MASTER EN SCIENCES INFORMATIQUES

Types abstraits pour la programmation logique

Mounji, Abdelaziz

Award date:
1991

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Rue Grangagnage, 21, B-5000 NAMUR (Belgium)

TYPES ABSTRAITS POUR LA PROGRAMMATION LOGIGUE

Abdelaziz MOUNJI

Promoteur: Professeur Baudouin LE CHARLIER

Mémoire présenté en vue
de l'obtention du titre de
Licencié et Maître en Informatique

Année académique 1990-91

J'exprime avec un grand plaisir toute ma gratitude et reconnaissance à Monsieur Baudouin Le Charlier, promoteur de ce mémoire, pour son aide précieuse et continuelle. Ses enseignements et conseils judicieux constituent la charpente de ce travail. Je le remercie pour l'amabilité et l'enthousiasme avec lesquels il m'a toujours reçu.

Mes sincères remerciements vont également à tous les membres de l'équipe Folon pour l'accueil chaleureux qu'ils m'ont réservé et particulièrement à Messieurs Pierre De Boeck et Pierre Flener pour leur aide théorique et pratique.

Je ne terminerai pas sans remercier toutes les personnes qui m'ont soutenu, de près ou de loin, pour réaliser ce mémoire.

Résumé

Le style de programmation utilisant les types abstraits simplifie grandement la tâche du développeur en réduisant le coût du développement logiciel ainsi que l'effort de maintenance. Ceci est réalisé par l'amélioration de la généralité des composants du logiciel. Cette technique a eu un impact considérable dans le domaine de développement de logiciels. L'application de cette technique à la programmation logique est une perspective intéressante et prometteuse. L'approche consiste à doter le langage Prolog de structures de données nouvelles et efficaces tout en gardant un niveau d'efficacité acceptable pour les programmes manipulant ces structures de données. Les T.A. sont implémentés à l'aide des descriptions logiques. Le but de ce mémoire est de définir et implémenter une première version du langage A-Prolog qui supporte un style de programmation par type abstrait. Une étape importante pour réaliser cet objectif est la génération automatique de code Prolog exécutable à partir de descriptions logiques.

Abstract

A programming style using abstract data types significantly simplifies the developer's task by reducing the software development cost and the maintenance effort. This is achieved by improving generality and reusability of the software components. This technique is of a major impact in today's arena of software engineering. The application of this technique to logic programming is an attractive and promising perspective. The approach consists of augmenting Prolog with new and efficient data structures without a considerable loss of efficiency and readability of the programs processing these data structures. Abstract data types are implemented through the use of logic descriptions. The goal of this MS-thesis is to define and implement a first version of the A-Prolog language which is an extension to Prolog that supports an abstract data type programming style. An important step towards achieving this goal is to automatically derive executable Prolog code from logic descriptions by means of data-flow analysis.

I Introduction.

La programmation logique connaît actuellement une popularité croissante dans le développement de logiciels. Ecrire un programme logique reste une démarche élégante et très aisée. Les problèmes paraissent être résolus dès qu'ils sont correctement posés. En effet, on peut (si on ne se donne pas trop de souci d'efficacité) écrire des programmes Prolog dès que l'on a une spécification assez précise du problème en question. L'écriture du programme est une reformulation de la spécification en termes de formules logiques.

Un autre attrait de la programmation logique provient du fait qu'elle possède des fondements mathématiques très élaborés: la théorie des prédicats. L'adéquation de la programmation logique pour le développement de gros logiciels reste cependant très controversée. Les critiques se résument en une certaine pauvreté de Prolog en structures de données riches et efficaces ainsi que le manque d'efficacité des programmes. L'objectif de ce mémoire est de montrer qu'il est possible de remédier à ces problèmes par la définition de structure de données efficaces et réutilisables. Cette démarche utilisera la notion de type abstrait. Cependant, il importe de remarquer que la réalisation de telles structures implique une augmentation de la complexification des programmes et une diminution de leur expressivité.

Le second objectif de ce mémoire, se situe au niveau de la réalisation d'un outil automatisé pour défaire le programmeur logique de cette complexification et de restaurer l'expressivité des programmes tout en conservant un niveau d'efficacité acceptable.

La notion de type abstrait utilisée ici nous permettra de définir une extension au langage Prolog qui supportera la notion de type abstrait. Mais avant cela, il est nécessaire d'adapter cette notion pour qu'elle soit utilisable dans le cadre particulier de la programmation logique. Le style de programmation utilisant les types abstraits implique la construction d'une librairie de types abstraits. Pour cela, on a besoin d'un langage de spécification et d'implémentation de T.A. Le plan du mémoire découle naturellement de ces motivations.

Structure et contenu du mémoire

Ce mémoire est divisé en deux parties principales: la présentation informelle et la présentation formelle. Dans la première, on explique de façon intuitive les concepts de base que nous aurons à développer et formaliser dans la seconde partie.

Présentation informelle.

Dans une première étape, on va rappeler les concepts de base de la théorie des prédicats, de la programmation logique, ainsi que les caractéristiques de l'interpréteur Prolog. Dans la seconde étape, on introduira la notion de type abstrait. On montrera notamment comment spécifier un type abstrait à trois niveaux: la spécification fonctionnelle, la description logique et la représentation physique. La troisième partie sera consacrée à un essai d'intégration de ces deux notions: programmation logique et type abstrait. Les descriptions logiques des opérations définies sur un T.A. serviront de biais pour réaliser cette intégration. On montrera par des exemples les problèmes que posent l'utilisation de descriptions logiques pour l'implémentation des opérations sur un T.A. La notion de directionalité pour les procédures Prolog sera étendue pour tenir compte de ces problèmes. On parlera alors de directionalités abstraites. Dans la quatrième et dernière étape, on définira d'abord deux langages pour exprimer les descriptions logiques: A-Prolog et AD-log. Ensuite, on définira un troisième langage pour la définition et l'implémentation des T.A.

Présentation formelle.

Dans cette partie, on exposera de façon plus formelle les idées et concepts présentés dans la première partie. On définira notamment trois langages par leur syntaxe sémantique et règle de traduction. Les problèmes présentés dans la première partie concernant l'instanciation destructive des arguments seront reformulés et résolus de manière précise. On montrera ensuite comment transformer les descriptions logiques pour remédier à ces problèmes. Ces transformations nous serviront de base pour la traduction des programmes A-Prolog et AD-log. Finalement, on illustrera notre démarche par des exemples d'implémentation de T.A. tels que les entiers, les files d'attente et les piles.

II Partie informelle

1 Introduction

Dans cette partie du mémoire, nous nous attacherons à introduire les concepts fondamentaux de la programmation logique. A l'inverse des autres langages de programmation, appelés impératifs, les langages de programmation logiques ont des fondements mathématiques très élaborés. Ces langages sont basés sur la théorie des prédicats du premier ordre. Dans de tels langages, l'opération de base est la déduction logique. Un programme logique est un ensemble d'axiomes. L'exécution de ce programme par un interpréteur revient en fait à soumettre à celui-ci un but qui est une assertion ou formule à prouver. Le calcul consiste à établir que ce but est une conséquence logique du programme. L'ensemble des axiomes représente une formalisation en termes de formules logiques de la connaissance qu'on a du problème à résoudre. Il est intéressant de noter ici que la programmation logique est basée sur un modèle abstrait qui est indépendant d'une machine ou l'autre: le fait qu'une formule logique se déduit d'un certain ensemble d'axiomes est tout à fait indépendant de l'organisation des mécanismes d'exécution de la machine et de son modèle de fonctionnement en l'occurrence celui de Von Neumann. De part sa nature déclarative, un programme logique permet de résoudre un problème en ne s'intéressant essentiellement qu'aux objets du problème et aux relations qui peuvent exister entre ces objets. On évite ainsi de trop s'occuper des contraintes propres au langage, telles que la façon dont les instructions vont être exécutées.

Un programme logique écrit en Prolog par exemple, possède deux sémantiques. Un axiome:

A ssi B et C et D

où A, B, C et D sont des formules, peut s'interpréter de deux manières différentes. Lu déclarativement, cet axiome signifie que A est vraie si et seulement si B, C et D le sont. Lu procéduralement, il signifie que pour résoudre le problème A, il faut et il suffit de résoudre les problèmes B, C et D. Après une formalisation des concepts introduits ici, on va en fait montrer dans les sections suivantes que ces deux sémantiques sont équivalentes. Les définitions et théorèmes exposés ici sont basés sur [J.W. Lloyd 87] où on peut trouver une formalisation complète des concepts définis ici ainsi que la démonstration des propriétés et théorèmes.

2 Programmation logique

2.1 Introduction

On s'intéressera d'abord ici aux deux niveaux de considération des programmes logiques: le niveau déclaratif et le niveau procédural. Le premier de ceux-ci définit la signification des programmes logiques en termes de modèles et interprétations. On définira également ce qu'est une conséquence logique. Le second niveau spécifie comment fonctionne l'interpréteur abstrait lorsqu'il "exécute" un but. Pour cela, on définira l'arbre de dérivation qui constitue l'espace de recherche d'une solution par l'interpréteur. Ensuite, on s'attachera dans une troisième étape à établir un lien entre le niveau déclaratif et le niveau procédural. En énonçant deux résultats fondamentaux (*Soundness and completness*), on montrera qu'en fait, ces deux niveaux sont équivalents; Autrement dit, à toute conséquence logique d'un programme logique correspond une réponse calculée par l'interpréteur abstrait et réciproquement.

Finalement, l'interpréteur Prolog sera abordé. Il sera alors important de remarquer que l'introduction des informations de contrôle et les caractéristiques extralogiques nous font perdre cette équivalence entre le procédural et le sémantique. On peut alors avoir des réponses calculées qui ne sont pas des conséquences logiques et inversement des conséquences logiques qui ne peuvent être calculées par l'interpréteur Prolog. Cet écart entre la sémantique déclarative et procédurale est le prix que les programmeurs logiques payent pour obtenir des niveaux d'efficacités acceptables. En effet, l'utilisation des cuts permet de réduire l'espace de recherche de l'interpréteur Prolog.

2.2 Niveau déclaratif

2.2.1 Théorie du premier ordre

La théorie du premier ordre comprend un alphabet, un langage du premier ordre, un ensemble d'axiomes et un ensemble de règles d'inférence qui permettent de déduire des conséquences logiques à partir de ces axiomes et d'autres formules logiques.

L'alphabet est constitué par tous les symboles qu'on peut écrire dans le langage du premier ordre.

Définition: Un alphabet consiste en sept types de symboles :

1) Les variables

2) Les constantes

3) Les symboles de fonctions

4) Les symboles de prédicats

5) Les connecteurs logiques: $\sim$, $\Rightarrow$, $\Leftrightarrow$, $\vee$ et $\wedge$.

6) Les quantificateurs

7) Les symboles de ponctuation: '(', ')', ','.

Dans les langages de programmation logique, le terme constitue la structure de données de base. Un terme est défini récursivement par:

Définition: - Une variable est un terme.

- Une constante est un terme.

- Si f est un symbole de fonction d'arité n et t_1,..., t_n sont des termes, alors $f(t_1, \ldots, t_n)$ est aussi un terme.

Définition: Une *formule bien formée* (notée fbf) est définie inductivement comme suit:

- Si p est un symbole de prédicat d'arité n et t_1,... , t_n sont des termes alors $p(t_1, \ldots, t_n)$ est aussi une fbf appelée *atome*.
- Si F et G sont deux formules bien formées alors $\sim$F, $F \wedge G$ $F \vee G$, $F \Rightarrow G$, $F \Leftrightarrow G$ sont aussi des fbfs.
- Si F est une fbf et X est une variable, alors $\forall$ X et $\exists$ X sont des fbfs.

Définition: Une formule bien formée est dite fermée si toutes ses variables sont quantifiées universellement ou existentiellement.

Exemple: $\forall$ X,Y p(X,Y) est une formule fermée tandis que la formule $\forall$ X, Y q(X,Y,Z) n'en n'est pas une.

Définition: Un littéral est soit un atome soit la négation d'un atome.

Définition: Une clause est une formule de la forme:

$$A \text{ <-- } B_1, \ldots, B_n \qquad (1)$$

où toutes les variables sont quantifiées universellement et les virgules dans le second membre dénotent une conjonction. Le symbole <-- dénote l'implication logique.

Donc une clause de la forme (1) est équivalente à :

$$\forall x_1, \ldots, x_n (A \vee \sim B_1 \vee \sim B_2, \ldots, \vee \sim B_n)$$

où les x_i représentent les variables figurant dans la formule.

A est appelé *tête* de clause, $B_1, \ldots, B_n$ est appelé *corps* de la clause.

Définition: Un *but* est une formule de la forme :

$$<-- \ B_1, \ldots, B_r$$

C'est une clause qui a une tête vide. Un but est donc équivalent à la formule

$$\sim \exists x_1, \ldots, x_n (B_1 \wedge B_2 \wedge \ldots \wedge B_r)$$

Définition: Une clause dont la tête et le corps sont vides est appelée *clause vide* ou *contradiction* et est notée $\oplus$

Un *programme logique* est un ensemble fini de clauses.

2.2.2 Interprétations et modèles:

Jusqu'à présent, nous n'avons considéré que l'aspect syntaxique en définissant les symboles du langage ainsi que les formules bien formées constituant le langage du premier ordre. Dans cette section, on va s'attacher à l'aspect sémantique des programmes logiques en fixant une signification aux symboles et formules du langage. Autrement dit, on va définir une *interprétation* des formules des programmes. Une interprétation consiste à définir un domaine de valeurs pour les constantes, les variables, les symboles de fonction et les symboles de prédicat. Une interprétation d'un programme logique, ou d'un ensemble de formules logiques en général, est une manière de fixer un domaine de discours pour les symboles du langage. La définition d'une interprétation permet de parler de la valeur de vérité des formules. La *valeur de vérité* d'une formule est définie de la façon suivante:

Si la formule se réduit à un atome $p(x_1, \ldots, x_n)$, alors sa valeur de vérité est fixée par l'interprétation; sinon c'est une formule construite à partir de connecteurs et de quantificateurs logiques; sa valeur de vérité est obtenue par les tables de vérité de la façon habituelle.

Etant donné un programme logique, il existe certaines interprétations où toutes les clauses de ce programme ont une valeur de vérité VRAIE. De telles interprétations sont appelées *modèles*.

Définition: Soit $S = \{F_1, \ldots, F_n\}$ un ensemble de formules fermées d'un langage du premier ordre. On dit que F est une conséquence logique de S si et seulement si, tout modèle de S est aussi un modèle de F.

Définition: Soit $S = \{F_1, \ldots, F_n\}$ un ensemble de formules fermées d'un langage du premier ordre. On dit que S est insatisfaisable si et seulement si, il n'existe aucune interprétation de S qui soit un modèle pour S.

Proposition: F est une conséquence logique de $S = \{F_1, \ldots, F_n\}$ si et seulement si, l'ensemble des formules $S \cup \{\sim F\}$ est insatisfaisable.

Démonstration: Voir [J.W.Lloyd. 87].

Exemple: Soit $S = \{p(a, b), \forall x, y : p(x, y) <= p(y, x)\}$. Alors $F = p(b, a)$ est une conséquence logique de S. Soit en effet I, une interprétation quelconque de S qui soit un modèle pour S alors: $\forall x, y : p(x, y) <= p(y, x)$ est vraie donc $p(b, a) <= p(a, b)$ est vraie en particulier. Comme p(a, b) est vraie par rapport à I, il en résulte que p(b, a) est aussi vraie.

Etant donné un programme logique P, l'application de cette définition montre que lorsqu'on soumet un but $G : <-- B_1, \ldots, B_r$ à l'interpréteur, on lui demande de montrer que $P \cup \{ G \}$ est insatisfaisable. Autrement dit, le système montre que G est faux dans tout modèle de P considéré comme un ensemble de formules. Autrement dit encore que la formule:

$$\exists y_1, \ldots, y_n (B_1 \wedge B_2 \wedge \ldots \wedge B_r)$$

est vraie dans tout modèle de P.

Remarque: Pour montrer que $P \cup \{ G \}$ est insatisfaisable, il faut montrer que toute interprétation de $P \cup \{ G \}$ n'est pas un modèle. Dans la pratique, ce problème s'avère non résolvable puisqu'on ne peut pas vérifier que toutes les interprétations une à une ne sont pas des modèles. Il s'avère heureusement qu'il existe une classe réduite d'interprétations qui suffisent à établir l'insatisfaisabilité de $P \cup \{ G \}$. Ces interprétations sont appelées *interprétations d'Herbrand.*

Définition : Soit L un langage du premier ordre, l'*Univers d'Herbrand* est l'ensemble de tous les termes ground (i.e. les termes ne contenant pas de variables). Ces termes étant construits à partir des constantes et des symboles de fonctions du langage.

Définition : Une *base d'Herbrand* associée à un langage L est l'ensemble des atomes ground .

Une interprétation d'Herbrand est une interprétation dont le domaine de valeurs n'est autre que l'univers d'Herbrand. Pour se fixer une interprétation d'Herbrand, il suffit de fixer une partie de la base d'Herbrand puisque l'Univers d'Herbrand est fixé une fois pour toute. Donc, une interprétation est entièrement caractérisée par un sous-ensemble de la base d'Herbrand. Ce sous-ensemble est formé de tous les atomes vrais. Inversement, étant donné une partie de la base d'Herbrand, on peut lui associer de façon univoque une interprétation d'Herbrand. La base d'Herbrand (notée B_P) associée à un programme P est toujours un modèle de ce programme. L'ensemble des modèles d'un programme n'étant donc jamais vide, on peut parler de l'intersection de tous les modèles d'un programme donné. Cette intersection est appelée *le plus petit modèle d'Herbrand* associé à ce programme; on le note M_P.

On démontre que,

$M_P = \{ A \in B_P$: tel que A est une conséquence logique de P $\}$.

M_P constitue la *sémantique déclarative* du programme P i.e. l'ensemble de tous les atomes ground qui peuvent être déduits de P.

Exemple: Soit le programme logique:

append([X|Xs], Ys, [X| Zs]) <-- append(Xs, Ys, Zs).

append([ ], Xs, Xs).

L'Univers d'Herbrand de ce programme est l'ensemble de tous les termes ground qu'on peut former avec la constante [ ] représentant la liste vide et la fonction • (a, L) qui permet de construire une liste avec un terme et une liste.

$U_H = \{ [ ], [[ ]], [[ ] , [ ]], \ldots \}$

La base d'Herbrand est l'ensemble de tous les atomes *append* ayant comme arguments des éléments de l'Univers d'Herbrand:

$M_P = \{$ append([ ], Ys, Ys) , Ys $\in U_H \} \cup$

$\{$ append([[ ] | Xs], [ ], [[ ] | Xs]) , Xs $\in U_H \}$.

Donc M_P est la sémantique déclarative de append.

2.2.3 Description logique

Une description logique est la définition d'un prédicat donné par l'intermédiaire d'une formule logique ayant la forme particulière suivante:

$\forall x_1 \ldots \forall x_n \ (p(x_1, \ldots, x_n) \Leftrightarrow \text{Def})$

où Def est une formule.

Dans le cadre du développement systématique de programmes logiques, on considère trois étapes principales. Dans la première, on élabore une spécification du problème en question dans un langage informel. A partir de cette spécification, on construit une description logique de ou des relations entre les objets décrits par cette spécification. C'est une façon plus formelle de décrire les mêmes objets. A ce stade, on ne s'intéresse qu'à l'aspect logique du problème indépendamment de tout langage. Dans cette étape, on peut s'intéresser à la correction d'une description logique par rapport à sa spécification; ce concept ne sera pas défini ici.

La troisième étape est la dérivation d'un programme logique à partir de la description logique obtenue dans la seconde étape. On obtient ce programme logique en réécrivant la description logique sous forme de clauses. Par la suite, il faut considérer d'autres transformations telles que la permutation de l'ordre des clauses, les directionalités,... etc, avant d'arriver à un programme logique correct par rapport à un langage de programmation donné tel que Prolog.

Dans[Deville.90], la notion de description logique est traitée de façon complète. La construction, la transformation et la correction de decriptions logiques y sont également développées. Ces démarches s'inscrivent dans le cadre plus général de la méthodologie de développement systématique de programmes logiques.

2.3 Niveau procédural

2.3.1 Introduction

Dans cette partie on va introduire la notion de *sémantique procédurale*. Cette sémantique, décrit l'effet du mécanisme de contrôle lié à la manière dont l'interpréteur fonctionne. On va d'abord commencer par quelques définitions empruntées à [Lloyd 87], où on peut trouver une vue plus détaillée de ces concepts et surtout la démonstration des résultats qui ne seront qu'énoncés dans ce qui suit.

2.3.2 Définitions préliminaires

. Une *substitution* est un ensemble fini de la forme $\theta = \{ x_1/ t_1, \ldots , x_n/t_n \}$ où les x_i sont des variables distinctes et les t_i des termes. Chaque t_i est distinct de v_i. Chaque élément x_i / t_i est appelé *liaison*. Si t_i est ground pour tout i, alors la substitution θ est dite ground.

. Une *expression* est soit un terme, un littéral ou une conjonction ou disjonction de littéraux.

. Soit E une expression et soit $\theta = \{ x_1/ t_1, \ldots, x_n / t_n \}$ une substitution. $E\theta$ est l'expression obtenue en remplaçant simultanément chaque occurrence de la variable x_i par le terme t_i ($i = 1 , \ldots , n$). $E\theta$ est appelé *instance* de E par θ.

. Soient $\theta = \{ u_1/ s_1, \ldots, u_n / s_n \}$ et $\sigma = \{ v_1/ t_1, \ldots , v_m/ t_m \}$ deux substitutions. La composition $\theta\sigma$ des deux substitutions θ et σ est la substitution définie par l'ensemble: $\{u_1/s_1\sigma,\ldots , u_m/ s_m \sigma, v_1/ t_1, \ldots , v_n/ t_n \}$ en supprimant tout élément tel que $u_i = s_i\sigma$ et tout élément v_i / t_i tel que $v_i \in \{ u_1, \ldots , u_n \}$.

La composition de deux substitutions θ et σ est la substitution γ dont l'application sur toute expression E a le même effet que l'application de la substitution θ à l'expression E suivie de l'application de la substitution σ à l'expression $E\theta$.

. Soient E_1, E_2 deux expressions. Soit θ une substitution. On dit que θ unifie E_1 et E_2 si et seulement si les deux expressions E_1 et E_2 sont syntaxiquement identiques, symbole par symbole.

Ex: Si

$E_1 = p(f(x), z)$,

$E_2 = p(y, a)$,

alors la substitution $\theta = \{ x/ a, y/ f(a), z/ a\}$ unifie E_1 et E_2 .

. Soient E_1, E_2 deux expressions. Soit θ une substitution. On dit que θ est un *mgu* (*most general unifier*) de E_1 et E_2 si et seulement si θ unifie E_1 et E_2 et pour toute substitution σ qui unifie E_1 et E_2 on a :

$\exists \gamma$ une substitution telle que $\sigma = \theta\gamma$.

Exemple: Considérons l'exemple précédent, le mgu de E_1et E_2 est :

$$\sigma = \{y / f(x) , z / a\}.$$

Remarquons que $\theta = \sigma \{ x / a \}$.

L'unification est l'opération de base en programmation logique; c'est l'analogue de l'instruction d'affectation dans les langages conventionnels.

2.3.3 SLD-dérivation

. Soit P un programme logique, et G le but <-- $B_1, \ldots, B_q, \ldots, B_r$ Le but G' *dérivé* de G en sélectionnant le littéral B_q et la clause C est le but :

<-- ($B_1, \ldots, B_{r-1}, C_1, \ldots, C_n, B_{r+1}, \ldots, B_r$) θ

où θ est un mgu de A et B_q .

B_q est appelé le *littéral sélectionné.*

. Soit P un programme logique, et G un but. La SLD-dérivation de $P \cup \{G\}$ est constituée d'une séquence de substitutions $\theta_1, \ldots, \theta_n$ et d'une séquence de buts $G_0 = G, G_1, \ldots, G_n$ dans lesquels G_{k+1} est dérivé de G_k et de la clause C_{k+1} en utilisant la substitution θ.

. Une SLD-refutation de $P \cup \{G\}$ est une SLD-dérivation dans laquelle le dernier but est la clause vide $G_n = \Diamond$.

Expliquons à présent comment l'interpréteur prouve qu'une formule:

$$\exists y_1, \ldots, y_n \ (B_1 \wedge \ldots \wedge B_r)$$

est une conséquence logique d'un programme P. Pour prouver ce but, l'interpréteur travaille par réfutation. D'abord, on nie le but qu'on veut prouver, puis on le rajoute aux clauses du programme et on essaye d'aboutir à une contradiction i.e. la clause vide. Donc, partant du but:

<--- $B_1, \ldots, B_r$

on dérive à chaque étape un nouveau but en sélectionnant un littéral dans ce but et en le remplaçant par le corps d'une clause dont la tête s'unifie avec ce littéral. Certaines clauses sont des faits i.e. leur corps est vide. Donc on peut progressivement diminuer le nombre de littéraux dans le but qu'on veut prouver.

Les négations sont considérées de la manière suivante (*négation par échec*):

Not(p) réussit (i.e. on réussit à prouver que p est vraie) si p échoue et inversement.

2.3.4 Arbre de dérivation

Le comportement de l'interpréteur peut être expliqué par la construction d'un arbre appelé *arbre de dérivation.*

Définition : Une règle de sélection de but est une fonction de l'ensemble des buts vers l'ensemble des littéraux. Cette fonction associe à chaque but

$$<\text{---}\ B_1, \ldots, B_r$$

un littéral $B_q \in \{ B_1, \ldots, B_r \}$. B_q est appelé le *sous-but sélectionné.*

Définition: Soit P un programme, G un but et R une règle de sélection de buts. L'arbre de dérivation de $P \cup \{ G \}$, en utilisant la règle de sélection R, est définie comme suit:

(1) chaque nœud de l'arbre est un but.

(2) une substitution est associée à chaque arc de l'arbre.

(3) le nœud racine est le but initial G.

(4) les nœuds qui sont des buts vides n'ont pas de descendants. Ces nœuds sont appelés *nœuds de succès.* La branche de l'arbre partant du nœud racine et aboutissant à un nœud succès est appelée *branche de succès.*

(5) soit $<\text{---}\ L_1, \ldots, L_k, \ldots, L_n$ un nœud G' de l'arbre de dérivation ($n \geq 1$) et L_k le littéral sélectionné par la règle de sélection R.

(a) si L_k est un littéral positif i.e. un littéral qui n'est pas une négation, alors :

(i) ce nœud possède un nœud descendant pour chaque clause

$$A <\text{---}\ B_1, \ldots, B_r$$

dont la tête s'unifie avec L_k.

(ii) le mgu θ tel que $A\theta = L_k\theta$ est attaché à l'arc.

(iii) le nœud descendant est

$$<\text{---}\ (L_1, \ldots, L_{k-1}, B_1, \ldots, B_r, L_{k+1}, \ldots, L_n)\ \theta$$

si le nœud n'a pas de descendant, ce nœud est appelé *nœud échec* et la branche reliant le nœud racine à ce nœud est appelée *branche d'échec.*

(b) si L_k est une négation not(A_k), alors, ou bien :

(i) l'arbre de dérivation de $P \cup \{ <\text{---}\ A_k \}$ est un arbre d'échec. Dans un tel cas, l'unique nœud descendant est :

$$<\text{---}\ L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_n$$

et la substitution attachée à cet arc est la substitution identité.

(ii) il existe une branche de succès dans l'arbre de résolution de P ∪ {<--- A_k } via R. Dans ce cas il n'y a pas de nœud descendant pour G. G' est un nœud d'échec et toute branche partant du nœud racine et aboutissant au nœud G' est une branche d'échec.

(iii) L'unique descendant pour le nœud G' est lui-même et la substitution attachée à cet arc est la substitution identité. On obtient donc une branche infinie.

Définition : Un arbre de dérivation est un arbre d'échec si et seulement si, toute branche de cet arbre est une branche d'échec.

Une explication plus détaillée de l'arbre de dérivation peut être trouvée notamment dans [Y. Deville 90] et [J.W. Lloyd. 87].

Définition : Soit P un programme logique et G un but. La substitution θ est une réponse calculée de P ∪ { G } si et seulement s'il existe dans l'arbre de dérivation de P ∪ { G } une branche de succès telle que θ soit la composition des substitutions attachées aux arcs de cette branche.

2.4 Lien entre la sémantique procédurale et déclarative :

2.4.1 Introduction

Maintenant que nous avons défini les deux sémantiques d'un programme logique, on peut se poser la question de savoir s'il existe un lien entre ces deux sémantiques. Notamment, on peut se demander si toute conséquence logique d'un programme peut être prouvée par réfutation. Autrement dit, si toute réponse correcte peut être calculée par un arbre de dérivation ou encore, si à toute réponse correcte, il correspond une réponse calculée et inversement.

Nous allons énoncer deux résultats fondamentaux (complétude et correction). Cependant, on va se restreindre aux programmes logiques et buts ne comportant pas de négation. Ces résultats sont également étendus aux programmes et buts avec négation. On ne donnera pas non plus les preuves de ces résultats car ceci n'est pas le but recherché ici. Le lecteur intéressé peut se référer à [J.W. Lloyd. 87].

2.4.2 Théorèmes de complétude et de correction

Définition : Soit P un programme sans négation, l'*ensemble succès* de P est l'ensemble des atomes ground (i.e. appartenant à B_P base d'Herbrand de P) tel que $P \cup \{ G \}$ admette une réfutation.

Théorème :

L'ensemble succès d'un programme P est égal à son plus petit modèle d'Herbrand.

Le second résultat affirme que toute réponse correcte est une instance d'une réponse calculée.

Théorème :

Soit P un programme logique sans négation et G un but sans négation. Pour toute réponse correcte θ de $P \cup \{ G \}$ il existe une réponse calculée σ de $P \cup \{ G \}$ telle qu'il existe une substitution γ telle que l'on ait : $\theta = \sigma \gamma$.

Puisque dans l'arbre de dérivation on cherche à chaque fois un mgu pour unifier un sous-but avec la tête d'une clause, il est prévisible que la réponse calculée soit plus générale que la réponse correcte. Donc, la réponse correcte est une instance d'une réponse calculée. Le troisième résultat est le suivant appelé *complétude*:

Théorème :

Soit P un programme logique sans négation et G un but sans négation, alors, toute réponse calculée pour $P \cup \{G\}$ est une réponse correcte pour $P \cup \{G\}$.

2.4.3 Sémantique dénotationnelle

A chaque programme logique, on peut associer une fonction très intéressante. Cette fonction permet de faire un lien entre la sémantique déclarative et la sémantique procédurale.

Définition :

Soit P un programme sans négation. La fonction :

$T_P : P(B_P)$ --------> $P(B_P)$

(où $P(B_P)$ désigne l'ensemble des parties de B_P) est définie comme suit :

Soit I une interprétation d'Herbrand, alors :

$T_P(I) = \{ A \in B_P : A <\text{---} A_1, \ldots, A_n$ est une instance d'une clause de P et

$\{ A_1, \ldots, A_n \} \subseteq I \}$.

Propriétés de T_P :

T_P est une fonction monotone i.e. pour toutes interprétations d'Herbrand I, J, on a:

$$I \subseteq J \Rightarrow T_P(I) \subseteq T_P(J).$$

T_P est également une fonction continue, mais ce concept ne sera pas défini ici. Pour une définition détaillée de ce concept on peut consulter [J.W. Lloyd 87].

Exemple :

Soit P le programme:

p(f(x)) <--- p(x).

q(x) <--- p(x).

Posons $I_1 = B_P$, $I_2 = T_P(I_1)$ et $I_3 = \emptyset$. Alors:

$T_P(I_1) = \{ q(a) \} \cup \{ p(f(t)) : t \in U_P \}$.

$T_P(I_2) = \{ q(a) \} \cup \{ p(f(f(t))) : t \in U_P \}$.

$T_P(I_3) = \emptyset$.

Où U_P est l'Univers d'Herbrand i.e. l'ensemble de tous les termes que l'on peut former avec les constantes et symboles de fonction apparaissant dans le programme P : $U_P = \{$ a, f(a), f(f(a)), ..., f(... f(f(a)) ...) , ... $\}$. U_P est un ensemble infini.

Proposition 1:

Soit P un programme logique sans négation et I une interprétation d'Herbrand. Alors, I est un modèle de P si et seulement si $T_P(I) \subseteq I$.

Définition :

Soit T une fonction de A vers B et soit x un élément de A. x est un point fixe de T si et seulement si T(x) = x.

Si $\Re$ est une relation d'ordre définie sur A, alors le *plus petit point fixe* p de T, noté lfp(T), est un point fixe de T tel que : $\forall x \in A : (T(x) = x \Rightarrow p \Re x)$.

Proposition 2 : (Caractérisation du plus petit modèle d'Herbrand)

Soit P un programme logique sans négation alors, $M_P = lfp(T_P)$.

Soit M un modèle de P, donc $T_P(M) \subseteq M$ par la proposition 1 et donc $T_P(T_P(M)) \subseteq T_P(M) \subseteq M$. Ainsi, à chaque fois qu'on applique T_P à un modèle de P, on obtient un modèle plus petit. Intuitivement, pour trouver le plus petit modèle d'Herbrand, on peut partir de n'importe quel modèle d'Herbrand et lui appliquer T_P un certain nombre de fois jusqu'à trouver le

plus petit point fixe. Partant à chaque fois d'un modèle d'Herbrand et après plusieurs applications successives de T_P on trouve d'autres points fixes. Le plus petit point fixe parmi tous ceux-là est précisément le plus petit modèle d'Herbrand.

2.5 L'interpréteur Prolog

2.5.1 Introduction

On va s'intéresser ici à la façon dont l'interpréteur Prolog fonctionne. Pour résoudre un but, l'interpréteur construit un arbre de dérivation. Dans l'arbre de dérivation qu'on a défini précédemment, nous n'avons fait aucun choix ni sur la règle de sélection des littéraux dans un but, ni dans quel ordre il fallait considérer les clauses du programme dont la tête s'unifie avec le littéral sélectionné.

2.5.2 Règles d'exécution

On peut spécifier l'interpréteur Prolog en fixant un choix pour la règle de sélection des littéraux et pour l'ordre de considération des clauses. En Prolog, la règle de sélection des littéraux est celle qui pour un but donné choisit le littéral le plus à gauche. Les clauses sont inspectées dans l'ordre suivant lequel elles figurent dans le programme.

Il en résulte que l'ordre des clauses dans un programme Prolog ainsi que l'ordre des littéraux dans un but a beaucoup d'importance.

2.5.3 Backtracking

Le langage Prolog utilise le mécanisme de *backracking* dans la recherche d'une solution à un but. On peut expliquer brièvement ce mécanisme comme suit:

Supposons qu'à un instant donné de l'exécution, l'interpréteur Prolog exécute le but <--- $B_1, \ldots, B_r$. Pour que ce but puisse réussir, il faut que chacun de ses sous-buts B_i réussisse. Supposons que ce but échoue, (i.e. il existe un sous-but B_j tel que B_j échoue) alors le mécanisme de backtracking permet de remonter dans l'exécution à la dernière fois où l'on a unifié un littéral avec la tête d'une clause. S'il y a une autre clause qui pouvait s'unifier avec ce littéral, alors toutes les liaisons de variables faites jusque là sont défaites et on essaie de satisfaire ce but avec un autre jeu de substitutions pour la deuxième clause. Pour plus de détails on peut consulter [Clocksin 84].

2.5.4 Caractéristiques extralogiques

2.5.4.1 Le cut

En Prolog, il existe un moyen pour modifier le comportement "normal" de l'interpréteur. Ceci est utilisé pour rendre les programmes plus efficaces. Il y a donc ici un écart entre l'interpréteur abstrait qu'on a spécifié précédemment et l'interpréteur Prolog. On peut donner une explication procédurale de l'effet de l'introduction d'un cut dans une clause.

Le cut permet d'éviter l'exploration d'une partie de l'arbre de résolution. Cette partie est telle qu'on est capable de déterminer à priori qu'il n'y aura aucune solution et donc qu'il n'y aura pas besoin d'explorer cette partie de l'arbre.

Le cut est un prédicat sans argument qui réussit toujours et qui ne peut être réétabli. Appelons *nœud parent*, le nœud qui contient un littéral qui s'unifie avec la tête d'une clause dont le corps contient un cut. Tous les choix d'instanciation de variables faits pour essayer d'obtenir ce nœud parent une fois le cut passé sont gelés: Le système ne pourra jamais essayer d'autres choix, d'autres combinaisons pour ces variables. Le système est lié à ces choix.

Exemple

Soit P le programme suivant:

A <--- B , C.

...

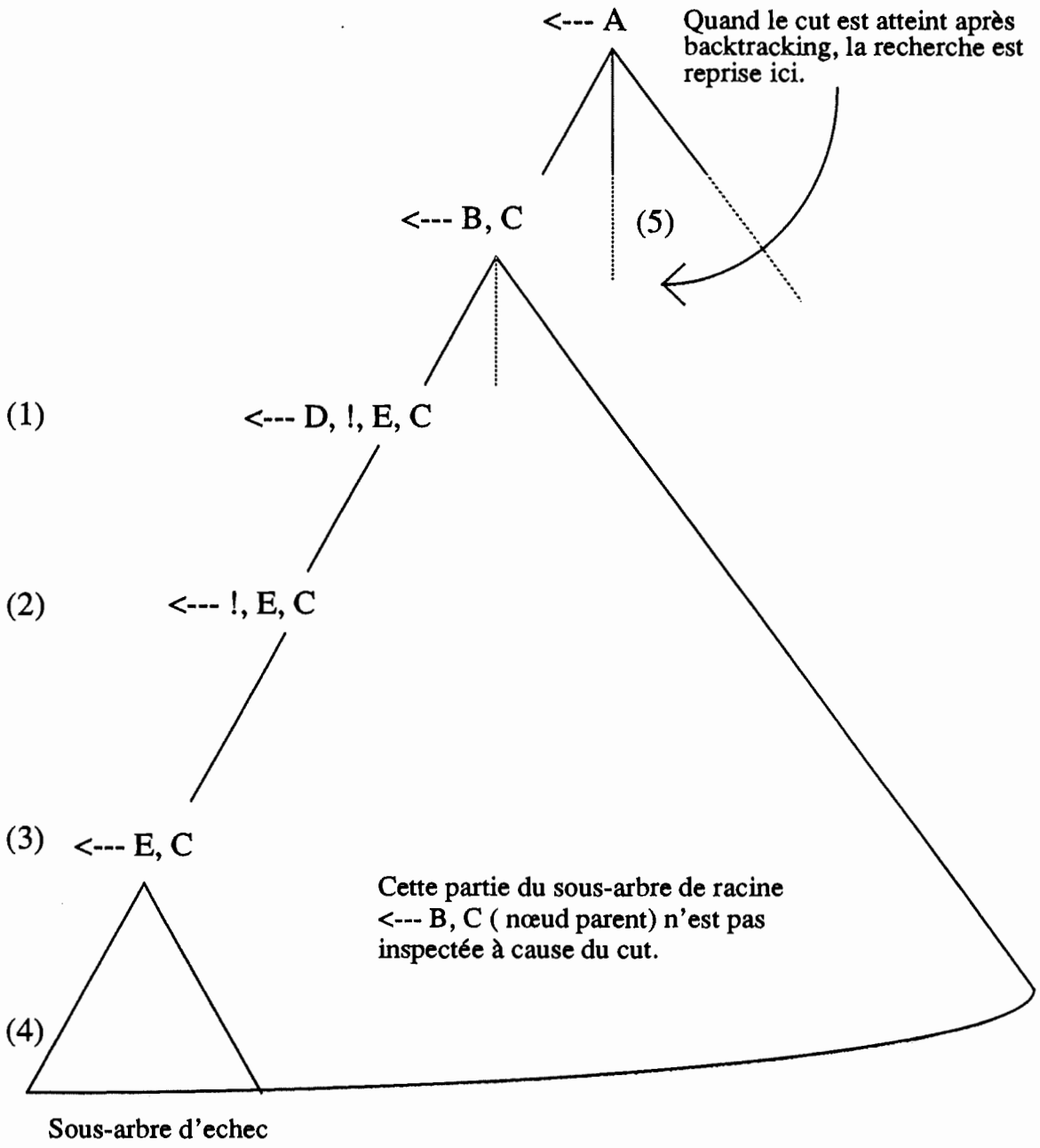
B <--- D , ! , E.

o

D <---

...

Pour illustrer l'effet de l'introduction du cut, voici l'arbre de dérivation de $P \cup \{<\!\!-\!\!-A\}$ construit par l'interpréteur.



(1) L'atome sélectionné B s'unifie avec la tête d'une clause dont le corps contient un cut.

(2) L'atome D est sélectionné et réussit car il s'unifie avec un fait.

(3) Le cut ! réussit par définition.

(4) L'atome E échoue et le système fait un backtracking jusqu'au cut pour essayer de nouveaux jeux d'instanciation.

(5) Le système suspend toute recherche dans le sous-arbre de racine : <--- B, C mais reprend la recherche avec un nouveau choix pour le but <--- A.

2.5.4.2 Caractéristiques extralogiques

Pour que Prolog soit un langage de programmation pratique, il est nécessaire de disposer d'un certain ensemble de mécanismes qui n'ont pas de signification dans le modèle de programmation logique mais qui permettent de réaliser certaines tâches telles que les entrées sorties.

En Prolog, il existe en effet trois classes de prédicats: Les prédicats d'entrées sorties, les prédicats d'accès et manipulation de programmes et les prédicats d'interfaçage avec le système d'exploitation sous-jaçent. Tous ces prédicats réalisent un effet de bord lorsqu'ils sont satisfaits comme buts logiques.

2.5.4.3 Prédicats d'entrées sorties :

Il existe principalement deux prédicats d'entrées sorties:

read(x)

Ce but lie un terme à la console. Le terme lu est unifié par la suite avec x. Si x est une variable, alors elle est instanciée à ce terme et le prédicat réussit. Si x est un terme ground, alors read(x) réussit ou échoue selon le résultat de l'unification de x avec ce terme.

write(x)

L'effet de ce terme est d'écrire sur la console de sortie courante le terme x. Ce prédicat réussit donc toujours.

Ces deux prédicats ne sont pas backtrackables i.e. il ne donnent pas de solution par backtracking. Par exemple, lorsque read(x) échoue parce que x ne s'unifie pas avec le terme lu, alors ce terme est perdu et on ne peut pas réessayer de satisfaire ce but.

2.5.4.4 Prédicat d'accès et manipulation de programmes

Dans les langages de programmation conventionnels, une fois qu'un programme est compilé et chargé en mémoire centrale, le texte du programme reste le même pendant toute l'exécution, sauf si on arrête cette exécution, qu'on modifie le programme et qu'on le recharge en mémoire centrale pour une nouvelle exécution.

En Prolog, on peut modifier dynamiquement le texte d'un programme *pendant* son exécution : le programme "se modifie par lui-même". On peut ainsi rajouter et retirer certaines clauses du programme pendant qu'il s'exécute. Pour cela, Prolog offre principalement deux prédicats (En BIM_Prolog, il existe plusieurs variantes à ces prédicats) qui agissent par effet de bord.

Assert(p)

Où p est une clause. L'effet de ce but est de rajouter la clause p au programme. Cette clause est mise à la fin du programme.

Exemple :

Supposons que le programme en mémoire soit :

```
student( René ).
student( toto ).
teacher( ultra ).
```

Si on exécute le but ?- assert(student (Dupond)). Alors, le nouveau texte du programme sera :

```
student( René ).
student( toto ).
teacher( ultra ).
student( Dupond ).
```

Retract(p)

Où p est une clause. Ce prédicat retire du programme en mémoire la première clause qui s'unifie avec p. Si une telle clause existe, le prédicat réussit et la retire du programme. Sinon, le prédicat échoue. Dans l'exemple précédent, si on exécute le but:

?-retract(student (René)).

alors en mémoire, on aura le programme :

student(toto).

teacher(ultra).

student(Dupond).

Il existe plusieurs variantes à ces prédicats dont la spécification dépend des différentes implémentations de Prolog.

Remarque:

En fait, Prolog contient une base de données interne qui contient l'ensemble de toutes les clauses compilées donc connues par Prolog. Les prédicats *assert* et *retract* permettent de rajouter ou de retirer des clauses à cette base.

Il existe une directive Prolog qui permet de déclarer qu'un prédicat est dynamique i.e. peut être manipulé par *assert* et *retract*. Cette directive doit être incluse dans le programme contenant ces prédicats.

2.5.4.5 Prédicats système

En Prolog, il est possible de faire des appels système à partir d'un prédicat Prolog et sans devoir quitter l'interpréteur pour aller au niveau du système d'exploitation. Ceci peut être très utile pour certaines applications. On peut par exemple consulter ou modifier le contenu d'une directorie ou d'un fichier.

Il existe également des prédicats qui réalisent l'interface avec certains outils de développement logiciels tels que SGBDs, langage C ... etc. Ces prédicats et leur syntaxe étant dépendants de chaque implémentation de Prolog et de l'OS, il ne serait pas possible de les examiner davantage.

2.5.5 Directionalités

Dans les langages de programmation impératifs tel que Pascal, les paramètres d'une procédure jouent un seul rôle bien précis: Ou bien il s'agit d'un paramètre donné qui doit recevoir une valeur avant exécution de la procédure, ou bien il s'agit d'un paramètre résultat qui est instancié à une valeur après exécution (normale) de la procédure.

En programmation logique, les paramètres d'une procédure peuvent être utilisés aussi bien comme données que comme résultats : il suffit d'instancier lors de l'appel de procédure les paramètres correspondant aux données et de laisser les autres non instanciés. C'est une caractéristique propre aux langages logiques.

Cependant, il est rare de trouver des procédures logiques qui s'exécutent correctement (par rapport à leurs spécifications), quel que soit le type des paramètres : donnée ou résultat. Donc, pour spécifier complètement une procédure, il est nécessaire de décrire de façon précise le degré d'instanciation des paramètres avant exécution. Cette description est appelée *directionalité* de la procédure.

Pour une procédure, on distingue trois formes de paramètres actuels :

. ground : lorsque le paramètre actuel ne doit contenir aucune variable.

. variable : lorsque le paramètre actuel doit être une variable.

. ngv : lorsque le paramètre doit contenir au moins un nom de variable et ne doit pas être réduit à une variable.

Pour être plus complète, une directionalité d'une procédure doit également spécifier le degré d'instanciation des paramètres actuels après exécution. Ainsi, dans [Y. Deville 90], une directionalité est définie par :

$$\text{in}(m_1, \ldots , m_n) : \text{out} (M_1, \ldots , M_n) \qquad (1)$$

Où $m_i, M_i \in \{ \text{ground}, \text{var} , \text{ngv} \} \ \forall \ i : 1 \leq i \leq n$.

La directionalité d'une procédure est donc une liste de directionalités de la forme (1). Ces directionalités représentent les utilisations possibles de cette procédure. La directionalité d'une procédure peut être considérée comme faisant partie de la précondition et de la postcondition de cette procédure. Par convention, si un paramètre actuel ne respecte pas la directionalité d'une procédure, alors le résultat de l'exécution est indéfini.

Exemple :

La procédure classique append / 3 peut illustrer ces concepts:

append([], L, L).

append([H|T], L, [H|Lapp]) :- append(T, L, Lapp).

Sa directionalité est :

in (ground, ground, ground) : out (ground, ground, ground).

in(var, ground, ground) : out (ground, ground, ground).

Des concepts plus élaborés tels que cohérence, minimalité, compatibilité et correction d'une directionalité sont exposés dans [Deville. 90].

3 Notion de Type Abstrait

3.1 Introduction

La notion de Type Abstrait (T.A) connaît actuellement un succès certain dans le cadre de développement logiciel. L'état de l'art en la matière est très avancé aujourd'hui. Cette technique a reçu et reçoit toujours un grand intérêt de la part des chercheurs. L'application de cette notion au développement logiciel réduit beaucoup les coûts de maintenance du fait que les modules utilisant cette notion acquièrent une plus grande généralité et un faible degré de dépendance. Le style de programmation avec T.A. constitue l'un des fondements de la programmation orientée-objet.

L'idée directrice derrière la notion de T.A. est de traiter les données comme étant des objets logiciels à part entière. Ceci est réalisé en considérant les données comme des objets abstraits entièrement caractérisés par leurs propriétés fonctionnelles abstraites, intrinsèques et indépendantes des détails d'implémentation ou de représentation liés à une machine ou un langage de programmation donné. Ces propriétés fonctionnelles abstraites sont stables et donc invariables lorsqu'on envisage de modifier certains modules utilisant ces T.As.

Les T.As. peuvent être réutilisés pour des applications très différentes de celles utilisées au départ. On parvient ainsi à assurer la réutilisabilité et l'exportabilité des composants logiciels. Un T.A est donc une abstraction de plusieurs structures de données différentes, par leurs détails d'implémentation mais ayant les mêmes propriétés spécifiques. En effet, toute structure de données peut être décrite à trois niveaux différents, allant du plus abstrait vers le plus concret.

Dans ce qui suit, on se base sur [Meyer.78] et [J. Uhl 91]. Dans ce dernier, on peut notamment trouver une librairie complète de T.As. Chaque T.A. étant défini et implémenté dans le langage ADA. Cette librairie est en fait une réadaptation d'une librairie de T.As. développée chez IBM par l'un des auteurs.

3.2 Spécification fonctionnelle

3.2.1 Les ensembles

Elle consiste en la définition d'un ensemble support de T.A. et d'un ensemble d'opérations permises sur ces T.As. ainsi que les propriétés de ces opérations et de leurs interrelations. Les ensembles supports de T.As. peuvent être définis à partir de T.As. plus élémentaires prédéfinis ou à partir d'autres T.A déjà définis. Il est clair que pour ces objets abstraits, les opérations et leurs interrelations sont totalement indépendantes des caractéristiques d'implémentation; ils sont donc stables et invariables.

3.2.2 Les opérations

Les opérations définies sur un T.A doivent être spécifiées de manière complète et précise. Pour toute opération associée à un T.A, on définit sa signature de la manière suivante :

nom_operation : Dom ------> Codom

(argument) (résultat)

* Dom : c'est le type de l'entrée de l'opération. Cet ensemble peut être simple ou structuré sous forme d'un produit cartésien de plusieurs types : $A_1, \ldots, A_n$.

*Codom : type de la sortie de l'opération; c'est souvent un type composé.

Remarque : Au moins l'un des deux ensembles, domaine ou codomaine, doit être le T.A lui-même.

On peut également spécifier ces opérations par la méthode "Pré-Post". Pour chaque opération, on doit spécifier sa précondition c'est-à-dire des propriétés des arguments en entrée et sa postcondition c'est-à-dire une propriété des arguments en sortie de l'opération. On peut éventuellement donner un invariant.

Une autre manière de spécifier les T.A., est la spécification algébrique. Dans cette méthode, on donne un certain nombre d'axiomes qui définissent toutes les lois de compositions possibles des opérations. Chaque axiome a la forme d'une équation. Celles-ci permettent de caractériser de manière précise et complète les caractéristiques des opérations et leurs interrelations. Ces propriétés sont exprimées en termes mathématiques, selon un schéma prédéfini, dans l'espoir de favoriser l'élaboration de spécifications précises, complètes et aisément compréhensibles. A chaque T.A sont associées trois types d'opérations:

3.2.2.1 Les constructeurs de T.A.

Ce sont des opérations qui permettent de reconstruire n'importe quelle opération sur le T.A par des compositions successives. Ils sont de deux types:

* élémentaires : indéfinissables en termes d'autres constructeurs.

* non-élémentaires : composites.

Exemples :

. INSERT(e, s)

Où e est un élément d'un certain type et s une séquence d'éléments de même type. Cette opération permet d'insérer l'élément e dans la séquence s. C'est un constructeur élémentaire.

. CONCAT(s_1, s_2)

Où s_1, s_2 sont deux séquences d'éléments du même type. Cette opération permet de concaténer les deux séquences. C'est un constructeur non élémentaire car il peut s'exprimer en fonction de plusieurs opérations INSERT successives.

Remarque :

Pour de telles opérations, le T.A apparait à droite de la flèche de la signature.

3.2.2.2 Les opérations d'accès au T.A.

Ce sont des opérations qui permettent d'accéder au contenu des structures existantes. Elles sont telles que le T.A apparait seulement à droite de la flèche de la signature du T.A.

3.2.2.3 Les opérations de modification

Elles sont telles que le T.A apparaisse à gauche et à droite de la flèche de la signature. Elles permettent de créer de nouveaux objets appartenant au T.A à partir d'autres objets de ce T.A. ayant déjà été créé (et d'autres éléments éventuels).

Exemple: REMOVE(e, s).

Où e et s sont respectivement un élément et une séquence d'éléments de même type. Cette opération permet de retirer l'élément e de la séquence s.

Exemples :

Nous proposons d'illustrer ces concepts en définissant le T.A. PILE .

1) Une pile est un ensemble formé d'un nombre variables, éventuellement nul, de données appartenant à un certain type et sur lesquels on peut effectuer les opérations suivantes:

. Ajout d'un élément en tête de la structure.

. Test déterminant si la pile est vide.

. Consultation de la dernière donnée ajoutée et non supprimée depuis, s'il y en a une.

. Suppression de la dernière donnée ajoutée et non supprimée depuis, s'il y en a une .

2) Spécification fonctionnelle.

Le type $PILE_T$ ou pile d'objets de type T est caractérisé par les opérations suivantes:

.) CREATE : $\emptyset$ ---------> $PILE_T$

domaine = $\emptyset$ = des occurrences vides.

codomaine = le type abstrait $PILE_T$ qu'on définit.

.) PUSH : $PILE_T \times T$ ---------> $PILE_T$

cette opération permet d'empiler un élément de type T.

.) TOP : $PILE_T$: ----------> $T \cup$ UNDEFINEDELEM.

cette opération fournit l'élément sommet de la pile ou UNDEFINEDELEM qui est une valeur retournée par l'opération lorsque la pile est vide. D'une certaine manière, on peut considérer que toute pile contient l'élément UNDEFINEDELEM comme dernier élément.

.) POP : $PILE_T$ ---------> $PILE_T \cup$ UNDEFINEDSTACK

qui fournit une nouvelle pile obtenue par retrait de l'élément au sommet de la pile. Comme pour TOP, lorsque la pile est vide, on retourne la valeur UNDEFINEDSTACK.

.) EMPTY-STACK : $PILE_T$ --------> BOOLEAN

c'est l'opération qui teste si la pile est vide et retourne la valeur VRAI ou FAUX selon le cas.

3) Axiomes et propriétés des opérations :

Pour tout élément s de $PILE_T$ et tout élément t de T, on a les propriétés suivantes:

(A1) : EMPTY-STACK(CREATE) = TRUE .

(A2) : EMPTY-STACK(PUSH(s, t)) = FALSE .

Si la dernière opération est un ajout d'élément au sommet de la pile, alors celle-ci n'est pas vide.

(A3) : POP(PUSH(s, t)) = s

(A4) : TOP(CREATE) = UNDEFINEDELEM.

(A5) : POP(CREATE) = UNDEFINEDSTACK.

(A6) : TOP(PUSH(s, t)) = e.

3.3 Descriptions logiques

Notons tout de suite que la description logique décrite ci-après pour les types abstraits ne doit pas être confondue avec la notion de description logique énoncée dans la section traitant de la programmation logique.

La description logique d'un T.A est le deuxième niveau de considération des T.A. Etant donné un T.A. dont la spécification fonctionnelle est connue, sa description logique est un moyen de définir ce T.A. par l'intermédiaire de types abstraits élémentaires considérés comme prédéfinis et / ou de types ayant déjà été créés et éventuellement ce T.A. lui-même (définition récursive). La description logique d'un T.A inclut aussi l'association d'une procédure à chaque opération permise sur le T.A.

Il existe trois manières de construire une description d'un T.A. à partir d'autres T.A. déjà "connus": la *juxtaposition*, la *séparation de cas* et l'*énumération*.

3.3.1 Juxtaposition

Avec ce moyen de construction de T.As., on considère qu'un T.A. est une séquence ou juxtaposition de plusieurs autres T.As. connus. Donnons nous un exemple:

Pour une pile, on considère que l'élément sommet joue un rôle bien particulier: c'est le seul élément qu'on peut manipuler. Le reste des éléments, appelé corps de la pile est lui-même une pile. Donc, on peut considérer qu'une pile est constituée d'une pile et d'un élément sommet. Si les éléments de la pile sont de type T, alors le T.A est la juxtaposition du type T et du type $PILE_T$ lui-même. On a ici un cas de récursion, puisque le type $PILE_T$ est constitué de lui-même et d'autres types.

Aux opérations fonctionnelles définies ci-dessus pour le type $PILE_T$, on va associer des procédures qui ont les mêmes propriétés que ces opérations:

. procédure CREATE : $PILE_T$

CREATE <--- VIDE.

. procédure PUSH : $PILE_T$ (x : T, p: $PILE_T$)

PUSH <--- $PILE_T$ (x, p).

. procédure TOP : T(p :$PILE_T$)

Si p est VIDE alors TOP <--- UNDEFINEDELEM

sinon TOP <--- p \ corps(p).

. procédure POP : $PILE_T$ (p : $PILE_T$)

POP <--- corps (p).

. procédure EMPTY-STACK : BOOLEAN (p : $PILE_T$)

EMPTY-STACK <--- (p = VIDE).

3.3.2 Séparation de cas

L'autre moyen de composition possible est la séparation de cas. Supposons que le T.A. T, soit le produit cartésien de plusieurs T.As. $T_1 \times T_2 \times ... T_r$. Ici, On considère que l'un au moins des types T_i est selon le cas soit l'un parmi les types $T_{i1}, T_{i2}, ..., T_{iq}$

Exemple : $PILE_T$ = VIDE | PILE_NON_VIDE.

Où PILE_NON_VIDE = $T \times PILE_T$

Donc ici, on distingue deux cas possibles : celui où la pile est vide et celui où la pile est constituée d'un élément sommet et du reste de la pile qui est éventuellement une pile vide.

3.3.3 Enumération

Dans certains cas où le nombre d'occurrences possibles d'un T.A. est très limité, on peut construire ce T.A. à partir de ces occurrences considérées comme des constantes.

Exemple :

. CHIFFRE = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9') .

. ETAT_CIVIL = ('CELIBATAIRE', 'MARIE', 'DIVORCE', 'VEUF').

Plus généralement, la description logique d'un T.A. T a la forme générale suivante:

T = ($const_1$, $const_2$, ..., $const_n$).

Où $const_1$, $const_2$, ..., $const_n$ sont des constantes de même type.

3.4 Représentation physique

La représentation physique d'un type abstrait est l'implémentation de ce T.A. pour une machine donnée et un langage de programmation donné. Pour cela, on doit représenter les occurrences du T.A. ainsi que les procédures associées aux opérations fonctionnelles par des sous-programmes écrits dans un langage de programmation donné. Il s'agit donc d'une définition interne.

Remarque :

Pour une spécification fonctionnelle donnée, il existe en général plusieurs descriptions logiques correspondantes qui pourront elles-mêmes être réalisées par plusieurs représentations physiques. On doit cependant s'assurer que chaque niveau représente bien la décomposition du niveau immédiatement supérieur. En d'autres termes, l'analyse au niveau fonctionnel doit correspondre à la description logique, elle-même compatible avec la représentation physique.

Exemple :

Une pile peut être implémentée par un tableau indicé ou par une liste chaînée. Une liste peut être implémentée par un chaînage linéaire ou circulaire des éléments. Chaque élément de cette liste chaînée peut avoir un pointeur vers le suivant ou deux pointeurs l'un vers le suivant et l'autre vers le précédent. Chaque élément peut aussi avoir soit un pointeur vers le premier élément de la liste soit deux pointeurs, l'un vers le premier et l'autre vers le dernier élément de la liste. Avec ces quatre implémentations possibles, on a $2^4 = 16$ implémentations possibles. Ici on doit choisir la bonne implémentation en raisonnant en terme de simplicité de représentation et de besoin d'efficacité. Ces considérations ne sont cependant pas le propos qui nous intéresse ici.

Remarque

Dans la pratique, il peut s'avérer que la démarche de construction de T.As. introduite ici et qui implique une démarche top-down ne soit pas réaliste. En effet, on est souvent amené à faire une démarche à la fois top-down et botom-up. Certains choix de représentation physique peuvent parfois imposer des retours en arrière qui peuvent altérer les spécifications fonctionnelles. Par ailleurs, on peut également considérer plusieurs niveaux d'abstraction où on affine progressivement les concepts jusqu'à atteindre finalement les objets. La démarche exposée ici se limite cependant aux trois niveaux d'abstraction, à savoir la spécification fonctionnelle, la description logique et la représentation physique.

4 Un essai d'intégration

4.1 Introduction

Dans les deux paragraphes précédents, nous avons défini les deux notions de programmation logique et de types abstraits. Dans la présente section, on va montrer comment il est possible d'intégrer ces deux notions en définissant une extension du langage Prolog (appelée A-Prolog pour *Abstract* Prolog) qui supporte la notion de type abstrait. On va en effet montrer qu'il est possible de représenter les opérations d'un type abstrait par des relations. Conformément à l'aspect informel de cette première partie, on va se limiter ici à des explications plutôt intuitives. Dans la seconde partie, on décrira les choses de manière plus formelle en définissant la syntaxe et la sémantique de A-Prolog. Dans la section consacrée aux T.A. nous avons essayé de définir cette notion de façon générale sans avoir en tête une application quelconque de cette notion. Dans le contexte qui nous intéresse ici, on va restreindre cette notion en considérant qu'un T.A. est simplement constitué d'un ensemble d'objets abstraits et d'opérations de base sur ces objets abstraits appelées dans ce qui suit des *primitives*. Il en résulte qu'une primitive sera décrite par une relation.

4.2 Représentation d'une opération par une relation

Les opérations sur un T.A. sont habituellement définies par des fonctions de la forme:

$$op : D \text{ -------> } CD$$

où D et CD désignent respectivement le domaine et le codomaine de l'opération. Dans le contexte qui nous intéresse ici, on va représenter ces opérations par des relations définies sur le produit cartésien $D \times CD$. Cette transformation est possible puisqu'on peut toujours associer à l'opération op une relation $\mathfrak{R}$ définie sur le produit cartésien $D \times CD$ telle que :

$$\forall (x, y) \in D \times CD \;\; y = op(x) \Leftrightarrow \mathfrak{R}(x, y).$$

Réciproquement, étant donnée une relation $\mathfrak{R}$ définie sur un produit cartésien A, avec $A = E \times F$, et de graphe $G \subseteq E \times F$, on peut lui associer la fonction f définie par:

$$f : A \text{ --------> } BOOLEEN = \{ VRAI, FAUX \}$$

$$f(x) = VRAI \text{ si } x \in G$$

$$= FAUX \text{ sinon.}$$

Cette fonction s'appelle l'*indicatrice* de la relation $\mathfrak{R}$. On obtient une correspondance biunivoque entre les fonctions et les relations.

Exemple:

Soit l'opération

PUSH : T $\times$PILE$_T$ ---------> PILE$_T$

on peut lui associer la relation $\Re$_PUSH telle que :

$\forall$ (e, s) $\in$ T $\times$ PILE$_T$ on ait : snew = PUSH(e, s) $\Leftrightarrow$ $\Re$_PUSH(e, s, snew)

où snew est la pile obtenue en ajoutant e au sommet de la pile e.

L'intérêt de cette transformation est de pouvoir utiliser les descriptions logiques en vue de définir une relation donnée en termes d'autres relations; celles-ci pouvant être soit des relations prédéfinies, soit des relations qu'on aura déjà définit quelque part et éventuellement en fonction de la relation qu'on est en train de définir. On permet ainsi des définitions récursives. Justement, un programme écrit dans le langage A-Prolog qu'on va définir complètement par la suite sera un ensemble de *définitions de relations.*

Une relation peut être décrite à deux niveaux: le *niveau déclaratif* et le *niveau procédural*:

4.3 Niveau déclaratif: le langage AD-log

Définition: Une *définition de relation* de prédicat p est une formule de la forme :

$p(x_1 : t_1, \ldots, x_n : t_n) \Leftrightarrow C_1 \vee \ldots \vee C_n$. (1)

où . C_i = Pref : $L_{i1} \wedge \ldots \wedge L_{iqi}$ $i = 1, \ldots, q$

. Pref est le préfixe de la conjonction. Ce préfixe a la forme :

$\exists\, y_1 \in t_1, \ldots, y_r \in t_n$

avec $\{ y_1, \ldots, y_r \} \cap \{ x_1, \ldots, x_n \} = \varnothing$.

les L_i sont des littéraux pour $i = 1, \ldots, q_i$.

les t_i sont des noms de types abstraits pour $i = 1, \ldots, n$

Les variables $x_1, \ldots, x_n$ sont quantifiées universellement. Une définition de relation est donc une description logique où l'on introduit explicitement la spécification du type abstrait auquel appartient chaque variable.

Définition : Une *déclaration de primitive* est une entête de la forme:

$$p(x_1 : t_1, \ldots, x_n : t_n).$$

qui spécifie le nom de la primitive et le type abstrait auquel appartient chaque paramètre.

Au niveau déclaratif, on considère que (1) est une description logique où l'ordre des littéraux dans une conjonction n'a aucune importance. De même, l'ordre des conjonctions dans une disjonction :

$$C_1 \vee \ldots \vee C_n$$

n'a pas non plus d'importance. Ceci découle du fait qu'en sémantique déclarative, on ne s'intéresse qu'à la valeur de vérité de la formule (1) et non à la façon dont l'interpréteur va "exécuter" les appels de procédure. La sémantique déclarative d'une définition de relation est l'ensemble des n-uplet $(x_1, \ldots, x_n)$ ground tels que $p(x_1, \ldots, x_n)$ soit vraie par rapport à la spécification de p; autrement dit le n-uplet $(x_1, \ldots, x_n)$ appartient au domaine de la relation décrite dans la spécification de p. On peut trouver dans [Y. Deville 90] une méthode générale de construction de descriptions logiques à partir d'une spécification. On y présente également des heuristiques pour faciliter cette construction. La notion de correction d'une description logique par rapport à sa spécification y est également définie, formalisée et développée.

Considérons le T.A. $PILE_T$ où T est un T.A. quelconque. Ecrivons la description logique de la relation length (s : $PILE_T$, l : integer).

Length (s : $PILE_T$; l : integer) <=>

EMPTY - STACK (s) $\wedge$ null (l)

$\vee$ $\exists$ s_1 : $PILE_T$, e_1 : T : NOT (EMPTY - STACK(s))

$\wedge$ PUSH (e_1 , s_1 , s)

$\wedge$ Length (s_1 , l_1)

$\wedge$ SUCC (l_1 , l)

La notion de directionalité pour les procédures Prolog doit être étendue aux définitions de relation. En effet, pour les T.As., on ne peut parler du degré d'instanciation d'une variable. Ou bien une occurrence de ce T.A. est suffisamment instanciée pour représenter une valeur de ce T.A., ou bien elle ne l'est pas et dans ce cas on n'a pas de valeur. Pour tenir compte de ce comportement, on définit la notion de *directionalité abstraite* d'une définition de relation.

Directionalité abstraite

Une directionalité abstraite associée à une définition de relation ou une déclaration de primitive est une description des utilisations possibles de cette relation. Elle décrit la forme des paramètres actuels avant et après exécution d'une opération définie sur un T.A.

Définition :

Une directionalité abstraite associée à une définition de relation est un n-uplet dont chaque composant peut prendre l'une des trois valeurs suivantes: input, destroyed input, output.

. Input signifie que la variable correspondante doit être instanciée à une valeur au moment de l'appel et que cette valeur demeurera inchangée après cet appel.

. Destroyed input signifie que la variable correspondante doit être instanciée à une valeur au moment de l'appel mais que cette valeur peut être perdue (réutilisation de la structure) après cet appel.

. Output signifie que la valeur de cette variable peut être créée.

Dans le point qui suit concernant le niveau procédural,on donnera un exemple d'instanciation destructive.

Définition : Un programme AD-log est un ensemble de définitions de relations, de déclarations de primitives et de spécifications de directionalités abstraites pour chaque relation définie et chaque primitive déclarée. Dans les conjonctions figurant dans les seconds membres de ces définitions de relations, on ne suppose aucun ordre sur les atomes; c'est le système qui se charge de permuter cet ordre pour obtenir des conjonctions correctes par rapport aux directionalités. Nous préciserons la signification exacte de cela dans la partie formelle.

4.4 Niveau procédural: Le langage A-Prolog

Il existe une différence capitale entre une variable en programmation logique et une variable en programmation impérative. La première correspond à un objet non spécifié mais unique, tandis que la seconde correspondent à une case mémoire où on peut ranger des valeurs. Il faut donc les traiter différemment.

En Prolog, lorsqu'il s'agit de modifier un terme ground, la seule façon de faire est de créer un terme qui sera une copie totale ou partielle du premier. Donc, on est obligé de recopier totalement ou partiellement une structure avant de pouvoir la modifier. D'autre part, une procédure peut rendre inutilisable l'un ou plusieurs de ses paramètres. Ces problèmes sont dus au fait que les objets abstraits sont représentés par des termes contenant des variables. Certaines de ces variables peuvent être instanciées (totalement ou partiellement) lorsqu'elles sont unifiées avec des termes ground (intanciation destructive). Il en résulte que le terme contenant ces variables est modifié ou détruit. Lorsque ce paramètre est référencé par un appel ultérieur, il faut le recopier avant de le passer en argument à l'appel qui va le détruire.

Exemple

Considérons le terme Prolog représentant une file d'attente (une implémentation complète du T.A. FILE est présentée au point 4.7.2 de la seconde partie):

file(_D,_F)

où D et F sont deux listes Prolog telles que les éléments de la liste différence D\F soient les membres de la file (par ex. file([e_1, e_2, e_3|_V],_V) est une liste à trois éléments e_1, e_2 et e_3; la file vide est représentée par le terme file(_V,_V)). C'est donc un terme contenant une variable.

Considérons l'opération d'ajout d'un élément en fin de file. Elle est implémentée par la procédure Prolog suivante:

Ajout($_f_1$, $_f_2$, _X) :- $_f_1$ = file($_D_1$, $_F_1$),

$_F_1$ = [_X|$_F_2$],

$_f_2$ = file($_D_1$, $_F_2$).

(Où $_f_2$ est la file $_f_1$ à laquelle on a rajouté l'élément _X à la fin. Par exemple si $_f_1$ = file([e_1,e_2,e_3|_V],_V) et _X = e_4 alors $_f_2$ = file([e_1, e_2, e_3, e_4|$_V_1$],$_V_1$)).

Cette procédure détruit son premier argument. En effet, si on a l'exécution:

?- $_f_1$ = file([e_1,e_2,e_3|_V],_V), Ajout($_f_1$,$_f_2$,e_4).

$_f_1$ = file([e_1,e_2,e_3,e_4 | _65],[e_4 | _65])

_V = [e_4 | _43]

$_f_2$ = file([e_1,e_2,e_3,e_4 | _43],_43)

Yes

?-

alors, on voit bien que la valeur de $_f_1$ a été modifiée par la procédure et donc $_f_1$ est devenu inutilisable de telle façon que le but (où on fait deux opérations d'ajout successives) ci-dessous échouera:

?- $_f_1$ = file([e_1,e_2,e_3|_V],_V),Ajout($_f_1$,$_f_2$,e_4), Ajout($_f_1$,$_f_3$,e7).

No

Car, on réutilise la variable _f1, dans la seconde opération d'ajout, alors qu'elle est détruite par la première. Pour remédier à ce problème, il faut donc faire une copie $\_f_4$, de la variable $\_f_1$ avant de la passer à un appel ultérieur et de remplacer la référence à $\_f_1$ dans cet appel par sa copie $\_f_4$. $\_f_4$ étant en fait un terme représentant la même file que _f1, mais à la différence de $\_f_1$, il contient une variable non instanciée. On a alors l'exécution:

?- $\_f_1$ = file([e_1,e_2,e_3|_V],_V),$\_f_4$ = file([e_1,e_2,e_3|$\_V_1$],$\_V_1$),Ajout($\_f_1$,$\_f_2$,e_4),
Ajout($\_f_4$,$\_f_3$,e_7).

$\_f_1$ = file([e_1,e_2,e_3,e_4 | _117],[e_4 | _117])

_V = [e_4 | _117]

$\_f_4$ = file([e_1,e_2,e_3,e_7 | _123],[e7 | _123])

$\_V_1$ = [e_7 | _123]

$\_f_2$ = file([e_1,e_2,e_3,e_4 | _117],_117)

$\_f_3$ = file([e_1,e_2,e_3,e_7 | _123],_123)

Yes

?-

Un peu plus généralement, lorsqu'on a deux appels successifs:

p(x) , q(x, y)

où on suppose que p détruit son argument et que q doit avoir son premier argument instancié avant appel, alors, la variable x ne peut pas être référencée par la procédure q car x est rendue inutilisable par q. Il faut donc faire une copie de la variable x avant de la passer à p et de remplacer la référence à x dans l'appel q par sa copie. On obtient :

égal(x, x_1) , p(x) , q(x_1, y).

où égal est supposé être une opération qui recopie la valeur de x dans la variable x_1. Ces transformations seront détaillées et considérées de manière plus générale dans la partie formelle.

Pour rendre compte de ce comportement, on définit la notion d'*état des variables* qui permet de classifier les différents arguments d'une procédure.

Définition :

Soit p (x_1 , ..., x_n) un appel de procédure. Avant et après exécution de cet appel, un paramètre actuel peut prendre 3 formes possibles :

* value: lorsque ce paramètre a reçu une valeur.

* destroyed : lorsque ce paramètre a reçu une valeur, mais celle-ci peut être détruite.

* variable : lorsque ce paramètre n'a pas reçu de valeur.

Soit $L_1, \ldots, L_q$ une conjonction de littéraux apparaissant dans le second membre d'une définition de relation. En sémantique procédurale, cette conjonction est considérée comme une suite ordonnée d'appels de procédure. Il en résulte que lorsqu'on a une définition de relation de la forme :

$$p(x_1 : t_1, \ldots, x_n : t_n) \Leftrightarrow \text{ConjPref}_1 \vee \text{ConjPref}_2 \vee \ldots \vee \text{ConjPref}_n.$$

sa sémantique procédurale est la suivante :

Pour exécuter l'appel de procédure $p(X_1, \ldots, X_n)$, on doit exécuter la suite des appels ConjPref_1 ou ... ou la suite des appels ConjPref_n. Où ConjPref_i désigne une conjonction de littéraux de la forme

$$\text{Pref}_i : L_{i1}, \ldots, L_{in} .$$

Définition :

Un *état des variables* est un ensemble $vs = \{\langle x_1, d_1\rangle, \ldots, \langle x_n, d_n\rangle\}$

où $d_i \in \{$ value , destroyed , variable $\}$.

Définition :

Un programme A-Prolog est un ensemble de définitions de relations, de déclarations de primitives et de spécifications de directionalités abstraites pour chaque relation définie et chaque primitive déclarée. En plus, on suppose que dans chaque conjonction figurant dans le second membre d'une définition de relation, les directionalités abstraites de chaque appel sont respectées. Nous préciserons plus loin à quoi cela correspond. Donc, à ce niveau, on suppose que l'utilisateur a pris la peine de vérifier lui-même que les directionalités abstraites ont été bien respectées. Cependant, le système (le traducteur) se chargera d'intercaler, au bon endroit dans les conjonctions, les procédures de recopie.

5 Implémentation des langages

5.1 Introduction

Après avoir défini les deux langages AD-log et A-Prolog, on va voir ici comment les implémenter en Prolog. Pour cela, il suffit de montrer comment il est possible de :

. Représenter les types abstraits, c'est-à-dire :

. les occurrences d'un T.A.

. les opérations sur ce T.A.

. Traduire un programme AD-log en Prolog.

. Traduire un programme A-Prolog en Prolog.

Après une analyse syntaxique et sémantique du programme A-Prolog ou AD-log du programme à traduire, on a besoin de transformer ce programme en remédiant au problème de l'instanciation destructive introduit à la section précédente. Dans une conjonction d'appels de procédures, on doit notamment vérifier qu'une procédure n'utilise pas un argument détruit par un appel antérieur. Nous montrerons comment s'effectue cette transformation. Une fois que nous disposons de conjonctions correctes, leur traduction en code Prolog consiste à faire des remplacements: il suffira de remplacer les appels de primitives par les appels de procédure Prolog qui les implémentent.

Enfin, nous définirons le langage de définition des T.A. en précisant la manière dont s'effectue la mise-à-jour de la base de données des T.As. et la génération de code Prolog correspondant aux programmes AD-log et A-Prolog

5.2 Représentation d'un T.A.

5.2.1 Les ensembles

A priori, une occurrence d'un T.A. peut être représentée par un terme Prolog quelconque. Le choix de représentation doit évidemment faciliter l'implémentation des opérations primitives qui vont manipuler ces occurrences. Il incombe donc au programmeur de choisir la bonne implémentation des éléments d'un T.A.

5.2.2 Les opérations

Une opération primitive peut être représentée par une définition de relation dans le langage AD-log ou A-Prolog ou directement en Prolog en utilisant les techniques classiques de construction de programme Prolog qu'on peut trouver dans [Y. Deville 90], [A. O'Keefe 88-90] ou [E. Shapiro 86]. Une fois que l'on a des primitives correctes, on peut les réutiliser comme blocs de base pour définir des opérations de plus haut niveau en utilisant A-Prolog ou AD-log. Ces deux langages seront alors utilisés de manière réaliste et effective.

Remarque :

Il est en général difficile d'écrire une seule procédure Prolog qui respecte toutes les directionalités que l'on veut implémenter. Par conséquent, très souvent, à une même opération sur un T.A. donné, correspondra plusieurs procédures Prolog, chacune implémentant une directionalité différente associée à cette opération.

5.2.3 T.As. prédéfinis

Dans les deux extensions de Prolog à savoir A-Prolog et AD-log on va considérer que les termes A-Prolog sont aussi des termes AD-log et A-Prolog. De même, on considère que l'opération d'unification de deux termes est aussi une opération prédéfinie. Donc, les termes et l'unification des langages A-Prolog et AD-log seront directement implémentés en Prolog.

5.3 Traduction de A-Prolog et AD-log en Prolog

Soit un texte source d'un programme A-Prolog. La traduction de ce texte en code Prolog directement "exécutable" comporte les étapes suivantes:

5.3.1 Analyse syntaxique

Dans cette étape, on vérifie que le texte du programme respecte bien la syntaxe concrète du langage. Cette syntaxe sera donnée dans la seconde partie. L'analyse syntaxique prévoit aussi un module de gestion d'erreurs convivial qui permet d'aider l'utilisateur à trouver rapidement les erreurs éventuelles. En effet, ce module permet de repérer l'endroit exact où l'erreur a eu lieu dans le texte et informe l'utilisateur du type d'erreur rencontrée.

5.3.2 Analyse sémantique

Dans l'analyse sémantique, on fait d'autres vérifications. On vérifie notamment que le typage des variables est correct, que les variables sont déclarées avant d'être passées en argument et que les types des variables sont des types connus.

Chaque type abstrait (comme nous l'expliquerons bientôt) est en effet enregistré dans une base de données qui maintient toutes les informations sur tous les types abstraits connus du système. Donc lorsque l'analyseur sémantique rencontre une référence à un T.A., il doit interroger cette base de données pour vérifier que ce T.A. existe bien.

5.3.3 Analyse data-flow

Une analyse data-flow appliquée à une conjonction

$$A_1 , \ldots , A_n$$

permet de détecter deux types d'anomalies principales :

. La référence dans un appel donné à une variable rendue inutilisable par un appel précédent.

. Deux appels successifs ont des directionalités abstraites incompatibles i.e. la forme des variables (value , variable, destroyed) à la sortie du premier appel n'est pas compatible avec la directionalité abstraite de l'appel suivant (le sens exact de ceci sera précisé dans la seconde partie).

Dans le premier cas, la conjonction :

$$A_1 , \ldots , A_n . \qquad (1)$$

doit être transformée par :

- L'ajout de procédures de recopie des variables détruites.
- Le remplacement des références à ces variables en références à leurs copies respectives.

Dans le second cas, la conjonction (1) doit être transformée en soit :

- Une conjonction où l'on a, simplement intercalé des procédures de recopie de valeurs aux atomes de la conjonction (1). On garde donc l'ordre des atomes dans (1). Cette transformation est effectuée pour des programmes A-Prolog.

- Une permutation de cette conjonction telle que les directionalités abstraites des atomes de la conjonction obtenue soient compatibles. Cette transformation est effectuée pour des programmes AD-log.

Dans ce dernier cas, il existe en général plusieurs permutations à cette conjonction qui répondent à cette propriété (à savoir, la compatibilité des directionalités). Ces permutations seront classées selon un certain critère qui sera discuté plus tard. On peut alors choisir la meilleure permutation selon ce critère.

5.3.3.1 AD-log

On a vu que les programmes AD-log étaient des programmes pour lesquels on ne suppose aucune garantie sur la compatibilité des directionalités abstraites dans les conjonctions. L'utilisateur écrit les définitions de relations comme des descriptions logiques.

Donc, après une analyse syntaxique et sémantique d'un programme AD-log, la troisième phase dans la traduction de ce programme est de transformer les conjonctions en des conjonctions correctes par l'analyse data-flow. Cette transformation peut impliquer une permutation de l'ordre des conjonctions. Donc ici, on suppose que cet ordre n'est pas fixé.

5.3.3.2 A-Prolog

Dans la traduction d'un programme A-Prolog, on suppose que l'utilisateur a fait les vérifications nécessaires pour garantir que les conjonctions soient correctes. L'analyseur data-flow vérifie simplement qu'il n' y a pas de référence à des variables rendues inutilisables.Il se charge éventuellement d'insérer dans la conjonction les opérations de recopie aux bons endroits sans permuter l'ordre des conjonctions. On suppose donc ici, que l'ordre des conjonctions est fixe..

5.3.4 Génération de code Prolog exécutable

C'est la dernière phase de la traduction. Cette phase se résume par les étapes suivantes :

. Eclatement des disjonctions en clauses Prolog.

. Suppression des noms des T.As. dans les entêtes de relations.

. Suppression des préfixes dans les conjonctions.

. Substitution aux opérations primitives sur les T.As. les procédures Prolog qui les implémentent.

Les quatre étapes de la traduction d'un programme A-Prolog ou AD-log ne diffèrent donc qu'au niveau de l'analyse data-flow; le reste des étapes est quasi identique.

Ces quatre étapes, sont entièrement automatisées et prises en charge par le système. Une définition plus complète sera donnée dans la deuxième partie, notamment en ce qui concerne la notion d'analyse data-flow.

5.4 Langage de définition de T.A.

5.4.1 Introduction

Dans une section précédente, on a considéré qu'un T.A. pouvait être représenté par un ensemble de valeurs et d'opérations sur ces valeurs. Nous avons vu aussi que le système sera composé d'une base de données de T.A. qui gère les informations utiles sur les T.A connus du système. Cette base de données est en fait une librairie que l'on peut étendre au gré du besoin en rajoutant progressivement des nouveaux T.As. Etant donné qu'on peut définir un nouveau T.A. en fonction d'autres T.As. déjà définis, il importe de construire bien soigneusement des T.As. primitifs qui vont servir de blocs de base pour la construction de T.As. plus élaborés et plus compliqués. Il en résulte que l'utilisateur doit disposer d'un outil qui lui permette de définir un nouveau T.A. et l'"injecter" dans la base de données: c'est le langage de définition des T.As.

Dans la suite, on va donner une description informelle de ce langage. La syntaxe et la sémantique seront présentées dans la partie formelle qui suit. On donnera également quelques exemples d'illustration.

5.4.2 Description informelle

La description informelle suivante est empruntée à [B. Le Charlier 90].

- Un programme écrit dans ce langage permet de définir un T.A. Ce programme est appelé une *définition de T.A.*
- Un T.A. possède un nom qui l'identifie parmi les autres T.As. présents dans la base de données des T.As.
- La définition d'un T.A. peut utiliser d'autres T.As. Dans ce cas, il faut spécifier dans le programme la liste des T.As. utilisés.
- Une définition de T.As. comporte une *partie formelle* et une *partie informelle.*

- La partie informelle comporte la définition de l'ensemble des valeurs du T.A., dans un langage laissé au choix de l'utilisateur, et la définition des conventions de représentations d'une valeur quelconque du type au moyen, soit de termes Prolog, soit de valeurs des types utilisés par la définition. Cette partie comporte aussi la spécification des relations implémentées par les différentes opérations du type. La partie informelle n'occupe pas une portion contiguë de la définition mais est disséminée dans celle-ci conformément à la syntaxe décrite plus loin.

- La partie formelle comporte les éléments suivants:

  . Le nom du T.A. à définir.

  . le nom des T.As. utilisés s'il y en a.

  . la définition des opérations du type.

- Parmi les opérations définies sur un T.A., on distingue certaines opérations standards obligatoirement présentes et d'autres, optionnelles:

- Les opérations obligatoires sont :

  . le test d'égalité de deux valeurs de même type.

  . la copie d'une valeur dans une variable de même type.

  . la génération d'une valeur d'un certain type.

En fait, pour les deux premières il s'agira de la même opération (égalité) avec des directionalités abstraites différentes. Ces opérations seront notamment utilisées par l'analyseur data-flow pour insérer là où il faut les procédures de recopie et de test d'égalité dans les conjonctions.

Parmi ces opérations obligatoires, on a aussi l'opération qui génère une valeur quelconque d'un type donné. Cette opération sera également utilisée par l'analyseur data-flow.

- Les opérations optionnelles sont la lecture d'une valeur, l'écriture d'une valeur, la conversion d'un terme Prolog en une valeur du type et la conversion inverse.

Ces opérations se justifient du fait que l'utilisateur d'un T.A. ne peut pas connaître (ou en tout cas utiliser) la représentation des valeurs de ce type, mais il pourrait vouloir lire ou imprimer des valeurs.

Ces opérations sont optionnelles parce que certains types peuvent servir uniquement de façon "interne" et dans ce cas, la visualisation de leurs valeurs est inutile (sauf pour le "debugging").

- La définition d'une opération comporte les éléments suivants :
    - son nom qui est un identificateur.
    - son entête qui est une liste de paramètres formels de l'opération ainsi que leurs types respectifs. Ces types doivent être connus ou en cours de définition.
    - la spécification de la relation implémentée :
        - s'il s'agit d'une opération sans effet de bord, alors il faut spécifier les directionalités abstraites possibles.
        - s'il s'agit d'une opération avec effet de bord, alors il faut donner une description de son effet.
    - l'implémentation de l'opération.
- L 'implémentation d'une opération pour chaque directionalité une procédure de calcul de cette opération selon une des modalités décrites ci-dessous :

1) Si le type abstrait en question est directement implémenté en Prolog, alors on a deux possibilités :
    a) Spécifier une procédure BIM_Prolog sans aucune restriction : Cette procédure sera un code Prolog à part entière.
    b) Spécifier un algorithme logique, augmenté des indications permettant sa transformation automatique en code BIM_Prolog par le traducteur du langage. Ces indications spécifient, comme on l'a dit dans la définition d'un programme A-Prolog, les directionalités abstraites des primitives utilisées dans les définitions de relations de ce programme.
2) Si le type abstrait en question utilise d'autres T.As., alors chaque opération est implémentée sous forme d'une définition de relation. Ici aussi on a deux possibilités :
    a) Soit cette définition de relation sera écrite dans le langage A-Prolog, auquel cas, l'ordre des conjonctions et des littéraux dans une conjonction est fixé. Dans ce cas, l'analyseur data-flow vérifiera simplement que les directionalités abstraites sont compatibles. Il placera éventuellement des procédures de recopie là où c'est nécessaire.

b) Soit cette définition de relation sera écrite dans le langage A-Prolog auquel cas, l'ordre des conjonctions et des littéraux n'est pas fixé. Dans ce cas, l'analyseur cherchera les meilleures permutations des littéraux et donnera les directionalités abstraites correspondant à ces permutations.

- Une définition de relation peut être récursive et utiliser des opérations du type en cours de définition. Ceci ne pose aucune difficulté. Il se peut que la définition de p pour une directionalité abstraite $adir_1$, utilise p avec une autre directionalité $adir_2$.
- La définition des opérations d'un type peut nécessiter des procédures (lorsque implémentée directement en Prolog) ou relations (lorsque implémentée en A-Prolog ou en AD-log) auxiliaires qui ne seront pas connues de l'extérieur ("non exportables").
- On distinguera les types à représentation ground des autres types, car pour ceux-là, le problème des variables détruites ne se pose pas. Un type abstrait à représentation ground est un type dont les occurrences sont toutes des termes ground i.e. ne comportent aucune variable.

5.4.3 Traduction du langage de définition de type abstrait

Le traducteur du langage de définition de types abstraits comporte les modules suivants:

5.4.3.1 L'analyseur syntaxique

Ce module reçoit un fichier source contenant la définition d'un type abstrait. Il vérifie que le texte du programme respecte bien la syntaxe concrète du langage qui sera donnée plus loin. Cette analyse syntaxique prévoit aussi un module de gestion d'erreurs. Ce module est d'ailleurs le même pour les trois langages : A-Prolog, AD-log et le langage de définition de types abstraits. En absence d'erreurs syntaxiques, l'analyseur syntaxique produit un terme Prolog représentant le programme. Ce terme sera exploité, ensuite, par l'analyseur.

5.4.3.2 L'analyseur sémantique

Ce module reçoit une représentation interne d'une la définition de type abstrait et vérifie que les règles sémantiques additionnelles suivantes ont bien été respectées:

- Lorsqu'on définit un nouveaux type abstrait, celui-ci ne doit pas être déjà présent dans la base de données des T.As.
- Lorsque la définition d'un T.A. utilise d'autres T.As., ceux-ci doivent être déjà présents dans la base de données des T.As.
- L'arité d'une opération doit être égale à la longueur de la liste des directionalités.
- Les paramètres qui figurent dans l'entête d'une opération doivent soit être déjà définis, soit être le type en cours de définition.
- Une opération primitive ne peut être déclarée plus d'une fois dans une définition de T.A.

Lorsqu'au moins une de ces règles n'est pas vérifiée par la définition du T.A., le module de gestion des erreurs affiche un message indiquant le type d'erreur ainsi que l'endroit où celle-ci a eu lieu.

5.4.3.3 Mise-à-jour de la base de données

Lorsque la définition d'un T.A. est admise par l'analyseur syntaxique et sémantique, le type sera intégré à la base de données. Cette mise-à-jour implique l'enregistrement de toutes les informations utiles sur le type. Ces informations sont les suivantes:

. Le nom du type abstrait.

. La liste de tous les types abstraits utilisés.

. Pour chaque opération sur ce T.A. :

  . Le nom de cette opération.

  . son arité.

  . La liste reprenant le type de chaque paramètre de l'opération.

  . Pour chaque directionalité abstraite :

    . Le nom de la procédure ou de la relation qui implémente cette opération pour cette directionalité.

    . Le nom du fichier qui contient le code BIM_Prolog de cette procédure si celle-ci a été implémentée directement en Prolog, ou le nom du fichier qui contient le code A-Prolog ou AD-Prolog si cette procédure a été implémentée par une définition de relation.

. Le nom du module qui contient toutes les procédures implémentant les opérations associées au type abstrait. Ce module a, pour des raisons de facilité d'impélmentation, le même nom que le T.A. en question.

La base de données des T.As. est d'ailleurs constituée d'une suite de "records", chaque record contenant les champs correspondant aux informations ci-dessus. Ce module permettra la création d'une base de données vide, l'ajout ou la suppression d'un T.A. et la modification d'un T.A. avec toutes les vérifications de cohérence qui s'en suivent.

III Présentation formelle et implémentation

1 Introduction

Dans la partie précédente, nous avons essayé, après quelques rappels des principaux concepts de la programmation logique et de type abstrait, de fixer les objectifs du système et d'expliquer les idées générales de façon informelle. Nous avons ensuite montré comment on pouvait définir complètement (du moins pour les besoins de notre application) un T.A. par un ensemble d'éléments et d'opérations sur ces éléments. On a ensuite montré que ces opérations pouvaient être, soit implémentées par des procédures Prolog, soit par des définitions de relations. On a alors défini une extension à Prolog qui permet à l'utilisateur d'écrire ces définitions de relations. Celles-ci étant traduites en définitive en des programmes Prolog à part entière.

On a aussi défini un langage de définition de T.As. qui permet d'ajouter de nouveaux types à la base de données. Enfin, la structure de cette base a été analysée.

Dans cette seconde partie, on va raffiner les concepts introduits auparavant en adoptant une approche plus formelle. Pour chacun des trois langages, on définira, en effet, la syntaxe précise en BNF, la sémantique sera donnée de manière intuitive. On s'attachera également à donner des exemples théoriques et pratiques pour illustrer les concepts. On énoncera enfin les règles de traduction de ces trois langages.

2 Le langage A-Prolog

2.1 Introduction

Le langage A-Prolog, rappelons-le, est une extension de Prolog qui utilise la notion de T.A. Cette extension permet de pouvoir manipuler des structures de données qui n'existent pas en Prolog (rappelons que la seule structure de données disponible en Prolog est le terme sur lequel est définie l'opération de base qui est l'unification). Grâce à cette caractéristique, A-Prolog permet de gérer les données de façon autonome par rapport au traitement. De cette manière, on augmente la modularité du travail de développement logiciel. Donnons à présent la syntaxe concrète du langage.

2.2 Symboles de base du langage

L'ensemble des symboles de base du langage A-Prolog est défini ci-dessous par sa syntaxe BNF :

<symbole de base > ::= <variable > | <constante> | <symbole special>

<variable> ::= <underscore> <rest_var>

<rest_var> ::= <vide> | <alphanumerique> <rest_var>

<alphanumerique> ::= <lettre> <chiffre> | <underscore>

<lettre> := <lettre majuscule> | <lettre minuscule>

<chiffre> ::= 0 | 1 | 2 | 3 | 4 |5 | 6 | 7 | 8 | 9

<lettre minuscule> ::= a| b ... y | z

<lettre majuscule> ::= A | B ... | Y | Z

<underscore> ::= _

<constante> ::= <atome > | <entier>

<atome> ::= atome prolog

<entier> :: <chiffre> |<entier> <chiffre>

<symbole special> ::= + | * | - | / | (|) | > | > | : | ; | [|] | , | = | { | }

2.3 Syntaxe concrète du langage A-prolog

La syntaxe concrète du langage A-Prolog est définie ci-dessous en sa syntaxe BNF (tirée de [J.P. Hogne 90]):

<programme> ::= <composant> | <programme> <composant>

<composant> ::= <definition de relation > |

<declaration de primitive>

<declaraion de primitive> ::= <dec.de.prim>

< spécification de ad >

<dec.de.prim> ::= <primitive> <entete> |

<primitive> <entete> <identif_module>

<identif_module> ::= is <nom de primitive> . <nom_module>

<definition de relation> ::= <entête> <=> <disjonction>

< specification de ad >

<entete> ::= <nom de primitive ou de relation> |

<nom de primitive ou de relation> (<liste de variables et types>)

<disjonction> ::= <conjonction prefixee> <disjonction>

<conjonction prefixee> ::= <prefixe> : <conjonction>

<prefixe> ::= exist <liste de variables et de types>

<liste de variables et types> ::= <nom de variable> : <nom de type>|

<nom de variable> : <nom de type> ,

<liste de variables et types>

<nom de variable> ::= <identificateur>

<nom de type> ::= <identificateur>

<conjonction> ::= <literal> |

<literal> , <conjonction>

<literal> ::= <structure> |

not (<structure>)

<structure> ::= <unification> |

<appel de procedure>

<unification> ::= <terme> = <terme>

<appel de procedure> := <nom de primitive ou de relation> |

<nom de primitive ou de relation> (<liste de termes>)

<nom de primitive ou de relation> ::= <nom de primitive> |

<nom de relation>

< specification de ad > ::= directionality for

< nom de primitive ou de relation > Is

< liste de spéc.de.dir.abst >

< nom de primitive ou de relation > ::= <identificateur>

< liste de spec.de.dir.abst > ::= < dir.abst > { , < dir.abst > }

< dir.abst > ::= < mode d'appel > |

[< mode.d'appel > { , < mode d'appel > }]

< mode d'appel > ::= in | out | inout .

<nom de relation> ::= <identificateur>

<nom de primitive> ::= <identificateur>

<liste de termes> ::= <terme> |

<liste de termes> , <terme>

<terme> ::= terme prolog

2.4 Règle sémantiques

En plus de ces règle syntaxiques, un programme A-Prolog doit respecter certaines contraintes additionnelles. Commençons d'abord par quelques conventions terminologiques :

- La *table des procédures* d'un programme est constituée de l'union de sa *table des relations* et de sa *table de primitives.*
- La table de relations d'un programme est l'ensemble de toutes les entêtes apparaissant dans ses définitions de relations.
- La table des primitives est l'ensemble de toutes les entêtes apparaissant dans ses déclarations de primitives.
- Soient:
  - . R une définition de relation, de la forme :

    $p(x_1 : t_1 , \ldots , x_n : t_n) \Leftrightarrow CP_1 \vee \ldots \vee CP_n$.
  - . CP_j l'une des conjonctions préfixées de cette définition de la forme:

    exist $y_1 : u_1 , \ldots , y_n : u_n : L_1 \wedge \ldots \wedge L_q$.
  - . L_k l'un des littéraux de cette conjonction préfixée, contenant soit une unification, soit un appel de procédure, éventuellement suivi du symbole 'not' et soit z une variable apparaissant dans ce littéral :
- Si z appartient à $\{ y_1 , \ldots , y_n \}$, on dit que z est une variable *locale*. Soit alors i, l'indice qui lui correspond; on dit que le *type* de la variable z est u_i.
- Si z appartient à $\{ x_1 , \ldots , x_n \}$, on dit que z est un *paramètre*. Soit alors i, l'indice qui lui correspond; on dit que le type de la variable z est t_i.

Disposant de ces conventions terminologiques, on peut énoncer les règles suivantes:

- Soit P un programme. Soit A un appel de procédure quelconque apparaissant dans P de la forme:

$$p(x_1, \ldots, x_n).$$

La table des procédures de P doit contenir une entête E, de la forme:

$$p(y_1 : t_1, \ldots, y_n : t_n).$$

On dit alors que A référence une *procédure définie*, et que E est son entête correspondante.

- Soit P un Programme. Soit A un appel de procédure apparaissant dans P, et référençant une procédure définie. Si A est de la forme:

$$p(x_1, \ldots, x_n),$$

alors, pour tout $i \in [1..n]$, la variable x_i doit être soit un paramètre, soit une variable locale.

En outre, si E, l'entête correspondante de A, est de la forme:

$$p(y_1 : t_1, \ldots, y_n : t_n),$$

alors, pour tout $i \in [1..n]$, la variable x_i doit être de type t_i.

2.5 Sémantique opérationnelle du langage A-Prolog

On donnera ici la sémantique opérationnelle du langage i.e. un ensemble de règles qui précisent comment à partir d'un certain nombre de "valeurs" en entrée au programme, on obtient des résultats en sortie. Cette sémantique est très similaire à la sémantique procédurale que nous avons défini pour les langages logiques. On va en effet donner l'arbre de dérivation construit par l'interpréteur A-Prolog pour résoudre un but donné. Cette sémantique opérationnelle est empruntée à [J.P. Hogne. 90] où il expose les principes d'une démarche de programmation logique utilisant la notion de T.A.

2.5.1 Arbre de dérivation de A-Prolog

Soit P un programme, et B un but. L'exécution de P par rapport à B peut être décrite par l'arbre de dérivation que l'on définit ci-après :

- Chaque nœud de l'arbre est un but.

- Une substitution est attachée à chaque arc.

- Le nœud racine est le but initial B.

- Soit N un nœud quelconque, de la forme $B_1, B_2, \ldots, B_n$. Trois cas sont possibles:

1. B_1 est une unification, de la forme $t_1 = t_2$.

   Dans ce cas, le système exécute cette unification, et deux cas sont possibles:

   a. L'unification échoue.

      Dans ce cas, le nœud n'a pas de successeur dans l'arbre.

   b. L'unification réussit et a pour résultat la substitution σ.

      Dans ce cas, le nœud N possède un successeur N_1, obtenu en appliquant σ au but $B_2, \ldots, B_n$. En outre, on associe la substitution σ à l'arc reliant N à N_1.

2. B_1 est un appel de procédure, de la forme $p(x_1, \ldots, x_n)$, où p est un nom de primitive.

   Dans ce cas, le système exécute la primitive, et deux cas sont possibles:

   a. Les instanciations des variables de B_1 ne respectaient pas les directionalités de p.

      Dans ce cas, le nœud N n'a pas de successeur dans l'arbre. On lui adjoint une étiquette portant la mention "erreur".

   b. Les directionalités de p étaient respectées, et l'exécution a produit un ensemble de substitutions $\{ \sigma_1, \ldots, \sigma_r \}$, avec $r \geq 0$.

      Dans ce cas, le nœud N possède r successeurs $N_1, \ldots, N_r$. Pour $i \in [0..p]$, le nœud N_i est obtenu en appliquant la substitution σ_i au but $B_2, \ldots, B_n$. En outre, on associe la substitution σ_i à l'arc reliant le nœud N à N_i.

3. B_1 est un appel de procédure, de la forme $p(x_1, \ldots, x_n)$, où p est un nom de relation.
   Il lui correspond dans P une définition de relation de la forme:

   $$p(y_1, \ldots, x_n) \Leftrightarrow CP_1 \text{ ou } \ldots \text{ ou } CP_r , r \geq 0,$$

   où, pour $0 \leq i \leq r$, CP_i est une conjonction préfixée de la forme:

   $$\text{exist } y_{m+1} : t_1, \ldots, y_{m+s} : t_s : C_i ,$$

   dans laquelle C_i est une conjonction de littéraux, c'est-à-dire un but.

   Dans ce cas, le nœud N possède r successeurs $N_1, \ldots, N_r$. Pour i: $0 \leq i \leq r$, on crée alors le nœud N_i à partir de la conjonction C_i de la façon suivante:

- On renomme les variables $y_1,\ldots,y_m,y_{m+1},\ldots,y_{m+s}$, en créant une substitution de renommage que l'on applique simultanément:
  - à la conjonction C_i, ce qui donne une nouvelle conjonction CR_i,
  - à l'entête $p(y_1,\ldots,y_m)$, ce qui donne une nouvelle entête $p(z_1,\ldots,z_m)$.
- On unifie cette nouvelle entête avec l'appel de procédure B_1, ce qui produit une substitution σ_i.
- On applique cette substitution à la conjonction CR_i, ce qui produit une nouvelle conjonction $CR\sigma_i$.

- On crée le nœud N_i en remplaçant B_1 par $CR\sigma_i$ dans le but N. Ce nœud contient donc le nouveau but $CR\sigma_i$ et B_2 et ... et B_n. En outre, on associe la substitution σ_i à l'arc reliant N à N_i.

Dans l'arbre ainsi constitué, il peut exister un certain nombre de nœuds qui ne possèdent pas de successeur. On distingue parmi eux:

- Ceux dont le but ne contient aucun littéral. On les appelle *nœuds de réussite*.
- Ceux dont le but contient un ou plusieurs littéraux. On distingue parmi ces derniers:
  - Ceux qui possèdent l'étiquette "erreur". On les appelle *nœuds d'erreur*.
  - Les autres, que l'on appelle *nœuds d'échec*.

On peut alors donner les définitions suivantes qui constituent le cœur de la sémantique opérationnelle du langage:

- Une *branche de réussite* est une branche qui joint la racine d'un arbre de dérivation à l'un de ses nœuds de réussite.
- Une *substitution-résultat* est la substitution obtenue en composant les substitutions associées à tous les arcs qui forment une branche de réussite, et en restreignant cette substitution aux seules variables présentes dans le but initial.
- Le résultat de l'exécution d'un but est l'ensemble de toutes les substitutions-résultats de son arbre de dérivation.
- La signification d'un programme est l'ensemble de tous les résultats de toutes les exécutions de buts que l'on peut requérir par rapport à ce programme.

2.5.2 Le problème de la négation.

La sémantique que nous venons de présenter ne s'applique qu'aux programmes qui ne contiennent pas de négations. Pour prendre en compte ces négations, il suffirait d'incorporer à notre sémantique le mécanisme de la négation par échec. L'introduction de ce mécanisme n'apporte rien de nouveau, nous ne la détaillerons pas dans ce mémoire. Le lecteur intéressé peut consulter le chapitre 3 de [Lloyd 87].

2.6 Traduction du langage A-Prolog

2.6.1 Introduction

La traduction d'un programme A-Prolog consiste à (mise à part la mise-à-jour de la B.D.) transformer les conjonctions dans les seconds membres des définitions de relations pour qu'elles soient correctes par rapport aux directionalités abstraites. Après avoir défini ce qu'est une conjonction correcte, on va développer en détail toutes les étapes de cette transformation. Etant donnée une conjonction d'atomes, sa transformation en conjonction correcte s'effectue par les étapes suivantes: on parcourt toute la conjonction de gauche à droite en transformant chaque atome. La transformation de la conjonction de départ est alors la concaténation des transformations obtenues pour chaque atome. La transformation d'un atome contenant les variables $X_1,...,X_n$ est à son tour la concaténation des transformations de cet atome pour chacune de ces variables. Enfin, la transformation d'un atome par rapport à une variable permet entre autre de voir si cette variable est détruite par l'appel de procédure correspondant à cet atome. Si c'est le cas, on précède cet atome par une procédure qui sauvegarde cette variable dans une autre.

Selon la directionalité abstraite utilisée pour chaque atome de cette conjonction, on obtient une transformation différente. Donc, lorsque les atomes de la conjonction ont plusieurs directionalités abstraites, on peut obtenir plusieurs transformations possibles. On est alors amenés à choisir une transformation parmi celles-ci. Pour cela, il faut choisir la transformation la plus efficace en terme de vitesse d'exécution. On définira alors un critère de choix ou de classement de ces transformations. Ce critère sera basé sur des considérations d'efficacité telles que vitesse d'exécution et consommation de mémoire. L'analyse qui va suivre est empruntée à [B. Le Charlier 90].

2.6.2 Rappels

Tout T.A. possède les opérations obligatoires suivantes:

- Test d'égalité de deux valeurs de type t.
- Copie d'une valeur dans une variable du même type.
- Génération d'une valeur ayant un certain type.

Notation: Une directionalité abstraite sera notée:

$$< dp_1 , \ldots , dp_n >$$

Où $dp_i \in \{$ input, destroyed input, output $\}$.

En fait pour les deux premières opérations, il s'agit d'une seule opération ayant les deux directionalités abstraites suivantes:

$dp_1 = <$ input , input $>$

$dp_2 = <$ input , output $>$

. Dans ce qui suit on notera :

égal(x_1, x_2) les deux premières opérations (test d'égalité ou recopie).

generate(x) la troisième opération.

2.6.3 Suites fortement correctes

Définition : Soient:

- $A_1 , \ldots , A_n$ une conjonction d'atomes.
- $x_1 , \ldots , x_n$ l'ensemble des variables figurant dans cette conjonction.
- $\{ < x_1, d^0_1 > , \ldots , < x_m, d^0_m > \} = vs_0$

  l'état des variables à l'entrée de la procédure (c'est-à-dire avant exécution du premier atome A_1).

La suite sera dite fortement correcte, par rapport à vs_0 s'il existe n états de variables :

$$vs_j = \{ <x_1, d_{j1}> , \ldots , < x_m, d_{jm}> \}$$

tels que l'on ait pour tout j : $1 \leq j \leq n$:

1) Posons $A_j = p(x_{k1}, \ldots , x_{kq})$ avec $1 \leq k_i \leq m$ pour tout i : $0 \leq i$

On doit avoir $x_{k\alpha} \neq x_{k\beta}$ si $\alpha \neq \beta$, sauf si le type de ces variables est à représentation ground.

2) Il existe dans la spécification de l'opération une directionalité abstraite:

$<dp_1 , \ldots , dp_q>$

où $dp_i \in$ { input, destroyed input, output }

telle que, $d_{k\alpha}$ est compatible avec $d_{p\alpha}$ pour tout $\alpha : 1 \leq \alpha \leq q$ selon la table ci-dessous :

$d^j_{k\alpha}$	dp_α	$d_{k\alpha}{}^{j+1}$
value	input	value
value	destroyed input	destroyed
variable	output	value

(T1)

On voit que $d_{k\alpha}{}^j$ ne peut être destroyed. La table donne le nouvel état des variables $x_{k1}, \ldots , x_{kq}$ après l'appel A_j. L'état des autres variables étant inchangé.

Remarque :

Les deux premières colonnes de cette table donnent la compatibilité entre les états des variables et la directionalité abstraite de p, tandis que les deux dernières donnent l'état des variables après cet appel.

Exemple :

Soit la conjonction $p(x_1, x_2)$, $q(x_2, x_3, x_4)$ avec l'état initial des variables:

vs_0 = { $< x_1$, value> , $<x_2$, variable> , $<x_3$, variable>, $<x_4$, variable> }.

et les directionalités abstraites suivantes :

. Pour p : 1) $dp_1 =$ < input, output >.

2) $dp_2 =$ < destroyed input >.

3) $dp_3 =$ < output, input >

. Pour q : 1) dq_1 = < output, input, input >.

2) dq_2 = < destroyed input, output, output >.

Cette conjonction est fortement correcte par rapport à vs_0 et les directionalités abstraites ci-dessus; elle n'est pas fortement correcte par rapport à ce même vs_0 et la directionalité suivante:

. Pour p : 1) dp_1 = < output, input >.

. Pour q : 1) dq_1 = < output, input, input >.

2) dq_2 = < destroyed input, output, output >.

2.6.4 Justification de la définition

L'objectif est de justifier pourquoi on a défini une conjonction fortement correcte de cette façon.

1) Dans un atome A = p(x_1, ... , x_n), un nom de variable doit apparaître au plus une fois. Ce qui correspond à la première condition de la définition d'une suite fortement correcte sauf pour les types à représentation ground. Expliquons cela par un exemple:

Soit l'appel p(x, x) où p possède la directionalité suivante:

dp = < input, output > avec x = value.

Dans un langage de programmation impératif, la sémantique de cette appel est que la procédure p calcule une certaine valeur *v* à partir de la valeur du premier argument x, cette valeur étant assignée par la suite à ce même x.. Cela ne pose donc aucun problème.

Par contre, en programmation logique, cet appel a une sémantique différente : Le but <--- p(x, x) est résolu par l'interpréteur. Par unification, avec la tête d'une clause, la valeur du second argument sera égale à la valeur du premier c'est-à-dire, que la procédure va renvoyer comme valeur de x, la même valeur qu'avant l'appel. Or, cela ne correspond pas à ce que l'on veut. On doit alors remplacer cette conjonction par :

p(x, y), égal(x, y).

Lorsque la variable aura reçu une valeur après l'appel p(x, y), cette valeur sera comparée avec celle de x au moyen de la primitive de comparaison égal(x, y).

Ce problème ne se pose évidemment pas pour les types abstraits à représentation ground puisque, pour de tels types, les termes représentant les occurrences de ce type sont tous ground et donc, aucun argument n'a la valeur variable.

Cette explication, certes, ne justifie pas le cas général, mais peut être aisément étendue aux autres cas.

2) - Il est clair qu'un argument ne peut avoir la valeur destroyed puisqu'on ne peut pas réutiliser une variable dont la valeur a été détruite par un appel antérieur.

- Le cas $d_{k\alpha}^{j}$ = value et dp_{α} = output est inadmissible car l'argument correspondant ayant déjà été instancié à une valeur (puisque $d_{k\alpha}^{j}$ = value) il conservera cette même valeur après l'appel.

- Le cas variable, input ou destroyed input

Ce cas est également inadmissible puisque dans les deux cas input et destroyed input, l'argument correspondant doit être instancié à une valeur avant l'appel. Donc, par exemple, un appel:

$p(x)$ avec dp_1 = <input>

dp_2 = <input destroyed> et

vs_0 = { < x - variable > }

doit être transformé en la conjonction:

generate(x) , p(x).

En effet, d'après les deux dernières colonnes de la table, l'argument x sera instancié à une valeur et alors x aura la forme value; ce qui est maintenant compatible avec la directionalité de p.

2.6.5 Conjonction correcte.

2.6.5.1 Conjonction associée à un atome et une variable

Soient:

- $A_1, \ldots, A_n$ une conjonction d'atomes.
- A_j un atome de cette conjonction ($1 \leq j \leq n$).
- $x_1, \ldots, x_m$ l'ensemble des variables apparaissant dans cette conjonction.

Posons : $A_j = p(x_{k1}, \ldots, x_{kq})$

avec - $q \geq 0$

- pour tout i : $1 \leq i \leq q$; $1 \leq k_i \leq m$.

Soient:

- $ad_j = < dp_1, \ldots , dp_q >$ une directionalité abstraite apparaissant dans la spécification de la procédure p; avec :

$dp_j \in \{$ variable, input, destroyed input $\}$

- $vs_j = \{ <x_1, d_{j1}> , \ldots , < x_m, d_{jm}> \}$

un état des variables juste avant l'appel à p; avec:

pour tout i : $1 \leq i \leq m : d_i^j \in \{$ variable, value, destroyed $\}$

- $x_{k\alpha}$ un argument fixé de p; avec $1 \leq \alpha \leq q$.

- k le nombre d'occurrences de $x_{k\alpha}$ dans $p(x_{k1}, \ldots , x_{kq})$.

- α le nombre d'occurrences de $x_{k\alpha}$ dans $p(x_{k1}, \ldots , x_{kq})$ correspondant à la directionalité abstraite input ou destroyed input dans ad_j.

- β le nombre d'occurrences de $x_{k\alpha}$ dans $p(x_{k1}, \ldots , x_{kq})$ correspondant à la directionalité abstraite output dans ad_j. Donc $k = \alpha + \beta$.

- n le nombre d'occurrences de $x_{k\alpha}$ dans la conjonction :

$$A_j, A_{j+1}, \ldots, A_n.$$

Alors, à l'atome A_j, on peut toujours associer la conjonction d'atomes fortement correcte suivante: (pour simplifier l'analyse, on remplacera $x_{k\alpha}$ par x).

a) Si x correspond à value dans vs_j :

a.1) Si $k < n$:

Cela veut dire qu'il y a des occurrences de $x_{k\alpha}$ dans le restant des appels (i.e. dans la conjonction $A_j, A_{j+1}, \ldots , A_n$).

Pour les occurrences de x qui correspondent à la directionalité input ou destroyed input, il faut les sauver dans d'autres variables par la procédure de recopie égal(input, output), et remplacer ces occurrences par leurs copies correspondantes.

Pour les occurrences de x qui correspondent à la directionalité output, il faut les remplacer par les variables $x_{l\alpha+1}, \ldots , x_{l\alpha+\beta}$ et tester après l'appel si les valeurs retournées par p sont égales à x. Ce test sera fait par égal(input, input).

D'où la conjonction:

égal(x, x_{l1}) , égal(x, x_{l2}),... , égal(x, $x_{l\alpha}$),...

p(... , x_{kj},...),...

égal(x, $x_{l\alpha+1}$), éga(x, $x_{l\alpha+2}$),... , égal(x, $x_{l\alpha+\beta}$).

a.2) Si k = n:

Cela veut dire qu'on aura plus besoin de x après l'appel p. Donc, si x correspond à destroyed input, on a plus besoin de la sauver avant l'appel. On distinguera encore deux cas :

a.2.1. Si $\beta = 0$: (i.e. toutes les occurrences de x correspondent à input ou destroyed input)

En principe, il n'y a pas besoin de sauvegarder x; cependant, puisqu'on veut obtenir une conjonction fortement correcte, il doit y avoir au plus une occurrence d'un nom de variable dans p. Il en résulte que toutes les occurrences de x seront sauvegardées sauf pour une (quelconque) d'entre elles. Convenons que cette occurrence soit la première. On obtient la conjonction suivante:

égal(x, x_{l2}),... , égal(x, $x_{l\alpha}$), p(..., x,... , x_{lj},...).

a.2.2. Si $\beta > 0$

On est dans un cas semblable au point a.1. sauf que la première occurrence de x correspondant à input ou destroyed input n'a pas besoin d'être sauvée puisqu'elle ne sera plus référencée dans les appels suivants (k = n).

Dans ce cas, la conjonction obtenue est alors:

égal(x, x_{l2}),... , égal(x, $x_{l\alpha}$), p(... , x, ..., x_{lj},...) ,

égal(x, $x_{l\alpha+1}$), égal($x_{l\alpha+2}$) ,... , égal(x, $x_{l\alpha+\beta}$).

b) Si x correspond à variable dans vs_j:

b.1) Si $\alpha = 0$ (toutes les occurrences de x correspondent à output).

Aucune occurrence de x n'a besoin d'être sauvée puisqu'aucune de celles-ci ne correspond à destroyed input ($\alpha = 0$). Par contre, puisqu'il ne doit pas y avoir de répétition d'occurrences de x, on doit remplacer ces occurrences (sauf une quelconque, soit la première) par des variables $x_{l\alpha+1}$,... , $x_{l\alpha+\beta}$ et faire des tests après l'appel p pour comparer les valeurs de x avec chacune de ces variables à l'aide de la procédure égal(input, input). D'où la conjonction :

p(..., x,... ,x_{lj} ,...), égal(x, x_{l1}) ,... ,égal(x, $x_{l\alpha-1}$).

b.2 Si $\alpha > 0$

b.2.1 Si $k < n$:

On peut se ramener au cas a.1. en remplaçant p(...) par :

generate(x), p(... , x,...).

puisqu'après le generate(x), x aura la forme value. Plus précisément on aura dans ce cas la conjonction :

generate(x), égal(x, x_{l2}),..., égal(x, $x_{l\alpha}$), p(... , x,... , x_{lj},...).

b.2.3 Si $k = n$ et $\beta > 0$

On se ramène ici, de la même façon, au cas a.2.2 et l'on obtient :

generate(x), égal(x, x_{l2}),... , égal(x, $x_{l\alpha}$), p(... , x,... x_{lj},...) ,

égal(x, $x_{l\alpha+1}$), égal($x_{l\alpha+2}$) , ..., égal(x, $x_{l\alpha+\beta}$).

Remarque:

. Il est clair que les conjonctions obtenues dans chaque cas sont fortement correctes par construction.

. Les atomes "generate" et "égal" doivent évidemment être remplacés par leur nom respectifs pour chaque T.A. particulier.

. La conjonction obtenue est appelée: *conjonction associée à l'atome A par rapport à la directionalité abstraite ad, l'état des variables vs et la variable x.* Cette conjonction sera notée C(A, ad, vs, x).

2.6.5.2 Conjonction associée à un atome

Au point précédent, on a vu qu'à tout atome A, peut être associée une conjonction. Celle-ci peut se mettre sous la forme:

C(A, ad, vs, x) = C, A', D.

Où :

- C est une conjonction éventuellement vide, formée d'atomes "égal" et éventuellement d'atomes "generate".
- D est une conjonction éventuellement vide, formée d'atomes "égal".
- A' est l'atome A où l'on a remplacé les occurrences de x par ses copies respectives.

Soient

- $A_1,\ldots, A_n$ une conjonction d'atomes.

- A un atome de cette conjonction.

- $x_1,\ldots, x_q$ les variables apparaissant dans l'appel p(...).

Supposons que $C(A, ad, vs, x_j) = C_j, A', D_j$ pour tout $j : 1 \leq j \leq q$. Alors, on appelle *conjonction associée à l'atome A, par rapport à la directionalité ad et l'état des variables vs*, la conjonction :

$$C_1, C_2,\ldots, C_q, A'', D_1, D_2,\ldots, D_q.$$

Où

A'' désigne l'atome A où l'on a remplacé les occurrences de chaque variable $x \in \{x_1,\ldots, x_q\}$ par ses copies respectives.

Notation: Cette conjonction sera notée C(A, ad, vs).

Remarque:

. Il est également clair que C(A, ad ,vs) est une conjonction fortement correcte par construction.

. Soient C_1 et C_2 deux conjonctions d'atomes telles que:

$C_1 = A_1,\ldots, A_n$.

$C_2 = B_1,\ldots, B_m$.

alors, la notation C_1, C_2 désigne évidemment la conjonction:

$C = A_1,\ldots, An, B_1,\ldots, Bn.$

. En général, une opération possède plusieurs directionalités abstraites $ad_1,\ldots, ad_n$ dans sa spécification. Dans les "expressions":

C(A, ad, vs) et C(A, ad, vs, x),

ad désigne l'une des ad_j $j = 1,\ldots, n$.

2.6.5.3 Conjonction associée à une conjonction

Soient :

- $A_1,\ldots, A_n$ une conjonction d'atomes.

- $x_1,\ldots, x_n$ les variables apparaissant dans $A_1,\ldots, A_n$.

- $vs_0 = \{ < x_1, d^0_1 > , \ldots , < x_m, d^0_m > \}$. Un état des variables à l'entrée de la conjonction (c'est-à-dire avant "exécution" de A_1).

Supposons qu'il existe dans la spécification de la procédure A_j, une directionalité abstraite ad_j. Supposons que pour tout $j : 1 \leq j \leq n$, C_j soit la conjonction associée à A_j par rapport à vs_{j-1} et ad_j. L'état des variables vs_j étant obtenu d'après la table (T1) de la page 56. Alors, par définition, la conjonction associée à $A_1,\ldots, A_n$ est la conjonction $C_1,\ldots,C_n$.

Remarque:

. D'une manière analogue encore, $C_1,\ldots ,C_n$ est une conjonction fortement correcte par construction.

.Supposons, que pout tout $j \in [1..n]$, A_j possède k_j directionalités dans sa spécification, alors, selon la directionalité choisie pour chaque A_j, on peut obtenir $k_1 * k_2 * \ldots * k_n$ conjonctions associées. Dans la section suivante, on examinera un critère de choix d'une "meilleure" conjonction associée à une conjonction donnée.

2.6.6 Un ordre sur les conjonctions fortement correctes

Dans cette section, on se propose de répondre à la question: Etant donné une conjonction d'atomes ayant plusieurs conjonctions associées, comment choisir une meilleure (ou des meilleures) conjonction(s) associée(s)?

Dans ce qui suit, on va définir, pour toute conjonction associée à une conjonction donnée, un triplet de nombres entiers. On définira ensuite un ordre lexicographie sur ces triplets.

2.6.6.1 Coût de correction d'un atome

Soit C(A, ad, vs, x) = C A' D la conjonction associée à l'atome A par rapport à la directionalité abstraite ad, l'état de variables vs et l'argument x. Pour cette conjonction, on définit un triplet de nombre entiers $T = (n_1, n_2, n_3)$ où l'on désigne par:

- n_1 le nombre d'atomes "generate" dans C.
- n_2 le nombre d'atomes "égal" dans C.
- n_3 le nombre d'atomes "égal" dans D.

Soit A un atome, $x_1,\ldots , x_q$ l'ensemble des variables apparaissant dans A. Supposons que $C(A, ad, vs, x_j) = C_j A' D_j$ pour tout $j : 1 \leq j \leq q$ et que le triplet associé soit T_j. Alors, à la conjonction associée à l'atome A par rapport à la directionalité ad et l'état de variable vs est associé le triplet T défini par :

$$T = T_1 + T_2 + \ldots + T_q.$$

Où l'on a l'addition habituelle sur les triplets :

$$(m_1, m_2, m_3) + (n_1, n_2, n_3) = (m_1+n_1, m_2+n_2, n_3+n_3).$$

Le triplet T est appelé *coût de correction* de l'atome A.

2.6.6.2 Coût de correction d'une conjonction d'atomes

Soit $A_1,\ldots,A_n$ une conjonction d'atomes. Le coût de correction de cette conjonction est le triplet $T = T_1 + T_2 + \ldots + T_n$

où T_j désigne le coût de correction de A_j pour tout j: $1 \leq j \leq n$.

2.6.6.3 Relation d'ordre total

Soit $A_1, \ldots,A_n$ une conjonction d'atomes. Soit E l'ensemble des conjonctions associées à cette conjonction. Sur E, on peut définir la relation d'ordre définie par:

$$\forall\, C_1, C_2 \in E :\ C_1 < C_2 \ \Leftrightarrow T_1 <_{lex} T_2.$$

où - T_1 et T_2 désignent respectivement les coût de corrections de C_1 et C_2.

- $<_{lex}$ désigne l'ordre lexicographique défini récursivement sur les n-uplets de nombres entiers par :

$$(n_1, ..,n_p) <_{lex} (m_1,\ldots,m_p) \Leftrightarrow p = 1 \text{ et } n_1 < m_1$$

$$\text{ou } p > 1 \text{ et } n_1 = m_1 \text{ et } (n_2,\ldots,n_p) <_{lex} (m_2,\ldots,m_p).$$

La raison pour laquelle on a choisi précisément cette relation sera donnée plus loin.

2.6.6.4 Meilleure conjonction associée à une conjonction

Soit $A_1,\ldots, A_n$ une conjonction, E l'ensemble de ses conjonctions associées, notons par T_C le coût de correction d'un élément C de E. Alors, par définition, une meilleure conjonction associée à $A_1,\ldots, A_n$ est un élément de l'ensemble M défini par :

$$M = \{\ C\_ass \in E : T_{C\_ass} = \min T_C \text{ pour } C \in E\ \}.$$

Le fait qu'on ait choisit pour meilleure conjonction associée celle de coût minimum se justifie comme suit :

. "generate":

Un atome "generate" est la principale cause d'inefficacité dans une conjonction. En effet, dans un but :

<-- ganerate(x) , A.

où A est un sous-but, l'interpréteur Prolog va backtracker autant de fois que nécessaire sur genarate(x) pour faire réussir A. genarate(x) dans un but se comporte comme un générateur de valeurs (d'où l'appellation generate) d'un certain type. Ceci s'apparente avec le principe des algorithmes "generate and test" qui sont très inefficaces.

. Un atome égal (input, input) est plus efficace que égal (input , output) puisque dans le dernier cas, on doit créer une toute nouvelle structure identique à la précédente, ce qui implique une consommation de mémoire, alors que dans le premier, on se contente de parcourir les deux structures pour vérifier si elles sont identiques.

Une conjonction pouvant avoir plusieurs meilleures conjonctions associés possibles, on convient de choisir une quelconque parmi celles-là.

Remarque:

L'algorithme de recherche de la meilleure conjonction associée est un algorithme de type "generate and test " qui génère les permutations et calcule leurs coûts respectifs. La meilleure conjonction associée est en fait la première permutation générée par l'algorithme de recherche et appartenant à l'ensemble M défini ci-dessus.

2.6.7 Définition de relation associée

Soit $p(x_1 : t_1, \ldots, x_n : t_n) \Leftrightarrow Pref_1\ C_1$

$\vee\ Pref_2\ C_2$

$\ldots$

$\vee\ Pref_n\ C_q$.

où $Pref_i$ désigne le préfixe de C_i pour $i = 1, \ldots, q$.

A cette définition de relation, on va associer une autre définition de relation en remplaçant les conjonctions dans le second membre par leurs conjonctions associées. Soit $ad_p = <ad_1,\ldots,ad_n>$ une directionalité abstraite de p. Soit C_i une conjonction ($1 \leq i \leq q$) et, soient $x_{n+1},\ldots, x_{n+r}$ les variables supplémentaires figurant dans le préfixe $Pref_i$ et soit l'état des variables initial suivant:

$vs_i = \{<x_1, d_1>,\ldots ,<x_n, d_n> , <x_{n+1}, d_{n+1}>, < xn+1, \underline{variable}>,\ldots , < x_{n+r}, \underline{variable}>\}$.

Où d_j se déduit de ad_j selon la table:

ad_j	d_j
<u>input</u>	<u>value</u>
<u>destroyed input</u>	<u>value</u>
<u>output</u>	<u>variable</u>

(T2)

Alors la définition de relation associée à (1) est la définition de relation:

$p(x_1 : t_1, \ldots , x_n : t_n) \Leftrightarrow C'_1 \vee C'_2 \quad \ldots \quad \vee C'_q$.

Où pour tout i : $1 \leq i \leq q$, C'_i est la meilleure conjonction associée à C_i par rapport à la directionalité abstraite ad_p et l'état de variable vs_0.

<u>Remarque</u>:

. Soit C_i une conjonction de la définition de relation (1). Supposons que:

$C_i = A_1, \ldots,A_r$

et que l'état des variables $x_1,\ldots ,x_n$ après l'appel A_r soit:

$vs_f = \{< x_1, d_1'>,\ldots, <x_n, d_n'> \}$

Alors, pour tout j : $1 \leq j \leq n$, d_j' doit vérifier la table :

d_j'	adp_j
<u>value</u>	<u>input</u>
<u>value</u>	<u>output</u>
<u>destroyed</u>	<u>destroyed input</u>

(T3)

. Dans cette table, on remarque que d_j' ne peut être variable car, on ne voit pas l'utilité de passer à une procédure un argument non instancié avant l'appel et qui le reste après cet appel.

2.6.8 Exemples

2.6.8.1 Exemple théorique

Donnons ci-dessus des résultats d'exécution de l'analyseur data-flow pour le calcul des conjonctions associées. Pour simplifier les notations, on remplace:

- input ou destroyed input par in.

- output par out.

1. . C_1 = p(x,x,x,x1) et q(x,x1).

   . dp = <out,out,out,in>.

   . dq = <in,in>

   . vs_0 ={x-val,x1-var}

   . $C(C_1, <dp,dq>, vs_0)$= p(x$0,x$1,x$2,x1),egal(x,x$0),egal(x,x$1),
   egal(x,x$2),q(x,x1).

2. . C_2 = p(x,x,x,x1) et q(x,x1).

   . dp = <in,in,in,out>.

   . dq = <in,in>.

   . vs_0 = {x-val,x1-var}

   . $C(C_2, <dp,dq>, vs_0)$= generate(x1), egal(x,x$0), egal(x,x$1), egal(x,x$2),
   p(x$0,x$1,x$2,x1$0), egal(x1,x1$0), q(x,x1)

3. . C_3 = p(x,x,x,x1).

   . dp = <in,in,in,out>.

   . vs_0 ={x-val,x1-var}

   . $C(C_3, dp, vs_0)$= generate(x1),egal(x,x$0),egal(x,x$1),p(x,x$0,x$1,x1$0),
   egal(x1,x1$0)

4. . C_4 = p(x,x,x,x1) et q(x,x1).

. dp = <out,out,out,in>.

. dq = <in,out>.

. vs_0 = {x-var,x1-val}.

. $C(C_4, <dp,dq>, vs_0)$= p(x$0,x$1,x$2,x1),egal(x,x$0),egal(x,x$1),
egal(x,x$2),q(x,x1).

2.6.8.2 Exemple pratique

Soit la conjonction suivante extraite d'une relation qui calcule la longueur d'une file. Cette conjonction correspond au cas récurssif:

C = not(file_vide(f)) et

retrait_début(f, f1, x) et

length(f1, l1) et

succ(l1, l).

Supposons avoir:

1. Les directionalités suivantes:

- file_vide
  - ad1=<input>.
- retrait_début
  - ad2.1= <input, input, input>.
  - ad2.2 = <input, output, input>.
  - ad2.3 = <output, input, input>.
  - ad2.4 = <input, output, output>.
- length
  - ad3.1 = <input, input>.
  - ad3.2 = <input, output>.
- succ
  - ad4.1 = <input, output>.

- ad4.2 = (output, input>.

2. L'état des variable initial:

vs_0 ={ <f, value>, <f1, variable>, <l, variable>, <l1, variable>}.

Dans ces conditions, à la conjonction C, on peut associer au moins les trois conjonctions:

- C_1 = not(file_vide(f)), generate(f1), generate(x), retrait_début(f, f1, x), generate(l1), length(f1, l1), succ(l1, l), egal(x, x1).

c'est la conjonction associée à C par rapport à <ad1, ad2.1, ad3.1, ad4.2 >. Son coût est T_{C1} = (2, 0, 1).

- C2 = not(file_vide(f)), generate(f1), generate(x), retrait_début(f2, f1, x), égal(f, f2), length(f1, l1), succ(l1, l).

c'est la conjonction associée à C par rapport à <ad1, ad2.3, ad3.2, ad4.1 >. Son coût est T_{C1} = (2, 0, 1).

- C3 = not(file_vide(f)), retrait_début(f, f1, x), generate(l1), length(f1, l1), succ(l1, l).

c'est la conjonction associée à C par rapport à <ad1, ad2.1, ad3.1, ad4.2 >. Son coût est T_{C1} = (1, 0, 0).

2.6.9 Traduction de A-Prolog

Maintenant que nous disposons de la terminologie nécessaire, on peut définir complètement la traduction d'un programme A-Prolog. Soit donc P, un programme A-Prolog, P peut se mettre sous la forme:

$DR_1,\ldots, DR_n$

$DP_1,\ldots, DP_n$.

$DA_1,\ldots, DA_{m+n}$.

Où

- $DR_1,\ldots, DR_n$ sont les définitions de relations de P.
- $DP_1,\ldots, DP_n$ sont les déclarations de primitives de P.
- $DA_1,\ldots, DA_{m+n}$ sont les spécifications des directionalités abstraites des relations et primitives de P.

A chaque définition de relation ou déclaration de primitive correspond un DA_i qui spécifie les directionalités abstraites possibles comme spécifié dans la syntaxe de A-Prolog. Pour traduire ce programme, on procède comme suit :

1) On remplace chaque définition de relation DR_i par sa définition de relation associée DR_i'.

2) Dans DR_i', on doit remplacer chaque nom de primitive par le nom de la procédure Prolog qui implémente cette primitive pour la directionalité correspondante (celle ayant servi pour construire la définition de relation associée).

3) On remplace chaque définition de relation obtenue au point 2) et contenant n conjonctions par n clauses Prolog, la ième clause étant une clause dont la tête est l'entête de la définition de relation; et dont le corps est la ième conjonction de cette définiton de relation.

4) Soit une déclaration de primitive p et soient $dp_1,\ldots,dp_r$ les directionalités abstraites ayant servi à construire les meilleures conjonctions associées dans les définitions de relations. La traduction de cette déclaration de primitive est un ensemble de r directives Prolog. Ces directives permettent d'importer une procédure d'un certain module. Pour chacune des directionalités $dp_1,\ldots,dp_r$, on importe la procédure qui implémente la primitive p pour cette directionalité.

5) Toutes les spécifications des directionalités abstraites sont supprimées.

Remarque : En BIM-Prolog, la directive permettant d'inclure dans un programme P une procédure pr / n appartenant au module *module_name* est

:- import *predicat_name / arity* from *module_name*.

Or pour rappel, on a convenu qu'un module d'un T.A. T a le nom T; donc dans notre cas, on écrira :

:-import pr / n from T.

Soit donc DR une définition de relation, DR' sa définition de relation qui ne contient pas de déclaration de type et dont les noms des primitives ont été remplacés par les noms des procédures Prolog correspondantes.

Supposons $DR = p(X_1 , \ldots , X_n) <=> C'_1$ ou ... ou C'_r , alors la traduction de DR est l'ensemble des n clauses Prolog :

$p(X_1, \ldots ,X_n)$: - C'_1.

$p(X_1, \ldots ,X_n)$: - C'_2.

...

$p(X_1, ... ,X_n)$: - C'_r .

Soit DP une déclaration de primitive de nom p. Soit *adt_name* le nom du type abstrait correspondant et $pr_1,\ldots, pr_n$, les procédures Prolog qui implémentent p pour les directionalités $dp_1,\ldots, dp_n$. Alors, DP est traduite par les directives Prolog suivantes:

:- import(pr_1) from *adt_name*.

:- import(pr_2) from *adt_name*.

...

:- import(pr_n) from *adt_name*.

Adaptations:

a) La traduction d'une primitive telle que nous venons de l'expliquer pose un problème. En effet, on risque d'écrire plusieurs fois la directive d'implémentation de procédure et ceci pour deux raisons:

- Il se peut qu'une même procédure pr_i implémente la primitive p pour plusieurs directionalités.
- Une procédure peut figurer plusieurs fois dans le second membre de la définition de relation traduite.

La solution adoptée ici est d'utiliser une table qui reprend toutes les procédures ayant déjà été importées. Lorsqu'on rencontre un nom de procédure dans une conjonction, on consulte d'abord cette table pour savoir si cette procédure a déjà été importée ou pas. Si c'est le cas, on ne fait rien (on continue à traiter les autres atomes de la conjonction). Sinon, on génère une directive pour importer cette procédure.

Une ligne de cette table est composée des éléments suivants:

- Nom de la procédure.
- Son arité.
- Le module de T.A. auquel elle appartient.

b) A chaque directionalité d'une définition de relation correspond une traduction de celle-ci. Par exemple, pour traduire la définition de relation

$$R(X_1, ..., X_n) <=> C.$$

pour les deux directionalités dir_1, dir_2, on obtient une traduction pour dir_1 et une autre pour dir_2. On aura donc:

$R(X_1, ..., X_n) :- C_1$

$R(X_1, ..., X_n) :- C_2$

Le système va donc backtracker et donner des solutions multiples. Pour éviter cela, on génère automatiquement des noms de procédures différents pour chaque directionalité. On obtient alors: $R1(X_1, ..., X_n) :- C_1$.

$R2(X_1, ..., X_n) :- C_2$

Donc, étant donnée une définition de relation ayant les directionalités $dp_1, \ldots, dp_n$, alors, les traductions de p pour ces directionalités auront les noms $p_1, \ldots, p_n$.

3 Le langage AD-log

3.1 Introduction

Le langage AD-log est une autre extension de Prolog très similaire au langage A-Prolog. Pour AD-log, l'analyseur data-flow est plus évolué. Ici, le programmeur AD-log ne se préoccupe pas de l'ordre des littéraux dans les conjonctions. C'est l'analyseur data-flow qui se charge de trouver la bonne permutation des littéraux de telle façon que les conjonctions résultantes soient fortement correctes (en insérant éventuellement des atomes generate et égal). Une fois cette meilleure permutation construite, la traduction du programme AD-log est identique à celle d'un programme A-Prolog.

3.2 Syntaxe concrète de AD-log

La syntaxe de AD-log est identique à celle de A-Prolog sauf pour la syntaxe des conjonctions.

<conjonction> ::= <littéral>|

<littéral> et <conjonction>.

En fait, pour insister sur le fait que les littéraux ne seront pas exécutés de droite à gauche comme pour A-Prolog, on les sépare par des et pour dire que l'ordre des littéraux serait peut-être changé par l'analyseur data-flow.

3.3 Règles sémantiques de AD-log

(idem pour A-Prolog (voir page 50)).

3.4 Sémantique intuitive de AD-log

La sémantique de AD-log est exactement la même que celle de A-Prolog où au lieu de parler des conjonctions de littéraux, on parle des meilleures permutations de ces conjonctions qui sont définies dans la suite.

3.5 Traduction du langage AD-log

3.5.1 Introduction

Dans la traduction d'une conjonction d'atomes d'un programme A-Prolog, on a parcouru cette conjonction en transformant chacun de ses atomes. Donc, on garde l'ordre des atomes. Ici, pour une conjonction AD-log, sa traduction consiste à faire exactement les mêmes transformations, mais celles-ci sont effectuées pour toutes les permutations de cette conjonction. Là aussi et à fortiori, on a le choix entre plusieurs transformations possibles. On utilisera le même critère de choix parmi ces transformations, à savoir le coût de correction.

3.5.2 Meilleure permutation associée

Soient

. $C = A_1$ et ... et A_n une conjonction d'atomes.

. E_C l'ensemble des conjonctions obtenues en permutant l'ordre des atomes de C. E_C est muni de la relation d'ordre < définie au paragraphe 2.6.6.3.

. M(C), la meilleure conjonction associée à un élément p de E_C.

. T_p son coût de correction.

. $M_C = \{ M(C) : C \in E_C \}$.

Alors, la meilleure permutation M associée à C est la conjonction dont le coût T_m est défini par :

$$T_m = \min T_C \text{ pour } C \in M_C.$$

En général, il existe plusieurs permutations correspondant à ce minimum. On convient d'appeler la meilleure conjonction associée, une quelconque de ces permutations.

Exemple

Exemple théorique

Soient - $C = q(x_1, x_2, x_2)$ et $r(x_1)$ et $p(x_1, x_1)$.

- $dp_1 = <in, in>$, $dp_2 = <in, out>$, $dp_3 = <out, out>$.
- $dq_1 = <in, in, out>$, $dq_2 = <in, out, out>$.
- $dr = <in>$.
- $vs_0 = \{x_1\text{-var}, x_2\text{-val}\}$.

Alors, la meilleure permutation associée est $C_P = p(x_1, x_1), q(x_1, x_2, x_2), r(x_1)$; son coût est $T_{CP} = (0, 1, 2)$.

Exemple pratique

Considérons l'exemple du paragraphe 2.6.8.2. La meilleure permutation associée à la conjonction C est elle-même; son coût est donc nul: $T_{CP} = (0, 0, 0)$.

3.5.3 Définition de relation associée

Soit $p(x_1 : t_1, \ldots, x_n : t_n) \Leftrightarrow \text{Pref}_1\ C_1$

$\vee\ \text{Pref}_2\ C_2$

$\ldots$

$\vee\ \text{Pref}_n\ C_q$.

où Pref_i désigne le préfixe de C_i pour $i = 1, \ldots, q$.

A cette définition de relation, on associe la définition de relation suivante :

$p(x_1 : t_1, \ldots, x_n : t_n) \Leftrightarrow C_1 \vee C_2 \vee \ldots \vee C_q$.

Où pour tout $i = 1, \ldots, q$, C_i' désigne la meilleure permutation associée à Ci au sens du paragraphe précédent.

On a maintenant tous les éléments nécessaires pour définir précisément la traductions d'un programme AD-log:

Soit P un programme AD-log. P peut s'écrire sous la forme:

$DR_1, \ldots, DR_n$

$DP_1,\ldots, DP_m$.

$DA_1,\ldots, DA_{m+n}$.

Où

- $DR_1,\ldots, DR_n$ sont les définitions de relations de P.
- $DP_1,\ldots, DP_m$ sont les déclarations de primitives de P.
- $DA_1,\ldots, DA_{m+n}$ sont les spécifications des directionalités abstraites des relations et primitives de P.

La traduction de P est la traduction du programme A-Prolog suivant:

$DR_1',\ldots, DR_n'$.

$DP_1,\ldots, DP_m$.

$DA_1,\ldots, DA_{m+n}$.

Où

- DR_i' est la définition de relation obtenue à partir de DR_i en remplaçant chaque conjonction C dans le second membre de DR_i par la permutation de C correspondant à la meilleure permutation associée.

4 Langage de définition de T.A.

4.1 Introduction

Le langage de définition de T.As. permet de définir un T.A. par un ensemble de valeurs et d'opérations primitives. Lorsque cette définition est correcte par rapport à la syntaxe et par rapport aux règles sémantiques, un nouveau T.A. est rajouté à la base de données. Une définition de T.A. peut utiliser comme T.As. prédéfinis des T.As. déjà existants dans cette base.

4.2 Symboles de base

Les symboles de base de ce langage sont les mêmes que pour A-Prolog et AD-log. (voir page 48)

4.3 Syntaxe du langage

La syntaxe concrète du langage de définition de T.A. est donnée par la syntaxe BNF suivante: (empruntée à [B. Le Charlier 90])

<abstract data type definition> ::= <abstract data type head>
{ <domain declaration> <operation declaration> }
<abstract data type tail>

<abstract data type tail> ::= end of <a.d.t. name>

<abstract data type head> ::= Abstract Data Type <a.d.t.name> ;
< usage list>

<domains declaration> ::= { <domain declaratin> }

<domain declaration> ::= domain <domain name> { , <domain name> } ;
description <domain description> ;
implementation <domain imlementation> ;

<domain description> ::= <comment>

<domain imlementation> ::= <comment> | Is ground <comment>

<domain name> ::= <identifier>

<comment> ::= { { <comment element> } }

<comment element> ::= <comment> | <any character diffrent from { or }>

<operation declaration> ::= { <o.d. element> }

<o.d. element> ::= <operation declaration> | <module>

<operation declaration> ::= <operation head>
<operation description>
<operation implementation>

<operation head> ::= operation <operation name> ({ ; <parameters description> })

<parameters description> ::= <parameter> { , <parameter> } : <domain designation>

<domain designation> ::= <domain name> | <a.d.t. name> . <domain name>

<parameter> ::= <identifier>

<operation description> ::= description <comment>

<operation implementation> ::= { <o.i. element> } <o.i. element> ;

<o.i. element> ::= for <abstract directionality> implementation Is
<parameter designation>

<abstract dirctionality> ::= <a.d. element> { , <a.d.t. element> }

<a.d element> ::= <call mode> | [<call mode> { , <call mode> }]

<procedure designation> ::= <procedure name> |
<module name> . <procedure name>

<call mode> ::= in | out | inout

<module> ::= <AD module> | <AP module | <BIM_prolog module>

<AD module> ::= Abstract description module <module name> ;
<usage> <AD module> end ;

<AP module> ::= Abstract prolog module <module name>;
<usage> <AP program> end ;

<module name> ::= <vide> | <identifier>

<BIM_Prolog module> ::= BIM prolog module <module name>;
{ <BIM_Prolog program> } end ;

<usage> ::= <representation mode> |
<usage list>

<representation mode> ::= representation dependent ; |
representation independent ;

<usage list> ::= <vide> | uses <a.d.t. name> { , <a.d.t. name> } ;

4.4 Règles sémantiques

En plus de cette syntaxe, un programme du langage de définition de T.A. doit vérifier les règles sémantiques que nous rappelons ici :

1. On ne peut définir un T.A. déjà existant dans la base de données des T.As.

2. Si un T.A. utilise d'autres T.As. , ceux-ci doivent être présents dans la B.D.

3. L'arité d'une opération doit être égale à la longueur de la liste des directionalités.

4. Les types des variables qui figurent dans l'entête d'une opération doivent, soit être déjà présents dans la B.D., soit être le T.A. en cours de définition .

5. Une opération primitive ne peut être déclarée plus d'une fois.

4.5 Sémantique intuitive du langage de définition de T.A.

La sémantique exacte du langage n'a pas encore été définie; on se contente ici d'un aperçu. Un "programme" de définition de T.A. définit:

1) Un ensemble de T.A. par:

- la définition des ensembles avec leurs noms.
- la définition des opérations sur ces ensembles.

Ces ensembles et opérations sont définis sous forme de commentaires non vérifiés par le compilateur. Ces commentaires sont néanmoins obligatoires et peuvent servir de documentation pour les utilisateurs.

2) La représentation de ces T.A. au moyen de:

- Procédures BIM_Prolog.
- Descriptions AD-log ou A-Prolog.

Cette partie spécifie comment les occurrences et les opérations sur ces occurrences seront représentés.

La cohérence entre ces deux points est laissée à la responsabilité de l'utilisateur. Le compilateur va stocker les informations descriptives du T.A. et générer du code Prolog correspondant aux procédures et descriptions AD-log. Le code Prolog correspondant aux opérations implémentées directement en Prolog est stocké dans un fichier tandis que celles écrites sous forme de descriptions sont d'abord traduites en code Prolog, celui-ci est ensuite copié dans ce même fichier.

Remarque:

La sémantique complète du langage doit normalement être exprimée en terme de modèles de la logique des prédicats comme on l'a fait pour le langage Prolog.

4.6 Structure de la base de données des T.A

La base de données des T.A. est implémentée par un programme Prolog qui est une base de faits ; tous ces faits étant des prédicats dynamiques.

Dans la partie informelle, on a convenu que pour un T.A. donné, il existe un fichier unique qui contient le texte source de toutes les procédures Prolog correspondant aux opérations primitives de ce T.A. Les fichiers doivent donc avoir un nom unique qui les distingue des autres fichiers appartenant à d'autres T.A. On doit donc trouver un moyen pour générer des noms de fichiers différents des autres noms déjà générés. Une manière de faire est de créer des noms de fichiers "paramètrés" par un indice qu'on incrémente à chaque fois que l'on crée un nouveau fichier. De cette manière, on est certain de ne pas avoir de confusion. On doit alors disposer d'une information sous forme d'un prédicat dynamique qui mémorise le numéro ou l'indice du dernier nom de fichier et qui est mise à jour chaque fois que l'on crée un nouveau fichier. Par exemple, on a choisi ici que tout fichier généré a un nom de la forme '*fi*.pro' où *i* désigne cet indice. Le prédicat dynamique est :

internal_file_name(i)

où le *i* correspond à l'indice contenu dans le nom du fichier '*fi*.pro'. Donc, la B.D. contient toujours un unique prédicat dynamique internal_file_name(i).

En plus de ce prédicat, la B.D. des T.As. contient un ensemble de faits de la forme :

adtype (adt_name, Lused_adt, oper_name , internal_oper_name, oper_arity,
Lparam_types, internal_file_name , abstract_dir)

dont chaque argument représente un des attributs d'un T.A. Ces attributs ayant été expliqués dans la première partie :

adt_name est le nom du type abstrait.

Exemple : PILE, FILE, ARRAY, etc...

Lused_adt est la liste des noms des T.As. utilisés par le T.A. adt_name.

Exemple :[integer, byte, array] .

oper_name est le nom d'une opération primitive correspondant à ce T.A.

Exemple.: PUSH , CONCATE , ...etc.

internal_oper_name est le nom de la procédure Prolog, AD-log ou A-Prolog qui implémente l'opération de nom _oper_name pour la directionalité abstraite abstract_dir.

oper_arity est un nombre entier qui représente l'arité de l'opération primitive de nom oper_name.

Lparam_types est la liste des types des paramètres de cette opération. La longueur de cette liste est évidemment oper_arity.

internal_file_name est le nom du fichier Prolog qui contient le texte source BIM_Prolog de la procédure qui implémente cette opération pour la directionalité abstraite abstract_dir.

abstract_dir est une liste représentant une directionalité de l'opération de nom _oper_name.

4.7 Exemples d'implémentations de T.A.

4.7.1 Exemple 1: Implémentation du T.A. integer

4.7.1.1 Représentation

Ce type de donnée est déjà disponible en Prolog, on le présente ici pour illustrer la définition des T.As. Une valeur du type abstrait entier sera représentée par l'entier Prolog correspondant. Ce T.A. est donc à représentation ground. Sur ce type sont définies les opérations suivantes:

- l'addition.
- la soustraction.
- la multiplication.
- la division.
- l'opération qui détermine le quotient et le reste de la division euclidienne de deux entiers.
- calcul du successeur d'un entier.
- le symétrique d'un entier.
- l'opération de génération d'un entier.
- l'opération qui teste si deux entiers sont premiers entre eux.

Certaines opérations sont superflues dans la mesure où elles sont exprimables en fonction d'autres plus élémentaires; cependant, on les a retenues pour montrer comment on peut utiliser des descriptions AD-log ou A-Prolog pour implémenter de telles opérations.

Remarque:

Pour simplifier l'écriture, on convient d'adopter les correspondances suivantes:

- input = in.

- output = out.

4.7.1.2 "Programme"

Le programme suivant est basé sur [B. Le Charlier 90].

```
Abstract data type integer;
        domain integer;
        description { ensemble des entiers };
        implementation is ground { chaque entier est représenté par l'entier Prolog correspon-
                                   dant };
operation plus( x, y, z: integer);
description { plus(x, y, z) ⇔ x + y = z };
for in, in, [in, out]  implementation is p1;
for [in, out], in, in, implementation is p2;
for in, [in, out], in implementation is p3;
for out, out, in implementation is p4;
for in, in, out implementation is p5;
for out, out, out implementation is p6;
for out, in, out implementation is p7;
BIM_Prolog module;
{
p1(_X, _Y, _Z) :- _Z is _X + _Y.
p2(_X, _Y, _Z) :- _X is _Z - _Y.
p3(_X, _Y, _Z) :- _Y is _Z - _X.
```

```
naturel1(_X) : _X = 1 .
naturel1(_X) :-naturel1(_X1), _X is _X1 + 1 .
entier(0).
entier(_X) :-naturel1(_X1), pn(_X1, _X).
pn(_X, _X).
pn(_X, _MoinsX) :- _MoinsX is - _X.
p4(_X, _Y, _Z) :- entier(_X), _Y is _Z - _X.
p5(_X, _Y, _Z) :- entier(_Y), _Z is _X + _Y.
p6(_X, _Y, _Z) :- entier2(_X, _Y), _Z is _X + _Y.
p7(_X, _Y, _Z) :- entier(_X), _Z is _X + _Y.
pn0(0, 0) :- !.
pn0(_X, _X).
pn0(_X,_MoinsX) :- _MoinsX is - _X.
entier0_n(_N, 0).
entier0_n(_N, 0) :- _N > 0 , _N_1 is _N - 1, entier0_n(_N_1, _X_1),
                    _X is _X_1 + 1 .
entier2_n(_X, _Y, _N) :-
                    entier0_n(_N, _X1), _Y1 is _N - _X1,
                    pn0(_X1, _X), pn0(_X1, _Y) .
entier2(_X, _Y) :- entier(_N), entier2_n(_X, _Y, _N).
} end;
operation generate(x: integer);
description { generate(x) ⇔ true};
for out implementation is entier;
operation generate(x, y: integer);
description { generate(x, y) ⇔ true };
for out, out implementation is entier2;
operation moins(x, y, z: integer);
```

```
description { moins(x, y) ⇔ z = x - y };
for [in, out], [in, out], [in, out] implementation is moins;
Abstract description module;
{
relation moins(x: integer; y: integer; z: integer) ⇔ plus(y, z, x).
primitive plus(x: integer; y: integer; z: integer).
directionality for moins is [in, out], [in, out], [in, out].
directionality for plus is [in, out], [in, out], [in, out].
} end;

operation multiplication(x, y, z: integer);
description { multiplication(x, y, z) ⇔ x * y = z }
for [in, out], [in, out], [in, out] implementation is fois;
Abstract description module;
{
 relation fois(x: integer; y: integer; z: integer) ⇔
                    y = 0   et z = 0
     ou  exist yMoins: integer :
          sup(y, 0) et moins( y, 1, yMoins) et fois(x, yMoins) et plus( y1, x, y)
     ou  exist y1: integer; z1: integer:
          sup(0, y) et sym(y, y1) et fois(x, y1, z1) et sym(z1, z).
primitive sup(x; integer; y: integer).
primitive moins(x: integer; y: integer; z: integer).
primitive plus(x: integer; y: integer; z: integer).
primitive sym(x: integer; y: integer).
directionality for fois is [in, out], [in, out], [in, out].
directionality for sup is in, in.
directionality for moins is [in, out], [in, out], [in, out].
```

```
directionality for plus is [in, out], [in, out], [in, out].
directionality for sym is [in, out], [in, out].
} end;

operation division(a, b, q, r: integer);
description { division(a, b, q, r) ⇔a = b * q + r et 0 ≤ r < b };
for  in, in, [in, out], [in, out] implementation is division;
Abstract description module;
{
relation division(a: integer; b: integer; q: integer; r: integer) ⇔
        exist q1: integer: sup(r, 0) et sup(r, b) et fois(b, q, q1) et plus(q1, r, a).
primitive sup(x; integer; y: integer).
primitive plus(x: integer; y: integer; z: integer).
primitive fois(x: integer; y: integer; z: integer).
directionality for division is in, in, [in, out], [in, out].
directionality for sup is in, in.
directionality for plus is [in, out], [in, out], [in, out].
directionality for fois is [in, out], [in, out], [in, out].
} end;
operation premier(a, b: integer);
description { premier(a, b) ⇔ a et b sont premiers entre eux };
for in, in implementation is premier;
Abstract description module;
{
relation premier(a: integer; b: integer) ⇔
    b = 1 ou b = -1 ou a = 1 ou a = -1 ou
    exist q: integer; r: integer: division(a, b, q, r) et premier(a, r).
primitive division(a: integer; b: integer; q: integer; r: integer).
```

```
directionality for premier is in, in.
directionality for division is in, in, [in, out], [in, out].
} end;
operation succ(x, y: integer);
description { succ(x, y) ⇔ plus(x, 1, y) };
for [in, out], [in, out]  implementation is succ;
Abstract Prolog module;
{
relation succ(x: integer; y: integer) ⇔ plus(x, 1, y).
primitive plus(x: integer; y: integer; z: integer).
directionality for succ is in, [in, out].
directionality for plus is in, in, [in, out].
directionality for plus is [in, out], in, in.
directionality for plus is in, [in, out], in.
directionality for plus is out, out, in.
directionality for plus is in, in, out .
directionality for plus is out, out, out.
} end;
operation sym(x, y: integer);
description { sym(x, y) ⇔ x = -y };
for in, [in, out] implementation is sym;
Abstract Prolog module;
{
relation sym(x: integer; y: integer) ⇔  multiplication(x, -1, y).
primitive multiplication(x: integer; y: integer; z: integer).
directionality for sym is in, [in, out].
directionality for multiplication is [in, out], [in, out], [in, out].
} end;
```

operation sup(x, y: integer);

description { sup(x, y) ⇔ x ≥ y };

for in, in implementation is s1;

BIM_Prolog module;

{

s1(_X, _Y) :- _X >= _Y.

} end;

end of integer.

4.7.1.3 Mise-à-jour de la Base de données

Supposons que l'on parte à zéro i.e. la base de données des T.As. est vide. Elle est cependant initialisée par l'enregistrement du numéro interne de fichier contenant le code Prolog. Elle contient donc:

```
:- alldynamic.
internal_file_number(0).
```

Les opérations implémentées par des descriptions AD-log ou A-Prolog sont d'abord traduites en code Prolog puis ce code est rajouté à celui des procédures implémentées directement en Prolog. Le contenu de la base de données après compilation de ce "programme" est:

```
:- alldynamic.
internal_file_number(1).
adtype(integer, [], plus, p1, 3, [integer, integer, integer],'f0.pro', [in, in, in]).
adtype(integer, [], plus, p1, 3, [integer, integer, integer],'f0.pro', [in, in, out]).
adtype(integer, [], plus, p2, 3, [integer, integer, integer],'f0.pro', [in, in, in]).
adtype(integer, [], plus, p2, 3, [integer, integer, integer],'f0.pro', [out, in, in]).
adtype(integer, [], plus, p3, 3, [integer, integer, integer],'f0.pro', [in, in, in]).
adtype(integer, [], plus, p3, 3, [integer, integer, integer],'f0.pro', [in, out, in]).
adtype(integer, [], plus, p4, 3, [integer, integer, integer],'f0.pro', [in, in, in]).
adtype(integer, [], plus, p5, 3, [integer, integer, integer],'f0.pro', [out, out, in]).
adtype(integer, [], plus, p6, 3, [integer, integer, integer],'f0.pro', [out, out, out]).
```

```
adtype(integer, [], plus, p7,  3, [integer, integer, integer],'f0.pro', [out, in, out]).

adtype(integer, [], generate, entier,  1, [integer],'f0.pro', [out]).
adtype(integer, [], generate, entier2,  2, [integer, integer],'f0.pro', [out, out]).

adtype(integer, [], moins, moins1,  3, [integer, integer, integer],'f0.pro', [in, in, in]).
adtype(integer, [], moins, moins2,  3, [integer, integer, integer],'f0.pro', [in, in, out]).
adtype(integer, [], moins, moins3,  3, [integer, integer, integer],'f0.pro', [in, out, in]).
adtype(integer, [], moins, moins4,  3, [integer, integer, integer],'f0.pro', [in, out, out]).
adtype(integer, [], moins, moins5,  3, [integer, integer, integer],'f0.pro', [out, in, in]).
adtype(integer, [], moins, moins6,  3, [integer, integer, integer],'f0.pro', [out, in, out]).
adtype(integer, [], moins, moins7,  3, [integer, integer, integer],'f0.pro', [out, out, in]).
adtype(integer, [], moins, moins8,  3, [integer, integer, integer],'f0.pro', [out, out, out]).

adtype(integer, [], multiplication, fois1,  3, [integer, integer, integer],'f0.pro', [in, in, in]).
adtype(integer, [], multiplication, fois2,  3, [integer, integer, integer],'f0.pro', [in, in, out]).
adtype(integer, [], multiplication, fois3,  3, [integer, integer, integer],'f0.pro', [in, out, in]).
adtype(integer, [], multiplication, fois4,  3, [integer, integer, integer],'f0.pro', [in, out, out]).
adtype(integer, [], multiplication, fois5,  3, [integer, integer, integer],'f0.pro', [out, in, in]).
adtype(integer, [], multiplication, fois6,  3, [integer, integer, integer],'f0.pro', [out, in, out]).
adtype(integer, [], multiplication, fois7,  3, [integer, integer, integer],'f0.pro', [out, out, in]).
adtype(integer, [], multiplication, fois8,  3, [integer, integer, integer],'f0.pro', [out, out, out]).

adtype(integer, [], division, division1, 4, [integer, integer, integer, integer],
                                                        'f0.pro', [in, in, in, in]).
adtype(integer, [], division, division2, 4, [integer, integer, integer, integer],
                                                        'f0.pro', [in, in, in, out]).
adtype(integer, [], division, division3, 4, [integer, integer, integer, integer],
```

'f0.pro', [in, in, out, in]).

adtype(integer, [], division, division4, 4, [integer, integer, integer, integer],
'f0.pro', [in, in, out, out]).

adtype(integer, [], premier, premier, 2, [integer, integer],'f0.pro', [in, in]).

adtype(integer, [], succ, succ1, 2, [integer, integer],'f0.pro', [in, in]).
adtype(integer, [], succ, succ2, 2, [integer, integer],'f0.pro', [in, out]).
adtype(integer, [], succ, succ3, 2, [integer, integer],'f0.pro', [out, in]).
adtype(integer, [], succ, succ4, 2, [integer, integer],'f0.pro', [out, out]).

adtype(integer, [], sym, sym1, 2, [integer, integer],'f0.pro', [in, in]).
adtype(integer, [], sym, sym1 2, [integer, integer],'f0.pro', [in, out]).

adtype(integer, [], sup, s1, 2, [integer, integer],'f0.pro', [in, in]).

4.7.1.4 Fichier Prolog généré

Remarque:

la directive BIM_Prolog

:- module(integer).

permet de déclarer que toutes les procédures Prolog appartenant au T.A. integer forment un module. De cette façon, toutes ces procédures deviennent locales à ce module et ne sont pas connues de l'extérieur. Il en résulte qu'il n' y a pas de risque de confusion entre des procédures de même nom appartenant à des T.A. différents.

Le contenu du fichier'f0.pro' est le suivant:

```
:- module(integer).
p1(_X, _Y, _Z) :- _Z is _X + _Y.
p2(_X, _Y, _Z) :- _X is _Z - _Y.
p3(_X, _Y, _Z) :- _Y is _Z - _X.
```

```
naturel1(_X) : _X = 1 .
naturel1(_X) :-naturel1(_X1), _X is _X1 + 1 .
entier(0).
entier(_X) :-naturel1(_X1), pn(_X1, _X).
pn(_X, _X).
pn(_X, _MoinsX) :- _MoinsX is - _X.
p4(_X, _Y, _Z) :- entier(_X), _Y is _Z - _X.
p5(_X, _Y, _Z) :-  entier(_Y), _Z is _X + _Y.
p6(_X, _Y, _Z) :- entier2(_X, _Y), _Z is _X + _Y.
pn0(0, 0) :- !.
pn0(_X, _X).
pn0(_X,_MoinsX) :- _MoinsX is - _X.
entier0_n(_N,0).
entier0_n(_N, 0) :- _N > 0 , _N1 is _N - 1, entier0_n(_N1, _X1),
                    _X is _X1 + 1 .
entier2_n(_X, _Y, _N) :-
                    entier0_n(_N, _X1), _Y1 is _N - _X1,
                    pn0(_X1, _X), pn0(_X1, _Y) .
entier2(_X, _Y) :- entier(_N), entier2_n(_X, _Y, _N).
fois1(_x, _y, _z) :- _y = 0 , _z=0 .
fois2(_x, _y, _z) :- _y = 0 , _z=0  .
fois3(_x, _y, _z) :- _y = 0 , _z=0 .
fois4(_x, _y, _z) :- _y = 0 , _z=0  .
fois5(_x, _y, _z) :- _y = 0 , _z=0 .
fois6(_x, _y, _z) :- _y = 0 , _z=0  .
fois7(_x, _y, _z) :- _y = 0 , _z=0 .
fois8(_x, _y, _z) :- _y = 0 , _z=0  .
```

```
fois1(_x, _y, _z) :- s1(_y, 0), moins2(_y, 1, _yMoins), fois1(_x, _yMoins, _y2), _y1 = _y2 ,
                     p1(_y1, _x, _y).

fois2(_x, _y, _z) :- s1(_y, 0), moins2(_y, 1, _yMoins), fois2(_x, _yMoins, _y1), p1(_y1, _x, _y).

fois3(_x, _y, _z) :- entier(_y),  s1(_y, 0), moins2(_y, 1, _yMoins), fois3(_x, _y2, _y1),
                     _yMoins = _y2,  p1(_y1, _x, _y).

fois4(_x, _y, _z) :-entier(_y), s1(_y, 0), moins2(_y, 1, _yMoins), fois4(_x, _y2, _y1),
                     _yMoins = _y2,  p1(_y1, _x, _y).

fois5(_x, _y, _z) :- s1(_y, 0), moins2(_y, 1, _yMoins), fois5(_x, _yMoins, _y2),
                     _y2 = _y1, p1(_y1, _x, _y).

fois6(_x, _y, _z) :- s1(_y, 0), moins2(_y, 1, _yMoins), fois6(_x, _yMoins, _y1), p1(_y1, _x, _y).

fois7(_x, _y, _z) :- entier(_y),  s1(_y, 0), moins2(_y, 1, _yMoins), entier(_x),
                     fois7(_x, _yMoins, _y1), p1(_y1, _x, _y).
fois8(_x, _y, _z) :- entier(_y), s1(_y, 0), moins2(_y, 1, _yMoins), fois8(_x, _y2, _y1),
                     _yMoins = _y2,  p1(_y1, _x, _y).

fois1(_x, _y, _z) :- entier(_y), s1(0, _y), sym(_y, _y1), fois1(_x, _y1, _z2), _z1 = _z2,
                     sym(_z1, _z).
fois2(_x, _y, _z) :- s1(0, _y), sym(_y, _y1), fois2(_x, _y1, _z1), sym(_z1, _z).
fois3(_x, _y, _z) :- entier(_y),  s1(0, _y), sym(_y, _y1), fois3(_x, _y2, _z1), _y1 = _y2,
                     sym(_z1, _z).
fois4(_x, _y, _z) :- entier(_y),  s1(0, _y), sym(_y, _y1), fois4(_x, _y2, _z1), _y1 = _y2,
                     sym(_z1, _z).
```

```
fois5(_x, _y, _z) :- s1(0, _y), sym(_y, _y1), entier(z1), fois5(_x, _y1, _z1), sym(_z1, _z).
fois6(_x, _y, _z) :- s1(0, _y), sym(_y, _y1), fois6(x, _y1, z1), sym(z1, z).
fois7(_x, _y, _z) :- entier(_y), s1(0, _y), sym(_y, _y1), entier(z1), fois7(_x, _y2, _z1),
                     _y1 = _y2, sym(_z1, _z).
fois8(_x, _y, _z) :- entier(_y), s1(0, _y), sym(_y, _y1), fois8(_x, _y2, _z1), _y1 = _y2,
                     sym(_z1, _z).
:moins1(_x, _y, _z) :- p1(_x, _y, _z).
:moins2(_x, _y, _z) :- p1(_x, _y, _z).
:moins3(_x, _y, _z) :- p2(_x, _y, _z).
moins4(_x, _y, _z) :- p3(_x, _y, _z).
moins5(_x, _y, _z) :- p3(_x, _y, _z).
moins6(_x, _y, _z) :- p4(_x, _y, _z).
moins7(_x, _y, _z) :- p5(_x, _y, _z).
moins8(_x, _y, _z) :- p6(_x, _y, _z).

division1(_a, _b, _q, _, r) :- s1(_r, 0), s1(_r, _b), fois2(_b, _q, _q1), p1(_q1, _r, _a).
division2(_a, _b, _q, _, r) :- entier(_r), s1(_r, 0), s1(_r, _b), fois2(_b, _q, _q1), p1(_q1, _r, _a).
division3(_a, _b, _q, _, r) :- s1(_r, 0), s1(_r, _b), fois4(_b, _q, _q1), p1(_q1, _r, _a).
division4(_a, _b, _q, _, r) :- entier(_r), s1(_r, 0), s1(_r, _b), fois4(_b, _q, _q1), p1(_q1, _r, _a).

premier(_a, _b) :- _a = 1 .
premier(_a, _b) :- _a = -1 .
premier(_a, _b) :- _b = 1 .
premier(_a, _b) :- _b = -1 .
premier(_a, _b) :- division4(_a, _b, _q, _r), premier(_a, _r).

succ1(_x, _y) :- p1(_x, 1,_y).
succ2(_x, _y) :- p1(_x, 1,_y).
```

succ3(_x, _y) :- p2(_x, 1,_y).

succ4(_x, _y) :- p7(_x, 1,_y).

sym1(_x, _y) :-fois1(_x, -1,_y).

sym2(_x, _y) :-fois2(_x, -1,_y).

s1(_X, _Y) :- _X >= _Y.

4.7.2 Exemple 2: Implémentation du T.A. File

4.7.2.1 Représentation

Une file peut être représentée par une liste Prolog. Cette représentation est commode pour l'opération d'ajout d'éléments en fin de file; elle est cependant très inefficace pour l'opération de retrait d'un élément début de file. Une meilleure façon est de représenter une file par une liste différence en utilisant la technique des structures de données incomplètes. La tête de la liste différence représente la fin de la file; le début de la file est représentée par la queue de la liste différence et les éléments de la file sont ceux de la liste différence. La définition du T.A. $FILE_T$ est la suivante:

4.7.2.2 Le programme

Abstract data type file;

uses integer;

domain integer;

description { Une file d'attente est un réceptacle, de contenance variable, dans lequel on peut stocker séquentiellement des éléments, et duquel on peut les retirer dans l'ordre où on les a stockés };

implementation { une file f est représentée par un terme de la forme f = file(N, D, F)

où

- N est le nombre d'éléments de la file.
- D et F sont deux listes Prolog telles que les éléments de la file soient les mêmes que ceux de la liste différence D \ F

Remarque: Ce T.A. n'est pas à représentation ground. Donc, il faut prévoir des opérations de recopie. };

```
operation file_vide(f: file);
description { file_vide(f) ⇔ f est une file vide };
for [in, out] implementation is f1;
BIM_ Prolog module;
{
 f1(_f) :- _f = file(0, _L, _L).
} end;
operation creer_file(f: file);
description { creer_file(f) ⇔ true};
for out implementation is f2;
BIM_ Prolog module;
{
f2(_f) :-  generate_list(_L1), nbre_elements(L1, _l), append(_L1,_V,_D1),
          f = file(_l, _D1, _V).
nbre_elements([],0) :- !.
nbre_elements(_l, [_|_L]) :-nbre_elements(_L, _l1), _l  is _l1 + 1 .
} end;
operation element_debut(f: file, e: integer);
description { f est une file non vide dont l'élément au début est e};
for in, [in, out] implementation is f3;
BIM_ Prolog module;
{
f3(_f, _e) :- f = file(0,_, _), !, write(' opération impossible: la file est vide').
f3(_f,_e) :- f= file(_, [_e|_D], _).
} end;
operation element_fin(f: file, e: integer);
```

```
description { f est une file non vide dont l'élément à la fin est e};
for in, [in, out] implementation is f4;
BIM_ Prolog module;
{
f4(_f, _e) :-
                    _f = file(_N, _D, _F),
                    append(_L, _F, _D),
                    last_one(_L, _e).
last_one([], _) :- write('opération impossible: la liste est vide'), !.
last_one([_e], _e) :- !.
last_one([_e1|_L], _e) :- last_one(_L, _e).
} end;
operation retrait_debut(f0, f1: file, e: integer);
description { f1 est la file f0 à laquelle on a retiré l'élément e au début};
for [in, out], in, in implementation is f5;
for in, [in, out], [in, out] implementation is f5;
BIM_ Prolog module;
{
f5(_f0, _f1, _e) :-
                    _f0 = file(_N0, _D0, _F0),
                    _D0 = [_e |_D1],
                    _N1 is _N0 - 1,
                    _f1 = file(_N1, _D1, _F0).
} end;
operation ajout_fin(f0, f1: file, e: integer);
description { f1 est la file f0 à laquelle on a rajouté l'élément e en fin de file};
for in, [in, out], in implementation is f6;
for [in, out], in, [in, out] implementation is f7;
```

```
BIM_ Prolog module;
{
f6(_f0, _f1, _e) :-
        _f0 = file(_N0, _D0, _F0),
        _F0 = [_e |_F1],
        _N1 is _N0 + 1,
        _f1 = file(_N1, _D0, _F1).
f7(_f0, _f1, _e) :-
        _f1 = file(_N1, _D1, _F1),
        _F0 = [_e |_F1],
        _N0 is _N1- 1,
        _f1 = file(_N0, _D0, _F0).
} end;
operation longueur_file(f: file, l: integer);
description {longueur_file(f, l) ⇔ l est le nombre d'éléments de f};
for in, [in, out] implementation is length;
Abstract Prolog module;
{
relation length(f: file; l: integer) ⇔
            file_vide(f) et l = 0
      ou    exist f1: file; l1: integer; x: integer:
            retrait_debut(f, f1, x) et
            not(file_vide(f)) et
            length(f1, l1)  et
            succ(l1, l).
primitive file_vide(f: file).
primitive succ(l1: integer; l2: integer) is succ.integer..
primitive retrait_debut(f0: file; f1: file; e: integer).
```

```
directionality for length is in, [in, out].

directionality for succ is out, in.

directionality for retrait_debut is in, out, out.

directionality for file_vide is [in, out].

} end;

operation egal_file(f1, f2: file);

description { egal_file(f1, f2) ⇔ f1 et f2 sont deux files ayant les mêmes éléments et dans le
              même ordre };

for in, [in, out] implementation is recopie_file;

for [in, out], in implementation is eq;

BIM_Prolog module;

{

append([],_L,_L) :- !.

append([_t|_T],_L,[_t|_Tapp]) :- append(_T,_L,_Tapp).

Suf([_e|_L],[_e|_Lgr]) :-  ground(_e),! , Suf(_L,_Lgr).

Suf(_,[]).

recopie_file(file(_D1,_F1),_f2) :- Suf(_D1,_D1ground),
                                   append(_D1ground,_V,_D2),
                                   _f2 = file(_D2,_V).

eq(_f1, _f2,) :- _f1 = _f2.

} end;

end of file.
```

4.7.2.3 Mise-à-jour de la base de données

Supposons que la base de données des T.As. soit dans l'état qui résulte de l'intégration du T.A. entier de l'exemple précédent. Dans ces conditions, le contenu de la base de données devient:

```
:- alldynamic.

internal_file_number(2).
```

$$$$$$ Ensembles des prédicats du T.A. entier (voir exemple précédent) $$$$$$

adtype(file, [integer], file_vide, f1, 1, [file], 'f1.pro', [in]).

adtype(file, [integer], file_vide, f1, 1, [file], 'f1.pro', [out]).

adtype(file, [integer], creer_file, f2, 1, [file], 'f1.pro', [out]).

adtype(file, [integer], file_tete, f3, 2, [file, integer], 'f1.pro', [in, in]).

adtype(file, [integer], file_tete, f3, 2, [file, integer], 'f1.pro', [in, out]).

adtype(file, [integer], file_debut, f4, 2, [file, integer], 'f1.pro', [in, in]).

adtype(file, [integer], file_debut, f4, 2, [file, integer], 'f1.pro', [in, out]).

adtype(file, [integer], retrait_debut, f5, 3, [file, file, integer], 'f1.pro', [in, in, in]).

adtype(file, [integer], retrait_debut, f5, 3, [file, file, integer], 'f1.pro', [in, in, out]).

adtype(file, [integer], retrait_debut, f5, 3, [file, file integer], 'f1.pro', [out, in, in]).

adtype(file, [integer], retrait_debut, f5, 3, [file, file, integer], 'f1.pro', [out, in, out]).

adtype(file, [integer], retrait_debut, f5, 3, [file, file, integer], 'f1.pro', [in, out, in]).

adtype(file, [integer], ajout_fin, f6, 3, [file, file, integer], 'f1.pro', [in, in, in]).

adtype(file, [integer], ajout_fin, f6, 3, [file, file, integer], 'f1.pro', [in, in, out]).

adtype(file, [integer], ajout_fin, f6, 3, [file, file, integer], 'f1.pro', [in, out, in]).

adtype(file, [integer], ajout_fin, f7, 3, [file, file, integer], 'f1.pro', [in, out, out]).

adtype(file, [integer], ajout_fin, f7, 3, [file, file integer], 'f1.pro', [out, in, in]).

adtype(file, [integer], longueur_file, length1, 2, [file, integer], 'f1.pro', [in, in]).

adtype(file, [integer], longueur_file, length2, 2, [file, integer], 'f1.pro', [in, out]).

adtype(file, [integer], egal_file, recopie_file, 2, [file, file], 'f1.pro', [in, in]).

adtype(file, [integer], egal_file, recopie_file, 2, [file, file], 'f1.pro', [in, out]).

adtype(file, [integer], egal_file, eq, 2, [file, file], 'f1.pro', [in, in]).

adtype(file, [integer], egal_file, eq, 2, [file, file], 'f1.pro', [out, in]).

4.7.2.4 Fichier Prolog généré

Le fichier 'f1.pro' de code Prolog généré par le compilateur a le contenu suivant:

```
:- module( file).
f1(_f) :- _f = file(0, _L, _L).
f2(_f) :-  generate_list(_L1), nbre_elements(L1, _l),  append(_L1,_V,_D1),
          f = file(_1, _D1, _V).
nbre_elements([],0) :- !.
nbre_elements(_l, [_|_L]) :-nbre_elements(_L, _l1), _l  is _l1 + 1 .
f3(_f, _e) :- f = file(0,_, _), !, write(' opération impossible: la file est vide').
f3(_f,_e) :- f= file(_, [_e|_D], _).
f4(_f, _e) :- f = file(0,_, _), !, write(' opération impossible: la file est vide').
f4(_f, _e) :- f= file(_, _,[_e|_F]).
f5(_f0, _f1, _e) :-
                _f0 = file(_N0, _D0, _F0),
                _D0 = [_e |_D1],
                _N1 is _N0 - 1,
                _f1 = file(_N1, _D1, _F0).
f6(_f0, _f1, _e) :-
                _f0 = file(_N0, _D0, _F0),
                _F0 = [_e |_F1],
                _N1 is _N0 + 1,
                _f1 = file(_N1, _D0, _F1).
f7(_f0, _f1, _e) :-
                _f1 = file(_N1, _D1, _F1),
                _F0 = [_e |_F1],
                _N0 is _N1- 1,
                _f1 = file(_N0, _D0, _F0).
```

```
:- import succ1 / 2 from integer.
- import succ2 / 2 from integer.
length1(_f, _L) :-
                    recopie_file(_f,_f2),
                    entier(_l1),
                    f5(_f, _f1, _x),
                    length1(_f1, _l1),
                    not( f1(_f2)),
                    succ1(_l1, _l).
length2(_f, _l) :- f1(_f), _l = 0 .
length2(_f, _L) :-
                    recopie_file(_f,_f2),
                    f5(_f, _f1, _x),
                    length2(_f1, _l1),
                    not( f1(_f2)),
                    succ2(_l1, _l).

append([],_L,_L) :- !.
append([_t|_T],_L,[_t|_Tapp]) :- append(_T,_L,_Tapp).
Suf([_e|_L],[_e|_Lgr]) :-  ground(_e),! , Suf(_L,_Lgr).
Suf(_,[]).
recopie_file(file(_D1,_F1),_f2) :- Suf(_D1,_D1ground),
                                   append(_D1ground,_V,_D2),
                                   _f2 = file(_D2,_V).
eq(_f1, _f2,) :- _f1 = _f2.
```

4.7.3 Exemple 3: Implémentation du T.A. Pile

4.7.3.1 Représentation

Une pile est simplement représentée par une liste Prolog. Malgré que ce choix d'implémentation est très simple, il est bien adapté au cas de la pile. Ce T.A. étant à représentation ground, sa traduction ne présente aucune difficulté. On se contente ici d'en donner la définition.

4.7.3.2 Le programme

```
Abstract data type pile;
domain integer;
description { une pile d'entiers est un ensemble formé d'un nombre variable, éventuellement
              nul, d'entiers sur lequel on peut effectuer les opérations suivantes:
                - ajout d'une nouvelle donnée.
                - test déterminant si la pile est vide.
                - consultation de la dernière donnée ajoutée et non supprimée depuis s'il y en
                  a une.
                - suppression de la dernière donnée ajoutée et non encore supprimée, depuis
                  s'il y en a une. };
implementation  { une pile est simplement représentée par une liste Prolog };
operation creer_pile (p: pile);
description { creer_pile(p) ⇔ true };
for out implementation is create;
BIM_Prolog module;
{
create(_p) :- generate_list(_p).
} end;
operation pile_vide (p: pile);
description { pile_vide(p) ⇔  p est une pile vide };
for [in,out] implementation is p_vide;
```

```
BIM_Prolog module;
{
p_vide(_p) :- _p = [].
} end;
operation empiler(e: integer, p, q: pile);
description { empiler(e, p, q) ⇔  p et q sont deux piles ayant les mêmes éléments excepté pour
                l'élément sommet e de q};
for [in,out], in, in implementation is empile1;
for in, in out implementation is empile1;
for [in,out], out, in implementation is empile1;
for out, in, out implementation is empile2;
BIM_Prolog module;
{
empile1( _s, _p, [_s | _p].
empile2(_s, _p, _q) :- entier(_s), _q = [_s | _p].
naturel1(_X) : _X = 1 .
naturel1(_X) :-naturel1(_X1), _X is _X + 1 .
entier(0).
entier(_X) :-naturel1(_X1), pn(_X1, _X).
pn(_X, _X).
pn(_X, _MoinsX) :- _MoinsX is _X.
} end;
operation depiler(p, q: pile);
description {depiler(p, q) ⇔  p est non vide et ∃ s ∈ PILE : empiler(s, q, p)};
for [in,out], in implementation is depile1;
for in, [in out] implementation is depile1;
BIM_Prolog module;
{
```

```
depile1([_s|_p], _p).
} end;
operation egal_pile(p, q: integer);
description { egal_pile(p, q) ⇔ p = q };
for in, [in, out implementation is same_stack;
for [in, out], in implementation is same_stack;
BIM_Prolog module;
{
same_stack([],[]) :- !.
same_stack(_x|_T1], [_x|_T2]) :- same_stack(_T1, _T2).
} end;
operation longueur_pile(p: pile, l: integer);
description { longueur_pile(p, l) ⇔ l est le nombre d'éléments de la pile p};
for  in, [in, out] implementation is long_pile;
Abstract description module;
{
relation long(p: pile; l: integer) ⇔ pile_vide(p) et l = 0
                                ou   exist p1: pile; l1: integer : not(pile_vide(p)) et
                                                                   depiler(p, p1) et
                                                                   long(p1, l1) et
                                                                   succ(l1, l).
primitive pile_vide(p: pile).
primitive depiler(p: pile; p1: pile).
primitive succ(l: integer; lsucc: integer) is succ.integer.
directionality for long is in, [in, out].
directionality for pile_vide is in.
directionality for  depiler is out.
directionality for succ is in, out.
```

```
} end;
end of pile.
```

IV Conclusion

L'objectif premier de ce mémoire était de pouvoir exploiter la puissance et l'élégance de la technique de programmation utilisant la notion de types abstraits. On a pu constater (comme la plupart des langages impératifs tels que C, PASCAL, FORTRAN, ... etc) que Prolog peut supporter cette notion moyennant des adaptations attentives. Le programmeur logique peut alors utiliser des types abstraits très élaborés et adaptés à ses besoins de développement d'applications. Ces types abstraits sont manipulables exclusivement par des opérations primitives spécifiques, très limitées et bien déterminées. On construit ainsi une librairie de base regroupant des types abstraits prédéfinis. Grâce à la propriété de réutilisabilité des types abstraits, cette librairie peut être agrandie très rapidement ("reuse within reuse").

Le langage A-Prolog a été conçu pour implémenter les opérations d'un T.A. par des descriptions logiques. Celles-ci étant traduites automatiquement en code Prolog. Cependant, rien ne nous empêche d'utiliser ce langage pour écrire des descriptions logiques ne correspondant pas nécessairement à des opérations sur un T.A. mais à des procédures quelconques.

Dans le cadre de la méthodologie de développement de programmes logiques exposée par [Y.Deville 90], on distingue trois grandes étapes:

1 L'élaboration d'une spécification.

2 La construction d'une description logique.

3 La dérivation d'un programme logique à partir de cette description logique.

Le second objectif de la démarche exposée dans ce mémoire est justement d'automatiser la troisième étape: la transformation de ces descriptions logiques et leur traduction en code Prolog exécutable. On permet ainsi au programmeur logique de consacrer pleinement ses capacités intellectuelles à l'étape la plus créative du processus de développement de programmes logiques. On parvient alors à écrire des programmes de façon déclarative.

Une limite à ce système, est qu'il ne prévoit pas la gestion des informations de contrôle et ne possède pas de caractéristiques extralogiques comme en Prolog. Une amélioration intéressante de ce système serait donc la gestion du cut et l'introduction de mécanismes d'accès et manipulation de programmes tels que *assert* et *retract* en Prolog.

Pour A-Prolog, nous n'avons défini que la sémantique opérationnelle. Pour doter ce langage de fondements théoriques, il serait également intéressant de définir sa sémantique déclarative en termes de la théorie des prédicats. Il en va de même pour le langage définition de T.A.

Annexe

Spécifications et procédures Prolog

Cette annexe regroupe le code de toutes les procédures Prolog développées. Nous proposons cependant de spécifier les principales qui figurent dans l'architecture du système (voir annexe 2). Dans l'ordre, nous avons les modules suivants:

1) Analyseur lexical.

2) Analyseur syntaxique des langages A-Prolog et AD-log.

3) Analyseur sémantique

4) Gérant des erreurs.

5) L'analyseur data-flow.

6) L'analyseur syntaxique du langage de définiton de T.A.

7) L'analyseur sémantique du langage de définiton de T.A.

8) Le gérant de la base de données de T.A.

9) Le générateur de fichiers Prolog.

10) Le compilateur du langage de définition de T.A.

1 Spécification de l'analyseur lexical

. Entête :

read_word(_leftover0,_l0,_n0,_symb,_l1,_n1,_leftover1,_stat)

. Types :

_leftover,_l0,_n0,_symb,_l1,_n1,_leftover1 : term.

_stat : list.

. Relation :

Supposons que le fichier d'entrée soit dans la situation suivante :

$$s_0, s_1 \cdots s_{i-1}, s_i , s_{i+1} , \cdots s_n.$$

et supposons que le dernier symbole lu soit s_{i-1} alors on a :

$\_l_0$: numéro de ligne du dernier symbole lu

$\_n_0$: numéro de colonne du dernier symbole lu

$\_leftover_0$: code ascii du premier caractère de la suite de symboles de base qui reste à lire dans le fichier d'entrée.

Remarque:

Les deux premiers paramètres servent à repérer dans le texte du programme, l'endroit où une erreur a eu lieu.

Dans ces conditions read_word/8 a l'effet suivant :

. Directionalité:

in : [val,val,val,var,val,val,val,var] : out [val,val,val,val,val,val,val,var] :

Dans ce cas, _symb est instancié à $\_s_i$, $\_l_1$ au numéro de ligne de $\_s_i$, $\_n_1$ au numéro de colonne de $\_s_i$, $\_leftover_1$ au code ascii du premier caractère de la suite de symboles restant à lire i.e s_{i+1}, ... ,s_n et _stat est non instancié.

in : [val,val,val,val,val,val,val,var] : out [val,val,val,val,val,val,val,val] :

Dans ce cas, si _symbole = $\_s_i$ alors _stat = [_l0,_n0,"",$\_s_i$,"", 'unexpected' , "" ,_symb,"",expected]; $\_l_1$, $\_n_1$, $\_leftover_1$ sont instanciés de la même façon que dans le premier cas.

Remarque :

. La fin de fichier est détectée par la lecture d'un symbole spécial '~'.

. Ce prédicat réussit toujours.

2 Spécification de l'analyseur syntaxique

. Entête:

synt(_filename,_str,_stat).

. Type:

_filename: atom.

_str: term.

_stat: list.

. Relation:

L'analyseur syntaxique reçoit un nom de fichier de texte *_filename* .Deux cas sont possibles. Ou bien ce texte respecte la syntaxe du langage A-Prolog et dans ce cas l'analyseur produit un terme Prolog ,_str, qui représente des informations utiles pour faire une analyse sémantique et aussi pour pouvoir traduire ce programme en code Prolog. Ou bien il existe dans ce texte une erreur syntaxique auquel cas _stat est instancié à une liste de la forme

_stat = [_l,_n|_msg]

où

- _l et _n sont respectivement le numéro de la ligne et de la colonne du symbole qui a causé l' erreur.
- _msg est une liste qui est le message d'erreur.

Dans le cas d'erreur _str est non défini.

. Directionalité:

in(val, var, var) : out(val, val, var).

in(val, var, var) : out(val, val, val).

in(val, var, val) : out(val, var, val).

3 Spécification de l'analyseur sémantique

. Entête :

pvsem(_str, _stat).

. Type:

_str: term.

_stat: list.

. Relation:

_str est une représentation interne du programme à analyser. S'il y a une erreur de sémantique alors

_stat = [_error_line_number , _error_column_number | _error_message]

où

. _l et _n sont respectivement le numéro de ligne et de colonne du symbole qui a causé l'erreur.

. _msg est une liste qui représente le message d'erreur.

S'il n'y a pas d'erreur, on ne fait rien et _stat est indéfini.

. Directionalité :

in(val, val) : out(val, val).

in(val, var) : out(val, val).

out(val, var) :out(val, var).

Principe :

L'analyseur sémantique reçoit le terme *_str* retourné par l'analyseur syntaxique. On distingue les deux étapes principales suivantes effectuées par l'analyseur sémantique :

La procédure

parcourir(_str,_TabProc,_Lappels,_stat)

permet, à partir de *_str*, de construire la table des procédures _TabProc et la liste Lappels qui est la liste de tous les appels figurant dans chaque définition de relation. Les appels true et false sont insérés d'office dans cette table. Ils sont donc considérés comme des appels "prédéfinis".

La procédure

louk_up(_TabProc,_Lappels,_stat)

permet, de vérifier que chaque élément de la liste _Lappels correspond à un élément de la table _TabProc.; autrement dit que tout appel correspond à une relation ou une primitive déjà déclarée. Pour accélérer la recherche d'un appel dans la table, celle-ci est représentée sous forme d'un arbre binaire trié par ordre alphabétique sur les noms des appels. Ceci peut s'avérer utile si le programme est assez grand.

Les messages d'erreurs possibles sont:

1. 'relation or primitive declared twice'.

2. 'unknown call'.

3. 'type mismach with variable v in call to call_name / arity .

4. 'undeclared variable'.

Dans le cas où il y a une erreur sémantique, le message correspondant est unifiés avec la variable _stat ; _stat est non instancié dans le cas contraire.

4 Spécification du gérant d'erreurs

Entete :

trt_diagn_err(_filename, _stat)

. Type:

_filename: atom.

_stat: list.

Principe :

Rappelons que

_stat = [_error_line_number , _error_column_number | _error_message]

D'abord on commence par écrire a l'écran le type d'erreur (syntaxe ou sémantique) et le numéro de la ligne ou l'erreur a eu lieu. Ensuite, on écrit le message d'erreur _error_message, puis la ligne de cette erreur avec un curseur en dessous du symbole erroné pour mieux aider la correction par l'utilisateur. Ceci est réalisé de la façon suivante:

On sait que read_word / 8 donne le symbole de base suivant ainsi que le numéro de la ligne et de colonne de celui-ci. Donc, il suffit de relire depuis le début le fichier contenant le programme jusqu'a atteindre un symbole de base dont le numéro de ligne et de colonne s'unifient avec ceux du message d'erreur. Ensuite on peut afficher la ligne contenant l'erreur.

la procédure countup(_line_number) permet de lire le fichier courant jusqu'a la ligne de numéro _line_number ; la procédure writeline(_ascii_code) permet de lire sur le fichier courant et d'écrire a l'écran jusqu'a rencontrer le caractère de fin de ligne; son code ascii est 10.

. Relation;

Si _stat = [_error_line_number , _error_column_number | _error_message]

est var on ne fait rien. Sinon lorsque c'est ground alors on affiche le texte du fichier de nom _filename et le message _error_message. Un curseur est placé à la ligne numéro _error_line_number et à la colonne _error_column_number.

. Directionalité:

in(val, var) out(val, var).

in(val, val) out(val, val).

5 Spécification de l'analyseur data-flow

Principe :

L'analyseur data-flow reçoit une conjonction de littéraux $liter_1$, ... , $liter_N$. Il génère une permutation de cette conjonction qui minimise le coût:

coût = [_nbre_de_generate, nbre_egal_avant, _nbre_egal_apres]

suivant l'ordre lexicographique.

_nbre_de_generate: est le nombre de generate(_var).

_nbre_egal_avant : est le nombre de egal(_x, _xcopie) avant l'appel $_liter_i$.

_nbre_egal_apres : est le nombre de egal(_x , _xcopie) apres l'appel $_liter_i$.

Ce problème est résolu par un algorithme du type "generate and test" . Il y a deux façons de réaliser ceci:

1. A un moment donné de l'exécution, on a déjà une meilleure permutation courante avec son coût. A la prochaine itération on génère (complètement) une nouvelle permutation et on regarde son coût. S'il est meilleur que celui obtenu jusque là, on la retient et on mémorise son coût, sinon, on passe à la suivante.
2. On fait la même stratégie que dans le premier cas sauf qu'au lieu de générer complètement une permutation et d'examiner son coût, on va générer des préfixes de permutations. Cette deuxième démarche s'avère plus efficace.

Ces deux algorithmes son implémentés respectivement par:

1 . Entêtes:

datanalyser(_filename, _tabinst, _bestconj, _bestcost)

datanalyser1(_filename, _tabinst, _bestconj, _bestcost).

. Type:

_filename: atom

_tabinst, bestconj, bestcost: list.

. Relation:

_filename est le nom du fichier contenant la conjonction à optimiser.

_tabinst décrit l'état initial des variables contenues dans la conjonction

ex: _tabinst = [_x1- val,_x2-var]

_bestconj et _bestcost représentent repectivement la meilleure conjonction associée et le coût de cette conjonction.

. Directionalités:

in(val, val, var, var) : out(val, val, val, val).

6 Spécification de l'analyseur syntaxique du langage de définition de T.A

. Entête :

sxdatatypedef(_filename,_str, _stat).

. Type:

_filename: atom.

_str: term.

_stat: list.

Principe :

Le principe est le même que pour l'analyseur syntaxique des langages AD-log et A-Prolog. Il suffit de remplacer dans la spécification de ce dernier le mot A-Prolog ou AD-log par langage de définition de T.A.

Directionalité:

in(val, var, var) : out(val, val, val).

in(val, var, var) : out(val, val, var).

in(val, var, val) : out(val, var, val).

7 Spécification de l'analyseur syntaxique du langage de définition de T.A

Entête:

```
adtsem(_str, _info, _stat).
```

Type:

_str: atom.

_info: term.

_stat: list.

Relation:

_str est le terme retourné par l'analyseur syntaxique. Ce terme est inspecté par cette procédure pour voir si le programme en question vérifie bien les règles sémantiques. Si c'est le cas, on construit un autre terme _info qui regroupe toutes les informations utiles sur le T.A. qu'on est en train de définir. _info sera exploité plutart par le module de gestion de la base de données pour enregistrer ces informations. Sinon (c'est-à-dire si au moins une de ces règles n'est pas respectée) alors _info est indéfini et _stat contient le diagnostic de l'erreur qui en résulte.

Directionalité:

in(val, var, var) : out(val, val, var).

in(val, var, var) : out(val, val, val).

in(val, var, val) : out(val, var, val).

8 Spécification du gérant de la base de données des T.A.

Il est composé de trois procédures principales:

1 . Entête:

dbwrite(_fact, _filename).

. Type:

_fact: term.

_filename: atom.

. Relation:

_filename est le nom du fichier Prolog qui est une base de faits qui implémente la base de données des T.A. _fact est un terme représentant un fait. L'effet de cette procédure est de rajouter le fait _fact à la base de faits contenue dans le fichier de nom _filename. Ce fait est mis à la fin.

. Directionalité:

in(val, val) : out(val, val).

2 . Entête:

dbdelete(_fact, _filename).

. Type:

_fact: term.

_filename: atom.

. Relation:

_filename est le nom du fichier Prolog qui est une base de faits qui implémente la base de données des T.A. _fact est un terme représentant un fait. L'effet de cette procédure est de retirer le premier fait de cette base qui s'unifie avec _fact. S'il n'y a aucun fait qui s'unifie avec _fact alors cette procédure échoue.

. Directionalité:

in(val, val) : out(val, val).

3 . Entête:

dbmodify(_fact, _newfact, _filename).

. Type:

_fact: term.

_filename: atom.

. Relation:

_filename est le nom du fichier Prolog qui est une base de faits qui implémente la base de données des T.A. _fact est un terme représentant un fait. L'effet de cette procédure est de substituer au premier fait de cette base qui s'unifie avec _fact le fait _newfact. S'il n'y a aucun fait qui s'unifie avec _fact alors cette procédure échoue.

. Directionalité:

in(val, val, val) : out(val, val, val).

9 Spécification du générateur de code Prolog

. Entête:

writeprogram(_filename, _structure).

. Type:

_filename: atom.

_structure: list.

. Relation:

_filename est le nom du fichier contenant le code Prolog correspondant au programme A-Prolog ou AD-log dont _structure est le résultat de la transformation par l'analyseur data-flow de la représentation interne renvoyée par l'analyseur sémantique.

. Directionalité:

in(val, var) : out(val, var).

10 Compilateur du langage de définition de T.A

. Entête:

adt_translator(_filename).

. Type

_filename: atom.

. Relation:

_filename est le nom d'un fichier contenant le texte d'un programme de définition de T.A. Deux cas sont possibles.

Ou bien ce programme respecte la syntaxe et la sémantique du langage et dans ce cas:

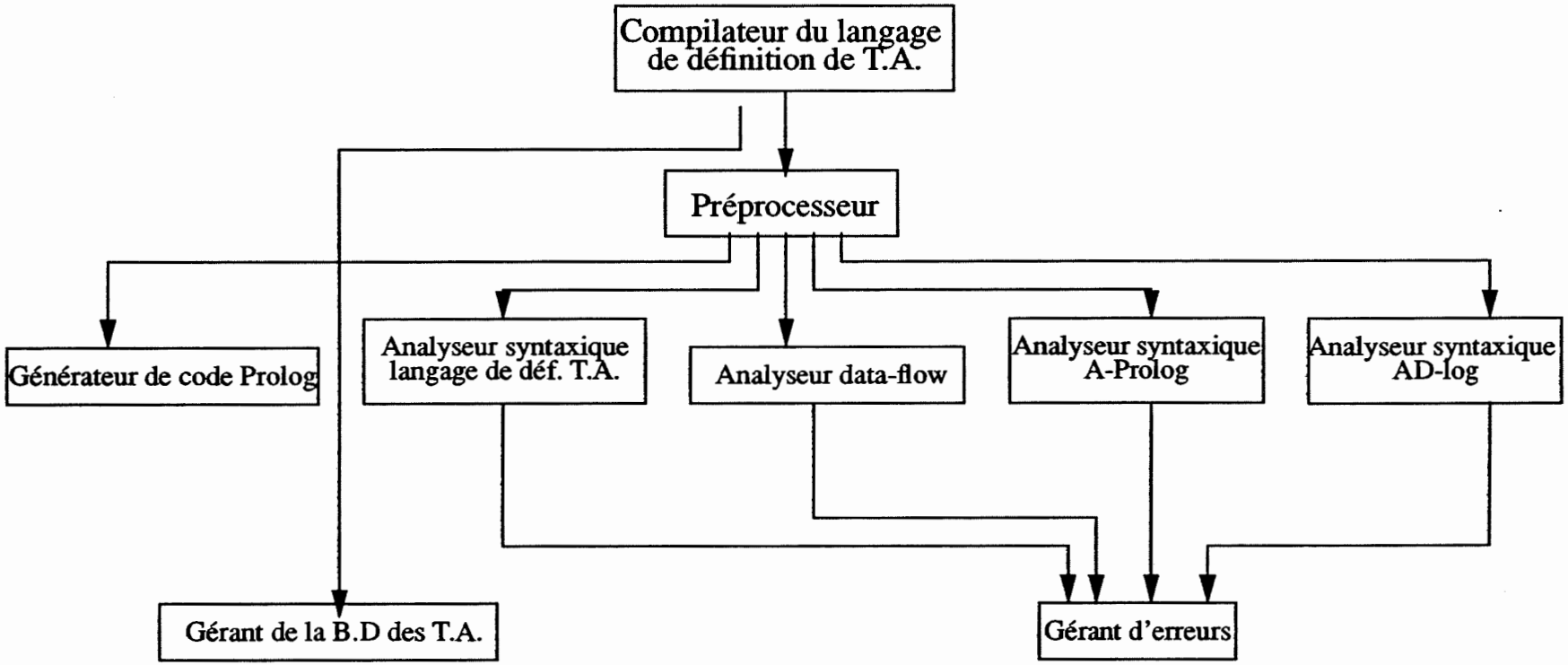
. la base de données des T.A est mise à jour pour enregistrer toutes les informations sur ce T.A.

. Un fichier contenant l'ensemble de toutes les procédures Prolog correspondant à ce T.A.

Ou bien il existe au moins une erreur, auquel cas le message d'erreur correspondant est affiché à l'écran.

. Directionalité:

in(val) : out(val).



Architecture des modules

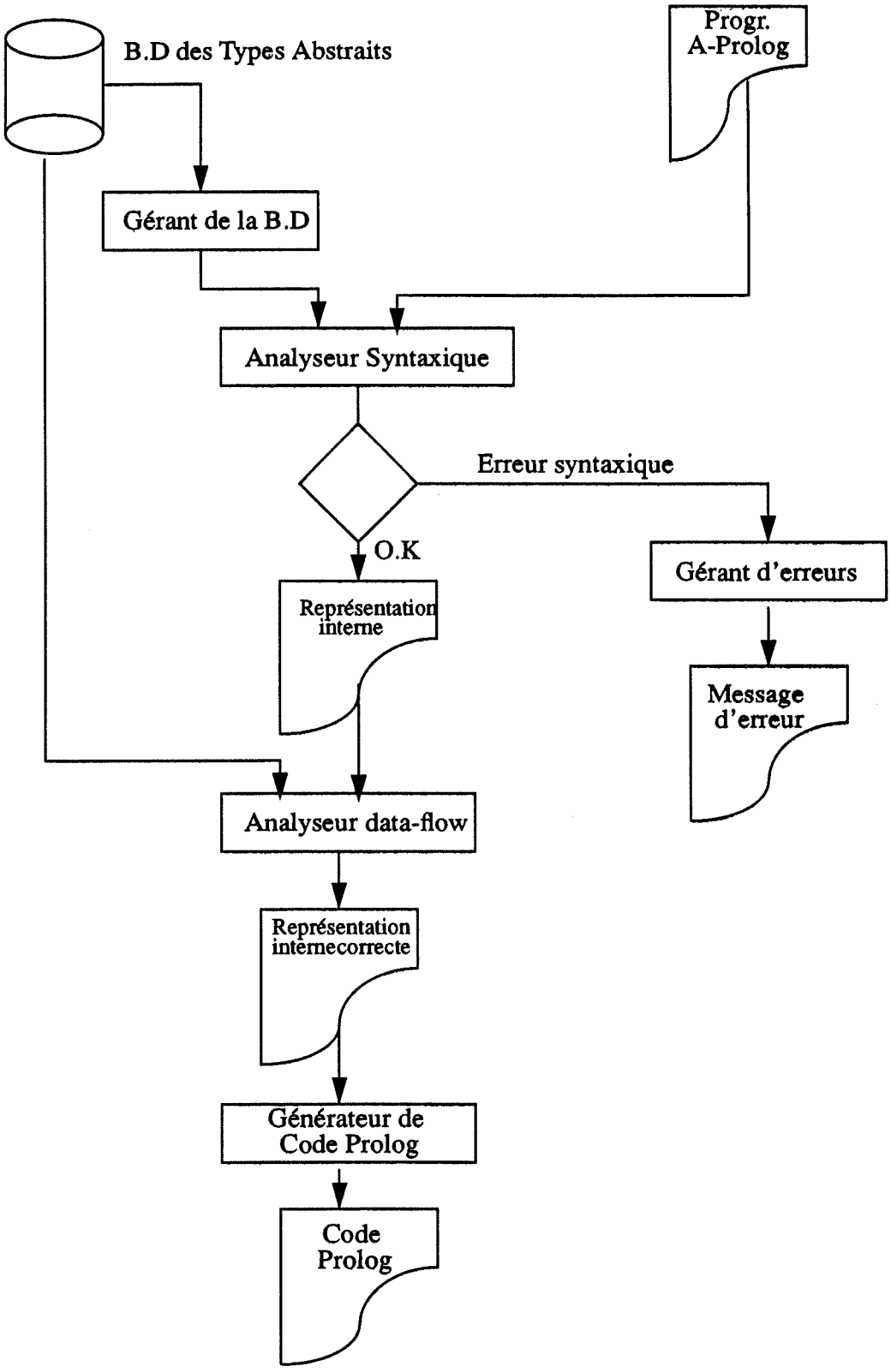


Diagramme des flux du compilateur A-Prolog ou AD-log

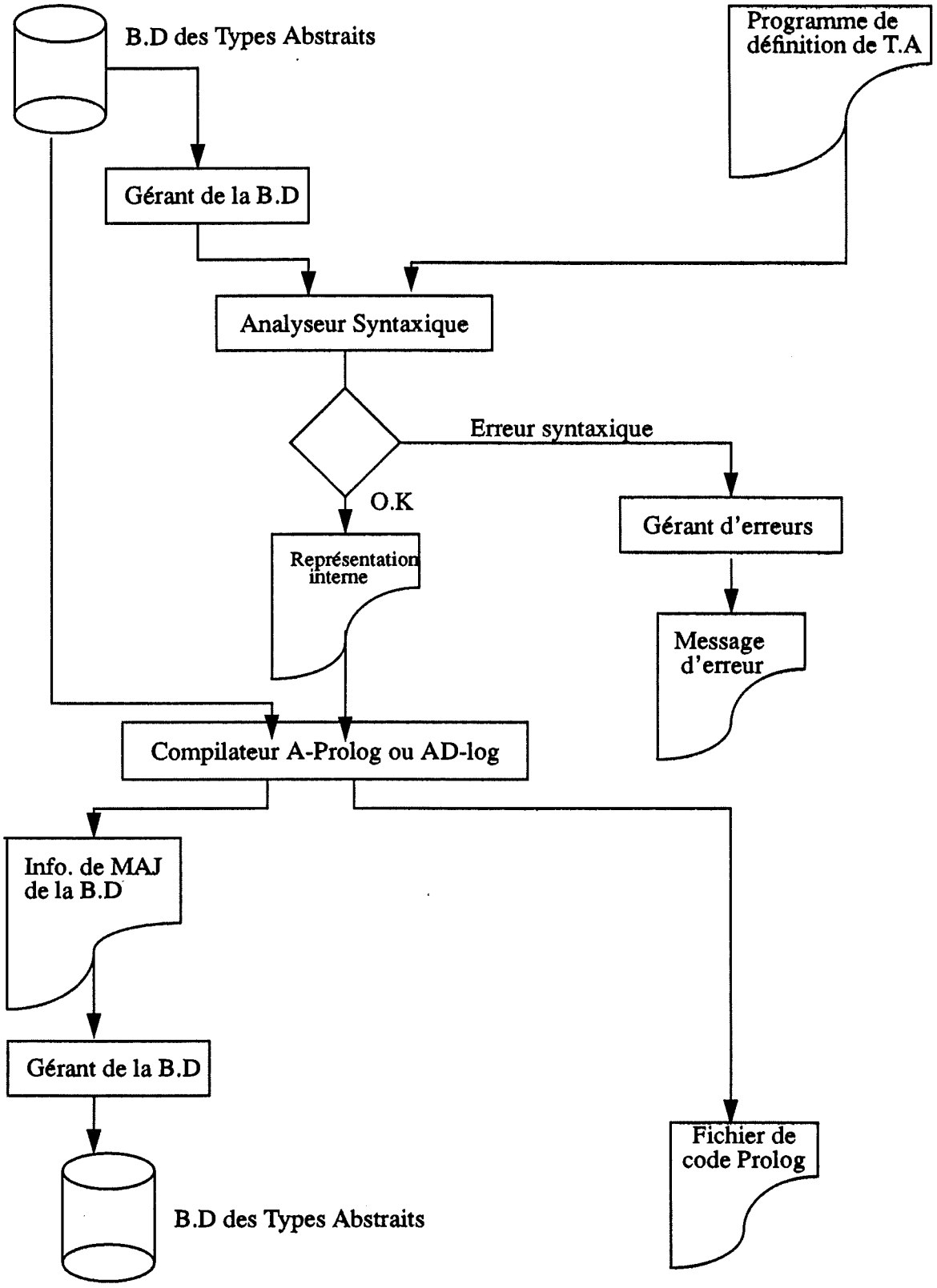


Diagramme des flux du compilateur de T.A

Bibliographie

[Clocksin. 84] **Clocksin, W.F. and Mellish, C.S.** *Programming in Prolog*. 2nd edition. Berlin: Springer-Verlag, 1984.

[Y.Deville 90] **Deville, Y.,** *Logic Programming: Systematic Program Development*, Addison-Wesley, International Series in Logic Programming,1990, 326 pages.

[Guttag. 77] **Guttag, J.V.,** *Abstract Data Types and The Development of Data Structures*, Communications of the ACM, june 1977, Vol 20, No 6, 9 pages.

[J.P. Hogne 90] **Hogne, J.P.,** *Une Démarche de Programmation Logique utilisant la notion de Type Abstrait*. Projet de Recherche Folon. 1990, 29 pages.

[B. Le Charlier 90] **Le Charlier B.,** *Types Abstraits et Programmation Logique*, Document communiqué lors du stage, 1990, 10 pages.

[B. Le Charlier 90] **Le Charlier B.,** *Syntaxe Concrète du Langage de Définition deTypes Abstraits*, Document communiqué lors du stage. 1990, 2 pages.

[B. Le Charlier 90] **Le Charlier B.,** *Exemple d'Implémentation de Types Abstraits*, Document communiqué lors du stage. 1990, 2 pages.

[B. Le Charlier 90] **Le Charlier B.,** *Langage Simple Didactique (LSD 80)*. Document communiqué lors du stage. 1990, 42 pages.

[J.W. Lloyd 87] **Lloyd J.W.,** *Foundations of Logic Programming*. 2nd edition. Symbolic Cmputation. Springer-Verlag, 1987, 212 pages.

[B.Meyer. 78] **Meyer, B., et Baudoin, C.,** *Méthodes de Programmation*, Collection de la Direction des Etudes et Recherches d'Electricité de France, Editions Eyrolles, 1978, 661 pages.

[E. Shapiro 86] **Shapiro, E., and Sterling, L.,** *The Art of Prolog*: Advanced Programming Techniques, Series in Logic Programming, The MIT Press, 1986, 437 pages.

[J. Uhl 90] **Uhl, J., and Albrecht Schmid, H.,** *A Systematic Catalogue Of Reusable Abstract Data Types*, Lecture Notes in Computer Science, Springer-Verlag, 1990, 344 pages.

```
read_word(_C,_l1,_n1,_Word,_l2,_n2,_leftover,_stat) :-
                var(_stat),!,
                read_word1(_C,_l1,_n1,_Word,_l2,_n2,_leftover,_stat).
read_word(_,_,_,_,_,_,_,_).

read_word1(_C,_l1,_n1,_Word,_l2,_n2,_leftover,_stat) :-
        specsymb(_C),!,
        name1(_Word,[_C],_l1,_n1,_stat),
        get1(_l1,_n1,_l2,_n2,_leftover).
read_word1(_C,_l1,_n1,_Word,_l2,_n2,_leftover,_stat)  :-
        a_z0_9_(_C),!,
        read_rest_of_word(_l1,_n1,_Chars,_l2,_n2,_leftover),
        name1(_Word,[_C|_Chars],_l1,_n1,_stat).



read_word1(_C,_l1,_n1,_Word,_l3,_n3,_leftover,_stat) :-
        get1(_l1,_n1,_l2,_n2,_c1),
        read_word1(_c1,_l2,_n2,_Word,_l3,_n3,_leftover,_stat).

read_rest_of_word(_l1,_n1,_Chars,_l3,_n3,_leftover) :-
        get1(_l1,_n1,_l2,_n2,_C),
        read_rest_of_word(_C,_l2,_n2,_Chars,_l3,_n3,_leftover).

read_rest_of_word(_C,_l1,_n1,[_C|_Chars],_l2,_n2,_leftover) :-
                        a_z0_9_(_C),!,
                        read_rest_of_word(_l1,_n1,_Chars,_l2,_n2,_leftover).
read_rest_of_word(_C,_l,_n,[],_l,_n,_C) .    {not(a_z0_9_(_C)) }

name1(_Word,_L,_l0,_n0,_stat) :- name(_Word,_L) ,!.
name1(_Word,_L,_l0,_n0,[_l0,_n0,'''',_Word1,'''',' unexpected ;',
                        '''',_Word,'''',' expected']) :- name(_Word1,_L).

a_z0_9_(_c) :- a_z_(_c).
a_z0_9_(_c) :- _c >= 48 , _c =< 57 .
a_z_(_c) :- _c >= 65 , _c =< 90 .
a_z_(_c) :- _c = 95 .
a_z_(_c) :- _c >= 97 , _c =< 122 .

specsymb(21).
specsymb(22).
specsymb(59).
specsymb(60).
specsymb(61).
specsymb(62).
specsymb(91).
specsymb(93).
specsymb(123).
specsymb(125).
specsymb(126).
specsymb(_c) :- _c >= 39 , _c =< 47 .
specsymb(_c) :- _c >= 53 , _c =< 58 .

get1(_l1,_n1,_l2,_n2,_c) :-
        get0(_c),name(_a,[_c]),
        !,
        getlsub(_l1,_n1,_l2,_n2,_c).
get1(_l,_n1,_l,_n2,126) :- _n2 is _n1 + 1 .
getlsub(_l1,_n1,_l2,0,10) :- !,_l2 is _l1 + 1 .
getlsub(_l,_n1,_l,_n2,_):- _n2 is _n1 + 1 .
```

```
synt(_filename,_str,_stat) :-
        seen,
        see(_filename),
        read_word(32,1,0,_symb,_l0,_n0,_leftover0,_stat),
        sxprogram1(_symb,_l0,_n0,_leftover0,_lbsymb,_l1,_n1,_leftover1,
        _str,_stat),
        seen.

sxprogram1(_symb,_l0,_n0,_leftover0,_lbsymb,_l1,_n1,_leftover1,
        _str,_stat) :-
        var(_stat) ,! ,
        sxprogram(_symb,_l0,_n0,_leftover0,_lbsymb,_l1,_n1,_leftover1,
        _str,_stat).
sxprogram1(_,_l,_n,_,_,_,_,_,_,_).

sxprogram('~',_,_,_,_,_,_,_,_,nil,_) :- !.
sxprogram(_symb,_l0,_n0,_leftover0,_lbsymb,_l,_n,_leftover,lcomp(_comp,_lcomp),
        _stat) :-
        sxcomp(_symb,_l0,_n0,_leftover0,_nextsymb,_l1,_n1,_leftover1,
        _comp,_stat),
        read_word(_leftover1,_l1,_n1,_nextsymb1,_l2,_n2,_leftover2,_stat),
        sxprogram1(_nextsymb1,_l2,_n2,_leftover2,_lbsymb,_l,_n,
        _leftover,_lcomp,_stat) .

sxcomp(primitive,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        decprim(nprim([_l0,_n0,_s0]),_lp),_stat) :-!,
        read_word(_leftover0,_l0,_n0,_s0,_l1,_n1,_leftover1,_stat),
        identif(_s0,_l1,_n1,_stat),
        read_word(_leftover1,_l1,_n1,_s1,_l2,_n2,_leftover2,_stat),
        sxcompsub1(_s1,_l2,_n2,_leftover2,_nextsymb,_l,_n,_leftover,_lp,
sxcomp(relation,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        decrel(nrel([_l0,_n0,_s0]),
        _lp,_ldisj),_stat) :- !,
        read_word(_leftover0,_l0,_n0,_s0,_l1,_n1,_leftover1,_stat),
        identif(_s0,_l1,_n1,_stat),
        read_word(_leftover1,_l1,_n1,_s1,_l2,_n2,_leftover2,_stat),
        sxlparam1(_s1,_l2,_n2,_leftover2,')',_l31,_n31,_le0, _lp,_stat),
        read_word(_le0,_l31,_n31,'<',_l32,_n32,_le1,_stat),
        read_word(_le1,_l32,_n32,'=',_l33,_n33,_le2,_stat),
        read_word(_le2,_l33,_n33,'>',_l4,_n4,_leftover4,_stat),
        read_word(_leftover4,_l4,_n4,_symb,_l5,_n5,_leftover5,_stat),
       sxdisj2(_symb,_l5,_n5,_leftover5,_nextsymb,_l,_n,_leftover,_ldisj,_stat).
sxcomp(_,_l,_n,_,_,_,_,_,_,
        [_l,_n,'''relation'' or ''primitive'' or end of file expected']).

sxdisj2(_symb,_l5,_n5,_leftover5,_nextsymb,_l,_n,_leftover,_ldisj,_stat) :-
        var(_stat),!,
        sxdisj(_symb,_l5,_n5,_leftover5,_nextsymb,_l,_n,_leftover,_ldisj,_stat).
sxdisj2(_,_,_,_,_,_,_,_,_,_).

sxdisj(exist,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        ldisj(conjpref(_lp,_lconj),_ldisj),_stat) :- !,
        read_word(_leftover0,_l0,_n0,_symb,_l1,_n1,_leftover1,_stat),
        sxprefixe1(_symb,_l1,_n1,_leftover1,':',_l2,_n2,_leftover2,
        _lp,_stat),
        read_word(_leftover2,_l2,_n2,_s0,_l3,_n3,_leftover3,_stat),
        sxconj1(_s0,_l3,_n3,_leftover3,_nextsymb1,_l4,_n4,_leftover4,
        _lconj,_stat),
        sxdisj3(_nextsymb1,_l4,_n4,_leftover4,_nextsymb,_l,_n,_leftover,
        _ldisj,_stat).
sxdisj(_symb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,ldisj(conjpref(nil,
        _lconj),_ldisj),_stat) :- !,
        identif(_symb,_l0,_n0,_stat),
        sxconj1(_symb,_l0,_n0,_leftover0,_nextsymb1,_l1,_n1,_leftover1,
        _lconj,_stat),
        sxdisj3(_nextsymb1,_l1,_n1,_leftover1,_nextsymb,_l,_n,_leftover,
        _ldisj,_stat).
sxdisj(_,_l,_n,_,_,_,_,_,_,[_l,_n,'''exist'' or an atom expected']).

sxdisj3(_nextsymb1,_l1,_n1,_leftover1,_nextsymb,_l,_n,_leftover,_ldisj,_stat):-
        var(_stat),!,
        sxdisj1(_nextsymb1,_l1,_n1,_leftover1,_nextsymb,_l,_n,_leftover,
        _ldisj,_stat).
sxdisj3(_,_,_,_,_,_,_,_,_,_).

sxcompsub1(_s1,_l2,_n2,_leftover2,_nextsymb,_l,_n,_leftover,_lp,_stat) :-
        var(_stat) , !,
        sxcompsub(_s1,_l2,_n2,_leftover2,_nextsymb,_l,_n,_leftover,_lp,_stat).
sxcompsub1(_,_,_,_,_,_,_,_,_,_).

sxcompsub('(',_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_lp,_stat) :- !,
        sxlparam1('(',_l0,_n0,_leftover0,')',_l2,_n2,_b, _lp,_stat),
        read_word(_b,_l2,_n2,_nextsymb,_l,_n,_leftover,_stat).
sxcompsub('.',_l0,_n0,_leftover0,'.',_l0,_n0,_leftover0,nil,_stat) :- !.
sxcompsub(_,_l,_n,_,_,_,_,_,_,[_l, _n , '''.'' or ''('' expected. ']).

sxdisj1(ou,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_ldisj,_stat) :- !,
        read_word(_leftover0,_l0,_n0,_symb,_l1,_n1,_leftover1,_stat),
        sxdisj(_symb,_l1,_n1,_leftover1,_nextsymb,_l,_n,_leftover,_ldisj,_stat).

sxdisj1('.',_l,_n,_x,'.',_l,_n,_x,nil,_stat) :- !.
sxdisj1(_,_l,_n,_,_,_,_,_,_,[_l,_n , '''ou'' or ''.'' expected']).

sxconj1(_symb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover
        ,conj(_liter,_lconj),_stat) :-
        var(_stat),!,
        sxconj(_symb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        conj(_liter,_lconj),_stat) .
sxconj1(_,_,_,_,_,_,_,_,_,_).

sxconj(_symb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,conj(_liter,_lconj),
        _stat) :-
        sxstructure(_symb,_l0,_n0,_leftover0,_nextsymb1,_l1,_n1,_leftover1,
        _liter,_stat),
        read_word(_leftover1,_l1,_n1,_nextsymb2,_l2,_n2,_leftover2,_stat),
        sxconjsub1(_nextsymb2,_l2,_n2,_leftover2,_nextsymb,_l,_n,_leftover,
        _lconj,_stat).

sxconjsub1(_nextsymb2,_l2,_n2,_leftover2,_nextsymb,_l,_n,_leftover,_lconj,
        _stat) :-
        var(_stat),!,
        sxconjsub(_nextsymb2,_l2,_n2,_leftover2,_nextsymb,_l,_n,_leftover,
```

```
        _lconj,_stat).
sxconjsub1(_,_,_,_,_,_,_,_,_,_).

sxconjsub(et,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_lconj,_stat) :- !,
        read_word(_leftover0,_l0,_n0,_symb,_l1,_n1,_leftover1,_stat),
        sxconj(_symb,_l1,_n1,_leftover1,_nextsymb,_l,_n,_leftover,_lconj,_stat).
sxconjsub(ou,_l,_n,_x,ou,_l,_n,_x,nil,_stat) :- !.
sxconjsub('.',_l,_n,_x,'.',_l,_n,_x,nil,_stat) :- !.
sxconjsub(_,_l,_n,_,_,_,_,_,_,[_l, _n ,'''et'' or ''ou'' or ''.'' expected']).

sxstructure(_symb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_struct,_stat):-
        read_word(_leftover0,_l0,_n0,_nextsymb1,_l1,_n1,_leftover1,_stat),
        sxstructuresub(_symb,_nextsymb1,_l1,_n1,_leftover1,_nextsymb,_l,_n,
        _leftover,_struct,_stat).

sxstructuresub(_symb,'=',_l0,_n0,_leftover0,_x,_l,_n,_leftover,
        unif([_l0,_n0,_symb],[_l0,_n0,_x]),_stat) :- !,
        not(_symb = 'not'),
        identif(_symb,_l0,_n0,_stat),
        read_word(_leftover0,_l0,_n0,_x,_l,_n,_leftover,_stat),
        identif(_x,_l,_n,_stat).
sxstructuresub('not','(',_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        neg(_struct),_stat) :- !,
        read_word(_leftover0,_l0,_n0,_symb1,_l1,_n1,_leftover1,_stat),
        sxstructure(_symb1,_l1,_n1,_leftover1,_nextsymb1,_l3,_n3,_leftover3,
        _struct,_stat),
        read_word(_leftover3,_l3,_n3,_nextsymb,_l,_n,_leftover,_stat),
        err1(_nextsymb,_l,_n,_stat).

err1(_nextsymb,_l,_n,_stat) :- var(_stat),!, err(_nextsymb,_l,_n,_stat).
err1(_,_,_,_).
err(et,_,_,_) :- !.
err(ou,_,_,_) :- !.
err('.',_,_,_) :- !.
err(_,_l,_n,[_l,_n,'''.'' or et or ou  expected']).

sxstructuresub(_symb,'(',_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        appel([_l0,_n0,_symb],_larg),_stat) :- !,
        _n01 is _n0 - 2 ,
        identif(_symb,_l0,_n01,_stat),
        not(_symb = 'not'),
        read_word(_leftover0,_l0,_n0,_x,_l1,_n1,_leftover1,_stat),
        sxlarg(_x,_l1,_n1,_leftover1,_nextsymb,_l,_n,_leftover,_larg,_stat).
sxstructuresub(_,_,_l,_n,_,_,_,_,_,_,_,
        [_l , _n , ' ''=''' or ''('' or ''not'' expected ']).

sxlparam1(_symb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_lp,_stat) :-
        var(_stat),!,
        sxlparam(_symb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_lp,_stat).
sxlparam1(_,_,_,_,_,_,_,_,_,_).

sxlparam('(',_l01,_n01,_leftover01,_nextsymb,_l,_n,_leftover,_lp,_stat) :- !,
        read_word(_leftover01,_l01,_n01,_bsymb,_l0,_n0,_leftover0,_stat),
        sxparam1(_bsymb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_lp,_stat).
sxlparam(_,_l,_n,_,_,_,_,_,_,[_l,_n,'''('' expected']).

sxparam1(_bsymb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_lp,_stat) :-
        var(_stat) , ! ,
        sxparam(_bsymb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_lp,_stat).
sxparam1(_,_,_,_,_,_,_,_,_,_).

sxparam(_bsymb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        [v(_l0,_n0,_bsymb)-_y|_T],_stat) :-
        identif(_bsymb,_l0,_n0,_stat),
        read_word(_leftover0,_l0,_n0,':',_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,_y,_l2,_n2,_leftover2,_stat),
        identif(_y,_l2,_n2,_stat) ,
        read_word(_leftover2,_l2,_n2,_symb,_l3,_n3,_leftover3,_stat),
        sxlparamsub1(_symb,_l3,_n3,_leftover3,_nextsymb,_l,_n,_leftover,
        _T,_stat).

sxlparamsub1(_symb,_l3,_n3,_leftover3,_nextsymb,_l,_n,_leftover,_lp,_stat) :-
        var( _stat) ,! ,
        sxlparamsub(_symb,_l3,_n3,_leftover3,_nextsymb,_l,_n,_leftover,_lp,
        _stat).
sxlparamsub1(_,_,_,_,_,_,_,_,_,_).

sxlparamsub(',',_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_lp,_stat ) :- !,
        read_word(_leftover0,_l0,_n0,_x,_l1,_n1,_leftover1,_stat),
        identif(_x,_l1,_n1,_stat) ,
        sxparam1(_x,_l1,_n1,_leftover1,_nextsymb,_l,_n,_leftover,_lp,_stat ).
sxlparamsub(')',_l,_n,_x,')',_l,_n,_x,nil,_stat) :- !. {entete de relation }
sxlparamsub(_,_l,_n,_,_,_,_,_,_,[_l,_n,''')'' or  '','' expected']).

sxprefixe1(_bsymb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        [v(_l0,_n0,_bsymb)-_y|_T],_stat) :-
        var(_stat),!,
        sxprefixe(_bsymb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        [v(_l0,_n0,_bsymb)-_y|_T], _stat).
sxprefixe1(_,_,_,_,_,_,_,_,_,_).

sxprefixe(_bsymb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        [v(_l0,_n0,_bsymb)-_y|_T] , _stat) :-
        identif(_bsymb,_l0,_n0,_stat),
        read_word(_leftover0,_l0,_n0,':',_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,_y,_l2,_n2,_leftover2,_stat),
        identif(_y,_l2,_n2,_stat) ,
        read_word(_leftover2,_l2,_n2,_symb,_l3,_n3,_leftover3,_stat),
        sxprefixesub1(_symb,_l3,_n3,_leftover3,_nextsymb,_l,_n,
        _leftover, _T ,_stat).

sxprefixesub1(_symb,_l3,_n3,_leftover3,_nextsymb,_l,_n,_leftover,_lp,_stat) :-
        var( _stat) ,! ,
        sxprefixesub(_symb,_l3,_n3,_leftover3,_nextsymb,_l,_n,_leftover,
        _lp,_stat).
sxprefixesub1(_,_,_,_,_,_,_,_,_,_).

sxprefixesub(',',_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_lp,_stat ) :- !,
        read_word(_leftover0,_l0,_n0,_x,_l1,_n1,_leftover1,_stat),
        identif(_x,_l1,_n1,_stat) ,
        sxprefixe1(_x,_l1,_n1,_leftover1,_nextsymb,_l,_n,_leftover,_lp,_stat).
sxprefixesub(':',_l,_n,_x,':',_l,_n,_x,nil,_stat) :- ! . { prefixe }
```

```
sxprefixesub(_,_l,_n,_,_,_,_,_,_,[_l,_n,''':'' or '','' expected']).

sxlargl(_bsymb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        [v(_l0,_n0,_bsymb)|_T],_stat):-
        var(_stat),!,
        sxlarg(_bsymb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        [v(_l0,_n0,_bsymb)|_T],_stat).
sxlargl(_,_,_,_,_,_,_,_,_,_).

sxlarg(_bsymb,_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,
        [v(_l0,_n0,_bsymb)|_T],_stat) :-
        identif(_bsymb,_l0,_n0,_stat),
        read_word(_leftover0,_l0,_n0,_symb,_l1,_n1,_leftover1,_stat),
        sxlargsub1( _symb,_l1,_n1,_leftover1,_nextsymb,_l,_n,
        _leftover,_T,_stat).

sxlargsub1( _symb,_l1,_n1,_leftover1,_nextsymb,_l,_n,_leftover,_larg,_stat) :-
        var(_stat),!,
        sxlargsub( _symb,_l1,_n1,_leftover1,_nextsymb,_l,_n,_leftover,_larg,
        _stat).
sxlargsub1(_,_,_,_,_,_,_,_,_,_).

sxlargsub(',',_l0,_n0,_leftover0,_nextsymb,_l,_n,_leftover,_larg,_stat) :- !,
        read_word(_leftover0,_l0,_n0,_x,_l1,_n1,_leftover1,_stat),
        identif(_x,_l1,_n1,_stat),
        sxlargl(_x,_l1,_n1,_leftover1,_nextsymb,_l,_n,_leftover,
sxlargsub(')',_l,_n,_x,')',_l,_n,_x,nil,_stat) :- !.
sxlargsub(_,_l,_n,_,_,_,_,_,_,[_l,_n,''',''' or '')'' expected']).
```

```
pvsem(_filename):-
        synt(_filename,_str,_stat),

        sem(_str,_TabProc,_Lappel,_stat1 ,_stat),
        trt_diagn_err(_filename,[_TabProc,_Lappel],_stat).

sem(_str,_TabProc,_Lappel,_stat1,_stat2) :-
        var(_stat1),!,
        seml(_str,_TabProc,_Lappel,_stat2).
sem(_,_,_,_stat1,_stat1).

seml(_str,_TabProc,_Lappel,_stat) :-
        parcourir(_str,_TabProc1,_Lappel,_stat),
        inserer([0,0,true],[],_TabProc1,_TabProc2,_stat),
        inserer([0,0,false],[],_TabProc2,_TabProc,_stat),
        look_up(_TabProc,_Lappel,_stat).

parcourir(_str,_TabProc,_Lappel,_stat) :-
        var(_stat) ,!,
        parcourir1(_str,_TabProc,_Lappel,_stat).
parcourir(_,_,_,_).

parcourir1(nil,[],[],_stat) :- !.
parcourir1(lcomp(decrel(nrel(_N),_LVarTyp,_ldisj),_lcomp),_TabProc,_Lappel,
       _stat) :- !,
        construire(_LVarTyp,_ldisj,_L1,_stat),
        parcourir(_lcomp,_A1,_L2,_stat),
        inserer(_N,_LVarTyp,_A1,_TabProc,_stat),
        appendl(_L1,_L2,_Lappel,_stat).
parcourir1(lcomp(decprim(nprim(_N),_LVarTyp),_lcomp),_TabProc,_Lappel,_stat) :-
        parcourir(_lcomp,_A1,_Lappel,_stat),
        inserer(_N,_LVarTyp,_A1,_TabProc,_stat).

inserer(_N,_L,_A1,_T,_stat) :-
        var(_stat) ,!,
        insererl(_N,_L,_A1,_T,_stat).
inserer(_,_,_,_,_).

insererl(_N,_LVarTyp,_A,_TabProc,_stat) :-
        listype(_LVarTyp,_ltyp),
        inserersub(_N,_ltyp,_A,_TabProc,_stat).

inserersub(_N,_L,nil,TabProc(nil,_N,_L,nil),_stat) :- !.
inserersub([_l0,_n0,_N],_L,TabProc(_left,[_l1,_n1,_M],_L1,_right),
        TabProc(_leftOrd,[_l1,_n1,_M],_L1,_right),_stat) :-
        inf(_N,_M),!,
        inserersub([_l0,_n0,_N],_L,_left,_leftOrd,_stat).
inserersub([_l0,_n0,_N],_L,TabProc(_left,[_l1,_n1,_M],_L1,_right),
        TabProc(_left,[_l1,_n1,_M],_L1,_rightOrd),_stat) :-
        inf(_M,_N),!,
        inserersub([_l0,_n0,_N],_L,_right,_rightOrd,_stat).
inserersub([_l0,_n0,_M],_L,TabProc(_left,[_l1,_n1,_M],_L1,_right),
        TabProc(_left,[_l1,_n1,_M],_L1,_rightOrd),_stat) :-
        size(_L,_s),
        size(_L1,_s1),
        not(_s = _s1),!,
        _rightOrd = (nil,[_l0,_n0,_M],_L,_right).
inserersub([_,_,_M],_L,TabProc(_left,[_l0,_n0,_M],_L1,_right),_,_stat) :-
        size(_L,_l),
        size(_L1,_l),
        _stat = [_l0,_n0,_M,'/',_l,' relation or primitive declared twice'].

listype([],[]) :- !.
listype([v(_,_,_)-_t|_T],[_t|_Typ]) :-
        listype(_T,_Typ).

look_up(_TabProc,_Lappel,_stat) :-
        var(_stat),!,
        look_up1(_TabProc,_Lappel,_stat).
look_up(_,_,_).

look_up1(_TabProc,[],_stat) :- !.
look_up1(_TabProc,[_H|_T],_stat):-
        look_for(_TabProc,_H,_stat),
        look_up(_TabProc,_T,_stat).

look_for(TabProc(_left,[_l0,_n0,_M],_L1,_right),appel([_l1,_n1,_M],_L2),
        _stat) :-
        !,
        argum(_l1,_n1,_M,_L1,_L2,_stat).

look_for(TabProc(_left,[_l0,_n0,_N],_L1,_right),appel([_l1,_n1,_M],_L2),
        _stat) :-
        inf(_M,_N),
        !,
        look_for(_left,appel([_l1,_n1,_M],_L2),_stat).
look_for(TabProc(_left,[_l0,_n0,_N],_L1,_right),appel([_l1,_n1,_M],_L2),
        _stat):-
        {not(inf(_M,_N)),}
        !,
        look_for(_right,appel([_l1,_n1,_M],_L2),_stat).
look_for(nil,appel([_l0,_n0,_M],_L2),[_l0,_n0,' unknoun call : ',_M,'/',_l]):-
        size(_L2,_l).

inf(_x,_y) :-
        name(_x,_l),
        name(_y,_m),
        infx(_l,_m).

infx([],[_|_]) :- !.
infx([_t|_T1],[_t|_T2]) :- !,infx(_T1,_T2).
infx([_t1|_],[_t2|_]) :- _t1 < _t2 .


argum(_,_,_M,_L,_L,_stat) :- !.
argum(_l0,_n0,_M,_L1,_L2,_stat) :-
        size(_L1,_l1),
        size(_L2,_l2),
        _l1 = _l2,
        first_diff(_L1,_L2,_v),
        _stat = [_l0,_n0,'type mismach with ',_v,'in call to',_M,'/',_l2].
```

```
first_diff([_v1|_T1],[_v2|_T2],_v2) :-
        not(_v1 = _v2),!.
first_diff([_v1|_T1],[_v1|_T2],_v2) :-
        first_diff(_T1,_T2,_v2).

size([],0) :- !.
size([_t|_T],_N) :-
        size(_T,_M),
        _N is _M + 1 .
```

```
analyse(_filename,_tabinst,_Lapp) :-
        syntconj(_filename,_conj,_stat) ,
        analysconj(_tabinst,_conj,_Lapp).

analysconj(_tabinst,_conj,_Lapp) :-
        map(_conj,_conjlist),
        constrVarTabl(_conjlist,_tabvar),
        analysconjsub(_tabinst,_tabvar,_conjlist,_Lapp).

analysconjsub(_T,_T1,[],[]) :-!.
analysconjsub(_tabinst1,_tabvar1,[_p|_Tconjlist],_Lapp) :-
     analyscall(_p,_tabvar1,_tabinst1,_Legal1,_pnew,_Legal2,_tabvar2,_tabinst2),
     analysconjsub(_tabinst2,_tabvar2,_Tconjlist,_Lapp1),
     append(_Legal1,[_pnew|_Legal2],_Lapp0),
     append(_Lapp0,_Lapp1,_Lapp).

map(nil,[]) :-!.
map(conj(appel([_,_,_nrel],_L),_conj),[_appel|_T]) :-
        list(_L,_Larg),
        _appel =..[_nrel|_Larg],
        map(_conj,_T).
list([],[]) :- !.
list([v(_,_,_v)|_T],[_v|_T1]) :-
        list(_T,_T1).

constrVarTabl([],[]) :- !.
constrVarTabl([_appel|_T],_L) :-
        enter(_appel,_L1),
        constrVarTabl(_T,_L2),
        add(_L1,_L2,_L).

add(_L,[],_L) :- !.
add([],_L,_L) :-!.
add([_v1-_n1|_T1],_L1,_L) :-
        addsub(_v1,_n1,_L1,_L21),
        add(_T1,_L21,_L).

addsub(_v,_n,[],[_v-_n]) :-! .
addsub(_v1,_n1,[_v1-_n2|_T],[_v1-_n3|_T]) :-!,
        _n3 is _n1 + _n2 .
addsub(_v1,_n1,[_v2-_n2|_T],[_v2-_n2|_T1]) :-
        addsub(_v1,_n1,_T,_T1).

enter(_appel,_L):-
        _appel =..[_f|_Larg],
        entersub(_Larg,_L).
entersub([],[]) :- !.
entersub([_v1|_T1],_L1) :-
        entersub(_T1,_L12),
        addsub(_v1,1,_L12,_L1).
```

```
 :- dynamic(bestsofar/2).

datanalyser(_filename,_tabinst,_bestconj,_bestcost) :-
        syntconj(_filename,_conj,_stat),
        map(_conj,_conjlist),
        initbase ,
        generatePerm(_conjlist,_cl),
        isbest(_tabinst,_cl).

datanalyser(_filename,_tabinst,_bestconj,_bestcost) :-
        bestsofar(_bestconj,_bestcost).

initbase :-
        retractall(bestsofar(_x,_y)),
        assert(bestsofar([],[10000,0,0])).

generatePerm([],[]) .
generatePerm([_X|_Xs],_Ys1):-
        generatePerm(_Xs,_Ys),
        insert(_Ys,_X,_Ys1).

insert(_L,_X,[_X|_L]).
insert([_H|_T],_X,[_H|_L]):-
        insert(_T,_X,_L).


isbest(_tabinst,_c) :-
        checkbest(_tabinst,_c),
        fail.

checkbest(_tabinst,_c) :-

        takeIt(_tabinst,_c,_bettercost),
        retract(bestsofar(_cbest,_bcost)),
        assert(bestsofar(_c,_bettercost)).

takeIt(_tabinst,_c,_bettercost) :-
        constrVarTabl(_c,_tabvar),
        takeItsub(_tabinst,_tabvar,_c,[0,0,0],_bettercost).

takeItsub(_,_,[],_acccost,_bettercost) :-!,
        bestsofar(_,_cost),
        lessThan(_acccost,_cost),
        _acccost = _bettercost.
takeItsub(_tabinst,_tabvar,_,_acccost,_bettercost) :-
        bestsofar(_,_cost),
        lessThan(_cost,_acccost),
        !,
        fail.
takeItsub(_tabinst,_tabvar,[_p|_cl],_acccost,_bettercost) :-
        _p=..[_f|_],
        tabin(_f,_Lad),
        analyscall(_p,_Lad,_tabvar,_tabinst,_Legal1,_,_Legal2,_tabvar1,
        _tabinst1),
        cost(_Legal1,_Legal2,_cost1),
        sum(_cost1 , _acccost,_newcost ),
        takeItsub(_tabinst1,_tabvar1,_cl,_newcost,_bettercost).

cost(_Legal1,_Legal2,_cost) :-
        cout(_Legal1,_cost1),
        size(_Legal2,_s),
        append(_cost1,[_s],_cost).

cout(_Legal1,_cost) :-
        cout(_Legal1,[0,0],_cost).
cout([],_L,_L1) :- !,
        _L = _L1.
cout([egal(_,_)|_T],[_g,_e],[_g1,_e1]):- !,
        _esuiv is _e + 1 ,
        cout(_T,[_g,_esuiv],[_g1,_e1]).
cout([generate(_)|_T],[_g,_e],[_g1,_e1]):- !,
        _gsuiv is _g + 1 ,
        cout(_T,[_gsuiv,_e],[_g1,_e1]).
lessThan([_g|_L1],[_g|_L2]) :- !,
        lessThan(_L1,_L2).
lessThan([_g1|_L1],[_g2|_L2]) :-
        _g1 < _g2 .

sum([],[],[]) :- !.
sum([_n|_T],[_m|_R],[_nsum|_Sum]) :-
        _nsum is _n + _m,
        sum(_T,_R,_Sum).
```

```
 :- dynamic(bestsofar/2).

datanalyser1(_filename,_tabinst,_bestconj,_bestcost) :-
        syntconj(_filename,_conj,_stat),
        map(_conj,_conjlist),
        initbase,
        constrVarTabl(_conjlist,_tabvar),
        generatePrefPerm(_tabvar,_tabinst,[],_conjlist,[0,0,0]).

datanalyser1(_filename,_,_bestconj,_bestcost) :-
        bestsofar(_bestconj,_bestcost).

generatePrefPerm(_,_,_,_,_acccost):-
        bestsofar(_bestconj,_bestcost),
        lessThan(_bestcost,_acccost),
        !,
        fail.
generatePrefPerm(_,_,_c,[],_acccost) :-
        bestsofar(_bestconj,_bestcost),
        lessThan(_acccost,_bestcost),
        !,
        commit(_c,_acccost),
        fail.
generatePrefPerm(_tabvar,_tabinst,_Pref,_Suff,_acccost) :-
        selectone(_Suff,_Al,_SuffWithout),
        bestsofar(_,_currentcost),
        lessThan(_acccost,_currentcost),
        _Al =.. [_f|_],
        tabin(_f,_Lad),
        analyscall(_Al,_Lad,_tabvar,_tabinst,_Legall,_,_Legal2,
        _tabvar1,_tabinst1),
        cost(_Legall,_Legal2,_cost1),
        sum(_acccost,_cost1,_newcost),
        append(_Pref,[_Al],_Prefnew),
        generatePrefPerm(_tabvar1,_tabinst1,_Prefnew,_SuffWithout,
        _newcost).

selectone([_Al|_T],_Al,_T) .
selectone([_H|_T],_Al,[_H|_T1]) :-
        selectone(_T,_Al,_T1).

commit(_c,_cost) :-
        retract(bestsofar(_,_)),
        assert(bestsofar(_c,_cost)).
```

```
sxdatatypedef(_filename,_str,_stat) :-
        var(_stat),!,
        sxdatatypedef1(_filename,_str,_stat).
sxdatatypedef(_filename,_str,_stat).

sxdatatypedef1(_filename,_str,_stat) :-
        seen,
        see(_filename),
        read_word(32,1,0,_symb,_l0,_n0,_leftover0,_stat),
        sxparser(_symb,_l0,_n0,_leftover0,_lbsymb,_l1,_n1,_leftover1,_str,
        _stat),
        seen.
sxparser(_symb,_l0,_n0,_leftover0,_lbsymb,_l,_n,_leftover,_info,_stat) :-
        var(_stat),!,
        sxparser1(_symb,_l0,_n0,_leftover0,_lbsymb,_l,_n,_leftover,_info,_stat).
sxparser(_symb,_l0,_n0,_leftover0,_lbsymb,_l,_n,_leftover,_info,_stat).
sxparser1(_symb,_l0,_n0,_leftover0,_lbsymb,_l,_n,_leftover,
        adtdef(_names,_Ldomoperdecl),_stat) :-
        sxhead(_symb,_l0,_n0,_leftover0,_s1,_l1,_n1,_leftover1,_names,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxdomoperdecl(_s2,_l2,_n2,_leftover2,_s3,_l3,_n3,_leftover3,_names,
        _Ldomoperdecl,_stat),
        sxtail(_s3,_l3,_n3,_leftover3,_lbsymb,_l,_n,_leftover,_stat).
sxhead(_s0,_l0,_n0,_leftover0,_symb,_l,_n,_leftover,_info,_stat) :-
        var(_stat),!,
        sxhead1(_s0,_l0,_n0,_leftover0,_symb,_l,_n,_leftover,_info,_stat).
sxhead(_s0,_l0,_n0,_leftover0,_symb,_l,_n,_leftover,_info,_stat).

sxhead1(Abstract,_l0,_n0,_leftover0,_symb,_l,_n,_leftover,
        adthead(v(_s,_l2,_n2),_Lnames),_stat) :- !,
        read_word(_leftover0,_l0,_n0,data,_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,type,_l2,_n2,_leftover2,_stat),
        read_word(_leftover2,_l2,_n2,_s,_l3,_n3,_leftover3,_stat),
        identif(_s,_l2,_n2,_stat),
        read_word(_leftover3,_l3,_n3,';',_l4,_n4,_leftover4,_stat),
        read_word(_leftover4,_l4,_n4,uses,_l5,_n5,_leftover5,_stat),
        sxlistnames(',',_l5,_n5,_leftover5,_symb,_l,_n,_leftover,_Lnames,
        _stat).
sxhead1(_,_l0,_n0,_,_,_,_,_,_,[_l0,_n0,'''Abstract'' expected']).

sxlistnames(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lnames,_stat) :-
        var(_stat),!,
        sxlistnames1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lnames,_stat).
sxlistnames(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lnames,_stat).

sxlistnames1(',',_l0,_n0,_leftover0,_s,_l,_n,_leftover,
        [v(_s1,_l1,_n1)|_Lnames],_stat):- !,
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        identif(_s1,_l1,_n1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxlistnames(_s2,_l2,_n2,_leftover2,_s,_l,_n,_leftover,_Lnames,_stat).
sxlistnames1(';',_l,_n,_leftover,';',_l,_n,_leftover,[],_stat) :-!.
sxlistnames1(_,_l0,_n0,_,_,_,_,_,[_l0,_n0,'''',''' or ''';''' expected']).

sxdomoperdecl(_symb,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_L,_stat):-
        var(_stat),!,
        sxdomoperdecl1(_symb,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,
        _L,_stat).
sxdomoperdecl(_symb,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_L,_stat).

sxdomoperdecl1(end,_l,_n,_leftover,end,_l,_n,_leftover,_names,[],_stat):- !.
sxdomoperdecl1(domain,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,
        [domoperdecl(_Ldomdecl,_Loperdecl)|_T],_stat):- !,
        sxdomdecl(_symb,_l0,_n0,_leftover0,_s1,_l1,_n1,_leftover1,_Ldomdecl
        ,_stat),
        sxoperdecl(_s1,_l1,_n1,_leftover1,_s2,_l2,_n2,_leftover2,_Loperdecl,
        _names,_filename,_stat),
        sxdomoperdecl(_s2,_l2,_n2,_leftover2,_s,_l,_n,_leftover,_names,
        _T,_stat).
sxdomoperdecl1(_,_l,_n,_,_,_,_,_,_,_,[_l,_n,'''domain'' or ''end'' expected']).
sxdomdecl(_symb,_l0,_n0,_leftover0,_s1,_l1,_n1,_leftover1,_Ldomdecl,_stat) :-
        var(_stat),!,
        sxdomdecl1(_symb,_l0,_n0,_leftover0,_s1,_l1,_n1,_leftover1,_Ldomdecl
        ,_stat).
sxdomdecl(_symb,_l0,_n0,_leftover0,_s1,_l1,_n1,_leftover1,_Ldomdecl,_stat).

sxdomdecl1(domain,_l0,_n0,_leftover0,_s,_l,_n,_leftover,[domdecl(_Lnames,
                                                    _impltype)|_T],_stat) :- !,
        sxlistnames(',',_l0,_n0,_leftover0,_s3,_l3,_n3,_leftover3,_Lnames,
                                                                    _stat),
        read_word(_leftover3,_l3,_n3,_s31,_l31,_n31,_leftover31,_stat),
        sxdomdescrip(_s31,_l31,_n31,_leftover31,_s4,_l4,_n4,_leftover4,_stat),
        read_word(_leftover4,_l4,_n4,_s41,_l41,_n41,_leftover41,_stat),
        sxdomimpl(_s41,_l41,_n41,_leftover41,_s5,_l5,_n5,_leftover5,_impltype,
        _stat),
        read_word(_leftover5,_l5,_n5,_s6,_l6,_n6,_leftover6,_stat),
        sxdomdecl(_s6,_l6,_n6,_leftover6,_s,_l,_n,_leftover,_T,_stat).
sxdomdecl1(operation,_l,_n,_leftover,operation,_l,_n,_leftover,[],_stat) :- !.
sxdomdecl1(_,_l,_n,_,_,_,_,_,_,[_l,_n,'''domain'' or ''operation'' expected']).

sxdomdescrip(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        var(_stat),!,
        sxdomdescrip1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).
sxdomdescrip(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).

sxdomdescrip1(description,_l0,_n0,_leftover0,';',_l,_n,_leftover,_stat) :- !,
        read_word(_leftover0,_l0,_n0,'{',_l1,_n1,_leftover1,_stat),
        sxcomment('{',_l1,_n1,_leftover1,'}',_l2,_n2,_leftover2,_stat),
        read_word(_leftover2,_l2,_n2,';',_l,_n,_leftover,_stat).
sxdomdescrip1(_,_l,_n,_,_,_,_,_,[_l,_n,'''description'' expected']).


sxdomimpl(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_impltype,_stat) :-
        var(_stat),!,
        sxdomimpl1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_impltype,_stat).
sxdomimpl(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_impltype,_stat).

sxdomimpl1(implementation,_l0,_n0,_leftover0,';',_l,_n,_leftover,_impltype,
        _stat) :- !,
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        sxdomimplsub(_s1,_l1,_n1,_leftover1,_s,_l,_n,_leftover,_impltype,
        _stat).
```

```
sxdomimpl1(_,_l,_n,_,_,_,_,_,_,[_l,_n,'''implementation'' expected']).

sxdomimplsub(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_impltyp,_stat) :-
        var(_stat),!,
        sxdomimplsub1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_impltyp,_stat).
sxdomimplsub(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_impltyp,_stat).

sxdomimplsub1(Is,_l0,_n0,_leftover0,';',_l,_n,_leftover,ground,_stat) :- !,
        read_word(_leftover0,_l0,_n0,ground,_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,'{',_l2,_n2,_leftover2,_stat),
        sxcomment('{',_l2,_n2,_leftover2,'}',_l3,_n3,_leftover3,_stat),
        read_word(_leftover3,_l3,_n3,';',_l,_n,_leftover,_stat).
sxdomimplsub1('{',_l0,_n0,_leftover0,';',_l,_n,_leftover,comment,_stat) :- !,
        sxcomment('{',_l0,_n0,_leftover0,'}',_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,';',_l,_n,_leftover,_stat).
sxdomimplsub1(_,_l,_n,_,_,_,_,_,_,
       [_l,_n,'''is ground '' or a comment expected']).

sxoperdecl(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_T,_names,_filename,
        _stat):-
        var(_stat),!,
        sxoperdecl1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_T,_names,
        _filename,_stat).
sxoperdecl(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_T,_names,_filename,_stat).

sxoperdecl1(operation,_l0,_n0,_leftover0,_s,_l,_n,_leftover,[_H|_T],_names,
        _filename,_stat):-
        !,
        sxopersdecl(operation,_l0,_n0,_leftover0,_s1,_l1,_n1,_leftover1,_H,
        _filename,_stat),
        sxoperdecl(_s1,_l1,_n1,_leftover1,_s,_l,_n,_leftover,_T,_names,
        _filename,_stat).
sxoperdecl1(Abstract,_l0,_n0,_leftover0,_s,_l,_n,_leftover,[Abstract|_T],_,_
        ,_stat):- !,
        sxmodule(Abstract,_l0,_n0,_leftover0,_s1,_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxoperdecl(_s2,_l2,_n2,_leftover2,_s,_l,_n,_leftover,_T,_stat).
sxoperdecl1(BIM,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_T,_names,_filename,
        _stat):- !,
        sxmodule(BIM,_l0,_n0,_leftover0,_s1,_l1,_n1,_leftover1,_names,
        _filename,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxoperdecl(_s2,_l2,_n2,_leftover2,_s,_l,_n,_leftover,_T,_names,_,_stat).
sxoperdecl1(domain,_l,_n,_leftover,domain,_l,_n,_leftover,[],_,_,_stat):- !.
sxoperdecl1(end,_l,_n,_leftover,end,_l,_n,_leftover,[],_,_,_stat):- !.

sxoperdecl1(_,_l,_n,_,_,_,_,_,_,_,_,_,
        [_l,_n,'''operation'',''Abstract'',''BIM'' or ''end'' expected']).
sxopersdecl(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_filename,_stat) :-
        var(_stat),!,
        sxopersdecl1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_filename,
        _stat).
sxopersdecl(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_filename,_stat).

sxopersdecl1(operation,_l0,_n0,_leftover0,_s,_l,_n,_leftover,
        operdecl(_opername,_Lparam,_filename,_Lpri),_filename,_stat) :- !,
        sxoperhead(operation,_l0,_n0,_leftover0,_s1,_l1,_n1,_leftover1,
        _opername,_Lparam,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxoperdescrip(_s2,_l2,_n2,_leftover2,_s3,_l3,_n3,_leftover3,_stat),
        read_word(_leftover3,_l3,_n3,_s4,_l4,_n4,_leftover4,_stat),
        read_word(_leftover4,_l4,_n4,_s41,_l41,_n41,_leftover41,_stat),
        newfilename(_filename),
        sxoperimpl(_s41,_l41,_n41,_leftover41,_s,_l,_n,_leftover,_Lpri,_stat).

sxoperhead(_s0,_l0,_n0,_left0,_s,_l,_n,_left,_opername,_Lparam,_stat) :-
        var(_stat),!,
        sxoperhead1(_s0,_l0,_n0,_left0,_s,_l,_n,_left,_opername,_Lparam,_stat).
sxoperhead(_s0,_l0,_n0,_left0,_s,_l,_n,_left,_opername,_Lparam,_stat).

sxoperhead1(operation,_l0,_n0,_leftover0,_s,_l,_n,_leftover,v(_opname,_l0,_n0),
        _Lparam,_stat) :-!,
        read_word(_leftover0,_l0,_n0,_opname,_l1,_n1,_leftover1,_stat),
        identif(_opname,_l0,_n0,_stat),
        read_word(_leftover1,_l1,_n1,'(',_l2,_n2,_leftover2,_stat),
        sxparamlist(';',_l2,_n2,_leftover2,_s,_l,_n,_leftover,_Lparam,_stat).
sxoperhead1(_,_l,_n,_,_,_,_,_,_,[_l,_n,'''operation'' expected']).

sxparamlist(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_stat) :-
        var(_stat),!,
        sxparamlist1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_stat).
sxparamlist(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_stat).

sxparamlist1(';',_l0,_n0,_leftover0,_s,_l,_n,_leftover,[_Ll-_tl|_L],_stat) :-
        !,
        sxlistpl(',',_l0,_n0,_leftover0,_s2,_l2,_n2,_leftover2,_Ll,_stat),
        read_word(_leftover2,_l2,_n2,_s3,_l3,_n3,_leftover3,_stat),
        identif(_s3,_l3,_n3,_stat),
        read_word(_leftover3,_l3,_n3,_s4,_l4,_n4,_leftover4,_stat),
        sxparamlistsub(_s3,_s4,_l4,_n4,_leftover4,_s,_l,_n,_leftover,_tl,_L,
        _stat).
sxparamlist1(')',_l,_n,_leftover,')',_l,_n,_leftover,_stat) :- !.
sxparamlist1(_,_l,_n,_,_,_,_,_,[_l,_n,'''';'' or '')'' expected']).

sxparamlistsub(_s0,_s1,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_t,_T,_stat) :-
        var(_stat),!,
        sxparamlistsub1(_s0,_s1,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_t,_T,
        _stat).
sxparamlistsub(_s0,_s1,_l0,_n0,_leftover0,_s,_l,_n,_leftover_t,_T,_stat).

sxparamlistsub1(_s0,'.',_l0,_n0,_leftover0,_s,_l,_n,_leftover,
        t(v(_s0,_l0,_n0),v(_s1,_l1,_n1)),_T,_stat) :- !,
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        identif(_s1,_l1,_n1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxparamlist(_s2,_l2,_n2,_leftover2,_s,_l,_n,_leftover,_T,_stat).
sxparamlistsub1(_s0,';',_l0,_n0,_leftover0,_s,_l,_n,_leftover,
        v(_s0,_l0,_n0),_T,_stat) :-
        !,
        sxparamlist(';',_l0,_n0,_leftover0,_s,_l,_n,_leftover,_T,_stat).
sxparamlistsub1(_s,')',_l0,_n0,_leftover0,';',_l,_n,_leftover,
        v(_s,_l0,_n0),[],_stat) :-!,
```

```
        read_word(_leftover0,_l0,_n0,';',_l,_n,_leftover,_stat).
sxparamlistsub1(_,_,_l,_n,_,_,_,_,_,_,_,_,
        [_l,_n,'''';'',''.'' or '')''expected']).

sxlistp1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_stat):-
        var(_stat),!,
        sxlistp(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_stat).
sxlistp1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_stat) .

sxlistp(':',_l,_n,_leftover,':',_l,_n,_leftover,[],_stat):- !.
sxlistp(',',_l0,_n0,_leftover0,_s,_l,_n,_leftover,[v(_s1,_l0,_n0)|_T],_stat):-
        !,
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        identif(_s1,_l1,_n1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxlistp1(_s2,_l2,_n2,_leftover2,_s,_l,_n,_leftover,_T,_stat).
sxlistp(_,_l,_n,_,_,_,_,_,_,[_l,_n,''',''or '':'' expected']).

sxoperdescrip(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        var(_stat),!,
        sxoperdescrip1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).
sxoperdescrip(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).

sxoperdescrip1(description,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :- !,
        read_word(_leftover0,_l0,_n0,'{',_l1,_n1,_leftover1,_stat),
        sxcomment('{',_l1,_n1,_leftover1,_s,_l,_n,_leftover,_stat).
sxoperdescrip1(_,_l,_n,_,_,_,_,_,['''description'' expected']).

sxoperimpl(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lad,_stat) :-
        var(_stat),!,
        sxoperimpl1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lad,_stat).
sxoperimpl(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lad,_stat).

sxoperimpl1(for,_l0,_n0,_leftover0,_s,_l,_n,_leftover,[_Lad1-_p1|_Lad],_stat):-
        !,
        sxabstdir(',',_l0,_n0,_leftover0,_s2,_l2,_n2,_leftover2,_Lad1,_stat),
        sxoperimplsub(_s2,_l2,_n2,_leftover2,_s3,_l3,_n3,_leftover3,_p1,_stat),
        read_word(_leftover3,_l3,_n3,_s4,_l4,_n4,_leftover4,_stat),
        sxoperimpl(_s4,_l4,_n4,_leftover4,_s,_l,_n,_leftover,_Lad,_stat).
sxoperimpl1(operation,_l,_n,_leftover,operation,_l,_n,_leftover,[],_stat):-!.
sxoperimpl1(Abstract,_l,_n,_leftover,Abstract,_l,_n,_leftover,[],_stat):-!.
sxoperimpl1(BIM,_l,_n,_leftover,BIM,_l,_n,_leftover,[],_stat):-!.
sxoperimpl1(end,_l,_n,_leftover,end,_l,_n,_leftover,[],_stat):-!.
sxoperimpl1(_,_l,_n,_,_,_,_,_,_,
    [_l,_n,'''for'',''operation'',''Abstract'',''BIM'' or ''end'' expected']).

sxabstdir(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lad,_stat) :-
        var(_stat),!,
        sxabstdir2(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lad,_stat).
sxabstdir(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lad,_stat).

sxabstdir2(',',_l0,_n0,_leftover0,_s,_l,_n,_leftover,[_ad|_Lad],_stat) :- !,
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        sxabstdir1(_s1,_l1,_n1,_leftover1,_s2,_l2,_n2,_leftover2,_ad,_stat),
        sxabstdir(_s2,_l2,_n2,_leftover2,_s,_l,_n,_leftover,_Lad,_stat).
sxabstdir2(implementation,_l,_n,_leftover,implementation,_l,_n,_leftover,[],
sxabstdir2(']',_l,_n,_leftover,'}',_l,_n,_leftover,[],_stat) :- !.
sxabstdir2(_,_l,_n,_,_,_,_,_,_,
        [_l,_n,''',''  or ''implementation'' expected']).
sxabstdir1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lad,_stat) :-
        var(_stat),!,
        sxabstdir11(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lad,_stat).
sxabstdir1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lad,_stat).

sxabstdir11('[',_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lad,_stat) :- !,
        sxabstdir(',',_l0,_n0,_leftover0,']',_l1,_n1,_leftover1,_Lad,_stat),
        read_word(_leftover1,_l1,_n1,_s,_l,_n,_leftover,_stat).
sxabstdir11(in,_l0,_n0,_leftover0,_s,_l,_n,_leftover,in,_stat) :- !,
        read_word(_leftover0,_l0,_n0,_s,_l,_n,_leftover,_stat).
sxabstdir11(out,_l0,_n0,_leftover0,_s,_l,_n,_leftover,out,_stat) :- !,
        read_word(_leftover0,_l0,_n0,_s,_l,_n,_leftover,_stat).
sxabstdir11(inout,_l0,_n0,_leftover0,_s,_l,_n,_leftover,inout,_stat) :- !,
        read_word(_leftover0,_l0,_n0,_s,_l,_n,_leftover,_stat).
sxabstdir11(']',_l,_n,_leftover,']',_l,_n,_leftover,[],_stat) :- !.
sxabstdir11(_,_l,_n,_,_,_,_,_,_,
        [_l,_n,'''in'',''out'',''inout''or''implementation''expected']).

sxcomment(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        var(_stat),!,
        sxcomment1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) .
sxcomment(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).

sxcomment1('{',_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        sxanychar(_s1,_l1,_n1,_leftover1,_s,_l,_n,_leftover,_stat).

sxanychar(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        var(_stat),!,
        sxanychar1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).
sxanychar(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).

sxanychar1('{',_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :- !,
        sxcomment('{',_l0,_n0,_leftover0,_s1,_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxanychar(_s2,_l2,_n2,_leftover2,_s,_l,_n,_leftover,_stat).
sxanychar1('}',_l,_n,_leftover,'}',_l,_n,_leftover,_stat) :- !.
sxanychar1('~',_l,_n,_,_,_,_,_,[_l,_n,'unexpected end of file ']) :- !.
sxanychar1(_anychar,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        sxanychar(_s1,_l1,_n1,_leftover1,_s,_l,_n,_leftover,_stat).

sxoperimplsub(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_name,_stat):-
        var(_stat),!,
        sxoperimplsub2(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_name,_stat).
sxoperimplsub(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_name,_stat).

sxoperimplsub2(implementation,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_opername,
        _stat) :- !,
        read_word(_leftover0,_l0,_n0,Is,_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        read_word(_leftover2,_l2,_n2,_s3,_l3,_n3,_leftover3,_stat),
        sxoperimplsub1(_s2,_s3,_l3,_n3,_leftover3,_s,_l,_n,_leftover,
```

```
        _opername,_stat).
sxoperimplsub2(_,_l,_n,_,_,_,_,_,_,[_l,_n,'''implementation'' expected']).

sxoperimplsub1(_s0,_s1,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_t,_stat) :-
        var(_stat),!,
        sxoperimplsub11(_s0,_s1,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_t,_stat).
sxoperimplsub1(_s0,_s1,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_t,_stat).

sxoperimplsub11(_s0,'.',_l0,_n0,_leftover0,';',_l,_n,_leftover,
        n(v(_s0,_l0,_n0),v(_s1,_l1,_n1)),_stat) :- !,
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        identif(_s1,_l1,_n1,_stat),
        read_word(_leftover1,_l1,_n1,';',_l,_n,_leftover,_stat).
sxoperimplsub11(_s0,';',_l,_n,_leftover,';',_l,_n,_leftover,
        v(_s0,_l,_n),_stat) :- !.
sxoperimplsub11(_,_,_l,_n,_,_,_,_,_,_,[_l,_n,'''.'' or '';'' expected']).

sxmodule(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,_stat) :-
        var(_stat),!,
        sxmodule1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,
        _stat).
sxmodule(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,_stat).

sxmodule1(Abstract,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,
        _stat) :- !,
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        sxmodulesub(_s1,_l1,_n1,_leftover1,_s,_l,_n,_leftover,_stat).
sxmodule1(BIM,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,_stat) :-
         !,
        read_word(_leftover0,_l0,_n0,prolog,_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,module,_l2,_n2,_leftover2,_stat),
        read_word(_leftover2,_l2,_n2,_s3,_l3,_n3,_leftover3,_stat),
        sxmodulesub3(_s3,_l3,_n3,_leftover3,_s,_l,_n,_leftover,_names,
        _filename,_stat).
sxmodule1(_,_l,_n,_,_,_,_,_,_,_,[_l,_n,'''Abstract'' or ''BIM'' expected']).

sxmodulesub(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        var(_stat),!,
        sxmod(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).
sxmodulesub(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).

sxmod(description,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :- !,
        read_word(_leftover0,_l0,_n0,module,_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxmodulesub1(_s2,_l2,_n2,_leftover2,_s,_l,_n,_leftover,_stat).
sxmod(prolog,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :- !,
        read_word(_leftover0,_l0,_n0,module,_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxmodulesub2(_s2,_l2,_n2,_leftover2,_s,_l,_n,_leftover,_stat).

sxmodulesub1(_s0,_l0,_n0,_left0,_s,_l,_n,_leftover,_Lparam,_stat) :-
        var(_stat),!,
        sxmodulesub11(_s0,_l0,_n0,_left0,_s,_l,_n,_leftover,_Lparam,_stat).
sxmodulesub1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_Lparam_stat).

sxmodulesub11(';',_l0,_n0,_leftover0,';',_l,_n,_leftover,_Lparam,_stat) :- !,
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxusage(_s2,_l2,_n2,_leftover2,_s3,_l3,_n3,_leftover3,_Lparam,_stat),
        sxADprogram(_s3,_l3,_n3,_leftover3,_s4,_l4,_n4,_leftover4,_stat),
        read_word(_leftover4,_l4,_n4,end,_s5,_l5,_n5,_leftover5,_stat),
        read_word(_leftover5,_l5,_n5,';',_l,_n,_leftover,_stat).
sxmodulesub11(_s0,_l0,_n0,_leftover0,';',_l,_n,_leftover,_stat) :-
        identif(_s0,_l0,_n0,_stat),
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        sxmodulesub1(_s1,_l1,_n1,_leftover1,';',_l,_n,_leftover,_stat).

sxmodulesub2(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        var(_stat),!,
        sxmodulesub21(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).
sxmodulesub2(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).

sxmodulesub21(';',_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :- !,
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxusage(_s2,_l2,_n2,_leftover2,_s3,_l3,_n3,_leftover3,_stat),
        sxAPprogram(_s3,_l3,_n3,_leftover3,_s4,_l4,_n4,_leftover4,_stat),
        read_word(_leftover4,_l4,_n4,end,_s5,_l5,_n5,_leftover5,_stat),
        read_word(_leftover5,_l5,_n5,';',_l,_n,_leftover,_stat).
sxmodulesub21(_s0,_l0,_n0,_leftover0,';',_l,_n,_leftover,_stat) :-
        identif(_s0,_l0,_n0,_stat),
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        sxmodulesub2(_s1,_l1,_n1,_leftover1,';',_l,_n,_leftover,_stat).

sxmodulesub3(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,
        _stat) :-
        var(_stat),!,
        sxmodulesub31(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,
        _filename,_stat).
sxmodulesub3(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,_stat).

sxmodulesub31(';',_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,
        _stat) :- !,
        read_word(_leftover0,_l0,_n0,'{',_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        sxprolog(_s2,_l2,_n2,_leftover2,_s4,_l4,_n4,_leftover4,_names,
        _filename,_stat),
        read_word(_leftover4,_l4,_n4,end,_l5,_n5,_leftover5,_stat),
        read_word(_leftover5,_l5,_n5,';',_l,_n,_leftover,_stat).
sxmodulesub31(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,
        _stat) :- !,
        identif(_s0,_l0,_n0,_stat),
        read_word(_leftover0,_l0,_n0,';',_l1,_n1,_leftover1,_stat),
        sxmodulesub1(';',_l1,_n1,_leftover1,';',_l,_n,_leftover,_names,
        _filename,_stat).

sxusage(_s0,_s1,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_stat):-
        var(_stat),
        sxusage1(_s0,_s1,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_stat).
sxusage(_s0,_s1,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_L,_stat).

sxusage1(representation,dependent,_l0,_n0,_leftover0,_s,_l,_n,_leftover,[],
```

```
        _stat):- !.
sxusage1(representation,independent,_l0,_n0,_leftover0,_s,_l,_n,_leftover,[],
        _stat):- !.
sxusage1(uses,_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,[_s0|_T],_stat):- !,
        identif(_s0,_l0,_n0,_stat),
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        sxlistnames(',',_l1,_n1,_leftover1,_s,_l,_n,_leftover,_T,_stat).
sxusage1(_,_,_l,_n,_,_,_,_,_,_,
        [_l,_n,'unexpected character check usage part']).

sxprolog(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,_stat) :-
        var(_stat),!,
        sxprolog1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,
        _filename,_stat).
sxprolog(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,_stat).
sxprolog1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_filename,_stat) :-
        atomtolist(_filename,_latoms),
        append([''''|_atomnoquotes],[''''],_latoms),
        atomtolist(_name,_atomnoquotes),
        tell(_name),
        sxtell(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_names,_stat),
        told,
        reconsult(_name).

sxADprogram(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        var(_stat),!,
        sxADprogram1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) .
sxADprogram(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) .

sxADprogram1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        tell('ADprogram.pro'),
        sxtell(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat),
        told,
        sxdatatypedef(ADprogram,_stat).

sxADprogram1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        tell('APprogram.pro'),
        sxtell(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat),
        told,
        sxpvsem(ADprogram,_stat).

sxtell(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,adthead(v(_adtname,_,_),_),_sta
        write(':- module('),
        write(_adtname),
        write(').'),
        nl,
        sxtellsub(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).

sxtellsub('~',_l,_n,_,_,_,_,_,[_l,_n,'unexpected end of file ']) :- !.
sxtellsub('}',_l,_n,_leftover,'}',_l,_n,_leftover,_stat) :- !.
sxtellsub(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        write(_s0),
        read_word(_leftover0,_l0,_n0,_s1,_l1,_n1,_leftover1,_stat),
        sxtellsub(_s1,_l1,_n1,_leftover1,_s,_l,_n,_leftover,_stat).

sxtail(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat) :-
        var(_stat),!,
        sxtail1(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).
sxtail(_s0,_l0,_n0,_leftover0,_s,_l,_n,_leftover,_stat).
sxtail1(end,_l0,_n0,_leftover0,'.',_l,_n,_leftover,_stat) :-
        read_word(_leftover0,_l0,_n0,of,_l1,_n1,_leftover1,_stat),
        read_word(_leftover1,_l1,_n1,_s2,_l2,_n2,_leftover2,_stat),
        identif(_s2,_l2,_n2,_stat),
        read_word(_leftover2,_l2,_n2,'.',_l,_n,_leftover,_stat).
```

```
adtsem(_str,_info,_stat):-
        var(_stat),!,
        adtseml(_str,_info,_stat).
adtsem(_str,_info,_stat).

adtseml(adtdef(adthead(v(_adtname,_l,_n),_Lusedadt),_Ldomoperdecl),
        A(_adtname,_Lusedadtl,_Lopers),_stat):-
        simple(_Lusedadt,_Lusedadtl),
        adtexist(_Lusedadt,_stat),
        adtnotexist(v(_adtname,_l,_n),_stat),
        adtsemsub(_Ldomoperdecl,_Lopers,_stat).

simple([],[]) :- !.
simple([v(_name,_,_)|_L],[_name|_T]) :-
        simple(_L,_T).

adtsemsub(_Ldomoperdecl,_Lopers,_stat) :-
        var(_stat),!,
        adtsemsubl(_Ldomoperdecl,_Lopers,_stat).
adtsemsub(_Ldomoperdecl,_Lopers,_stat).

adtsemsubl([],[],_stat) :- !.
adtsemsubl([domoperdecl(_Ldomdecl,_Loperdecl)|_Ldomoperdecl],[_Loper|_Lopers],
        _stat):-
        domdecl(_Ldomdecl,_stat),
        operdecl(_Loperdecl,_Loper,_stat),
        adtsemsub(_Ldomoperdecl,_Lopers,_stat).

domdecl(_Ldomdecl,_stat) :-
        var(_stat),!,
        domdecll(_Ldomdecl,_stat).
domdecl(_Ldomdecl,_stat).

domdecll([],_stat) :-!.
domdecll([domdecl(_Luseddom,_impltype)|_Ldomdecl],_stat) :-
        _impltype = ground,
        !,
        domground(_Luseddom,_stat),
        domdecl(_Ldomdecl,_stat).
domdecll([domdecl(_Luseddom,_impltype)|_Ldomdecl],_stat) :-
        adtexist(_Luseddom,_stat),
        domdecl(_Ldomdecl,_stat).

operdecl(_Loperdecl,_Loper,_stat) :-
        var(_stat),!,
        operdecll(_Loperdecl,_Loper,_stat) .
operdecl(_Loperdecl,_Loper,_stat) .

operdecll(_Loperdecl,_Loper,_stat) :-
        operdecl(_Loperdecl,_Loper,[],_stat).

operdecl(_Loperdecl,_L,_Lopers,_stat) :-
        var(_stat),!,
        operdecll(_Loperdecl,_L,_Lopers,_stat).
operdecl(_Loperdecl,_L,_Lopers,_stat).

operdecll([],[],_,_stat) :- !.
operdecll([operdecl(v(_opername,_l,_n),_Lparam,_filename,_Lprim)|_Loperdecl],
        [_opername,_arity,_Ltypes,_filename,_Ldir|_L],_Lopers,_stat) :-
        arity(_Lparam,_arity),
        _opername1 =.. [_opername,_arity],
        duplicatename(_Lopers,v(_opername1,_l,_n),_stat),
        typecheck(_Lparam,_Ltypes,_stat),
        primcheck(_arity,_Lprim,_Ldir,_stat),
        operdecl(_Loperdecl,_L,[v(_opername1,_l,_n)|_Lopers],_stat).

arity([],0) :- !.
arity([_L-_t|_T],_arity) :-
        size(_L,_s),
        arity(_T,_arity1),
        _arity is _arity1 + _s.

duplicatename(_Lopers,v(_opername,_l,_n),_stat) :-
        occurs(_opername,_Lopers),
        !,
        _stat = [_l,_n,'duplicate operation name '].
duplicatename(_,_,_).

typecheck(_L,_Ltypes,_stat) :-
        var(_stat),!,
        typecheck1(_L,_Ltypes,_stat).
typecheck(_L,_Ltypes,_stat).

typecheck1([],[],_stat) :-!.
typecheck1([_L1-v(_type1,_l,_n)|_L2],[_type1|_Ltypes],_stat) :-
        typeexist(v(_type1,_l,_n),_stat),
        typecheck(_L2,_Ltypes,_stat).

primcheck(_arity,_T,_Ldir,_stat) :-
        var(_stat),!,
        primcheck1(_arity,_T,_Ldir,_stat).
primcheck(_arity,_T,_Ldir,_stat).

primcheck1(_,[],[],_stat) :-!.
primcheck1(_arity,[_dir-v(_p,_l,_n)|_T],[_dir-_name|_Ldir],_stat) :-
        size(_dir,_arity),
        !,
        newopername(_name),
        primcheck(_arity,_T,_Ldir,_stat).
primcheck1(_arity,[_dir-v(_p,_l,_n)|_],_,[_l,_n,'check arity for ',_p]).

domground(_L,_stat) :-
        var(_stat),!,
        domground1(_L,_stat).
domground(_L,_stat).

domground1([],_stat) :-!.
domground1([v(integer,_l,_n)|_L],_stat) :- !,
        domground(_L,_stat).
domground1([v(integer,_l,_n)|_L],[_l,_n,'non ground domain']).

typeexist(_t,_stat) :-
```

```
        var(_stat),!,
        typeexist1(_t,_stat).
typeexist(_t,_stat).

typeexist1(t(_s0,_s1),_stat) :-!,
        adtexist([_s0],_stat).
typeexist1(_s,_stat) :-
         adtexist([_s],_stat).

adtexist(_L,_stat) :-
        var(_stat),!,
        adtexist1(_L,_stat).
adtexist(_L,_stat).

adtexist1([],_stat) :- !.
adtexist1([v(_adtypename,_l,_n)|_L],_stat) :-
        _functor =.. [adtype,_adtypename,_,_,_,_,_,_,_],
        call(_functor),
        !,
        adtexist(_L,_stat).
adtexist1([v(_adtypename,_l,_n)|_L],
        [_l,_n,_adtypename,'unknown abstract data type']) .

adtnotexist(_adtname,_stat) :-
        var(_stat),!,
        adtnotexist1(_adtname,_stat).
adtnotexist(_adtname,_stat).

adtnotexist1(_adtname,_stat1) :-
        adtexist([_adtname],_stat),
        nonvar(_stat),!.
adtnotexist1(v(_adtname,_l,_n),
        [_l,_n,'the abstract data type ',_adtname,'exist already']).

assertinfo(A(_adtname,_Lused,_L),_stat) :-
        var(_stat),!,
        assertinfosub1(_adtname,_Lused,_L).
assertinfo(_,_).

assertinfosub1(_adtname,_Lused,[]) :- !.
assertinfosub1(_adtname,_Lused,[_Linfooper|_L]) :-
        assertinfosub(_adtname,_Lused,_Linfooper),
        assertinfosub1(_adtname,_Lused,_L).

assertinfosub(_adtname,_Lused,[]) :- !.
assertinfosub(_adtname,_Lused,
        [_opername,_arity,_Ltypes,_filename,_Ldir|_L]) :-
        prepare(_adtname,_Lused,_opername,_arity,_Ltypes,_filename,_Ldir,
        _Listoffacts),
        dbwritel(_Listoffacts),
        assertinfosub(_adtname,_Lused,_L).

prepare(_,_,_,_,_,_,[],[]) :-!.
prepare(_adtname,_Lused,_opername,_arity,_Ltypes,_filename,[_dir-_p|_L],
        [_fact|_facts]):-
        _fact =..[adtype,_adtname,_Lused,_opername,_arity,_Ltypes,_filename,
        _dir,_p],
        prepare(_adtname,_Lused,_opername,_arity,_Ltypes,_filename,_L,_facts).

dbwritel([]) :- !.
dbwritel([_fact|_facts]):-
        dbwrite(_fact,'adtbase.pro'),
        dbwritel(_facts).

newopername(_name) :-
        internalopername(_nbre),
        inttoatom(_nbre,_longAtom),
        atomtolist(_longAtom,_atom),
        atomtolist(_name,[p|_atom]),
        _newnbre is _nbre + 1 ,
        dbmodify(internalopername(_nbre),internalopername(_newnbre),
        'adtbase.pro').
newfilename(_filename) :-
        internalfilename(_nbre),
        inttoatom(_nbre,_longAtom),
        atomtolist(_longAtom,_atom),
        append([''',f|_atom],['.',p,r,o,'''],_Latom),
        atomtolist(_filename,_Latom),
        _newnbre is _nbre + 1 ,
        dbmodify(internalfilename(_nbre),internalfilename(_newnbre),
        'adtbase.pro').
```

```
writeprogram(_filename,_structure) :-
        collectLclauses(_structure,_Lclause),
        fopen(outputfile,_filename,w),
        writeclause(outputfile,_Lclauses),
        fcloses),
        fclose(outputfile).

writeLclauese(_outputfile,[]) :_!.
writeLclauses(_outputfile,[_head-body|_Lclauses]) :-
        writeclause(_outputfile,_head,_body,),
        writeLclauese(_outputfile,_Lclauses).

writeclause(_outputfile,_head,[]) :-
        !,
        nl(_outputfile),
        write(_outputfile,_head,),
        write(_outputfile,'.').
writeclause(_outputfile,_head,[_c|_body]) :-
        writehead(_outputfile,_head,_c),
        writeclausesub(_outputfile,_body).

writehead(_head,_c) :-
        nl(_outputfile),
        writehead(_outputfile,_head),
        write(_outputfile,':-'),
        nl(_outputfile),
        tab(_outputfile),
        write(_outputfile,_c).

wrteclausesub(_outputfile,[]) :-!,
        write(_outputfile,'.').
writeclausesub(_outputfile,[_c|_body]) :-
        write(_outputfile,','),
        nl(_outputfile),
        tab(_outputfile),
        write(_outputfile,_c),
        writeclausesub(_outputfile,_body).
collectLclauses(nil,[],[]) :-!.
collectLclauses(lcomp(_comp,_lcomp),_Limport,_Lclauses) :-
        collectsub(_comp,_Limport1,_Lclauses1),
        collectLclauses(_lcomp,_Limport2,_Lclauses2),
        append(_Limport1,_Limport2,_Limport),
        append(_Lclauses1,_Lclauses2,_Lclauses).

collectsub(decrel(_nrel,_Lparam,_ldisj),[],_L) :-
        !,
        collectsubrel(_nrel,_Lparam,_ldisj,_L).

collectsub(decprim(_nprim,_Lparam),_L,[]) :-
        collectsubprim(_nprim,_Lparam,_L).

collectsubrel(nrel([_,_,_relname]),_Lparam,_ldisj,_Lclauses) :-
        collectsubrell(_relname,_Lparam,_ldisj,_Lclauses).

collectsubrell(_,_,nil,nil) :-!.
collectsubrell(_relname,_Lparam,ldisj(conjpref(_,_lconj),_ldisj),
        [_head-_Body|_Lclauses]) :-
        collectoneclause(_relname,_Lparam,_lconj,_head,_Body),
        collectsubrell(_relname,_Lparam,_ldisj,_Lclauses).

collectoneclause(_relname,_Lparam,_lconj,_head,_Body) :-
        head(_relname,_Lparam,_head),
        body(_lconj,_Body).

head(_relname,_Lparam,_head) :-
        listofarg1(_Lparam,_arg),
        _head =.. [_relname|_arg].

body(nil,[]) :-!.
body(conj(_call,_lconj),[_literal|_Literals]) :-
        setliteral(_call,_literal),
        body(_lconj,_Literals).

setliteral(appel([_,_,_functor],_Lp),_literal) :-
        !,
        listofarg2(_Lp,_arg),
        _literal =.. [_functor|_arg].
setliteral(neg(_call),_literal) :-
        setliteral(_call,_liter),
        _literal =.. [Not,_liter].

collectsubprim(nprim([_,_,_nprim]),_Lparam,[_nprim-_arity-_paramtype|_L]) :-
        listoftypes(_Lparam,_paramtype),
        size(_paramtype,_arity).

listoftypes([],[]) :- !.
listoftypes([_-_t|_Lparam],[_t|_paramtype]) :-
        listoftypes(_Lparam,_paramtype).

listofarg1([],[]) :-!.
listofarg1([v(_,_,_arg)-_|_Lparam],[_arg|_Larg]) :-
        listofarg1(_Lparam,_Larg).

listofarg2([],[]) :- !.
listofarg2([v(_,_,_arg)|_Lp],[_arg|_Larg]) :-
        listofarg2(_Lp,_Larg).
```

```
{dbdelete(_fact,_filename) <=>  Si _fact est un fait existant dans la base de
                                types abstraits adtbase.pro alors ce fait est
                                enleve de la base et le predicat reussit;
                                Sinon le predicat echoue.}


dbdelete(_fact,_filename) :-
        retract(_fact),!,
        dbdeletesub(_filename).

dbdeletesub(_filename) :-
        fopen(outputfile1,_filename,w),
        write(outputfile1,(:-alldynamic)),
        write(outputfile1,'.'),
        nl(outputfile1),
        dbdeletesub,
        fclose(outputfile1).
dbdeletesub :-
        internalfilename(_nbre),
        writeq(outputfile1,internalfilename(_nbre)),
        write(outputfile1,'.'),
        nl(outputfile1),
        fail.
dbdeletesub :-
        internalopername(_nbre),
        writeq(outputfile1,internalopername(_nbre)),
        write(outputfile1,'.'),
        nl(outputfile1),
        fail.
dbdeletesub :-
        adtype(_a,_b,_c,_d,_e,_f,_g,_h),
        writeq(outputfile1,adtype(_a,_b,_c,_d,_e,_f,_g,_h)),
        write(outputfile1,'.'),
        nl(outputfile1),
        fail.
dbdeletesub.

{dbwrite(_fact,_filename) <=> Le fait _fact est ajoute a la base
                              de fait adtbase.pro }
dbwrite(_fact,_filename) :-
        fopen(outputfile,_filename,a),
        write(outputfile,_fact),
        write(outputfile,'.'),
        nl(outputfile),
        fclose(outputfile),
        reconsult(_filename).
{dbwritesub :-
        dynamic_functor(adtype(_a,_b,_c,_d,_e,_f,_g,_h)),
        adtype(_a,_b,_c,_d,_e,_f,_g,_h),
        write(outputfile,adtype(_a,_b,_c,_d,_e,_f,_g,_h)),
        write(outputfile,'.'),
        nl(outputfile),
        fail.
dbwritesub.}
dbmodify(_fact,_newfact,_filename) :-
```

```
dbdelete(_fact,_filename),
dbwrite(_newfact,_filename).
```

```
adt_translator(_filename) :-
        nl,
        write('verification syntaxique ... '),nl,nl,
        sxdatatypedef(_filename,_str,_stat1),
        write('verification semantique ... '),nl,nl,
        adtsem(_str,_Linfo,_stat1,_stat2),
        write('mise a jour BD ... '),nl,nl,
        assertinfo(_Linfo,_stat2,_stat3),
        write('gestion des erreurs ... '),nl,
        trt_diagn_err(_filename,[_str],_stat3).

adtsem(_str,_Linfo,_stat1,_stat2) :-
        var(_stat1),!,
        adtsem(_str,_Linfo,_stat2).
adtsem(_str,_Linfo,_stat1,_stat1).

assertinfo(_Linfo,_stat1,_stat2) :-
        var(_stat1),!,
        assertinfo(_Linfo,_stat2).
assertinfo(_Linfo,_stat1,_stat1).
assertinfo(A(_adtname,_Lused,_L),_stat) :-
        var(_stat),!,
        assertinfosub1(_adtname,_Lused,_L).
assertinfo(_,_).

assertinfosub1(_adtname,_Lused,[]) :- !.
assertinfosub1(_adtname,_Lused,[_Linfooper|_L]) :-
        assertinfosub(_adtname,_Lused,_Linfooper),
        assertinfosub1(_adtname,_Lused,_L).

assertinfosub(_adtname,_Lused,[]) :- !.
assertinfosub(_adtname,_Lused,
        [_opername,_arity,_Ltypes,_filename,_Ldir|_L]) :-
        prepare(_adtname,_Lused,_opername,_arity,_Ltypes,_filename,_Ldir,
        _Listoffacts),
        dbwritel(_Listoffacts),
        assertinfosub(_adtname,_Lused,_L).

prepare(_,_,_,_,_,_,[],[]) :-!.
prepare(_adtname,_Lused,_opername,_arity,_Ltypes,_filename,[_dir-_p|_L],
        [_fact|_facts]):-
        _fact =..[adtype,_adtname,_Lused,_opername,_arity,_Ltypes,_filename,
        _dir,_p],
        prepare(_adtname,_Lused,_opername,_arity,_Ltypes,_filename,_L,_facts).

dbwritel([]) :- !.
dbwritel([_fact|_facts]):-
        dbwrite(_fact,'adtbase.pro'),
        dbwritel(_facts).

newopername(_name) :-
        internalopername(_nbre),
        inttoatom(_nbre,_longAtom),
        atomtolist(_longAtom,_atom),
        atomtolist(_name,[p|_atom]),
        _newnbre is _nbre + 1 ,
        dbmodify(internalopername(_nbre),internalopername(_newnbre),
        'adtbase.pro').
newfilename(_filename) :-
        internalfilename(_nbre),
        inttoatom(_nbre,_longAtom),
        atomtolist(_longAtom,_atom),
        append([''',f|_atom],['.',p,r,o,'''],_Latom),
        atomtolist(_filename,_Latom),
        _newnbre is _nbre + 1 ,
        dbmodify(internalfilename(_nbre),internalfilename(_newnbre),
        'adtbase.pro').
```