Une étude sur la base de la programmation algorithmique : notation et environnement de travail

Philippe Morat

THESE

présentée à

l'Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR DE 3ème CYCLE

Informatique

par

Philippe MORAT

UNE ETUDE SUR LA BASE DE LA PROGRAMMATION ALGORITHMIQUE : NOTATION ET ENVIRONNEMENT DE TRAVAIL

Thèse soutenue le 9 décembre 1983 devant la Commission d'Examen :

Monsieur	L. BOLLIET	: Président
Messieurs	M. ADIBA P.C. SCHOLL L. TRILLING G. VEILLON	Examinateurs

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

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

.../...

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

PROFESSEURS ASSOCIES

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

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

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

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

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

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

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

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

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

.../...

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

.../...

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

LABORATOIRES EXTERIEURS

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

Je tiens à remercier :

- Monsieur le Professeur Louis BOLLIET, Directeur de l'I.U.T. Informatique de Grenoble, pour l'honneur qu'il me fait de bien vouloir présider le jury de cette thèse. Je profite de ces quelques lignes pour rapeller qu'il guida mes premiers pas en informatique, ce qui fût déterminant par la suite. J'ai donc de multiples raisons de le remercier ici, et je le fait avec le plus grand plaisir.

- Monsieur Michel ADIBA, Professeur à l'Université scientifique et médicale de Grenoble, d'avoir accepté de juger ce travail, et pour les encouragements qu'il m'a prodigués tout au long de la préparation de cette thèse. Qu'il soit certain de l'amitié que je lui témoigne.

- Monsieur Pierre-Claude SCHOLL, Professeur à l'Université scientifique et médicale de Grenoble, pour m'avoir accepté dans son équipe à une époque où son attention était requise par d'autres soucis. Ses critiques ont été, pour moi, certes des éléments de remise en cause permanents, mais surtout une source indispensable à la progression de mon travail. Son expérience de l'enseignement, et sa façon d'aborder les problèmes ont été pour moi décisives. Ici, je tiens à le remercier pour ces années de travail passées ensemble.

- Monsieur Laurent TRILLING, Professeur à l'Université de Rennes, d'avoir accepté de participer au jury de cette thèse.

- Monsieur Gérard VEILLON, Directeur de l'Ecole Nationale Supérieure d'Informatique et Mathématiques Appliquées, pour l'intérêt qu'il a porté à mon travail, et pour l'honneur qu'il m'a fait en acceptant de participer au jury de cette thèse.

Le travail présenté dans cette thèse est le fruit d'une longue collaboration étroite avec Claude CASERY. Les nombreuses discussions que nous avons eues ont été pour moi une source précieuse d'inspiration. Sans lui, ce travail n'aurait jamais trouvé cette concrétisation, je l'en remercie vivement.

Je remercie également toutes les personnes de l'équipe pour toutes les réunions de toutes natures que nous avons faites ensemble, j'espère que cette tradition survivra...

Je voudrais terminer en dédiant ces quelques mots à ma femme et ma fille qui ont pendant l'élaboration de cette thèse supporté mes nombreuses absences, et m'ont toujours chaleureusement soutenu.

sommaire

1 Introduction

Contexte du travail présenté

Une notation algorithmique est un intermédiaire entre les méthodes d'analyse et de conception d'algorithmes, et l'environnement informatique où se réalise cette activité de programmation. Il s'agit de donner au programmeur un support pour concrétiser son travail, sans avoir à aborder la problématique des langages et des systèmes.

Utiliser une démarche méthodique dans la programmation consiste à s'appuyer sur des méthodes existantes, et à utiliser systématiquement des techniques bien définies [Abr 81] [CGL 75] [CoV 74] [DDH 72] [GaM 78] [Ger 77] [LPS 83] [Sch 79] [Wir 71b]. L'intérêt d'une telle démarche est de produire des programmes fiables, d'aider à la résolution de problèmes complexes et d'assurer, tout au long du processus de construction, un guide à l'analyse.

Pour étudier de façon plus précise l'impact d'une méthodologie dans le cadre de l'enseignement de la programmation, nous avons décidé de réaliser un traducteur d'une notation algorithmique vers un langage de programmation cible utilisable sur un ordinateur de notre environnement de recherche.

Dans cette démarche on cherche à atteindre deux objectifs distincts.

> D'une part, offrir aux étudiants un environnement de travail correspondant à ce qui leur était enseigné [Che 76] [Ger 77] [LuS 75] [Sch 77b]. Ceci nécessite de fournir les moyens d'écrire les programmes conçus dans la notation algorithmique, de les exécuter sur un site informatique.

D'autre part, de compléter ou de modifier au vu de l'expérience acquise, la notation algorithmique, d'affiner les concepts et les méthodes employés, et d'établir un ensemble d'éléments complémentaires servant à étayer les concepts et les techniques vus par ailleurs.

Il a été envisagé, dès le début, une traduction multi-cible de notre notation :

D'abord pour plus de commodité (nous ne voulions pas être tributaire du langage hôte, et l'étude de plusieurs traductions nous apparaissait utile).

Ensuite pour étudier la relation entre notation algorithmique et langage de programmation.

Enfin, dans le cadre de l'enseignement, pour concrétiser cette idée de la programmation (montrer les différents aspects des langages).

Démarche suivie

Ecrire un traducteur de la notation algorithmique vers un langage cible est en lui-même un problème algorithmique. Il nous a paru intéressant d'aborder la résolution de ce problème en appliquant les méthodes qui sont à la base de la notation algorithmique.

Ceci nous a conduit, dans un premier temps, à analyser la nature de l'information que l'on voulait traiter. Cette étude pleine d'enseignement, nous a amenés au fur et à mesure à nous intéresser aux expériences et aux outils utilisés dans la compilation des langages de programmation. Nous avons donc étudié un ensemble d'outils, tel que les grammaires (LL/1, LR(1), affixe, à attributs) [Der 71] [Kos 71] [BaG 81] [Sim 81].

Parallèlement à cette étude, nous développions de plus en plus l'idée de créer un système plus général qui aurait pour but de manipuler des programmes. Le besoin d'un tel système était lié à deux constats :

premièrement, la notion d'environnement de programmation se développait de manière importante au travers de projets conséquents [Ade 82] [Bux 80] [DHK 80] [Loy 81] [TeR 81]. Ces environnements de programmation cherchent à fournir aux programmeurs :

un contexte lui assurant une plus grande aisance pour son travail d'analyse et de programmation.

Un ensemble d'outils spécialisés leurs offrant les fonctions nécessaires à la manipulation des programmes (éditeurs, fonctions d'archivage, fonctions de documentation, ...).

Deuxièmement, dans une approche didactique, il semblait intéressant de fournir un tel environnement de travail aux étudiants, et ceci pour deux raisons essentielles :

intégrer, dans une activité de travaux pratiques, tous les aspects de la programmation algorithmique en harmonie avec les méthodes et les techniques enseignées par ailleurs.

Présenter ces nouvelles fonctionnalités des systèmes, et offrir un support pour l'apprentissage et la maîtrise de ces nouveaux outils.

Un tel constat nous a amenés à poursuivre notre effort dans deux directions. D'une part à développer la notation algorithmique qui est le reflet des méthodes de programmation employées. D'autre part à élaborer un système pour la construction d'environnement de programmation.

Travail présenté

Première partie

Dans la première partie de cette thèse, nous présentons une notation algorithmique [Cas 79] [Dub 78] [Mor 79] [SGC 78]. Nous l'avons fondée sur l'existence de méthodes d'analyse et de techniques de programmation. La méthodologie de la programmation s'est beaucoup développée, de nombreux ouvrages en décrivent les aspects, l'intérêt et la situent par rapport à d'autres orientations telles que la preuve de programmes, la spécification formelle, la synthèse de programmes, ...

Un des intérêts de la méthodologie est qu'elle concerne tous les utilisateurs des systèmes informatiques, et ceci quel que soit le niveau de communication choisi, quel que soit le langage de programmation utilisé et quel que soit le domaine de l'application à traiter.

Le travail présenté dans cette première partie, s'insère donc dans ce cadre plus général qui a pour objectif de contribuer à une meilleure fiabilité des programmes par le recours systématique, tout au long du processus de conception et de construction de programmes, à :

des méthodes bien définies

des outils spécifiques, adaptés aux méthodes.

Ceci est réalisé par une étude se situant à trois niveaux :

Un niveau conceptuel à moyen ou long terme, dans lequel on cherche à définir des méthodes et à formaliser le processus de construction de programmes, sur la base d'une étude minutieuse d'algorithmes divers.

Un niveau plus concret à plus court terme, dans lequel on définit les caractéristiques d'un système interactif de logiciels d'aide à la production systématique de logiciels.

Un niveau pratique, dans lequel on s'attache à expérimenter et valider en grandeur nature, les résultats obtenus par ailleurs, sur des applications spécifiques [Ada 83].

Dans la présentation de la notation algorithmique on distingue, d'une part les concepts qui nous semblent à la base de l'activité de programmation, d'autre part les choix d'une notation à un instant donné, reflet aussi fidèle que possible de ces concepts.

Deuxième partie

Dans la deuxième partie de cette thèse, nous décrivons un système permettant de créer un environnement de travail autour de la notation algorithmique.

Un environnement de travail pour l'algorithmique et la programmation est essentiellement constitué de deux choses :

une notation qui est l'élément commun existant entre l'utilisateur et l'environnement. Elle est à la base de la communication homme machine.

Un ensemble d'outils mis à la diposition de l'utilisateur lui permettant de manipuler les programmes qu'il a écrits dans la notation algorithmique.

Ces outils, parce qu'ils manipulent le même type de donnée, sont en fait très proches les uns des autres ; de façon générale les méthodes et les techniques qui vont servir à leur élaboration seront semblables.

Nous avons cherché à dégager ce qu'il y avait de commun parmi les outils à réaliser (analyseur de la notation, traducteur de cette notation dans un langage cible du système informatique utilisé, transformateur d'algorithmes, ...), et à fournir un cadre adapté pour l'élaboration de ces divers outils de l'environnement de programmation.

Notre travail a porté, d'une part sur la manière de représenter un programme écrit par un utilisateur, d'autre part sur la façon de réaliser les différents outils de l'environnement de programmation.

Notre démarche, là encore, se basait sur l'emploi systématique de méthodes et de techniques répertoriées a priori. Une étude de diverses expériences dans le domaine nous a permis de progresser et de construire un modèle de description approprié aux problèmes évoqués précédemment.

Le système réalisé permet de décrire, à partir d'une notation algorithmique fixée, un modèle servant à manipuler les objets programmes.

Il permet :

de modéliser la représentation d'un objet "programme" ;

de définir des outils prenant en charge la représentation physique d'un objet "programme" et la réalisation des opérations élémentaires associées ;

de fournir à partir de ce formalisme des moyens d'interprétation (évaluation) de tout ou partie d'un objet "programme", en vue de la réalisation d'un outil de l'environnement de programmation.

Notre travail a été étayé, tout au long de cette étude, par des maquettes, des réalisations qui nous ont servi à préciser nos besoins et à vérifier l'utilité du système construit. Ce travail s'est concrétisé par des programmes réalisés en langage PL/1 sous le système Multics du CICG, et par un ensemble de réalisations partielles nous permettant de vérifier les possibilités expressives de notre système.

2 Notation algorithmique

2.1 Introduction

Le plan de cette partie est fondé sur un modèle simplifié du processus d'analyse descendante. Partant d'un problème P spécifié en terme d'une information I (et après une étape d'identification -5-)

1 DECOMPOSER: réduire le problème P et l'information I en sous-problèmes et sous-informations plus simples.

2 SPECIFIER : spécifier ces sous-problèmes et sous-informations de manière à pouvoir y faire référence précisement et correctement, sans en connaître la réalisation.

3 COMPOSER : composer les sous-problèmes et sous-informations de manière à décrire une solution du problème initial.

4 PROUVER : prouver que la composition réalisée est correcte vis-à-vis des règles de composition d'une part, des spécifications d'autre part. (cette étape de preuve rendue explicite ici, n'interdit pas d'accompagner chaque autre étape des preuves adéquates).

5 IDENTIFIER: décider de la réalisation des sous-problèmes et sous-informations en cherchant à identifier des problèmes connus. Dans la négative, regrouper sous-problèmes et sous-informations, et reprendre le processus pour chacun d'eux.

2.2 Abstraction

L'algorithmique s'appuie fortement sur la notion d'abstraction qui permet de ne retenir des choses ou des événements que les éléments nécessaires pour en parler. On remarque dès lors qu'il n'existe pas une seule abstraction, mais que celle-ci est dépendante du contexte où on l'utilise. Il existe plusieurs abstractions d'une même chose ou d'un même événement [GaM 78] [LiZ 74] [Par 72b] [Wir 71b].
Dans le cadre algorithmique, on distingue deux activités complémentaires où cette notion intervient : la lecture d'un algorithme et la construction d'un algorithme.

la lecture d'un algorithme met en jeu un premier mécanisme d'abstraction. Un algorithme décrit en effet un ensemble d'événements potentiels dans un schéma unique, et une exécution de l'algorithme ne retient que les points communs à tous ces événements et fait abstraction de ceux qui les différencient.

Lors de la construction d'un algorithme un second mécanisme d'abstraction est mis en jeu. Les techniques d'analyse descendante développées depuis quelques années concrétisent la nécessité d'une élaboration progressive de l'ensemble des composants d'un algorithme. Le processus de construction d'un algorithme n'est applicable et efficace que si l'on peut en nommer les composants, s'y référer avant de les avoir élaborés, et distinguer dans leur description les caractéristiques intrinsèques de ce qui dépend de décisions ultérieures lors de leur élaboration définitive. Ce processus de construction fait apparaître deux étapes lors de l'élaboration d'un composant : une première étape de spécification où le composant est repéré par un nom auquel on associe les caractéristiques spécifiques désirées ; une deuxième étape de représentation où l'on fixe pour ce composant les éléments restés non précisés.

La suite du chapitre est consacrée à la présentation de ces différents mécanismes d'abstraction, et des constructions linguistiques qu'ils nécessitent.

Nous considérons un algorithme comme une "composition" de deux types d'objets (composants) fondamentaux : les informations et les traitements de ces informations.
Nous introduisons ainsi les notions d'information nommée et d'action nommée. Le qualificatif "nommé" souligne dans notre terminologie la relation entre le concept envisagé et un mécanisme d'abstraction.

2.2.1 Information nommée

2.2.1.1 Concepts

Un algorithme décrit les traitements des informations caractérisant le problème étudié. Lors de la lecture d'un algorithme on fait abstraction des valeurs successives que prend une information au cours de son traitement. Pour comprendre un algorithme, on ne tient compte que des caractéristiques communes à ces valeurs. On fait abstraction des valeurs d'une information.

Pour désigner cette catégorie d'abstraction, nous définissons la notion d'information nommée.

Une information nommée comporte deux composants :

Le nom de l'information, support du mécanisme d'abstraction, qui permet de se référer à l'information sans avoir à parler de sa valeur.

La description algorithmique de l'information. Il s'agit de l'ensemble des caractéristiques que l'on retient comme pertinentes au niveau algorithmique (ce dont on ne veut pas faire abstraction). Ces caractéristiques définissent entièrement cette information, et permettent de l'utiliser correctement.

La description algorithmique comporte deux parties.

La **représentation** de l'information qui est décrite par la composition (à l'aide de constructeurs) d'informations plus élémentaires (@ 2.4).

La **spécification** de l'information qui est une caractérisation indépendante de la représentation, et définit le comportement de l'information durant son traitement, ainsi que sa nature logique.

L'état d'une information nommée à un instant donné est une spécification de celle-ci à cet instant. Du point de vue de l'abstraction qui est mise en oeuvre, on introduit une dimension supplémentaire, le temps, pour la caractériser. Le choix des instants où l'on s'intéresse à l'état d'une information est directement lié aux algorithmes décrivant les traitements de cette information.

Remarque :

Lors de l'élaboration d'un algorithme, on doit pouvoir se référer à une information sans encore la connaître précisément (éléments constitutifs, nature, domaine de valeur,...). La description algorithmique d'une information se fait donc progressivement tout au long des diverses étapes de l'analyse du problème. Ceci permet notamment de ne choisir les caractéristiques de représentation pour une information qu'au dernier moment. On se base alors sur une connaissance précise de la spécification de l'information.

Pour rendre compte de cette élaboration progressive, nous introduisons plus loin les notions de **type nommé** et d'**intervalle nommé** (@ 2.3.2, @ 2.3.3).

2.2.1.2 Notation

On introduit une information nommée dans un algorithme à l'aide d'une déclaration d'information nommée.

Cette construction permet d'ajouter un nom au répertoire des noms d'informations de l'algorithme et d'associer à ce nom les spécifications désirées.

La notation doit permettre de définir la représentation et la spécification de l'information de manière formelle ou informelle. On distingue le type de l'information qui est la spécification de la représentation au sens habituel de structure de données. La description du type est réalisée par composition de types plus élémentaires (standards ou élaborés) à l'aide de constructeurs appropriés (@ 2.4).

On associe à la déclaration d'une information nommée deux formes de commentaires. L'un précise les choix qui ont prévalu à l'élaboration de cette information nommée. Le second permet d'exprimer l'ensemble des autres caractéristiques de l'information. Le mode d'expression de ces caractéristiques dépend du "contexte culturel" des interlocuteurs (ce peut être en langage naturel, ou à l'aide de la logique du premier ordre,...).

Remarque :

Une information peut être un objet complexe, composé à partir d'autres informations (@2.4). Le traitement d'une information nommée se ramène à deux types de traitements. La consultation ou la modification de l'information, et ceci de façon totale ou partielle.

Bien que dans les deux cas le nom de l'information soit utilisé, c'est avec des sens différents. Dans un cas il s'agit de désigner sa valeur, on parle alors de consultation (évaluation), dans l'autre cas il s'agit de désigner l'accès à la valeur, on parle alors de modification (tabulation). La consultation de la valeur de l'information peut être considérée comme une fonction sans argument délivrant une valeur d'un type précis (@2.3.2). La modification de l'information revient à tabuler une nouvelle valeur pour la fonction de consultation.

2.2.2 Action nommée

2.2.2.1 Concepts

Un algorithme décrit les traitements de l'information qui caractérisent le problème étudié. Lors de la lecture d'un algorithme, on fait abstraction des divers traitements de l'information que décrit l'algorithme. Pour comprendre l'algorithme, on ne retient que les caractéristiques qui sont communes à ces traitements.

Pour désigner cette catégorie d'abstraction nous définissons la notion d'action nommée : Une action nommée permet de décrire la solution algorithmique d'un problème et de s'y référer tout en faisant abstraction du processus engendré lors de son exécution.

Une action nommée comporte deux composants :

Le **nom** de l'action qui permet de se référer à l'action tout en faisant abstraction de son exécution. (Comme pour l'information nommée, le nom est le support du mécanisme d'abstraction).

La **description algorithmique** de l'action : Il s'agit de l'ensemble des caractéristiques de l'action que l'on retient (ce dont on ne fait pas abstraction pour le niveau algorithmique que l'on considère). Ces caractéristiques définissent entièrement l'action et permettent son utilisation correcte.

La description algorithmique est composée de deux parties :

la **représentation** de l'action : c'est la solution algorithmique du problème étudié. On compose l'action en termes d'actions plus élémentaires à l'aide de **schémas de composition** (@ 2.4). La représentation de l'action peut dépendre d'un contexte particulier. Les notions d'**univers** et d'**interface** entre univers, que nous introduisons aux paragraphes 2.3.1 et 2.3.5, permettent de formaliser cette dépendance en exprimant les relations qui existent entre les diverses informations nommées et actions nommées.

La spécification de l'action : c'est une caractérisation indépendante de la représentation. De façon générale, une manière de spécifier une action consiste à caractériser son effet en fonction de l'ensemble des informations auxquelles l'action est liée dans l'intervalle de temps correspondant à son exécution. Deux instants sont plus particulièrement distingués, les instants initial et final, et ceci indépendemment du moment où se passe l'exécution.

La spécification porte alors sur l'état initial et l'état final de ces informations.

Le fait qu'une information nommée, dont dépend l'effet de l'action, ne soit pas modifiée par l'action est une propriété importante de l'action. Pour affiner la spécification de l'action, nous distinguons deux catégories d'effets : la création et la modification d'une information nommée.

Dans le cas d'une création, la spécification de l'action concerne l'information nommée créée.

Dans le cas d'une modification, la spécification précise cette modification par une comparaison entre l'état initial et final de l'information nommée modifiée.

Propriété :

Une modification peut être interprétée comme une forme particulière de création, l'état final de l'information nommée modifiée étant la spécification de l'information créée.

De même une création peut être interprétée comme une forme particulière de modification. L'information créée est utilisée dans le contexte où intervient l'action nommée pour modifier celui-ci. On peut dire alors que l'information nommée n'est modifiée qu'à l'instant final de l'action nommée, on fait de plus abstraction du nom de cette information lors de la modification.

Rôle dans l'analyse descendante :

Dans le principe général de l'analyse descendante, l'étude d'un problème algorithmique implique la spécification de sous-problèmes plus simples.

Dans une première phase on décompose le problème initial en sous-problèmes, dans une deuxième phase on élabore une solution au problème initial par composition des noms des actions associées aux sous-problèmes. Cette composition fait totalement abstraction de la représentation de ces actions en ne se référant qu'a leurs spécifications.

La résolution des sous-problèmes est alors fondée sur la connaissance précise de leurs spécifications.

2.2.2.2 Notation

On introduit une action nommée à l'aide d'une déclaration d'action nommée.

Cette construction permet de rajouter un nom au répertoire des noms d'actions et d'associer à celui-ci les caractéristiques nécessaires.

Une déclaration d'action nommée comporte deux parties :

Le nom et la spécification de l'action : Le nom permet de se référer à l'action sans avoir à connaître la façon dont elle est réalisée. La spécification est l'ensemble des caractéristiques dont on ne veut pas s'abstraire, et qui permettent d'utiliser correctement l'action nommée.

La **représentation** de l'action est le résultat d'une étape supplémentaire dans le processus de l'analyse descendante. Elle est décrite de façon algorithmique par une composition d'actions (élémentaires ou élaborées) à l'aide de schémas algorithmiques appropriés.

Spécification d'une action nommée

La spécification fournit l'ensemble des caractéristiques qu'il est nécessaire de connaître pour pouvoir utiliser l'action dans une composition. Ces caractéristiques sont décrites à l'aide de constructions linguistiques et de commentaires.
Nous distinguons dans la notation trois classes d'effet, c'est-à-dire trois comportements différents vis-à-vis de l'information manipulée. On associe un nom et des règles particulières à chacunes d'elles.

Les "fonctions" sont des actions dont l'effet est entièrement caractérisé par la création d'un ensemble d'informations. Par définition ce type d'action ne modifie pas d'informations du contexte où elles sont définies. Les informations créées ne sont pas nommées puisque le nom intervient dans le contexte où l'action est utilisée.
La spécification de la fonction comporte la spécification de l'information créée qui est une suite non vide de descriptions des types des informations créées. Les éléments de cette liste n'étant que des descriptions (aucun nom n'y est associé) elle doit être ordonnée.
On associe deux types de commentaires à la spécification d'une fonction. Le premier permet de décrire la nature logique des informations créées, ainsi que leurs propriétés si nécessaires. Le deuxième indique les informations dont dépend l'effet de la création, et décrit la nature de cette dépendance.

Les "modifications" sont des actions dont l'effet est décrit par la modification de leurs contextes de représentation (ensemble d'informations nommées). La spécification de la modification est caractérisée par l'état initial et final. Comme pour les fonctions, on associe à la spécification de la modification des commentaires sur sa nature et son rôle logique, ainsi que sur les informations nommées dont dépend l'effet de la modification.

Les "actions", nous conservons le terme action pour désigner ce cas dont l'effet est caractérisé à la fois par la création d'informations et la modification d'informations nommées du contexte de représentation.

L'action réagit pour sa partie création d'information de la même manière qu'une fonction. L'action modifiant aussi son contexte de définition, on trouve au niveau des commentaires des précisions sur les informations modifiées et leurs rôles. On mentionne, entre autres choses, leur état au début et à la fin logique de l'action (état initial et final).

Exemple

```
modification échange_AB
        {Soit A et B deux informations entières, l'action
         "échange" échange les valeurs de A et B}
        {état initial A=a, B=b}
        {état final   A=b, B=a}


fonction A_au_carré ->entier
        {Soit A une information entière, la fonction
         "A_au_carré" produit la valeur entière de A
         à la puissance deux}
        {Si A possède une valeur trop grande par rapport
         à la capacité offerte, le résultat sera erroné}
```

Représentation d'une action nommée

La représentation de l'action nommée concrétise la solution apportée au sous-problème posé. L'analyse du sous-problème peut conduire à introduire des informations nommées liées à cette ananlyse. Ces informations nommées sont dites locales à l'action nommée. Leurs déclarations apparaissent dans la description de la représentation. Elles ne sont pas accessibles en dehors de l'action.

La notion de localité (liée au mécanisme d'imbrication des déclarations) est un moyen classique pour structurer les algorithmes. Plus précisement, on peut ainsi exprimer une hiérarchie entre les objets intervenant dans la composition d'un algorithme. Nous ne permettons pas de définir des actions nommées locales à une autre action nommée (l'imbrication n'est pas autorisée). En effet, nous utilisons d'autres moyens ou constructions (univers, modules de définition) pour exprimer des relations entre actions nommées.

Les règles permettant de décrire les compositions seront données dans le paragraphe 2.4. On peut néanmoins remarquer ici qu'il est possible d'indiquer la fin "logique" de la représentation en opposition à la fin "physique". Cette construction permet, dans le cas d'une création d'informations (fonction, action) de décrire les valeurs des informations créées.

La fin physique ne peut être une fin logique pour les actions de création (fonction, action) puisque celles-ci doivent restituer une information. Par contre elle peut l'être pour les modifications.

On associe à la représentation d'une action nommée un commentaire dont la nature précise le principe de représentation choisie, et les choix qui ont conduit à cette représentation. Il s'agit, ici entre autres choses, d'énumérer les caractéristiques qui rendent cette solution particulière.

La fin "physique" d'une action nommée est repérée à l'aide d'un terminateur composé d'un terme rappelant la classe d'action nommée dont il s'agit, ainsi que le nom de l'action.

Règles d'utilisation des actions nommées :

La distinction entre les trois classes d'actions nommées entraine une règle précise concernant les actions nommées qui peuvent participer à la représentation d'une autre action nommée.

Une fonction ne peut pas, par définition, modifier d'informations de son contexte de représentation (autre que ses propres informations locales).

Par ailleurs, il n'y a pas d'imbrication d'actions nommées. Par conséquent, la représentation d'une fonction ne peut pas faire intervenir des modifications ou des actions.

2.3 Spécification

Les informations et les actions nommées fixent le niveau d'abstraction auquel est placé un algorithme (abstraction des valeurs des informations, et des événements engendrés par les actions).

L'élaboration progressive de ces composants nécessite de séparer la spécification de la représentation (dans le temps et au niveau des personnes intervenant). La notation doit favoriser cette démarche en fournissant des mécanismes appropriés [Abr 81] [Par 72a] [Wir 71b].

Nous complétons, dans ce paragraphe, l'examen des notions liées à la spécification des informations et des actions nommées. Nous présentons les notations correspondantes en gardant toujours le double point de vue qui caractérise la notation algorithmique.

Le point de vue de la lecture d'un algorithme. Quelles sont les possibilités d'expression de structure et d'architecture ?

Le point de vue de la construction d'un algortihme. Quelles sont les possibilités d'élaboration progressive ?

Nous présentons ici les trois notions d'univers, de type nommé et d'intervalle nommé.

La notion d'univers est centrée sur l'existence de relations entre les différentes composantes d'un algorithme. D'une part, elle fournit le moyen de spécifier ces relations. D'autre part, elle permet d'indiquer l'analyse simultanée d'un ensemble de composants.
On définit ainsi un outil de spécification du comportement d'un ensemble d'informations nommées par le jeu de la spécification d'un ensemble d'actions nommées.

La notion de type nommé est liée à la description algorithmique des informations nommées. Elle contribue à séparer de façon précise la définition d'une information de sa représentation, cette séparation devant être exprimée dans l'algorithme.

On peut ainsi repérer une unicité dans la représentation de plusieurs informations nommées avant de la choisir. Il devient alors possible d'exprimer au sein de l'algorithme que l'on construit, la relation qui existe entre plusieurs informations nommées.

La notion de type nommé joue par conséquent un rôle important dans l'expression de relations entre univers. Comme pour la spécification d'informations nommées, on peut associer un univers à un type nommé pour le spécifier par un ensemble d'actions nommées.

La notion d'**intervalle nommé** est une manière d'aborder les problèmes spécifiques à la gestion d'un ensemble d'informations. Elle permet de spécifier la relation existant entre une "table" et les index d'accès à cette table.

Remarque :

Nous entendons par comportement, l'ensemble des règles qui régissent les différents états de l'objet auquel on s'attache. Pour une information, il s'agit de définir les valeurs successives qu'elle peut prendre.

2.3.1 Univers

2.3.1.1 Concepts

Dans le paragraphe 2.2, nous avons montré l'étroite relation qui existe entre les informations et les actions nommées. L'existence d'une information nommée dans l'algorithme est liée à un ensemble précis de traitements :

les modifications de cette information nommée

les créations ou modifications d'informations qui en sont dépendantes (c'est-à-dire les traitements faisant intervenir une consultation de cette information nommée).

Certains de ces traitements influent sur la description de l'information nommée, notamment dans le choix de ses caractéristiques et de sa représentation. Inversement l'existence d'une action nommée dans un algorithme est liée à un ensemble précis d'informations nommées:

les informations nommées éventuellement modifiées par l'action

les informations nommées dont dépend l'effet de l'action.

Ces informations nommées interviennent dans la description de l'action nommée, tant dans sa spécification que dans sa représentation.

La notion d'univers est liée au besoin d'expression de relations existant entre informations et actions nommées. Plus précisément, nous avons décidé d'orienter ces relations, en considérant l'ensemble des actions de traitement d'une information (ou d'un ensemble d'informations liées logiquement) comme une spécification de cette information. La connaissance de ces actions est strictement suffisante pour utiliser l'information dans la description algorithmique [Ada 79b] [CGL 75] [CKL 76] [GaM 78] [GMS 77] [Loy 81] .

Un univers est consistué de deux composants :

Les "**accès à l'univers**". C'est l'ensemble des noms des actions nommées qui permettent de se référer aux informations qui se trouvent regroupées dans l'univers. Chacun des noms est un accès à l'univers. Nous introduisons plus loin la notion de module de définition (@2.3.5) qui permet de regrouper un ensemble d'accès à un univers pour en donner une "vision" particulière (protocole d'accès).

La **description algorithmique** de l'univers. Elle regroupe des informations et des actions nommées interdépendantes comme nous l'avons expliqué précédemment. Nous parlons plus précisément des informations caractéristiques et des actions primitives de l'univers. Elle comporte deux parties qui sont la représentation et la spécification de l'univers.

Dans la représentation de l'univers, on trouve la description des informations caractéristiques et des primitives de l'univers. On peut en outre trouver des informations et des actions nommées locales à l'univers qui ne sont introduites que pour représenter les primitives de l'univers.

La spécification de l'univers en est une caractérisation indépendante de sa représentation. C'est une manière de spécifier l'ensemble des informations caractéristiques de l'univers. Elle est composée de l'ensemble des spécifications de ses primitives. Les modules de définition que nous décrivons dans le paragraphe 2.3.5 sont un moyen complémentaire pour spécifier un univers.

La construction d'un univers se déroule en deux étapes consécutives.

La première étape consiste en un regroupement d'informations et d'actions interdépendantes. Il a lieu alors que les informations et les actions ne sont pas encore représentées. Ce regroupement est concrétisé par la définition des accès de l'univers et par la spécification de ses primitives. Il sert à l'élaboration de la spécification des informations nommées de l'univers.

Le sens d'un tel regroupement est d'organiser le travail de construction en mettant en évidence les composants d'un algorithme qui doivent être étudiés ensemble et ceux qui, inversement, peuvent être étudiés indépendamment.

Remarque :

> Du point de vue du processus de construction de l'algorithme, on est très intéressé par la non dépendance qui assure une désynchronisation des représentations.

Ainsi la construction d'un algorithme se ramène à la construction successive d'univers (ceci réapparaît dans la forme de l'algorithme final).

L'étape suivante est une étape de représentation. Elle suit la première, mais peut être totalement dissociée grâce au mécanisme de nom permettant de se référer à l'univers, indépendamment de sa représentation. On a ainsi toute liberté pour décider du moment où l'on va faire cette représentation, ainsi que de la personne pour le faire.

On peut, par ailleurs, changer la représentation sans que cela n'intervienne sur les autres composants de l'algorithme qui est élaboré. Ainsi le mécanisme d'univers apparaît comme une protection des informations et des actions qu'il regroupe vis-à-vis des autres composants.

2.3.1.2 notation

On introduit un univers à l'aide d'une déclaration d'univers. Cette déclaration peut être accompagnée de modules de définition (@ 2.3.5) de l'univers qui indiquent diverses spécifications complémentaires de l'univers.
On associe trois types de commentaires à une déclaration d'univers. Le premier indique la nature logique du regroupement d'informations nommées caractérisant l'univers et rappelle les accès à l'univers en donnant les spécifications des primitives. Le second commentaire résume les relations de l'univers avec les autres composants en énumérant les accès des univers utilisés.Enfin le troisième commentaire explique les choix qui ont conduit à cette représentation de l'univers.

Un accès de l'univers est défini en associant à la déclaration concernée l'attribut "accès". On distingue deux catégories d'accès :

l'accès est un nom d'action : dans ce cas il s'agit d'une primitive de l'univers

l'accès est un nom d'information : dans ce cas on interprète le nom d'information comme un nom de fonction. La valeur créée est simplement la valeur de l'information nommée. Il s'agit donc d'une manière simple de définir une primitive de consultation de la valeur de l'information nommée de l'univers. Remarquons qu'il ne peut s'agir que d'une consultation globale de l'information. Dans le cas contraire, cela voudrait dire que l'utilisateur, qui n'a pas accès au contenu de l'univers par définition, connaît la représentation de l'information.

La représentation d'un univers est faite en regroupant les déclarations des informations caractéristiques, et des primitives. Par ailleurs, on y trouve les informations et actions propres à l'univers, ainsi que les déclarations de types nommés ou d'intervalles nommés introduits pour l'élaboration de l'univers.

Remarque :

La portée des noms d'un algorithme est définie par rapport aux univers. Nous appelons "contexte de déclaration" d'une action nommée, l'ensemble des déclarations de l'univers auquel appartient l'action nommée. Les seuls noms utilisables dans cette action nommée sont ceux de son contexte de déclaration, les accès des autres univers, et les noms appartenant à son contexte local.

Exemple

```
univers machine_à_café
    {cet univers décrit le fonctionnement de ma cafetière}

    {informations décrivant les composants de la machine}
    Réservoir ....
    Filtre ....
    Horloge ....
    Heure_mise_en_marche ....
    .
    .
    .

    {actions primitives de la machine}
    action Remplir_Réservoir
          {remplit le Réservoir d'eau}

    action Remplir_Filtre
          {remplit le Filtre avec une potion}

    action Programmer_démarrage
          {mémorisation de l'heure de mise en route}

    action Immédiat
          {mise en route immédiate}
```

notation algorithmique

```
fonction heure -> date
        {délivre l'heure officielle}

fonction' heure_mise_en_marche -> date
        {délivre l'heure de mise en marche}

funivers machine_à_café
```

Réalisation de l'action "Programmer_démarrage" :

On dispose d'une fonction logique "poursuivre", dont la valeur caractérise le choix de l'utilisateur (touche enfoncée ou non).

```
action Programmer_démarrage
    {mémorisation de l'heure de mise en marche}

    h entier {représente l'heure c [0,23]}
    m entier {représente les minutes c [0,59]}

    tantque poursuivre faire
            h <- (h+1) modulo 24
    ftantque
    tantque poursuivre faire
            m <- (m+1) modulo 60
    ftantque

faction programmer_démarrage
```

2.3.2 Type nommé

2.3.2.1 Concepts

L'interdépendance entre informations nommées et actions nommées requiert des moyens d'expression spécifiques. Les univers sont déjà un moyen mis en place pour permettre de regrouper des composants. Mais il faut de plus des moyens pour pouvoir exprimer des relations entre informations et actions au sein même de l'univers, ou celles qui existent entre composants d'univers différents. La manière de spécifier les actions nommées implique que l'expression de ces relations est liée à la description des informations [Ada 79b] [Wir 71a].

La description algorithmique d'une information nommée comporte sa représentation et sa spécification. Cette dernière reste très informelle, si bien que les relations entre composants n'apparaissent qu'implicitement au niveau des représentations.
Dans certain cas, ceci entraine des ambiguïtés. Ainsi, lorsque deux informations nommées ont une même représentation, on peut s'interroger sur la signification de cette identité. Exprime-t-elle un lien logique entre les deux informations ? ou au contraire, n'est elle pas issue d'une faiblesse d'expression de la notation conduisant à une description identique de composants intrinsèquement différents ?

Dans l'exemple de la machine à café traité dans le paragraphe 2.3.1.2, on a deux informations (h et m) servant à la représentation de l'action "programmer_démarrage". Toutes les deux sont décrites comme étant des informations entières. Leur nature respective sont pourtant très différentes, la notation doit permettre de faire apparaître cette différence. A l'opposé, les informations "Horloge" et "Heure_mise_en_marche" sont de nature semblable. La notation devra permettre de concrétiser la propriété qui lie ces deux informations.

Nous voulons donc exprimer les relations entre composants indépendamment de leurs représentations. Nous introduisons pour cela la notion de type nommé.

Un type nommé permet de faire abstraction d'une représentation dans la description d'une information nommée. Il se compose de deux parties :

Le nom du type, support de l'abstraction, est utilisé dans la description des informations nommées comme spécification de leur représentation. La description des informations ne comporte plus la représentation (qui apparaît maintenant dans la description du type). La description devient ainsi indépendante d'une représentation particulière.
Le nom du type permet de ce fait d'exprimer explicitement une relation existante entre toutes les informations nommées auxquelles il est associé.

La description du type, comme la description d'une information nommée, comporte une spécification et une représentation. La spécification reste informelle et caractérise les informations associées à ce type. La représentation est une composition, à l'aide de constructeurs appropriés (@2.4), de types plus élémentaires.

La construction d'un nouveau type nommé se fait en deux étapes :

on choisit dans un premier temps le nom du type, auquel on associe une spécification. Généralement le nom d'un type est introduit à l'occasion de la définition d'une information nommée, et aide à une élaboration progressive de celle-ci.
On peut, dès lors, utiliser ce nom de type pour spécifier actions ou informations nommées, et exprimer les relations entre divers composants.
Pour cela les types nommés jouent un rôle important dans la définition de l'architecture de l'algorithme construit et donc dans l'organisation du travail de construction.

L'élaboration d'un type nommé peut être repoussée au moment le plus adéquat, notamment de manière à décider de la représentation en ayant connaissance de toutes les informations nommées et actions nommées nécessaires qui sont spécifiées en terme de ce type.

2.3.2.2 Notation

On introduit un type nommé dans un algorithme à l'aide d'une déclaration de type nommé. La spécification du type apparaît à l'aide d'un commentaire qui indique le rôle logique du type nommé, et les caractéristiques des informations qu'il permet de manipuler.

La représentation est une composition de types "standard" ou d'autres types nommés, à l'aide de constructeurs que nous donnons au paragraphe 2.4. Les choix impliqués par cette représentation sont contenus dans un commentaire.

Remarque :

Les types standard sont des types nommés pré-définis.
Dans le cadre de cette étude, nous avons décidé que les types ne seraient pas récursifs (@2.4.5.2). Par contre nous verrons plus tard que les types nommés peuvent être paramétrés, et nous compléterons alors leur description.

Le nom d'un type nommé est unique dans son contexte de déclaration, et il n'y a pas d'équivalence implicite entre types nommés. Cette règle peut être toutefois assouplie pour les cas qui seraient fréquents. Dans les autres cas la conversion de type devra être exprimée explicitement à l'aide d'un opérateur de conversion.

Exemple

```
type heure est entier {appartenant a l'intervalle 0,23}
type minute est entier {appartenant à l'intervalle 0,59}
```

2.3.3 Intervalle nommé

2.3.3.1 Concepts

Le concept de "table" est la solution algorithmique générale au problème de gestion d'un ensemble d'informations de même nature. Une table permet de mémoriser un tel ensemble, de le mettre à jour, d'y rechercher une information particulière. Toutes ces manipulations sont fondées sur la manière de repérer (caractériser) une information, au sein de l'ensemble. La notion d'"indicatif" répond à cette question, si bien qu'une table est toujours un ensemble de couples "indicatif,valeur". L'indicatif est ce qui permet de lier l'information à l'ensemble. La valeur est l'information propement dite. Il se peut très bien que l'indicatif fasse partie intégrante de la valeur.

La description algorithmique de la gestion d'une table fait donc toujours intervenir deux sortes d'informations nommées. D'une part celles qui formalisent la table, d'autre part celles qui servent à désigner les éléments de la table par leur indicatif. Ces dernières sont généralement appelées des **indices**.

La notion d'intervalle est introduite pour permettre de spécifier la nature des indicatifs d'une table et exprimer la relation existant entre une table et les indices qui interviennent dans sa gestion.

Un intervalle nommé se compose de deux parties :

Le nom de l'intervalle : support de l'abstraction que l'on fait pour désigner un élément dans un ensemble. Il formalise la nature de l'ensemble des indicatifs. Il sert à spécifier d'une part les informations nommées modélisant la table, d'autre part les indices utilisés pour gérer la table.
La présence d'un nom d'intervalle dans la spécification de diverses informations nommées explicite deux sortes de relations : tables définies sur le même ensemble d'indicatifs ; indices dédiés à la gestion d'une table.

La description de l'intervalle se compose d'une spécification et d'une représentation. On y précise notamment la nature des indicatifs, l'existence éventuelle d'une relation d'ordre sur ces indicatifs. Elle peut être complétée par un certain nombre d'actions nommées liées à l'ensemble des indicatifs (qui sont une spécification de cet ensemble d'indicatifs).

La construction d'un intervalle nommé est bien sûr liée à la mise en évidence d'une table et à l'analyse de cette table. On procède en deux étapes successives. On fait d'abord le choix d'un nom pour l'intervalle, et caractérise les indicatifs ; puis, dans une étape séparée, on décrit les indicatifs.
Ce processus permet de rendre la spécification de la table indépendante de la description des indicatifs, et il contribue à une meilleure expression de l'architecture de l'algorithme construit (relation exprimée par l'intermédiaire de noms).

Remarque :

Nous avons introduit les intervalles nommés dans le contexte de gestion de table. Cette notion sert notamment à caractériser les valeurs prises par certaines informations nommées comme les indices. De ce point de vue, les intervalles nommés sont une manière d'exprimer la spécification d'une information nommée. On peut donc recourir à ce moyen de spécification indépendamment de tout problème de gestion de table.

2.3.3.2 Notation

La notion de table est supportée par le constructeur "tableau" présenté plus loin dans le chapitre Composition (@2.4).

On introduit un intervalle nommé dans un algorithme à l'aide d'une déclaration d'intervalle nommé.On décrit donc un intervalle en énoncant les bornes inférieure et supérieure de l'intervalle fermé. La borne inférieure doit effectivement être inférieure ou égale à la borne supérieure.

Pour pouvoir s'abstraire au niveau algorithmique des valeurs réelles caractérisant l'intervalle, on introduit un ensemble d'opérateurs. Ces opérateurs (binf, bsup, card, suc, pred), qui sont des fonctions, forment une spécification de l'intervalle.

La déclaration d'un indice comporte le nom de l'indice et le nom de l'intervalle sur lequel il est défini. Toute modification de la valeur de l'indice est assujettie au contrôle de validité qui assure le respect de sa spécification.
Toute indexation d'un tableau nécessite une vérification de la compatibilité de l'indice et de l'intervalle, il en est de même pour la modification de la valeur d'un indice. On remarque que suivant les cas cette vérification peut être faite statiquement ou dynamiquement.

Exemple

Cet exemple décrit ce que pourrait être la base d'un échangeur de monnaie. Le type monnaie définit les valeurs que l'on manipule, l'intervalle décrit les possibilités de l'échangeur (les pièces de un franc à dix francs).

```
type monnaie est énumération(P1, P2, P5, P10, B20, B50, B100)

intervalle pièce est monnaie(P1,P10)

échangeur_monnaie tableau(pièce) de monnaie

pièce_courante indice(pièce)
```

2.3.4 Spécification de type nommé

2.3.4.1 Concepts

Décrire une information nommée consiste à choisir une représentation et à définir la manière de consulter et modifier les valeurs des informations élémentaires qui la composent.
La notation algorithmique doit nous donner des moyens de décrire une telle représentation. Nous abordons ici la question de la spécification de cette représentation.

La notion de type permet de faire abstraction d'un représentation lorsque l'on décrit une information. Le nom sert de lien entre la description de l'information et celle de la représentation.

De plus le type caractérise à la fois les informations et les actions manipulant ces informations. Il permet donc d'exprimer des relations à un niveau plus général, entre informations et actions nommées. Ainsi le type nommé est un moyen de décrire le comportement de l'information. Il participe de ce fait à la spécification de l'information au même titre que l'univers.

Spécifier un type nommé consiste à caractériser une représentation autrement que par sa description. Ceci contribue à une meilleure structure des algorithmes. On favorise l'élaboration séparée des traitements et des représentations. On facilite la compréhension et la modification des algorithmes, en faisant apparaître explicitement cette séparation.

Spécifier un type à l'aide d'un univers de types permet de décrire le comportement d'une "classe" d'informations élaborées avec ce type. Cela montre que l'univers de type reste une spécification d'une information nommée. On y fait par contre abstraction du nom et du contexte de définition de cette information.

2.3.4.2 Notation

Le mécanisme d'univers doit permettre de spécifier un type nommé de la manière suivante (nous n'introduisons donc pas de syntaxe particulière pour une telle description) :

on regroupe dans l'univers associé au type nommé, l'ensemble des actions nommées permettant :

soit de consulter individuellement les informations élémentaires qui composent le type nommé : ce sont des fonctions à un paramètre dont le type est celui spécifié par l'univers.

Soit de créer une information du type spécifié, à partir des informations élémentaires la composant. Il s'agit alors d'un constructeur d'une valeur du type concerné.

Toutes les actions ainsi associées au type spécifié sont donc des fonctions. L'ensemble de leurs spécifications constitue la spécification du type.

les caractéristiques particulières d'un univers de type apparaissent dans un commentaire où l'on mentionne sa logique.
Notons que la représentation d'un univers de type ne comporte aucune déclaration d'information nommée.

Exemple

Dans cet exemple, nous caractérisons le comportement d'une pile d'entiers. Pour cela nous définissons le type pile et les opérations associées. L'utilisateur disposant d'un tel univers, pourra spécifier des informations ayant le type pile, et appliquer les opérations sur ces informations.

```
univers pile_entier
    {définit les opérations associées à une pile d'entiers}

    type pile est ...

    fonction empiler(pile P,entier E) ->pile
          {range l'entier E au sommet de la pile P}

    fonction dépiler(pile P) ->pile
          {supprime l'entier au sommet de la pile P}

    fonction est_vide(pile P) ->logique
          {vrai si la pile P ne comporte aucun élément}

    fonction sommet(pile P) ->entier
          {renvoi la valeur entière du sommet de la pile P}

    .
    .
    .

funivers pile_entier
```

2.3.5 Module de définition

2.3.5.1 Concepts

La construction d'algorithmes est basée sur l'élaboration répartie d'algorithmes plus petits qui sont les solutions des sous-problèmes mis en évidence. La structuration finale de l'algorithme est le reflet de cette analyse, et en fait apparaître les éléments essentiels.

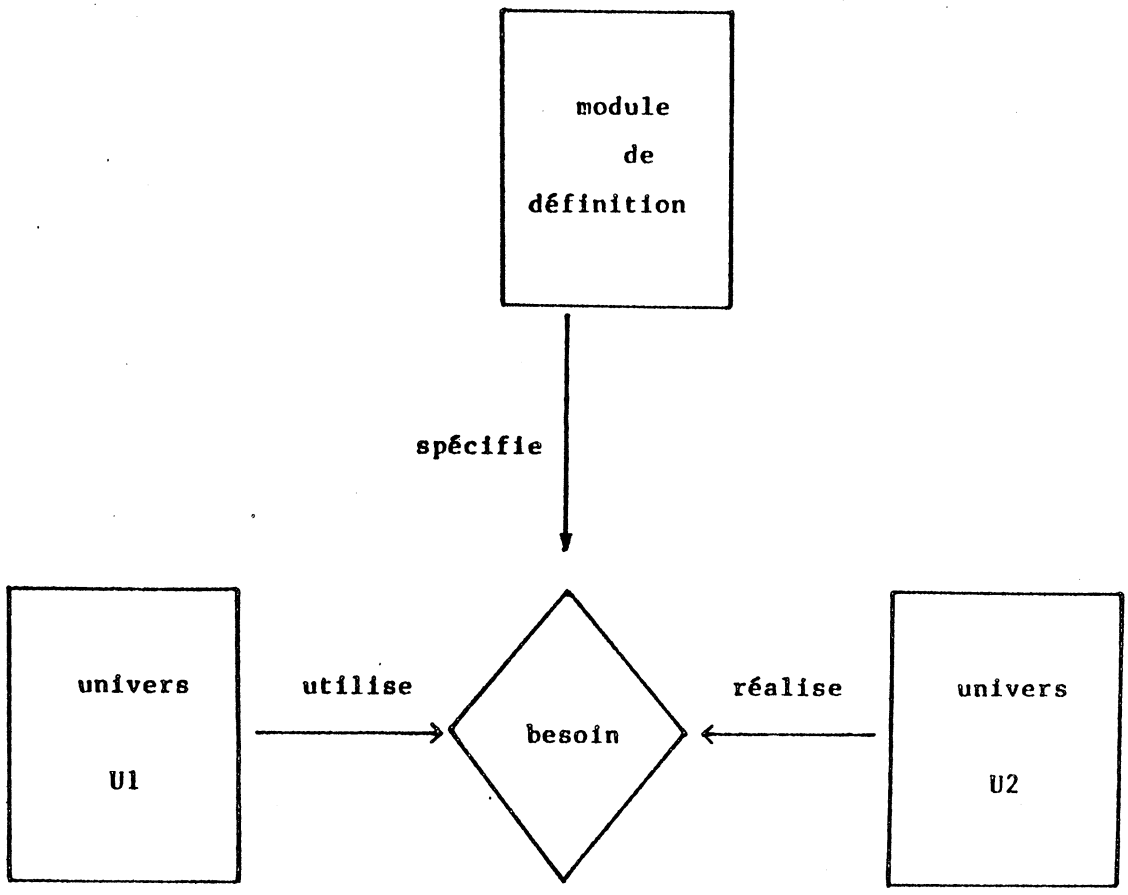
Un univers exprime, par le simple fait d'un regroupement (structuration), certaines relations existant entre des informations et des actions nommées.

Un univers est aussi le moyen de spécifier une information nommée ou un type nommé sous forme d'un ensemble de spécifications d'actions. L'univers permet ainsi d'indiquer qu'un groupe d'objets (actions et informations) ont des relations fortes, et doivent être considérés comme indissociables.

Un univers est utilisé par l'intermédiaire des ses noms d' actions primitives (ses accès) dans la représentation d'actions nommées et plus généralement dans la représentation d'autres univers. Du point de vue de la portée des noms, on dit que l'univers **exporte ses accès** et qu'un univers utilisant de tels accès **importe ces accès.**
Le partage de noms entre deux univers est l'expression de l'existence de relations précises entre ces univers. En effet l'élaboration du premier intervient dans l'analyse du second, et inversement la représentation du second est faite en terme des primitives du premier. En pratique, un univers est généralement utilisé par plusieurs univers ou spécifié au cours de l'analyse de plusieurs autres univers. Dans ce qui suit nous nous intéressons à la liaison entre un univers et ses univers utilisateurs.

Le module de définition est le moyen de regrouper les diverses spécifications utilisées dans la description d'un univers. Il permet d'établir un regroupement logique de ces spécifications qui conduira à l'élaboration d'un univers réalisant celles-ci.

Nous pouvons schématiser cette relation par le dessin suivant :



Le module de définition comporte :

le nom du module de définition qui permet de se référer aux traitements sans en connaître la représentation, ainsi que d'exprimer la hiérarchie existant entre un univers et ses univers utilisateurs.

la description du module de définition qui associe à ce nom une certaine spécification de l'univers considéré. Cette spécification reflète les besoins qu'ont un ou plusieurs autres univers.
Cette description rassemble notamment le nom de l'univers spécifié, les spécifications des primitives nécessaires aux univers utilisateurs.

Elaborer la spécification d'un univers se fait en élaborant un ensemble de modules de définition. La construction d'un module de définition concrétise une partie de l'élaboration de l'univers défini. On explicite dans ce module de définition la spécification de l'univers au vu des besoins exprimés lors de l'analyse d'univers utilisateurs.
On dissocie ainsi très clairement spécification et représentation d'un univers. On peut surtout repousser, au moment le plus opportun, le choix de représentation concernant cet univers. Cette facon de décrire la spécification d'un univers laisse la possibilité de définir plusieurs modules de définition pour un même univers avant d'envisager sa représentation. Chacun de ces modules de définition est une "vision" spécifique de l'univers considéré. Le problème de la représentation est alors de satisfaire aux contraintes exprimées par ces divers modules de définition. Il s'en suit évidemment que cet ensemble de modules de définition doit être cohérent.

Résumé :

La spécification d'un univers consiste donc en un ensemble de modules de définition. Chacun d'eux caractérise d'une certaine manière l'univers considéré.

Un module de définition exprime la hiérarchie entre l'univers spécifié et les univers utilisateurs. Il exhibe aussi la relation qui existe entre les univers utilisateurs d'un même univers selon une même caractérisation.

2.3.5.2 Notation

L'utilisation de modules de définition reste optionnelle, de manière à ne pas alourdir la description de petits programmes par un surcroît de structuration. Cet outil de spécification devient utile pour la description d'applications plus importantes, dans lesquelles on ne peut pas visualiser l'ensemble des composants de façon simple et claire. De toute manière un processeur de la notation peut restituer cette structuration à partir de l'analyse des algorithmes fournis.

Une déclaration de module de définition se compose du nom du module de définition, du nom de l'univers spécifié et d'un ensemble de spécifications décrivant les accès de l'univers utilisés. La fin de la déclaration est repérée par un terminateur particulier auquel est associé le nom du module de définition.

L'utilisation d'un univers dans un autre univers est exprimée par la présence d'une clause d'utilisation qui énumère l'ensemble des modules de définition utilisés. Cette clause apparaît en tête de l'univers utilisateur.

Chaque spécification apparaissant dans un module de définition concerne un accès à l'univers spécifié. La spécification de ces accès dépend de leurs natures.

Les déclarations de modules de définition apparaissent au même niveau que les déclarations de l'univers utilisateur. Les déclarations qui sont contenues dans le module de définition se situent donc aussi à ce même niveau. Tous les modules de définition d'un algorithme ont des noms différents.

Si jamais, par le truchement des modules de définition, il se trouve que plusieurs "objets algorithmiques" ont le même nom, alors le nom provenant d'un module de définition doit être préfixé par le nom de ce module de définition. Cette règle assure qu'il n'y a jamais d'ambiguïté lors de l'utilisation d'un nom dans un algorithme.

On associe à un module de définition deux commentaires, l'un indique le rôle logique de l'univers spécifié, l'autre énumère l'ensemble des noms d'univers où l'univers spécifié est utilisé sous cette définition. A chaque primitive contenue dans le module de défintion on associe un commentaire qui rappelle le rôle de celle-ci, ainsi que toute spécification informelle qui lui est associée dans sa déclaration.

Exemple

```
module de définition usage_distributeur

    type argent est ...
    type parfum est énumération(citron, thé, café, ...)
    type temp est énumération(chaud, froid)
    type boisson est ...

    fonction présent(parfum P) ->logique
          {vrai si on peut servir cette boisson}

    fonction commander(parfum P, temp T, argent A) ->boisson
          {produit la boisson demandée}

fmodule de définition usage_distributeur


module de définition maintenance_distributeur

    .
    .
    .

    fonction encaisser ->argent
          {délivre la recette}

    modification réapprovisionner
          {recharge la machine}

fmodule de définition maintenance_distributeur
```

2.4 Composition

L'analyse descendante est une succession d'étapes de réduction des problèmes considérés. Ce processus se termine toujours par l'identification de problèmes pour lesquels on dispose de solution (soit au niveau du langage, soit par l'utilisation de programmes d'une bibliothèque).

Il existe plusieurs manières de réduire la complexité d'un problème. La description de la solution du problème fait apparaître le mode de réduction employé au travers des schémas de compositions utilisés.

Nous voulons pouvoir rendre compte de ces choix dans l'algorithme obtenu. La notation algorithmique doit donc comporter des outils de composition qui assurent la souplesse de construction, et la bonne conduite de l'opération, ainsi que le respect des spécifications utilisées.

Dans la suite de ce paragraphe, nous faisons l'étude des différentes compositions qui servent dans la construction des actions et des informations. Nous présentons systématiquement leur double rôle :

d'une part ce sont des éléments qui concrétisent une suite de décisions prises tout au long du processus de construction du programme ;

d'autre part il s'agit de directives vis-à-vis de l'exécution soit du point de vue vérification, soit du point de vue implantation.

Les outils présentés concernent la dénotation et la construction de valeurs, la représentation des informations et le contrôle d'exécution.

Dans la partie production de valeur nous définissons le niveau minimal d'abstraction auquel nous acceptons de nous placer pour parler d'information.

Les quatre parties suivantes sont consacrées à quatre manières de composer les informations et les traitements. Nous associons à chaque fois information et traitement pour dégager, et mettre en valeur les similitudes qui existent entre ces deux éléments fondamentaux de l'algorithmique.

2.4.1 Production de valeurs

Une information nommée est élaborée par composition de sous-informations plus élémentaires. Le processus de décomposition d'une information s'arrête lorsque l'on identifie les sous-informations pour lesquelles il existe une construction linguistique prédéfinie. Ceci revient à fixer le niveau d'abstraction auquel est placé l'algorithme. Pour chaque élément de ce niveau on dispose d'une spécification standard. Cette spécification est décrite à l'aide d'un univers de type. Il se compose d'un ensemble de primitives qui décrivent le type, ces primitives sont des éléments de base de la notation.
Ainsi, la valeur d'une information nommée est décrite par un agrégat de ces valeurs élémentaires.

Les univers qui décrivent les abstractions de base sont implicitement prédéfinis dans la notation algorithmique.

2.4.1.1 Notation de valeurs

Types prédéfinis

On doit associer aux quatre types de bases (entier, réel, caractère, logique) une notation de valeurs en respectant si possible les habitudes usuelles. Pour les valeurs logiques on peut utiliser des noms symboliques tels que VRAI et FAUX.

Pour les valeurs caractères on a recours à un alphabet qui peut varier suivant les sites informatiques et les normes qui sont en vigueur.

Le problème de la notation de valeurs est lié à la valeur intrinsèque de ce que l'on veut désigner, et à l'étape de construction ou d'exécution du programme où l'on veut faire intervenir cette notation.

Types définis par l'utlisateur

Une valeur est soit de type standard soit composée à partir de valeurs de type standard. Pour décrire cette valeur on énumère l'ensemble des valeurs élémentaires mises en jeu (on discrétise la valeur de l'information nommée en un ensemble de valeurs élémentaires associées à chaque sous-informations). Du fait de cette discrétisation, il faut introduire une règle qui assure la cohérence entre l'ensemble des valeurs et l'ensemble des sous-informations. Une solution consiste à associer à chaque valeur élémentaire le nom de la sous-information à laquelle on l'a rattachée. Une autre solution consiste à ordonner l'ensemble des sous-informations et l'ensemble des valeurs, la correspondance se fait alors par position. Il existe d'autres formes plus élaborées comme la saisie "pleine page" avec "masque", menu qui permet d'énumérer l'ensemble du domaine de définition de la valeur....

Pour les valeurs associées aux types standard on dispose de primitives de conversion d'une représentation externe dans une représentation interne (conversion d'une chaine de caractères dans un type de base, menu,...).

2.4.1.2 Association "information nommée-valeur"

On effectue un transfert de valeur lorsque l'on modifie la valeur d'une information nommée. Cette opération couramment dénommée "affectation" comporte une information repérée par un nom,et une expression qui décrit la nouvelle valeur associée à l'information.

L'affectation peut être assimilée à la spécification d'une action de "modification", dont le contexte est réduit à l'information modifiée.

Une opération d'affectation n'est correcte que si les types spécifiant les deux intervenants de l'opération sont identiques. Néanmoins cette opération reste indépendante du type de ces opérandes : elle est générique.

Pour effectuer des transferts de valeurs de types différents mais compatibles, il faut faire apparaître explicitement la conversion nécessaire pour ce transfert. On note cette transformation à l'aide d'un opérateur (unaire) de conversion qui est appliqué à une information. Cette opération vérifie la compatibilité des types en s'assurant que leurs représentations sont compatibles. Dans les cas nécessaires et lorsque c'est possible une conversion de la valeur doit être effectuée.

L'acquisition (lecture) de valeurs se fait suivant le protocole de notation de valeurs, la restitution (écriture) de valeurs est l'opération inverse, elle est spécifiée avec les mêmes éléments que nous avons décrits dans le paragraphe précédent.

2.4.2 Discrétisation

2.4.2.1 Concepts

Dans la notion de processus, on exprime le fait que des événements se succèdent dans le temps. Un algorithme est une modélisation d'un certain comportement dans le temps. Un des schémas de composition le plus simple est celui qui permet de décrire cette succession d'événements. On nomme cette composition "discrétisation" ("séquence") car elle correspond au fait que l'on découpe le temps en une suite d'événements discrets (la réduction est basée sur l'introduction d'états intermédiaires).

Dans cette composition on introduit implicitement un ordre entre les événements. Parfois cet ordre est nécessaire, mais dans d'autres circonstances l'analyse du problème ne fait pas apparaître cette nécessité. Nous aurons donc deux manières de décrire la discrétisation selon que l'on veut exprimer un ordre ou pas.

2.4.2.2 Notation

Formalisme appliqué aux informations

Appliquée à une information, la discrétisation consiste à mettre en évidence un ensemble fini d'informations composites.

Une information est décrite de façon statique, la discrétisation permet de décrire des informations comme étant composées d'un agrégat d'informations plus élémentaires. Ces sous-informations ne sont pas, a priori, de même nature, et ne sont donc pas comparables. Pour ces raisons il nous semble que discrétiser une information se fait sans tenir compte d'un ordre particulier.

Pour désigner une sous-information élaborée par discrétisation, du fait qu'il n'y a pas d'ordre, on doit assigner à chaque sous-information d'une discrétisation un nom unique. La désignation se fait alors en donnant les noms de l'information et de la sous-information considérée.

Le processus de l'analyse descendante permet de décrire à nouveau des sous-informations comme étant elles-mêmes des informations composées. Cette composition peut être une discrétisation, on peut ainsi construire des informations de façon non limitative. La désignation se fait alors en utilisant tous les niveaux de noms depuis le nom de l'information générale jusqu'à celui de la sous-information désirée.

Exemple

```
type personne est structure(Identité nom,
                            Date_naissance date,
                            Résidence adresse)

type date est structure(Jour, Mois, Année entier)

Sir personne
...<- Sir.Identité
...<- Sir.Date_naissance.Jour
```

Formalisme appliqué aux actions

La discrétisation appliquée aux actions peut être séparée en trois compositions distinctes.

La composition séquentielle :

Si l'état initial d'un traitement élémentaire dépend des états finaux d'autres traitements élémentaires, il doit être composé en séquence par rapport à ces traitements.
Les traitements d'une composition séquentielle sont exécutés les uns après les autres dans l'ordre précisé par la composition.
Dans le cas de primitives standard une séquence d'appels peut être remplacée par un seul appel avec la liste ordonnée des paramètres des appels.

La composition collatérale :

Si pour tout traitement élémentaire d'un ensemble de traitements élémentaires, l'état initial de celui-ci est indépendant des états finaux des autres traitements élémentaires, ils peuvent être composés collatéralement (dans un ordre quelconque). Les expressions sont des exemples de composition collatérale : Le sens de l'expression ne dépend pas de l'ordre d'évaluation des différents termes de l'expression.

Les traitements d'une composition collatérale sont exécutés dans un ordre quelconque. Pour qu'une composition collatérale soit correcte, il est nécessaire que les informations nommées modifiées ne le soient que par un des traitements élémentaires, et n'interviennent pas dans la description de l'état initial des autres traitements.

La composition simultanée :

Si un ensemble de traitements élémentaires ont une partie de leurs états initiaux en commun, et si ces traitements élémentaires modifient cet état, ces traitements doivent être organisés à l'aide d'une composition simultanée.

Dans notre notation les compositions simultanées se réduisent pour l'instant à des affectations. On utilise alors des n-uplets [Mor 79] de chaque côté du symbole de l'affectation. Pour qu'une affectation de n-uplet soit correcte, il faut qu'une information nommée modifiée par cette affectation globale ne le soit que par une des affectations élémentaires.

L'affectation de n-uplet est utilisée pour transmettre les valeurs produites par une action ou une fonction délivrant un n-uplet de valeurs. La partie droite de l'affectation est alors réduite au nom de l'action nommée, et dans le cas d'une action la partie gauche ne peut contenir de noms indicés, les indices pouvant être modifiés par l'action.

2.4.3 Partition

2.4.3.1 Concepts

L'analyse par cas consiste à réduire un problème en un ensemble de sous-problèmes de la manière suivante. Quel que soit le sous-problème, il est une solution du problème initial, le choix d'un sous-problème dépend des valeurs de critères caractérisant le problème initial.

L'analyse par cas consiste donc d'une part à mettre en évidence les critères qui caractérisent le problème et à effectuer une partition sur cet ensemble de critères, d'autre part à élaborer les sous-problèmes associés aux différents cas de la partition.

Le choix d'un ordre dans l'élaboration des deux ensembles (critère, action) n'est pas imposé. Il dépend de la nature du problème et de la perception que l'utilisateur a lui même du problème qu'il veut résoudre. Il est parfois plus aisé de déduire l'un de l'élaboration de l'autre.
Dans la notation algorithmique on impose pour l'instant que la partition examine tous les cas possibles et qu'effectivement on réalise une partition sur cet ensemble de cas. Ce deuxième point implique le déterminisme des algorithmes produits.

2.4.3.2 Notation

Formalisme appliqué aux informations

La méthode appliquée aux informations consiste à définir l'ensemble des caractéristiques pertinentes d'une information pour le problème considéré, et à partitionner cet ensemble. Cette décomposition se base tout d'abord sur l'élaboration d'un ensemble de sous-informations, puis sur la détermination de l'existence ou non de relation entre les sous-informations élaborées. La prise en compte de telles relations au niveau de la spécification de l'information conduit à élaborer une information ayant plusieurs descriptions.

L'outil de composition associé est l'"union". L'ensemble des informations reliées par une union sont, a priori, de natures différentes bien qu'ayant de fortes relations logiques entre elles.

La partie critère associée à une partition est définie de façon statique en énoncant les valeurs pertinentes, d'un état, associées à chaque cas.

Pour plus de simplicité le domaine de valeur de l'état est réduit à un sous-domaine de valeur d'une information. Le nom de cette information est précisé pour toute l'union. A chaque cas est associé une valeur caractéristique de cette information. Toutes ces valeurs sont différentes deux à deux pour conserver le déterminisme de la construction.

Exemple

```
type volume est énumération(cube, sphère, cylindre)

type container est union forme volume dans(
        cube : long,haut,larg reel
        sphère : diam reel
        cylindre : base, grand reel)
```

Formalisme appliqué aux actions

L'analyse par cas appiquée à un traitement consiste à définir les informations pertinentes pour ce traitement, à définir l'état initial de ce traitement en terme des différents états des informations considérées, puis à appliquer sur cet ensemble d'état une partition. On associe à chaque partie élaborée un traitement élémentaire compatible avec l'état initial de départ.

Comme nous l'avons dit précédemment la partition peut se faire en premier lieu sur l'ensemble des traitements possibles, on associe alors à chaque traitement élaboré une condition appartenant au domaine de valeur de l'état initial.

L'outil de composition qui permet de décrire la partition est le "choix". Chaque "cas" du choix est représenté par un couple (condition, traitement). L'expression fait partie des critères et détermine les conditions dans lesquelles le traitement associé est solution du problème initial.

Pour que l'exécution de l'algorithme soit déterministe, une et une seule des expressions doit être vraie à un instant donné. L'exécution est erronée dans l'hypothèse où plusieurs des expressions sont vraies en même temps, ou si aucune des expressions n'est vraie.

On ne restreint pas, pour les traitements, le domaine d'expression des critères. Il en résulte que l'évaluation et les vérifications sont faites au moment de l'exécution.

Lorsque qu'une des expressions est le complément de toutes les autres par rapport à l'ensemble des critères initiaux, on peut utiliser une forme syntaxique réduite où l'expression du cas considéré est remplacée par le mot clé "sinon".

Si un choix se réduit à deux cas dont l'expression de l'un est la négation de l'autre on utilise la forme conditionnelle "si".

Exemple

```
{range dans MAX la valeur maximun de trois informations
entières A,B et C différentes deux à deux}
choix
    A>B et A>C : MAX <- A
    B>A et B>C : MAX <- B
    C>A et C>B : MAX <- C
fchoix
```

Dans cet exemple la partition est réalisée sur le domaine des actions. A chaque action on fait correspondre la condition qui lui est associée.

2.4.4 Itération

2.4.4.1 Concepts

Lorsque l'on décompose une information ou un traitement on obtient un ensemble de sous-informations ou de sous-problèmes. Dans les paragraphes précédents nous avons étudié des décompositions où les éléments constitutifs sont de natures diverses. Dans le formalisme de l'itération nous considérons les cas où la nature des sous-informations (resp. sous-problèmes) est la même.

La composition tient compte de la nature des éléments constitutifs par définition, mais aussi de l'ordre qui existe entre ces différents éléments (ce qui permet de les différencier).

2.4.4.1 Notation

Formalisme appliqué aux informations

La méthode appliquée à une information nommée consiste à la raffiner en un ensemble d'informations plus élémentaires de même nature.
Ce formalisme correspond au concept de "table" qui permet de mémoriser un ensemble d'informations identiques.
Pour pouvoir accéder à l'une de ces informations on dispose de la notion d'indicatif.

Pour décrire de telles compositions nous disposons de l'outil algorithmique "tableau". Il permet de spécifier la nature des éléments constitutifs ainsi que le domaine de valeur des indicatifs. Nous avons vu dans le chapitre "Spécification" que la notion d'intervalle nommé permet de spécifier la nature des indicatifs.

Un tableau doit comporter d'une part la spécification de la nature de ses éléments, d'autre part la spécification de la nature de ses indicatifs.

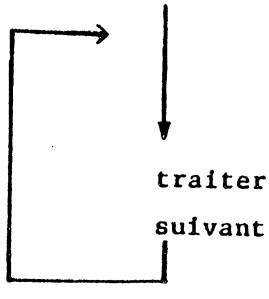
La désignation d'une information dans un ensemble d'informations représentées à l'aide d'un tableau, se fait grâce à l'opérateur d'indexation que l'on note "(nom d'indice)". L'indexation se fait à l'aide d'un indice défini sur le même intervalle que le tableau indicé. Cette règle permet d'assurer la correction de l'opération, au moins pour ce qui est du domaine de validité de l'indice.

On peut décrire le type des éléments constitutifs comme étant lui-même un tableau d'éléments, on obtient alors une matrice de ces éléments. L'opération d'indexation d'un de ces éléments se fait en associant deux indexations : la première s'applique à l'information générale, la deuxième s'applique à l'information indicée lors de la première indexation.

Formalisme appliqué aux actions

Dans le cas des actions nommées, on débouche sur diverses notations d'itération. La question de la construction d'itérations a été étudiée de nombreuses fois, notamment autour de la notion d'invariant d'itération [BeB 83]. Dans notre cas, nous nous sommes intéressés à la méthode du traitement séquentiel [Sch 79] : On interprète le problème en termes d'une file d'informations élémentaires de manière à pouvoir le résoudre sous la forme de l'application répétée d'une même action élémentaire à chaque information élémentaire dans l'ordre défini par la file.

Cette démarche fait apparaître d'une part une action élémentaire de traitement des éléments de la file (traiter), d'autre part une action élémentaire qui permet d'obtenir l'élément suivant, s'il existe, dans la file (suivant). Le processus de l'itération peut être représenté de la façon suivante :



Nous présentons trois schémas d'itération de files correspondant aux différentes propriétés caractéristiques des files.

Le schéma "itérer" :

Il s'applique à des files non vides (il existe au moins un élément à traiter). La fin est repérée losrque le dernier élément de la file a été traité.

Le schéma "repéter" :

Il s'applique à des files non vides. La fin est repérée lorsqu'on arrive sur le dernier élément de la file séquentielle. Ce dernier élément n'est donc pas traité, cet élément sert uniquement, vis-à-vis de l'itération, à marquer la fin (sentinelle).

Le schéma "tantque" :

Il s'applique à des files pouvant être vides (aucun élément n'est à traiter). La fin est repérée, de la même manière que dans le schéma "repéter", par une sentinelle.

Le corps de l'itération est séparé en deux parties : une partie correspond au traitement appliqué à chaque élément, l'autre partie correspond au passage à l'élément suivant dans la file. La détermination de la fin de la file est décrite dans une expression logique attachée à l'itération.

On associe à l'itération un commentaire qui décrit son invariant.

2.4.5 Récursion

2.4.5.1 Concepts

Pendant le processus de l'analyse descendante on réduit la complexité des informations et des traitements, en mettant en évidence un ensemble de sous-informations ou sous-problèmes. La solution du problème initial se trouve alors dans une composition de ces éléments élaborés.

L'analyse récurrente consiste à reconnaître parmi les sous-problèmes le problème initial (ceci avec un degré de liberté portant sur des éléments qui différencient les occurences entre elles).

2.4.5.2 Notation

Formalisme appliqué aux informations

Le formalisme de récurrence appliqué aux informations permet de décrire une information comme un ensemble, ordonné d'une certaine manière, d'informations élémentaires. Le type de l'information élémentaire doit être une union pour que la représentation ne soit pas infinie (la spécification d'une information nous permet de décrire sa représentation).

La création d'informations récursives se fait obligatoirement de manière dynamique et ceci en remontant vers l'information générale, puisqu'une information ne peut être élaborée que lorsque ses composants le sont eux-mêmes.

La désignation d'un élément de l'ensemble doit se faire à travers un accès récursif ou une forme développée de cet accès récursif.
Nous n'avons, pour l'instant, pas autorisé la description d'information sous un formalisme récurrent.

Les types, dans la notation algorithmique, ne peuvent donc être récursifs, et ceci de manière directe (utilisation du type lui-même dans sa description) ou indirecte (utilisation de récursivité croisée).

Formalisme appliqué aux actions

L'analyse récurrente appliquée aux actions permet de spécifier qu'une action nommée est composée à l'aide d'une action de même nature (dont les spécifications sont les mêmes). Associé à ce schéma de composition on retrouve systématiquement le schéma de l'analyse par cas.

L'effet d'une action nommée récursive est entièrement défini par l'information nommée qui décrit le domaine de critère de l'analyse par cas. En effet les constructions étant toujours déterministes, seule une modification des informations contrôlant le choix peut changer l'effet de l'action récursive.

2.5 Identification

Dans le modèle de l'analyse descendante la dernière phase du cycle correspond à une identification des sous-problèmes élaborés lors des phases précédentes. Ces sous-problèmes ayant fait l'objet d'une spécification, plusieurs cas dépendant de cette spécification peuvent se présenter.

Le sous-problème peut être résolu par un "atome" de la notation (on s'abstrait d'un problème considéré comme élémentaire).

C'est la première fois que l'on rencontre ce problème, il faut alors recommencer le processus de l'analyse descendante pour ce problème.

Un problème identique a déjà été traité, il suffit alors d'appliquer à nouveau la même démarche pour celui-ci.

Dans les deux premières solutions, le processus de construction se poursuit normalement avec les outils présentés lors des chapitres précédents. Dans le troisième cas, nous sommes confrontés à une nouvelle problématique pour laquelle nous allons introduire de nouveaux concepts et les outils associés. Nous avons à répondre aux deux questions suivantes :

Comment identifier deux problèmes ?
Comment décrire qu'une solution est commune à plusieurs problèmes ?

Nous avons déjà répondu à un type de problème similaire en introduisant les modules de définition. Ceux-ci permettent d'associer à plusieurs sous-problèmes une solution commune réalisée par un univers (on identifie une solution unique).

Dans la suite de ce chapitre nous étudions les moyens de généraliser la solution d'un problème afin d'obtenir un "modèle" devenu indépendant des caractéristiques propres du problème particulier de départ.

Nous nous intéressons à la façon de généraliser la spécification d'une information par la modélisation de type, par la généralisation d'univers pour obtenir un modèle de comportement d'une information.

Dans tous les cas que nous verrons dans ce chapitre, il s'agit d'une manière ou d'une autre, de paramétrer une construction algorithmique afin de la libérer d'une contrainte déterminée. La technique de paramétrisation est le support de la modélisation.

2.5.1 Paramétrisation

Dans le mécanisme de paramétrisation on libère l'information ou le traitement d'éléments qui ne sont pas déterminants dans son comportement. Le paramètre permet de généraliser l'information ou le traitement pour toute valeur correcte de l'élément qu'il remplace. On s'abstrait du contexte de définition des éléments considérés, ainsi que de la manière de les obtenir.

La paramétrisation constitue un des facteurs qui permettent d'unifier les différents "points de vue" d'un univers. Ces différents "points de vue" sont représentés par les modules de définition qui sont autant de spécifications de cet univers. L'"unification" se fait en paramétrisant les éléments qui ne sont pas semblables dans les modules de définition.

2.5.1.1 Formalisme appliqué aux informations

Une information nommée est l'abstraction d'un ensemble de valeurs. Elle est décrite par un nom auquel est associé une spécification précisant les éléments caractéristiques de cette information.

Les contraintes associées à une information sont les caractéristiques qui figent la spécification dans le cadre particulier de cette information. La paramétrisation se fera donc au niveau du type, moyen dans la notation algorithmique de décrire la spécification de l'information. De même dans le cadre d'un univers de spécification de type, cette spécification ne doit pas être tributaire d'éléments propres à une information précise.

Les seuls éléments qui, dans la description d'une information, sont spécifiques à cette information, sont les intervalles. Ils fixent la taille de l'information, et de ce fait restent propres à cette information.

Pour définir un type sans avoir à préciser des intervalles effectifs, nous utilisons des noms formels d'intervalles. La substitution se fait lors de la déclaration d'une information à l'aide de ce type, en donnant la liste des intervalles effectifs qui doivent être considérés pour cette information.

Dans la définition d'un type, au nom de ce type est attachée une liste de noms formels d'intervalles (ces noms restent locaux à la définition du type). Ces noms formels sont utilisés dans la description du type pour préciser les endroits où ils interviennent.

On peut à l'aide de ces noms formels, exhiber à l'intérieur même du type, les relations propres à ce type (comme la propriété qu'une matrice est carrée par le biais de l'utilisation du même nom formel pour désigner les deux dimensions,...).

Exemple

```
type matcar(dim) est tab(dim) de tab(dim) de ...
type esp_liste(dim) est tab(dim) de struct(I ...,S itv(dim))
```

Lors de la déclaration d'une information, on doit préciser la liste des intervalles effectifs à prendre en considération pour cette information particulière. L'association avec les noms formels se fait par position dans les deux listes (liste des intervalles formels associée à la déclaration du type, liste des intervalles effectifs associée à cette utilisation du type).

Exemple

```
mat_10_10 matcar(10)
itv taille_liste est 0,100
mémoire_liste esp_liste(taille_liste)
```

2.5.1.2 Formalisme appliqué aux actions

Dans le chapitre sur l'abstraction et la spécification (@2.2, @2.3) nous avons montré que l'action nommée est dépendante d'un certain contexte : dans le cadre des fonctions il détermine la valeur à produire ; dans le cadre des modifications il détermine la modification qui doit intervenir. Cet environnement est une contrainte pour cette action nommée, puisqu'elle en dépend.
Pour libérer l'action de cette contrainte on introduit la notion d'information nommée formelle.

L'information nommée formelle est l'abstraction d'une partie du contexte d'utilisation de cette action. Elle est locale à cette action nommée. Elle est définie par un nom et un type.

La définition d'une action nommée comporte une liste de paramètres. Lors de l'utilisation de cette action nommée, on précise la liste des informations nommées effectives qui doivent compléter le contexte d'utilisation de l'action. L'association entre paramètre effectif et paramètre formel se fait par position dans les deux listes.

La règle d'identité des types doit être respectée entre le paramètre effectif et le paramètre formel auquel il est substitué.

Lorsque le type du paramètre formel comporte des intervalles, ceux-ci s'interprètent des deux manières suivantes :

C'est une déclaration formelle d'intervalle. Celui-ci sera donc remplacé au moment de l'appel par l'intervalle effectif correspondant qui est associé au paramètre effectif. Toute utilisation dans le corps de l'action de cet intervalle est une utilisation de l'intervalle effectif. Il s'agit ici d'un passage implicite de l'intervalle en paramètre.

C'est un intervalle effectif (ceci est noté par une convention d'écriture particulière), on indique dans ce cas que les objets effectifs pouvant être associés aux paramètres doivent être munis d'une taille définie par l'action nommée.

Il faut dans ce cas que les deux intervalles effectifs mis en présence lors de l'appel de l'action nommée soient compatibles, c'est-à-dire, soit qu'ils désignent le même intervalle (il y a alors identité) soit qu'ils aient des descriptions compatibles.

2.5.2 Modélisation

2.5.2.1 Modélisation de la structure d'univers

L'application de méthodes de programmation sert à décrire le plus précisément possible les caractéristiques des objets que l'on est amené à manipuler. Cette description est faite de façon structurelle et comportementale à l'aide d'informations et d'actions nommées associées à ces informations.
On a vu précédemment qu'il était nécessaire de spécifier une information autrement que par sa description. Pour cela on a introduit la notion de spécification de type nommé. Cette spécification qui est décrite par un univers est en fait un ensemble de primitives associées à une information nommée. Dans cette manière de faire l'information nommée n'est pas décrite au niveau de la spécification. Cette séparation rend l'algorithme moins lisible, et nuit à la structuration de l'algorithme final.

La notion de type a été introduite pour pouvoir spécifier la nature des informations manipulées par un algorithme. Les opérations (primitives) que l'on peut appliquer à une information nommée caractérisent cette information. Tout au long des chapitres précédents nous avons montré l'importance de cette association.

On introduit alors l'univers comme une manière de spécifier l'information au même titre que le type nommé. Pour modéliser un type d'informations nommées, l'univers devient une description de type. Il s'en suit que l'on peut avoir plusieurs instances du même univers puisque celui-ci est devenu un modèle d'informations.

L'introduction de l'univers en tant que modèle d'informations intervient comme une généralisation de la notion de type défini dans les chapitres précédents. Elle unifie les notions d'action nommée qui manipule une information nommée et d'univers qui regroupe les actions nommées manipulant une information nommée.

2.5.2.2 Paramétrisation des Modèles

Dans un grand nombre de cas l'analyse de problèmes algorithmiques conduit à l'élaboration d'informations nommées qui sont des compositions d'un ensemble de sous-informations de même nature. On est amené à décrire le comportement de cet ensemble d'informations, mais cette description ne tient pas compte (ne doit pas tenir compte) de la nature de la sous-information.
La paramétrisation de l'univers est un moyen de modéliser une information en faisant abstraction des éléments de description qui ne sont pas pertinents (généricité). Cette façon de faire correspond très bien à la notion de niveau d'abstraction.

2.6 Programme et algorithme

Le processus de l'analyse descendante impose de décrire les solutions des problèmes analysés en termes d'une composition des solutions des sous-problèmes mis en évidence.

Pour que le processus de l'analyse se développe correctement, on doit organiser les éléments élaborés au cours d'une étape de l'analyse. Cette structuration se fait en fonction des dépendances logiques existant entre ces éléments. Elle sert aussi bien dans la construction d'une solution en permettant de ne considérer au même moment que les éléments essentiels (ayant des relations entre eux), que dans la lecture de la solution si cette structuration est conservée (la structuration rappelle les choix qui ont conduit à l'élaboration de cette solution).

Dans la notation algorithmique cette structuration est représentée par la notion d'univers. Un univers regroupe les informations et les traitements de ces informations.

La solution d'un problème est décrite par la composition d'un ensemble d'éléments appartenant à ces univers (ces éléments sont des actions nommées ou des informations nommées qui ont ici la signification de fonctions de consultation). On obtient ainsi un programme.

Un programme comporte deux composants :

> Le nom du programme qui est l'abstraction des moyens mis en oeuvre pour résoudre le problème. Il permet de se référer à la solution sans parler des éléments de cette représentation.
>
> La **description algorithmique** du programme. Il s'agit ici des éléments intervenant dans la solution du problème.

La description algorithmique est composée de deux parties :

> Le **répertoire**, ensemble des primitives du niveau d'abstraction du programme, est décrit par l'ensemble des univers qui le composent (soit en décrivant l'univers à cet endroit, soit en utilisant un module de définition associé à cet univers).
>
> L'**algorithme**, est une composition des primitives du répertoire. C'est la solution algorithmique du problème, il peut posséder un ensemble d'informations nommées locales.

Un univers donne accès à un ensemble de primitives. Un programme est la description d'un processus d'exécution utilisant les primitives des univers qui lui sont associés. Cette différence ne permet pas de considérer un univers comme un programme, elle admet cependant la transformation d'un programme en un univers. Cette transformation se fait de la manière suivante :

> On construit un univers ayant le même nom que le programme. Cet univers ne possède qu'une seule action primitive, dont le nom est le même que celui du programme (univers).

Le programme qu'elle représente ne dépend d'aucun contexte, cette action ne possède pas de paramètre. Les informations caractéristiques de ce nouvel univers sont les informations nommées locales au programme correspondant.

Si le programme converti possède des séquences d'initialisation, celles-ci peuvent être regroupées dans une primitive d'initialisation.

2.7 Conclusion

2.7.1 Influence de l'environnement sur la notation

Dans les paragraphes précédents nous avons introduit les différents outils qui nous semblent nécessaires à l'analyse méthodique d'un problème.

Nous avons décrit nos besoins en ne considérant que le coté analyse d'algorithme dans le processus de construction. C'est dans cette notation que l'on exprime les choix et la nature profonde de la solution proposée.
Cette étude ne fait pas référence à d'autres problèmes très concrets de la programmation (construction, modification,... des programmes). La manipulation des programmes est un problème qui nécessite des outils appropriés.

Il nous semble, par contre, préférable de ne pas mélanger les différents niveaux de problématique où l'utilisateur doit intervenir. Afin de minimiser les risques d'erreur, les différentes façons d'aborder un programme doivent rester très distinctes pour l'utilisateur.

Dans la suite nous allons décrire quels sont les outils et les propriétés qui nous semblent nécessaires pour constituer un contexte favorable à la construction de programme. Nous désignerons cet ensemble sous le terme d'**environnement de programmation.**

Le centre de l'environnement de programmation sera constitué par la notation (les notations, puisque nous allons définir plusieurs notations suivant le niveau où l'on veut intervenir).
Les outils de l'environnement de programmation seront des outils "intelligents", c'est-à-dire qu'ils traiteront les programmes en connaissant la structure de ceux-ci (à l'inverse d'éditeurs de texte qui ne connaissent pas la structure des programmes).

On voit qu'il va exister des liens importants entre les notations et les outils de l'environnement de programmation. L'interactivité existant entre une notation et l'environnement va nécessiter l'adaptation de cette notation aux outils de l'environnement, ceci afin d'avoir une bonne intégration des deux constituants.

Dans la suite nous proposons quelques remarques concernant les caractéristiques induites par l'environnement de programmation sur la notation algorithmique. En tout état de cause, il ne s'agit que d'adapter certaines constructions de la notation pour les rendre plus souples ou d'introduire des éléments de connection entre la notation et l'environnement.

2.7.2 Eléments de simplification

Nous considérons dans ce paragraphe les éléments qui rendent l'activité de programmation plus aisée pour l'utilisateur.

En décrivant la notation algorithmique nous nous sommes intéressés aux concepts qui nous semblent à la base de l'activité de programmation. Nous avons associé à ces concepts les outils qui nous paraissaient nécessaires, mais nous n'avons décrit ces outils que par leur sémantique.

Un des facteurs importants, qui facilite l'activité de programmation est la syntaxe concrète de la notation utilisée. Elle doit être simple et traduire le mieux possible les concepts associés.

Un deuxième facteur important de simplification est la stucturation même de ce que l'on manipule. Il est important de pouvoir décomposer physiquement le programme en modules en couches qui permettent un accès sélectif (par spécificité).

Le découpage en modules permet de créer de petites unités facilement manipulables.

On peut disposer pour un module particulier de couches distinctes assurant des rôles complémentaires.

Une couche **programme** qui contient l'algorithme

Une couche **assertion** qui peut contenir des éléments de preuve ou de vérification du programme en un certain point, la notation pour exprimer ce niveau peut être celle de la logique du premier ordre

Une couche **implémentation** qui permettrait de préciser les choix de codage des informations en mémoire

Une ou plusieurs couches **commentaire**.

2.7.3 Eléments d'intégration

La notion de simplification est relative à la manière dont l'utilisateur va pouvoir appréhender son programme.
La notion d'intégration est un point de vue associé à l'environnement de programmation. La question étant de déterminer comment les "objets" (programmes, unités, etc...) doivent être organisés pour que l'on puisse les manipuler de manière "intelligente".

Il s'agit ici de définir de quelle structure on doit disposer pour définir et réaliser des outils bien adaptés aux traitements des programmes.

Là encore on fait intervenir la notion de décomposition d'un programme en différentes unités ("modules") qui ont des spécifications précises et remplissent un rôle bien déterminé dans l'ensemble.
Par exemple la décomposition en deux modules différents de la partie "spécification" et de la partie "représentation" d'un univers.

Il faut adapter les modules de définition afin de pouvoir les manipuler et leur associer des évaluations, ou pour permettre de leur donner un rôle de langage de connexion.

3 Système pour la production d'environnement

3.1 Introduction

La conception d'un environnement de programmation repose sur la notion d'objet "programme" à partir duquel sont définies des fonctionnalités diverses.

Dans cet environnement deux éléments peuvent varier pour en donner des versions différentes.

D'une part le nombre et la nature des fonctionnalités de l'environnement : il est, à l'heure actuelle, difficile de définir précisément et a priori les fonctionnalités suffisantes. De plus leurs natures même n'est pas encore figée.

D'autre part la nature de la notation employée qui forme le noyau de l'environnement : elle peut aussi varier. Il arrive fréquemment au cours d'une phase de conception que la notation évolue (extension ou modification).

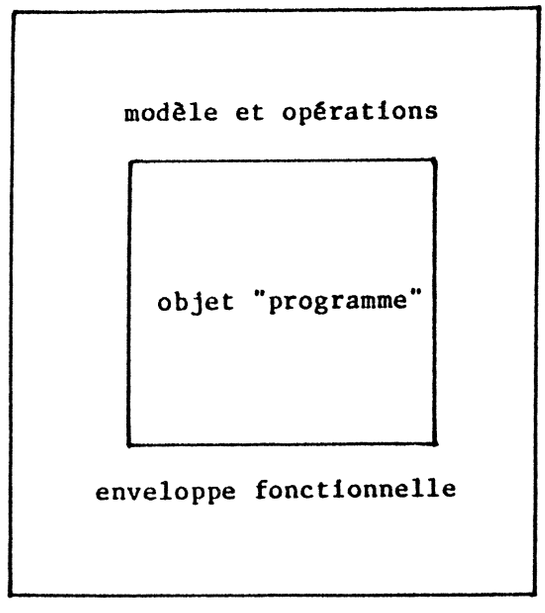
On a montré dans le première partie, qu'une notation est un moyen de construire des programmes qui sont d'autres choses que du texte. Il est maintenant acquis qu'un programme est un "objet" possédant une forte structuration. Nous pensons qu'il est bénéfique de disposer au niveau interne d'une représentation du programme différente de sa représentation externe.
Cette représentation interne peut elle aussi évoluer au cours du développement de l'environnement pour des raisons tout à fait spécifiques et d'ordre technique.

Dans notre cadre, nous considérons donc comme très important les variations qui pourraient intervenir pendant l'élaboration de l'environnement, tant au niveau qualitatif que quantitatif.

Cette possibilité de grande mutation de l'environnement nécessite de se doter d'outils spécifiques assurant un certain nombre de tâches sans lesquelles il n'est pas possible de développer correctement l'environnement.

La notation sert de point commun à tous les traitements que l'on peut vouloir effectuer. On peut d'ailleurs tous les considérer comme une évaluation particulière appliquée à un programme. Les programmes étant représentés par une structure arborescente, écrire un composant de l'environnement revient à écrire l'évaluation désirée associée au modèle d'arbre décrivant les programmes. La conception de la représentation interne joue donc un rôle essentiel pour la qualité de l'environnement.

De manière systématique, nous cherchons à éviter de nous exprimer en termes de la représentation des objets, mais plutôt en termes des fonctions associées à ces objets. Ceci conduit à cette organisation, désormais bien connue, où l'information n'est accessible qu'à travers une enveloppe fonctionnelle qui est la définition abstraite de l'objet, en opposition à la définition concrète qu'est l'information elle-même.



Dans le chapitre qui suit nous décrivons les caractéristiques d'un système qui permet :

de modéliser la représentation d'un objet "programme"
de définir un outil prenant en charge la représentation physique d'un objet "programme" et la réalisation des opérations élémentaires associées.
Ceci définit le formalisme de description d'une forme interne.

de fournir à partir de ce formalisme des moyens d'interprétation (évaluation) de tout ou partie d'un objet "programme" suivant la sémantique d'un des outils de l'environnement que l'on construit, afin de faciliter sa programmation.

3.2 Les systèmes existants

Dans ce chapitre, nous rendons compte de l'étude que nous avons faite sur les expériences existantes dans le domaine : nous en décrivons deux qui nous parraissent les plus caractéristiques, ou tout du moins qui se rapprochent le plus de notre propre démarche.

3.2.1 Mentor-Metal

3.2.1.1 Le système Mentor

Mentor est un processeur qui permet de manipuler des informations structurées. La structure choisie dans le cadre de ce projet est une arborescence qui permet de formaliser la syntaxe abstraite des langages (principalement les langages de programmation) [Kah 78] [DHK 80] [Mel 81].
Initialemment Mentor a été appliqué au langage Pascal, il s'en suit que tous les exemples donnés dans ce paragraphe sont écrits en Pascal.

L'arborescence est composée de noeuds opérateurs et de noeuds opérandes. On distingue les opérateurs ayant un nombre de fils fixe déterminé a priori, et les opérateurs dits associatifs qui possèdent un nombre variable de fils. Un opérateur associatif permet de décrire la structure de liste. Les feuilles de l'arborescence sont considérées comme des opérateurs d'arité nulle.

Dans Mentor, les opérateurs sont regroupés en "phyla" qui sont des ensembles d'opérateurs. Chaque arbre est caractérisé par le phylum de son opérateur racine, et chaque opérateur est, lui, caractérisé par les phyla de ces opérandes.

exemple :

le phylum "stat", servant à décrire les instructions du langage Pascal, contient les opérateurs suivants :

IF FOR CASE ... LSTAT (liste d'instructions) ...

les opérateurs IF FOR LSTAT sont décrits par :

IF : expr x stat x stat -> stat

"expr" représente une expression conditionnelle

FOR : ident x step x stat -> stat

"ident" représente un nom et "step" un couple d'expressions.

LSTAT : stat ...

les points précisent que l'opérateur est associatif, ici une liste de "stat".

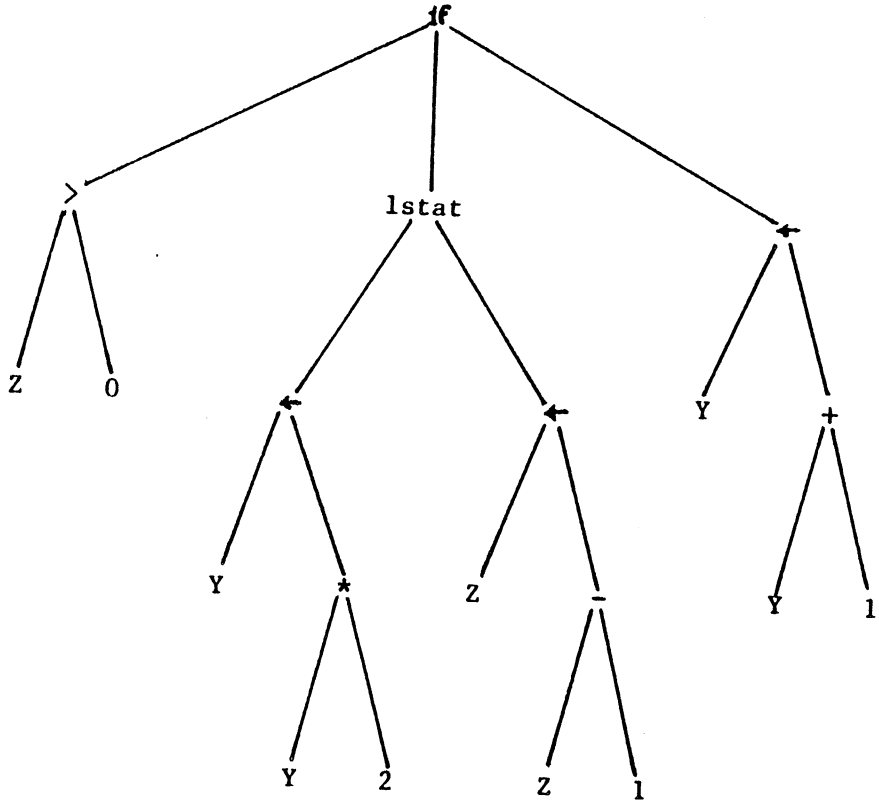
Les opérateurs associatifs ont tous leurs composants du même phylum comme on le voit dans l'exemple de la liste d'instructions ci-dessus.

Nous pouvons schématiser l'arbre de syntaxe abstraite décrit en Mentor dans l'exemple qui suit.

exemple : soit le programme Pascal :

```
if Z > 0 then begin
                  Y := Y*2;
                  Z := Z-1;
              end
else Y := Y+1;
```

on obtient l'arbre de syntaxe abstraite suivant :



Les noeuds ici sont des opérateurs et n'ont rien à voir avec la syntaxe concrète du langage, ni l'arbre d'analyse syntaxique de celui-ci.

3.2.1.2 Le langage Mentol

Mentor n'est que le processeur qui gère cette structure arborescente. Pour avoir accès à l'information contenue dans la structure, l'utilisateur doit communiquer avec le système Mentor. Il dispose pour cela d'un langage de manipulation d'arbres appelé Mentol [Mel 80].

Les valeurs manipulées par Mentol sont des arbres. Les variables de Mentol que l'on appelle des "markers" peuvent recevoir des adresses d'arbres (loc).

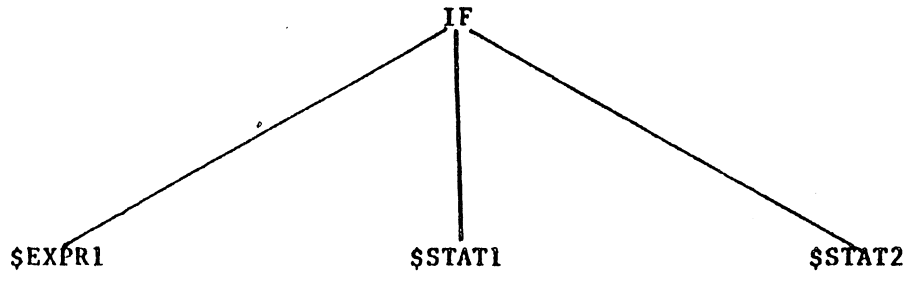
Mentol est un langage interprété qui va permettre de se déplacer dans la structure arborescente et de modifier cette structure en ajoutant des sous-arbres ou bien en en supprimant. Il permet aussi des recherches par valeur qui sont appelées "recherches programmées". Il permet de décompiler la forme interne pour obtenir une image lisible de l'information traitée.

Mentol se compose de deux niveaux. Un premier niveau est composé par l'ensemble des primitives qui définissent le langage et qui permettent à l'utilisateur certaines manipulations. Le deuxième niveau correspond à la possibilité de programmer des procédures propres à l'utilisateur ceci combiné à l'ensemble de procédures prédéfinies et les métavariables.

Les métavariables sont des modèles d'arbres. Chaque opérateur du langage est décrit par une arborescence, dont les fils sont des métavariables portant un nom qui est en correspondance avec le phylum du sous-arbre représenté. Ces métavariables sont utilisées pour occuper une place dans l'arborescence sans avoir à définir une valeur réelle pour cet arbre. Elles servent aussi pour la recherche dans l'arborescence. Une métavariable peut être instanciée avec un ensemble de sous-arbres possédant les phyla appropriés.

Exemple :

Le métavariables sont dénotées par le "$" suivi d'un nom.



Dans cet exemple, on décrit l'arbre général d'une instruction "IF". $EXPR1, $STAT1 et $STAT2 sont des méta-variables. Le nom des méta-variables est constitué à partir de leur phylum : $EXPR1 est du phylum "expr", $STAT1 est du phylum "stat", ..., ce qui permet de savoir quel type d'arbre peut instancier la méta-variable.

Exemple de procédures prédéfinies :

EQTYPE <<@1>,<@2>>

réussit si les deux arbres ont le même opérateur comme racine.

EQUAL <<@1>,<@2>>

réussit si les deux paramètres désignent le même arbre.

IS <<repère>,<@1>>

réussit si l'arbre est une instance de l'arbre "repère".

APL <action>

applique l'action à tous les éléments de la liste sur laquelle on se trouve puis revient au point de départ.

FORALL <<repère>,<action>>

cherche toutes les occurrences du schéma "repère", en ordre préfixé depuis le point courant. Elle applique l'action sur chaque occurrence trouvée.

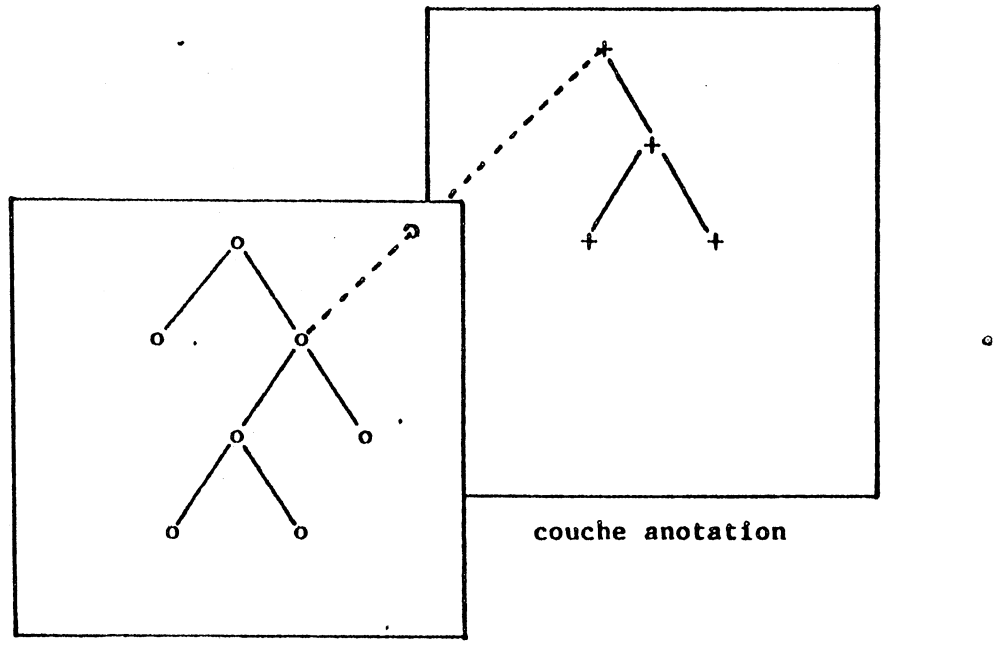
3.2.1.3 La structure en couche

Le système Mentor manipule une structure d'arbre conforme à l'arbre abstrait d'un programme Pascal. Cette structure abstraite ne prend en compte que les éléments pertinents du langage. Entre autres les mots clés du langage n'apparaissent pas dans cette structure. De même les commentaires qui ne font pas partie intégrante du langage sont omis de l'arbre abstrait.

Ceci n'est pas satisfaisant pour au moins deux raisons ; D'une part le système Mentor permettant de manipuler des programmes Pascal doit effectivemment restituer ceux-ci dans leur intégralité (commentaires compris), d'autre part les commentaires sont un moyen de préciser le programme lui-même. Ils ont une importance non négligeable pour une bonne compréhension du programme, et peuvent être eux-mêmes des informations structurées.

Mentor permet d'associer à chaque noeud de l'arbre un structure d'information qui n'est pas considérée de manière identique par le processeur : il la situe dans un niveau différent (couche anotation). Il faut préciser que l'on peut reproduire ce découpage en couche plusieurs fois, obtenant ainsi une couche anotation d'une couche elle-même anotation.

Exemple :



3.2.1.4 Le système Métal

Le systeme Mentor est lié au langage Pascal. Le processeur ne connait que les arborescences nécessaires à la description du langage Pascal, et ne peut manipuler que celles-ci. Pour rendre le système indépendant du langage, les auteurs ont ajouté au système initial un mécanisme de description des formalismes (langages). Ce mécanisme s'appelle Métal [Mel 82].

Pour manipuler un nouveau formalisme en Mentor-Métal il faut tout d'abord commencer par décrire dans le langage Métal ce formalisme. La compilation d'un programme Métal fournit un ensemble de tables qui permettent au système de manipuler ce formalisme et un ensemble de fonctions pour la construction de l'arbre associé, la décompilation,etc...
Après cette compilation, on utilise le système Mentor_Mentol sans différence avec ce qui a été présenté dans le paragraphe précédent.

La description du nouveau formalisme dans le langage Metal est constituée en chapitres. Dans chaque chapitre on décrit la syntaxe concrète de la partie concernée du formalisme, ainsi que la syntaxe abstraite représentant les phyla qui seront manipulables par le système.

Le système peut être multi-formalisme. Plusieurs formalismes pouvant être utilisés dans une même structure arborescente, les sous-arborescences de l'arbre général sont caractérisées par le formalisme dans lequel elles ont été construites. Ceci permet d'avoir dans un programme Pascal des sous-arbres décrits dans un langage de prédicat pour noter les assertions. On peut de même structurer les commentaires en les assujetissant à un modèle particulier.

3.2.2 Le système Delta

Les objectifs du système Delta [Lor 74] [Bla 73] sont de fournir un outil adapté permettant de décrire les compilateurs des langages de programmation. Delta se présente comme un méta-compilateur. Sa structure est donc voisine de celle des compilateurs eux-mêmes, avec une analyse lexicale, une analyse syntaxique et une évaluation sémantique.

Nous retrouvons ces différents aspects dans Delta, mais organisés de manière quelque peu différente. Cette particularité est due aux objectifs de Delta, qui sont de fournir un cadre privilégié pour décrire et manipuler la sémantique des langages.

Le système Delta permet de construire :

un analyseur syntaxique du langage considéré, à partir d'une définition donnée sous forme BNF.

un analyseur sémantique à partir d'un ensemble de fonctions écrites dans un langage au choix de l'utilisateur (pascal, fortran, lisp, ...).

La création de ces analyseurs constitue la première phase du processus à suivre pour utiliser le système Delta. Cette première phase s'applique à la définition du langage.

La deuxième phase s'applique à un programme écrit dans ce dit langage et se décompose en deux étapes. Tout d'abord, le système effectue une analyse syntaxique du programme. Pendant cette analyse, le système Delta construit le système d'équations représentant l'évaluation sémantique du programme fourni. L'étape suivante correspond à la résolution de ce système d'équations.

L'analyse syntaxique est faite à l'aide d'un analyseur ascendant de type LR(k) produit automatiquement par le système Syntax [Bou 80].

Dans la première phase, le système analyse les définitions sémantiques et construit une représentation interne de ces définitions pour les phases ultérieures. Toujours dans cette étape, une vérification de la cohérence du système d'équations peut être faite. Cette vérification qui est coûteuse, concerne l'unicité de la définition d'un attribut, et de la non circularité du système. Elle permet ainsi de certifier que toute évaluation sémantique aboutira.
La représentation interne des définitions sémantiques sert lors de l'analyse syntaxique d'un programme pour produire le système d'équations associé.

La résolution du système d'équations qui a été produit pendant l'analyse syntaxique permet de produire soit directement le résultat de l'évaluation sémantique, soit un programme séquentiel qui, lui-même interprété par un programme "évaluateur", fournit le résultat de l'évaluation sémantique. Une phase d'élimination des impasses peut être effectuée avant la résolution du système. Cette phase consiste à éliminer les équations de définition des attributs qui n'interviennent pas dans le calcul des attributs caractérisant le résultat, minimisant ainsi le travail de l'évaluation.

Exemple de définition :

Il s'agit d'un exemple proposé par Knuth, qui calcule la valeur décimale d'un nombre écrit en numérotation binaire [May 78].
Dans la partie gauche on trouve la grammaire décrivant un nombre binaire, et dans la partie droite la définition des attributs sémantiques.

"n" est un nombre sous forme binaire
"s" est une suite de chiffres binaires
"b" est un chiffre binaire
"V" est l'attribut qui synthétise la valeur décimale
"P" est l'attribut qui hérite le poids du chiffre binaire
"I" est l'attribut qui synthétise la taille de "s"

n → s	V(n) = V(s) P(s) = 0
n → s.s	V(n) = V(s1)+ V(s2) P(s1) = 0; P(s2) = -I(s2)
s → b	V(s) = V(b); I(s) = 1 P(b) = P(s)
s → sb	V(s) = V(s1)+V(b); I(s) = I(s1)+1 P(s1) = P(s)+1; P(b) = P(s)
b → 0	V(b) = 0
b → 1	V(b) = 2↑P(b)

Lorsqu'un même symbole apparait plusieurs fois dans la partie droite d'une règle de production, on indice ces occurences par leur numéro d'apparition dans celle-ci.
Dans cet exemple on remarque que pour évaluer l'attribut V(s2), on doit avoir évalué l'attribut I(s2), ce qui nécessite deux passages sur le sous-arbre s2.

L'analyse syntaxique utilisée a une influence sur l'évaluation des attributs.

Nous reprenons l'exemple précédent avec une analyse descendante déterministe.

n → s	V(n) = V(s) P(s) = 0
n → s.s	V(s) = V(s1)+V(s2) P(s1) = I(s1); P(s2) = -1
s → b	V(s) = V(b); I(s) = 1 P(b) = P(s)
s → bs	V(s) = V(s1)+V(b); I(s) = I(s1)+1 P(s1) = P(s)-1; P(b) = P(s)
b → 0	V(b) = 0
b → 1	V(b) = 2↑P(b)

On voit que les changements concernent l'attribut P qui est effectivement dépendant du sens de parcours de la liste de chiffres binaires (b).

La solution idéale pour ce problème serait de disposer de la première solution pour traiter la partie entière du nombre, et de la deuxième solution pour traiter la partie décimale. L'attribut "I" ne serait alors plus nécessaire, et par conséquent un seul balayage suffirait.

3.3 Description d'un système de production

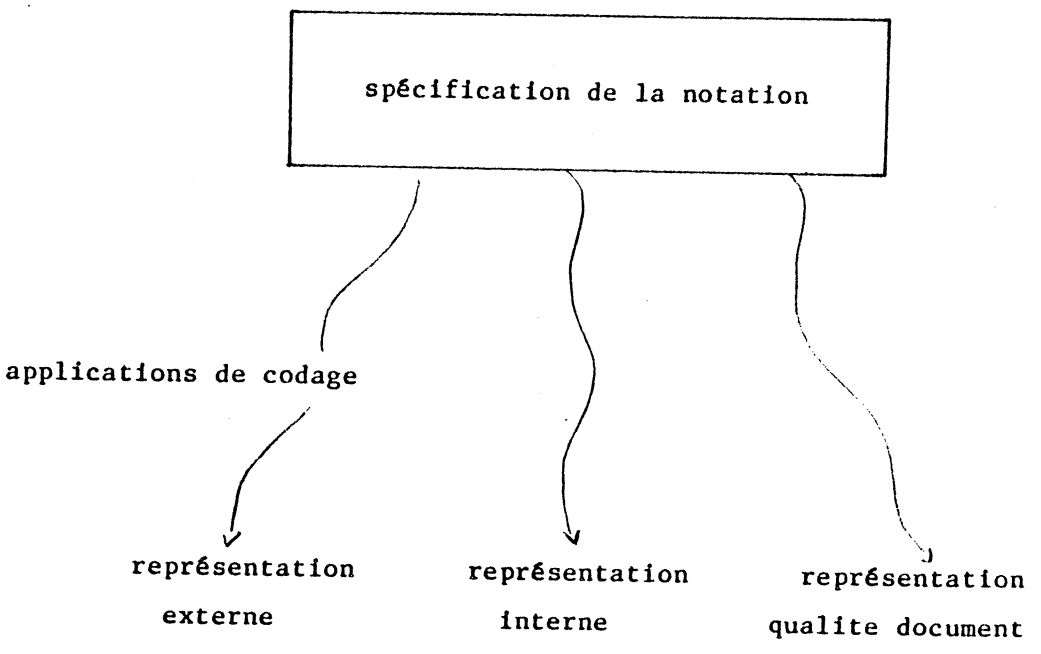
3.3.1 Introduction

Dans la démarche que nous avons suivie la notation algorithmique est la base de l'environnement [Bux 80] [MKS 76] [MRR 82] [Wat 77]. Elle sert, tout au long du processus de construction de programme, dans les différentes activités de la programmation. Elle définit les moyens que l'on se donne pour résoudre les problèmes ; En ce sens la notation n'est qu'un ensemble de fonctionnalités, offrant un niveau d'abstraction bien défini, permettant de décrire les solutions produites.

La notation n'a pas de représentation particulière, mais un ensemble de représentations suivant les conditions d'utilisation. La façon de présenter la notation algorithmique dépend du cadre dans lequel on veut en faire l'usage. Nous repérons très vite au moins trois représentations distinctes correspondant à trois usages différents de la notation, la représentation externe, la représentation interne et, en fin, une représentation de qualité "document".

Chacune de ces représentations répond à un ensemble bien particulier de critères. On comprend que les critères d'une représentation de type document, qui nécessite d'être lisible avec un grand effort de présentation, n'imposent pas les mêmes contraintes que celles d'une représentation interne, pour laquelle on va plutôt insister sur la performance des calculs.

La figure ci-après représente un état idéal, où l'on disposerait d'une description très générale de la notation. Cette description servirait pour produire les différentes représentations nécessaires.



Dans notre système, nous privilégions la représentation interne [MKS 79] [Wat 79] [BLW 79]. Les programmes seront décrits et manipulés dans cette représentation. C'est à partir de celle-ci que l'on produira les autres représentations si nécessaire. Il s'en suit que la représentation interne doit être le résultat d'un certain nombre d'adéquations, ce qui rend son élaboration difficile et justifie le fait qu'elle puisse évoluer durant la période de conception.

Elle doit respecter la structuration profonde de la notation qu'elle représente. Cette structuration est arborescente et doit contenir tout le message sémantique que l'on attribue à l'objet "programme" ainsi représenté.

Elle doit respecter des contraintes techniques, qui vont de la taille aux caractéristiques d'accès à l'information, en passant par la dynamicité et la modifiabilité de l'information.

3.3.1.1 Objectifs du système

Le système que nous voulons réaliser doit intégrer la gestion de la structure arborescnte interne. La définition des constituants de cette arborescence doit se faire au travers des noms donnés par l'utilisateur (ayant un sens pour lui), et non pas par le biais de fonctions générales des arbres (fils,frère,père,...). Ceci a pour but d'offrir un système ayant une bonne lisibilité.

Le système devra fournir la possibilité de décrire des calculs (programmation). Cette description devra s'intégrer à la structure utilisée, et être bien adaptée à tous les types de programmes à produire. Cette programmation devra pouvoir se réaliser de façon modulaire, afin de minimiser la lourdeur d'écriture et de lecture des programmes produits.

3.3.2 Structure d'information et opérations

Un programme est une structure arborescente. Ceci provient du fait qu'un programme est composé de constituants plus élémentaires que l'on assemble ou que l'on imbrique pour créer des constituants plus complexes. Cette décompostion se fait jusqu'à ce que l'on obtienne un niveau de base où les constituants sont des atomes.

Le modèle d'arbre que l'on veut définir, pour représenter la notation, possède des noeuds ayant un nombre quelconque de fils (arité non fixe) : on le qualifie de n-aire. Le contenu des noeuds n'est pas identique d'un noeud à un autre : on dit que l'arbre est hétérogène.

La modélisation de la représentation d'un objet "programme" revient à définir quels sont les modèles élémentaires décrivant les constituants simples de la notation, et quels sont les outils de composition dont on a besoin pour exprimer la structure hiérarchique de la notation.

3.3.2.1 Terminologie

Nous utiliserons au cours de ce chapitre un certain nombre de termes qui nous sont propres, pour lesquels nous donnons ici une définition.

Une "**construction**" :

une entité qui est définie au niveau de la notation. Par exemple, type_entier, déclaration_variable,...etc.

Un "**modèle**" :

Le modèle est constitué d'un nom et d'une spécification de la représentation interne de l'ensemble d'informations qui servent dans la définition de la construction concernée. Par exemple, les modèles tantque, déclaration_variable, ...etc.

Une "**valeur**" :

un ensemble d'informations internes qui vérifient la définition d'un "modèle".

Une "opération" :

une manipulation d'une "valeur".

Un "arbre du programme", "sous-arbre" et "arborescence" :

structure de donnée représentant l'information du programme.

Une "modélisation" :

ensemble de modèles concourant à définir un objet.

3.3.2.2 Les modèles élémentaires

Ces modèles élémentaires correspondent aux domaines en termes desquels s'exprime la notation. Ils sont le moyen de fixer l'abstraction minimum dont on dispose dans la notation.

On considère, d'une part les modèles élémentaires non standards définis par l'utilisateur. Ces modèles définissent les concepts que l'on ne décompose pas dans la notation. Par exemple les modèles "réel" et "caractère",..., qui représentent des concepts de base pour la notation.

D'autre part les modèles élémentaires standards. Ces modèles sont des outils nécessaires à la manipulation de la forme interne. Ils sont au nombre de trois pour l'instant, et pourront être complétés par d'autres ultérieurement. Ces modèles ne font, bien sûr, pas partie de la notation que l'on veut modéliser, mais du contexte initial de toute modélisation.

3.3.2.2.1 Le modèle "nom"

Le modèle "nom" est introduit par le besoin toujours existant de désigner des "entités" par un nom. Il est lié à la notion de "table" et de "recherche dans une table". Le nom servira à définir dans le système les différentes tables qui associent à un nom une déclaration. Par exemple, la réalisation de l'"environ" d'un bloc d'instructions.

Remarque :

Un environ correspond à l'ensemble des objets, et leur définition, accessible depuis un certain point du programme. Il définit le contexte dans lequel les instructions prennent leur sens.

3.3.2.2.2 Le modèle "entier"

Le modèle "entier" décrit des valeurs entières. Il permet de représenter le cardinal d'une liste ou d'un ensemble. Il permet d'associer à un modèle récursif le niveau de récursivité. Il permet le codage d'un ensemble ordonné de valeurs, facilitant la manipulation de cet ensemble. Bien que ces nombres soient toujours recalculables lors d'un parcours de la structure arborescente, leur mémorisation simplifie l'expression des programmes décrits sur cette modélisation.

3.3.2.2.3 Le modèle "logique"

Le modèle "logique" permet de manipuler les valeurs logiques dans le système. Il est utile, pour conditionner les évaluations et pour exprimer les propriétés du modèle. Il devient nécessaire pour intégrer, de façon complète, la programmation de l'environnement au niveau de la modélisation. On verra plus tard, qu'il peut servir dans la définition de modèles virtuels liés à l'évaluation conditionnelle.

Les modèles élémentaires standards que l'on présente dans les trois paragraphes suivants, sont introduits pour faciliter et rendre orthogonale la mise en oeuvre du système. On leur associe certaines restrictions de manipulation vis-à-vis de l'utilisateur.

3.3.2.2.4 Le modèle "code"

Le modèle "code" permet de décrire les noms des modèles, et d'utiliser ces noms comme des variables dans une évaluation d'un arbre du programme.

Il sert dans la définition des "listes" et des "ensembles" pour caractériser le type d'informations modélisées par ces constructeurs.

Bien que ce nom de modèle soit accessible à l'utilisateur, nous ne fournissons pas, à l'heure actuelle, de moyen pour construire des informations de ce modèle. Ceci reste le privilège du système. Nous verrons un exemple dans le paragraphe 3.2.4.8 où ce modèle apparait comme un type énuméré décrivant tous les modèles connus du système.

3.3.2.2.5 Les modèles "adresse" et "terminal"

Ces modèles sont des méta-modèles qui permettent de parler soit de modèles non élémentaires (qui sont représentés à l'aide d'un sous-arbre), soit de modèles élémentaires. Ils permettent de ne pas considérer le type de l'information lorsqu'on aborde celle-ci avec un de ces modèles.
Ces modèles permettent de manipuler la structure arborescente sans tenir compte de ce qu'elle représente.

Ces noms de modèles ne sont pas accessibles à l'utilisateur, ils servent dans le noyau du système à la réalisation des primitives.

3.3.2.2.6 Les modèles "elt", "ens", "lst" et "vide"

Les modèles "ens", "lst" et "elt" sont des codes discriminants que l'on associe automatiquement à certains modèles produits par la macro-génération des "listes" et "ensembles". Le modèle "vide" permet de construire les sentinelles en fin des "listes" ou des "ensembles".

Pour tous ces modèles on trouvera des exemples dans le paragraphe 3.3.2.3. Ils sont surtout nécessaires au système. Ils sont aussi accessibles à l'utilisateur qui doi les utiliser avec précaution.

Les modèles élémentaires standards sont des modèles prédéfinis. Les modèles liés à la notation considérée, seront des "paramètres" de la modélisation.

Les modèles "nom", "entier", "logique" et "code" ne peuvent pas servir à discriminer les descriptions simples d'une sélection (voir @3.3.2.3.2).

3.3.2.3 Les constructeurs

Les constructeurs sont au nombre de trois et permettent de créer de nouveaux modèles par composition de modèles plus élémentaires. Nous retrouvons ici les décompositions que nous avons déjà présentées dans la première partie. Il s'agit, de la discrétisation qui se traduit pour une information par la définition d'une "**structure**", de la partition représenté par ce que l'on nommera désormais la "**selection**", de l'itération à laquelle on fait correspondre la "**liste**".

Ces trois constructeurs forment la base qui permet de modéliser de façon minimum la notation. A ceux-ci on ajoute d'autres constructeurs qui vont permettre d'enrichir la modélisation. Il s'agit du constructeur "**ensemble**" lié au modèle standard "nom", et qui permet de définir, entre autre, les environs dans l'arbre du programme. Du constructeur "**lien**" qui permet de faire référence à un sous-arbre attaché en un autre point de l'arbre. Du constructeur "**synonyme**" qui permet de renommer un modèle déjà existant.

3.3.2.3.1 Le constructeur "structure"

Il permet d'exprimer le fait qu'une construction est raffinée en un ensemble de constructions hétérogènes non ordonnées. Par exemple on exprime qu'un programme est composé d'un partie déclaration et d'une partie instruction ..., ou bien qu'un schéma "tantque" est composé d'une condition et d'une suite d'instructions. Le constructeur "structure", comme dans la notation algorithmique, n'ordonne pas les composants. L'exemple suivant illustre la notation que nous utiliserons désormais pour noter un modèle défini à l'aide de ce constructeur

modèle tantque est struct(expression,instruction)

les modèles correspondent à une déclaration des types de données. Dans notre exemple, "expression" définit une classe d'informations construites suivant un certain modèle. Dans le cadre de la modélisation on ne fait apparaître que le nom du modèle (type) d'arbre.
Il est nécessaire, pour qu'il n'y ait pas d'ambiguité sémantique, que les modèles apparaissant dans une même construction soient deux à deux différents. Cette règle qui peut apparaître comme contraignante, évite de simplifier abusivement la sémantique associée aux constructions.

L'exemple suivant est ambigu et correspond à une perte d'informations au moment de la modélisation.

modèle expr_bin est struct(opérateur,expr,expr)

Dans cet exemple on décide de façon implicite d'attribuer des sémantiques différentes aux deux occurences du modèle "expr". L'implicite ne peut reposer que sur l'ordre d'apparition des occurences, ce qui est en condradiction avec la définition du constructeur "structure". Nous verrons dans le paragraphe 3.2.3.6 comment remédier à ce problème.

Un modèle décrit avec un constructeur "structure" ne peut pas apparaître dans sa propre définition. Effectivement dans cette alternative la structure d'information décrite est infinie. Cette règle peut être assouplie en admettant que le processeur réalisant le parcours de la structure d'information sache reconnaître les valeurs neutres (@3.3.2.4.7) des différents modèles.

3.3.2.3.2 Le constructeur "sélection"

On introduit des méta-constructions afin de mettre en évidence la structure sémantique d'une notation lors de sa définition. Une méta-construction est ensuite décrite en terme d'un choix entre des constructions de même niveau. Le constructeur "sélection" permet ainsi de définir une construction en termes d'un ensemble possible de constructions jugées de même niveau sémantique.

L'exemple suivant décrit une entité procédure :

```
modèle dcl_proc est sélect(dcl_action,dcl_fonction)
```

Dans cet exemple une "procédure" est soit une "action" soit une "fonction". On regroupe ces modèles sous un modèle unique "dcl_proc" car ils sont extrêmement liés au niveau sémantique.

Lors de la consultation d'une information décrite par un constructeur "sélection", il faut être capable de définir quel est le modèle, composant de la "sélection", qui a été choisi lors de la création de cette information. Pour ce faire, il est nécessaire d'associer un code discriminant à chaque composant de la "sélection". Ce code servira à qualifier le modèle qu'il désigne par rapport à l'ensemble des modèles définis dans la sélection. On notera alors les modèles "sélection" de la facon suivante :

```
modèle dcl_proc est sélect(c_action  :dcl_action
                           c_fonction:dcl_fonction)
```

Les termes "c_action" et "c_fonction" sont les codes discriminants du modèle "dcl_proc". On peut remarquer que les codes discriminants d'une "sélection" et les modèles élémentaires servent dans tous les cas à caractériser des descriptions. Pour cette raison les codes discriminants seront considérés comme des modèles élémentaires et par conséquent fournis par l'utilisateur.

Le constructeur "sélection" permet de modéliser des informations qui comportent dans leur description leur propre définition. Reprenons les expressions comme exemple :

```
modèle expr est sélect(c_opérande : opérande
                       c_expr     : struct( opérateur
                                            expr
                                            expr))
```

3.3.2.3.3 Le constructeur "liste"

Le constructeur "liste" permet d'exprimer qu'une construction est raffinée en une famille de constructions homogènes ordonnées. Chaque élément de la famille sera discriminé par sa position. Par exemple dans l'appel d'une procédure avec des paramètres nous utiliserons le modèle suivant :

```
modèle lst_param est liste(param)
```

Dans cet exemple, le modèle "param" décrit comment est constitué un paramètre d'une procédure. On différencie un paramètre d'un autre par les positions qu'ils occupent dans la liste.

Pour produire un tel raffinement, il faut tout d'abord établir un ordre sur les constituants, puis représenter cet ensemble à l'aide du constructeur "liste".

Le constructeur "liste" n'est pas un constructeur de base, car on peut le réaliser à l'aide des constructeurs "structure" et "sélection". On peut dire qu'il s'agit en fait d'un macro-constructeur qui engendre un ensemble de modèles définissant une liste.

```
modèle lst_param      est struct("lst",
                                 adr(dsc_lst_param),
                                 elt_lst_ param)

modèle dsc_lst_param est struct("code",
                                 "entier",
                                 lien(elt_lst_param))

modèle elt_lst_param est sélect("vide":
                                 "elt":struct(
                                        param,
                                        elt_lst_param))

modèle param est ...
```

Un modèle sera rarement décrit comme étant uniquement une "liste", ce qui ne ferait que rajouter un niveau de description. La "liste" sera utilisée dans la description d'un modèle comme dans l'exemple suivant :

```
modèle dcl_action est struct(nom,liste(param))
```

Remarque :

adr(xxx) indique que le modèle "xxx" est occulté pour le parcours de l'arborescence (@3.3.3.4).

3.3.2.3.4 Le constructeur "ensemble"

Il permet d'exprimer qu'une construction est raffinée en un ensemble de constructions homogènes non ordonnées. Un nom est associé à chacune d'elles, ce qui permet de les différencier. Par exemple la partie déclarative d'un bloc pourra être représentée par un ensemble :

```
modèle dcl est ensemble(déclare)
```

Cette définition indique que l'on définit une table dont les indicatifs sont des "noms" auxquels l'application fait correspondre une déclaration représentée par le modèle "déclare". Les indicatifs étant implicites, ils ne sont pas spécifiés au niveau de la modélisation. Comme pour "liste", "ensemble" engendre trois modèles :

```
modèle ens_dcl      est struct("ens",
                             adr(dsc_ens_dcl),
                             elt_ens_dcl)

modèle dsc_ens_dcl est struct("code",
                            "entier",
                            lien(elt_ens_dcl))
```

```
modèle elt_ens_dcl est sélect("vide":
                              "elt":struct(
                                    "nom",
                                    déclare,
                                    elt_ens_dcl))

modèle déclare est ...
```

Le constructeur "ensemble" n'est pas purement structurel (servant à décrire l'information minimale) : l'ensemble des noms associés aux informations permet de recourir à des fonctions supplémentaires. Par exemple :

L'appartenance à un ensemble, ou bien l'insertion d'un nouvel élément dans l'ensemble, le tri suivant l'ordre lexicographique des noms,...

Les modèles "dsc_ens_xxx" et "dsc_lst_xxx" sont les descripteurs de l'ensemble (resp liste). Ils contiennent un "code" qui donne le modèle des éléments contenus, un "entier" qui est le cardinal de l'ensemble (resp liste) et un lien vers le dernier élément qui permet d'accélérer les opérations d'insertion (il s'agit de l'accès au dernier élément valide).

Notons que nous pouvons aussi choisir de représenter les éléments d'une "liste" ou d'un "ensemble" de la façon suivante :

```
modèle elt_lst_xxx est struct(xxx,elt_lst_xxx)
```

et

```
modèle elt_ens_xxx est struct("nom",xxx,elt_ens_xxx)
```

La détermination de la fin de la structure ainsi représentée se fait par l'utilisation de la valeur neutre du modèle elt_zzz_xxx (voir @3.3.2.4.7).

3.3.2.3.5 Le constructeur "lien"

Le constructeur "lien" est particulier, il est d'ordre structurel puisqu'il permet d'indiquer qu'un noeud de l'arbre du programme possède comme fils un sous-arbre désigné par le lien. Le modèle utilisant le constructeur "lien" ne possède effectivement pas (en propriété) l'information associée au lien. Mais, lors d'un parcours de l'arborescence, on aura accès à ce sous-arbre.

Ce constructeur permet en fait de figer un certain nombre d'attributs sémantiques. Ces attributs sont considérés comme toujours nécessaires quel que soit l'outil à décrire. Afin de ne pas utiliser le mécanisme d'attributs qui est dynamique, on calcule une fois pour toute l'attribut en question que l'on conserve dans la modélisation grâce au constructeur "lien" (il s'agit le plus souvent de l'information nécessaire à l'évaluation de l'attribut). Par exemple si dans une instruction on utilise une variable, celle-ci apparaitra de la manière suivante :

```
modèle var_utilisée est struct("nom",lien(déclare))
```

ou bien

```
modèle var_utilisée est lien(elt_ens_dcl)
```

avec

```
modèle dcl est ensemble(déclare)
```

Dans cet exemple, on a une utilisation de variable, on dispose d'un ensemble qui réalise l'association "nom", "dcl".

Pour cette utilisation on calculera une seule fois le lien à la déclaration, lien qui sera stocké par le constructeur "lien". C'est ici, on voit, une facon simple et peu coûteuse de mémoriser un environ associé à une instruction.

Le constructeur "lien" permet d'enrichir, de manière importante, la structure d'information. Il permet de créer une redondance de l'information sans être obligé de dupliquer cette information : c'est du point de vue manipulation de la donnée une caractéristique intéressante.

Remarque :

Ce constructeur introduit un inconvénient non négligeable : il modifie la structure parfaite de l'arbre, en la transformant en un graphe quelconque. Le lien peut, et ceci dans un nombre important de cas, référer un de ses propres aïeux, créant ainsi un pseudo-cycle dans la structure d'information.

3.3.2.3.6 Le constructeur "synonyme"

Le constructeur "synonyme" permet de renommer un modèle défini par ailleurs. Il intervient chaque fois que deux sous-arbres sont de même nature. Mais ne se trouvant pas au même endroit dans la modélisation, on veut, par le nom qu'il leur est attribué, faire apparaître cette analogie. Ce constructeur permet de pallier au fait que les modèles sont des types de données, et non des informations.

Reprenons l'exemple donné en 3.3.2.3.1 :

Dans cet exemple il apparaissait deux fois le modèle "expr", ce qui était ambigu. Pour lever cette ambiguïté nous proposons la définition suivante :

```
modèle expr_bin est struct(opérateur,opg,opd)
avec
    modèle opg est synonyme(expr)
    modèle opd est synonyme(expr)
```

"opg" et "opd" sont de type "expr" (modèle "expr"), mais on note en plus que "opg" dénote un "expr" apparaissant à gauche de l'opérateur, et pour "opd" un "expr" apparaissant en opérateur droit.

On utilise aussi le constructeur "synonyme" pour redéfinir des modèles élémentaires standards dont les noms ne comportent aucune caractéristique spécifique. Ce constructeur ne peut s'appliquer qu'aux modèles "entier", "nom" et "logique".

Par exemple :

```
modèle nb_dcl est synonyme("entier")
modèle égalité_type est synonyme("logique")
```

On admettra aussi la possibilité de renommer les modèles "code", "adresse" et "terminal" pour une meilleure lisibilité des programmes (voir @3.3.4.6 pour une utilisation dans un programme système).

Le constructeur "synonyme" permet d'étendre le répertoire des noms mis à la disposition de l'utilisateur. Il lui permet d'ajouter une sémantique supplémentaire à un modèle existant.

Remarque :

un modèle ne peut être synonyme d'un modèle lui-même synonyme. Cette indirection n'apporterait rien de nouveau pour le modèle ainsi défini.

3.3.2.4 Les opérations

Dans ce paragraphe nous décrivons les opérations primitives que l'on associe à la structure d'information, pour faciliter sa manipulation.

Ces opérations doivent permettre de créer, de consulter et de modifier une valeur vérifiant un modèle particulier. L'environnement de programmation évoluant dans le temps, à un instant précis les besoins des outils sont partiellement définis. C'est pourquoi les opérations que nous proposons à ce stade se restreignent aux manipulations élémentaires de l'information.

Nous parlerons dans ce paragraphe uniquement de la "création" d'une valeur. Nous avons lié la consultation d'une valeur au problème de l'interprétation (évaluation) d'une représentation interne d'un objet "programme". Les modifications d'une valeur étant a priori peu importantes, elles sont réalisées par la "re-création" de la valeur que l'on désire modifier, ceci à l'aide des opérations de consultation et de création.

Les opérations de "création" dépendent des différentes descriptions d'un modèle et des différents modèles. Ces opérations doivent assurer la correction de la représentation interne conformément à la modélisation qui en a été faite.

3.3.2.4.1 Création de la valeur d'un modèle élémentaire

Il n'existe des opérations de création que pour les modèles "nom", "entier" et "logique".

Les autres modèles élémentaires étant la définition de constantes, le système connait leur valeur,et il est capable de les engendrer au moment opportun.

```
ctr_nom(code_lex cl)    -> nom
ctr_entier(integer e)   -> entier
ctr_logique(booleen b) -> logique
```

"code_lex" est le type d'information représentant un nom dans la lexicographie du programme. "integer" et "booleen" sont les types habituels des valeurs entières et logiques. "nom", "entier" et "logique" sont les types des informations associées respectivement aux modèles "nom", "entier" et "logique".

Remarque :

Dans cette démarche, on accède aux valeurs par un nom qui est soit fourni par l'utilisateur, soit élaboré par le système lexicographique. Une constante immédiate entière de la notation algorithmique peut apparaitre de la manière suivante :

xxx est struct("cste_entière","nom").

Dans cette alternative le processeur fournissant le "nom", doit aussi fournir les moyens de connaitre la nature de la valeur associée à ce "nom" (par exemple : entier, réel, logique, idf,...).

Dans le cas d'une constante immédiate, celle-ci est conservée dans la valeur lexicale associée au nom de la constante.

On dispose de fonctions qui permettent de récupérer les propriétés des objets repérés par un nom. Nous pensons ici par exemple à la valeur lexicale associée au nom, à la nature de l'objet (cste entière, cste logique, identificateur,...).

3.3.2.4.2 Création de la valeur de modèle structure

A chaque modèle décrit par un constructeur "structure" est associée une opération de "création" ayant pour paramètres la liste des valeurs des modèles constituants la structure, dans l'ordre défini par la description du modèle. Les valeurs correspondants aux modèles élémentaires pour lesquels il n'y a pas d'opération de "création" sont omises de la liste de paramètres. Ils sont automatiquement construits par le système.

Par exemple au modèle suivant :

```
modèle tantque est struct("tantque",
                          condition,
                          instruction)
```

correspond l'opération de création :

```
ctr_tantque(condition C, instruction I) -> tantque
```

Cette fonction prend comme paramètres deux arborescences, la première associée au modèle "condition", la seconde associée au modèle "instruction", et construit une nouvelle arborescence, ayant pour fils les deux premières, qui sera associée au modèle "tantque".
Toutes ces informations sont des t-uples comportant un accès à un arbre et un type qui est le modèle d'arbre supporté par cet accès. Toutes les fonctions peuvent donc vérifier les paramètres qui leurs sont fournis.

3.3.4.2.3 Création de la valeur d'un modèle sélection

A chaque modèle décrit à l'aide du constructeur "sélection" est associé un ensemble d'opérations de création. Cet ensemble est défini par les opérations associées à chaque description simple composant la sélection. Ces opérations ont pour nom général "ctr_xxx", ou "xxx" représente le nom du modèle, et pour suffixe le nom du modèle élémentaire qui discrimine la decription simple correspondante.

Par exemple au modèle "dcl_proc" :

```
modèle dcl_proc est sélect(c_action  :dcl_action,
                           c_fonction:dcl_fonction)
```

on fait correspondre les constructions suivantes :

```
ctr_dcl_proc_c_action(dcl_action A) -> dcl_proc
ctr_dcl_proc_c_fonction(dcl_fonction F) -> dcl_proc
```

Généralement les descriptions simples d'une "sélection" sont des "structures", la spécification des opérations retranscrit la différenciation qui est faite.

3.3.2.4.4 Création de la valeur d'un modèle liste

Le modèle "liste" correspond à un noeud d'arité variable (le nombre de ces fils peut varier). Pour cette raison, la création d'une liste se fait en deux temps : initialisation de la liste en créant une liste vide (ne comportant aucun élément), puis insertion d'éléments dans la liste existante. On définit trois primitives associees aux listes :

ctr_lst_xxx -> lst_xxx
ou "xxx" est le nom du modèle des éléments de la liste.

int_lst(lst_xxx L, xxx E)
inq_lst(lst_xxx L, xxx E)

Ces deux dernières primitives sont génériques à toutes les listes.

Les primitves "int_lst" et "inq_lst" sont respectivement l'insertion de l'élément "E" en tête et en queue de la liste "L". Le système vérifie la cohérence entre le modèle des éléments de la liste et le modèle de l'élément à insérer. En cas d'incohérence une erreur est produite.

3.3.2.4.5 Création de la valeur d'un modèle ensemble

Comme pour une liste, les éléments d'un ensemble sont acquis au fur et à mesure de l'interprétation du texte initial. C'est pourquoi on dispose d'un ensemble de primitives proches de celles élaborées pour le constructeur "liste". On possède donc deux primitives : initialisation de l'ensemble, et insertion d'un élément dans l'ensemble.

ctr_ens_xxx -> ens_xxx
ou "xxx" est le nom du modèle associé au élément de l'ensemble.

ins_ens(ens_xxx E, nom N, xxx X) -> logique

ou E est l'ensemble dans lequel on insère,
N est le nom discriminant,
X est la valeur de l'application associée au nom.

Cette opération vérifie la cohérence de l'insertion. Si le modèle associé à l'ensemble ne correspond pas au modèle fourni en paramètre on produit une erreur. Si le nom passé en paramètre de la primitive existe déjà dans l'ensemble, celle-ci renvoie la valeur faux et aucune modification n'intervient dans l'ensemble. Dans le cas contraire l'insertion est effectuée et la valeur vrai est restituée.

3.3.2.4.6 Les valeurs des constructeurs "lien" et "synonyme"

Le constructeur "lien" correspond à un emprunt d'un sous-arbre. Il ne définit pas une valeur effectivement présente à l'endroit où le lien est utilisé. Donc il n'y a pas d'opération particulière attachée à ce constructeur.

Le constructeur "synonyme" permet de renommer un modèle déjà existant pour lequel on possède une ou plusieurs opérations de création. Mais au niveau du typage des informations, une information décrite avec un modèle M, et une information décrite avec un synonyme de M (que l'on notera SM), n'ont pas le même type. Ce constructeur ne définit pas un nouveau modèle, mais des nouveaux noms pour les compositions de modèle. Il s'en suit qu'initialement on ne crée que des valeurs du type initial M, puis lors de la création d'un modèle utilisant SM, il faut alors effectuer une conversion du type M dans le type SM. Si par contre on a déjà un paramètre bien typé en SM alors aucune conversion n'est nécessaire.

Cette conversion peut être faite automatiquement par le système comme le montre l'exemple suivant :

modèle expr est struct(opérateur,opd,opg)

avec

modèle opd est synonyme(expr)

modèle opg est synonyme(expr)

la fonction de construction associée au modèle "expr" :

ctr_expr(opérateur O , opd D , opg G)

acceptera comme paramètre pour D soit un arbre de modèle "opd", soit un arbre de modèle "expr". Elle réagira de façon similaire pour le paramètre G.

Il peut apparaître pour des raisons multiples la nécessité de disposer d'opération de conversion explicite. Associées aux constructeur "synonyme" nous disposerons des deux fonctions suivantes :

dns_xxx(yyy P) -> xxx

hrs_xxx(xxx P) -> yyy

qui sont respectivement la conversion "dans" et "hors" du type "xxx" qui est synonyme de "yyy".

3.3.2.4.7 Création de valeur neutre

Lors de la création de l'arborescence, il se peut qu'un certain nombre de sous-arbres ne soient pas constructibles, soit parce qu'on ne possède pas l'information nécessaire à son élaboration, soit parce qu'on a décidé de remettre ce calcul à un moment ultérieur pour simplifier le programme. Dans ces cas il est quand même nécessaire de fournir une valeur pour ce sous-arbre. On délivre alors une valeur particulière d'adresse, que l'on appelle valeur neutre, qui indique que le sous-arbre est absent. Pour chaque modèle on dispose de la primitive suivante :

ctr_neutre_xxx -> xxx

ou "xxx" est le nom du modèle concerné.

La valeur neutre sert aussi à la construction de structure d'information finie décrite à l'aide d'une modélisation infinie (voir @3.3.2.3.3 et @3.3.2.3.4 sur les listes et les ensembles).

Remarque :

Ces primitives ne créent pas de structure d'arbre, mais délivre une valeur particulière qui peut servir dans la construction d'une structure arborescente.

3.3.2.4.8 Généralisation des primitives

Cette manière de procéder, qui constiste à fournir une "batterie" de primitives de création est fort agréable pour l'utilisateur. Elle est par contre coûteuse en taille des programmes nécessaires au fonctionnement du système, et ceci proportionnellement à la taille des modélisations que l'on veut gérer. Ceci devient vraiment rédhibitoire pour des modélisations de taille importante.

On remarque les similitudes qui existent dans des groupes de primitives. Il est donc souhaitable et possible de généraliser ces groupes de primitives en créant une primitive par groupe, celle-ci devant être paramétrée avec les informations nécessaires pour conserver la même fonctionnalité que précédemment.

Pour les modèles "structure" et "sélection" on crée une primitive de création qui possède en paramètre le modèle à créer, la description simple à créer dans le cas d'une sélection, suivi de la liste de paramètres nécessaires au modèle concerné.

Pour les modèles "liste" et "ensemble", on a déjà un certain nombre de primitives génériques. Il suffit de transformer les primitives d'initialisations en paramétrant par le modèle à créer.

Pour la création de valeur neutre, on fait de même en définissant une primitive unique ayant en paramètre le modèle pour lequel on veut produire la valeur neutre.

Pour les fonctions de conversion associées au constructeur "synonyme", il faut préciser le modèle synonyme concerné. Le système est alors en mesure de retrouver le modèle initial à utiliser.

Les autres primitives que nous avons présentées précédemment ne sont pas modifiées. Cette solution a l'avantage de fournir un ensemble de programmes, qui est indépendant de la modélisation que l'on réalise. La différence est faite sur l'information nécessaire à ces programmes pour qu'ils soient adaptés à une modélisation particulière.

ctr_nom(code_lex cl) -> nom

ctr_entier(integer e) -> entier

ctr_logique(booleen b) -> logique

ctr_modèle(modèle xxx,sélect yyy, ...) ->xxx

ctr_ens(modèle xxx) ->ens_xxx

ins_ens(ens_xxx E, nom N, xxx X) ->logique

ctr_lst(modèle xxx) ->lst_xxx

int_lst(lst_xxx L, xxx E)

inq_lst(lst_xxx L, xxx E)

cnv_dns(modèle xxx, yyy E) ->xxx

cnv_hrs(modèle yyy, xxx E) ->yyy

ctr_ntr(modèle xxx) ->xxx

La généralisation des primitives nécessite de paramétrer avec de nouvelles informations ("modèle" et "sélect").

Nous choisissons de définir ces nouveaux types d'information de la façon suivante :

modèle modèle est synonyme(code)

modèle sélect est synonyme(terminal)
ou modèle sélect est synonyme (code)

Pour une plus grande souplesse d'utilisation, on se sert de valeurs symboliques associées au type "code".

exemple :

ctr_modèle(dcl_proc, c_action, ...)

va construire un arbre du modèle "dcl_proc" en utilisant la sélection ayant comme discriminant "c_action".

3.3.2.5 Exemple de modélisation

Il s'agit de l'exemple de description d'un nombre binaire. Dans un premier temps, nous montrons la forme externe fournit au système, elle est suivie d'une décompilation de la modèlisation.

```
$1 liste des modèles élémentaires et des discriminants

ent
frc
one
zéro

$2 liste des modèles décrivant la structure

nombre_binaire ent partie_entière,
               frc partie_entière partie_décimale*

bit         one,
            zéro*

partie_entière synonyme(liste(bit))
partie_décimale synonyme(liste(bit))

$3 fin de la description
```

```
                Liste des modeles élémentaires      PAGE 001

  REF    NOM    (SYNONYME DU MODELE)

    1  lst
    2  vid
    3  lst
    4  ent
    5  frc
    6  one
    7  zéro
```

Liste des modèles non élémentaires PAGE 002

REF NOM DESCRIPTION

8 nombre_binaire

1 ent :
"ent" partie_entière

2 frc :
"frc" partie_entière partie_décimale

9 bit

1 one :
"one"

2 zero :
"zéro"

10 lst_bit

1 lst :
"lst" adr(dsc_*lst_bit) elt_lst_bit

11 elt_lst_bit

1 vid :
"vid"

2 elt :
"elt" bit elt_lst_bit

```
12  partie_entière synonyme(lst_bit)
```

```
13  partie_décimale synonyme(lst_bit)
```

Références croisées PAGE 003

DCL	NOM	TYPE	UTIL	
9	bit	base	11(2)	
1	elt	élémentaire prédéfini	11(2)	
4	ent	élémentaire	8(1)	
11	elt_lst_bit	base	10(1)	11(2)
5	frc	élémentaire	8(2)	
3	lst	élémentaire prédéfini	10(1)	
10	lst_bit	base		
6	one	élémentaire	9(1)	
13	partie_décimale	base synonyme(lst_bit)	8(2)	
12	partie_entière	base synonyme(lst_bit)	8(1)	8(2)
2	vid	élémentaire prédéfini	11(1)	
8	val_binaire	base axiome		
7	zéro	élémentaire	9(2)	

3.3.3 Programmation d'une modélisation

Dans ce chapitre, nous montrons comment on peut décrire la réalisation d'un outil. Dans un premier paragraphe nous établissons la relation qui existe entre notre modélisation et les techniques associées aux grammaires de syntaxe abstraite. Nous enchaînons ensuite en établissant la notion de contexte et d'attribut. Nous terminons en montrant comment on peut envisager une programmation modulaire dans un tel système.

3.3.3.1 La notion d'interprétation

Pour réaliser un programme de traitement d'une information vérifiant la définition qui en est faite dans la modélisation, nous établisons une interprétation de cette information. Le résulat de cette interprétation est le résultat désiré du programme à écrire.
Pour interpréter un objet, nous devons disposer, afin d'exprimer simplement l'interprétation, d'un moyen d'évaluation d'une forme interne. Cette évaluation est définie par une expression dont les opérandes sont les informations de l'objet considéré, et dont les opérateurs sont définis par l'utilisateur lui-même.

Le système offre, pour réaliser cette évaluation, deux moyens complémentaires.

D'une part un "parcours" de l'objet à évaluer issu de la modélisation de cet objet.

D'autre part la possibilité pour l'utilisateur d'insérer des appels à des actions qu'il a définies.
Ces actions décrivent les manipulations effectuées par un outil et sont définies en terme de composition des opérations (création, consultation) présentées précédemment.

Le système prendra en charge la gestion des appels aux actions externes.

Une évaluation d'un objet revient donc à parcourir l'arborescence associée. Le résultat de cette évaluation peut être formalisé comme un "but" qui est remonté au niveau de la racine de l'objet à traiter. La constitution de ce "but" est faite tout au long du parcours de l'arborescence.

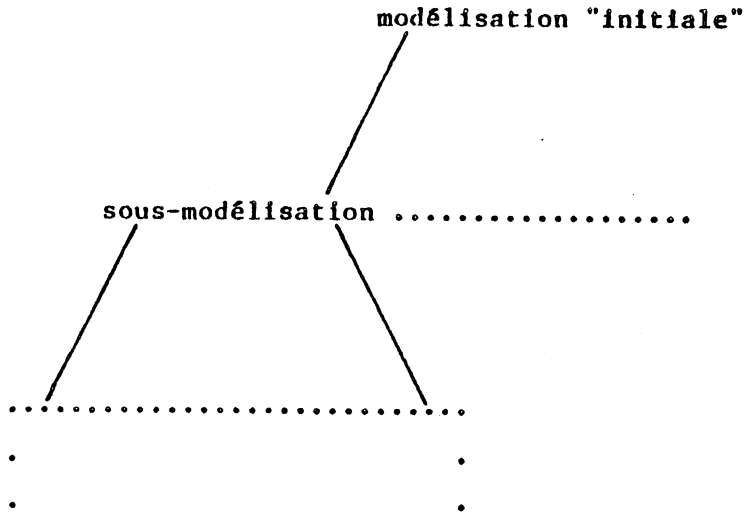
En fonction de l'outil que l'on veut réaliser, les informations constituant l'objet seront plus ou moins pertinentes. Dans ces conditions, l'utilisateur doit pouvoir préciser effectivement les informations qu'il désire parcourir et celles qu'il ne désire pas considérer pour cet outil particulier. Nous sommes amenés à prendre en considération deux classes de modélisations.

D'une part celles que nous avons présentées jusqu'ici. Elles permettent de décrire l'ensemble des domaines (types) connus du système pour une modélisation particulière. Nous parlons alors de modélisation "initiale", toutes les autres en seront issues.

D'autre part les sous-modélisations, dans lesquelles un certain nombre d'informations peuvent être cachées, et où sont insérées des actions réalisant l'outil. Il s'agit ici du résultat d'une étape de programmation dans notre système

De cette manière, on obtient une structure hiérachique où toute modélisation est issue d'une autre.

L'avantage de cette solution réside dans le fait que tout élément de la hiérachie est structurellement correct, de par la façon dont elle est constituée. On n'effectue la vérification qu'une seule fois sur la modélisation "initiale".



De façon pratique, on considère trois types de modélisations dans cette structure hiérachique.

La modélisation "initiale" qui est la racine de la structure.

Les modélisations non terminales dans la structure, qui sont la restriction à un ensemble de modèles, de la modélisation dont elles sont issues.

Les modélisations terminales dans la structure, qui sont les programmations effectives. Ce sont ces modélisations qui seront "exécutées" lors de l'évaluation d'un "but".

Nous verrons dans le chapitre concernant les outils de manipulation des modélisations comment créer ces nouvelles sous-modélisations.

3.3.3.1.1 La consultation de valeur

Les actions que l'on insère dans la modélisation sont dépendantes du contexte où elles se trouvent. Pour que ce soit le cas, elles doivent être paramétrables. Pour ce faire, on doit pouvoir accéder à l'information contenue dans l'objet.

Nous distinguons deux classes de consultation d'une valeur d'un objet. Les consultations liées au parcours et les consultations indépendantes de ce parcours.

- Lors d'un parcours, on peut utiliser le nom des modèles comme fonction de consultation des valeurs qui y sont associées. Par exemple dans le modèle suivant :

  modèle ttque est struct(condition,instruction)

  les noms "condition" et "instruction" peuvent être utilisés comme fonction de consultation des valeurs qu'ils définissent lors du parcours du modèle "ttque".

  Les consultations indépendantes du parcours sont relatives à des constructeurs particuliers. Pour le constructeur "liste" on dispose des primitives de consultation :

  ```
  fonction elt_lst(lst_xxx L, entier I) -> xxx
  fonction lst_vid(lst_xxx L) -> logique
  .
  ```

  Pour le constructeur "ensemble" on dispose des primitives de consultation :

  ```
  fonction rch_ens(ens_xxx E, nom N) -> xxx
  fonction ens_vid(ens_xxx E) -> logique
  .
  ```

La fonction "elt_lst" fournit le ième élément de la liste en renvoyant l'accès au sous-arbre considéré, ou la valeur neutre du modèle correspondant dans le cas où cet élément n'existe pas.

La fonction "rch_ens" fournit l'accès à l'arbre associé au nom si celui-ci existe dans l'ensemble, et en cas d'échec elle renvoie la valeur neutre du modèle correspondant.

Dans le paragraphe suivant nous élargirons les fonctions de consultation en introduisant de nouvelles valeurs consultables.

3.3.3.2 Parallèle avec les grammaires

Les grammaires hors-contexte ont été introduites pour décrire la syntaxe des langages. Elles permettent de définir de manière simple et lisible la structure d'informations complexes. Lorsqu'elle est bien choisie, une grammaire peut être utile pour décrire la sémantique du langage. Ceci se fait par diverses techniques qui permettent d'enrichir la définition hors-contexte, en lui adjoignant des calculs relatifs à la sémantique.

Une grammaire hors-contexte est définie par :

un ensemble fini de symboles terminaux (Vt)
un ensemble fini de symboles non terminaux (Vn)
un ensemble fini de productions (P) de la forme :

$$u \rightarrow v \text{ ou } u \in Vn \text{ et } v \in (Vn \cup Vt)^*$$

On dit que "u" dérive vers "v". "u" est un symbole non terminal et "v" est une chaine de symboles quelconques de l'alphabet.

on définit "s" comme étant le symbole non terminal de départ. On l'appelle "axiome". L'axiome ne peut pas réapparaitre en partie droite d'une règle de production.

On distingue plusieurs classes de grammaires hors-contextes et d'analyses de ces grammaires.

Les grammaires "propres" qui ne possèdent ni règles improductives, ni règles inaccessibles, ni symboles inutiles.

Les grammaires non "ambiguës" qui ne permettent qu'une seule dérivation possible.

Les analyses "déterministes" qui permettent de décider de la substitution à réaliser avec un contexte maximum déterminé.

Les analyses "descendantes" {ll(k)} où la dérivation se fait par remplacement de la partie gauche d'une règle par sa partie droite.

Les analyses "ascendantes" {lr(k)} où la substitution se fait par réduction d'une partie droite d'une règle à sa partie gauche.

3.3.3.2.1 Les grammaires à attributs

Les grammaires à attributs ont été introduites en compilation des langages pour permettre d'exprimer la sémantique (sous-contexte) à partir d'une grammaire hors-contexte [Lor 74] [Rai 79] [Sim 81].

Les systèmes d'attributs permettent de définir la signification des programmes qui sont décrits par une grammaire hors-contexte. Les attributs sont des valeurs attachées aux symboles non terminaux de l'alphabet. Ils sont définis par des règles d'évaluation associées à chaque règle de production où ils apparaissent. On distingue deux sortes d'attributs :

les attributs "**hérités**" correspondent aux valeurs qui vont descendre dans l'arbre. Ce sont les attributs calculés pour la partie droite d'une règle de production.

les attributs "**synthétisés**" qui eux, à l'inverse des précédents, remontent dans l'arbre : ce sont les attributs dont la valeur ne dépend que des attributs attachés en partie droite d'une règle de production. Ce sont ceux calculés en partie gauche de la règle de production.

C'est dans chaque règle de production que l'on définit comment est évalué l'attribut. Ceci a l'avantage de cerner très précisemment l'endroit et la portée d'une modification de l'évaluation d'un attribut. Il se peut que l'évaluation d'un attribut demande plusieurs balayages sur l'information. Dans un certain nombre de cas l'évaluation des attributs peut se faire en un seul balayage de la gauche vers la droite.

Remarque :

On peut d'une certaine manière, attacher des attributs aux symboles terminaux de la grammaire. Il s'agit pour les attributs synthétisés, de manipuler de façon simple les propriétés attachées à ces terminaux. Pour les attributs hérités, il s'agit de définir le contexte permettant de calculer les propriétés à synthétiser pour un symbole terminal.

Ce système de grammaire attribuée permet de sélectionner un sous-langage du langage défini par la grammaire hors-contexte. Cette technique est surtout utilisée pour rendre compte de contraintes contextuelles que l'on veut assurer [BoR 80] [KaZ 81] [KMP 79] [Wat 79].

3.3.3.2.2 Similitude avec notre système

Nous pouvons établir un parallèle important entre notre modélisation et un tel système. D'une part, notre modélisation initiale correspond totalement à la structure de grammaire hors-contexte. Les modèles élémentaires correspondent aux symboles terminaux et les modèles non élémentaires correspondent aux symboles non terminaux de la grammaire. Les règles sur l'axiome sont relaxées car on peut admettre dans notre système qu'il apparaisse dans la définition d'un autre modèle.

Les modèles non élémentaires sont bien des productions qui associent à un nom un sous-arbre particulier. Le système étant purement structurel, il s'associe bien à une définition hors-contexte.

Dans notre système, les attributs seront représentés par des variables dont la valeur représentera l'attribut considéré. Les attributs hérités seront affectés lors d'une descente dans un sous-arbre, les attributs synthétisés seront produits lorsque l'on s'échappera d'un sous-arbre.

3.3.3.3 La notion de contexte

Comme pour les grammaires à attributs, les modèles forment un contexte fermé où sont évalués les attributs. Cette notion implicite de contexte peut être renforcée pour donner effectivemment un outil puissant de décomposition et de programmation.

On associe à chaque modèle un contexte qui est représenté par l'ensemble des objets accessibles.

Nous avons précédemment vu que pour la consultation de l'information modélisée, un moyen simple et précis était d'utiliser comme désignation le nom du modèle caractérisant cette information. Une partie du contexte d'un modèle sera représentée par les noms de modèles non élémentaires apparaissant dans sa définition. Il en sera de même pour les modèles élémentaires "nom", "entier", "logique" et "code" s'ils apparaissent dans la définition du modèle.

Pour représenter les attributs il faut mettre en place un système de paramètres. Ceci se fait facilemment car le modèle peut être considéré comme une action nommée dont la mission est d'évaluer ces attributs. Elle comporte alors des paramètres d'"entrée" qui représentent les attributs hérités et des paramètres de "sortie" qui sont les attributs synthétisés. Nous utiliserons désormais le terme "attributs" à la place de "paramètres" et de façon générale à la place de "variables".

Nous ajoutons à cette définition du contexte d'un modèle un ensemble d'attributs locaux servant dans les élaborations intermédiaires. Ils sont définis par une déclaration au même titre que les attributs hérités.

Nous définissons enfin, un attribut particulier qui représente le sous-arbre modélisé.

Les modèles élémentaires qui n'ont pas été cités forment un contexte global à la modélisation toute entière. Ce contexte est constitué des constantes manipulées par la modélisation.

Remarquons que tous les attributs sont typés. Le type d'un attribut est défini par le modèle d'arbre qu'il peut contenir. Ce typage servira à établir des vérifications sur l'utilisation des attributs dans la programmation.

En résumé :

on dispose d'un contexte global, et d'un contexte local associé à chaque modèle contenant :

les attributs hérités
les attributs locaux
les valeurs composant le modèle
l'attribut définissant le modèle globalemment

Exemple :

soit le modèle "ttque" auquel on associe les attributs hérités h1,...,hn et les attributs locaux l1,...,lm. On a la structure arborescente suivante :

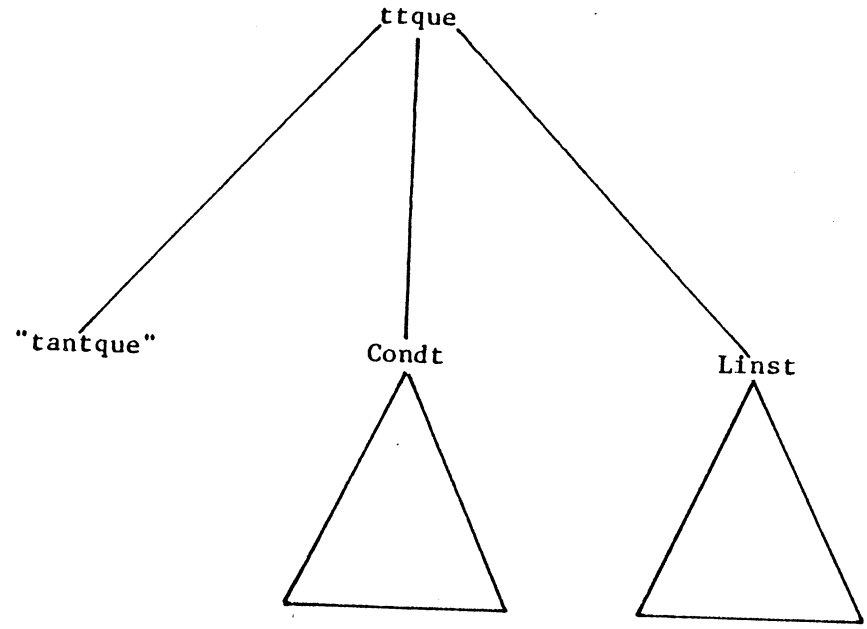
modèle ttque est struct("tantque",Condt,Linst)

Contexte local :

h1,...,hn
l1,...,lm
Cond
Linst
$racine (accès au noeud ttque)

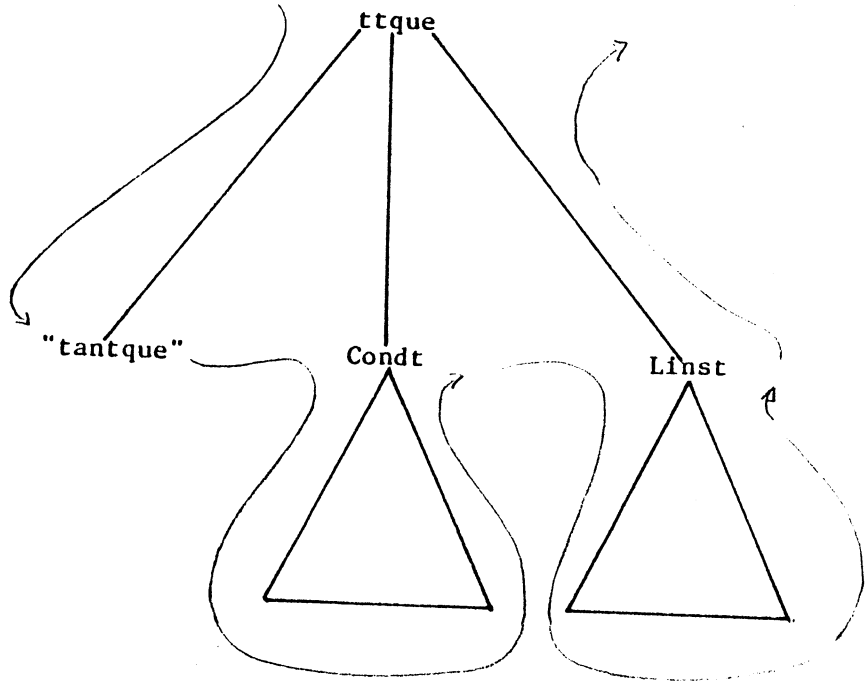
Contexte global :

.
.
.
tantque
.
.

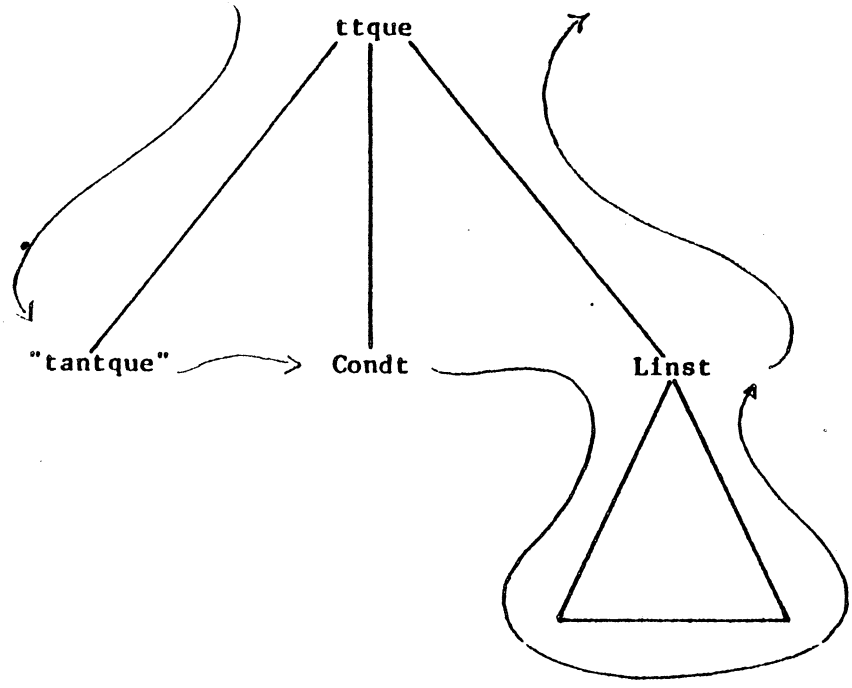


3.3.3.4 Parcours de l'arbre

Le système offre un parcours postfixé de l'arborescence n-aire, comme le représente le schéma sur l'exemple du modèle "ttque" :



Nous avons déjà évoqué, dans un paragraphe précédent, le fait qu'un sous-arbre peut être occulté, car l'information qu'il contient n'est pas pertinente pour le problème concerné. Cette occultation revient à inhiber le parcours pour le sous-arbre correspondant. Dans l'exemple du dessus si on occulte le sous-arbre "Condt" on obtient un nouveau parcours qui peut être schématisé comme suit :



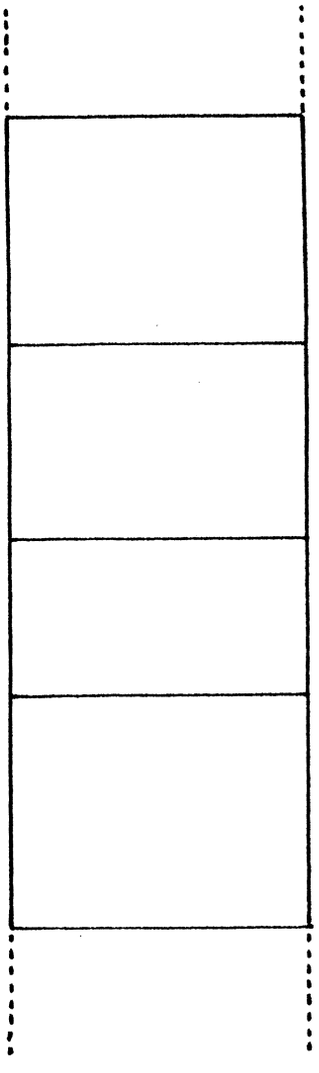
Dans la description des "listes" et des "ensembles" que nous avons détaillée aux paragraphes 3.3.2.3.3 et 3.3.2.3.4, on engendre trois modèles dont un est un descripteur de la "liste" (resp. "ensemble"). Ce modèle est systématiquement occulté par le système sans que l'utilisateur puisse opérer l'opération inverse, afin de protéger les informations caractérisant la liste (resp. ensemble) qui y sont contenues.

Pendant le parcours de l'arbre, aucune vérification syntaxique n'est nécessaire. La création de la représentation interne d'un objet "programme" assure cette correction.

3.3.3.4.1 Gestion des attributs

Comme nous l'avons remarqué les attributs se comportent comme des objets dans un programme. Il y a les variables locales et les paramètres données et résultats. Le contexte d'un modèle est empilé à chaque fois que l'on descend dans un sous-arbre de ce modèle. La pile se compose de la manière suivante :

Un attribut est un couple <type,valeur>. Le type est un des modèles définis dans la modélisation. La valeur est la valeur, dans le type considéré, de l'attribut.
Il n'est pas nécessaire de conserver dynamiquemment le type de l'attribut, car les vérifications sémantiques peuvent être faites au moment de la compilation de la modélisation. Il n'y a que pour les actions externes que cette vérification de compatibilité ne peut être faite statiquemment.



zone des attributs locaux. Ce sont eux qui sont modifiés par les attributs synthétisés d'un sous-arbre

zone des attributs synthétisés. Ce sont les adresses des attributs à modifier qui sont données ici

zone des attributs hérités. Ce sont les valeurs qui sont rangées ici

zone système contenant des informations pour la gestion de la la pile,et l'adresse de retour

Remarque :

Pour les opérations de création d'une structure arborescente le type des sous-arbres est aussi nécessaire. Il permet d'assurer une construction cohérente avec la définition faite dans la modèlisation.

Lorsque l'on appelle un modèle, c'est-à-dire que l'on se prépare à descendre dans un sous-arbre, on met en place dans la pile des contextes les informations systèmes, on range les valeurs des attributs hérités par ce sous-arbre, et on range les adresses des attributs locaux devant recevoir les valeurs des attributs synthétisés par ce sous-arbre.
Quand on descend effectivemment dans le sous-arbre, on met à jour la pile des contextes en réservant la place nécessaire aux attributs locaux de ce sous-arbre.

Les attributs sont évalués de la gauche vers la droite, en fonction du parcours de l'arbre. Tout attribut hérité doit posséder une valeur. Le système vérifie cette condition lors de la compilation du modèle.
Les attributs hérités par le modèle considéré sont valués par définition. Les attributs locaux du modèle considéré sont valués après avoir fait l'objet d'une synthétisation.

Exemple :

ttque est "tantque" Condt(typ) verif_typ(typ) Linst

"typ" est un attribut local ayant pour "type" un modèle qui décrit tous les types du langage (entre autres le type logique de la notation modélisée).

Dans cet exemple on vérifie grâce à l'action externe "verif_typ", que "Condt" (expression conditionnelle) a bien le type logique.

3.3.3.5 La modularité de la programmation

L'intérêt de modulariser un programme n'est plus à démontrer. Nous avons montré dans la première partie les raisons et les moyens que nous nous donnions au niveau d'une notation algorithmique. Nous appliquons ici une démarche similaire.

Il existe déjà un premier niveau de modularité dans le système. Chaque modèle est considéré comme indépendant des autres vis-à-vis de la programmation. Mais cette modularité repose sur un découpage structurel qui constitue des entités parfois trop petites pour que l'on puisse véritablement parler de modularité.

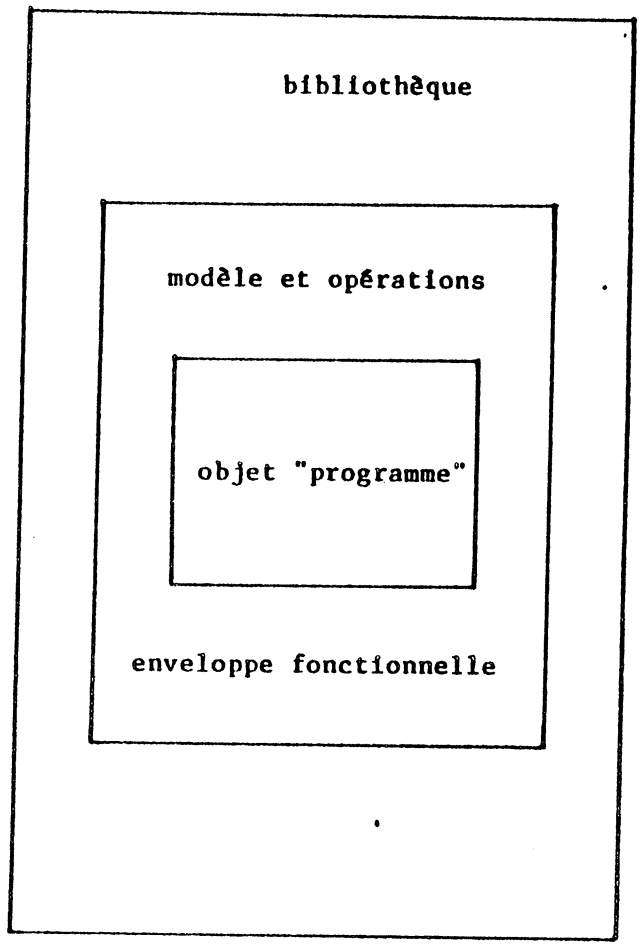
Nous devons disposer d'un découpage qui permette de définir une modularité que l'on qualifie de fonctionnelle. C'est-à-dire que l'on dispose d'entités permettant de construire des fonctions associées.

Nous voulons disposer d'un mécanisme qui nous permet de définir, d'une part des sous-structures, et de programmer des fonctions sur ces sous-structures.

La sous-modélisation est le moyen de construire une fonction de la sous-structure qu'elle modélise.
Dans un premier temps on restreint la modélisation à l'ensemble des modèles que l'on veut traiter. Cette opération se fait en choisissant un "axiome" pour la nouvelle sous-modélisation, et en occultant les sous-arbres inutiles. Dans un deuxième temps on peut programmer une fonction associée à cette sous-modélisation en insérant des attributs et des calculs sur ces attributs.

Cette façon de procéder nous permet de reconsidérer la structure générale du système telle que nous l'avions décrite dans l'introduction (@3.2.2). Nous faisons apparaitre une couche d'outils intermédiaires entre le domaine d'application et la enveloppe fonctionnelle que nous venons de décrire en partie. Cette couche qui peut évoluer à tout instant, correspond à une bibliothèque de primitives (d'un niveau d'abstraction plus élévé que celui du noyau du système).

domaine des outils



3.3.3.5.1 Mise en oeuvre des sous-modélisations

Les sous-modélisations apparaissent comme des actions externes que l'on peut insérer dans la modélisation. A une sous-modélisation on associe un sous-arbre sur lequel est effectué le calcul du "but" de cette sous-modélisation.

Nous avons alors un système qui permet d'étendre les possibilités expressives d'une modélisation par l'introduction d'appel à d'autres modélisations. On augmente la capacité de calcul associé à un point de l'arbre en insérant des sous-arbres virtuels en ce point. On peut également résoudre si besoin est le problème de l'évaluation d'un attribut, si celui-ci ne peut être calculé en un seul passage sur un sous-arbre.

Cette technique permet de décrire les programmes à écrire en fonction de l'aspect logique du problème et de façon presque indépendante de l'aspect physique de l'information associée.

3.3.3.6 Gestion des appels externes

Les actions externes sont les opérateurs qui servent à évaluer le "but" synthétisé au niveau de la racine de l'arbre de l'objet "programme". Ces actions sont paramétrées par des attributs accessibles dans les contextes de leurs utilisations. Elles fournissent en résultat zéro, un ou plusieurs attributs.

On voit de cette façon que les actions externes ont un comportement similaire aux modèles. On les active avec des attributs hérités et elles restituent des attributs synthétisés.

Pour cette raison le système à l'exécution fonctionne de manière identique. A l'appel un contexte est empilé contenant les attributs hérités, les adresses des attributs à synthétiser et comme attributs locaux un ensemble correspondant aux attributs synthétisés. Les paramètres effectifs de l'action externe sont ceux qui se trouvent au sommet de la pile de contexte. De cette manière on simule un sous-arbre en lieu et place de l'appel à l'action externe.

3.3.3.7 Le langage d'exécution

Pour réaliser le parcours de l'arbre, et faire la gestion des attributs et des appels aux actions externes, nous nous dotons d'un petit langage d'exécution. Lors de la compilation d'une modélisation, on peut demander la production du code exécutable. Ce code exécutable est décrit dans ce langage pour lequel on dispose d'un interprèteur.

Le langage se compose d'un jeu d'instructions que nous divisons en quatres classes.

La première classe comporte les instructions de parcours de l'arborescence.

init_modèle_simple : (init_s)

Instruction débutant tout parcours d'un modèle qui n'est pas une sélection.

init_modèle_multiple : (init_m)

Instruction débutant tout parcours d'un modèle qui est une sélection. Cette instruction a un format variable puisqu'elle comporte l'ensemble des cas de la sélection avec, associée à chaque cas, l'adresse d'implantation du code correspondant.

appel_modèle : (go_mod)

Instruction de branchemment, avec retour au point d'appel, au sous-arbre dont la racine est le point courant.

résume_modèle : (resume)

Instruction de retour au point d'appel lorsque l'on a fini d'explorer un sous-arbre.

nop : (no_ope)

Instruction n'ayant comme effet que le parcours de l'élément courant.

La deuxième classe comporte les instructions de test de la structure parcourue.

test_code : (test_c)

Instruction vérifiant la valeur du noeud terminal courant.

test_adr : (test_a)

Instruction vérifiant que l'on se trouve en présence d'un noeud non terminal.

La troisième classe comporte l'instruction d'appel aux actions externes.

appel-ext : (go_ext)

Instruction réalisant l'appel à une action externe au système.

La quatrième classe comporte les instructions de gestion des attributs.

set_attr_h : (set_ah)

Instruction qui empile la valeur d'un attribut hérité dans la pile des contextes.

set_attr_s : (set_as)

Instruction qui empile l'adresse de la variable associée à un attribut synthétisé dans la pile des contextes.

put_attr_s : (put_as)

Instruction qui met à jour la valeur de la variable associée à un attribut synthétisé.

3.3.3.7.1 Exemple de code produit

Il s'agit ici du programme qui assure le parcours de la structure arborescente (sans executer de calculs particuliers).

```
MODELE                @  INST      ARGS    SYMBOLE

nombre_binaire         1 init_m    0  2
                                      4   14 ent
                                      5   18 frc
                      14 go_mod          25 lst_bit
                      17 resume
                      18 go_mod          25 lst_bit
                      21 go_mod          25 lst_bit
                      24 resume

lst_bit               25 init_s    0
                      28 test_c    3          lst
                      30 test_a
                      31 go_mod          35 elt_lst_bit
                      34 resume

elt_lst_bit           35 init_m    0  2
                                      2   48 vid
                                      3   49 elt
                      48 resume
                      49 go_mod          56 bit
                      52 go_mod          35 elt_lst_bit
                      55 resume

bit                   56 init_m    0  2
                                      6   69 one
                                      7   70 zero
                      69 resume
                      70 resume
```

3.3.3.8 Exemple de programmation d'une modélisation

Nous reprenons l'exemple de description d'un nombre binaire.

Pour cet exemple nous donnons deux réalisations possibles. La première est constituée de quatres modules séparés. elle correspond à une analyse où l'on a séparé le problème initial en deux sous-problèmes. D'une part calculer la partie entière, d'autre part calculer la partie décimale. Ces deux sous-problèmes utilisent une même fonction qui transforme un chiffre binaire en valeur entière.

Dans la deuxième solution nous reprenons la manière de faire utilisée dans le paragraphe décrivant le système **Delta** (@3.2.2). Il s'agit de généraliser le problème, afin d'obtenir une fonction unique pour le calcul de la partie entière et de la partie décimale.

Dans l'exemple suivant les actions externes sont réalisées par des modélisations.L'appel à ces dites fonctions externes comporte comme premier argument le sous-arbre concerné par le calcul, les arguments suivants sont les attributs associés à la fonction.

La première solution est constituée des fonctions "val_décimale", "eval_pe", "eval_pd" et "eval_bit". La deuxième solutions est composée de "val_décimale_bis" et de "longueur_pe".

fonction "val binaire"

```
8  val_décimale : ↑entier,entier V1,V2

1 ent :↑V1
    "ent" adr(partie_entière)
          eval_pe partie_entière↑V1

2 frc :↑(V1+V2)
    "frc" adr(partie_entière) adr(partie_décimale)
          eval_pe partie_entière↑V1
          eval-pd partie_décimale↑V2
```

Cette définition fait appel à d'autres fonctions de calcul, qui sont "eval_pe" et "eval_pd". Elle sont définies dans la partie externe de "val_décimale".

Nous détaillons maintenant la définition de la fonction "eval_pe". Cette fonction calcule la valeur décimale de la partie entière du nombre que l'on traite.

fonction "eval_pe"

```
12  partie_entière synonyme(lst_bit)

10  lst_bit :↑entier, entier V,L

    lst :↑V
      "lst" adr(dsc_*lst_bit) elt_lst_bit↑V↑L
```

```
11  elt_lst_bit :↑entier,↑entier, entier V,V1,L
-----------------------------------------------
 1 vid :↑0↑0
     "vid"

 2 elt :↑(V+V1)↑(L+1)
     "elt" adr(bit) elt_lst_bit↑V↑L
           eval_bit bit↓L↑V1
```

La fonction que nous élaborons maintenant calcule la valeur décimale de la partie décimale du nombre à traiter.

fonction "eval_pd"

```
13  partie_décimale synonyme(lst_bit)
```

```
10  lst_bit :↑entier, entier V
-------------------------------
    lst :↑V
      "lst" adr(dsc_*lst_bit) elt_lst_bit↓(-1)↑V
```

```
11  elt_lst_bit :↓entier P,↑entier, entier V
---------------------------------------------
 1 vid :↑0
     "vid"

 2 elt :↑(V+V1)
     "elt" adr(bit) eval_bit bit↓P↑V
                    elt_lst_bit↓(P-1)↑V1
```

Les deux fonctions que nous avons décrites précédemment utilisent elles-mêmes une fonction de calcul. Cette fonction délivre la valeur entière associée à un chiffre binaire placé dans le nombre.

fonction "eval bit"

```
9  bit :↓entier PUIS,↑entier
```

```
1 one :↑(2↑PUIS)
    "one"

2 zero :↑0
    "zéro"
```

fonction "val binaire bis"

```
8  val_décimale_bis : ↑entier,entier V1,V2,L
```

```
1 ent :↑V1
    "ent" longueur_pe↑L partie_entière↓(L-1)↑V1

2 frc :↑(V1+V2)
    "frc" longueur_pe↑L partie_entière↓(L-1)↑V1
          partie_décimale↓(-1)↑V2
```

```
12  partie_entière synonyme(1st_bit)
```

```
13  partie_décimale synonyme(1st_bit)
```

```
10  lst_bit :↓entier P,↑entier, entier V
```

```
    lst :↑V
      "lst" adr(dsc_*lst_bit) elt_lst_bit↓P↑V
```

```
11  elt_lst_bit :↓entier P,↑entier, entier V,V1
```

```
 1 vid :↑0
     "vid"

 2 elt :↑(V+V1)
     "elt" bit↓P↑V elt_lst_bit↓(P-1)↑V
```

```
 9  bit :↓entier PUIS,↑entier
```

```
 1 one :↑(2↑PUIS)
     "one"

 2 zero :↑0
     "zéro"
```

fonction "longueur pe"

```
12  partie_entière synonyme(lst_bit)
```

```
10  lst_bit :↑entier, entier L
```

```
    lst :↑L
      "lst" adr(dsc_*lst_bit) elt_lst_bit↑L
```

```
11  elt_lst_bit :↑entier, entier L
```

```
 1 vid :↑0
     "vid"

 2 elt :↑(L+1)
     "elt" adr(bit) elt_lst_bit↑L
```

3.3.4 Outils pour manipuler les modélisations

Ces outils doivent permettre de manipuler des modélisations pour les créer, les mettre à jour, les modifier,...etc.
Nous voulons que ces outils soient très interactifs, car nous le verrons dans le paragraphe suivant, la production de programme se fera à l'aide de la modélisation.

3.3.4.1 Représentation externe

Le problème de l'acquisition d'une modélisation doit être traitée au niveau des moyens de communication dont on dispose.

Au niveau interactif, un programme se charge de maintenir une image externe toujours cohérente de la représentation interne de la modélisation, dans une fenêtre appropiée. On peut alors ajouter, modifier ou détruire des parties de cette modélisation.

Au niveau "off-line", on crée une donnée respectant la syntaxe de la représentation externe. Cette donnée est compilée pour créer une représentation interne de la modélisation.

La représentation externe doit être la plus simple possible. elle se décompose de la manière suivante :

$1

ensemble des modèles élémentaires définis par l'utilisateur

$2

ensemble des modèles non élémentaires

$3

Un modèle élémentaire est un nom, ou un nom suivi de "synonyme(...)", les modèles élémentaires sont séparés par au moins un blanc ou par un retour à la ligne. Un modèle "structure" est composé d'un nom suivi de la liste des noms de modèle qui la compose, le tout terminé par une étoile (*). Un modèle "sélection" est composé d'un nom suivi de la liste des descriptions simples séparées par des virgules (,), le tout étant terminé par une étoile (*). Les listes et les ensembles sont constitués d'un terme spécifique suivi du nom du modèle des éléments de la liste ou de l'ensemble. L'axiome qui est le modèle racine de l'arborescence doit être le premier des modèles non élémentaires, ou précisé dans une requête.

3.3.4.2 Construction d'une modélisation

Suivant le mode d'interaction choisi elle se fera soit de façon globale par un programme de compilation soit de facon incrémentale lorsque l'on a choisi le mode interactif. Dans les deux cas, des vérifications de cohérence sont faites sur la modélisation.

Il s'agit entre autre :

de vérifier qu'il n'y a pas double déclaration d'un modèle

de vérifier qu'un modèle est utilisé dans les conditions requises par sa définition

de s'assurer que, dans une sélection, tous les cas sont bien discriminés

de vérifier que lorsqu'un ensemble de modèles forme une entité définie récursivement, cette définition décrit une information finie.

.
.
.

3.3.4.3 Le générateur des opérations

Comme nous l'avons vu précédemment, le système offre aux utilisateurs un ensemble des fonctionnalités qui lui permettent de construire des informations qui respectent la définition d'une modélisation. Ce programme de génération produit soit l'ensemble des actions primitives de constructions, soit la table nécessaire aux primitives générales de construction. Cet ensemble de primitives assure à l'utilisateur une correction syntaxique des informations construites. Elles forment une définition fonctionnelle de la structure de données utilisées pour implémenter l'arborescence.

Dans les deux cas, la construction d'une arborescence se fait de façon ascendante. On fournit à une primitive générique ou non, un ensemble de sous-arbres. On doit s'assurer que cet ensemble est conforme à celui attendu et construire, dans le cas correct, le nouveau noeud à partir de l'ensemble donné et des noeuds terminaux que l'on insère automatiquement.

Cet outil peut être décrit à l'aide du système lui-même, en effet il s'agit de parcourir tous les modèles et en fonction de leur réalisation de décrire soit un programme soit des tables.

3.3.4.4 L'editeur de modélisation

L'éditeur de modélisation permet de manipuler interactivement des modélisations. Il parcours la représentation interne et fourni au terminal une image cohérente de la modélisation.

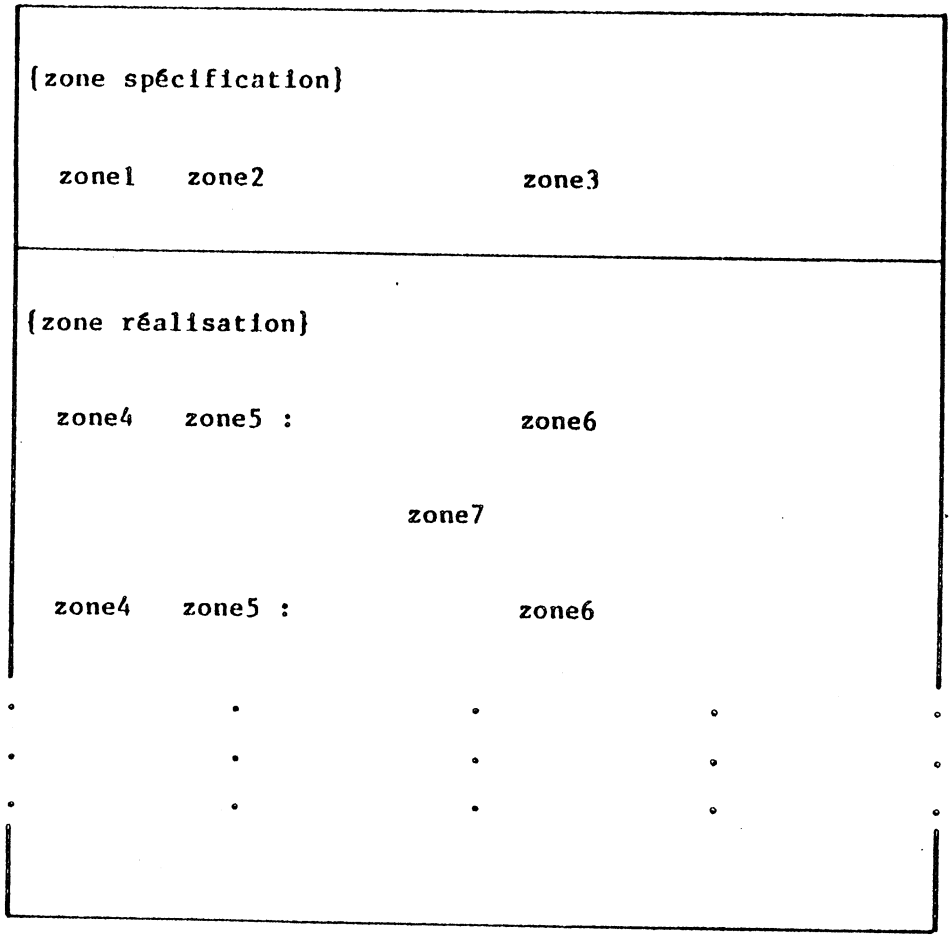
Chaque modèle élémentaire ou modèle non élémentaire synonyme est représenté comme suit :

```
{zone spécification}

 zone1    zone2    zone3
```

La zone1 contient le numéro de référence du modèle.
La zone2 contient le nom du modèle.
La zone3 contient l'argument de synonymie

Chaque modèle non élémentaire est représenté de la façon suivante :



La zone1 contient le numéro de référence du modèle.
La zone2 contient le nom du modèle.
La zone3 contient les attributs hérités et les types des attributs synthétisés.
La zone4 contient le numéro de référence de la description simple correspondante.
La zone5 contient le code discriminant pour cette description simple.
La zone6 contient les valeurs des attributs synthétisés.
La zone7 contient la description associée.

L'éditeur est un outil de base qui permet de se déplacer dans la structure arborescente indépendemment du principe de parcours associé à l'évaluation, énoncé dans le chapitre précédent. Cependant un certain nombre de fonctions utilisées par cet éditeur peuvent être programmées avec le système lui-même.

3.3.4.5 Le décompilateur

Le décompilateur fourni la forme document d'une modélisation, la présentation choisie est la même que celle utilisée pour construire de façon interactive la modélisation, ou que celle qui nous servira pour la programmation d'une modélisation. Cette condition nécéssite de paramétrer le support d'édition, afin qu'en toutes circonstances la présentation soit claire et accessible.
Elle comprend la liste des modèles élémentaires suivie de la liste des modèles non élémentaires et d'un index des modèles utilisés avec leurs références.

3.3.4.6 Stockage de la structure d'information

La structure manipulée par le système est un arbre concrêt, avec ces pointeurs et les informations de bases. Cette représentation est évidemment coûteuse en place occupée. Afin de minimiser le coût de stockage de la forme interne, on doit linéariser cet arbre. Cette opération aura pour effet d'éliminer les pointeurs qui seront recalculés au prochain chargement de la structure d'information.

La linéarisation d'un arbre (modélisation n'ayant aucun "lien") ne pose pas de problème particulier. Pour chaque noeud de la structure d'information on connait la place qu'il occcupe ainsi que le nombre et le type de ces fils. On peut donc produire une forme préfixée de l'arbre.

La reconstruction à partir de la forme linéarisée se fait de façon similaire. On connait la racine de la structure arborescente, on sait retrouver les informations associées au noeud et recréer les sous-arbres du noeud.

Par contre l'utilisation de "lien" dans la modélisation, qui sont des "faux" fils accédant un sous-arbre dont la situation, dans l'ensemble, peut être quelconque, oblige à des opérations supplémentaires. En effet avant de linéariser l'arbre, il faut calculer pour chaque sous-arbre pointé par un "lien" la situation qu'il aura lors du prochain chargement de la structure d'information. Ceci est nécessaire car aucune règle ne permet de calculer (de façon syntaxique, par rapport à la structure) cette association (lien, sous-arbre lié).

Dans ce travail, qui demande deux passages sur l'arborescence, on cherche à construire l'ensemble des triplets suivants :

{origine du lien,situation courante,situation à venir}

pour cela on construit dans une première étape la structure suivante :

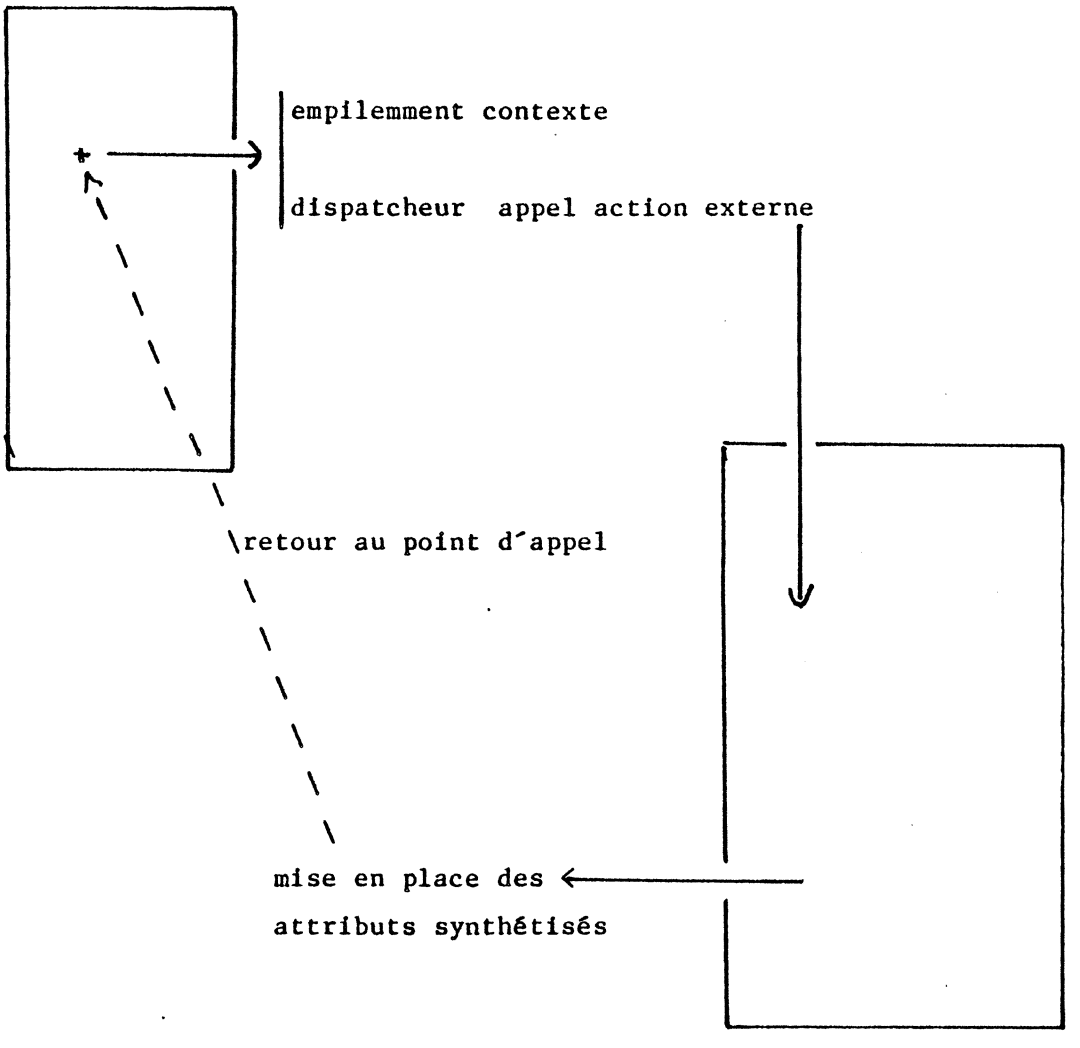
```
linear   est liste(triplet)
triplet  est struct(adr_lien,adr_lié,nouv_adr)
adr_lien est synonyme(adresse)
adr_lié  est synonyme(adresse)
nouv_adr est synonyme(adresse)
```

3.3.4.7 L'assembleur de modélisations

Dans les paragraphes "modularité de la programmation" et "gestion des appels externes", nous avons montré que l'on pouvait appeler une sous-modélisation dans le corps d'une autre. cette possibilité fort utile pour l'analyse et la programmation constitue une gène au moment de l'exécution. En effet la procédure d'appel à une action externe est longue. Hormis la mise en place du contexte qui sera de toutes les manières nécessaire, on engendre un appel à une première procédure de "dispatching" qui réalise l'appel effectif de l'action externe.

Dans le cas d'une sous-modélisation on devra créer une nouvelle pile pour le contexte, initialiser le premier contexte, et enfin on pourra activer le processus d'évaluation.

On peut schématiser ce processus comme suit :



L'assembleur de modélisation prend un ensemble de modélisations (exécutables) et réduit tous les appels aux actions externes qui sont des modélisations, à la condition que ces actions fassent partie de l'ensemble des noms donnés et soit accessibles au moment de cette opération. Cette réduction se fait à partir d'une modélisation principale en ajoutant à la fin du programme le programme translaté de la modélisation dont on réduit l'appel externe. On retrouve ici les techniques associées aux chargeurs-translateurs des langages d'assemblage.

3.4 Conclusion

Les programmes sont des informations fortement structurées, comportant un assez grand nombre de constructions différentes, en général imbriqués profondément.Il est d'autant plus important d'appliquer des techniques systématiques pour écrire des programmes manipulant de telles données. Ces techniques doivent permettre de bien restituer la structuration, Pour assurer une programmation simple et précise de telles données, les techniques doivent s'appuyer sur leur structuration.

La structure utilisée ne doit pas être celle de la syntaxe concrète de la notation qui, bien que relatant les concepts utilisés, est soumise à des règles syntaxiques pouvant apporter des facteurs inutiles ou déformants.

Nous pensons avoir, sur ce point de vue, apporté une réponse positive : La structure décrite dans le système est celle de la description abstraite de la notation ; les outils fournis permettent de structurer au mieux cette description ; l'insertion de calculs dans cette structure permet de décrire les applications avec précision et une relative aisance.

Comme nous l'avons dit dans l'introduction de cette partie, une solution souhaitable serait de disposer d'une spécification générale de la notation qui permettrait par une fonction de codage appropriée de produire une représentation concrète de cette spécification. On pourrait alors envisager de posséder un système plus ou moins automatique pour décrire les différentes représentations d'une spécification (représentation externe de la notation, représentation interne, ...). Dans cette optique on peut espérer pouvoir disposer d'un outil assurant la transformation d'une représentation à une autre.

Le système tel qu'il est décrit ici, est très statique. La modélisation ne peut évoluer que par "recompilation" de toutes ses composantes. En effet, le nombre de modèles décrits et leur composition sont figés.

La limitation imposée par l'obligation de déclarer tout modèle au niveau de la modélisation "initiale" est une contrainte qui va à l'encontre de la modularité. Une partie des informations manipulées sont temporaires et locales à un contexte particulier, ou servent à transmettre des valeurs d'un point à un autre de l'arborescence.
Cette rigidité se situe à deux niveaux, il parait difficile et peu souhaitable dans l'état des choses, de pouvoir modifier dynamiquement la composition des modèles. Il est, à l'opposé, assez aisé et souhaitable d'admettre que l'information manipulée n'est plus un "arbre" mais une "forêt", et que de ce fait on puisse définir de nouveaux modèles dans n'importe quelle modélisation.

4 Conclusion

La notation algorithmique représente les niveaux d'abstractions et les concepts que l'on veut utiliser. Elle est élaborée à partir de méthodes de programmation développées par ailleurs [Sch 78a] [Sch 79]. Nous avons validé cette notation au travers d'expériences de programmation, mais aussi dans un contexte d'enseignement avec la présentation des méthodes et l'utilisation de la notation pour décrire les solutions des problèmes abordés.

Notre démarche ne cherche pas à figer la notation dans une syntaxe particulière, ou dans un ensemble de fonctionnalités précises. Elle doit être en permanence adaptée aux différentes évolutions possibles et aux différents contextes d'utilisation. Cette évolution est dirigée principalement sur deux axes :

le premier correspond à l'évolution des langages de programmation et des environnements développés actuellement [Ade 82] [Big 82] [Bux 80] [TeR 81] ;

le second correspond à l'évolution des méthodes d'analyse utilisées. Il s'agit de compléter la méthode utilisée par de nouveaux concepts ou par de nouvelles techniques.

Dans la première partie de cette thèse, nous avons décrit un modèle d'élaboration et de présentation d'une notation algorithmique basée sur les constructions classiques correpondant aux langages tels que Algol, Pascal et Ada [Ada 79a] [Wir 71a].

Il serait intéressant de poursuivre cette étude en appliquant notre démarche dans d'autres types de programmation. Nous pensons ici à la programmation logique [Luc 83] ou à la programmation applicative qui est concrétisée dans des langages tels que Lisp ou Smalltalk [MAE 69] [Coi 83].

Une comparaison entre les méthodes et la notation que nous avons utlisées, et d'autres méthodes d'analyse développées par ailleurs serait pour nous riche d'enseignement. Nous pensons ici aux méthodes des machines abstraites développées à Toulouse et présentées dans [GaM 78] ou au modèle Z utilisé dans le cadre de l'utilisation de base de données [Abr 78].

Le problème de l'adaptation de la notation à l'environnement de programmation est un facteur important pour sa bonne intégration et son utilité. L'aspect syntaxique de celle-ci n'est pas à négliger. Les matériels offrent des possibilités de plus en plus grandes et il est nécessaire de s'intéresser à leur impact dans la communication homme-machine (Un exemple d'intégration de l'outil graphique se trouve dans [Lem 83] et dans [Coi 83]).

Dans la deuxième partie de cette thèse, nous avons présenté un modèle permettant de décrire un objet programme. Notre démarche, dans ce travail, a été constructif. Nos objectifs initialement étaient réduit au problème de trouver une mémorisation appropriée de l'information programme. Ceci dans le but d'écrire un traducteur de la notation algorithmique.

Nous avons par la suite étendu nos objectifs en essayant : d'une part de suivre une démarche similaire à celle que nous avions empruntée dans la première partie ; d'autre part nous avons décidé de nous intéresser à tout l'environnement de programmation, le traducteur devenant un des outils à produire parmis d'autres.

Nous n'avons pas, dans cette étude, approfondi le problème de la création et de la modification interactive de l'objet programme. Ce problème n'est pas simple, surtout si l'on veut assurer une certaine correction des programmes ainsi manipulés. Des réalisations et des études sont faites à l'heure actuelle dans le domaine [Edi 83] [FeM 81] [RTD 83] [TeR 81], nous souhaiterions étudier et voir dans quelle mesure notre système peut s'adapter pour réaliser un éditeur interactif.

Bibliographie

[Abr 78] J.R. Abrial
Manuel du langage Z
rapport interne 1978

[Abr 81] J.R Abrial
Methodes de spécification et construction
dans [ACC 81]

[ACC 81] Outil pour la Conception et la Production de grands Logiciels
Troisièmes journées francophones sur l'informatique
Genève 1981

[Ada 79a] Preliminary ADA reference manual
Sigplan notices vol 14 n- 6 part A 1979

[Ada 79b] Rational for the design of ADA
Programming language
Sigplan notices vol 14 n- 6 part B 1979

[Ada 83] J.M. Adam
Méthodes et outils pour la production de didactiels :
l'environnement informatique du projet Mosaique
Thèse de 3 cycle INPG 83

[Ade 82] ADELE : un atelier de developpement de logiciel
rapport de recherche n- 299
laboratoire IMAG 1982

[AHU 74] A.V. Aho, J.E. Hopcroft, J.D. Ullman
The design and analysis of computer
Addison Wesley 1974

[Ars 79] J. Arsac
Syntactic source to source transforms and program manipulation
CACM vol 22 n- 7 July 1977

bibliographie

[Ars 83] J. Arsac
Les bases de la programmation
Editeur Dunod 1983

[Baa 78] S. Baase
computer algorithms : introduction to designs and analysis
Addison Wesley 1978

[BaG 81] G.Bartmub, R. Giegerich
Compiler develompement with MUG2 an intructory example
TUM april 1981

[BeB 83] P. Berlioux, P. Bizard
Algorithmique
construction, preuve et évaluation des programmes
Editeur Dunod 1983

[Ber 78] P. Berlioux
Application des propriétés des compositions séquentielle et parallèle des instructions à la transformation des programmes récursifs
RR.100 Laboratoire IMAG-USMG Grenoble 1978

[Ber 79] D. Bert
La programmation générique :
Construction de logiciel, spécification algébrique et vérification.
Thèse d'état USMG 1979

[Big 82] Systèmes intégrés de production de logiciels
BIGRE n- 28-29 Janvier 82

[Bla 73] L. Blaizot
Système de description de langages et de traducteurs par attributs
RR 20 IRIA 73

[BLW 79] P. Branquart, G. louis, P. Wodon
Une méthode de description systématique pour les langages algorithmiques.
Laboratoire de recherche Philips Bruxelles

[BoR 80] P. Boullier, K. Ripken
Building an Ada compiler following meta-compilation methods.
dans [SLT 81]

[Bou 80] P. Boullier
Le système Syntax, manuel d'utilisation
IRIA 77

[BPS 73] M. Bouckaert, A. Pirotte, M. Snelling
Soft, a tool for writing software.
Report R212 MBLE Bruxelles

[Bux 80] J.N. Buxton
An informal bibliography on programming support environments
Sigplan Notices Vol 15 n- 12 dec 80

[Cas 79] C. Casery
Construction méthodique de programmes : réalisation d'un traducteur de la notation MEFIA, vers PL/1
Généralisation à d'autres langages
Rapport de.DEA - USMG Grenoble 1979

[CGL 75] B. Cherbonneau, M. Galinier, J.P. Lagasse, H. Massie, B. Mathis, J.L. Paul
Programmation structurée
Ecole d'été de l'AFCET - Rabat 1975 - Rapport n- 112 UER Informatique - Université Paul Sabatier - Toulouse 1975

[CGS 78] P.Y. Cunin, M. Griffiths, P.C. Scholl
Aspect fondamentaux du langage MEFIA
Journées d'étude sur la fiabilité des programmes dans les applications industrielles (IRIA-EDF-Chapitre français de l'ACM) Clamart 26-27 avril 1978

[Che 76] B. Cherbonneau
Conception d'un projet étudiant par machines abstraites
Congrès AFCET 1976, Atelier Enseignement de l'informatique Novembre 1976

[Coi 83] P. Cointe
Une réalisation de Smalltalk en Vlisp
T.S.I vol 1 n- 4 1982

[CKL 76] J.L. Cheval, S Krakowiak, M. Lucas, J. Montuelle, J. Mossiere
Conception modulaire et système d'exploitation exemple de réalisation
RR n- 32 IMAG 76 .

[CoV 74a] J. Courtin, J. Voiron
Introduction à l'algorithmique et aux structures de données
IUT B cours - Grenoble 1974

[CoV 74b] J. Courtin, J. Voiron
Traduction des schémas de programmes en FORTRAN
IUT Informatique - USSG Grenoble 1974

[CSV 76] P.Y. Cunin, M. Simonet, J. Voiron
Méthodologie d'écriture de compilateurs . Une expérience du langage ALGOL 68
Thèse de docteur ingénieur - USMG, INPG Grenoble 1976

[Cun 79] P.Y. Cunin
Fiabilité et sécurité des programmes : propositions autour d'un langage d'essai
Thèse - CRIN, INPL - Nancy 1979

[DDH 72] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare
Structured Programming
Academic Press 1972

[DeF 79] J.C Derniame, J.P. Finance
Types abstraits de données : Spécification, utilisation et réalisation
CRIN 79

[Der 71] F.L. Deremer
Simple LR(k) grammars
CACM vol 14 n- 7 1971

[Der 78] P. Deransart
La formulation algébrique des attributs sémantiques selon L.M. Chirica et D.F. Martin
dans [SLT 81]

[DHK 80] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang
Programming environments based on structured editors: the Mentor experience
RR 26 INRIA 80

[Dub 78] J.P. Dubourreau
Construction méthodique de programmes : étude du langage MEFIA et d'un traducteur
Rapport de DEA - INPG - Grenoble 1978

[Edi 83] Les éditeurs dirigés par la syntaxe
2 tomes
Cours et Séminaires INRIA Avril 83

[FeM 80] P.H. Feiler, R. Medina-Mora
An incremental programming environement
C.M.U. Avril 80

[GaM 78] M. Galinier, A. Mathis
Les machines abstraites : unités de conception de logiciel fiable
Congrés sur la fiabilité des programmes dans les applications industrielles. Chapitre français de l'ACM, EDF, IRIA, 26-27 avril 1978

[Ger 77] A. Gerbier
Mes premières constructions de programmes
Lectures Notes in Computer Science n- 55 - Springer Verlag 1977

[GMS 77] X.M. Geschke, J.H. Morris, E.H. Satterthwaite
Early experience with Mesa
CACM vol 20 n- 8 Aout 77

[Jac 78] P. Jacquet
Les types génériques : propositions pour un mécanisme d'abstraction dans les langages de programmation
Thèse 3ème cycle - USMG. INPG Grenoble 1978

[Kah 78] G. Kahn
synthèse, manipulation et transformation de programmes
IRIA-SESORI Mai 78

[KaZ 81] U. Kastens, E. Zimmermann
A generator based on attribued grammar
Université Karlsruhe BRD
dans [SLT 81]

[KMP 79] B.B Kristensen, O.L. Madsen, B.M. Pedersen, K. Nygaard
Beta language proposal
Daimi PB-98 Aarhus University

[Kos 71] C.H.A. Koster
Affix grammars
Algol 68 inplementation
North Holland pub co (71)

[Lem 83] S. Lemoine
Graphique et enseignement assisté par ordinateur
Rapport de DEA INPG Juin 83

[LiZ 74] B. Liskov, Z. Zilles
Programming with abstract data types
Proceeding of ACM SIGPLAN Conference on very high level languages - SIGPLAN Notices vol. 9 n- 4 April 1974

[Lor 74] B. Lorho
De la définition à la traduction des langages de programmation; Méthodes des attributs sémantiques.
Thèse d'état 637 Ups Toulouse 1974

[Loy 81] M. Loyer
Modularité et compilation séparée : les choix du langage Legos
Thèse de docteur-ingénieur 771 Ups Toulouse 1981

[LPS 83] M. Lucas, J.P. Peyrin, P.C. Scholl
Algorithmique et représentation des données
Editeur Masson 1983

[LSV 77] M. Lucas, P.C. Scholl, J. Voiron
Apprentissage et utilisation du traitement séquentiel pour la construction de programmes
Rapport de recherche RR 74 - Laboratoire IMAG-USMG Grenoble 1977

[Luc 83] A. Lucci
Programmation logique et enseignement assisté par ordinateur
Rapport de DEA Juin 83

[LuS 75] M. Lucas, P.C. Scholl
Propositions pour une initiation à l'algorithmique
(2 volumes) Laboratoire IMAG-USMG Grenoble 1975

bibliographie

[MaB 78] R. Mahl, J.C. Boussard
Algorithmique et structures de données
Laboratoire d'informatique - Université de Nice 1978
et Editeur Eyrolles 1983

[MAE 69] J. MacCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, M.I. Lewin
Lisp programmer's manual
M.I.T. Press 69

[May 78] B.H. Mayoh
Attribute grammars and mathematical semantics
Daimi PB-90 Aarhus University

[MeB 78] B. Meyer, C. Baudoin
Méthodes de programmation
Collection de la D.E.R. d'EDF - Eyrolles 1978

[Mel 80] B. Melèse
Manipulation de programmes Pascal au niveau des concepts du langage
Thèse de 3 cycle Paris 11 1980

[Mel 81] B. Melèse
Mentor : l'environnement Pascal
RR 5 INRIA 81

[Mel 82] B. Melèse
Métal, un langage de spécification pour le système Mentor
TSI vol 1 n- 4 Juillet 82

[MKS 79] O.L. Madsen, B.B. Kristensen, J. Staunstrup
Use of design criteria for intermediate languages
Daimi PP-59 Aarhus University Aug 1976

[Mor 79] P. Morat
Construction méthodique de programmes : réalisation d'un traducteur de la notation MEFIA, étude de quelques primitives de contrôle
Rapport de DEA - INPG - Grenoble 1979

[MRR 82] J. Mossiere, J. Raymond, Y. Rouzaud
Représentation interne et manipulation de programmes dans l'atelier de logiciel ADELE
dans [Ade 82]

[Par 72a] D.L. Parnas
A technique for the specification of software modules with examples
Communications of the ACM, vol. 15, n- 5, pp. 330-336
May 1972

[Par 72b] D.L. Parnas
Onthe criteria to be used in decomposing systems into modules
Communications of the ACM, vol. 15, n- 12, pp. 1053-58
December 1972

[PeD 78] J.P Peyrin, M. Delaunay
Conception d'algorithmes et langages
RR 120 IMAG 1978

[Rai 79] K.J. Raiha
Bibliography on attribute grammars
Dept of Computer Science
University of Helsinki

[RTD 83] T. Reps, T. Teitelbaum, A demers
Incremental context-dependent analysis for langage-based editors
dans [Edi 83]

[ScC 76] P.C. Scholl, M. Cabric
Outils informatiques d'aide à l'enseignement et l'apprentissage de l'algorithmique : analyse prospective
Séminaire de programmation - Laboratoire IMAG - Mars 1976

[Sch 76] P.C. Scholl
Interprétation de programmes comme le traitement d'arbres : un aspect de la production de programmes par transformations successives
Rapport de recherche RR 54 - Laboratoire IMAG Grenoble 1976

[Sch 77a] P.C. Scholl
Méthodologie de la programmation : une étude de cas : construction d'un index
dans [LSV 77] Grenoble 1977

[Sch 77b] P.C. Scholl
Introduction à la récursivité et aux arbres
Support de cours - Institut de Programmation - USMG Grenoble 1977

[Sch 78a] P.C. Scholl
Etude du traitement séquentiel et d'une méthode de construction de programmes associée
Bulletin AFCET Informatique Enseignement, Vol.2 n- 2 Juin 1978

[Sch 78b] P.C. Scholl
Le traitement séquentiel : une classe de problèmes et une méthode de construction de programmes
Congrès AFCET_TTI Gif sur Yvette - Novembre 1978

[Sch 79] P.C. Scholl
Vers une programmation systématique : étude de quelques méthodes, techniques, outils
Thèse d'état INPG 79

[Sed 77] R. Sedgewick
Permutation Generation Methods
Computing surveys vol. 9, n- 2 June 1977

[Sim 81] M. Simonet
W-grammaires et logique du premier ordre pour la définition et l'implantation des langages
Thèse d'état - USMG Grenoble 1981

[SGC 78] P.C. Scholl, M. Griffiths, P.Y. Cunin
Construction méthodique et vérification systématique de programmes : éléments d'un langage
Congrés AFCET-TTI Gif sur Yvette - Novembre 1978

[SLT 81] Langages et Traducteurs
Séminaire INRIA 1981

[TeR 81] T. Teitelbaum, T. Reps
The Cornell program synthesizer : A syntax directed programming environment
CACM Vol 24 n- 9 Sept 81

[Tur 76] D.A. Turner
SASL language manual
1976

[Tur 80] J. Turner
The structure of modular programs
CACM vol 23 n- 5 1980

[VeC 72] F. Veillon, J.M Cagnat
Cours de programmation en langage PL/1
3 tomes
Collection U Armand Colin 72

[Vei 74] G. Veillon
Algorithmique
Cours 2ème A. ENSIMAG-C3 Logique et Programmation
USMG-INPG Grenoble - Décembre 1974

bibliographie

[Wat 79] D.A. Watt
An extended attribute grammar for Pascal
Sigplan notice 14.2 (page 60-74)

[Wir 71a] N. Wirth
The programming language PASCAL
Acta Informatica, vol. 1n N- 1, pp. 35-63

[Wir 71b] N. Wirth
Program development by stepwise refinement
Communications of the ACM, vol. 14 n- 4, pp. 221-27 -
April 1971

[Wir 76] N. Wirth
Algorithms + data structures = Programs
Prentice-Hall series in Automatic Computation 1976

[Yeh 77] R.T. Yeh (editor)
Current trends in programming methodology , vol. 1 :
software specification an design
Prentice Hall, Englewoods Cliffs, N.J. 1977

AUTORISATION DE SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974,

VU le rapport de présentation de Monsieur P.C SCHOLL, Professeur

Monsieur Philippe MORAT

est autorisé à présenter une thèse en soutenance pour l'obtention du titre de DOCTEUR DE TROISIEME CYCLE, spécialité "Informatique".

Fait à Grenoble, le 24 novembre 1983

Le Président de l'INP-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,