
00014 STW       S:SIN,0   X'55'
00015 LW        S:SIN,0   X'32'
00016 BAL       S:SIN,    X'6A'
00017 OR        S:SIN,0   X'49'
00018 LH  S:SIN,    X'52'
00019 STB       S:SIN,0   X'75'
00020 LCFI      S:SIN,0   X'02'
00021 XOR       S:SIN,0   X'64'
00022 STM       S:SIN,0   X'2B'
00023 LVAW      S:SIN,0   X'34'
00024 CALl      S:SIN,0   X'04'
00025 TEST      DATA      0
00026           LIAI,1    -($+1)
00027           B         B1!ABSVAL(CREATE)
00028           LIAI,1    -($+1)
00029           B         B1!ABSVAL(CRE)
00030           LIAI,1    -($+1)
00031           B         B1!ABSVAL(OPENG)
00032           LIAI,1    -($+1)
00033           B         B1!ABSVAL(OPENP)
00034           LIAI,1    -($+1)
00035           B         B1!ABSVAL(OPENU)
00036           LIAI,1    -($+1)
00037           B         B1!ABSVAL(CLBEGIN)
00038           LIAI,1    -($+1)
00039           B         B1!ABSVAL(CLEND)
00040           LIAI,1    -($+1)
00041           B         B1!ABSVAL(GET)
00042           LIAI,1    -($+1)
00043           B         B1!ABSVAL(PUT)
00044           LIAI,1    -($+1)
00045           B         B1!ABSVAL(REPLACE)
00046           LIAI,1    -($+1)
00047           B         B1!ABSVAL(DELETE)
00048           LIAI,1    -($+1)
00049           B         B1!ABSVAL(KILL)
00050 CRE       BAL,14    *X'3FFE'   PROLOGUE
00051 *  M:MOVEDCB DCBS,(PTR,*2)
00052           LCFI,2    0
00053           STM,0     X'3E02'
00054           LVAW,5    B1!ABSVAL(DCBS)
00055           LW,6      X'3E04'
00056           CALl,0    X'0E'
00057           LW,5      *2
00058           LW,6      B7!60-16
00059           AI,6      1
00060           LW,6      *6
00061           LW,7      *B7!60-17
00062           LW,8      *B7!60-18
00063           BAL,15    *X'3FE0'   M:SETDCB *5,(OPL,*6),(REL,*7),(BKL,*8)
```

```
00065 * RECOPIE DU NOM POUR TENIR COMPTE DES CONSTANTES
00066           LVAW,5     B1!ABSVAL(NOM)
00067           LW,10      B7!60-19
00068           AI,10      1
00069           LI,7       4
00070           LW,4       *10,7
00071           STW,4      *5,7
00072           AI,7       -1
00073           CI,7       0
00074           BGE        B1!ABSVAL($-4)
00075 * COMPLETER LE NOM AVEC DES BLANCS
00076           LVAW,5     B1!ABSVAL(NOM)
00077           LW,7       *B7!60-19
00078           LI,8       X'40'
00079           CI,7       16
00080           BG         B1!ABSVAL($+4)
00081           STB,8      *5,7
00082           AI,7       1
00083           B          B1!ABSVAL($-4)
00084 *   FIN REMPLISSAGE
00085           MI,5       4    R5 = ADRESSE OCTET DU NOM
00086           BAL,15     *X'3FE2' M:ASSIGN (OPL,*6),(FIL,*5),(UNT,OPL,'LECT'),
00087           LIAI,1     -($+1)
00088           B          *X'3FFD'    EPILOGUE
00089 OPENG     BAL,14     *X'3FFE'   PROLOGUE
00090           LW,5       *2
00091 *  M:OPEN *5,I
00092           LCFI,2     0
00093           STM,0      X'3E02'
00094 *         LW,5       X'3E07'
00095           LI,6       0
00096           LI,7       0
00097           CAL1,0     4
00098           B          *X'3FFD'    EPILOGUE
00099 OPENP     BAL,14     *X'3FFE'   PROLOGUE
00100           LW,5       *2
00101 *  M:OPEN *5,O
00102           LCFI,2     0
00103           STM,0      X'3E02'
00104 *         LW,5       X'3E07'
00105           LI,6       1
00106           LI,7       0
00107           CAL1,0     4
00108           B          *X'3FFD'    EPILOGUE
00109 OPENU     BAL,14     *X'3FFE'   PROLOGUE
00110           LW,5       *2
00111 *  M:OPEN *5,U
00112           LCFI,2     0
00113           STM,0      X'3E02'
00114 *         LW,5       X'3E07'
00115           LI,6       3
00116           LI,7       0
00117           CAL1,0     4
00118           B          *X'3FFD'    EPILOGUE
00119 CLBEGIN   BAL,14     *X'3FFE'   PROLOGUE
00120           LW,5       *2
00121 *  M:CLOSE *5,(MTN)
00122           LCFI,2     0
00123           STM,0      X'3E02'
00124 *         LW,5       X'3E07'
00125           LI,6       2
00126           CAL1,0     5
00127           B          *X'3FFD'    EPILOGUE
```



```
00128 CLEND     BAL,14     *X'3FFE'   PROLOGUE
00129           LW,5       *2
00130 *   M:CLOSE *5,(MTN),(LVE)
00131           LCFI,2     0
00132           STM,0      X'3E02'
00133 *         LW,5       X'3E07'
00134           LI,6       10
00135           CAL1,0     5
00136           B          *X'3FFD'    EPILOGUE
00137 GET       BAL,14     *X'3FFE'   PROLOGUE
00138           LW,5       *2
00139           LW,7       B7!60-16
00140   LW,8  5
00141   AI,8  8
00142   ..,8  *8
00143   LI,6  1
00144   LH,9  8,6
00145   STW,9  *7
00146   AI,7  1
00147           MI,7       4
00148 *  M:GET *5,(REC,*7)
00149           LCFI,2     0
00150           STM,0      X'3E02'
00151 *         LW,5       X'3E07'
00152           LW,6       X'3E09'
00153           LI,7       0
00154           LI,8       0
00155           CAL1,1     0
00156           LI,8       1
00157           STW,8      *B7!60-17
00158           B          B1!ABSVAL(FIN)
00159 AX2       LI,8       -1
00160           STW,8      *B7!60-17
00161           LCFI,2     0
00162           STM,0      X'3E02'
00163           LVAW,5     B1!ABSVAL(FIN)
00164           CAL1,0     X'1E'
00165 FIN       B          *X'3FFD'    EPILOGUE
00166 REPLACE   EQU        $
00167 PUT       BAL,14     *X'3FFE'   PROLOGUE
00168           LW,5       *2
00169           LW,7       B7!60-16
00170   AI,7  1
00171           MI,7       4
00172 *  M:PUT *5,(REC,*7)
00173           LCFI,2     0
00174           STM,0      X'3E02'
00175 *         LW,5       X'3E07'
00176           LW,6       X'3E09'
00177           LI,7       0
00178           LI,8       0
00179           LI,9       0
00180           CAL1,1     1
00181           B          *X'3FFD'    EPILOGUE
00182 DELETE    BAL,14     *X'3FFE'   PROLOGUE
00183           LW,5       *2
00184 *  M:DELREC *5
00185           LCFI,2     0
00186           STM,0      X'3E02'
00187 *         LW,5       X'3E07'
00188           CAL1,1     3
00189           B          *X'3FFD'    EPILOGUE
```

```
00190 KILL      BAL,14    *X'3FFE'   PROLOGUE
00191           LW,5      *2
00192 *  M:CLOSE *5,(RLS),(NCG)
00193           LCFI,2    0
00194           STM,0     X'3E02'
00195 *         LW,5      X'3E07'
00196           LI,6      X'10'
00197           CALl,0    5
00198           B         *X'3FFD'     EPILOGUE
00199 CREATE    LI,13     1
00200           STW,13    B5!6
00201           LVAW,6    B1!ABSVAL(AX2)
00202           OR,6      B1!ABSVAL(=X'02000000')
00203           STW,6     B1!ABSVAL(DCBS+3)
00204           B         *15
00205 DCBS      DATA      0,X'84000',0,X'0200406B',0,0
00206           DATA      X'D3C5C3E3',0,0,0,0,0,0,0,0,0,0,0,0
00207 NOM       DATA      0,0,0,0,0
00208           END
EOF HIT
*
```



```
f/jc:t:0c:find/
00001*0,$1
00001 CAPSULE FIND FOR FIND;
00002     SOCW
00003 B1  EQU  X'4000'
00004 B5  EQU  X'14000'
00005 B7  EQU  X'1C000'
00006 LI    S:SIN,2    X'22'
00007 CI    S:SIN,2    X'21'
00008 MI    S:SIN,2    X'23'
00009 D     S:SIN,2    X'20'
00010 LIAI    S:SIN,2  X'01'
00011 BGE   S:SIN,1    X'681'
00012 B     S:SIN,1    X'680'
00013 BG    S:SIN,1    X'692'
00014 STW   S:SIN,0    X'35'
00015 LW    S:SIN,0    X'32'
00016 BAL   S:SIN,0    X'6A'
00017 OR    S:SIN,0    X'49'
00018 LH    S:SIN,0    X'52'
00019 STB   S:SIN,0    X'75'
00020 LCFI  S:SIN,0    X'02'
00021 STM   S:SIN,0    X'2B'
00022 LVAW  S:SIN,0    X'34'
00023 CALl  S:SIN,0    X'04'
00024 TEST  DATA  0
00025    LIAI,1  -($+1)
00026    B  B1!ABSVAL(CREATE)
00027    LIAI,1  -($+1)
00028    B  B1!ABSVAL(CRE)
00029    LIAI,1 -($+1)
00030    B B1!ABSVAL(OPENG)
00031   LIAI,1 -($+1)
00032    B B1!ABSVAL(OPENP)
00033    LIAI,1 -($+1)
00034    B B1!ABSVAL(OPENU)
00035    LIAI,1 -($+1)
00036    B  B1!ABSVAL(CLBEGIN)
00037    LIAI,1 -($+1)
00038    B  B1!ABSVAL(CLEND)
00039    LIAI,1 -($+1)
00040    B  B1!ABSVAL(GET)
00041    LIAI,1 -($+1)
00042    B  B1!ABSVAL(GETN)
00043    LIAI,1 -($+1)
00044    B  B1!ABSVAL(PUT)
00045    LIAI,1 -($+1)
00046    B  B1!ABSVAL(PUTN)
00047    LIAI,1 -($+1)
00048    B  B1!ABSVAL(DELETE)
00049    LIAI,1 -($+1)
00050    B  B1!ABSVAL(KILL)
```

```
00051 CRE  BAL,14  *X'3FFE'
00052    LCFI,2 0
00053    STM,0  X'3E02'
00054    LVAW,5  B1!ABSVAL(DCBI)
00055    LW,6  X'3E04'
00056    CAL1,0  X'0E'
00057    LW,5  *2
00058    LW,6  B7!60-16
00059    AI,6  1
00060    LW,6  *6
00061    LW,7  *B7!60-17
00062    LW,8  *B7!60-18
00063    LW,9  *B7!60-20
00064    LW,10  *B7!60-21
00065     BAL,15  *X'3FE0'
00066    BAL,15  *X'3FE1'
00067     LIAI,1 -($+1)
00068    LVAW,5  B1!ABSVAL(NOM)
00069    LW,10  B7!60-19
00070    AI,10  1
00071    LI,7  4
00072    LW,4  *10,7
00073    STW,4  *5,7
00074    AI,7  -1
00075    CI,7  0
00076    BGE  B1!ABSVAL($-4)
00077    LVAW,5  B1!ABSVAL(NOM)
00078    LW,7  *B7!60-19
00079    LI,8  X'40'
00080    CI,7  16
00081    BG  B1!ABSVAL($+4)
00082    STB,8  *5,7
00083    AI,7  1
00084    B  B1!ABSVAL($-4)
00085    MI,5  4
00086    BAL,15  *X'3FE2'
00087    LIAI,1  -($+1)
00088    B  *X'3FFD'
00089 OPENG  BAL,14  *X'3FFE'
00090    LW,5  *2
00091    LCFI,2  0
00092    STM,0  X'3E02'
00093    LI,6  0
00094    LI,7  0
00095    CAL1,0  4
00096    B  *X'3FFD'
00097 OPENP  BAL,14  *X'3FFE'
00098    LW,5  *2
00099    LCFI,2  0
00100    STM,0  X'3E02'
00101    LI,6  1
00102    LI,7  0
00103    CAL1,0  4
00104    B  *X'3FFD'
00105 OPENU  BAL,14  *X'3FFE'
00106    LW,5  *2
00107    LCFI,2  0
00108    STM,0  X'3E02'
00109    LI,6  3
00110    LI,7  0
00111    CAL1,0  4
00112    B  *X'3FFD'
00113 CLBEGIN  BAL,14  *X'3FFE'
00114    LW,5  *2
00115    LCFI,2  0
00116    STM,0  X'3E02'
00117    LI,6  2
00118    CAL1,0  5
00119    B  *X'3FFD'
```



```
00120 CLEND  BAL,14  *X'3FFE'
00121    LW,5  *2
00122    LCFI,2  0
00123    STM,0  X'3E02'
00124    LI,6  10
00125    CAL1,0  5
00126    B  *X'3FFD'
00127 GET  BAL,14  *X'3FFE'
00128    LW,5  *2
00129    LW,7  B7!60-16
00130    LW,8  5
00131    AI,8  8
00132    LW,8  *8
00133    LI,6  1
00134    LH,9  8,6
00135    STW,9  *7
00136    AI,7  1
00137    MI,7  4
00138    LW,4  B7!60-17
00139    LCFI,2  0
00140    STM,0  X'3E02'
00141    LW,6  X'3E09'
00142    LI,7  0
00143    LI,8  0
00144    CAL1,1  0
00145    LI,8  1
00146    STW,8  *4
00147    B  B1!ABSVAL(FIN)
00148 AX2  LI,8  -1
00149    STW,8  *4
00150    LCFI,2  0
00151    STM,0  X'3E02'
00152    LVAW,5  B1!ABSVAL(FIN)
00153    CAL1,0  X'1E'
00154 FIN  B  *X'3FFD'
00155 GETN  BAL,14  *X'3FFE'
00156    LW,5  *2
00157    LW,7  B7!60-17
00158    LW,8  5
00159    AI,8  8
00160    LW,8  *8
00161    LI,6  1
00162    LH,9  8,6
00163    AI,7  1
00164    MI,7  4
00165    LW,9  B7!60-16
00166     AI,9  1
00167    MI,9  4
00168    LW,4  B7!60-18
00169    LCFI,2  0
00170    STM,0  X'3E02'
00171    LW,6  X'3E09'
00172    LI,7  0
00173    LW,8  X'3E0B'
00174    CAL1,1  0
00175    LI,8  1
00176    STW,8  *4
00177    B  *X'3FFD'
```

```
00178 PUT  BAL,14  *X'3FFE'
00179   LW,5  *2
00180   LW,7  B7!60-16
00181   AI,7  1
00182   MI,7  4
00183   LW,4  B7!60-17
00184   LCFI,2  0
00185   STM,0  X'3E02'
00186   LW,6  X'3E09'
00187   LI,7  0
00188   LI,8  0
00189   LI,9  0
00190   CAL1,1  1
00191   LI,8  1
00192   STW,8  *4
00193   B  *X'3FFD'
00194 PUTN  BAL,14  *X'3FFE'
00195   LW,5  *2
00196   LW,7  B7!60-16
00197   AI,7  1
00198   MI,7  4
00199   LW,4  B7!60-17
00200   LCFI,2  0
00201   STM,0  X'3E02'
00202   LW,6  X'3E09'
00203   LI,7  0
00204   LI,8  1
00205   LI,9  0
00206   CAL1,1  1
00207   LI,8  1
00208   STW,8  *4
00209   B  *X'3FFD'
00210 DELETE  BAL,14  *X'3FFE'
00211   LW,5  *2
00212   LCFI,2  0
00213   STM,0  X'3E02'
00214   CAL1,1  3
00215   B  *X'3FFD'
00216 KILL  BAL,14  *X'3FFE'
00217   LW,5  *2
00218   LCFI,2  0
00219   STM,0  X'3E02'
00220   LI,6  X'10'
00221   CAL1,0  5
00222   B  *X'3FFD'
00223 CREATE  LI,13  I
00224   STW,13  B5!12
00225   LVAW,6  B1!ABSVAL(AX2)
00226   OR,6  B1!ABSVAL(=X'06000000')
00227   STW,6  B1!ABSVAL(DCBI+3)
00228   B  *15
00229 DCBI  DATA  0,X'1084000',X'1000000',0,0,0
00230   DATA  X'D3C5C3E3',0,0,0,0,0,0,0,0,0,0,0,0
00231 NOM  DATA  0,0,0,0,0
00232   END
EOF HIT
*
```

. capsules des types construits


```
 f/jc:t:07:date1/
00001*0,$1
00001 CAPSULE DATE1 FOR DATE;
00002   REP J:INTEGER; M:INTEGER; A:INTEGER;
00003   END
00004   CONSULT OUT ->T1:TEXT;
00005     VAR T2,T3:TEXT;
00006     END
00007     BEGIN
00008       T3:=''; @T3 AJUST(6); T2:=$CONV TEXTINT(J);
00009       @T3 SUBD(T2,1,2); T2:=$CONV TEXTINT(M);
00010       @T3 SUBD(T2,3,4); T2:=$CONV TEXTINT(A);
00011       @T3 SUBD(T2,5,6); T1:=T3;
00012     END
00013   BUILD IN(T1:TEXT);
00014     VAR T2:TEXT;
00015     END
00016     BEGIN
00017       T2:=@T1 SUBSTR(1,2); J:=$CONV INTTEXT(T2);
00018       T2:=@T1 SUBSTR(3,4); M:=$CONV INTTEXT(T2);
00019       T2:=@T1 SUBSTR(5,6); A:=$CONV INTTEXT(T2);
00020     END
00021   FUNCTION GT(D1:DATE) RETURNS BOOLEAN;
00022     BEGIN
00023       IF A>@D1 ANNEE THEN RETURN TRUE;
00024       ELSIF (A=@D1 ANNEE) AND (M>@D1 MOIS) THEN RETURN TRUE;
00025       ELSIF (A=@D1 ANNEE) AND (M=@D1 MOIS) AND (J>@D1 JOUR)
00026       THEN RETURN TRUE; ELSE RETURN FALSE; ENDIF;
00027     END
00028   FUNCTION JOUR RETURNS INTEGER;
00029     BEGIN
00030       RETURN J;
00031     END
00032   FUNCTION MOIS RETURNS INTEGER;
00033     BEGIN
00034       RETURN M;
00035     END
00036   FUNCTION ANNEE RETURNS INTEGER;
00037     BEGIN
00038       RETURN A;
00039    END
00040   BUILD CREDATJ;
00041     VAR T,T1:TEXT;
00042     END
00043     BEGIN
00044       T:=$TIME TIME;  T1:=@T SUBSTR(13,14);
00045       @T1 CONTEXT(@T SUBSTR(16,17));
00046       @T1 CONTEXT(@T SUBSTR(19,20));
00047       CREE IN(T1);
00048     END
00049   BUILD  CREDAT(I:INTEGER);
00050     VAR  D1:DATE; T:ARRAY(12) OF INTEGER; J1,M1,A1:INTEGER;
00051     END
00052     BEGIN
00053       T(1):=31; T(2):=28; T(3):=31; T(4):=30;
00054       T(5):=31; T(6):=30; T(7):=31; T(8):=31;
00055       T(9):=30; T(10):=31; T(11):=30; T(12):=31;
00056       @D1 CREDATJ; J1:=@D1 JOUR + I;
00057       M1:=@D1 MOIS; A1:=@D1 ANNEE;
00058       WHILE (J1>T(M1));
00059        J1:=J1-T(M1); M1:=M1+1;
00060        IF M1>12 THEN M1:=1; A1:=A1+1; ENDIF;
00061       ENDWHILE;
00062       I:=J1; M:=M1; A:=A1;
```

```
f/jc:t:07/date2/
00001*0,$1
00001 CAPSULE DATE2 FOR DATE;
00002   REP T:TEXT;
00003   END
00004   CONSULT OUT -> T1:TEXT;
00005     BEGIN  T1:=T;
00006     END
00007   BUILD IN(T1:TEXT);
00008     BEGIN T:=T1;
00009     END
00010   FUNCTION GT(D1:DATE) RETURNS BOOLEAN;
00011     BEGIN  IF @REP ANNEE > @D1 ANNEE THEN RETURN TRUE;
00012       ELSIF (@REP ANNEE = @D1 ANNEE) AND (@REP MOIS > @D1 MOIS)
00013       THEN RETURN TRUE;
00014        ELSIF (@REP ANNEE = @D1 ANNEE) AND (@REP MOIS = @D1 MOIS)
00015       AND (@REP JOUR > @D1 JOUR) THEN RETURN TRUE;
00016       ELSE RETURN FALSE; ENDIF:
00017     END
00018   FUNCTION JOUR RETURNS INTEGER;
00019     VAR T1:TEXT;
00020     END
00021     BEGIN  T1:= @T SUBSTR(1,2);
00022       RETURN $CONV INTTEXT(T1);
00023     END
00024   FUNCTION MOIS RETURNS INTEGER;
00025     VAR T1:TEXT;
00026     END
00027     BEGIN T1:=@T SUBSTR(3,4);
00028       RETURN $CONV INTTEXT(T1);
00029     END
00030   FUNCTION ANNEE RETURNS INTEGER;
00031     VAR T1:TEXT;
00032     END
00033     BEGIN  T1:=@T SUBSTR(5,6);
00034       RETURN $CONV INTTEXT(T1);
00035     END
00036   BUILD CREDATJ;
00037     VAR T1,T2:TEXT;
00038     END
00039     BEGIN  T1:=$TIME TIME;
00040      T2:=@T1 SUBSTR(13,14);
00041       @T2 CONTEXT(@T1 SUBSTR(16,17));
00042       @T2 CONTEXT(@T1 SUBSTR(19,20));
00043       T:=T2;
00044     END
00045   BUILD CREDAT(I:INTEGER);
00046     VAR D1:DATE; T1:ARRAY(12) OF INTEGER;
00047       J1,M1,A1:INTEGER; T2:TEXT;
00048     END
00049     BEGIN  T1(1):=31; T1(2):=28; T1(3):=31; T1(4):=30;
00050       T1(5):=31; T1(6):=30; T1(7):=31; T1(8):=31;
00051       T1(9):=30; T1(10):=31; T1(11):=30; T1(12):=31;
00052       @D1 CREDATJ; J1:=@D1 JOUR +I;
00053       M1:=@D1 MOIS; A1:=@D1 ANNEE;
00054       WHILE (J1>T1(M1));  J1:=J1-T1(M1);
00055        M1:=M1+1;
00056        IF M1>12 THEN M1:=1; A1:=A1+1; ENDIF;
00057       ENDWHILE;
00058       T2:=$CONV TEXTINT(J1);
00059       @T2 CONTEXT($CONV TEXTINT(M1));
00060       @T2 CONTEXT($CONV TEXTINT(A1));
00061       T:=T2;
00062     END
00063 END
EOF HIT
*
```



```
f/jc:t:08:livre/
00001*0,$1
00001 CAPSULE LIVRE FOR LIVRE;
00002   REP NO:INTEGER; S:TEXT; T:TEXT; A:TEXT;
00003   END
00004   BUILD CREER(I:INTEGER,S1:TEXT,T1:TEXT,A1:TEXT);
00005     BEGIN NO:=I; S:=S1;
00006           T:=T1; A:=A1;
00007     END
00008   FUNCTION CLE RETURNS INTEGER;
00009     BEGIN RETURN NO;
00010     END
00011   FUNCTION SUJET RETURNS TEXT;
00012     BEGIN RETURN S;
00013     END
00014   FUNCTION TITRE RETURNS TEXT;
00015     BEGIN RETURN T;
00016     END
00017   FUNCTION AUTEUR RETURNS TEXT;
00018     BEGIN RETURN A;
00019     END
00020   CONSULT OUT ->T1:TEXT;
00021     VAR T2,T3:TEXT;
00022     END
00023     BEGIN T3:='';
00024           T2:=$CONV TEXTINT(NO);
00025           @T3 AJUST(44);
00026           @T3 SUBD(T2,1,4);
00027           @T3 SUBG(S,5,14);
00028           @T3 SUBG(T,15,34);
00029           @T3 SUBG(A,35,44);
00030           T1:=T3;
00031     END
00032   BUILD IN(T1:TEXT);
00033     VAR T2:TEXT;
00034     END
00035     BEGIN T2:=@T1 SUBSTR(1,4);
00036           NO:=$CONV INTTEXT(T2);
00037           S:=@T1 SUBSTR(5,14);
00038           T:=@T1 SUBSTR(15,34);
00039           A:=@T1 SUBSTR(35,44);
00040     END
00041 END
EOF HIT
*
```

```
 f/jc:t:09:cherch/
00001*0,$1
00001 CAPSULE CHERCH FOR CHERCH;
00002   REP NO:INTEGER; N:TEXT;
00003   END
00004   BUILD CREER(I:INTEGER,T:TEXT);
00005     BEGIN NO:=I; N:=T;
00006     END
00007   FUNCTION CLE RETURNS INTEGER;
00008     BEGIN RETURN NO;
00009     END
00010   FUNCTION NOM RETURNS TEXT;
00011     BEGIN RETURN N;
00012     END
00013   CONSULT OUT ->T:TEXT;
00014     VAR T2,T3:TEXT;
00015     END
00016     BEGIN T3:='';
00017           @T3 AJUST(14);
00018           T2:=$CONV TEXTINT(NO);
00019           @T3 SUBD(T2,1,4);
00020           @T3 SUBG(N,5,14);
00021           T:=T3;
00022     END
00023   BUILD IN(T:TEXT);
00024     VAR T2:TEXT;
00025     END
00026     BEGIN T2:=@T SUBSTR(1,4);
00027           NO:=$CONV INTTEXT(T2);
00028           N:=@T SUBSTR(5,14);
00029     END
00030   FUNCTION EQ(C:CHERCH) RETURNS BOOLEAN;
00031       BEGIN
00032       IF NO = @C CLE THEN RETURN TRUE;
00033       ELSE RETURN FALSE; ENDIF;
00034     END
00035 END
EOF HIT
*
```



```
 f/jc:t:0a:pret/
00001*0,$1
00001 CAPSULE PRET FOR PRET;
00002   REP IL:INTEGER; IC:INTEGER; D1:DATE; D2:DATE;
00003   END
00004   BUILD CREER(L:LIVRE,C:CHERCH,DA1:DATE,DA2:DATE);
00005     BEGIN IL:=@L CLE;
00006           IC:=@C CLE;
00007           D1:=DA1; D2:=DA2;
00008     END
00009   FUNCTION LIVRE RETURNS LIVRE;
00010     VAR B:BOOLEAN; L:LIVRE;
00011     END
00012     BEGIN $LIVRES ACELEMNO(IL) ->L,B;
00013           RETURN L;
00014     END
00015   FUNCTION CHERCH RETURNS CHERCH;
00016     VAR B:BOOLEAN; C:CHERCH;
00017     END
00018     BEGIN $CHERCHS ACELEMNO(IC) ->C,B;
00019           RETURN C;
00020     END
00021   FUNCTION DATEPRE RETURNS DATE;
00022     BEGIN RETURN D1;
00023     END
00024   FUNCTION DATEDU RETURNS DATE;
00025     BEGIN RETURN D2;
00026     END
00027   CONSULT OUT ->T:TEXT;
00028     VAR T2,T3:TEXT;
00029     END
00030     BEGIN T3:='';
00031           @T3 AJUST(20);
00032           T2:=$CONV TEXTINT(IL);
00033           @T3 SUBD(T2,1,4);
00034           T2:=$CONV TEXTINT(IC);
00035           @T3 SUBD(T2,5,8);
00036           @D1 OUT ->T2;
00037           @T3 SUBG(T2,9,14);
00038           @D2 OUT ->T2;
00039           @T3 SUBG(T2,15,20);
00040           T:=T3;
00041     END
00042   BUILD IN(T:TEXT);
00043     VAR T2:TEXT;
00044     END
00045     BEGIN T2:=@T SUBSTR(1,4);
00046           IL:=$CONV INTTEXT(T2);
00047           T2:=@T SUBSTR(5,8);
00048           IC:=$CONV INTTEXT(T2);
00049           T2:=@T SUBSTR(9,14);
00050           @D1 IN(T2);
00051           T2:=@T SUBSTR(15,20);
00052           @D2 IN(T2);
00053     END
00054   FUNCTION NUMERO RETURNS INTEGER;
00055     BEGIN RETURN IL; END
00056     END
EOF HIT
*
```

```
 f/jc:t:0b:demande/
00001*0,$1
00001 CAPSULE DEMANDE FOR DEMANDE;
00002   REP IL:INTEGER; IC:INTEGER; D:DATE;
00003   END
00004   BUILD CREER(L:LIVRE,C:CHERCH,DA:DATE);
00005     BEGIN IL:=@L CLE;
00006           IC:=@C CLE;
00007           D:=DA;
00008     END
00009   FUNCTION LIVRE RETURNS LIVRE;
00010     VAR B:BOOLEAN; L:LIVRE;
00011     END
00012     BEGIN $LIVRES ACELEMNO(IL) ->L,B;
00013           RETURN L;
00014     END
00015   FUNCTION CHERCH RETURNS CHERCH;
00016     VAR B:BOOLEAN; C:CHERCH;
00017     END
00018     BEGIN $CHERCHS ACELEMNO(IC) ->C,B;
00019           RETURN C;
00020     END
00021   FUNCTION DATEDEM RETURNS DATE;
00022     BEGIN RETURN D;
00023     END
00024   CONSULT OUT ->T:TEXT;
00025     VAR T2,T3:TEXT;
00026     END
00027     BEGIN T3:="";
00028           @T3 AJUST(14);
00029           T2:=$CONV TEXTINT(IL);
00030           @T3 SUBD(T2,1,4);
00031           T2:=$CONV TEXTINT(IC);
00032           @T3 SUBD(T2,5,8);
00033           @D OUT ->T2;
00034           @T3 SUBG(T2,9,14);
00035           T:=T3;
00036     END
00037   BUILD IN(T:TEXT);
00038     VAR T2:TEXT;
00039     END
00040     BEGIN T2:=@T SUBSTR(1,4);
00041           IL:=$CONV INTTEXT(T2);
00042           T2:=@T SUBSTR(5,8);
00043           IC:=$CONV INTTEXT(T2);
00044           T2:=@T SUBSTR(9,14);
00045           @D IN(T2);
00046     END
00047   FUNCTION NUMERO RETURNS INTEGER;
00048     BEGIN RETURN IL; END
00049 END
EOF HIT
*
```


• machines prédéfinies

```
          f/jc:m:00/
00001*0,$1
00001 MACHINE READER;
00002     # LECTEUR DE CARTES
00003 PROC READ->TEXT,BOOLEAN;
00004     # $READER READ->C,B SIGNIFIE:
00005     # LIRE UNE CARTE DANS C SI C'EST POSSIBLE
00006     # SI LA LECTURE EST FAITE,B DEVIENT  VRAI
00007     # SI IL Y A 'END OF FILE',B DEVIENT FAUX
00008 END
EOF HIT
*
 f/jc:m:01/
00001*0,$1
00001 MACHINE WRITER;
00002   # IMPRIMANTE
00003 PROC WRITE(TEXT);
00004     # $WRITER WRITE(L) SIGNIFIE: IMPRIMER LA LIGNE L
00005 END
EOF HIT
*
```

```
 f/jc:m:07/
00001*0,$1
00001 MACHINE MAIN;
00002   PROC START;
00003 END
EOF HIT
*
```

```
 f/jc:m:08/
00001*0,$1
00001 MACHINE TIME;
00002 FUNCTION TIME RETURNS TEXT;
00003 END
EOF HIT
*
```

```
 f/jc:m:02/
00001*0,$1
00001 MACHINE CONV;
00002       FUNCTION TEXTNAT (INTEGER)  RETURNS TEXT;
00003       FUNCTION TEXTINT (INTEGER)  RETURNS TEXT;
00004       FUNCTION TEXTREAL(REAL)     RETURNS TEXT;
00005       FUNCTION TEXTCHAR(CHAR)     RETURNS TEXT;
00006       FUNCTION INTTEXT (TEXT)     RETURNS INTEGER;
00007       FUNCTION CHARTEXT(TEXT)     RETURNS CHAR;
00008       FUNCTION INTREAL (REAL)     RETURNS INTEGER;
00009       FUNCTION REALINT (INTEGER)  RETURNS REAL;
00010 END;
EOF HIT
*
```

• machines construites

```
 f/jc:m:09/
00001*0,$1
00001 MACHINE LIVRES   ;
00002
00003
00004
00005
00006
00007
00008
00009
00010
00011 PROC CREER;
00012
00013 PROC AJOUTER( LIVRE ) ->BOOLEAN;
00014
00015 PROC SUPPRIME(INTEGER);
00016
00017
00018
00019 PROC ACELEMNO(INTEGER) -> LIVRE ,BOOLEAN;
00020
00021
00022
00023 PROC LISTCOLL;
00024
00025 PROC PREMIER -> LIVRE ,BOOLEAN;
00026
00027
00028
00029 PROC SUIVANT -> LIVRE ,BOOLEAN;
00030
00031
00032
00033 PROC LOGIN;
00034
00035 PROC LOGOUT;
00036
00037 END
EOF HIT
*
```



```
 f/jc:m:0a/
00001*0,$1
00001 MACHINE CHERCHS  ;
00002
00003
00004
00005
00006
00007
00008
00009
00010
00011 PROC CREER;
00012
00013 PROC AJOUTER( CHERCH ) ->BOOLEAN;
00014
00015 PROC SUPPRIME(INTEGER);
00016
00017
00018
00019 PROC ACELEMNO(INTEGER) -> CHERCH ,BOOLEAN;
00020
00021
00022
00023 PROC LISTCOLL;
00024
00025 PROC PREMIER -> CHERCH ,BOOLEAN;
00026
00027
00028
00029 PROC SUIVANT -> CHERCH ,BOOLEAN;
00030
00031
00032
00033 PROC LOGIN;
00034
00035 PROC LOGOUT;
00036
00037 END
EOF HIT
*
```

```
 f/jc:m:0b/
00001*0,$1
00001 MACHINE PRETS;
00002   PROC CREER;
00003     # CREER LA COLLECTION VIDE #
00004   PROC AJOUTER(PRET) -> BOOLEAN;
00005     # AJOUTER UN PRET A LA COLLECTION #
00006   PROC SUPPRIME(INTEGER);
00007     # SUPPRIMER UN PRET DE LA COLLECTION #
00008     # FIND --> CELUI DE CLE I #
00009     # FSEQ --> CELUI TEL QUE CLE(P) = I #
00010   PROC ACELEMNO(INTEGER) ->PRET,BOOLEAN;
00011     #  ACCES A UN PRET DE LA COLLECTION #
00012     # FIND --> CELUI DE CLE I #
00013     # FSEQ --> CELUI TEL QUE CLE(P) = I #
00014   PROC LISTCOLL;
00015     # LISTER TOUTE LA COLLECTION #
00016   ITER SELECT YIELDS PRET;
00017     # ITERATEUR QUI DONNE LES PRETS #
00018   PROC LOGIN;
00019     # LOGIN DE LA COLLECTION #
00020   PROC LOGOUT;
```

```
 f/jc:m:0c/
00001*0,$1
00001 MACHINE DEMANDES;
00002   PROC CREER;
00003   PROC AJOUTER(DEMANDE) ->BOOLEAN;
00004   PROC SUPPRIME(INTEGER);
00005   PROC ACELEMNO(INTEGER) ->DEMANDE,BOOLEAN;
00006   PROC LISTCOLL;
00007   ITER SELECT YIELDS DEMANDE;
00008   PROC LOGIN;
00009   PROC LOGOUT;
00010 END
EOF HIT
*
```

```
 f/jc:m:0d/
00001*0,$1
00001 MACHINE BIB;
00002   PROC GO;
00003   PROC AJLIVR;
00004   PROC SUPLIVR;
00005   PROC AJCHER;
00006   PROC SUPCHER;
00007   PROC EMPRLIVR;
00008   PROC RESTLIVR;
00009   PROC P1;
00010   PROC P2;
00011   PROC P3;
00012   PROC P4;
00013   PROC P5;
00014   PROC P6;
00015   PROC P7;
00016  END
EOF HIT
*
```

```
 f/jc:m:0e/
00001*0,$1
00001  MACHINE MAIN1;
00002   PROC START;
00003 END
EOF HIT
*
```



```
 f/jc:m:00:lire/
00001*0,$1
00001 MODULE LIRE FOR READER;
00002            SOCW
00003 B1         EQU      X'4000'
00004 B7         EQU      X'1C000'
00005 OR         S:SIN,0  X'49'
00006 BGZ        S:SIN,1  X'692'
00007 LI         S:SIN,2  X'22'
00008 CAL1       S:SIN,0  X'04'
00009 STM        S:SIN,0  X'2B'
00010 LCFI       S:SIN,0  X'02'
00011 MI         S:SIN,2  X'23'
00012 LW         S:SIN,0  X'32'
00013 STW        S:SIN,0  X'35'
00014 AI         S:SIN,2  X'20'
00015 BAL        S:SIN,0  X'6A'
00016 B          S:SIN,1  X'680'
00017 LIAI       S:SIN,2  X'01'
00018 LVAW       S:SIN,0  X'34'
00019 *
00020 TEST       DATA     0
00021            LIAI,1   -($+1)
00022            B        B1!ABSVAL(CREATE)
00023            LIAI,1   -($+1)
00024            B        B1!ABSVAL(READ)
00025 READ       BAL,14   *X'3FFE'    PROLOGUE
00026            BAL,15   B1!ABSVAL(CREATE)
00027            LW,5     60-16,1     R5=ADRESSE MOT DU COMPTE
00028            LW,7     5
00029            AI,7     1
00030            MI,7     4       R7=ADRESSE OCTET DE LA CHAINE
00031            LW,8     B7!60-17      R8=ADRESSE DU BOOLEAN
00032 *GET LECT,(REC,*6),(RSA,*5)
00033            LCFI,2   0
00034            STM,0    X'3E02'
00035            LW,5     X'3FFB'     DCB
00036            LW,6     X'3E09'     *7
00037            LW,7     X'3E07'     *5
00038            LI,8     0
00039            CAL1,1   0
00040            LI,7     1
00041            STW,7    *8
00042 RETOUR     LW,5     B1!ABSVAL(SAVE)
00043            STW,5    X'3FFF'       ABORT
00044            B        *X'3FFD'     EPILOGUE
00045 *
00046 SAVE       DATA     0
```

```
00047 CREATE    LW,5      X'3FFF'     ABRT
00048           STW,5     B1!ABSVAL(SAVE)
00049           LW,5      B1!ABSVAL(=X'68004000')
00050           LI,6      ABSVAL(FINFICH)
00051           OR,6      5
00052           STW,6     X'3FFF'
00053 * LE CHAINAGE EST A L'ADRESSE DE DEROUTEMENT
00054           B         *15
00055 FINFICH   LI,7      -1
00056           STW,7     *8
00057 * FERMER LE FICHIER EET LE REOUVRIR
00058 * CLOSE @LECT,(RRD),(HLD)
00059           LCFI,2    0
00060           STM,0     X'3E02'     JTCB
00061           LW,5      X'3FFB'     DCB
00062           LI,6      5
00063           CAL1,0    5
00064 * OPEN @LECT,I
00065           LCFI,2    0
00066           STM,0     X'3E02'       JTCB
00067           LW,5      X'3FFB'   DCB
00068           LI,6      0
00069           LI,7      0
00070           CAL1,0    4
00071 * RETURN RETOUR
00072           LCFI,2    0
00073           STM,0     X'3E02'
00074           LVAW,5    B1!ABSVAL(RETOUR)
00075           CAL1,0    X'1E'
00076           END
EOF HIT
*
```



```
 f/jc:m:01;ecrire/
00001*0,$1
00001   MODULE ECRIRE FOR WRITER;
00002          SOCW
00003 B1       EQU       X'4000'
00004 LIAI     S:SIN,2   X'01'
00005 B        S:SIN,1   X'680'
00006 BAL      S:SIN,0   X'6A'
00007 LW       S:SIN,0   X'32'
00008 AI       S:SIN,2   X'20'
00009 MI       S:SIN,2   X'23'
00010 LCFI     S:SIN,2   X'02'
00011 STM      S:SIN,0   X'2B'
00012 LI       S:SIN,2   X'22'
00013 CAL1     S:SIN,2   X'04'
00014 TEST     DATA      0
00015          LIAI,1    -($+1)
00016          B         B1!ABSVAL(CREATE)
00017          LIAI,1    -($+1)
00018          B         B1!ABSVAL(ECRIRE)
00019 CREATE   B         *15
00020 ECRIRE   BAL,14    *X'3FFE'    PROLOGUE
00021          LW,5      60-16,1
00022          LW,7      5
00023          AI,7      1
00024          MI,7      4
00025          LW,5      *5
00026 * M:PUT @ECRI,(REC,*7),(RSA,*5)
00027          LCFI,2    0
00028          STM,0     X'3E02'
00029          LW,5      X'3FFA'   DCB
00030          LW,6      X'3E09'
00031          LW,7      X'3E07'
00032          LI,8      0
00033          LI,9      0
00034          CAL1,1  1
00035          B         *X'3FFD'  EPILOGUE
00036          END
EOF HIT
*
```

```
00001*0,$1
00001 MODULE CONV FOR CONV;
00002
00003    FUNCTION TEXTNAT(I:INTEGER) RETURNS TEXT;
00004      VAR
00005        I1,I2:INTEGER;
00006        T1,T2:TEXT;
00007      END
00008      BEGIN
00009        IF I>9
00010        THEN I1:=I/10;
00011             I2:=I-I1*10;
00012             T1:=$CONV TEXTNAT(I1);
00013        ELSE T1:="";I2=I;
00014        ENDIF;
00015
00016        T2:="0123456789";
00017        @T1 CONTEXT(@T2 SUBSTR(I2+1,I2+1));
00018        RETURN T1;
00019      END
00020
00021    FUNCTION TEXTINT(I:INTEGER) RETURNS TEXT;
00022      VAR
00023         T1:TEXT;
00024         I1:INTEGER;
00025      END
00026      BEGIN
00027         IF I=0
00028         THEN RETURN "0"; ENDIF;
00029         IF I<0
00030         THEN T1:="-";I1:=-I;
00031         ELSE T1:="";I1:=I;
00032         ENDIF;
00033
00034         @T1 CONTEXT($CONV TEXTNAT(I1));
00035         RETURN T1;
00036      END
00037
00038    FUNCTION TEXTREAL(R:REAL) RETURNS TEXT;
00039      VAR
00040        R1:REAL;
00041        T1:TEXT;
00042        I1,I2:INTEGER;
00043      END
00044      BEGIN
00045        IF R<0.
00046        THEN R1:=-R;
00047             T1:="-0.";
00048        ELSE R1:=R;
00049             T1:="+0.";
00050        ENDIF;
00051
00052        I1:=0;
00053        WHILE R1>1.;
00054           I1:=I1+1;
00055           R1:=R1/10.;
00056        ENDWHILE;
00057
00058        WHILE R1<0.1;
00059           I1:=I1-1;
00060           R1:=R1*10.;
00061           ENDWHILE;
00062
00063        R1:=R1*1000000.;
00064        @T1 CONTEXT($CONV TEXTNAT($CONV INTREAL(R1)));
00065        @T1 CONTEXT("E");
00066        @T1 CONTEXT($CONV TEXTINT(I1));
```



```
00070    FUNCTION TEXTCHAR(C:CHAR) RETURNS TEXT;
00071      VAR
00072        T1:TEXT;
00073      END
00074      BEGIN
00075        T1:="";
00076        @T1 CONCHAR(C);
00077        RETURN T1;
00078      END
00079
00080 FUNCTION INTTEXT(T:TEXT) RETURNS INTEGER;
00081      VAR
00082        I1,I2,I3,I4:INTEGER;
00083        B:BOOLEAN;
00084        C:CHAR;
00085      END
00086      BEGIN
00087        I2:=@T LENGTH;
00088        IF I2=0 THEN RETURN 0 ; ENDIF;
00089
00090        C:=@T GETCHAR(1);
00091  I1:=1; WHILE(C=" "); I1:=I1+1;
00092  C:=@T GETCHAR(I1); ENDWHILE;
00093        IF C="-"
00094        THEN B:=FALSE;
00095             I1:=I1+1;
00096        ELSIF C="+"
00097        THEN B:=TRUE;
00098             I1:=I1+1;
00099        ELSIF ((C>="0") AND (C<="9"))
00100        THEN B:=TRUE;
00101        ELSE RETURN 0;
00102        ENDIF;
00103
00104        I3:=0;
00105
00106        FOR I1:=I1 BY 1 WHILE I1<=I2
00107           C:=@T GETCHAR(I1);
00108           IF ((C<"0") OR (C>"9"))
00109           THEN RETURN 0;
00110           ENDIF;
00111
00112           I3:=I3*10;
00113           IF C>"7"
00114           THEN IF C = "8"
00115                THEN I3:=I3+8;
00116                ELSE I3:=I3+9;
00117                ENDIF;
00118           ELSE IF C>"3"
00119                THEN IF C>"5"
00120                     THEN IF C="6"
00121                          THEN I3:=I3+6;
00122                          ELSE I3:=I3+7;
00123                          ENDIF;
00124                     ELSE IF C="5"
00125                          THEN I3:=I3+5;
00126                          ELSE I3:=I3+4;
00127                          ENDIF;
00128                ENDIF;
00129           ELSE IF C>"1"
00130                THEN IF C="2"
00131                     THEN I3:=I3+2;
00132                     ELSE I3:=I3+3;
00133                     ENDIF;
00134                ELSE IF C="1"
00135                     THEN I3:=I3+1;
```

```
00139          ENDIF;
00140        ENDFOR;
00141
00142        IF B THEN RETURN I3;
00143             ELSE RETURN -I3;
00144           ENDIF;
00145      END
00146
00147    FUNCTION CHARTEXT(T:TEXT) RETURNS CHAR;
00148      BEGIN
00149        IF @T LENGTH = 0 THEN RETURN "@";ENDIF;
00150        RETURN @T GETCHAR(1);
00151      END
00152
00153    FUNCTION INTREAL(R:REAL) RETURNS INTEGER;
00154      VAR
00155        R1,R2,R3:REAL;
00156        B:BOOLEAN;
00157        I1,I2,I3,I4:INTEGER;
00158      END
00159      BEGIN
00160        IF R<0.
00161        THEN R1:=-R;
00162             B:=FALSE;
00163        ELSE R1:=R;
00164             B:=TRUE;
00165        ENDIF;
00166
00167        IF R1<1. THEN RETURN 0; ENDIF;
00168        IF R1>0.214748E+10 THEN
00169  $WRITER WRITE('*** REEL TROP GRAND POUR ETRE CONVERTI EN ENTIER');
00170           RETURN 0;   ENDIF;
00371
00172        FOR I2:=10 BY -1 WHILE I2>0
00173           I3:=1;
00174           R3:=1.;
00175           FOR I4:=1 BY 1 WHILE I4<=I2
00176                R3:=R3*10.;
00177                I3:=I3*10;
00178           ENDFOR;
00179
00180           WHILE R1>=R3
00181             R1:=R1-R3;
00182             I1:=I1+I3;
00183           ENDWHILE;
00184        ENDFOR;
00185
00186        IF B THEN RETURN I1; ELSE RETURN -I1; ENDIF;
00187      END
00188
00189    FUNCTION REALINT(I:INTEGER) RETURNS REAL;
00190      VAR
00191        R1,R2,R3,R4:REAL;
00192        B:BOOLEAN;
00193        I1,I2,I3,I4:INTEGER;
00194      END
00195      BEGIN
00196        IF I<0
00197        THEN I1:=-I1;
00198             B:=FALSE;
00199        ELSE I1:=I;
00200             B:=TRUE;
00201        ENDIF;
00202
00203        FOR I2:=10 BY -1 WHILE I2>0
00204           I3:=1;
00205           R3:=1.;
```



```
00207           FOR I4:=1 BY 1 WHILE I4<=I2
00208             R3:=R3*10.;
00209             I3:=I3*10;
00210           ENDFOR;
00211
00212           WHILE I1>I3
00213             R1:=R1+R3;
00214             I1:=I1-I3;
00215           ENDWHILE;
00216         ENDFOR;
00217
00218      IF B THEN RETURN R1 ; ELSE RETURN -R1; ENDIF;
00219    END
00220 END
EOF HIT
*
```

```
 f/jc:m:08:time/
00001*0,$1
00001  MODULE TIME FOR TIME ;
00002           SOCW
00003 B1        EQU       X'4000'
00004 B7        EQU       X'1C000'
00005 LIAI      S:SIN,2   X'01'
00006 B         S:SIN,1   X'680'
00007 BAL       S:SIN,0   X'6A'
00008 LW        S:SIN,0   X'32'
00009 AI        S:SIN,2   X'20'
00010 STW       S:SIN,0   X'35'
00011 LI        S:SIN,2   X'22'
00012 CAL1      S:SIN,0   X'04'
00013 LCFI      S:SIN,0   X'02'
00014 STM       S:SIN,0   X'2B'
00015 TEST      DATA      0
00016           DATA      0
00017           DATA      0
00018           LIAI,1    -($+1)
00019           B         B1!ABSVAL(MTIME)
00020 MTIME     BAL,14    *X'3FFE'
00021           LW,5      60-16,1
00022           LI,6      20
00023           STW,6     *5
00024           AI,5      1
00025           LCFI,2    0
00026           STM,0     X'3E02'
00027           CAL1,0    X'25'
00028           B         *X'3FFD'
00029           END
EOF HIT
*
```

```
 f/jc:m:09:livres/
00001*0,$1
00001 MODULE LIVRES FOR LIVRES;
00002   RESOURCE NF:FSEQ;
00003   END
00004   PROC CREER;
00005     BEGIN
00006       @NF CRE(´ETL´,44,1024,´GEDRBA08-LIV´); @NF OPENP;
00007     END
00008   PROC AJOUTER(E:LIVRE) ->B:BOOLEAN;
00009     VAR T:TEXT;
00010     END
00011     BEGIN
00012       @E OUT ->T; @NF CLEND;
00013       @NF OPENP; @NF PUT(T); B:=TRUE;
00014     END
00015   PROC SUPPRIME(I:INTEGER);
00016     VAR E:LIVRE; B:BOOLEAN;
00017     END
00018     BEGIN
00019       $LIVRES ACELEMNO(I) ->E,B;
00020       IF B THEN @NF DELETE; ENDIF;
00021     END
00022   PROC ACELEMNO(I:INTEGER) ->E:LIVRE, B:BOOLEAN;
00023     VAR E1:LIVRE; B1,B2:BOOLEAN;
00024     END
00025     BEGIN
00026       $LIVRES PREMIER ->E1,B1;
00027       IF I=@E1 CLE THEN B2:=TRUE; ELSE B2:=FALSE; ENDIF;
00028       WHILE (B1 AND NOT B2);
00029         $LIVRES SUIVANT ->E1,B1;
00030         IF I=@E1 CLE THEN B2:=TRUE; ELSE B2:=FALSE; ENDIF;
00031       ENDWHILE;
00032       IF NOT B2 THEN $WRITER WRITE(´L ELEMENT CHERCHE N EXISTE PAS´);
00033       ENDIF;
00034       B:=B2; E:=E1;
00035     END
00036   PROC LISTCOLL;
00037     VAR T:TEXT; B:BOOLEAN;
00038     END
00039     BEGIN
00040       @NF CLBEGIN; @NF OPENG;
00041       $WRITER WRITE(´LISTE DES ELEMENTS´); @NF GET ->T,B;
00042       WHILE B; $WRITER WRITE(T); @NF GET ->T,B;
00043       ENDWHILE;
00044       $WRITER WRITE(´FIN DE LA LISTE DES ELEMENTS´);
00045     END
00046   PROC PREMIER ->E:LIVRE, B:BOOLEAN;
00047     VAR T:TEXT; B1:BOOLEAN;
00048     END
00049     BEGIN
00050       @NF CLBEGIN; @NF OPENG; @NF GET ->T,B1;
00051       IF B1 THEN @E IN(T); ENDIF; B:=B1;
00052     END
00053   PROC SUIVANT ->E:LIVRE, B:BOOLEAN;
00054     VAR T:TEXT; B1:BOOLEAN;
00055     END
00056    BEGIN
00057       @NF GET ->T,B1;
00058       IF B1 THEN @E IN(T); ENDIF; B:=B1;
00059     END
```



```
00060   PROC LOGIN;
00061     BEGIN
00062       @NF CRE(´ETL´,44,1024,´GEDRBA08-LIV´);
00063       @NF OPENG; @NF CLEND;  @NF OPENP;
00064     END
00065   PROC LOGOUT;
00066     BEGIN
00067       @NF CLBEGIN;
00068     END
00069 END
EOF HIT
*
```

```
 f/jc:m:09:livres1/
00001*0,$1
00001 MODULE LIVRES1  FOR LIVRES   ;
00002
00003 RESOURCE NF:FIND;
00004 END
00005 PROC CREER;
00006 VAR T:TEXT; B:BOOLEAN;
00007 END
00008 BEGIN
00009 @NF CRE( ´ETL´ , 44 ,1024, ´GEDRBA08-LIV´ , 3 , 1 );
00010 @NF OPENP; T:=´´; @T AJUST( 44 );
00011 @T SUBD($CONV TEXTINT(0), 1 +1, 1 + 3 );
00012 @NF PUT(T) ->B;
00013 END
00014 PROC AJOUTER(E: LIVRE ) ->B:BOOLEAN;
00015 VAR T:TEXT;
00016 END
00017 BEGIN
00018 @E OUT -> T; @NF CLBEGIN; @NF OPENU;
00019 @NF PUTN(T) -> B;
00020 END
00021 PROC SUPPRIME(I:INTEGER);
00022 VAR E: LIVRE ; B:BOOLEAN;
00023 END
00024 BEGIN
00025 $ LIVRES   ACELEMNO (I) ->E,B;
00026 IF B THEN @NF DELETE; ENDIF;
00027 END
00028 PROC ACELEMNO(I:INTEGER) -> E: LIVRE ,B:BOOLEAN;
00029 VAR T,T1:TEXT; B1:BOOLEAN; E1: LIVRE ;
00030 END
00031 BEGIN
00032 @NF CLBEGIN; @NF OPENU;
00033   T:=´´; T1:=´´; @T AJUST(44); @T1 AJUST(3);
00034   @T1 SUBD($CONV TEXTINT(I),1,3);
00035 @NF GETN(T1) ->T,B1;
00036 IF B1 THEN @E1 IN (T); E:=E1;
00037 ELSE $WRITER WRITE(´L ELEMENT CHERCHE N EXISTE PAS´); ENDIF;
00038 B:=B1;
00039 END
```

```
00040 PROC LISTCOLL;
00041 VAR C,T:TEXT; B:BOOLEAN;
00042 END
00043 BEGIN
00044 @NF CLBEGIN; @NF OPENU;
00045 C:=´´; T:=´´; @C AJUST(3); @T AJUST(44);
00046 @C SUBD($CONV TEXTINT(0),1, 3 );
00047 @NF GETN(C) ->T,B;
00048 @NF GET ->T,B;
00049 $WRITER WRITE(´LISTE DES ELEMENTS´);
00050 WHILE B; $WRITER WRITE(T);
00051 @NF GET ->T,B; ENDWHILE;
00052 $WRITER WRITE(´FIN DE LA LISTE DES ELEMENTS´);
00053 END
00054 PROC PREMIER ->E: LIVRE ,B:BOOLEAN;
00055 VAR T,C:TEXT; B1:BOOLEAN;
00056 END
00057 BEGIN
00058 @NF CLBEGIN; @NF OPENU;
00059   C:=´´; T:=´´; @C AJUST(3); @T AJUST(44);
00060 @C SUBD($CONV TEXTINT(0),1, 3 );
00061 @NF GETN(C) ->T,B1;
00062 @NF GET -> T,B1;
00063 IF B1 THEN @E IN (T); ENDIF;
00064 B:=B1;
00065 END
00066 PROC SUIVANT -> E: LIVRE ,B:BOOLEAN;
00067 VAR T:TEXT; B1:BOOLEAN;
00068 END
00069 BEGIN
00070   T:=´´; @T AJUST(44);
00071 @NF GET -> T,B1;
00072 IF B1 THEN @E IN (T); ENDIF;
00073 B:=B1;
00074 END
00075 PROC LOGIN;
00076 BEGIN
00077 @NF CRE( ´ETL´ , 44 ,1024, ´GEDRBA08-LIV´ , 3 , 1 );
00078 @NF OPENG; @NF CLEND; @NF OPENP;
00079 END
00080 PROC LOGOUT;
00081 BEGIN
00082 @NF CLBEGIN;
00083 END
00084 END
EOF HIT
*
```



```
 f/jc:m:0a:cherchs/
00001*0,$1
00001 MODULE CHERCHS FOR CHERCHS;
00002   RESOURCE NF:FSEQ;
00003   END
00004   PROC CREER;
00005     BEGIN
00006       @NF CRE(´ETC´,14,1024,´GEDRBA08-CHE´); @NF OPENP;
00007     END
00008   PROC AJOUTER(E:CHERCH) ->B:BOOLEAN;
00009     VAR T:TEXT;
00010     END
00011     BEGIN
00012       @E OUT ->T; @NF CLEND;
00013       @NF OPENP; @NF PUT(T); B:=TRUE;
00014     END
00015   PROC SUPPRIME(I:INTEGER);
00016     VAR E:CHERCH; B:BOOLEAN;
00017     END
00018     BEGIN
00019       $CHERCHS ACELEMNO(I) ->E,B;
00020       IF B THEN @NF DELETE; ENDIF;
00021     END
00022   PROC ACELEMNO(I:INTEGER) ->E:CHERCH, B:BOOLEAN;
00023     VAR E1:CHERCH; B1,B2:BOOLEAN;
00024     END
00025     BEGIN
00026       $CHERCHS PREMIER ->E1,B1;
00027       IF I=@E1 CLE THEN B2:=TRUE; ELSE B2:=FALSE; ENDIF;
00028       WHILE (B1 AND NOT B2);
00029         $CHERCHS SUIVANT ->E1,B1;
00030         IF I=@E1 CLE THEN B2:=TRUE; ELSE B2:=FALSE; ENDIF;
00031       ENDWHILE;
00032       IF NOT B2 THEN $WRITER WRITE(´L ELEMENT CHERCHE N EXISTE PAS´);
00033       ENDIF;
00034       B:=B2; E:=E1;
00035     END
00036   PROC LISTCOLL;
00037     VAR T:TEXT; B:BOOLEAN;
00038     END
00039     BEGIN
00040       @NF CLBEGIN; @NF OPENG;
00041       $WRITER WRITE(´LISTE DES ELEMENTS´); @NF GET ->T,B;
00042       WHILE B; $WRITER WRITE(T); @NF GET ->T,B;
00043       ENDWHILE;
00044       $WRITER WRITE(´FIN DE LA LISTE DES ELEMENTS´);
00045     END
00046   PROC PREMIER ->E:CHERCH, B:BOOLEAN;
00047     VAR T:TEXT; B1:BOOLEAN;
00048     END
00049     BEGIN
00050       @NF CLBEGIN; @NF OPENG; @NF GET ->T,B1;
00051       IF B1 THEN @E IN(T); ENDIF; B:=B1;
00052     END
00053   PROC SUIVANT ->E:CHERCH, B:BOOLEAN;
00054     VAR T:TEXT; B1:BOOLEAN;
00055     END
00056     BEGIN
00057       @NF GET ->T,B1;
00058       IF B1 THEN @E IN(T); ENDIF; B:=B1;
00059     END
```

```
00060   PROC LOGIN;
00061     BEGIN
00062       @NF CRE(´ETC´,14,1024,´GEDRBA08-CHE´);
00063       @NF OPENG; @NF CLEND;  @NF OPENP;
00064     END
00065   PROC LOGOUT;
00066     BEGIN
00067       @NF CLBEGIN;
00068     END
00069 END
EOF HIT
*
```

```
 f/jc:m:0a:cherchs1/
00001*0,$1
00001 MODULE CHERCHS1 FOR CHERCHS  ;
00002
00003 RESOURCE NF:FIND;
00004 END
00005 PROC CREER;
00006 VAR T:TEXT; B:BOOLEAN;
00007 END
00008 BEGIN
00009 @NF CRE( ´ETC´ , 14 ,1024, ´GEDRBA08-CHE´ , 3 , 1 );
00010 @NF OPENP; T:=´´; @T AJUST( 14 );
00011 @T SUBD($CONV TEXTINT(0), 1 +1, 1 + 3 );
00012 @NF PUT(T) ->B;
00013 END
00014 PROC AJOUTER(E: CHERCH ) ->B:BOOLEAN;
00015 VAR T:TEXT;
00016 END
00017 BEGIN
00018 @E OUT -> T; @NF CLBEGIN; @NF OPENU;
00019 @NF PUTN(T) -> B;
00020 END
00021 PROC SUPPRIME(I:INTEGER);
00022 VAR E: CHERCH ; B:BOOLEAN;
00023 END
00024 BEGIN
00025 $ CHERCHS  ACELEMNO (I) ->E,B;
00026 IF B THEN @NF DELETE; ENDIF;
00027 END
00028 PROC ACELEMNO(I:INTEGER) -> E: CHERCH ,B:BOOLEAN;
00029 VAR T,T1:TEXT; B1:BOOLEAN; E1: CHERCH ;
00030 END
00031 BEGIN
00032 @NF CLBEGIN; @NF OPENU;
00033 T1:=´´; @T1 AJUST( 3 );
00034 @T1 SUBD($CONV TEXTINT(I), 1 +1, 1 + 3 );
00035 @NF GETN(T1) ->T,B1;
00036 IF B1 THEN @E1 IN (T); E:=E1;
00037 ELSE $WRITER WRITE(´L ELEMENT CHERCHE N EXISTE PAS´); ENDIF;
00038 B:=B1;
00039 END
```



```
00040 PROC LISTCOLL;
00041 VAR C,T:TEXT; B:BOOLEAN;
00042 END
00043 BEGIN
00044 @NF CLBEGIN; @NF OPENU;
00045 C:=´´; @C AJUST( 3 );
00046 @C SUBD($CONV TEXTINT(0),1, 3 );
00047 @NF GETN(C) ->T,B;
00048 @NF GET ->T,B;
00049 $WRITER WRITE(´LISTE DES ELEMENTS´);
00050 WHILE B; $WRITER WRITE(T);
00051 @NF GET ->T,B; ENDWHILE;
00052 $WRITER WRITE(´FIN DE LA LISTE DES ELEMENTS´);
00053 END
00054 PROC PREMIER ->E: CHERCH ,B:BOOLEAN;
00055 VAR T,C:TEXT; B1:BOOLEAN;
00056 END
00057 BEGIN
00058 @NF CLBEGIN; @NF OPENU;
00059 C:=´´; @C AJUST( 3 );
00060 @C SUBD($CONV TEXTINT(0),1, 3 );
00061 @NF GETN(C) ->T,B1;
00062 @NF GET -> T,B1;
00063 IF B1 THEN @E IN (T); ENDIF;
00064 B:=B1;
00065 END
00066 PROC SUIVANT -> E: CHERCH ,B:BOOLEAN;
00067 VAR T:TEXT; B1:BOOLEAN;
00068 END
00069 BEGIN
00070 @NF GET -> T,B1;
00071 IF B1 THEN @E IN (T); ENDIF;
00072 B:=B1;
00073 END
00074 PROC LOGIN;
00075 BEGIN
00076 @NF CRE( ´ETC´ , 14 ,1024, ´GEDRBA08-CHE´ , 3 , 1 );
00077 @NF OPENG; @NF CLEND; @NF OPENP;
00078 END
00079 PROC LOGOUT;
00080 BEGIN
00081 @NF CLBEGIN;
00082 END
00083 END
EOF HIT
*
```

```
CURRENT FILE : JC:M:0B:PRETS
00001*0,$1
00001 MODULE PRETS FOR PRETS;
00002   RESOURCE NF:FSEQ;TOTO:INTEGER;
00003     END
00004   PROC CREER;
00005     BEGIN
00006   @NF CRE('ETP',20,1024,'GEDRBA08-PRE'); @NF OPENP;
00007     END
00008   PROC AJOUTER(E:PRET) ->B:BOOLEAN;
00009     VAR T:TEXT;
00010     END
00011     BEGIN
00012       @E OUT ->T;  @NF CLEND;
00013   @NF OPENP; @NF PUT(T); B:=TRUE;
00014     END
00015   PROC SUPPRIME(I:INTEGER);
00016     VAR E:PRET; B:BOOLEAN;
00017     END
00018     BEGIN
00019       $PRETS ACELEMNO(I) ->E,B;
00020       IF B THEN  @NF DELETE; ENDIF;
00021     END
00022   PROC ACELEMNO(I:INTEGER) -> E:PRET, B:BOOLEAN;
00023     VAR J:INTEGER; END BEGIN
00024      FIRST E1 IN $PRETS SELECT
00025        WHERE @E1 NUMERO = I
00026        THEN  B:=TRUE; E:=E1;
00027        ELSE B:=FALSE;
00028      ENDFIRST;
00029    END
00030  PROC LISTCOLL;
00031    VAR T:TEXT; B:BOOLEAN;
00032    END
00033    BEGIN
00034      @NF CLBEGIN; @NF OPENG;
00035      $WRITER WRITE('LISTE DES ELEMENTS');
00036      @NF GET ->T,B;
00037      WHILE B; $WRITER WRITE(T); @NF GET ->T,B;
00038      ENDWHILE;
00039      $WRITER WRITE('FIN DE LA LISTE DES ELEMENTS');
00040    END
00041  ITER SELECT YIELDS PRET;
00042    VAR T:TEXT; B:BOOLEAN; E:PRET;
00043    END
00044    BEGIN
00045    @NF CLBEGIN; @NF OPENU;
00046      @NF GET ->T,B;
00047      WHILE B; @E IN(T); YIELD E;
00048      @NF GET ->T,B;
00049      ENDWHILE;
00050    END
00051  PROC LOGIN;
00052    BEGIN
00053      @NF CRE('ETP',20,1024,'GEDRBA08-PRE');
00054      @NF OPENG; @NF CLEND; @NF OPENP;
00055    END
00056  PROC LOGOUT;
00057    BEGIN
00058      @NF CLBEGIN;
00059    END
00060 END
EOF HIT
*
```



```
f/jc:m:0c:demandes/
00001*0,$1
00001 MODULE DEMANDES FOR DEMANDES;
00002   RESOURCE NF:FSEQ;
00003     END
00004   PROC CREER;
00005     BEGIN
00006   @NF CRE('ETD',14,1024,'GEDRBA08-DEM'); @NF OPENP;
00007     END
00008   PROC AJOUTER(E:DEMANDE) ->B:BOOLEAN;
00009     VAR T:TEXT;
00010     END
00011     BEGIN
00012       @E OUT ->T;  @NF CLEND;
00013   @NF OPENP; @NF PUT(T); B:=TRUE;
00014     END
00015   PROC SUPPRIME(I:INTEGER);
00016     VAR E:DEMANDE; B:BOOLEAN;
00017     END
00018     BEGIN
00019       $DEMANDES ACELEMNO(I) ->E,B;
00020       IF B THEN  @NF DELETE; ENDIF;
00021     END
00022   PROC ACELEMNO(I:INTEGER) -> E:DEMANDE, B:BOOLEAN;
00023     VAR J:INTEGER; END BEGIN
00024       FIRST E1 IN $DEMANDES SELECT
00025         WHERE @E1 NUMERO = I
00026         THEN  B:=TRUE; E:=E1;
00027         ELSE B:=FALSE;
00028       ENDFIRST;
00029     END
Q0030   PROC LISTCOLL;
00031     VAR T:TEXT; B:BOOLEAN;
00032     END
00033     BEGIN
00034       @NF CLBEGIN; @NF OPENG;
00035       $WRITER WRITE('LISTE DES ELEMENTS');
00036       @NF GET ->T,B;
00037       WHILE B; $WRITER WRITE(T); @NF GET ->T,B;
00038       ENDWHILE;
00039       $WRITER WRITE('FIN DE LA LISTE DES ELEMENTS');
00040     END
00041   ITER SELECT YIELDS DEMANDE;
00042     VAR T:TEXT; B:BOOLEAN; E:DEMANDE;
00043     END
00044     BEGIN
00045     @NF CLBEGIN; @NF OPENU;
00046       @NF GET ->T,B;
00047       WHILE B; @E IN(T); YIELD E;
00048       @NF GET ->T,B;
00049       ENDWHILE;
00050     END
00051   PROC LOGIN;
00052     BEGIN
00053       @NF CRE('ETD',14,1024,'GEDRBA08-DEM');
00054       @NF OPENG; @NF CLEND; @NF OPENP;
00055     END
00056   PROC LOGOUT;
00057     BEGIN
00058       @NF CLBEGIN;
00059     END
00060 END
EOF HIT
*
```

```
 f/jc:m:0d:bib/
00001*0,$1
00001 MODULE BIB FOR BIB;
00002    PROC GO;
00003      VAR B:BOOLEAN; T:TEXT;
00004      END
00005      BEGIN  $LIVRES LOGIN; $CHERCHS LOGIN;
00006      $PRETS LOGIN; $DEMANDES LOGIN;
00007    B:=TRUE; WHILE B;
00008    $WRITER WRITE('QUE VOULEZ VOUS FAIRE?');
00009     $WRITER WRITE('FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES');
00010    $READER READ ->T,B;
00011    IF (@T EQ('?')) THEN
00012     $WRITER WRITE('LES QUESTIONS POSSIBLES SONT:');
00013    $WRITER WRITE('AJOUTER UN LIVRE');
00014    $WRITER WRITE('SUPPRIMER UN LIVRE');
00015    $WRITER WRITE('AJOUTER UN CHERCHEUR');
00016    $WRITER WRITE('SUPPRIMER UN CHERCHEUR');
00017    $WRITER WRITE('EMPRUNTER UN LIVRE');
00018    $WRITER WRITE('RESTITUER UN LIVRE');
00019      $WRITER WRITE('LISTER LES LIVRES');
00020      $WRITER WRITE('LISTER LES CHERCHEURS');
00021      $WRITER WRITE('LISTER LES PRETS');
00022      $WRITER WRITE('LISTER LES DEMANDES');
00023  $WRITER WRITE('LISTER LES LIVRES D AUTEUR DONNE');
00024   $WRITER WRITE('LISTER LES LIVRES DE SUJET DONNE');
00025   $WRITER WRITE('LISTER LES LIVRES PRETES ET DEMANDES POUR UN CHERCHEUR DONNE')
00026 $WRITER WRITE('DATE DE RESTITUTION DU LIVRE DONNE');
00027  $WRITER WRITE('PROCHAIN DEMANDEUR DU LIVRE DONNE');
00028 $WRITER WRITE('LIVRES NON RENDUS A LEUR DATE DE RESTITUTION');
00029  $WRITER WRITE('EMPRUNTEUR DU LIVRE DONNE');
00030 $WRITER WRITE('RIEN');
00031 T:='';  $WRITER WRITE(T); $WRITER WRITE(T);
00032 ELSIF (@T EQ('RIEN')) THEN B:=FALSE;
00033 ELSIF (@T EQ('AJOUTER UN LIVRE')) THEN $BIB AJLIVR;
00034 ELSIF (@T EQ('SUPPRIMER UN LIVRE')) THEN $BIB SUPLIVR;
00035 ELSIF (@T EQ('AJOUTER UN CHERCHEUR')) THEN $BIB AJCHER;
00036 ELSIF (@T EQ('SUPPRIMER UN CHERCHEUR')) THEN $BIB SUPCHER;
00037 ELSIF (@T EQ('EMPRUNTER UN LIVRE')) THEN $BIB EMPRLIVR;
00038 ELSIF (@T EQ('RESTITUER UN LIVRE')) THEN $BIB RESTLIVR;
00039    ELSIF (@T EQ('LISTER LES LIVRES')) THEN $LIVRES LISTCOLL;
00040    ELSIF (@T EQ('LISTER LES CHERCHEURS')) THEN $CHERCHS LISTCOLL;
00041    ELSIF (@T EQ('LISTER LES PRETS')) THEN $PRETS LISTCOLL;
00042    ELSIF (@T EQ('LISTER LES DEMANDES')) THEN $DEMANDES LISTCOLL;
00043 ELSIF (@T EQ('LISTER LES LIVRES D AUTEUR DONNE'))
00044    THEN $BIB P1;
00045    ELSIF (@T EQ('LISTER LES LIVRES DE SUJET DONNE')) THEN $BIB P2;
00046 ELSIF (@T EQ('LISTER LES LIVRES PRETES ET DEMANDES POUR UN CHERCHEUR DONNE'))
00047    THEN $BIB P3;
00048 ELSIF (@T EQ('DATE DE RESTITUTION DU LIVRE DONNE'))
00049    THEN $BIB P4;
00050 ELSIF (@T EQ('PROCHAIN DEMANDEUR DU LIVRE DONNE')) THEN $BIB P5;
00051 ELSIF (@T EQ('LIVRES NON RENDUS A LEUR DATE DE RESTITUTION')) THEN $BIB P6;
00052 ELSIF (@T EQ('EMPRUNTEUR DU LIVRE DONNE'))
00053  THEN $BIB P7;  ENDIF;
00054   ENDWHILE;
00055  $LIVRES LOGOUT; $CHERCHS LOGOUT;
00056   $PRETS LOGOUT; $DEMANDES LOGOUT;
00057 END
```



```
00058   PROC AJLIVR;
00059    VAR T,S,A:TEXT; I:INTEGER; B:BOOLEAN; LI:LIVRE;
00060  END
00061   BEGIN
00062    $WRITER WRITE('NUMERO DU LIVRE?');
00063   $READER READ ->T,B; I:=$CONV INTTEXT(T);
00064   $WRITER WRITE('SUJET DU LIVRE?');
00065   $READER READ ->S,B;
00066   $WRITER WRITE('TITRE DU LIVRE?');
00067   $READER READ ->T,B;
00068   $WRITER WRITE('AUTEUR DU LIVRE?');
00069   $READER READ ->A,B;
00070   @LI CREER(I,S,T,A); $LIVRES AJOUTER(LI) ->B;
00071 END
00072   PROC SUPLIVR;
00073   VAR T:TEXT; B,B1:BOOLEAN; NO:INTEGER;
00074    END
00075   BEGIN
00076   $WRITER WRITE('NUMERO DU LIVRE?');
00077   $READER READ ->T,B; NO:=$CONV INTTEXT(T);
00078   $LIVRES SUPPRIME(NO);
00079    FIRST P IN $PRETS SELECT WHERE @P NUMERO = NO
00080    THEN B1:=TRUE; ELSE B1:=FALSE; ENDFIRST;
00081    IF B1 THEN $PRETS SUPPRIME(NO); ENDIF;
00082   B:=TRUE; WHILE B;
00083   FIRST D IN $DEMANDES SELECT
00084     WHERE @D NUMERO = NO THEN B:=TRUE;
00085     ELSE B:=FALSE; ENDFIRST;
00086    IF B THEN $DEMANDES SUPPRIME(NO); ENDIF; ENDWHILE;
00087 END
00088  PROC AJCHER;
00089   VAR T:TEXT; I:INTEGER; B:BOOLEAN; CH:CHERCH;
00090   END
00091    BEGIN  $WRITER WRITE('NUMERO DU CHERCHEUR?');
00092  $READER READ ->T,B; I:=$CONV INTTEXT(T);
00093  $WRITER WRITE('NOM DU CHERCHEUR?');
00094  $READER READ ->T,B; @CH CREER(I,T);
00095   $CHERCHS AJOUTER(CH) ->B;
00096   END
00097   PROC SUPCHER;
00098   VAR T:TEXT; B,B1,B2:BOOLEAN; N,NO:INTEGER;
00099    CH,CH1:CHERCH;LI:LIVRE; DA,DAD:DATE;
00100   P1:PRET;
00101   END
00102   BEGIN $WRITER WRITE('NUMERO DU CHERCHEUR?');
00103   $READER READ ->T,B; NO:=$CONV INTTEXT(T);
00104    $CHERCHS ACELEMNO(NO) -> CH,B;
00105    B1:=TRUE;  WHILE B1;
00106    FIRST P IN $PRETS SELECT WHERE @CH EQ(@P CHERCH)
00107    THEN  N:=@P NUMERO;  B1:=TRUE;
00108    ELSE B1:=FALSE;
00109    ENDFIRST;
00110    IF B1 THEN $PRETS SUPPRIME(N);
00111      FIRST D IN $DEMANDES SELECT WHERE
00112      @D NUMERO=N THEN B:=TRUE;
00113     CH1:=@D CHERCH;  ELSE B:=FALSE;
00114     ENDFIRST;
00115    IF B THEN $DEMANDES SUPPRIME(N);
00116     @DA CREDATJ;  $LIVRES ACELEMNO(N)->LI,B;
00117    @DAD CREDAT(30);
00118    @P1 CREER(LI,CH1,DA,DAD);
00119    $PRETS AJOUTER(P1) ->B2;  ENDIF; ENDIF;
00120    ENDWHILE;
00121    B:=TRUE;  WHILE(B);
00122    FIRST D IN $DEMANDES SELECT WHERE @CH EQ(@D CHERCH)
00123   THEN B:=TRUE;  N:=@D NUMERO;
00124   ELSE B:=FALSE;  ENDFIRST;
```

```
00127   PROC EMPRLIVR;
00128     VAR T:TEXT; B,B1,B2:BOOLEAN; NL,NC:INTEGER;
00129       LI:LIVRE; CH:CHERCH; P:PRET; D:DEMANDE;
00130       DA,DAD:DATE;
00131     END
00132     BEGIN
00133      $WRITER WRITE('NUMERO DU LIVRE?');
00134       $READER READ ->T,B; NL:=$CONV INTTEXT(T);
00135       $WRITER WRITE('NUMERO DU CHERCHEUR?');
00136       $READER READ ->T,B; NC:=$CONV INTTEXT(T);
00137       $LIVRES ACELEMNO(NL) ->LI,B;
00138       $CHERCHS ACELEMNO(NC) -> CH,B1;
00139    IF B AND B1 THEN
00140       $PRETS ACELEMNO(NL) ->P,B2;
00141       @DA CREDATJ;
00142       IF  B2 THEN $WRITER WRITE('LE LIVRE EST DEJA SORTI --> DEMANDE');
00143       @D CREER(LI,CH,DA); $DEMANDES AJOUTER(D) ->B;
00144       ELSE @DAD CREDAT(30); @P CREER(LI,CH,DA,DAD);
00145       $PRETS AJOUTER(P)->B;
00146       $WRITER WRITE('LE LIVRE N EST PAS ENCORE SORTI --> PRET');
00147       ENDIF; ENDIF; END
00148   PROC RESTLIVR;
00149     VAR T:TEXT; B,B1,B2:BOOLEAN; NL:INTEGER;
00150         P:PRET;    LI:LIVRE; CH:CHERCH; DA,DAD:DATE;
00151     END
00152     BEGIN
00153       $WRITER WRITE('NUMERO DU LIVRE?');
00154       $READER READ ->T,B; NL:=$CONV INTTEXT(T);
00155       $PRETS ACELEMNO(NL) ->P,B1;
00156       IF  B1 THEN $PRETS SUPPRIME(NL);
00157       FIRST D IN $DEMANDES SELECT WHERE @D NUMERO = NL
00158   THEN B2:=TRUE; CH:=@D CHERCH; ELSE B2:=FALSE; ENDFIRST;
00159   IF B2 THEN $DEMANDES SUPPRIME(NL);
00160       @DA CREDATJ; $LIVRES ACELEMNO(NL) ->LI,B;
00161     @DAD CREDAT(30);
00162       @P CREER(LI,CH,DA,DAD);
00163       $PRETS AJOUTER(P) ->B;
00164   ENDIF; ENDIF;
00165   END
00166   PROC P1;
00167     VAR T1,T,A:TEXT; B:BOOLEAN; LI:LIVRE;
00168     END
00169     BEGIN
00170       $WRITER WRITE('NOM DE L AUTEUR?');
00171    $READER READ ->T,B; @T AJUST(10); $LIVRES PREMIER ->LI,B;
00172       WHILE B;
00173       A:=@LI AUTEUR; IF @A EQ(T) THEN
00174       @LI OUT ->T1; $WRITER WRITE(T1); ENDIF;
00175       $LIVRES SUIVANT ->LI,B; ENDWHILE;
00176   END
00177   PROC P2;
00178     VAR T,T1,S:TEXT; B:BOOLEAN; LI:LIVRE;
00179     END
00180     BEGIN
00181       $WRITER WRITE('SUJET DU LIVRE?');
00182    $READER READ ->T,B; @T AJUST(10); $LIVRES PREMIER ->LI,B;
00183       WHILE B; S:=@LI SUJET;
00184         IF @S EQ(T) THEN @LI OUT ->T1;
00185         $WRITER WRITE(T1); ENDIF;
00186       $LIVRES SUIVANT ->LI,B; ENDWHILE;
00187     END
```



```
00188   PROC P3;
00189     VAR NC:INTEGER; T,T1:TEXT;
00190       LI:LIVRE; CH:CHERCH; B:BOOLEAN;
00191     END
00192     BEGIN
00193       $WRITER WRITE('NUMERO DU CHERCHEUR?');
00194       $READER READ ->T,B; NC:=$CONV INTTEXT(T);
00195       $CHERCHS ACELEMNO(NC) ->CH,B;
00196       FOREACH P IN $PRETS SELECT WHERE @CH EQ(@P CHERCH)
00197       THEN LI:=@P LIVRE; @LI OUT ->T1;
00198       @T1 CONTEXT(' P'); $WRITER WRITE(T1);
00199       ENDFOREC;
00200       FOREACH D IN $DEMANDES SELECT WHERE @CH EQ(@D CHERCH)
00201       THEN LI:=@D LIVRE; @LI OUT ->T1;
00202       @T1 CONTEXT(' D');
00203       $WRITER WRITE(T1);
00204       ENDFOREC;
00205   END
00206   PROC P4;
00207     VAR T:TEXT; B,B1:BOOLEAN; NL:INTEGER; P:PRET;
00208       D:DATE;
00209     END
00210     BEGIN
00211       $WRITER WRITE('NUMERO DU LIVRE?');
00212       $READER READ ->T,B; NL:=$CONV INTTEXT(T);
00213       $PRETS ACELEMNO(NL) ->P,B1;
00214       IF NOT B1 THEN $WRITER WRITE('LE LIVRE N EST PAS SORTI');
00215       ELSE D:=@P DATEDU;
00216       @D OUT ->T; $WRITER WRITE(T);
00217        ENDIF;
00218   END
00219   PROC P5;
00220     VAR T:TEXT; B,B1:BOOLEAN; NL:INTEGER; D:DEMANDE;
00221       CH:CHERCH;
00222     END
00223     BEGIN
00224       $WRITER WRITE('NUMERO DU LIVRE?');
00225       $READER READ ->T,B; NL:=$CONV INTTEXT(T);
00226       $DEMANDES ACELEMNO(NL) ->D,B1;
00227       IF NOT B1 THEN $WRITER WRITE('PAS DE DEMANDE');
00228       ELSE CH:=@D CHERCH; @CH OUT ->T;
00229       $WRITER WRITE(T); ENDIF;
00230   END
00231   PROC P6;
00232     VAR T:TEXT; B:BOOLEAN; LI:LIVRE; D1:DATE; N:INTEGER;
00233     END
00234     BEGIN
00235       @D1 CREDATJ;
00236       FOREACH P IN $PRETS SELECT WHERE @D1 GT(@P DATEDU)
00237       THEN LI:=@P LIVRE; @LI OUT ->T;
00238 $WRITER WRITE(T);
00239       ENDFOREC;
00240   END
00241   PROC P7;
00242     VAR T,N:TEXT; B:BOOLEAN; NL:INTEGER; P:PRET;
00243       CH:CHERCH;
00244     END
00245     BEGIN
00246       $WRITER WRITE('NUMERO DU LIVRE?');
00247       $READER READ ->T,B; NL:=$CONV INTTEXT(T);
00248       $PRETS ACELEMNO(NL) ->P,B;
00249       IF NOT B THEN $WRITER WRITE('LE LIVRE N EST PAS SORTI');
00250       ELSE CH:=@P CHERCH; @CH OUT ->N;
00251       $WRITER WRITE(N); ENDIF;
00252     END
00253 END
```

```
 f/jc:m:07:u1/
00001*0,$1
00001  MODULE U1 FOR MAIN;
00002   PROC START;
00003     VAR T:TEXT;
00004     END
00005     BEGIN
00006       $DEMANDES LOGIN;
00007       FIRST D IN $DEMANDES SELECT
00008         WHERE @D NUMERO = 1
00009         THEN @D OUT ->T; $WRITER WRITE(T);
00010         ELSE $WRITER WRITE('PAS DE DEMANDEUR DU LIVRE 1');
00011       ENDFIRST;
00012       $DEMANDES LOGOUT;
00013     END
EOF HIT
*
```

```
 f/jc:m:07:u2/
00001*0,$1
00001  MODULE U2 FOR MAIN;
00002    PROC START;
00003     VAR T:TEXT;
00004     END
00005     BEGIN
00006       $DEMANDES LOGIN;
00007       FOREACH D IN $DEMANDES SELECT
00008         WHERE @D NUMERO = 1
00009         THEN @D OUT ->T; $WRITER WRITE(T);
00010       ENDFOREC;
00011       $DEMANDES LOGOUT;
00012     END
EOF HIT
*
```

```
 f/jc:m:0e:log/
00001*0,$1
00001  MODULE LOG FOR MAIN1;
00002   PROC START;
00003     BEGIN
00004 $LIVRES CREER; $LIVRES LOGOUT;
00005 $CHERCHS CREER; $CHERCHS LOGOUT;
00006 $DEMANDES CREER; $DEMANDES LOGOUT;
00007 $PRETS CREER; $PRETS LOGOUT;
00008 END
EOF HIT
*
```


• machines génériques

```
 typ
STATION?
```

```
      T Y P   A   V O T R E   S E R V I C E

***** R E A L I S A T I O N ***************************************
*                                                                 *
*******************************************************************
*                 *                *                *             *
*  COMPILATEUR ATM * GENERICITE  * OBJETS CONSERVES * CONNECTEUR  *
*                 *                *                *             *
*******************************************************************
*                 *                *                *             *
* HENRY P.    9-79 * PROCH K.    * CHABRIER J.      * HENRY P.    *
* MINOT R.    3-79 *       1-80  *           10-79  *       9-79  *
* TOUNSI N.   1-80 *                *                *             *
*                 *                *                *             *
*******************************************************************

SI RIEN N'EST SPECIFIE
A TOUTES LES QUESTIONS REPONDEZ PAR "OUI" OU PAR "NON"
REPONDEZ "?" POUR OBTENIR DES EXPLICATIONS
UN RETOUR CHARIOT PERMET DE REMONTER D'UN NIVEAU.

INDIQUEZ L'ENVIRONNEMENT(DE 0 A 3 CAR.) : jc


INDIQUEZ LE TRAITEMENT PAR
"TYPE","MACHINE","MODULE","CAPSULE",
"LIST","INIT","EXECUT",
"UNITGEN","GENERER","LISTGEN","EDITGEN" :listgen



VOULEZ VOUS LES RESULTATS
AU TERMINAL? FRAPPEZ T
A L'IMPRIMANTE? FRAPPEZ I
PAS DE RESULTATS EXTERNES? FRAPPEZ S :t
        LISTE DES UNITES GENERIQUES
        MACHGEN BIDON
        MACHGEN COLL
        MODGEN COLL     FOR COLL
        MODGEN XBIDON   FOR BIDON



INDIQUEZ LE TRAITEMENT PAR
"TYPE","MACHINE","MODULE","CAPSULE",
"LIST","INIT","EXECUT",
"UNITGEN","GENERER","LISTGEN","EDITGEN" :editgen
```

```
VOULEZ VOUS LES RESULTATS
AU TERMINAL? FRAPPEZ T
A L'IMPRIMANTE? FRAPPEZ I
PAS DE RESULTATS EXTERNES? FRAPPEZ S :t
INDIQUEZ LA NATURE DE L'UNITE
"TYPEGEN","MACHGEN","CAPSGEN","MODGEN":machgen
INDIQUEZ LE NOM DE L'UNITE:coll
        MACHGEN COLL PARAM
          CONST TEXT ETOP,
          CONST INTEGER LGELEM,
          CONST TEXT NOM,
          TYPE ELEM(FUNCTION CLE RETURNS INTEGER;
                    CONSULT OUT -> TEXT;
                    BUILD IN(TEXT)),
         CONST INTEGER LGCLE,
         CONST INTEGER POSCLE  END;
           PROC CREER;
             # CREER LA COLLECTION VIDE #
           PROC AJOUTER(ELEM) -> BOOLEAN;
             # AJOUTER UN ELEMENT A LA COLLECTION #
           PROC SUPPRIME(INTEGER);
             # SUPPRIMER UN ELEMENT DE LA COLLECTION #
             # FIND --> E DE CLE I #
             # FSEQ --> E TEL QUE CLE(E) = I #
           PROC ACELEMNO(INTEGER) -> ELEM,BOOLEAN;
             # ACCES A UN ELEMENT DE LA COLLECTION #
             # FIND --> E DE CLE I #
             # FSEQ --> E TEL QUE CLE(E) = I #
           PROC LISTCOLL;
             # LISTER TOUTE LA COLLECTION #
           PROC PREMIER -> ELEM,BOOLEAN;
             # ACCES A UN ELEMENT #
             # FIND --> ELEMENT DE PLUS PETITE CLE #
             # FSEQ --> PREMIER ELEMENT CREE #
         PROC SUIVANT -> ELEM,BOOLEAN;
           # ACCES A L ELEMENT SUIVANT #
           # FIND --> CLE SUIVANTE #
           # FSEQ --> CREATION SUIVANTE #
         PROC LOGIN;
           # LOGIN DE LA COLLECTION #
         PROC LOGOUT;
           # LOGOUT DE LA COLLECTION #
    END
```


2.4. Création et utilisation du système de base de données

2.4.1. Manipulation paramétrée

```
!connect
* ACCOMPAGNATEUR LP80 VERSION 2 PRET 16*44*46*72 01*07*83
->c
CONNECTEUR: CONCU ET REALISE PAR HENRY P. SEPT. 1979
INDIQUEZ LE NOM DU CONTEXTE(0 A 3 CAR)
?jc
 00339: TIME := #0000005F4
JC : OK
?realiser $main avec log
REALISER $MAIN AVEC LOG
MODULE INEXISTANT : LOG
 00339: TIME := #0000003C
JC : OK
?realiser $main1 avec log
REALISER $MAIN1 AVEC LOG
 00339: TIME := #0000002A
JC : OK
?$main1 start
$MAIN1 START
 00339: TIME := #000002E4
JC : OK
?realiser $livres avec livres1
REALISER $LIVRES AVEC LIVRES1
 00339: TIME := #00000030
JC : OK
?realiser $cherchs avec cherchs1
REALISER $CHERCHS AVEC CHERCHS1
 00339: TIME := #00000030
JC : OK
?$bib go
$BIB GO
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
??
LES QUESTIONS POSSIBLES SONT:
AJOUTER UN LIVRE
SUPPRIMER UN LIVRE
AJOUTER UN CHERCHEUR
SUPPRIMER UN CHERCHEUR
EMPRUNTER UN LIVRE
RESTITUER UN LIVRE
LISTER LES LIVRES
LISTER LES CHERCHEURS
LISTER LES PRETS
LISTER LES DEMANDES
LISTER LES LIVRES D AUTEUR DONNE
LISTER LES LIVRES DE SUJET DONNE
LISTER LES LIVRES PRETES ET DEMANDES POUR UN CHERCHEUR DONNE
DATE DE RESTITUTION DU LIVRE DONNE
PROCHAIN DEMANDEUR DU LIVRE DONNE
LIVRES NON RENDUS A LEUR DATE DE RESTITUTION
EMPRUNTEUR DU LIVRE DONNE
RIEN


QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?lister les livres
LISTE DES ELEMENTS
FIN DE LA LISTE DES ELEMENTS
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?
```



```
 ajouter un livre
NUMERO DU LIVRE?
?2
SUJET DU LIVRE?
?info
TITRE DU LIVRE?
?vega
AUTEUR DU LIVRE?
?chabrier
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 ajouter un livre
NUMERO DU LIVRE?
?4
SUJET DU LIVRE?
?math
TITRE DU LIVRE?
?logique
AUTEUR DU LIVRE?
?martin
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 lister les livres
LISTE DES ELEMENTS
   2INFO      VEGA                CHABRIER
   4MATH      LOGIQUE             MARTIN
FIN DE LA LISTE DES ELEMENTS
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 lister les chercheurs
LISTE DES ELEMENTS
FIN DE LA LISTE DES ELEMENTS
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 ajouter un chercheur
NUMERO DU CHERCHEUR?
?1
NOM DU CHERCHEUR?
?jean
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?
```

```
ajouter un chercheur
NUMERO DU CHERCHEUR?
?3
NOM DU CHERCHEUR?
?didier
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 lister les chercheurs
LISTE DES ELEMENTS
   1JEAN
   3DIDIER
FIN DE LA LISTE DES ELEMENTS
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?emprunter un livre
NUMERO DU LIVRE?
?2
NUMERO DU CHERCHEUR?
?1
LE LIVRE N EST PAS ENCORE SORTI --> PRET
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 lister les prets
LISTE DES ELEMENTS
   2    1 1 78331 783
FIN DE LA LISTE DES ELEMENTS
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?lister les demandes
LISTE DES ELEMENTS
FIN DE LA LISTE DES ELEMENTS
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 emprunter un livre
NUMERO DU LIVRE?
?2
NUMERO DU CHERCHEUR?
?3
LE LIVRE EST DEJA SORTI --> DEMANDE
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?lister les demandes
LISTE DES ELEMENTS
   2    3 1 783
FIN DE LA LISTE DES ELEMENTS
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 lister les livres d auteur donne
NOM DE L AUTEUR?
?martin
   4MATH      LOGIQUE            MARTIN
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?
```



```
 lister les livres de sujet donne
SUJET DU LIVRE?
?info
   2INFO      VEGA               CHABRIER
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 prochain demandeur du livre donne
NUMERO DU LIVRE?
?2
   3DIDIER
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 emprunteur du livre donne
NUMERO DU LIVRE?
?2
   1JEAN
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 date de restitution du livre donne
NUMERO DU LIVRE?
?2
31 783
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 lister les livres pretes et demandes pour un chercheur donne
NUMERO DU CHERCHEUR?
?1
   2INFO      VEGA               CHABRIER   P
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 emprunter un livre
NUMERO DU LIVRE?
?4
NUMERO DU CHERCHEUR?
?3
LE LIVRE N EST PAS ENCORE SORTI --> PRET
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?


 lister les livres pretes et demandes pour un chercheur donne
NUMERO DU CHERCHEUR?
?3
   4MATH      LOGIQUE            MARTIN     P
   2INFO      VEGA               CHABRIER   D
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?rien
 00339: TIME := #000018FE
IC : OK
```

2.4.2. Manipulation à l'aide d'itérations abstraites

```
INDIQUEZ LE NOM DE LA MACHINE REALISEE
 PAR LE MODULE : main

INDIQUEZ LE NOM DU MODULE : u1

DESIREZ VOUS MODIFIER L'UNITE ? non

INDIQUEZ LE TRAITEMENT
PAR UNE CHAINE DE CARACTERES
COMPORTANT UNE COMBINAISON DE LETTRES C I T S L O M : lt
    1  MODULE U1 FOR MAIN;
    2   PROC START;
    3     VAR T:TEXT;
    4     END
    5     BEGIN
    6       $DEMANDES LOGIN;
    7       FIRST D IN $DEMANDES SELECT
    8         WHERE àD NUMERO = 1
    9         THEN àD OUT ->T; $WRITER WRITE(T);
   10         ELSE $WRITER WRITE('PAS DE DEMANDEUR DU LIVRE 1');
   11       ENDFIRST;
   12       $DEMANDES LOGOUT;
   13     END

DESIREZ VOUS MODIFIER L'UNITE ? non

INDIQUEZ LE TRAITEMENT
PAR UNE CHAINE DE CARACTERES
COMPORTANT UNE COMBINAISON DE LETTRES C I T S L O M :


INDIQUEZ LE TRAITEMENT PAR
"TYPE","MACHINE","MODULE","CAPSULE",
"LIST","INIT","EXECUT",
"UNITGEN","GENERER","LISTGEN","EDITGEN" :module



INDIQUEZ LE NOM DE LA MACHINE REALISEE
 PAR LE MODULE : main

INDIQUEZ LE NOM DU MODULE : u2

DESIREZ VOUS MODIFIER L'UNITE ? non

INDIQUEZ LE TRAITEMENT
PAR UNE CHAINE DE CARACTERES
COMPORTANT UNE COMBINAISON DE LETTRES C I T S L O M : lt
    1  MODULE U2 FOR MAIN;
    2    PROC START;
    3     VAR T:TEXT;
    4     END
    5     BEGIN
    6       $DEMANDES LOGIN;
    7       FOREACH D IN $DEMANDES SELECT
    8         WHERE àD NUMERO = 1
    9         THEN àD OUT ->T; $WRITER WRITE(T);
   10       ENDFOREC;
   11       $DEMANDES LOGOUT;
   12     END

DESIREZ VOUS MODIFIER L'UNITE ? non
```


• Exécution

```
 connect
* ACCOMPAGNATEUR LP80 VERSION 2 PRET 17*36*40*97 05*07*83
->c
CONNECTEUR: CONCU ET REALISE PAR HENRY P. SEPT. 1979
INDIQUEZ LE NOM DU CONTEXTE(0 A 3 CAR)
?jc
 00339: TIME := £00000736
JC : OK
?realiser $livres avec livres1
REALISER $LIVRES AVEC LIVRES1
 00339: TIME := £00000032
JC : OK
?realiser $cherchs avec cherchs1
REALISER $CHERCHS AVEC CHERCHS1
 00339: TIME := £00000030
JC : OK
?realiser $main1 avec log
REALISER $MAIN1 AVEC LOG
 00339: TIME := £00000032
JC : OK
?$main1 start
$MAIN1 START
 00339: TIME := £00000384
JC : OK
?realiser $main avec u1
REALISER $MAIN AVEC U1
 00339: TIME := £00000028
JC : OK
?$main start
$MAIN START
PAS DE DEMANDEUR DU LIVRE 1
 00339: TIME := £000000EE
JC : OK
?$bib go
$BIB GO
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?ajouter un livre
NUMERO DU LIVRE?
?1
SUJET DU LIVRE?
?info
TITRE DU LIVRE?
?vega
AUTEUR DU LIVRE?
?chabrier
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?ajouter un chercheur
NUMERO DU CHERCHEUR?
?2
NOM DU CHERCHEUR?
?jean
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
```

```
?ajouter un chercheur
NUMERO DU CHERCHEUR?
?4
NOM DU CHERCHEUR?
?didier
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?ajouter un chercheur
NUMERO DU CHERCHEUR?
?6
NOM DU CHERCHEUR?
?claude
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?emprunter un livre
NUMERO DU LIVRE?
?1
NUMERO DU CHERCHEUR?
?2
LE LIVRE N EST PAS ENCORE SORTI --> PRET
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?emprunter un livre
NUMERO DU LIVRE?
?1
NUMERO DU CHERCHEUR?
?4
LE LIVRE EST DEJA SORTI --> DEMANDE
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?emprunter un livre
NUMERO DU LIVRE?
?1
NUMERO DU CHERCHEUR?
?6
LE LIVRE EST DEJA SORTI --> DEMANDE
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?lister les demandes
LISTE DES ELEMENTS
   1   4 5 783
   1   6 5 783
FIN DE LA LISTE DES ELEMENTS
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?rien
 00339: TIME := £00000C74
JC : OK
?realiser $main avec u1
REALISER $MAIN AVEC U1
 00339: TIME := £0000002C
JC : OK
?$main start
$MAIN START
   1   4 5 783
 00339: TIME := £000000FC
JC : OK
```



```
 connect
* ACCOMPAGNATEUR LP80 VERSION 2 PRET 17*41*31*09 05*07*83
->c
CONNECTEUR: CONCU ET REALISE PAR HENRY P. SEPT. 1979
INDIQUEZ LE NOM DU CONTEXTE(0 A 3 CAR)
?jc
 00339: TIME := £000007A2
JC : OK
?realiser $livres avec livres1
REALISER $LIVRES AVEC LIVRES1
 00339: TIME := £00000032
JC : OK
?realiser $cherchs avec cherchs1
REALISER $CHERCHS AVEC CHERCHS1
 00339: TIME := £00000030
JC : OK
?$bib go
$BIB GO
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?lister les demandes
LISTE DES ELEMENTS
   1   4 5 783
   1   6 5 783
FIN DE LA LISTE DES ELEMENTS
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?rien
 00339: TIME := £00000468
JC : OK
?realiser $main avec u2
REALISER $MAIN AVEC U2
 00339: TIME := £0000002E
JC : OK
?$main start
$MAIN START
   1   4 5 783
   1   6 5 783
 00339: TIME := £0000019A
JC : OK
?end
END
```

2.4.3. Exécution avec trace de la fabrication dynamique d'un programme

```
 connect
* ACCOMPAGNATEUR LP80 VERSION 2 PRET 16*11*19*56 04*07*83
->c
CONNECTEUR: CONCU ET REALISE PAR HENRY P. SEPT. 1979
INDIQUEZ LE NOM DU CONTEXTE(0 A 3 CAR)
?jc
 00339: TIME := #000005D2
JC : OK
?realiser $main1 avec log
(REALISER $MAIN1 AVEC LOG
COMMANDE INCONNUE
COMMANDES POSSIBLES :
$<NOM DE MACHINE><NOM D'OPERATION>
MAP      : EDITION DE L'IMPLANTATION
NOMAP    : NON EDITION DE L'IMPLANTATION
INIT     : DECONNECTE TOUT
END      : SORTIE DU CONNECTEUR

REALISER $<MACHINE> AVEC <MODULE>
REALISER @<TYPE> AVEC <CAPSULE>
 00339: TIME := #000000A4
JC : OK
```



```
connect
* ACCOMPAGNATEUR LP80 VERSION 2 PRET 16*25*49*67 04*07*83
->c
CONNECTEUR: CONCU ET REALISE PAR HENRY P. SEPT. 1979
INDIQUEZ LE NOM DU CONTEXTE(0 A 3 CAR)
?jc
 00339: TIME := #000005B4
JC : OK
?realiser $livres avec livres1
REALISER $LIVRES AVEC LIVRES1
 00339: TIME := #0000002E
JC : OK
?realiser $cherchs avec cherchs1
REALISER $CHERCHS AVEC CHERCHS1
 00339: TIME := #00000034
JC : OK
?map
MAP
 00339: TIME := #00000028
JC : OK
?realiser $main1 avec log
REALISER $MAIN1 AVEC LOG
 00339: TIME := #0000002E
JC : OK
?$main1 start
$MAIN1 START
----------> $ MAIN1 START  : CHARGEMENT DU MODULE LOG DE L'ADRESSE DE7F A DEA8
----------> $ LIVRES CREER  : CHARGEMENT DU MODULE LIVRES1 DE L'ADRESSE DEA9 A E1B1
----------> @ FIND CREATE  : CHARGEMENT DE LA CAPSULE FIND DE L'ADRESSE E1B2 A E297
----------> @ TEXT CREATE  : CHARGEMENT DE LA CAPSULE MONTEXT DE L'ADRESSE E299 A E74D
----------> @ LIVRE CREATE  : CHARGEMENT DE LA CAPSULE LIVRE DE L'ADRESSE E74E A E900
----------> $ CONV TEXTINT  : CHARGEMENT DU MODULE CONV DE L'ADRESSE E901 A EE04
----------> $ CHERCHS CREER  : CHARGEMENT DU MODULE CHERCHS1 DE L'ADRESSE EE05 A F0DB
----------> @ CHERCH CREATE  : CHARGEMENT DE LA CAPSULE CHERCH DE L'ADRESSE F0DD A F21F
----------> $ DEMANDES CREER  : CHARGEMENT DU MODULE DEMANDES DE L'ADRESSE F220 A F3E5
----------> @ FSEQ CREATE  : CHARGEMENT DE LA CAPSULE FSEQ DE L'ADRESSE F3E6 A F49B
----------> @ DEMANDE CREATE  : CHARGEMENT DE LA CAPSULE DEMANDE DE L'ADRESSE F49D A F688
----------> @ DATE CREATE  : CHARGEMENT DE LA CAPSULE DATE1 DE L'ADRESSE F689 A FA0A
----------> $ PRETS CREER  : CHARGEMENT DU MODULE PRETS DE L'ADRESSE FA0B A FBD6
----------> @ PRET CREATE  : CHARGEMENT DE LA CAPSULE PRET DE L'ADRESSE FBD9 A FE20
 00339: TIME := #00000480
JC : OK
?$bib go
$BIB GO
----------> $ BIB GO  : CHARGEMENT DU MODULE BIB DE L'ADRESSE FE21 A 10B7F
----------> $ WRITER WRITE  : CHARGEMENT DU MODULE ECRIRE DE L'ADRESSE 10B80 A 10B94
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
----------> $ READER READ  : CHARGEMENT DU MODULE LIRE DE L'ADRESSE 10B95 A 10BC6
?ajouter un livre
NUMERO DU LIVRE?
?2
SUJET DU LIVRE?
?info
TITRE DU LIVRE?
?vega
AUTEUR DU LIVRE?
?chabrier
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?lister les livres
LISTE DES ELEMENTS
   2INFO      VEGA                CHABRIER
FIN DE LA LISTE DES ELEMENTS
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?rien
```

```
 connect
* ACCOMPAGNATEUR LP80 VERSION 2 PRET 16*29*45*93 04*07*83
->c
CONNECTEUR: CONCU ET REALISE PAR HENRY P. SEPT. 1979
INDIQUEZ LE NOM DU CONTEXTE(0 A 3 CAR)
?jc
 00339: TIME := #0000078A
JC : OK
?map
MAP
 00339: TIME := #00000024
JC : OK
?realiser $livres avec livres1
REALISER $LIVRES AVEC LIVRES1
 00339: TIME := #00000032
JC : OK
?realiser $cherchs avec cherchs1
REALISER $CHERCHS AVEC CHERCHS1
 00339: TIME := #00000032
JC : OK
?$bib go
$BIB GO
----------> $ BIB GO  : CHARGEMENT DU MODULE BIB DE L'ADRESSE DE7F A EBDD
----------> @ TEXT CREATE  : CHARGEMENT DE LA CAPSULE MONTEXT DE L'ADRESSE EBDE A F09
----------> @ LIVRE CREATE  : CHARGEMENT DE LA CAPSULE LIVRE DE L'ADRESSE F093 A F245
----------> @ CHERCH CREATE  : CHARGEMENT DE LA CAPSULE CHERCH DE L'ADRESSE F246 A F3
----------> @ DATE CREATE  : CHARGEMENT DE LA CAPSULE DATE1 DE L'ADRESSE F389 A F70A
----------> @ PRET CREATE  : CHARGEMENT DE LA CAPSULE PRET DE L'ADRESSE F70B A F952
----------> @ DEMANDE CREATE  : CHARGEMENT DE LA CAPSULE DEMANDE DE L'ADRESSE F953 A
----------> $ LIVRES LOGIN  : CHARGEMENT DU MODULE LIVRES1 DE L'ADRESSE FB3F A FE47
----------> @ FIND CREATE  : CHARGEMENT DE LA CAPSULE FIND DE L'ADRESSE FE48 A FF2D
----------> $ CHERCHS LOGIN  : CHARGEMENT DU MODULE CHERCHS1 DE L'ADRESSE FF2F A 1020
----------> $ PRETS LOGIN  : CHARGEMENT DU MODULE PRETS DE L'ADRESSE 10207 A 103D2
----------> @ FSEQ CREATE  : CHARGEMENT DE LA CAPSULE FSEQ DE L'ADRESSE 103D3 A 10488
----------> $ DEMANDES LOGIN  : CHARGEMENT DU MODULE DEMANDES DE L'ADRESSE 1048B A 10
----------> $ WRITER WRITE  : CHARGEMENT DU MODULE ECRIRE DE L'ADRESSE 10652 A 10666
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
----------> $ READER READ  : CHARGEMENT DU MODULE LIRE DE L'ADRESSE 10667 A 10698
?lister les livres
----------> $ CONV TEXTINT  : CHARGEMENT DU MODULE CONV DE L'ADRESSE 10699 A 10B9C
LISTE DES ELEMENTS
   2INFO      VEGA                CHABRIER
FIN DE LA LISTE DES ELEMENTS
QUE VOULEZ VOUS FAIRE?
FRAPPER ? SI VOUS VOULEZ CONNAITRE LES QUESTIONS POSSIBLES
?rien
 00339: TIME := #000005DA
JC : OK
?end
END
```

ANNEXE 2

VEGA

EXEMPLES

```
vega
SYSTEME?vega
SOURCE?esspilint
SAUVE?s
EXECUTION F4LIB VERSION V10-05 08*46*29*

?-tty§


-VEGA§
SYSTEM(PILINT)/
    SORTS(PILINT)/ IMPORTS(INT;BOOL)/
                                                    OKDEFINE
BUILD/

    PILINT(PVIDE<-PILINT)/
                                                    ISBUILD

    PILINT(EMPILER(PILINT,INT)<-PILINT)/
                                                    ISBUILD

OKOPNS/

    PILINT(DEPILER(PILINT)<-PILINT)/
                                                    ISOKOPNS

    PILINT(SOMMET(PILINT)<-INT)/
                                                    ISOKOPNS

    PILINT(ESTVIDE(PILINT)<-BOOL)/
                                                    ISOKOPNS

EROPNS/

    PILINT(SOMERR<-INT)/
                                                    ISEROPNS

    PILINT(PILERR<-PILINT)/
                                                    ISEROPNS

OKAXS/

    PILINT(DEPILER(EMPILER(*P,*I))=*P)/
                                                    ISOKAXS

    INT(SOMMET(EMPILER(*P,*I))=*I)/
                                                    ISOKAXS

    BOOL(ESTVIDE(PVIDE)=TRUE)/
                                                    ISOKAXS

    BOOL(ESTVIDE(EMPILER(*P,*I))=FALSE)/
                                                    ISOKAXS

ERAXS/

    PILINT(DEPILER(PVIDE)=PILERR)/
                                                    ISERAXS

    INT(SOMMET(PVIDE)=SOMERR)/
                                                    ISERAXS

END(PILINT)/

RUN(PILINT(PVIDE))?
```

```
                    PILINT:PVIDE

RUN(PILINT(EMPILER(EMPILER(PVIDE,1),2)))?
                    PILINT:EMPILER(EMPILER(PVIDE,1),2)

RUN(PILINT(EMPILER(DEPILER(EMPILER(PVIDE,1)),2)))?
                    PILINT:EMPILER(PVIDE,2)

RUN(PILINT(SOMMET(DEPILER(EMPILER(EMPILER(PVIDE,1),2)))))?
                    INT:1

RUN(PILINT(ESTVIDE(PVIDE)))?
                    BOOL:TRUE

RUN(PILINT(ESTVIDE(DEPILER(EMPILER(PVIDE,1)))))?
                    BOOL:TRUE

RUN(PILINT(ESTVIDE(EMPILER(PVIDE,1))))?
                    BOOL:FALSE

RUN(PILINT(SOMMET(PVIDE)))?
>>ERROR>>SOMERR                    INT:SOMERR

RUN(PILINT(DEPILER(PVIDE)))?
>>ERROR>>PILERR                    PILINT:PILERR

RUN(PILINT(EMPILER(PVIDE,
    PILINT(SOMMET(DEPILER(EMPILER(PVIDE,1)))))))?
>>ERROR>>SOMERR>>ERROR>>SOMERR                    PILINT:EMPILER(PVID
E,SOMERR)

RUN(PILINT(EMPILER(EMPILER(PVIDE,2),
    PILINT(SOMMET(EMPILER(PVIDE,3))))))?
                    PILINT:EMPILER(EMPILER(PVIDE,2),3)

INPUT/

?bye/
BYE/


*STOP* 0
!
```



```
vega
SYSTEME?vega
SOURCE?essfact
SAUVE?s
EXECUTION F4LIB VERSION V10-05 09*27*01*

?-tty§


-VEGA§
SYSTEM(FACT)/
     IMPORTS(INT;BOOL)/
                                                    OKDEFINE

OKOPNS/

   FACT(FACT(INT)<-INT)/
                                                    ISOKOPNS

EROPNS/

   FACT(ARGNEG<-INT)/
                                                    ISEROPNS

OKAXS/

   INT(FACT(*I)=ITE(INT(*I=0),1,
        INT(*I*FACT(*I-1))))/
                                                    ISOKAXS

ERAXS/

   INT(FACT(*I)=IT(INT(*I<0),ARGNEG))/
                                                    ISERAXS

END(FACT)/

RUN(FACT(0))?
                    INT:1

RUN(FACT(1))?
                    INT:1

RUN(FACT(3))?
                    INT:6

RUN(FACT(-3))?
>>ERROR>>ARGNEG                    INT:ARGNEG

RUN(INT(1+FACT(2)))?
                    INT:3

INPUT/

?bye/
BYE/


*STOP* 0
!
```

```
 vega
SYSTEME?vega
SOURCE?essfile
SAUVE?s
EXECUTION F4LIB VERSION V10-05 09*00*23*

?-tty§


-VEGA§
SYSTEM(FILE)/
   SORTS(FILE)/ IMPORTS(BOOL;INT)/

BUILD/                                                  OKDEFINE

   FILE(FVIDE<-FILE)/

                                                        ISBUILD

   FILE(AJOUT(FILE,INT)<-FILE)/

                                                        ISBUILD

OKOPNS/

   FILE(RETIR(FILE)<-FILE)/

                                                        ISOKOPNS

   FILE(TETE(FILE)<-INT)/

                                                        ISOKOPNS

   FILE(VIDE(FILE)<-BOOL)/

                                                        ISOKOPNS

   FILE(TAILLE(FILE)<-INT)/

                                                        ISOKOPNS

EROPNS/

   FILE(RETERR<-FILE)/

                                                        ISEROPNS

   FILE(TETERR<-INT)/

                                                        ISEROPNS

OKAXS/

   FILE(RETIR(AJOUT(*F,*I))=
       ITE(FILE(VIDE(*F)),FVIDE,
           AJOUT(RETIR(*F),*I)))/

                                                        ISOKAXS

   INT(TETE(AJOUT(*F,*I))=
       ITE(FILE(VIDE(*F)),*I,
           FILE(TETE(*F))))/

                                                        ISOKAXS

   BOOL(VIDE(FVIDE)=TRUE)/

                                                        ISOKAXS

   BOOL(VIDE(AJOUT(*F,*I))=FALSE)/

                                                        ISOKAXS

   INT(TAILLE(FVIDE)=0)/

                                                        ISOKAXS

   INT(TAILLE(AJOUT(*F,*I))=
          FILE(TAILLE(*F))+1)?

                                                        ISOKAXS
```



```
ERAXS/

   FILE(RETIR(FVIDE)=RETERR)/
                                                        ISERAXS

   INT(TETE(FVIDE)=TETERR)/

                                                        ISERAXS

END(FILE)/

RUN(FILE(RETIR(AJOUT(AJOUT(FVIDE,1),2))))?
                    FILE:AJOUT(FVIDE,2)

RUN(FILE(RETIR(RETIR(AJOUT(FVIDE,1)))))?
>>ERROR>>RETERR                    FILE:RETERR

RUN(FILE(TAILLE(AJOUT(AJOUT(FVIDE,1),2))))?
                    INT:2

RUN(FILE(TETE(AJOUT(AJOUT(FVIDE,1),2))))?
                    INT:1

RUN(FILE(TETE(RETIR(AJOUT(AJOUT(FVIDE,1),2)))))?
                    INT:2

RUN(FILE(TAILLE(RETIR(AJOUT(FVIDE,1)))))?
                    INT:0

 INPUT/

?bye/
 BYE/


*STOP* 0
!
```

```
 vega
SYSTEME?vega
SOURCE?esspilc
SAUVE?s
EXECUTION F4LIB VERSION V10-05 09*05*53*

?-tty§


 -VEGA§
 SYSTEM(PILC)/
     SORTS(PILC)/IMPORTS(INT)/

 BUILD/                                                  OKDEFINE

     PILC(PVIDE<-PILC)/

                                                         ISBUILD

     PILC(EMPILER(PILC,INT)<-PILC)/

                                                         ISBUILD

 END(PILC)/

 SYSTEM(PILE)/
     SORTS(PILE)/IMPORTS(PILC)/

 OKOPNS/                                                 OKDEFINE

     PILE(PILE(PILC)<-PILE)/

                                                         ISOKOPNS

     PILE(CARD(PILE)<-INT)/

                                                         ISOKOPNS

 OKAXS/

     INT(CARD(PVIDE)=0)/

                                                         ISOKAXS

     INT(CARD(EMPILER(*P,*I))=PILE(CARD(*P))+1)/

                                                         ISOKAXS

 END(PILE)/

 RUN(PILE(CARD(EMPILER(EMPILER(PVIDE,1),2))))?
                    INT:2

 INPUT/

?bye/
 BYE/


*STOP* 0
!
```



```
 vega
SYSTEME?vega
SOURCE?esspi
SAUVE?s
EXECUTION F4LIB VERSION V10-05 09*08*46*

?-tty§


 -VEGA§
 SYSTEM(PI)/
    SORTS(PI)/ MODEL(SEQ(INT))/
                                                         OKDEFINE
 OKOPNS/

    PI(PVIDE<-PI)/
                                                         ISOKOPNS

    PI(EMPILER(PI,INT)<-PI)/
                                                         ISOKOPNS

    PI(DEPILER(PI)<-PI)/
                                                         ISOKOPNS

    PI(SOMMET(PI)<-INT)/
                                                         ISOKOPNS

    PI(ESTVIDE(PI)<-BOOL)/
                                                         ISOKOPNS

 EROPNS/

    PI(PILERR<-PI)/
                                                         ISEROPNS

    PI(SOMERR<-INT)/
                                                         ISEROPNS

 AFUNC/

    PI(REP(NIL)=PVIDE)/
                                                         ISAFUNC

    PI(REP(*E.*F)=EMPILER(REP(*F),*E))/
                                                         ISAFUNC

 OKSEMS/

    PI(PVIDE=REP(NIL))/
                                                         ISOKSEMS

    PI(EMPILER(REP(*P),*I)=REP(ADG(*P,*I)))/
                                                         ISOKSEMS

    PI(DEPILER(REP(*P))=REP(DELG(*P)))/
                                                         ISOKSEMS

    INT(SOMMET(REP(*P))=CAR(*P))/
                                                         ISOKSEMS

    BOOL(ESTVIDE(REP(*P))=EMPTY(*P))/
                                                         ISOKSEMS

 ERSEMS/

    PI(DEPILER(PVIDE)=PILERR)/
```

```
   INT(SOMMET(PVIDE)=SOMERR)/
                                                    ISERSEMS
END(PI)/

RUN(PI(PVIDE))?
                    PI:PVIDE

RUN(PI(EMPILER(EMPILER(PVIDE,1),2)))?
                    PI:EMPILER(EMPILER(PVIDE,1),2)

RUN(PI(EMPILER(DEPILER(EMPILER(PVIDE,1)),2)))?
                    PI:EMPILER(PVIDE,2)

RUN(PI(SOMMET(DEPILER(EMPILER(PVIDE,1)))))?
>>ERROR>>SOMERR                    INT:SOMERR

RUN(PI(ESTVIDE(PVIDE)))?
                    BOOL:TRUE

RUN(PI(ESTVIDE(DEPILER(EMPILER(PVIDE,1)))))?
                    BOOL:TRUE

RUN(PI(ESTVIDE(EMPILER(PVIDE,1))))?
                    BOOL:FALSE

RUN(PI(SOMMET(PVIDE)))?
>>ERROR>>SOMERR                    INT:SOMERR

RUN(PI(DEPILER(PVIDE)))?
>>ERROR>>PILERR                    PI:PILERR

RUN(PI(DEPILER(DEPILER(DEPILER(EMPILER(PVIDE,1))))))?
>>ERROR>>PILERR                    PI:DEPILER(PILERR)

RUN(PI(EMPILER(EMPILER(PVIDE,1),
    PI(SOMMET(EMPILER(PVIDE,3))))))?
                    PI:EMPILER(EMPILER(PVIDE,1),3)

RUN(PI(SOMMET(EMPILER(EMPILER(PVIDE,1),2))))?
                    INT:2

RUN(PI(EMPILER(PVIDE,
    PI(SOMMET(DEPILER(EMPILER(PVIDE,1)))))))?
>>ERROR>>SOMERR>>ERROR>>SOMERR                    PI:EMPILER(PVIDE,SO
MERR)

INPUT/

?bye/
BYE/


*STOP* 0
!
```



```
vega
SYSTEME?vega
SOURCE?essgcd
SAUVE?s
EXECUTION F4LIB VERSION V10-05 09*12*36*

?-tty§


-VEGA§
SYSTEM(PGCD)/
   IMPORTS(INT;BOOL)/
                                                    OKDEFINE

OKOPNS/

   PGCD(PGCD(INT,INT)<-INT)/
                                                    ISOKOPNS

EROPNS/

   PGCD(ARGNEG<-INT)/
                                                    ISEROPNS

OKAXS/

   INT(PGCD(*I,*I)=*I)/
                                                    ISOKAXS

   INT(PGCD(*I,0)=*I)/
                                                    ISOKAXS

   INT(PGCD(0,*I)=*I)/
                                                    ISOKAXS

   INT(PGCD(*I,*J)=IT(INT(*J<*I) & INT(*J>0),
                      PGCD(*I-*J,*J)))/
                                                    ISOKAXS

   INT(PGCD(*I,*J)=IT(INT(*I<*J) & INT(*I>0),
                      PGCD(*I,*J-*I)))/
                                                    ISOKAXS

ERAXS/

   INT(PGCD(*I,*J)=IT(INT(*I<0) OR INT(*J<0),ARGNEG))/
                                                    ISERAXS

END(PGCD)/

RUN(PGCD(2,2))?
                    INT:2

RUN(PGCD(-2,3))?
>>ERROR>>ARGNEG                    INT:ARGNEG

RUN(PGCD(2,-3))?
>>ERROR>>ARGNEG                    INT:ARGNEG

RUN(PGCD(0,3))?
                    INT:3

RUN(PGCD(4,0))?
                    INT:4
```

```
 RUN(PGCD(3,5))?
                        INT:1

 RUN(PGCD(PGCD(2,2),1))?
                        INT:1

 RUN(PGCD(2+PGCD(2,2),2))?
                        INT:2

 INPUT/

?bye/
 BYE/


*STOP* 0
!
```



```
 vega
SYSTEME?vega
SOURCE?essmpita
SAUVE?s
EXECUTION F4LIB VERSION V10-05 09*17*04*

?-tty§


 -VEGA§
 SYSTEM(MPITA)/
   SORTS(TAB;PILC)/ IMPORTS(INT;BOOL)/
                                                    OKDEFINE
                                        OKDEFINE

 BUILD/

    PILC(PVIDE<-PILC)/                              ISBUILD

    PILC(EMP(PILC,INT)<-PILC)/                      ISBUILD

 BUILD/

    TAB(TVIDE<-TAB)/                                ISBUILD

    TAB(RANGER(TAB,INT,INT)<-TAB)/                  ISBUILD

 OKOPNS/

    TAB(LIRE(TAB,INT)<-INT)/                        ISOKOPNS

    TAB(ENLEVER(TAB,INT)<-TAB)/                     ISOKOPNS

    TAB(EXISTE(TAB,INT)<-BOOL)/                     ISOKOPNS

 EROPNS/

    TAB(VALERR<-INT)/                               ISEROPNS

 OKAXS(TAB)/

    INT(LIRE(RANGER(*T,*CL1,*V),*CL2)=
        ITE(INT(*CL1=*CL2),*V,
            TAB(LIRE(*T,*CL2))))/
                                                    ISOKAXS

    TAB(ENLEVER(TVIDE,*CL1)=TVIDE)/                 ISOKAXS

    TAB(ENLEVER(RANGER(*T,*CL1,*V),*CL2)=
        ITE(INT(*CL1=*CL2),ENLEVER(*T,*CL2),
            RANGER(ENLEVER(*T,*CL2),*CL1,*V)))/
                                                    ISOKAXS

    BOOL(EXISTE(TVIDE,*CL1)=FALSE)/                 ISOKAXS
```

```
    BOOL(EXISTE(RANGER(*T,*CL1,*V),*CL2)=
         ITE(INT(*CL1=*CL2),TRUE,
             TAB(EXISTE(*T,*CL2))))/

                                                    ISOKAXS
  ERAXS(TAB)/

     INT(LIRE(TVIDE,*I)=VALERR)/

                                                    ISERAXS
  END(MPITA)/

  RUN(TAB(LIRE(RANGER(RANGER(TVIDE,1,10),2,20),2)))?
                        INT:20

  RUN(TAB(ENLEVER(RANGER(RANGER(RANGER(TVIDE,1,10),2,20),
     3,30),2)))?
                        TAB:RANGER(RANGER(TVIDE,1,10),3,30)
  RUN(TAB(EXISTE(RANGER(RANGER(TVIDE,1,10),2,20),3)))?
                        BOOL:FALSE

  INPUT/

 ?bye/
  BYE/


*STOP* 0
!
```



```
 vega
SYSTEME?vega
SOURCE?esspil
SAUVE?s
EXECUTION F4LIB VERSION V10-05 09*20*22*

?-tty§


 -VEGA§
 SYSTEM(PILC)/
    SORTS(PILC)/ IMPORTS(INT)/
                                                    OKDEFINE
 BUILD/

    PILC(PVIDE<-PILC)/
                                                    ISBUILD

    PILC(EMP(PILC,INT)<-PILC)/
                                                    ISBUILD

    PILC(DESC(PILC)<-PILC)/
                                                    ISBUILD

 OKOPNS/

    PILC(DEP(PILC)<-PILC)/
                                                    ISOKOPNS

    PILC(VIDE(PILC)<-BOOL)/
                                                    ISOKOPNS

 EROPNS/

    PILC(DESCERR<-PILC)/
                                                    ISEROPNS

 OKAXS/

   PILC(DEP(EMP(*P,*I))=*P)/
                                                    ISOKAXS

    BOOL(VIDE(PVIDE)=TRUE)/
                                                    ISOKAXS

    BOOL(VIDE(EMP(*P,*I))=FALSE)/
                                                    ISOKAXS

    BOOL(VIDE(DESC(*P))=FALSE)/
                                                    ISOKAXS

 ERAXS/

    PILC(DESC(PVIDE)=DESCERR)/
                                                    ISERAXS

 END(PILC)/

 SYSTEM(PLI)/
    SORTS(PLI)/ IMPORTS(INT;BOOL;PILC)/
                                                    OKDEFINE
 OKOPNS/

    PLI(PLI(PILC)<-PLI)/
```

```
                                                    ISOKOPNS
   PLI(REM(PLI)<-PLI)/
                                                    ISOKOPNS
   PLI(VAL(PLI)<-INT)/
                                                    ISOKOPNS
   PLI(INDSOM(PLI)<-BOOL)/
                                                    ISOKOPNS
   PLI(DECR(PLI)<-PLI)/
                                                    ISOKOPNS
EROPNS/
   PLI(REMERR<-PLI)/
                                                    ISEROPNS
   PLI(DECRERR<-PLI)/
                                                    ISEROPNS
   PLI(DEPDESC<-PLI)/
                                                    ISEROPNS
   PLI(DESCERR<-PLI)/
                                                    ISEROPNS
   PLI(DEPVIDE<-PLI)/
                                                    ISEROPNS
   PLI(VALERR<-INT)/
                                                    ISEROPNS
OKAXS/
   PLI(REM(DESC(*P))=REM(*P))/
                                                    ISOKAXS
   PLI(REM(EMP(*P,*I))=EMP(*P,*I))/
                                                    ISOKAXS
   BOOL(INDSOM(PVIDE)=TRUE)/
                                                    ISOKAXS
   BOOL(INDSOM(DESC(*P))=FALSE)/
                                                    ISOKAXS
   BOOL(INDSOM(EMP(*P,*I))=TRUE)/
                                                    ISOKAXS
   PLI(DECR(EMP(*P,*I))=*P)/
                                                    ISOKAXS
   PLI(DECR(DESC(*P))=DEP(DECR(*P)))/
                                                    ISOKAXS
   INT(VAL(EMP(*P,*I))=*1)/
                                                    ISOKAXS
   INT(VAL(DESC(*P))=PLI(VAL(DECR(*P))))/
                                                    ISOKAXS
ERAXS/
```



```
   PLI(DEP(PVIDE)=DEPVIDE)/
                                                    ISERAXS
   PLI(DEP(DESC(*P))=DEPDESC)/
                                                    ISERAXS
   PLI(REM(PVIDE)=REMERR)/
                                                    ISERAXS
   PLI(DECR(PVIDE)=DECRERR)/
                                                    ISERAXS
   INT(VAL(PVIDE)=VALERR)/
                                                    ISERAXS
END(PIL)/

RUN(PLI(REM(DESC(DESC(PVIDE)))))?
>>ERROR>>DESCERR                    PLI:REM(PLI(DESC(DESCERR)))

RUN(PLI(DEP(EMP(PVIDE,2))))?
                    PLI:PVIDE

RUN(PLI(DECR(DESC(DESC(PVIDE)))))?
>>ERROR>>DESCERR                    PLI:DECR(PLI(DESC(DESCERR)))

RUN(PLI(INDSOM(EMP(EMP(PVIDE,2),3))))?
                    BOOL:TRUE

RUN(PLI(INDSOM(DESC(DESC(PVIDE)))))?
>>ERROR>>DESCERR                    BOOL:INDSOM(PLI(DESC(DESCERR)))

RUN(PLI(VAL(EMP(PVIDE,3))))?
                    INT:3

RUN(PLI(VAL(EMP(EMP(EMP(PVIDE,1),2),3))))?
                    INT:3

RUN(PLI(VAL(DESC(EMP(EMP(PVIDE,1),2)))))?
                    INT:1

RUN(PLI(VAL(DESC(DESC(EMP(EMP(PVIDE,1),2))))))?
>>ERROR>>VALERR>>ERROR>>
VINT:°(VALERR)
                    INT:VALERR

RUN(PLI(VAL(DEP(REM(DESC(EMP(EMP(PVIDE,1),2)))))))?
                    INT:1

RUN(PLI(DEP(REM(DESC(EMP(EMP(PVIDE,1),2))))))?
                    PLI:EMP(PVIDE,1)

RUN(PLI(VAL(DEP(DESC(EMP(EMP(PVIDE,1),2))))))?
>>ERROR>>DEPDESC                    INT:VAL(PLI(DEPDESC))

 INPUT/

?bye/
 BYE/


*STOP* 0
!
```

```
 VEGA
SYSTEME?VEGA
SOURCE?ESSPILINT
SAUVE?S
EXECUTION F4LIB VERSION V10-05 23*16*41*

?-TTY§


 -VEGA§
 SYSTEM(PILINT)/
     SORTS(PILINT)/ IMPORTS(INT;BOOL)/
                                        OKDEFINE
 BUILD/

     PILINT(PVIDE<-PILINT)/
                                        ISBUILD

     PILINT(EMPILER(PILINT,INT)<-PILINT)/
                                        ISBUILD

 OKOPNS/

     PILINT(DEPILER(PILINT)<-PILINT)/
                                        ISOKOPNS

     PILINT(SOMMET(PILINT)<-INT)/
                                        ISOKOPNS

     PILINT(ESTVIDE(PILINT)<-BOOL)/
                                        ISOKOPNS

 EROPNS/

     PILINT(SOMERR<-INT)/
                                        ISEROPNS

     PILINT(PILERR<-PILINT)/
                                        ISEROPNS

 OKAXS/

     PILINT(DEPILER(EMPILER(*P,*I))=*P)/
                                        ISOKAXS

     INT(SOMMET(EMPILER(*P,*I))=*I)/
                                        ISOKAXS

     BOOL(ESTVIDE(PVIDE)=TRUE)/
                                        ISOKAXS

     BOOL(ESTVIDE(EMPILER(*P,*I))=FALSE)/
                                        ISOKAXS

 ERAXS/

     PILINT(DEPILER(PVIDE)=PILERR)/
                                        ISERAXS

     INT(SOMMET(PVIDE)=SOMERR)/
                                        ISERAXS

 END(PILINT)/

 RUN(PILINT(PVIDE))?
```



```
                    PILINT:PVIDE

 RUN(PILINT(EMPILER(EMPILER(PVIDE,1),2)))?
                    PILINT:EMPILER(EMPILER(PVIDE,1),2)

 RUN(PILINT(EMPILER(DEPILER(EMPILER(PVIDE,1)),2)))?
                    PILINT:EMPILER(PVIDE,2)

 RUN(PILINT(SOMMET(DEPILER(EMPILER(EMPILER(PVIDE,1),2)))))?
                    INT:1

 RUN(PILINT(ESTVIDE(PVIDE)))?
                    BOOL:TRUE

 RUN(PILINT(ESTVIDE(DEPILER(EMPILER(PVIDE,1)))))?
                    BOOL:TRUE

 RUN(PILINT(ESTVIDE(EMPILER(PVIDE,1))))?
                    BOOL:FALSE

 RUN(PILINT(SOMMET(PVIDE)))?
 >>ERROR>>SOMERR                INT:SOMERR

 RUN(PILINT(DEPILER(PVIDE)))?
 >>ERROR>>PILERR                PILINT:PILERR

 RUN(PILINT(EMPILER(PVIDE,
     PILINT(SOMMET(DEPILER(EMPILER(PVIDE,1)))))))?
 >>ERROR>>SOMERR>>ERROR>>SOMERR                    PILINT:EMPILER(PVID
 E,SOMERR)

 RUN(PILINT(EMPILER(EMPILER(PVIDE,2),
     PILINT(SOMMET(EMPILER(PVIDE,3))))))?
                    PILINT:EMPILER(EMPILER(PVIDE,2),3)

 INPUT/

 ?BYE/
  BYE/


 *STOP* 0
 !
```

```
 VEGA
SYSTEME?VEGA
SOURCE?ESSFACT
SAUVE?S
EXECUTION F4LIB VERSION V10-05 23*19*13*

?-TTY§


 -VEGA§
 SYSTEM(FACT)/
      IMPORTS(INT;BOOL)/
                                                    OKDEFINE
 OKOPNS/

    FACT(FACT(INT)<-INT)/
                                                    ISOKOPNS

 EROPNS/

    FACT(ARGNEG<-INT)/
                                                    ISEROPNS

 OKAXS/

    INT(FACT(*I)=ITE(INT(*I=0),1,
         INT(*I&FACT(*I-1))))/
                                                    ISOKAXS

 ERAXS/

    INT(FACT(*I)=IT(INT(*I<0),ARGNEG))/
                                                    ISERAXS

 END(FACT)/

 RUN(FACT(0))?
                    INT:1

 RUN(FACT(1))?
                    INT:1

 RUN(FACT(3))?
                    INT:6

 RUN(FACT(-3))?
 >>ERROR>>ARGNEG                INT:ARGNEG

 RUN(INT(1+FACT(2)))?
                    INT:3

 INPUT/

?BYE/
 BYE/


*STOP* 0
!
```



```
 VEGA
SYSTEME?VEGA
SOURCE?ESSFILE
SAUVE?S
EXECUTION F4LIB VERSION V10-05 23*21*34*

?-TTY§


 -VEGA§
 SYSTEM(FILE)/
    SORTS(FILE)/ IMPORTS(BOOL;INT)/
                                                    OKDEFINE
 BUILD/

    FILE(FVIDE<-FILE)/
                                                    ISBUILD

    FILE(AJOUT(FILE,INT)<-FILE)/
                                                    ISBUILD

 OKOPNS/

    FILE(RETIR(FILE)<-FILE)/
                                                    ISOKOPNS

    FILE(TETE(FILE)<-INT)/
                                                    ISOKOPNS

    FILE(VIDE(FILE)<-BOOL)/
                                                    ISOKOPNS

    FILE(TAILLE(FILE)<-INT)/
                                                    ISOKOPNS

 EROPNS/

    FILE(RETERR<-FILE)/
                                                    ISEROPNS

    FILE(TETERR<-INT)/
                                                    ISEROPNS

 OKAXS/

    FILE(RETIR(AJOUT(*F,*I))=
        ITE(FILE(VIDE(*F)),FVIDE,
            AJOUT(RETIR(*F),*I)))/
                                                    ISOKAXS

    INT(TETE(AJOUT(*F,*I))=
        ITE(FILE(VIDE(*F)),*I,
            FILE(TETE(*F))))/
                                                    ISOKAXS

    BOOL(VIDE(FVIDE)=TRUE)/
                                                    ISOKAXS

    BOOL(VIDE(AJOUT(*F,*I))=FALSE)/
                                                    ISOKAXS

    INT(TAILLE(FVIDE)=0)/
                                                    ISOKAXS

    INT(TAILLE(AJOUT(*F,*I))=
```

```
    INT(TAILLE(AJOUT(*F,1))=
            FILE(TAILLE(*F))+1)?
                                                ISOKAXS

ERAXS/

    FILE(RETIR(FVIDE)=RETERR)/
                                                ISERAXS

    INT(TETE(FVIDE)=TETERR)/
                                                ISERAXS

END(FILE)/

RUN(FILE(RETIR(AJOUT(AJOUT(FVIDE,1),2))))?
                        FILE:AJOUT(FVIDE,2)

RUN(FILE(RETIR(RETIR(AJOUT(FVIDE,1)))))?
>>ERROR>>RETERR                    FILE:RETERR

RUN(FILE(TAILLE(AJOUT(AJOUT(FVIDE,1),2))))?
                        INT:2

RUN(FILE(TETE(AJOUT(AJOUT(FVIDE,1),2))))?
                        INT:1

RUN(FILE(TETE(RETIR(AJOUT(AJOUT(FVIDE,1),2)))))?
                        INT:2

RUN(FILE(TAILLE(RETIR(AJOUT(FVIDE,1)))))?
                        INT:0

INPUT/

?BYE/
 BYE/


*STOP* 0
!
```



```
 VEGA
SYSTEME?VEGA
SOURCE?ESSPILC
SAUVE?S
EXECUTION F4LIB VERSION V10-05 23*25*50*

?-TTY§


 -VEGA§
 SYSTEM(PILC)/
     SORTS(PILC)/IMPORTS(INT)/
                                                OKDEFINE

 BUILD/

     PILC(PVIDE<-PILC)/
                                                ISBUILD

     PILC(EMPILER(PILC,INT)<-PILC)/
                                                ISBUILD

 END(PILC)/

 SYSTEM(PILE)/
     SORTS(PILE)/IMPORTS(PILC)/
                                                OKDEFINE

 OKOPNS/

     PILE(PILE(PILC)<-PILE)/
                                                ISOKOPNS

     PILE(CARD(PILE)<-INT)/
                                                ISOKOPNS

 OKAXS/

     INT(CARD(PVIDE)=0)/
                                                ISOKAXS

     INT(CARD(EMPILER(*P,*I))=PILE(CARD(*P))+1)/
                                                ISOKAXS

 END(PILE)/

 RUN(PILE(CARD(EMPILER(EMPILER(PVIDE,1),2))))?
                        INT:2

 INPUT/

?BYE/
 BYE/


*STOP* 0
!
```

```
 VEGA
SYSTEME?VEGA
SOURCE?ESSPI
SAUVE?S
EXECUTION F4LIB VERSION V10-05 23*27*27*

?-TTY§


 -VEGA§
 SYSTEM(PI)/
    SORTS(PI)/ MODEL(SEQ(INT))/
                                                    OKDEFINE
 OKOPNS/
    PI(PVIDE<-PI)/
                                                    ISOKOPNS
    PI(EMPILER(PI,INT)<-PI)/
                                                    ISOKOPNS
    PI(DEPILER(PI)<-PI)/
                                                    ISOKOPNS
    PI(SOMMET(PI)<-INT)/
                                                    ISOKOPNS
    PI(ESTVIDE(PI)<-BOOL)/
                                                    ISOKOPNS
 EROPNS/
    PI(PILERR<-PI)/
                                                    ISEROPNS
    PI(SOMERR<-INT)/
                                                    ISEROPNS
 AFUNC/
    PI(REP(NIL)=PVIDE)/
                                                    ISAFUNC
    PI(REP(*E.*F)=EMPILER(REP(*F),*E))/
                                                    ISAFUNC
 OKSEMS/
    PI(PVIDE=REP(NIL))/
                                                    ISOKSEMS
    PI(EMPILER(REP(*P),*I)=REP(ADG(*P,*I)))/
                                                    ISOKSEMS
    PI(DEPILER(REP(*P))=REP(DELG(*P)))/
                                                    ISOKSEMS
    INT(SOMMET(REP(*P))=CAR(*P))/
                                                    ISOKSEMS
    BOOL(ESTVIDE(REP(*P))=EMPTY(*P))/
                                                    ISOKSEMS
 ERSEMS/
```



```
    PI(DEPILER(PVIDE)=PILERR)/
                                                    ISERSEMS
    INT(SOMMET(PVIDE)=SOMERR)/
                                                    ISERSEMS
 END(PI)/

 RUN(PI(PVIDE))?
                    PI:PVIDE
 RUN(PI(EMPILER(EMPILER(PVIDE,1),2)))?
                    PI:EMPILER(EMPILER(PVIDE,1),2)
 RUN(PI(EMPILER(DEPILER(EMPILER(PVIDE,1)),2)))?
                    PI:EMPILER(PVIDE,2)
 RUN(PI(SOMMET(DEPILER(EMPILER(PVIDE,1)))))?
 >>ERROR>>SOMERR                    INT:SOMERR
 RUN(PI(ESTVIDE(PVIDE)))?
                    BOOL:TRUE
 RUN(PI(ESTVIDE(DEPILER(EMPILER(PVIDE,1)))))?
                    BOOL:TRUE
 RUN(PI(ESTVIDE(EMPILER(PVIDE,1))))?
                    BOOL:FALSE
 RUN(PI(SOMMET(PVIDE)))?
 >>ERROR>>SOMERR                    INT:SOMERR
 RUN(PI(DEPILER(PVIDE)))?
 >>ERROR>>PILERR                    PI:PILERR
 RUN(PI(DEPILER(DEPILER(DEPILER(EMPILER(PVIDE,1))))))?
 >>ERROR>>PILERR                    PI:DEPILER(PILERR)
 RUN(PI(EMPILER(EMPILER(PVIDE,1),
     PI(SOMMET(EMPILER(PVIDE,3))))))?
                    PI:EMPILER(EMPILER(PVIDE,1),3)
 RUN(PI(SOMMET(EMPILER(EMPILER(PVIDE,1),2))))?
                     INT:2
 RUN(PI(EMPILER(PVIDE,
     PI(SOMMET(DEPILER(EMPILER(PVIDE,1)))))))?
 >>ERROR>>SOMERR>>ERROR>>SOMERR                    PI:EMPILER(PVIDE,SO
 MERR)
 INPUT/
 ?BYE/
 BYE/


 *STOP* 0
 !
```

```
 VEGA
SYSTEME?VEGA
SOURCE?ESSGCD
SAUVE?S
EXECUTION F4LIB VERSION V10-05 23*30*14*

?-TTY§


 -VEGA§
 SYSTEME(PGCD)/
    IMPORTS(INT;BOOL)/

 OKOPNS/                                              OKDEFINE

    PGCD(PGCD(INT,INT)<-INT)/

                                                      ISOKOPNS
 EROPNS/

    PGCD(ARGNEG<-INT)/

                                                      ISEROPNS
 OKAXS/

    INT(PGCD(*I,*I)=*I)/

                                                      ISOKAXS
    INT(PGCD(*I,0)=*I)/

                                                      ISOKAXS
    INT(PGCD(0,*I)=*I)/

                                                      ISOKAXS
    INT(PGCD(*I,*J)=IT(INT(*J<*I) & INT(*J>0),
                       PGCD(*I-*J,*J)))/

                                                      ISOKAXS
    INT(PGCD(*I,*J)=IT(INT(*I<*J) & INT(*I>0),
                       PGCD(*I,*J-*I)))/

                                                      ISOKAXS
 ERAXS/

    INT(PGCD(*I,*J)=IT(INT(*I<0) OR INT(*J<0),ARGNEG))/

                                                     ISERAXS
 END(PGCD)/

 RUN(PGCD(2,2))?
                    INT:2

 RUN(PGCD(-2,3))?
 >>ERROR>>ARGNEG                    INT:ARGNEG

 RUN(PGCD(2,-3))?
 >>ERROR>>ARGNEG                    INT:ARGNEG

 RUN(PGCD(0,3))?
                    INT:3

 RUN(PGCD(4,0))?
                    INT:4

 RUN(PGCD(3,5))?
                    INT:1
```



```
 RUN(PGCD(PGCD(2,2),1))?
                    INT:1

 RUN(PGCD(2+PGCD(2,2),2))?
                    INT:2

 INPUT/

 ?BYE/
  BYE/


 *STOP* 0
 !
```

```
 VEGA
SYSTEME?VEGA
SOURCE?ESSMPITA
SAUVE?S
EXECUTION F4LIB VERSION V10-05 23*33*04*

?-TTY§


 -VEGA§
 SYSTEM(MPITA)/
   SORTS(TAB;PILC)/ IMPORTS(INT;BOOL)/
                                                OKDEFINE
                                        OKDEFINE
 BUILD/

    PILC(PVIDE<-PILC)/
                                                ISBUILD

    PILC(EMP(PILC,INT)<-PILC)/
                                                ISBUILD

 BUILD/

    TAB(TVIDE<-TAB)/
                                                ISBUILD

    TAB(RANGER(TAB,INT,INT)<-TAB)/
                                                ISBUILD

 OKOPNS/

    TAB(LIRE(TAB,INT)<-INT)/
                                                ISOKOPNS

    TAB(ENLEVER(TAB,INT)<-TAB)/
                                                ISOKOPNS

    TAB(EXISTE(TAB,INT)<-BOOL)/
                                                ISOKOPNS

 EROPNS/

    TAB(VALERR<-INT)/
                                                ISEROPNS

 OKAXS(TAB)/

    INT(LIRE(RANGER(*T,*CL1,*V),*CL2)=
        ITE(INT(*CL1=*CL2),*V,
            TAB(LIRE(*T,*CL2))))/
                                                ISOKAXS

    TAB(ENLEVER(TVIDE,*CL1)=TVIDE)/
                                                ISOKAXS

    TAB(ENLEVER(RANGER(*T,*CL1,*V),*CL2)=
        ITE(INT(*CL1=*CL2),ENLEVER(*T,*CL2),
            RANGER(ENLEVER(*T,*CL2),*CL1,*V)))/
                                                ISOKAXS

    BOOL(EXISTE(TVIDE,*CL1)=FALSE)/
                                                ISOKAXS
```



```
    BOOL(EXISTE(RANGER(*T,*CL1,*V),*CL2)=
        ITE(INT(*CL1=*CL2),TRUE,
            TAB(EXISTE(*T,*CL2))))/
                                                ISOKAXS

 ERAXS(TAB)/

    INT(LIRE(TVIDE,*I)=VALERR)/
                                                ISERAXS

 END(MPITA)/

 RUN(TAB(LIRE(RANGER(RANGER(TVIDE,1,10),2,20),2)))?
                    INT:20

 RUN(TAB(ENLEVER(RANGER(RANGER(RANGER(TVIDE,1,10),2,20),
    3,30),2)))?
                    TAB:RANGER(RANGER(TVIDE,1,10),3,30)

 RUN(TAB(EXISTE(RANGER(RANGER(TVIDE,1,10),2,20),3)))?
                    BOOL:FALSE

 INPUT/

?BYE/
 BYE,


*STOP* 0
]
```

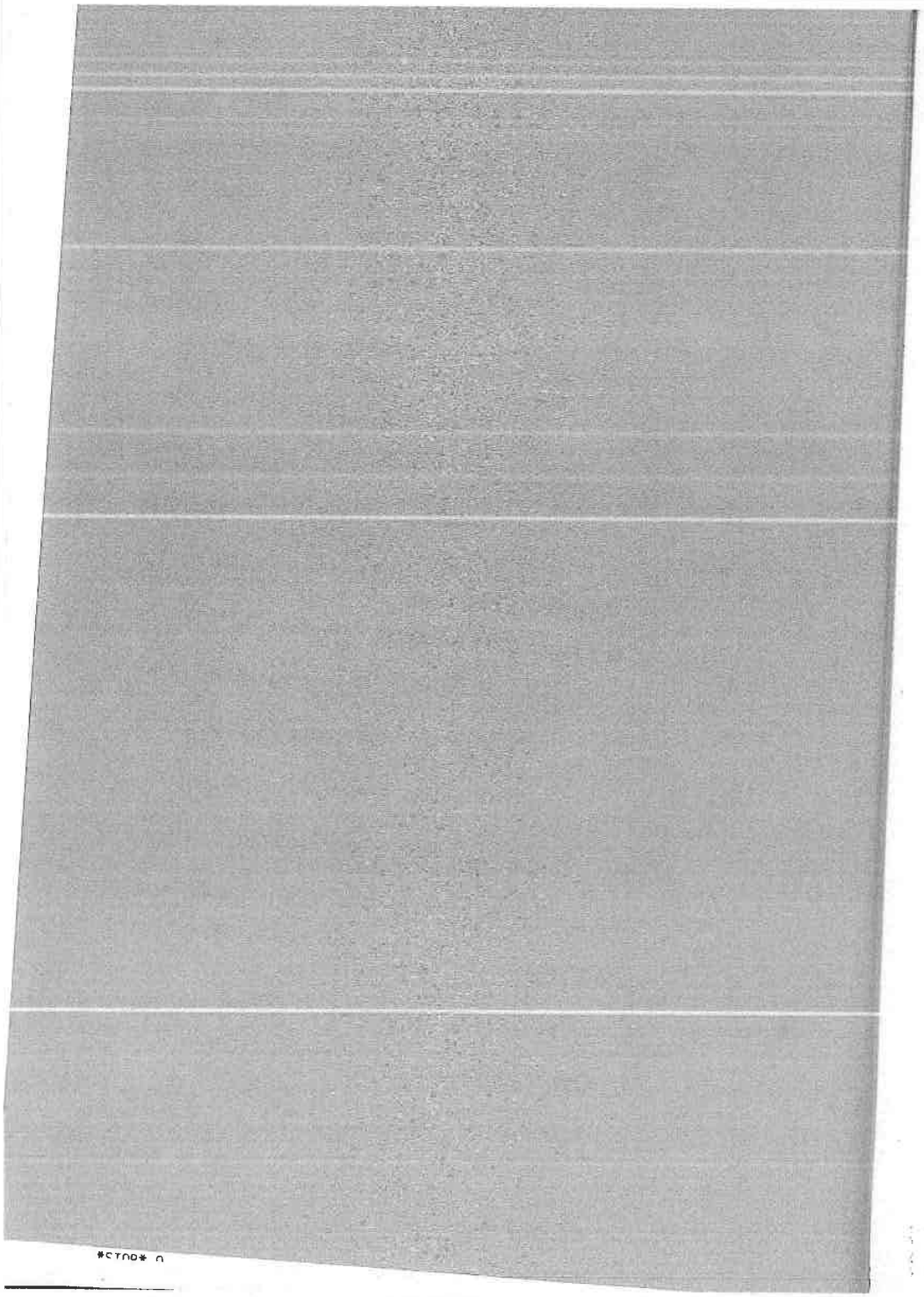
```
 VEGA
SYSTEME?VEGA
SOURCE?ESSAIL
SAUVE?S
EXECUTION F4LIB VERSION V10-05 23*36*29*

?-TTY§


-VEGA§
SYSTEM(PILC)/
   SORTS(PILC)/ IMPORTS(INT)/
                                                    OKDEFINE
BUILD/
   PILC(PVIDE<-PILC)/
                                                    ISBUILD
   PILC(EMP(PILC,INT)<-PILC)/
                                                    ISBUILD
   PILC(DESC(PILC)<-PILC)/
                                                    ISBUILD
OPNS/
   PILC(DEP(PILC)<-PILC)/
                                                    ISOKOPNS
   PILC(VIDE(PILC)<-BOOL)/
                                                    ISOKOPNS
EROPNS/
   PILC(DESCERR<-PILC)/
                                                    ISEROPNS
OKAXS/
  PILC(DEP(EMP(*P,*I))=*P)/
                                                    ISOKAXS
   BOOL(VIDE(PVIDE)=TRUE)/
                                                    ISOKAXS
   BOOL(VIDE(EMP(*P,*I))=FALSE)/
                                                    ISOKAXS
   BOOL(VIDE(DESC(*P))=FALSE)/
                                                    ISOKAXS
ERAXS/
   PILC(DESC(PVIDE)=DESCERR)/
                                                    ISERAXS
END(PILC)/

SYSTEM(PLI)/
   SORTS(PLI)/ IMPORTS(INT;BOOL;PILC)/
                                                    OKDEFINE
OKOPNS/
   PLI(PLI(PILC)<-PLI)/
                                                    ISOKOPNS
```



```
   PLI(REM(PLI)<-PLI)/
                                                    ISOKOPNS
   PLI(VAL(PLI)<-INT)/
                                                    ISOKOPNS
   PLI(INDSOM(PLI)<-BOOL)/
                                                    ISOKOPNS
   PLI(DECR(PLI)<-PLI)/
                                                    ISOKOPNS
EROPNS/
   PLI(REMERR<-PLI)/
                                                    ISEROPNS
   PLI(DECRERR<-PLI)/
                                                    ISEROPNS
   PLI(DEPDESC<-PLI)/
                                                    ISEROPNS
   PLI(DESCERR<-PLI)/
                                                    ISEROPNS
   PLI(DEPVIDE<-PLI)/
                                                    ISEROPNS
   PLI(VALERR<-INT)/
                                                    ISEROPNS
OKAXS/
   PLI(REM(DESC(*P))=REM(*P))/
                                                    ISOKAXS
   PLI(REM(EMP(*P,*I))=EMP(*P,*I))/
                                                    ISOKAXS
   BOOL(INDSOM(PVIDE)=TRUE)/
                                                    ISOKAXS
   BOOL(INDSOM(DESC(*P))=FALSE)/
                                                    ISOKAXS
   BOOL(INDSOM(EMP(*P,*I))=TRUE)/
                                                    ISOKAXS
   PLI(DECR(EMP(*P,*I))=*P)/
                                                    ISOKAXS
   PLI(DECR(DESC(*P))=DEP(DECR(*P)))/
                                                    ISOKAXS
   INT(VAL(EMP(*P,*I))=*I)/
                                                    ISOKAXS
   INT(VAL(DESC(*P))=PLI(VAL(DECR(*P))))/
                                                    ISOKAXS
ERAXS/
   PLI(DEP(PVIDE)=DEPVIDE)/
                                                    ISERAXS
```

```
                                                    ISERAXS

      PLI(DEP(DESC(*P))=DEPDESC)/

                                                    ISERAXS

      PLI(REM(PVIDE)=REMERR)/

                                                    ISERAXS

      PLI(DECR(PVIDE)=DECRERR)/

                                                    ISERAXS

      INT(VAL(PVIDE)=VALERR)/

                                                    ISERAXS

    END(PIL)/

    RUN(PLI(REM(DESC(DESC(PVIDE)))))?
    >>ERROR>>DESCERR                    PLI:REM(PLI(DESC(DESCERR)))

    RUN(PLI(DEP(EMP(PVIDE,2))))?
                        PLI:PVIDE

    RUN(PLI(DECR(DESC(DESC(PVIDE)))))?
    >>ERROR>>DESCERR                    PLI:DECR(PLI(DESC(DESCERR)))

    RUN(PLI(INDSOM(EMP(EMP(PVIDE,2),3))))?
                        BOOL:TRUE

    RUN(PLI(INDSOM(DESC(DESC(PVIDE)))))?
    >>ERROR>>DESCERR                    BOOL:INDSOM(PLI(DESC(DESCERR)))

    RUN(PLI(VAL(EMP(EMP(PVIDE,3))))?
                        INT:3

    RUN(PLI(VAL(EMP(EMP(EMP(PVIDE,1),2),3))))?
                        INT:3

    RUN(PLI(VAL(DESC(EMP(EMP(PVIDE,1),2)))))?
                        INT:1

    RUN(PLI(VAL(DESC(DESC(EMP(EMP(PVIDE,1),2))))))?
    >>ERROR>>
    VPLI:DECR(EMP(EMP(PVIDE,1),2))
    VINT:°(VAL(PLI(DEP(DECR(EMP(EMP(PVIDE,1),2))))))
                        INT:VAL(PLI(DEP(DECR(EMP(EMP(PVIDE,1),2)))))

    RUN(PLI(VAL(DEP(REM(DESC(EMP(EMP(PVIDE,1),2)))))))?
    >>ERROR>>
    VPLI:REM(DESC(EMP(EMP(PVIDE,1),2)))
                        INT:VAL(PLI(DEP(REM(DESC(EMP(EMP(PVIDE,1),2))))))


    RUN(PLI(DEP(REM(DESC(EMP(EMP(PVIDE,1),2))))))?
    >>ERROR>>
    VPLI:REM(DESC(EMP(EMP(PVIDE,1),2)))
                        PLI:PLI(DEP(REM(DESC(EMP(EMP(PVIDE,1),2)))))

    RUN(PLI(VAL(DEP(DESC(EMP(EMP(PVIDE,1),2))))))?
    >>ERROR>>DEPDESC                    INT:VAL(PLI(DEPDESC))

    INPUT/

  ?BYE/
   BYE/

  *STOP* 0
```

GESTION DE BIBLIOTHEQUE




```
-VEGA]
SYSTEM(NUML)/
   SORTS(NUML)/ MODEL(INT RESTRICTO (*I:0 TO 60))/
                                                    OKDEFINE

END(NUML)/



SYSTEM(NUMC)/
   SORTS(NUMC)/ MODEL(INT RESTRICTO (*I:50 TO 100))/
                                                    OKDEFINE

END(NUMC)/



SYSTEM(LIVRE)/ SORTS(LIVRE)/
   MODEL(TUPLE(<NUM:NUML;AUTEUR:STRING;SUJET:STRING;
         TITRE:STRING>))/
                                                    OKDEFINE

END(LIVRE)/



SYSTEM(CHERCH)/ SORTS(CHERCH)/
   MODEL(TUPLE(<NUM:NUMC;NOM:STRING>))/
                                                    OKDEFINE

END(CHERCH)/



SYSTEM(PRET)/ SORTS(PRET)/
   MODEL(TUPLE(<LI:NUML;CH:NUMC>) INVARIANT IPRET)/
                                                    OKDEFINE

INVAR(IPRET((*NL;*NC))<-
   PRET((*NL;*NC):(PRETS(A!(*NL;*N) IN EP:
                         (NUMC(*N=*NC)))),
        ERRPRET))/


END(PRET)/



SYSTEM(LIVRES)/ SORTS(LIVRES)/
   MODEL(SET(LIVRE))/
                                                    OKDEFINE

END(LIVRES)/



SYSTEM(CHERCHS)/ SORTS(CHERCHS)/
   MODEL(SET(CHERCH))/
                                                    OKDEFINE

END(CHERCHS)/



SYSTEM(PRETS)/ SORTS(PRETS)/
   MODEL(SEQ(PRET))/
                                                    OKDEFINE

END(PRETS)/
```

```
RUN(LIVRES(EL<-0))?
                    LIVRES:EL:0

RUN(LIVRES(EL<-ADD(EL,(1;MARTIN;INFO;COBOL))))?
                    LIVRE:EL:(1;MARTIN;INFO;COBOL#0)

RUN(LIVRES(EL<-ADD(EL,(3;DUPOND;MATH;LOGIQUE))))?
                    LIVRE:EL:(1;MARTIN;INFO;COBOL#3;DUPOND;MATH;LOGIQ
UE#0)

RUN(LIVRES(EL<-ADD(EL,(5;DURAND;INFO;PL1))))?
                    LIVRE:EL:(1;MARTIN;INFO;COBOL#3;DUPOND;MATH;LOGIQ
UE#5;DURAND;INFO;PL1#0)

RUN(LIVRES(EL<-ADD(EL,(7;CHABRIER;INFO;VEGA))))?
                    LIVRE:EL:(1;MARTIN;INFO;COBOL#3;DUPOND;MATH;LOGIQ
UE#5;DURAND;INFO;PL1#7;CHABRIER;INFO;VEGA#0)

RUN(LIVRES(EL<-ADD(EL,(72;MARTIN;INFO;PASCAL))))?
>>ERROR>>
NUML:72NUML:72VLIVRE:72;MARTIN;INFO;PASCAL
>>ERR-ASS>>
                    LIVRES:EL:ADD(1;MARTIN;INFO;COBOL#3;DUPOND;MATH;L
OGIQUE#5;DURAND;INFO;PL1#7;CHABRIER;INFO;VEGA#0,72;MARTIN;INFO;PASCAL
)

RUN(LIVRES(EL))?
                    LIVRES:1;MARTIN;INFO;COBOL#3;DUPOND;MATH;LOGIQUE#
5;DURAND;INFO;PL1#7;CHABRIER;INFO;VEGA#0



RUN(CHERCHS(EC<-0))?
                    CHERCHS:EC:0

RUN(CHERCHS(EC<-ADD(EC,(52;JEAN))))?
                    CHERCH:EC:(52;JEAN#0)

RUN(CHERCHS(EC<-ADD(EC,(54;CLAUDE))))?
                    CHERCH:EC:(52;JEAN#54;CLAUDE#0)

RUN(CHERCHS(EC<-ADD(EC,(56;JACQUES))))?
                    CHERCH:EC:(52;JEAN#54;CLAUDE#56;JACQUES#0)

RUN(CHERCHS(EC<-ADD(EC,(58;PAUL))))?
                    CHERCH:EC:(52;JEAN#54;CLAUDE#56;JACQUES#58;PAUL#0
)

RUN(CHERCHS(EC<-ADD(EC,(22;PIERRE))))?
>>ERROR>>
NUMC:22NUMC:22VCHERCH:22;PIERRE
>>ERR-ASS>>
                    CHERCHS:EC:ADD(52;JEAN#54;CLAUDE#56;JACQUES#58;PA
UL#0,22;PIERRE)



RUN(PRETS(EP<-NIL))?
                    PRETS:EP:NIL

RUN(PRETS(EP<-ADD(EP,(1;52))))?
                    PRET:EP:(1;52).NIL

RUN(PRETS(EP<-ADD(EP,(7;56))))?
                    PRET:EP:(1;52).(7;56).NIL
```

```
MRUN(EACH(*L IN LIVRES(EL));
     PRINT(LIVRE(*L));
     LINE)?
                    LIVRE:1;MARTIN;INFO;COBOL
                    LIVRE:3;DUPOND;MATH;LOGIQUE
                    LIVRE:5;DURAND;INFO;PL1
                    LIVRE:7;CHABRIER;INFO;VEGA



MRUN(EACH(*C IN CHERCHS(EC));
    PRINT(CHERCH(*C));
     LINE)?
                    CHERCH:52;JEAN
                    CHERCH:54;CLAUDE
                    CHERCH:56;JACQUES
                    CHERCH:58;PAUL



MRUN(EACH(*P IN PRETS(EP));
     PRINT(PRET(*P));
     LINE)?
                    PRET:1;52
                    PRET:7;56



MRUN(EACH((*NL;*NC) IN PRETS(EP));
     FIRST((*N;*A;*S;*T) IN LIVRES(EL):
         INT(*N=*NL));
     PRINT(*S);
     LINE)?
                    INFO
                    INFO



 MRUN(EACH((*NL;*NC) IN PRETS(EP));
      FIRST((*NL;*A;*S;*T) IN LIVRES(EL));
      PRINT(*S);
      LINE)?
                     INFO
                     INFO



MRUN(EACH((*NL;*NC) IN PRETS(EP));
     FIRST((*NC;*NO) IN CHERCHS(EC));
     PRINT(*NO);
     LINE)?
                   JEAN
                   JACQUES
```



```
MRUN(EACH((*NC;*NO) IN CHERCHS(EC));
     PRINT(*NO);LINE;
     EACH((*NL;*NC) IN PRETS(EP));
     PRINT(*NL);
     LINE)?
                    JEAN
                    1
                    CLAUDE
                    JACQUES
                    7
                    PAUL



RUN(PRET(P2<-(1;4)))?
>>ERROR>>
NUMC:4NUMC:4VPRET:1;4
>>ERR-ASS>>
                    PRET:P2:(1;4)

INPUT/
```

Bibliographie

<ABR74> J.R.ABRIAL

"Data semantics"

IFIP TC2 1974 pp 1-59

<ABR77> J.R.ABRIAL

"Présentation du langage Z: manuel d'utilisation, mécanismes de transformations, applications"

Rapport non publié 1977

<ABR78> J.R.ABRIAL

"Z: a specification language"

IFIP TOKYO 1978

<ACM81> ACM Computing Surveys

Vol 13,3 Septembre 1981

<ADA80> Reference Manual for the ADA programming language

Proposed Standard Document Juillet 1980

<ADI78> M.ADIBA

"A relational model and an architecture for distributed database systems"

Thèse d'Etat, Grenoble 1978

<ADI81> M.ADIBA

"Derived relations: a unified mechanism for views, snapshots and distributed data"

IBM Research Laboratory RJ 2881 7/24/80

<ADI82> Appel d'offre génie logiciel

Agence de l'informatique Juin 1982

<ADJ76> J.W.THATCHER, E.G.WAGNER, J.B.WRIGHT

"Specification of abstract data types using conditional

axioms"

IBM Research report RC 6214 1976

<ADJ78> J.A.GOGUEN, J.W.THATCHER, E.G.WAGNER

"An initial algebra approach to the specification, correctness and implementation of abstract data types"

Current trends in programming methodology Vol 3 1978

<ALF77> M.W.ALFORD

"A requirements engineering methodology for real-time processing environments"

IEEE Transactions on Soft. Engineering

Vol SE-3,1 Janvier 1977 pp 60-69

<ALG68> Groupe Algol de l'AFCET

"Manuel du langage algorithmique Algol 68"

Hermann 1975

<ALL76> E.ALLMAN, M.R.STONEBRAKER, G.HELD

"Embedding a relational data sublanguage in a general purpose programming language"

SIGPLAN Notices 11 Mars 1976 pp 25-35

<ANS75> ANSI/X3/SPARC

"Interim report from the study group on data base management systems"

Bulletin of ACM Vol 7,2 1975

<ASH77> E.A.ASHCROFT, W.W.WADGE

"LUCID: a non procedural language with iterations"

CACM,20 1977 pp 519-526

<AST75> M.ASTRAHAN, R.LORIE

"SEQUEL-XRM, a relational system"

Proc. ACM Pacific Regional Conf. Avril 1975 pp 34-38

<AST76> M.ASTRAHAN et al

"SYSTEM R: relational approach to database management"

ACM Trans. Database System 1,2 Juin 1976 pp 97-137

<BAK72> F.T.BAKER

"Chief programmer team management of production program"

IBM System 11,1 1972 pp 56-73

<BAN77> M.BANATRE, A.COUVERT, D.HERMAN, M.RAYNAL

"Conservation d'objets typés"

Journées BIGRE 1977 INRIA

<BAN78> M.BANATRE, A.COUVERT, D.HERMAN, M.RAYNAL

"Langages de programmation et mécanismes de protection"

IRISA Publ. interne n.89 1978

<BAN78> M.BANATRE, A.COUVERT, D.HERMAN, M.RAYNAL

"Types abstraits et pluralité de leurs représentations à l'exécution"

Proc. of the 3th Symposium on Programming Paris 1978

<BAR77> W.BARTUSSEK, D.L.PARNAS

"Using traces to write abstract specifications for software modules"

University of North California report TR 77-012 Dec. 1977

<BAR78> G.B.BARBEYRE, T.JOUBERT, M.MARTIN

Manuel d'utilisation PROLOG/CNET

Version PASCAL/MULTICS

Note technique NT/PAA/CLC/LSC/541 Octobre 1981

<BAR82> G.BARBEYRE, T.JOUBERT, M.MARTIN

"OASIS: un outil d'aide à la spécification interactive et structurée"

Actes journées BIGRE Grenoble Janvier 1982

<BEE77> C.BEERI, R.FAGIN, J.HOWARD

"A complete axiomatization for functional and multivalued

dependencies"
Proc ACM SIGMOD Int. Conf. on Management of data Aout 1977 pp 47-61

<BEL78> F.BELLEGARDE, J.P.FINANCE, B.HUC, J.JARAY, J.MAROLDT, C.PAIR, J.L.REMY
"MEDEE: a type of language for the deductive programming method"
Conf. on reliable software German ACM Chapter Bonn 1978

<BEN79> G.BENCI, C.ROLLAND
"Bases de données: conception canonique pour une réalisation extensible"
SCM Paris 1979

<BER79> V.A.BERZINS
"Abstract model specifications for data abstractions"
PMD MIT 1979

<BER79> D.BERT
"La programmation générique, construction de logiciel, spécification algébrique et vérification"
Thèse d'Etat, Grenoble 1979

<BER81> M.BERGMAN, P.DERANSART
"Abstract data types ans rewriting systems: application to the programming of algebraic abstract data types in Prolog"
Proc. 6th CAAP LNCS 112 Genes 1981

<BER81> D.M.BERRY and S.D.M.
"The application of the formal development methodology to data base design and integrity verification"
UCLA Computer Science quartely report Vol 9,4 1981

<BER82> B.BERTHOMIEU
"Techniques algébriques pour la spécification et vérification

de protocoles de communication"
ler colloque de génie logiciel Juin 1982 pp 345-355

<BER82> D.BERT
"Langage de programmation générique"
Actes journées GROPLAN-GRECO programmation Bergerac 1982

<BID81> M.BIDOIT
"Une méthode de présentation des types abstraits: applications"
Thèse 3ème cycle Paris Juin 1981

<BIG79> T.J.BIGGERSTAFF
"The unified design specification system UDS2"
Boeing Computer Services Company
Washington 98124 1979

<BOD81> F.BODARD
"Eléments de conception et d'analyse des systèmes d'information des organisations"
Ecole d'Eté d'informatique de l'AFCET 1981

<BOE76> B.W.BOEHM
"Software engineering"
IEEE Trans. on Computers Vol C-25,12 Dec. 1976 pp 1226-1240

<BOE82> B.W.BOEHM
"Les facteurs du coût du logiciel"
T.S.I. Vol 1,1 Février 1982 pp 5-24

<BOU81> J.BOURST
"Projet Mont-Lozère: une approche système expert et types abstraits pour la promotion du tourisme en Lozère"
Rapport D.E.A. CRIN 1981

<BOY74> R.F.BOYCE, D.D.CHAMBERLIN
"SEQUEL: a structured english language"

ACM SIGMOD Workshop on data description, access and control
Mai 1974 pp 249-264

<BOY75> R.F.BOYCE, D.D.CHAMBERLIN, F.W.KING, M.M.HAMMER
"Specifying queries as relational expressions: the SQUARE data sublanguage"
CACM 18,11 1975 pp 621-628

<BRO78> M.L.BRODIE
"Specification and verification of data base semantic integrity"
PHD Thesis Toronto Avril 1978

<BRO82> M.L.BRODIE, E.SILVA
"Active and passive component modeling"
ACM/PCM IFIP 1982 pp 41-91

<BUC77> N.BUCKLE
"Restricted data types: specification and enforcement of invariant properties of variables"
SIGPLAN Notices Vol 12,3 Mars 1977 pp 68-76

<BUR77> R.M.BURSTALL, J.A.GOGUEN
"Putting theories together to make specifications"
Proc. 5th IJCAI 1977

<CAB75> A.CABANES
"Le modèle standard d'accès"
INFORSID groupe II Publ. IRIA Pont-à-Mousson 1975

<CAP78> M.CAPLAIN
"Langage de spécifications"
Thèse d'Etat, Grenoble 1978

<CAV79> J.L.CAVARERO
"LAPAGE: un modèle et un outil d'aide à la conception de systèmes d'information"

Thèse d'Etat, Nice 1979

<CHA73> C.CHANG, R.C.LEE
"Symbolic logic and mechanical theorem proving"
Academic Press 1973

<CHA76a> D.CHAMBERLIN, J.GRAY, I.L.TRAIGER
"Views, authorization and locking in a relational database management system"
AFIPS Data base management systems 1976 pp 425-430

<CHA76b> J.J.CHABRIER
"Utilisation de types abstraits dans les bases de données"
Journées d'informatique de Nice Juin 1976

<CHA76c> D.D.CHAMBERLIN et al
"SEQUEL2: a unified approach to data definition, manipulation and control"
IBM JRD 20,6 Novembre 1976 pp 560-575

<CHA78> J.J.CHABRIER
"Construction structurée de systèmes d'information des organisations"
Actes des journées BIGRE Nancy Janvier 1979

<CHA82> J.CHABRIER
"Présentation et utilisation du langage PROLOG"
Rapport CRIN 1982 n.82-R-078

<CHA82> J.J.CHABRIER, J.C.DERNIAME
"TYP: un système de programmation modulaire utilisant des types abstraits de données"
Actes 1er colloque génie logiciel Paris 1982

<CHA82> J.J.CHABRIER, J.C.DERNIAME
"Programmation à l'aide de types abstraits: le système TYP"
T.S.I. Vol 1,4 1982 pp 287-295

<CHA82> J.J.CHABRIER, J.CHABRIER

"Présentation de VEGA"

Journées GROPLAN-GRECO programmation Bergerac 1982

<CHA82> J.J.CHABRIER, J.CHABRIER et al

"VEGA: un système d'aide à l'information touristique pour la région de Lozère"

Projet Mont-Lozère ADI Juin 1982

<CHE76> P.CHEN

"The entity-relationship model toward a unified view of data"

ACM Trans. on data base systems Vol 1 Mars 1976 pp 9-36

<CHE77> P.CHEN

"The entity-relationship model: a basis for the enterprise view of data"

AFIPS 1977 pp 77-84

<COD70> E.F.CODD

"A relational model of data for large shared data banks"

CACM 13 Juin 1970 pp 377-387

<COD71a> E.F.CODD

"A data sublanguage founded on the relational calculus"

Proc. of the ACM SIGFIDET Workshop on data description, access and control 1971 pp 35-68

<COD71b> CODASYL DBTG

"Report on the CODASYL data base task group"

ACM New York 1971

<COL82> A.COLMERAUER

"PROLOG II: manuel de référence et modèle théorique"

Université d'Aix-Marseille Luminy Mars 1982

<COM76> Computing Surveys

Vol 8,1 Mars 1976

<CRE75> M.CREHANGE

"Description formelle, représentation, interrogation des informations complexes: système PIVOINES"

Thèse d'Etat, Nancy 1975

<CUN79> P.Y.CUNIN

"Fiabilité et sécurité des programmes: proposition autour d'un langage d'essai"

Thèse d'Etat, Nancy Mars 1979

<CUN80> P.Y.CUNIN, M.GRIFFITHS, J.VOIRON

"Apprendre la compilation"

CRIN-Grenoble 1980

<DAH70> O.J.DAHL, B.MYHRHANG, K.NYGAARD

"The SIMULA 67 common base language"

Publ. S22 Norvegian Comp. Center OSLO 1970

<DAH72> O.J.DAHL, E.W.DIJKSTRA, C.A.R.HOARE

"Structured programming"

Academic Press N.Y.1972

<DAR78> P.DARONDEAU

"Objets et types dans un système multi-langages"

Thèse d'Etat, Grenoble 1978

<DAT77> C.J.DATE

"An introduction to data base systems"

Second edition Addison Wesley 1977

<DAV80> R.E.DAVIS

"Runnable specification as a design tool"

Santa Clara CA 950053

<DEL80> C.DELOBEL

"An overview of the relational data theory"

Proc. of the IFIP Conf. TOKYO 1980

<DEM75> R.DEMOLOMBE, A.LEMAITRE, J.M.NICOLAS
"SYNTEX 2"
Rapport DRME n.74222 CERT Toulouse 1975

<DEN73> J.B.DENNIS
Advanced Course on Software Engineering
Berlin Springer Verlag n.81 1973 pp 128-182

<DEN79> D.E.DENNING, P.J.DENNING
"Data security"
Computing Surveys Vol 11,3 1979

<DER74> J.C.DERNIAME
"Le projet CIVA: un système de programmation modulaire"
Thèse d'Etat, Nancy 1974

<DER79> J.C.DERNIAME, J.P.FINANCE
"Types abstraits de données: spécification, utilisation et réalisation"
Ecole d'Eté AFCET Monastir 1979

<DER82> P.DERANSART
"Dérivation de programmes PROLOG à partir de spécifications algébriques"
Rapport INRIA 1982

<DIJ72> E.W.DIJKSTRA
"Notes on structured programming"
Structured Programming Academic Press 1972

<DUF80> J.F.DUFOURD
"Maquettes pour évaluer les systèmes d'information des organisations"
Thèse d'Etat, Nancy Janvier 1980

<EAR73> J.EARLEY
"Relation-level data structures for programming languages"

Acta Informatica Vol 2 pp 293-309 1973

<EAR74> J.EARLEY
"High level operations in automatic programming"
Proc. Symp. Very High Level Lang.
SIGPLAN Notices 9 1974 pp 34-42

<EHR81> H.EHRIG
"Algebraic theory of parametrized specifications with requirements"
Proc. 6th CAAP LNCS 112 Genes 1981

<EMD79> M.H.VAN EMDEN
"Computation and deductive information retrieval"
Formal description of programming concepts
Neuhold Editions North Holland

<ENG74> D.M.ENGLAND
"Capability concept mechanism and structure in System 250"
Int. Workshop protection in Operating Systems IRIA 1974

<FER75> J.FERRIE
"Contrôle de l'accès aux objets dans les systèmes informatiques"
Thèse d'Etat, Paris VI 1975

<FIN79> J.P.FINANCE
"Etude de la construction des programmes: méthodes et langages de spécification et de résolution de problèmes"
Thèse d'Etat, Nancy 1979

<FLO67> R.W.FLOYD
"Assigning meanings to programs"
Proc. of the Symp. in Applied Mathematics
Mathematical Aspect of Computer Science 19
American Mathematical Society 1967 pp 19-32

<FLO77> A.FLORY

"Un modèle et une méthode pour la conception logique d'une base de données"

Thèse d'Etat, Lyon 1977

<FOI82> J.FOISSEAU

"Assistance à la spécification des fonctions et des types de données dans SPRAC"

Actes journées BIGRE Grenoble Janvier 1982

<FOU82> O.FOUCAUT

"Modèle et outil pour la conception des systèmes d'information dans les organisations; projet REMORA"

Thèse d'Etat, Nancy Juin 1982

<GAL76> M.GALINIER et al

"Programmation structurée"

Rapport 112 Université Paul Sabatier Toulouse

<GAL78> H.GALLAIRE, J.MINKER

"Logic and data bases"

Plenum Press 1978

<GAU80> M.C.GAUDEL

"Génération et preuve de compilateurs basés sur une sémantique formelle des langages de programmation"

Thèse d'Etat, Nancy 1980

<GER79> S.L GERHART

"An experiment in data representation verification"

Draft Septembre 1979 4 SC Inf. Sc. Institute

<GER79> S.L.GERHART, D.S.WILE

"Preliminary report on the delta experiment"

Specification on reliable software

IEEE Catalog 79 Avril 1979 pp 198-211

<GES77> M.GESCHKE, J.H.MORRIS, E.H.SATTERTHWAITE

"Early experience with MESA"

SIGPLAN Notices Vol 12,3 Mars 1977 pp 138

<GOD81> C.GODART

"SGMB: un système de gestion multibase"

Thèse 3ème cycle Nancy 1981

<GOG75> J.A.GOGUEN, J.W.THATCHER, E.G.WAGNER, J.W.WRIGHT

"Abstract data types as initial algebras and the correctness of data representations"

Proc. of Conf. on Comp. Graphics Beverly Hills pp 89-93

<GOG78> J.A.GOGUEN

"Order sorted algebras: exceptions and errors sorts, coercions and overloaded operators"

Report R14 Décembre 1978

University of California Los Angeles (CSD)

<GOG78> J.A.GOGUEN

"Abstract errors for abstract data types"

Formal descriptions of programming concepts

North Holland Pub. Comp. 1978 pp 491-525

<GOG79> J.A.GOGUEN, J.J.TARDO

"An introduction to OBJ: a language for writing and testing formal algebraic program specifications"

IEEE Conf. on Specifications of reliable software

Cambridge Avril 1979 pp 170-189

<GOG80> J.A.GOGUEN

"How to prove algebraic inductive hypothesis without induction, with applications to the correctness of data type implementation"

Proc. 5th Conf. on automated deduction

Les Arcs LNCS 87 1980

<GOO73> G.GOOS

"Documentation"

Springer Verlag Berlin n.81 1973 pp 385-393

<GOO75> J.B.GOODENOUGH

"Exception handling: issues and a proposed notation"

CACM Vol 18,12 pp 683-696

<GOO78> G.GOOS, U.KASTENS

"Programming languages and the design of modular programs"

Constructing quality software North Holland 1978

<GOO79> D.I.GOOD. R.M.COHEN, J.KEETON-WILLIAMS

"Principles of proving concurrent programs in GYPSY"

Proc. 6th POPL Janvier 1979

<GRA81> M.GRANDBASTIEN

"A propos de concept de spécification dans la littérature informatique"

Rapport CRIN n.81-R-054

<GRI76> P.P.GRIFFITHS, B.W.WADE

"An authorization mechanism for a relational database system"

ACM Trans. Database Syst. Vol 1,3 1976 pp 242-255

<GRI81> T.GRISON

"Génération automatique des textes ATM implantant une base de données relationnelle"

Rapport D.E.A. CRIN 1981

<GRO79> GROPLAN

"Panorama des langages d'aujourd'hui"

Cargèse Mai 1979

<GUT78> J.V.GUTTAG, J.J.HORNING

"The algebraic specification of abstract data types"

Acta Informatica 10 1978 pp 27-52

<GUT78> J.V.GUTTAG, E.H.HOROWITZ, D.R.MUSSER

"Abstract data types and software validation"

CACM Vol 21 Décembre 1978 pp 1048-1064

<GUT80> J.V.GUTTAG, J.J.HORNING

"Formal specification as a design tool"

CACM 1980 pp 251-262

<GUT80> J.V.GUTTAG

"Notes on type abstraction"

IEEE Trans. on Soft. Eng. SE6,1 Janvier 1980 pp 13-23

<GUY82> J.GUYARD

"MAIDAY"

Actes journées GROPLAN-GRECO programmation Bergerac 1982

<HAB73> A.N.HABERMANN

"Critical comments on the programming language PASCAL"

Acta Informatica 3 pp 47-57

<HAN79> HANSON, JACKSON, JULLIG, LEVY, PITTMANN

"Summary of the caracteristics of several "modern" programming languages"

SIGPLAN Notices Juin 1979 pp 28-44

<HEL75> G.D.HELD, M.R.STONEBRAKER, E.WONG

"INGRES: a relational data base system"

Proc. 1975 AFIPS Nat. Comp. Conf. Vol 44 N.J. pp 409-416

<HEN80> K.L.HENINGER

"Specifying software requirements for complex systems: new techniques and their applications"

IEEE Trans. on Soft. Engin. Vol 6,1 pp 2-12

<HEN80> P.HENRY

"La connexion dans le projet TYP"

Thèse 3ème cycle Nancy 1980

<HOA72a> C.A.R.HOARE

"Proof of correctness of data representations"

Acta Informatica Vol 1 pp 272-281 1972

<HOA72b> C.A.R.HOARE

"Notes on data structuring"

Structured Programming Academic Press 1972

<HOR77> J.J.HORNING

"Separate compilation for the SUE system language"

IFIP W.G.2.4 MOL Bulletin n.4 Février 1977 pp 13-15

<HOW82> W.E.HOWDEN

"Contemporary software development environments"

CACM Mai 1982 Vol 25,5 pp 318-325

<HUE80> G.HUET, D.C.OPPEN

"Equations and rewrite rules: a survey"

Formal languages: perspectives ans open problems

R.BOOK Ed. Academic Press 1980

<HUE80> G.HUET

"A complete proof of correctness of the Knuth-Bendix

completion algorithm"

Rapport n.25 INRIA 1980

<HUE80> G.HUET

"Confluent reductions: abstract properties and applications to

term rewriting systems"

J.Assoc. Comp. 27,4 pp 797-821 Mars 1980

<HUE82> G.HUET

"Le système FORMEL"

Actes journées GROPLAN-GRECO programmation Bergerac 1982

<HUL80> J.M.HULLOT

"Compilation de formes canoniques dans les théories

équationnelles"

Thèse 3ème cycle Paris Sud 1980

<ICH75> J.D.ICHBIAH, RISSEN, J.P.HELIARD, P.COUSOT

"The system implemantation language LIS"

Ref. manual CII 4549 E.N. 1975

<IFIP76> IFIP TC2 Working Conference

"Modelling in data base management systems"

Freundenstadt 1976

<IMP69> M.D'IMPERIO 1969

<INF75> INFORSID

"Séminaire sur le langage d'analyse et de conception des

systèmes d'information automatisés"

IRIA Pont-à-Mousson 1975

<INF77> INFOTECH International

"La programmation structurée"

Infotech State of the Art Report 1977

<JAC75> M.A.JACKSON

"Principles of program design"

Academic Press 1975

<JAC78> P.JACQUET

"Les types génériques. Propositions pour un mécanisme

d'abstraction dans les langages de programmation"

Thèse 3ème cycle Grenoble Septembre 1978

<JON76> A.K.JONES, B.H.LISKOV

"A language extension for controling access to shared data"

IEEE Trans. on Soft. Engineering Vol SE2,4 1976

<KAH82> G.KAHN, P.LEPETIT

"Le projet japonais de recherche pour les ordinateurs de la

5ème génération"

Conf. TOKYO MITI Octobre 1981

<KAP80> D.KAPUR

"Towards a theory for data abstractions"

PHD Thesis MIT 1980

<KEL78> C.KELLOG, P.KLAHR, L.TRAVIS

"Deductive planning and path finding for relational data bases"

Logic and data bases

Plenum Press New-York 1978 pp 179-200

<KIM79> W.KIM

"Relational data systems"

Comp. Surveys Vol 11,3 Septembre 1979 pp 185-211

<KIR82> C.et H.KIRCHNER

"Contribution à la résolution d'équations dans les algèbres libres et les variétés équationnelles d'algèbres"

Thèse 3ème cycle Nancy Mars 1982

<KNU68> D.E.KNUTH

"The art of programming"

Addison Wesley 1968

<KNU70> D.E.KNUTH, P.B.BENDIX

"Simple word problems in universal algebra"

Computational problems in abstract algebra

J.LEECH Ed. Pergamon Press 1970 pp 263-297

<KOW74> R.A.KOWALSKI

"Logic for problem-solving"

Memo n.75 Departement of computational logic

University of Edimburg 1974

<KOW76> R.A.KOWALSKI, VAN EMDEN

"The semantics of predicat logic as programming language"

JACM 23 Octobre 1976 pp 733-742

<KRA80> S.KRAKOWIAK

"Systèmes intégrés de production de logiciel"

Journées génie logiciel INRIA 1980

<KRA82> S.KRAKOWIAK

"Systèmes intégrés de production de logiciel: concepts et réalisations"

T.S.I. Vol 1,3 pp 187-200

<KRE67> G.KREISEL, J.L.KRIVINE

"Eléments de logique mathématique"

Ed Dunod Paris 1967

<LAM69> B.W.LAMPSON

"Dynamic protection structures"

Proc. AFIPS FJCC Vol 35 1969

<LAM73> B.W.LAMPSON

"A note on the confinement problem"

CACM Vol 16,10 pp 613-615 Octobre 1973

<LAM77> B.W.LAMPSON et al

"The programming language EUCLID"

M.O.L. Bulletin n.5 IRIA 1976

<LAM77> B.W.LAMPSON et al

"EUCLID report"

SIGPLAN Notices 12 Février 1977

<LAU76> J.L.LAURIERE

"Un langage et un programme pour énoncer et résoudre des problèmes combinatoires"

Thèse d'Etat, Paris VI 1976

<LAU82> J.L.LAURIERE

"Représentation et utilisation des connaissances: les systèmes experts" (première partie)

"Représentation des connaissances" (deuxième partie)

T.S.I.Vol 1 et 2 1982

<LES79> P.LESCANNE

"Etude algébrique et relationnelle des types abstraits et leurs représentations"

Thèse d'Etat Nancy 1979

<LES82> P.LESCANNE

"Computer experiments with the Reve term rewriting system generator"

POPL Austin 1983

<LEV77> G.LEVY, F.SIROVITCH

"TEL: a proof theoric language for efficient symbolic expression manipulation"

Nota Interna 877-3 IEI Pisa Février 1977

<LEV79> H.LEVESQUE

"The representation of incomplete knowledge as applied to a logic of sentences"

AI Memo 79.1 Dept. of Comp. Science Toronto 1979

<LEV79> K.N.LEVITT, L.ROBINSON, B.A.SILVERBERG

"The HDM handbook" Vol 1,3

Comp. Science Lab. SRI International Juin 1979

<LIS74> B.H.LISKOV

"A note on CLU"

Comp. structures group memo 112

MIT Project MAC Cambridge Mass 1974

<LIS74> B.H.LISKOV, S.N.ZILLES

"Programming with abstract data types"

SIGPLAN Notices 9 1974

<LIS77> B.H.LISKOV, S.N.ZILLES

"An introduction to formal specifications data abstractions"

Current trends in programming methodology

R.T.YEH Ed. Vol 1 Prentice Hall 1977

<LIS77> B.LISKOV

"Abstraction mechanism in CLU"

CACM 20,8 Aout 1977 pp 564-576

<LIS79> B.H.LISKOV, A.S.SNYDER

"Exception handling in CLU"

IEEE Trans. on software Eng. Vol 5,6 pp 547-557

<LIS81> B.H.LISKOV et al

"CLU reference manual"

Springer Verlag Berlin 1981 Vol 103

<LIV78> C.LIVERCY

"Théorie des programmes"

DUNOD 1978

<MAJ77> M.E.MAJSTER

"Limits of the "algebraic" specification of abstract data types"

SIGPLAN Notices 10,12 1977 pp 37-42

<MEY75> G.J.MEYERS

"Reliable software through composite design"

Petrocelli/Charter New-York 1975

<MIC79> D.MICHIE

"Expert systems in the micro electronic age"

Edited by D.MICHIE

<MIL71> H.D.MILLS

"Chief programmer teams: principles and procedures"

IBM Report FSC 71-S108 1971

<MIL71> H.D.MILLS

"Top-down programming in large systems"

Debugging techniques Prentice Hall 1971

<MIN73> J.MINKER, D.H.FISHMAN, M.C.SKIMIN

"The Q* algorithm -a search strategy for a deductive question answering system"

Artificial Intelligence 4 1973 pp 225-243

<MIN75> M.MINSKY

"A framework for representing knowledge"

Winston 75 pp 211-217

<MIN76> M.MINSKY

"International resolution of privacy protection in database systems"

CACM Vol 19,3 1976 pp 148-159

<MIN79> R.MINOT

"ATM: un système de fabrication de programmes basés sur les concepts de modularité et de type abstrait"

Thèse 3ème cycle Nancy Mars 1979

<MIR80> S.MIRANDA

"Architecture formelle d'un système d'intégrité pour une base de données réparties"

Thèse d'Etat, Toulouse 1980

<MOR73> J.MORRIS

"Protection in programming languages"

CACM Vol 16,1 pp 15-21 Janvier 1973

<MOR73> J.H.MORRIS

"Types are not sets"

Proc. 1st ACM Symp. on principles of programming languages

Boston Octobre 1973 pp 120-134

<MUS78> D.R.MUSSER

"Abstract data type specification in the AFFIRM system"

IEEE Trans. Soft. Eng. SE6,1 Janvier 1980 pp 24-32

<MYL79> J.MYLOPOULOS, P.A.BERNSTEIN, H.K.T.WONG

"A language facility for designing interactive database intensive applications"

Université de Toronto CSRG 105

<NAK78> R.NAKAJIMA, M.HONDA, H.NAKAHARA

"Describing and verifying programs with abstract data types"

Formal description of programming concepts North Holland 1978

<NAU63> P.NAUR

"Revised report on the algorithmic language ALGOL 60"

CACM Vol 6,1 pp 1-17 1963

<NAU68> P.NAUR, B.RANDELL

"Software engineering"

Brussels NATO Scientific Affairs Division 1968

<NEU74> E.J.NEUHOLD

"The use of formal description techniques in large data banks"

IBM Lab. Vienne Austria 1974

<NOT72> M.G.NOTLEY

"The Peterley IS/1 system"

Rep UKSC-18 IBM UK Scientific Center Peterley, England Mars 1972

<PAI74> C.PAIR

"Some theorical aspects of the notions of data, information and information structure"

Data base management systems

Klimbie-Koffman North Holland 1974

<PAI75> C.PAIR

"Some proposals for a very high level language on a variable universe"

New directions in algorithmic languages INRIA 1975

<PAI79> C.PAIR

"La construction de programmes"

RAIRO Informatique Vol 13,2 1979 pp 113-118

<PAI80> C.PAIR

"Sur les modèles de types abstraits algébriques"

CRIN 80-P-052 1980

<PAR71> D.L.PARNAS

"Information distribution aspects of design methodology"

Congrès IFIP TA-3 1971 pp 26-30

<PAR72> D.L.PARNAS

"On the criteria for decomposing systems into modules"

CACM 15,12 Décembre 1972 pp 1053-1058

<PAR77> D.L.PARNAS

"The use of precise specifications in the development of software"

IFIP 1977 pp 861-867

<PIN81> PINGREE PARK

Proc. of the workshop on "data abstraction, database and conceptual modeling"

Pingree Park Colorado Juin 1980

SIGPLAN Vol 16,1 Janvier 1980

<PIR74> A.PIROTTE, P.WODON

"A comprehensive formal query language for a relational database: FQL"

Technical report R283 M.B.L.E. Brussels Belgium Décembre 1974

<PIR76> A.PIROTTE

"Explicit description of entities and their manipulation in languages for the relational data base model"

Thèse Université libre de Bruxelles Belgique 1976

<POO73> P.C.POOLE

"Debugging and testing"

Advanced course on software engineering"

Berlin Springer Verlag n.81 pp 278-372

<PRO79> K.PROCH

"Paramétrisation des unités de compilation en ATM"

Publ. CRIN 79-R-058

<PRO82> K.PROCH

"ORSEC: un outil de recherche de spécifications équivalentes par comparaison d'exemples"

Thèse 3ème cycle Nancy Décembre 1982

<REI77> R.REITER

"An approach to deductive question-answering"

Bolt Beranek and Newman Inc, Cambridge Mass Tech. report 3649 pp 161

<REM82> J.L.REMY

"Etude des systèmes de réécriture conditionnels et applications aux types abstraits algébriques"

Thèse d'Etat Nancy 1982

<RIN79> O.RINES, T.AMBLE, K.BRATBERGENSEN

"ASTRAL: a structured and unified approach to data base design and manipulation"

IFIP TC 2 Work. Conf. Venise Juin 1979

<ROB65> J.A.ROBINSON

"A machine oriented logic based on the resolution principle"

JACM 12,1 1965 pp 23-41

<ROS70> D.ROSS

"Uniform referents: an essential property for a software engineering language"

Software engineering AP 1970 pp 91-101

<ROS77> D.T.ROSS, K.E.SHOMAN

"Structured analysis for requirements definition"

IEEE Trans. on Soft. Eng. SE-3 1 Janvier 1977

<ROU75> P.ROUSSEL

"PROLOG: manuel de référence et d'utilisation"

Groupe d'intelligence artificielle U.E.R. Luminy

Université d'Aix-Marseille Septembre 1975

<ROW79> L.A.ROWE, K.A.SHOENS

"Data abstraction, views and updates in RIGEL"

ACM SIGMOD Mai 1979 pp 71-80

<SCH72> M.D.SCHROEDER, J.H.SALTZER

"A hardware architecture for implementing protection rings"

CACM Vol 15,3 pp 157-170 Mars 1972

<SCH75> J.T.SCHWARTZ

"On programming: an interim report on the SETL project"

Courant Institute of Mathematical Science New-York University 1975

<SCH77> J.SCHMIDT

"Some high level language construct for data of type relation"

ACM TODS 2,3 Septembre 1977

<SEN73> SENKO, ALTMAN, ASTRAHAN, FEDHER

"Data structures and accessing in data base systems"

IBM System Journal Vol 12,1 1973 pp 30-93

<SHA77> M.SHAW, W.A.WULF

"Abstraction and verification in ALPHARD: defining and specifying iteration and generators"

CACM Vol 20,8 1977 pp 358-367

<SHN76> B.SHNEIDERMAN

"Database management systems"

AFIPS Press Montvale New Jersey 1976

<SIG76> SIGPLAN Notices

Proc. of Conf. on "data: abstraction, definition and structure"

Vol 8,2 1976

<SIG77> SIGPLAN Notices

Proc. of an ACM Conf. on "language design for reliable software"

Vol 12,3 1977

<SIR81> Journées SIRIUS Paris 1981

<SMI77> J.M.SMITH, D.C.P.SMITH

"Database abstractions: aggregation and generalisation"

ACM Trans. on database systems Vol 2,2 pp 105-133 Juin 1977

<SPR73> SPRINGER VERLAG

Advanced course on software engineering Berlin n.81 1973

<SIE74> J.SIEYERT, J.GOLDMAN

"The relational data management system: a perspective"

ACM SIGMOD Workshop on data description, access and control 1974 pp 295-320

<STO76> M.STONEBRAKER, E.WONG, P.KREPS, G.HELD

"The design and implementation of INGRES"

ACM Trans. database systems 1,3 Septembre 1976 pp 189-222

<STO77> M.R.STONEBRAKER, L.A.ROWE

"Observations on data manipulation languages and their

embedding in general purpose programming language"

VLDB Tokyo Octobre 1977

<SWA76> E.B.SWANSON

"The dimension of maintenance"

Proc. 2nd Int. Conf. on Soft. Eng. Octobre 1976 pp 492-497

<TEI75> D.TEICHROEW, M.J.BASTARACHE

"PSL user's manual"

ISDOS Working paper n.98 Mars 1975

<TOU79> N.TOUNSI

"Itération et itérateurs d'ATM"

Rapport D.E.A. Nancy Juin 1979

<TSI73> D.TSICHRITZIS

"Reliability"

Advanced course on software engineering

Berlin Springer Verlag n.81 pp 319-379

<TSO79> J.K.TSOTSOS, J.MYLOPOULOS, H.D.COWEY

"A framework for visual notion understanding"

AI Memo 79-2 Juin 1979 Toronto Canada

<VEL79> P.A.S.VELOSO

"Traversable stack with fewer errors"

SIGPLAN Notices 14,2 1979 pp 55-59

<VLDB78> Very Large Data Bases

West-Berlin Germany Septembre 13,15 1978

<WAR77> D.H.WARREN

"Implementing PROLOG"

DAI research report Vol 1 et 2

University of Edimburgh 1977

<WAR77> D.H.WARREN, L.M.PEREIRA, F.PEREIRA

"PROLOG: the language and its implementation compared with

LISP"

Proc. Symp. AI and programming languages

SIGPLAN Notices Vol 12,8 1977 pp 109-115

<WAS78> A.I.WASSERMAN

"A software engineering view of data base management"

IEEE 1978 pp 23-34

<WAS79> A.I.WASSERMAN

"The data management facilities of PLAIN"

ACM SIGMOD Mai 1979

<WAS82> A.I.WASSERMAN, S.GUTZ

"The future of programming"

CACM Vol 25,3 Mars 1982 pp 196-206

<WEB78> H.WEBER

"A software engineering view of data base systems"

VLDB West Berlin Germany Septembre 1978 pp 35-51

<WEB79> H.WEBER

"Modularity in data base system design"

Issues in data base management

Wasserman Ed. North Holland 1979

<WEG74> B.WEGBREIT

"The treatment of data types in ELl"

CACM Vol 17,5 1974 pp 251-264

<WIR71> N.WIRTH

"Program development by stepwise refinement"

CACM 14,4 Avril 1971 pp 221-227

<WIR71> N.WIRTH

"The programming language PASCAL"

Acta Informatica 1,1 1971 pp 35-63

<WIR76> N.WIRTH

"NODULA, a language for modular programming"

ETH Zurich 1976

<WUL74> W.WULF et al

"HYDRA: the kernel of a multiprocessor operating system"

CACM Vol 17,6 pp 337-345 Juin 1974

<WUL75> W.WULF, R.L.LONDON, M.SHAW

"Abstraction and verification in ALPHARD"

New directions in algorithmic languages

IFIP W.G.2-1 Munich 1975

<WUL78> W.WULF

"An informal definition of ALPHARD"

Computer Science Dept. CMU-CS 78-105

Carnegie Mellon University Février 1978

<ZEL78a> M.V.ZELKOWITZ

"Implementation of a capability based data abstraction"

IEEE Trans. Soft. Eng. Vol SE4,1 Janvier 1978

<ZEL78b> M.V.ZELKOWITZ

"Perspectives on software engineering"

Computing Surveys Vol 10,2 Juin 1978 pp 197-216

<ZIL80> S.N.ZILLES

"Types, algebras and modeling"

SIGPLAN Notices Vol 16,1 Janvier 1980 pp 207-209

<ZLO75> M.ZLOOF

"Query by example"

Proc. 1975 AFIPS Comp. Conf. Vol 44 N.J. pp 431-438

NOM DE L'ETUDIANT : CHABRIER Jean-Jacques

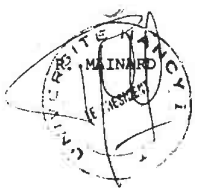
NATURE DE LA THESE : Doctorat ès sciences

VU, APPROUVE

ET PERMIS D'IMPRIMER

NANCY, le 27 OCT. 1982 2578

LE PRESIDENT DE L'UNIVERSITE DE NANCY I,



RESUME

SPECIFICATION ET CONSTRUCTION DE SYSTEMES ORIENTES BASES DE DONNEES: TECHNIQUES ET LANGAGES BASES SUR LE CONCEPT DE TYPE ABSTRAIT

Concevoir et produire de façon économique des systèmes logiciels d'application de bonne qualité est un des objectifs essentiels des recherches actuelles en génie logiciel. Notre travail vise à fournir une approche globale, cohérente du développement de systèmes logiciels d'application orientés bases de données.

Dans la première partie, on commence par rappeler quelles sont les qualités attendues des produits fabriqués, les étapes du développement d'un logiciel, puis on étudie de façon détaillée les problèmes rencontrés au cours de la conception et de la production des systèmes d'application. On expose ensuite une synthèse de différents langages (ou systèmes) pouvant supporter les concepts d'abstraction de données et/ou de type abstrait de donnée, des mécanismes de modularité, des techniques de construction par décomposition et raffinement: langages généraux, spécifiques aux bases de données ou encore mixtes.

Dans la deuxième partie, on aborde une étude de la spécification de problèmes avant de présenter une synthèse de travaux théoriques récents sur la spécification de types abstraits avec erreurs, les systèmes de réécriture, l'unification et la résolution en programmation logique, ayant permis de définir le système VEGA d'aide à la spécification algébrique (de types abstraits et/ou de programmes) et à la validation de la fidélité de la conception aux intentions initiales des utilisateurs.

VEGA est un système interactif opérationnel de spécification et de programmation comportant un environnement de spécification: caractère, booléen, entier, chaîne, constructeur de produit cartésien labellé, constructeur d'ensemble, constructeur de séquence,etc... particulièrement adapté à la conception d'applications orientées bases de données. Un langage interprété d'aide à l'élaboration de maquettes de données symboliques est aussi intégré au système VEGA; ceci permet de valider facilement les choix de conception effectués.

La troisième partie présente les concepts, les mécanismes et les langages du système interactif de programmation modulaire TYP. Ce système intègre un système de compilation séparée, un documentaliste, un gestionnaire, un système de généricité, un éditeur de textes et un système de fabrication de programmes (connecteur).

Cet ensemble de propositions doit permettre de disposer d'un système intégré de conception et de production offrant un continuum d'expression au long des étapes de résolution d'un problème.