MASTER EN SCIENCES INFORMATIQUES

Contributions à un système de dialogue homme-machine à composante orale

Lorant, Manu; Simonet, Serge

Award date:
1986

Awarding institution:
Universite de Namur

Facultés Universitaire Notre-Dame de la Paix

Institut d'informatique

Rue Grandgagnage, 21

B - 5000 Namur.

CONTRIBUTIONS A UN SYSTEME DE DIALOGUE HOMME-MACHINE A COMPOSANTE ORALE

Mémoire présenté par

Manu Lorant

et

Serge Simonet

en vue de l'obtention

du titre de

Licencié et Maitre en Informatique

Année académique 1985-1986

REMERCIEMENTS

Au terme de cette longue année de travail, nous tenons à présenter nos remerciements et à manifester toute notre sympathie à tous ceux qui nous ont aidés et soutenus.

Plus particulièrement, nous voudrions remercier Jean-Marie Pierrel, Pierre Mousel et Azim Roussanally pour l'accueil qu'ils ont bien voulu nous réserver, ainsi que pour les conseils et les encouragements qu'ils nous ont prodigués durant tout notre séjour à Nancy.

Grand merci au père Jacques Berleur qui nous a consacré de longs moments de son précieux temps afin de nous assister durant la période de rédaction en corrigeant et en guidant nos travaux.

Enfin, au terme de ces trois années d'études, nous souhaitons remercier l'ensemble du personnel du secrétariat administratif pour son sourire et sa gentillesse qui nous ont accompagné tout au long de ces années.


TAPEZ VOTRE NOM SUR LE CLAVIER
BOOKE.

INTRODUCTION.

Les ordinateurs prennent de plus en plus d'importance dans notre vie quotidienne. Ils envahissent tous les secteurs d'activités. Qu'il soit au bureau, à l'école, ou même à la maison, l'homme y est confronté presqu'en permanence.

Ce phénomène, bien que relativement récent, se développe et se généralise dans des proportions spectaculaires; il est certain qu'il ne cessera de s'amplifier dans les décénnies à venir.

Dans un tel développement, le mode de communication entre l'homme et l'ordinateur revêt une importance considérable.
La parole, élément important de la spécificité humaine, peut apporter une solution naturelle et adéquate à ce problème.
Sans entrer dans le débat éthique qui y est inévitablement lié, force nous est d'admettre que son utilisation peut être intéressante à bien des égards. Par exemple, elle pourrait élargir grandement le cercle restreint de ses utilisateurs en permettant une communication facile et aisée qui ne nécessiterait plus d'apprentissage long et fastidieux et, d'une manière plus générale, le rendrait plus accessible pour ceux qui, pour une raison quelconque, ne pourrait en bénéficier en temps normal.

Ce genre de considérations a donné lieu au développement d'une branche de recherche particulière : <u>la compréhension de la parole par ordinateur</u>.
Bien que, pour beaucoup de chercheurs dans ce domaine, les bases conceptuelles sous-jacentes à la compréhension de la parole ne soient pas entièrement formulables en toute généralité [Weizenbaum 84], des systèmes très limités ont vu le jour en Europe aussi bien qu'aux USA et au Japon.

Pour ceux qui traitaient la parole continue, l'objectif premier était d'étudier la reconnaissance et/ou la compréhension d'énoncés pour un locuteur donné.
Les conclusions auxquelles ils ont abouti montrent clairement que la caractéristique la plus fondamentale de tels traitements est qu'il n'existe pas de solution générale aux problèmes qu'ils posent, et qu'il s'agit en fait de processus non déterministes prenant place dans un environnement contenant inévitablement des erreurs [Haton 85]. Il faut souligner que ceci est valable malgré le fait que les langages utilisés étaient très limités et les domaines d'application fort restreints.

Une des approches les plus utilisées se base sur le fait que

l'être humain peut être considéré comme un système prototype et qu'il peut, à ce titre, nous aider à développer de tels systèmes. Il ne s'agit pas de le recréer structurellement, mais de développer quelque chose qui fonctionnellement s'apparente à lui.
Cette approche relève entièrement du domaine de l'Intelligence Artificielle car elle comprend l'intégration de sources de connaissances variées ainsi que des techniques de recherche de solutions dans un espace composé de diverses interprétations totales ou partielles du message vocal étudié [Haton 85].

Actuellement, les chercheurs s'orientent vers des systèmes de gestion de dialogues oraux à proprement parler.

Ils sont caractérisés par le fait qu'ils traitent des dialogues finalisés, c'est-à-dire orientés par une tâche à accomplir et réalisés avec des locuteurs motivés et coopératifs.

Bien que de tels systèmes soient caractérisés par l'utilisation de sous-langages se développant de manière spontanée et matérialisés par des constructions syntaxiques spécifiques et des restrictions lexicales et sémantiques relativement importantes, leur gestion se heurte à des problèmes tels que leur compréhension proprement dite - qui nécessite des représentations adéquates des discours et des contextes d'énonciation - et l'interprétation des références qu'ils contiennent.

La résolution de ces problèmes spécifiques passe nécessairement par la définition, la modélisation, et la mise en oeuvre de nouvelles sources de connaissances et de nouvelles fonctions par rapport à celles indispensables pour la compréhension des phrases "isolées".

Lors de notre stage à Nancy, nous avons travaillé au sein de l'équipe de recherche de J.M. Pierrel chargée du développement d'un tel système.
Leur objectif est de mettre au point un outil valable pour une classe d'applications de type "centre de renseignements".

De manière excessivement schématique, il est organisé autour d'un module de dialogue assurant toutes les fonctions centrales du système : raisonnement nécessaire à l'identification et à la satisfaction de la requête, gestion du dialogue proprement dit, génération des énoncés de la machine, etc...

Au sein de ce module, nous nous sommes vus attribuer la responsabilité de l'étude et si possible du développement d'un outil de vérification de cohérence et d'un outil d'interprétation contextuelle.

En un mot, le premier devra d'abord offrir une assistance à l'expert dans la formalisation de ses connaissances sous forme de règles de déduction en lui assurant leur cohérence. D'autre part, il prendra en charge certains aspects plus spécifiques de la cohérence dans un système de dialogue : cohérence entre les différentes interventions du locuteur, entre celles-ci et la tache à accomplir, etc...

Le second, quant à lui, devra pouvoir générer une représentation de la signification que prennent les énoncés dans le cadre de l'application et du dialogue en cours, et ce à partir d'une représentation syntaxique et d'une représentation sémantique de ces énoncés, cette dernière correspondant déjà à une première interprétation par rapport au contexte local des énoncés.

Ces deux outils correspondent aux deux réalisations personnelles qui sont les bases de ce mémoire.

Cet ouvrage est scindé en 2 parties.

La première constitue une approche générale du problème de la communication orale homme-machine.
Le premier chapitre positionne le problème et rappelle brièvement les principes généraux qui président à la compréhension automatique de phrases "isolées".
Le second introduit la problématique spécifique au dialogue oral.

Dans la seconde partie, nous présentons successivement le système de gestion de dialogues oraux finalisés en développement à Nancy (chapitre III), l'interprétation contextuelle (chapitre IV), et quelques aspects de la cohérence (chapitre V).

Les spécifications, les textes des programmes, et l'un ou l'autre exemple d'exécution sont disponibles en annexe.

CHAPITRE I : DIVERS ASPECTS DE LA COMMUNICATION HOMME-MACHINE.

1 La communication orale homme machine.

Avant de s'interroger sur la communication orale entre les hommes et les machines, il nous semble utile de s'interroger tout d'abord sur la communication elle-même. Ceci, afin d'essayer d'en dégager certaines carcatéristiques pour voir dans quelles mesures celles-ci pourraient retenir notre attention dans le cadre spécifique de la communication homme-machine.

1.1 La communication.

Dans un premier temps nous rappelerons ce qu'est pour nous la communication. Ensuite nous essaierons d'établir les rapports entre des termes qui reviendront souvent au cours de notre exposé, à savoir parole, langue et langage. Nous essaierons aussi de voir le lien existant entre ces concepts et le concept de communication. Ensuite nous définirons la notion de dialogue oral, pour enfin déboucher sur un typologie des langages dans lesquels peuvent se réaliser la communication homme-machine.

Qu'est-ce donc que la communication ?

Il est hors de notre propos ici d'entreprendre une étude complète de la définition de la communication, étude qui nécessiterait d'envisager bien des aspects que nous n'aborderons pas car ils sortent du cadre de nos travaux. Nous nous limiterons donc à l'étude de la réalisation de cette communication car elle seule nous intéresse ici.
Dès lors cette étude peut se reposer uniquement sur le modèle proposé par la cybernétique.
Ce modèle, présenté à la figure 1.1, considére qu'il s'agit d'une action par laquelle un message, ayant une source et une destination, est transmis par un émetteur, porté par un canal de transmission et, enfin recu par un recepteur.
Les émetteurs, ainsi que les récepteurs peuvent être humains, animaux, végétaux et mêmes mécaniques ou électroniques, dans le cas des machines. Selon la nature respective des êtres impliqués dans une communication ainsi que selon l'objectif poursuivi par celle-ci, le canal de communication sera différent.

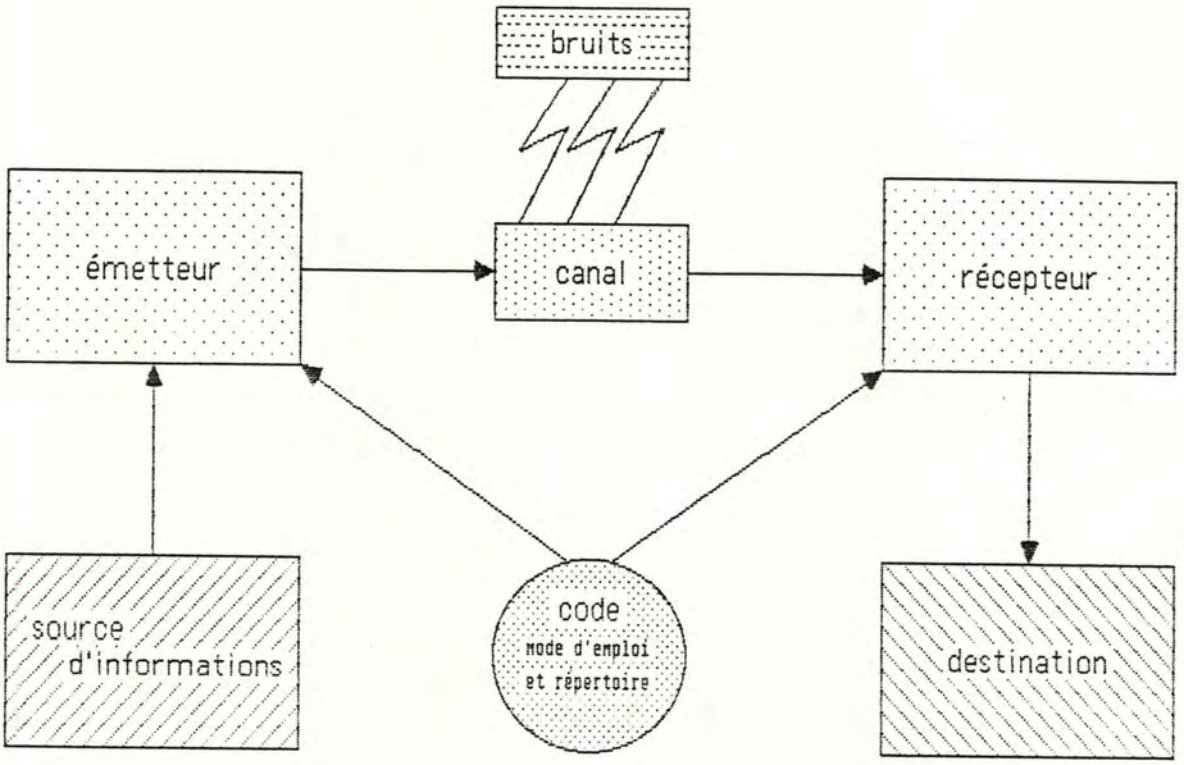


Figure 1.1: Système de communication

L'expérience personnelle montre à loisir que le mode de communication varie selon la nature des émetteurs et des récepteurs mis en présence.

- Le policier réglant la circulation communiquera ses adjonctions aux automobilistes par des gestes précis.

- Le journaliste communique avec ses lecteurs par voie de presse

- La machine qui doit communiquer avec l'homme le fait au moyen d'un voyant lumineux. On remarquera qu'en cette matière des travaux importants sont réalisés visant à permettre aux machines de communiquer avec l'homme de la manière la plus efficace et la plus agréable pour ce dernier : la parole.

Aux composantes évoquées d'un système de communication, il convient d'ajouter deux opérations supplémentaires.
La première, dite de codage, par laquelle l'émetteur rend son message apte à être transmis, et la seconde, inverse de la première, dite de décodage, par laquelle le récepteur reconvertit le message codé en une forme directement 'utilisable' pour lui.
La littérature spécialisée indique que le code est constitué à la fois d'un répertoire d'éléments et de leur mode d'emploi, c'est à dire les règles permettant de réaliser correctement le codage du message.

Les objectifs poursuivis par l'émetteur, ainsi que le contexte dans lequel il évolue l'amèneront à choisir plutôt tel ou tel code pour transmettre son message

- Le caractére secret d'un message aménera l'émetteur à choisir un code dont il sait que personne d'autre que le récepteur ne connait la clé.

- L'homme s'adressant à un animal ne le fera en général pas avec le même ton de voix ni avec les mêmes mots qu'il ne le ferait pour un autre être humain.

- Le professeur voulant intimider son auditoire s'adressera à lui d'une façon hautaine et distante.

De ceci, on devine aisément que pour que la communication entre l'émetteur et le récepteur puisse être efficace, il faut que le premier utilise un code connu et maitrisé totalement ou au moins partiellement par le second. Le non respect de cette condition risque d'entrainer l'échec de la communication.

Parmi l'ensemble des multiples moyens de communication à disposition, il en est un qui dépasse tous les autres dans l'utilisation qu'ils en font car il constitue une spécificité et un déterminant de la réalité humaine : la parole

A ce stade, il nous semble intéressant d'essayer d'en dégager les spécificités.

- La fonction de communication n'en est pas une puisque les langages oraux (que l'on nomme également langues), partagent cette caractéristique avec tous les autres moyens de communication.

- L'utilisation de signes arbitraires plutôt que de symboles n'en est pas une non plus. Il existe en effet de nombreux autres moyens de communication qui utilisent des signes arbitraires. L'exemple du code de la route dans lequel un cercle désigne une autorisation ou une interdiction, un triangle un danger et un rectangle une indication est éloquent à ce sujet. Il faut également ajouter de ce point de vue que la communication parlée se compose également de certains éléments symboliques, notamment pour ce qui est de l'ironie.

- Selon A. Martinet [Martinet 56], ce qui caractérise nos langages humains par rapport à tous les autres systèmes de communication, c'est une double articulation autour de laquelle ceux-ci sont construits. La première articulation est la suivante : nos langages peuvent se décomposer en chaines vocales qui sont intrinsèquement non significatives et que l'on nomme les phonèmes; Ces unités minimales ont une forme mais pas de signifié.
  La seconde articulation est celle selon laquelle nos langages peuvent également se décomposer en un certain nombre d'unités minimales possédant à la fois une forme et un sens. Celles-ci sont applelées les monèmes. C'est grâce au double codage réalisé tout d'abord sur les phonèmes (pour obtenir des monèmes) puis ensuite sur ces monèmes que nous pouvons exprimer l'infinité des situations pouvant survenir devant nous.
  La puissance du langage humain est donc de pouvoir tout exprimer à partir d'une cinquantaine de signes minimaux : les phonèmes, et cela uniquement par combinaison de ceux-ci.

Nous pouvons donc effectuer un retour sur ce que nous disions précédemment à propos du modèle de la communication présenté par la cybernétique et essayer de rapporter cela au cadre très particulier de la parole.
La source d'information et la destination de celle-ci sont les images ou les concepts qu'exprime la personne qui parle et que perçoit celle qui entend ce qui est dit.
L'émetteur et le récepteur sont les deux personnes qui communiquent entre elles.

Quant au codage il est réalisé sur base de cette double articulation. Toujours selon Martinet [Martinet op. cit.], c'est la réalisation de ce codage qui détermine une langue et qui va donc permettre de différencier le français par exemple de toutes les autres langues (plus de 56 familles de langues recensées)

Un dernier point reste à éclaicir, il s'agit du rapport entre langue et langage.
La différence n'a pas toujours été faite de façon claire par les auteurs, ceux-ci allant même parfois jusqu'à confondre les deux. Depuis le début du siècle, ils semblent s'être mis d'accord sur la distinction existant entre ces deux réalités. C'est ainsi que l'on doit à un auteur des années '30, J. Marouzeau [Marouzeau 31], une proposition de définition du terme langage.
Un langage serait tout système de signes permettant de servir de moyen de communication. On parlera du langage animal, du langage des gestes ou encore de celui des couleurs. L'appelation langue est, elle, réservée aux langages verbaux.
Même si nous acceptons cette distinction entre les deux, elle ne nous semble pas essentielle dans le contexte présent puisque nous nous situerons toujours dans le cadre de la parole.

Ayant défini tant la notion de communication que celle de langage, il devient nécessaire pour la suite d'évoquer ce que nous entendrons par dialogue. Le but de ce paragraphe n'est pas de donner une définition rigoureuse de la notion de dialogue, mais plutôt d'en dégager les caractéristiques et les fonctionnalités qui nous semblent importantes pour la suite.

La première fonction, la plus importante, est de permettre un échange d'informations entre deux, ou plusieurs interlocuteurs.

La deuxième pourrait se comparer aux protocoles utilisés dans de multiples résaux de transmission : c'est l'établissement du contact entre les interlocuteurs. Ils détermineront ainsi qui ils ont en face d'eux et ce sur quoi portera l'échange d'information.

La troisième fonction est celle qui va permettre de réaliser le contrôle et la validation d'élémemts du dialogue. Ainsi l'un des interlocuteurs pourra demander à l'autre de répéter une phrase ou une idée qu'il vient d'émettre. Le premier pourra demander au second si l'interprétation qu'il fait de ce que vient de dire ce dernier est bien correcte.

La dernière fonction est celle qui va permettre à l'un des interlocuteurs d' orienter l'échange d'informations dans tel ou tel sens qui lui semble intéressant. On peut dans ce cas considérer que celui qui oriente le dialogue est maitre de l'échange.

Comme le rappelle Pierrel [Pierrel 81], la difficulté réside dans le fait que tout dialogue n'est pas uniquement verbal et que donc les informations communiquées par les interlocuteurs ne le sont pas uniquement par le canal oral.

1.2 Le dialogue dans la communication orale homme machine.

L'importance croissante que prennent les machines dans notre vie n'est pas à démontrer, que ce soit au travail, à l'école ou même au domicile. Une des difficultés apparaissant souvent est le mode de communication entre l'homme demandeur d'un service et la machine fournisseur potentiel de ce service. Je n'en veux pour preuve que la difficulté avec laquelle certaines personnes du troisième âge règlent ces nouveaux petits réveils électroniques afin d'être réveillées à l'heure voulue.

Lorsqu'un enfant souhaite boire, il dit " j'ai soif " à sa mère ou à son père de sorte que ce dernier comprenne qu'il doit lui donner à boire. Lorsque je désire connaitre l'heure, je m'adresse à mon voisin en lui demandant " quelle heure est-il ? ". Dans bien des cas, la parole semble offrir le meilleur moyen de communication possible pour l'expression d'un besoin, d'une idée ou d'une constatation.

Pour la communication entre l'homme et la machine, la parole pourrait dès lors offrir un interface très aisé. Celui-ci permettant à l'homme d'exprimer et de communiquer ses ordres à la machine de la façon qui lui semble la plus naturelle possible.

Les avantages et désavantages que l'on peut mettre en évidence pour l'utilisation d'un tel mode de communication entre l'homme et la machine sont nombreux. Certains sont plus ou moins mis en évidence selon les points de vue qu'ils défendent. Notre but n'est pas ici de donner une liste exhaustive des intérêts ou des incovénients du traitement de la parole par ordinateur. Nous ne voulons pas non plus prendre position de façon définitive dans ce débat éthique pour ou contre la communication orale homme-machine car

cette prise de position constitue un choix d'un autre niveau que celui pouvant résulter d'une simple énumération d'arguments favorables et d'arguments défavorables.

Pour reprendre l'exemple cité plus haut, il serait bien pratique pour une personne âgée de n'avoir pas à se préoccuper d'appuyer simultanément sur telle et telle touche d'un réveil électronique pour en programmer l'alarme. Elle préfèrerait sans doute de loin n'avoir qu'à simplement énoncer à proximité de l'appareil l'heure à laquelle elle souhaite se réveiller pour que le système se programme de lui-même

L'exemple de la micro-chirurgie semble lui aussi assez éloquent. Le chirurgien ayant les deux mains occupées par ses instruments d'opération ne peut effectuer à l'aide de celles-ci les réglages du microscope dont il se sert pour observer la plaie de son patient. Il lui est possible de commander ce microscope à l'aide du pied, mais cet organe est peu apte à ce genre d'exercice de précision. L'utilisation de la commande vocale serait donc un facilitant considérable surtout lorsque les réglages se multiplient.

Les avantages mis en avant par les promoteurs de tels systèmes sont de deux types :

- rendre la plus naturelle possible la communication homme-machine en permettant qu'elle s'opère par le moyen que l'homme a le plus naturellement tendance à utiliser : la parole.

- permettre l'accès aux machines à ceux qui pour une raison quelconque ne pourrait pas y avoir accès (handicaps, opérations multiples, etc ...).

A ces avantages de type 'pratique', on peut opposer un premier argument qui est d'un tout autre niveau. C'est celui défendu par J. Weizenbaum(a) dans [Weizenbaum 84]. Il revient à se demander si la parole et la compréhension de celle-ci, qui comme nous l'avons déjà exprimé maintes fois, est une capacité dont lui seul dispose, et, qui plus est, constitue une de ces caractéristiques, n'est pas un objectif en inadéquation avec les objectifs des machines. Weizenbaum généralise d'ailleurs la première limite que nous avons évoquée en se demandant " si toutes les bases conceptuelles

---

(a) Notons que Weizenbaum se situe ici au niveau de la langue naturelle alors que bien des chercheurs s'intéresse plutôt aux langages dits "opératifs" que nous évoquerons ultérieurement.

sous-jacentes à la compréhension de la parole sont formulables " ainsi que le prétend Schank.
Cet argument ne peut être classé au rang des avantages et des inconvénients car il est, à notre avis, prépondérant sur le reste s'il faut reconnaitre que Weizenbaum a raison.

A côté de cet argument, on peut en ajouter un second dont la portée sera bien moins grande tant par son caractère temporaire que par sa nature.
Il tient aux limites de la reconnaissance et de la compréhension. Ces limites ont pour causes les connaissances incomplètes en matière de linguistique ainsi que les difficultés rencontrées ,qui seront évoquées plus loin, lors des analyses accoustiques et phonétiques des signaux. Il se peut en effet que le locuteur s'adressant à un tel système emploie un vocabulaire, des formes syntaxiques, des tournures sémantiques inconnues de la machine. Ceci aura pour effet d'amener la machine à échouer dans le processus de compréhension du locuteur.
Mais il se peut aussi que l'énoncé se fasse dans un environnement sonore tel qu'il soit accompagné de bruits de fond totalement extérieurs à la voix du locuteur. Ces bruits ne pouvant ni être aisément localisés ni, a fortiori, être éliminés, on imagine sans difficulté l'impossibilité d'une bonne compréhension par la machine.

Dès lors les deux arguments énoncés vont-ils de pair car si l'on arrive à la conclusion que l'ensemble des intervenants dans un processus de compréhension ne peut être formalisé - et donc ne peut être exploité par les machines - il faut en arriver à la conclusion qu'il existe des objectifs propres à l'homme et impropres aux machines. La parole étant un de ces objectifs, il y aurait là une limite intrinsèque au traitement automatique de la parole par ordinateur.

1.3 Typologie des langages de la communication homme-machine

Ayant vu en quoi consistait la communication homme-machine, quels en étaient les objectifs et les limites, il nous faut maintenant voir dans quelle mesure et comment l'homme pourrait s'adresser à une machine. Sachant que l'interaction entre les deux se fera au moyen d'un langage, il s'agit de déterminer quels peuvent être les différents types de langages utilisables dans ce contexte.

La littérature recense jusqu à présent quatre types de langages. Ceux-ci correspondent à des niveaux de connaissance différents et supposent des traitements plus ou

moins complexes selon les cas.

1.3.1 Langages mots isolés.

Le premier type de langage est celui appelé langage par mots isolés. Il consiste à émettre des messages en ne se servant que de mots choisis dans un vocabulaire très restreint. Chacun de ces mots est séparé de celui qui le précède et de celui qui le suit par une pause.

Dans une situation où un guide devrait indiquer à un robot des mouvements à exécuter par ce dernier, l'ordre de déposer un cube tenu par le robot s'énoncerait sans doute de la façon suivante :

"déposer ‹pause› cube"

De tels langages sont évidemment assez pauvres par rapport à la variété des messages qu'il permettent, mais ils se prêtent néanmoins très bien au traitement automatique de la parole. (a)

Ce genre de langage est essentiellement utilisé pour la commande vocale de machines.
Il s'adapte surtout bien à la communication homme-machine dans le sens de l'homme vers la machine; il s'avère peu utilisé dans l'autre sens du fait de l'inconfort d'écoute qu'il provoque et surtout des ambiguïtés qu'il laisse planer sur le contenu des messages émis.

1.3.2 Les langages artificiels.

Le second type de langage franchit le pas de la continuité dans l'élocution. Le locuteur ne se contente plus d'émettre une suite de mots isolés, mais il produit des phrases entières syntaxiquement correctes du point de vue de la langue utilisée. Néanmoins, la syntaxe utilisée pour la formation de telles phrases est très limitée et le locuteur ne peut se permettre aucun écart par rapport à celle-ci. C'est pourquoi le langage obtenu dans de telles conditions est appelé langage artificiel. Il convient bien selon Pierrel [Pierrel 81] pour les commandes de processus.

Plus riches que les langages mots isolés, ces langages présentent toutefois encore l'inconvénient par rapport à l'objectif initial de n'être pas encore très naturel puisque

---

(a) Plus de détails sont donnés au point suivant sur les méthodes de reconnaissance de mots isolés.

tout locuteur se trouve contraint à la fois par un vocabulaire encore restreint et surtout par la syntaxe. Ils nécessitent donc une phase d'apprentissage pour en acquérir la maitrise préalable de la part de celui qui souhaite les employer.

1.3.3 Les langages quasi (pseudo)-naturels.

Le troisième type de langage tente de se rapprocher le plus possible du langage naturel dans lequel nous nous exprimons quotidiennement. A cet effet un minimum de restrictions lui ont été adjointes aux niveaux tant lexical que syntaxique.

Les restrictions portant sur ces langages sont liées à leur domaine d'application. Elles sont de nature à restreindre le nombre d'interprétations possibles d'une phrase énoncée par un locuteur à un champ très spécifique. Cette grande latitude a pour conséquence de permettre d'aborder le grand public avec des systèmes traitant de tels langages puisqu'aucune phase d'apprentissage ne s'impose plus.
D'autre part, elle force les concepteurs des systèmes à devoir faire face à des difficultés de reconnaissance et de compréhension bien plus grandes que dans le cas des langages par mots isolés ou artificiels. Ces difficultés sont, bien entendu, liées aux phénomènes prosodiques, aux phénomènes d'altération, ainsi qu'à la syntaxe et à la latitude laissée quant au vocabulaire autorisé. (a)

Sans s'étendre trop sur la question, on peut d'ores et déjà dire que l'incertitude sur les éléments accoustiques et phonétiques composant les énoncés liée à la mauvaise reconnaissance dans le cas de discours continu sera compensée par l'utilisation d'un grand nombre de connaissances. Ces connaissances porteront, par exemple, sur le domaine d'application évoqué pour la restriction des interprétations.

1.3.4 La langue naturelle.

Le quatrième type de langage est celui déjà évoqué plus haut et sur lequel nous ne nous attarderons pas : la langue naturelle. Il s'agit ni plus ni moins du langage dans lequel nous nous exprimons à tout instant et dans lequel ce texte est écrit. Elle ne souffre d'autres limites que celles de la grammaire et du vocabulaire. Et encore la poésie et la

---

(a) Les méthodes permettant d'aborder et de solutionner des ces nouvelles difficultés seront abordées dans la partie concernant les principes généraux.

science font parfois fi de ces dernières que ce soit par la création de néologismes ou par l'usage de constructions syntaxiques incorrectes pour un grammairien puriste, mais qui sont néanmoins tout à fait compréhensibles.

1.3.5 Conclusion

Sans vouloir trop s'attarder sur la question, on peut déjà se rendre compte que cette classification induit des types d'applications différentes selon le type du langage qu'elles vont utiliser.En effet un langage par mot isolés ou un langage artificiel ne s'adapterait sans doute pas bien à une application grand public, alors qu'il serait sans doute irréaliste de vouloir contrôler un processus de production en utilisant la langue naturelle.

2 Les principes generaux.

2.1 Introduction.

La compréhension du dialogue postule nécessairement la reconnaissance et la compréhension des énoncés qui le composent. Nous nous attarderons donc dans un premier temps à étudier cette dernière.
Pour bien situer les concepts mis en jeu et les relations qui existent entre eux, il est important, nous semble-t-il, d'attirer l'attention sur la spécificité de la compréhension automatique de phrases.
C'est la raison pour laquelle ce chapitre comporte 4 parties. Dans la première, nous tenterons de montrer que sa caractéristique la plus fondamentale est qu'il n'existe pas de solution générale au problème qu'elle pose, et qu'il s'agit en fait d'un processus non déterministe prenant place dans un environnement contenant inévitablement des erreurs.
Dans la seconde, nous présenterons rapidement les 2 types d'approche actuellement utilisées, l'une orientée mathématique, l'autre plus orientée linguistique. Il faut souligner dès à présent que cette classification n'est pas dichotomique [LEA 80], et que la plupart des systèmes sont de composition mixte, même si l'une ou l'autre prédomine dans leur conception.
Dans la troisième partie, nous étudierons quelque peu les solutions principales adoptées en reconnaissance des mots isolés.
Enfin, dans la quatrième partie, nous aborderons le traitement de la parole continue pseudo-naturelle,. Celui-ci constituant un important changement de problématique par rapport aux mots isolés, et les solutions utilisées pour leur reconnaissance n'étant pas généralisables, nous envisagerons un autre type de solution, caractérisée par une approche plutôt linguistique.

Il faut dès à présent souligner qu'en raison de la difficulté des problèmes rencontrés [HATON 85], aucun système général n'est actuellement envisageable (cfr GPS). C'est la raison pour laquelle tous les travaux effectués dans le domaine sont fort limités, tant par le langage utilisé que par les applications visées.

Enfin, si, pour le lecteur averti, ces quelques pages ne font que rappeler des notions bien connues, nous espérons qu'elles constitueront une bonne introduction au domaine étudié pour les autres.

2.2 Spécificité de la compréhension de la parole.

Le but de tout système de traitement de la parole (phrases ou plutôt énoncés) est soit la reconnaissance, soit la compréhension.

Dans le premier cas, l'objectif est d'essayer de reconstituer le message oral, morceaux par morceaux, et sans aucune référence à sa signification [PIERREL 82].

Dans le second, il s'agit de transformer le signal acoustique en représentations discrètes auxquelles on peut assigner un sens déterminé et qui, une fois comprises, permettent de déclencher un certain comportement en réponse. Il n'est pas question ici d'une reconnaissance et d'une compréhension complète, mais plutôt suffisante pour produire une réponse appropriée. C'est surtout au niveau du second type de système que nous analyserons les principes généraux qui sous-tendent leur conception, car c'est également le cadre dans lequel nous avons été amenés à travailler à Nancy.

Quelque soit le système envisagé ou l'approche adoptée, la parole est constituée de 2 phénomènes complémentaires :

- un phénomène acoustique ;
- et un phénomène linguistique.

Ils font l'objet des 2 sections suivantes.

2.2.1 Le phénomène acoustique : Le signal vocal.

Le signal vocal, en tant qu'input de tout système de compréhension de la parole, se présente, à la sortie du microphone, comme un sinal acoustique numérisé.

Il est de nature extêmement variable et multiforme, en raison des processus physiques liés à la production même de la parole [HATON 85].

Retenons quelques-unes de ses caractéristiques essentielles :

* Le signal vocal est un semi-continuum dans lequel n'apparaissent pas les frontières entre les mots. Lorsqu'elle doit être faite, la segmentation du message est un problème complexe et difficile à résoudre [HATON 85].

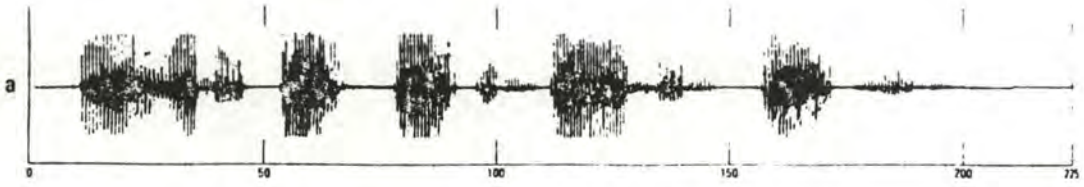


Figure 2.1: Signal acoustique numérisé.

* Il est également caractérisé par un débit d'information élevé (de l'ordre de 100 000 bps), et est extrêmement redondant, ce qui ne facilite pas son analyse [HATON 85].

* Il est très dépendant de l'environnement immédiat. En effet, il est impossible de déterminer, dans le signal que l'on traite, les sons appartenant au message oral de ceux provenant de l'environnement immédiat du locuteur ("bruits" extérieurs). La manifestation du même message oral sera donc sensiblement différente en cas de modification des conditions dans lesquelles ce dernier est produit [LEA 80].

* Il est très dépendant du locuteur. En effet, un même message oral produit par 2 locuteurs différents dans un même environnement donné aura des caractéristiques acoustiques différentes, les particularités physiques de chacun ayant une répercussion non négligeable sur le processus de production de la parole [LEA 80] [HATON 85].

* Il est également très dépendant de l'état du locuteur, c'est-à-dire de son humeur, de son débit d'élocution, de son état de santé, etc... [LEA op. cit.].

2.2.2 Le phénomène linguistique.

L'interpretation linguistique du signal acoustique constitue une opération très complexe non seulement, comme

on vient de le souligner, en raison des caractéristiques intrinsèques de ce dernier, mais également de par la nature même du processus de perception de la parole qui repose sur l'interaction de mécanismes linguistiques jouant à plusieurs niveaux, et à ce jour, pas toujours clairement compris et expliqués.
Il est hors de notre propos et de notre compétence d'en présenter ici une étude complète et détaillée. Il n'est pas inutile cependant de rappeler quelques principes de base, et de voir surtout comment ils sont transposables en traitement automatique de la parole.

Depuis l'idée qui doit être émise jusqu'au signal acoustique, un message vocal subit une série de transformations. La figure ci-dessous présente un modèle hierarchique de la perception humaine de la parole s'inspirant fort de celui de [LIBERMAN 70].

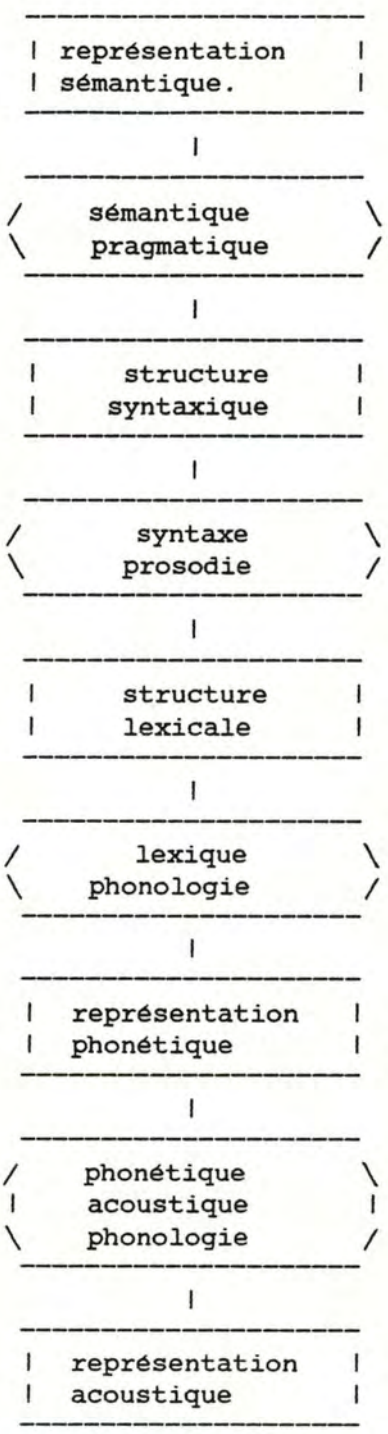


Figure 2.2: Modèle hierarchique de la perception humaine de la parole.

L'interprétation linguistique du signal acoustique s'appuye donc :

[1] sur l'utilisation de connaisances multiples et variées :

* acoustique : pour le traitement du signal ;

* phonétique : décrivant les caractéristiques des sons de la langue ;

* phonologiques : décrivant les phénomènes d'altération de ces sons dans des contextes donnés ;

* prosodiques : rendant compte des rythmes, des intonations et mélodies de la voix ;

* lexique : lié aux plus petites unités signifiantes ;

* syntaxiques : comprenant les règles de formation de phrases correctes ;

* semantiques : prenant en compte les problèmes de la signification des énoncés ;

* pragmatiques : décrivant l'univers dans lequel on se place (plus spécifiquement dans le cadre de dialogues).

[2] sur différents traitements et leurs interactions.

Classiquement, 4 niveaux d'analyse linguistique sont possibles : phonétique, lexicale, syntaxique et sémantique.

Pour comprendre les problèmes auxquels on se heurte, nous examinons un peu plus en détail chacun de ces niveaux dans les sections qui suivent.

2.2.2.1 L'analyse phonétique [LEA 80].

Le premier niveau d'analyse consiste en l'étude de la structure phonétique sous-jacente au signal acoustique. Elle présuppose l'adoption d'unités minimales de sons de parole (phone, phonème, diphone, syllabe, etc...) déterminées par des caractéristiques physiologiques et acoustiques, et la capacité de leur repérage et de leur identification dans un signal vocal donné. Le choix des segments minimaux en est donc restreint à un ensemble dont les éléments possèdent, du moins théoriquement, des caractéristiques distinctives.

L'analyse proprement dite doit être constituée d'une phase de segmentation du signal, et d'une phase d'identification de ces segments. Ces deux opérations sont difficiles à résoudre en raison du caractère continu de la manifestation acoustique du message oral, du caractère également continu

des phénomènes physiques liés à la production de la parole, provoquant un certain chevauchement des unités minimales, et finalement de la présence de phénomènes phonologiques non négligeables, modifiant relativement sensiblement suivant les cas les caractéristiques observées de ces unités de son de parole. Des écarts sont donc observés par rapport à leur définition théorique, écarts dûs soit à des variations individuelles, soit à l'environnement phonétique immédiat dans lequel elles apparaissent. Si certains phénomènes systématiques sont bien connus, d'autres, de nature totalement irrégulière, sont difficiles à mettre en oeuvre.

L'absence de solution générale à ces problèmes [LEA 80] impose aux éventuels niveaux d'analyse s'appuyant sur les résultats qu'elle peut fournir une structure phonétique relativement fortement erronnée suivant les cas, et dont il faut tenir compte si l'on désire développer un système relativement performant.

2.2.2.2 L'analyse lexicale.

Le second niveau d'analyse est constitué par la recherche de la structure lexicale de l'énoncé à partir de sa représentation phonétique ou acoustique. Cela suppose l'existence d'une représentation de chaque mot pouvant apparaitre dans un énoncé, et la capacité de pouvoir, à partir de cette représentation, déterminer si le mot est présent et à quel endroit dans le signal.

Il est facile de voir le genre de problèmes que cette démarche doit résoudre, s'il est tenu compte :

* de l'absence de frontières de mots dans la représentation acoustique et phonétique ;

* de la dépendance du signal (et donc d'une éventuelle représentation phonétique) vis-à-vis des bruits extérieurs, du locuteur, de son état, etc... ;

* et des erreurs qui ont pu être introduites dans la représentation phonétique si l'on travaille à ce niveau.

2.2.2.3 L'analyse syntaxique.

Le troisième niveau d'analyse est constituée par l'étude de la structure syntaxique de l'énoncé.
Cette analyse doit s'appuyer sur une définition du langage admis. Celle-ci est souvent faite à l'aide d'une grammaire, ensemble de symboles couplé avec une suite de règles décrivant le mécanisme de construction des chaines acceptables dans ce langage.
Bien que des grammaires relativement générales ont été développées pour le traitement du langage naturel par

ordinateur (WOODS, WINOGRAD, entre autres...), elles ne sont généralement pas utilisées telles quelles, la plupart des systèmes développés ne travaillant que sur des sous-ensembles des langues naturelles dans le cadre de tâches relativement fortement limitées. Pour cette raison, les concepteurs sont amenés à définir leur propre langage et les mécanismes le définissant, souvent de manière empirique et informelle, en se basant sur un certain nombre de corpus établis à partir de simulations réalisées dans le cadre des application visées. L'analyse proprement dite consiste à extraire d'une représentation lexicale la structure syntaxique de l'énoncé en se basant sur les règles de formation de phrases. Le caractère incertain de cette représentation et des modèles de définition des langages en complexifie sa mise en oeuvre. Cette constatation est à la base de l'idée d'interaction entre ces 2 niveaux d'analyse afin d'essayer de réduire cette incertitude.

2.2.2.4 L'analyse sémantique.

Ce dernier niveau d'analyse s'attache au problème de la signification de l'énoncé du locuteur, but ultime de tout système de compréhension de la parole.

L'interprétation sémantique d'un énoncé dépend simultanément :

de la signification des mots qu'elle contient,

des relations entre ces éléments,

et du contexte d'énonciation dans lequel elle est produite, dans le cadre plus particulier d'un dialogue et d'une application donnée.

Cette analyse est nécessairement réalisée à partir d'une structure syntaxique et/ou lexicale du message vocal.

Il est facile d'imaginer les problèmes qui en découlent en raison de l'ambiguité caractéristique du langage naturel (polysémie, homonymies, importance du contexte d'énonciation), et de l'ambiguité provenant du processus de reconnaissance (erreurs, homophones, etc...).

Il faut cependant souligner qu'il existe une différence importante entre les théories linguistiques et sémiotiques et leur application dans les systèmes de compréhension, et ce essentiellement en raison de la différence des objectifs poursuivis. La difficulté des problèmes rencontrés réduit les seconds à des domaines très limités [LEA 80].
La démarche est souvent calquée sur l'approche de la sémantique des énoncés de KRIPKE [KRIPKE 61].
Basée sur la théorie des modèles de Keisler, son point de

départ est la liaison entre des expressions logiques (extraites des énoncés), et des modèles de "mondes possibles".
Un "modèle de monde" est constitué d'une description d'objets, événements et relations existants à un moment et à un endroit donné. Les modèles de mondes possibles constituent les cadres sémantiques auxquels on peut accéder à partir du monde dans lequel on se place. Dans la plupart des systèmes, il n'existe qu'un seul monde possible, celui de l'application, ce qui facilite quelque peu l'interprétation sémantique.
Une expression logique, quant à elle, est un ensemble d'assertions propositionnelles reliées par des connectives booléennes.

Suivant cette approche, la signification d'un énoncé comprend les 3 volets suivants :

[1] extraction de l'expression logique sous-jacente à l'énoncé ;

[2] caractérisation de la relation existante entre cette expression et le modèle de monde que l'on s'est donné ;

[3] analyse de l'action caractérisée par la relation logique. Cette action va de l'extraction d'une information dans une base de données à une action plus physique comme par exemple le déplacement d'un objet.

En outre, l'interprétation sémantique des énoncés se heurte au problème de l'adoption d'un formalisme adéquat de représentation du modèle de monde, des expressions logiques, et de leur mise en correspondance avec les actions à réaliser.
A titre d'exemple, elle doit résoudre des problèmes tels que la définition du sens des mots, le traitement de synonymes (au niveau des mots, mais aussi des syntagmes et même des phrases de même signification telles que "reste !" et "ne pars pas !"), les inférences de propositions diverses à partir d'un énoncé (implications logiques, déductions, etc...) [MELONI 83].

2.2.2.5 Conclusion.

Il ressort de l'étude des phénomènes acoustique et linguistique que les caractéristiques essentielles du problème de la compréhension automatique de la parole sont :

[1] l'absence de solutions générales aux problèmes qu'elle pose ;

[2] le caractère non déterministe du processus, essentiellement dû à l'environnement incertain dans lequel il prend place (variabilité du signal, présence d'erreurs à tous les niveaux) ;

[3] et finalement, en tant que conséquence des 2 premières, la nécessité d'intégrer tous les connaissances disponibles à tous les niveaux d'analyse, jusqu'à une coopération étroite entre ces différents traitements, afin de réduire l'incertitude et l'explosion combinatoire qui en découle. Ainsi, comme on le verra, la connaissance utilisée à un niveau d'analyse peut également jouer comme une contrainte à un niveau plus bas.

Devant la spécificité du problème, les premiers travaux sont caractérisés par une approche tendant à limiter les difficultés, afin de leur apporter, tour à tour et de manière progressive, la solution la plus adéquate possible [LEA 80].
Avant d'aborder la parole continue, nous allons quelque peu nous attarder sur les solutions adoptées en reconnaissance des mots isolés. Mais avant, il faut dire quelques mots sur les démarches générales que l'on rencontre dans les principaux systèmes.

2.3 Les différentes approches possibles.

Il existe différentes approches possibles au problème de la reconnaissance et de la compréhension de la parole. Classiquement, on distingue celles qui sont principalement basées sur des méthodes de type mathématique , et celles qui font intervenir les considérations linguistiques en relation avec la perception de la parole chez l'homme.

L'approche purement mathématique [LEA 80] recouvre en fait 2 types de méthodes. D'une part, celles basées sur le fait que chaque input du système doit être comparée avec des modèles pré-enregistrés représentant une classe d'équivalence d'input, et le plus proche "voisin" (i.e. celui se différenciant le moins du signal en entrée) identifie le type d'input courant.
Les comparaisons s'appuyent sur des techniques classiques de reconnaisance de formes mettant en oeuvre des modèles mathématiques et statistiques. A la limite, ces modèles ne sont pas particuliers au traitement de la parole, mais pourrait être utilisés pour le traitement d'autres types de signal.
D'autre part, il existe celles qui sont basées sur des modèles statistiques et stochastiques très élaborés. Leur étude sort du cadre de ce rapport.

A l'autre extrême, une approche plus linguistique consiste à "... affirmer que le signal acoustique n'est pas à lui seul suffisant pour déterminer le message...d'autres sources de connaissances doivent être intégrées pour résoudre le problème de la reconnaissance [à fortioiri celui de la compréhension] " [LEA 80]. Elle se base sur le fait que l'être humain peut être considéré comme un système prototype et qu'il peut nous aider à développer de tels systèmes. Il ne s'agit pas de recréer structurellement le mécanisme humain, mais plutôt de développer quelque chose qui fonctionnellement s'apparente à lui.
Pour reprendre la distinction introduite par Weizenbaum dans [weizenbaum 84], elle se situe pleinement dans une démarche de type "performance" et pas "simulation".

Cette approche relève entièrement du domaine de l'intelligence artificielle, en ce sens qu'elle comprend l'intégration et la coopération de sources de connaissances variées, ainsi que des techniques de recherche de solution dans un espace, ce dernier étant ici composé des diverses interprétations partielles ou totales du message vocal étudié. C'est ce type de systèmes que nous analyserons par la suite.

Il faut insister sur le fait que la distinction entre les 2 approches ne correspond pas à une simple dichotomie.

Les sytèmes orientés mathématiques utilisent en effet des unités linguistiques telles que les mots, etc..., et dans les systèmes les plus orientés linguistiques, on utilise, comme on le verra, des techniques mathématiques d'analyse du signal.

2.4 La reconnaissance des mots isolés [LEA 80] [HATON 85] [PIERREL 82].

Comme on l'a vu dans la première partie de ce chapitre, dans ce genre de communication, les messages oraux sont constitués d'une suite de mots séparés par des pauses suffisamment longues que pour pouvoir traiter chaque mot isolément. L'interpretation du signal vocal revient ici à une analyse lexicale s'appuyant éventuellement sur une analyse phonétique. Au sein même de cette analyse, les gros problèmes de détermination des frontières entre les mots et de prise en compte des phénomènes phonologiques entre les mots sont donc purement et simplement évacués. Dans les systèmes pour lesquels les mots sont reliés syntaxiquement entre eux, une analyse grammaticale est également réalisée. Dans la plupart des cas, elle est relativement simple en raison du caractère très contraignant de la syntaxe adoptée.

Dans la plupart des systèmes ayant présentés de bon résultats, l'approche adoptée, dite globale, est clairement du type mathématique [HATON 85] [PIERREL 82]. L'analyse lexicale consiste à identifier le mot prononcé en comparant un certain nombre de caractéristiques observées avec des modèles pré-enregistrés. Le mot reconnu est celui dont le modèle est le plus proche "voisin" de la description du mot prononcé, par rapport à une certaine mesure de "distance".
Les caractéristiques qui font l'objet de la comparaison varient, de système à système, du signal acoustique entier à des paramètres extraits de ce dernier.
Cette démarche est appelée globale, car elle se fonde plus sur l'aspect global du mot que sur sa structure ou sa composition en éléments minimaux.

Retenons encore que vers 1960, DENES et MATHEWS introduisirent le concept important de normalisation temporelle, par lequel la représentation du mot prononcé est adaptée à celle pré-enregistrée au niveau de la vitesse de prononciation, en étirant ou en comprimant le signal acoustique en entrée afin de lui donner même durée que le modèle. L'expérience montre que cette technique diminue sensiblement le taux d'erreurs [LEA 80].

Enfin, presque tous les systèmes utilisent maintenant un algorithme de programmation dynamique pour la comparaison des formes en entrées avec les formes pré-enregistrées (pour plus de détails, voir [LEA 80]).

L'approche globale, bien que présentant de bons résultats pour un vocabulaire restreint (de l'ordre de 100 mots), comporte un certain nombre de limites :

[1] une phase d'apprentissage est nécessaire afin de pré-enregistrer les modèles de référence et leurs caractéristiques distinctives ;

[2] la qualité des résultats obtenus est fortement liée au fait que le locuteur et la personne ayant réalisé l'apprentissage soient la même personne (aspect monolocuteur).

[3] l'inadéquation à la parole continue pseudo-naturelle, et ce "... tant par le volume d'informations à stocker pour représenter les formes acoustiques d'un grand lexique (par exemple 10 000 mots) que par la relative inéfficacité des algorithmes de comparaison de formes devant la variabilité de la parole continue." [HATON 85].

Le passage au traitement de la parole continue nécessite, comme on va le voir, une démarche plus analytique.

2.5 La compréhension de la parole continue.

Donc, il existe plusieurs approches du problème de la compréhension de la parole. Nous restreignons ici notre analyse uniquement aux principes de solution apportés par les systèmes plus orientés linguistique, intégrant différentes sources de connaissances de ce type, car ils sont de loin les plus nombreux. De plus, c'est également l'approche qui a été adoptée à Nancy, comme nous le verrons plus loin. Pour rappel, de tels systèmes sont composés d'un certain nombre de composantes utilisant diverses sources de connaissances et coopérant pour la réalisation de l'objectif ultime suivant des stratégies déterminées.

Cette démarche postule nécessairement :

[1] la modélisation des connaissances disponibles, c'est-à-dire :

  [1.1] la définition des connaissances que l'on peut utiliser ;

  [1.2] le choix d'un formalisme de représentation et de mise en oeuvre ;

[2] la définition d'une structure de contrôle.

Elle aboutit à la conception de systèmes dont les principales composantes, correspondant grosso modo aux quatre niveaux d'analyse linguistique étudiés au début de ce chapitre, sont [LEA 80] :

- le décodage acoustico-phonétique, utilisant les connaissances de type acoustique, phonétique, et phonologique ;
- l'analyse lexicale, utilisant les connaissances de type lexicale, phonétique, et phonologique ;
- l'analyse syntaxique ou syntaxico-sémantique, utilisant des informations de type syntaxique, sémantique, et prosodique ;
- le traitement sémantico-pragmatique, utilisant des informations sémantiques, pragmatiques, et éventuellement prosodiques.

et mettant en oeuvre une stratégie d'utilisation de ces informations.

Les 2 sections suivantes traitent respectivement des différentes conmposantes d'un tel système de traitement de la parole et des stratégies de contrôle à y inclure.

2.5.1 Les composantes d'un système de traitement de la parole.

2.5.1.1 Décodage acoustico-phonétique [LEA 80].

Le décodage acoustico-phonétique nécessite la définition d'unités minimales de son de parole, et la mise en oeuvre d'un mécanisme permettant de les repérer et de les identifier dans le signal.

Le choix des unités minimales peut se faire suivant un ensemble de critères. Il existe beaucoup de possibilités (allophones, phonèmes, diphones, syllabes,...), chacune présentant un certain nombre d'avantages et d'inconvénients. Il semble cependant que l'unité la plus souvent utilisée est une unité de type phonème, c'est-à-dire grosso modo de même nature que le phonème de la linguistique traditionnelle. Ce dernier est "...utilisé pour l'ensemble complet des allophones [un allophone étant un ensemble d'éléments minimaux de son d'une langue donnée ayant mêmes caractéristiques distinctives] se comportant de manière similaire et ne se différenciant pas de manière sinificative les uns par rapport aux autres au sein d'une même langue"[LEA 80].
Ces unités possèdent un certain nombre de caractéristiques acoustiques et physiologiques, représentées par des valeurs de paramètres acoustiques (voir plus loin).

Le décodage acoustico-phonétique proprement dit est composé d'une phase de segmentation du signal vocal, puis d'une phase d'étiquetage phonétique (ou plutôt phonémique) des segments obtenus.
Ces 2 phases sont interdépendantes, et il n'est pas toujours facile de déterminer la limite entre les deux.
De plus, elles sont souvent précédées d'une phase de prétraitement du signal. Comme on l'a vu, il est caractérisé par un débit d'information fort élevé. Généralement, une autre représentation est adoptée, obtenue à partir d'une analyse fréquentielle de ce signal. Elle permet de travailler sur un débit d'information réduit, en gardant, dans la mesure du possible, toute l'information pertinente. Il s'agit d'une "...représentation temps/fréquence/intensité... correspondant à une succession de spectres à court terme du signal vocal" [HATON 85].

Cette représentation offre également l'avantage d'être facilement utilisable pour le calcul d'une série de paramètres acoustiques caractérisant le signal (pour plus de détails sur ces paramètres, voir LEA Ch. 5).
Les valeurs de ces paramètres sont calculés de manière périodique (50 à 200 fois par seconde).

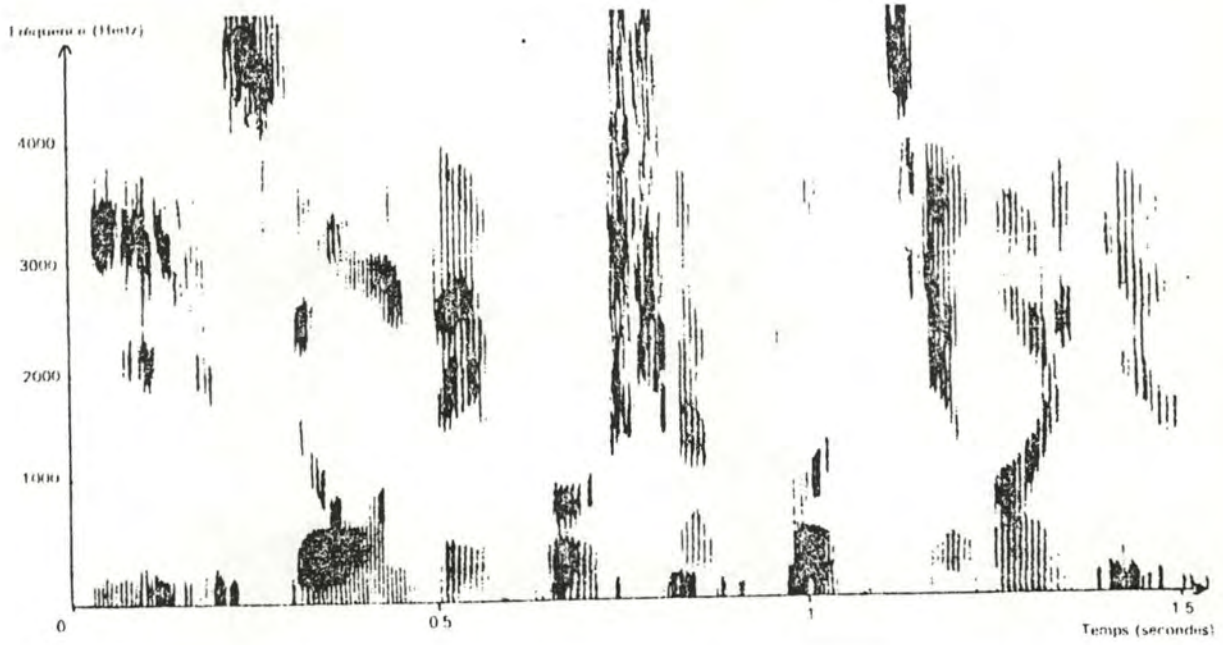


Figure 2.3: Spectrogramme.

Une fois cette première analyse réalisée, il faut segmenter le signal et étiqueter les segments ainsi obtenus. Il s'agit d'une tâche difficile car elle fait appel à la phonologie de la langue et aux caractéristiques acoustiques et physiologiques des unités minimales de son de parole. Et notre connaissance en ce domaine est encore incomplète.

Il existe actuellement deux approches différentes qui peuvent être utilisées.Le rôle du contexte dans la réalisation acoustique des sons a déjà été souligné. La première méthode est basée sur une approche ne tenant pas compte de ce contexte. Dans la seconde, ce dernier joue un rôle prépondérant.

2.5.1.1.1 Méthodes indépendantes du contexte.

Dans les méthodes indépendantes du contexte, le principe sur lequel s'appuie la segmentation est le suivant : chaque fois que les valeurs des paramètres calculés lors de la phase de prétraitement varient au delà d'un certain seuil, il y a frontière, et donc début d'un nouveau segment.

Ce principe a l'avantage d'être bien défini et simple à mettre en oeuvre.

Avec cette méthode, l'étiquetage phonétique des segments est réalisé à partir d'un matching avec des modèles pré-enregistrés, sur base des valeurs des paramètres calculés.

2.5.1.1.2 Méthodes dépendantes du contexte.

Plutôt que d'utiliser les mêmes caractéristiques pour déterminer tout type de frontière, cette seconde méthode met en oeuvre des caractéristiques et des seuils de variation en fonction du contexte phonétique immédiat.
Par exemple, s'il est établi que la partie courante du signal correspond à tel type de son (macro-classe), on peut savoir, à la lumière des connaissances phonologiques dont on dispose, les variations auxquelles il faut s'attendre pour les phonèmes suivants. ceci permet de se focaliser sur un nombre (restreint) de paramètres à observer et des seuils de variation spécifiques. La frontière sera introduite quand les valeurs des paramètres dépassent ces seuils.
En d'autres mots, plutôt que d'appliquer simplement des seuils à un ensemble de paramètres acoustiques, on utilise un certain nombre d'heuristiques et de règles s'appuyant sur les résultats de l'application des règles antérieures.

L'étiquetage se déroule alors de la manière suivante : en utilisant la connaissance que l'on a de la nature acoustique des différents phonèmes, des caractéristiques acoustiques sont extraites et sont utilisées par des heuristiques ou des techniques statistiques pour déterminer les étiquettes.

Bien évidemment, il existe également des méthodes "hybrides" combinant ces 2 approches.

2.5.1.1.3 Le problème des erreurs de segmentation.

Une erreur de segmentation peut se définir comme étant une erreur provenant soit du fait que plusieurs segments ont été regroupés en un seul, soit qu'un segment a été éclaté en plusieurs autres.

Une segmentation sans erreur est une tâche extrêmement difficile, voire impossible, car :

- Les frontières entre segments ne se manifestent pas de manière uniforme (certaines sont plus apparente que d'autres) ;

- une représentation satisfaisante de la connaissance acoustico-phonétique n'a pas encore vu le jour. Par exemple, si certains phonèmes peuvent être détectés en visualisant le spectrogramme du signal, il n'existe pas encore de méthode automatique réalisant la même opération.

- les caractéristiques acoustiques des segments de nature phonétique peuvent se recouvrir.
ex: bu s s top
a-t-on un /s/ long ?
a-t-on 2 /s/ successifs ?
Ce genre d'ambiguité ne peut être résolu qu'avec d'autres sources de connaissances (essentiellement lexicale et syntaxique).

En raison de l'inévitable présence d'erreurs, le traitement acoustico-phonétique ne fournit pas une structure phonétique, mais dans la plupart des systèmes un treillis d'unités phonétiques correspondant à plusieurs hypothèses, chacune éventuellement pondérée par score calculé de manière déterminée.
Les autres niveaux d'analyse tenteront de lever les ambiguités en utilisant leurs propres sources de connaissances qui ne sont pas utilisables à ce niveau.

2.5.1.2 Analyse lexicale [LEA 80].

Ce second niveau de traitement a pour objectif d'établir la structure lexicale de l'énoncé oral.

En fait, la plupart du temps, il comporte les 2 volets suivants :

- vérification : comparaison de la description d'un mot et de l'énoncé pour déterminer si le mot a été prononcé, et avec quel degré de certitude ;

- hypothétisation : établissement d'une liste de mots hypothèses à partir d'un fragment de l'énoncé et d'une description du lexique.

Ces 2 traitements se font soit sur base d'une représentation acoustique, soit sur base d'une représentation phonétique sous forme de treillis de phonèmes.
Ils présupposent également une certaine représentation de l'ensemble des mots constituant le lexique admis, diverses possiblités d'accès à ces mots.
Le terme "mot" désigne des mots de la langue, ainsi que des syntagmes considérés comme irréductibles ("c'est ça", "c'est-à-dire",...).

Il faut bien voir qu'il n'existe pas de "mapping" direct entre l'information acoustique ou phonétique, et un mot unique du vocabulaire admis, et ce en raison :

- des bruits extérieurs ;

- des différences d'un locuteur à l'autre ;

des différences du même locuteur d'un moment à l'autre ;

- des différentes prononciations possibles ;

- des phénomènes phonologiques (ex :co-articulation).

Avant de s'attarder un peu plus sur les 2 traitements composant l'analyse lexicale, il faut souligner que la partie vérification travaille à partir de mots-candidats, provenant soit des niveaux "supérieurs" (syntaxe, sématique), soit du niveau inférieur (l'hypothétiseur de mots), et vérifient s'ils se trouvent dans la chaine en entrée.
Cela signifie que le niveau lexical constitue le niveau charnière en ce sens qu'il fait le lien entre :

- les connaissances cognitives que l'on a de la langue (syntaxique, sémantique) et de l'application (sémantique, pragmatique), dans le cadre d'une approche "top-dowm" ;

- Les connaissances indépendantes de l'application (acoustiques, phonétiques, et phonologiques) dans le cadre d'une approche "bottom-up".

2.5.1.2.1 Le vérificateur de mots.

Le but du vérificateur de mots est d'examiner une liste de mots-candidats et un endroit particulier dans la chaine en entrée et de déterminer le mot le plus probable en fonction des informations acoustiques et phonétiques dont il

dispose.

Cette tâche se réalise en :

- extrayant du lexique la représentation phonétique/acoustique de chaque candidat ;

- transformant cette représentation, via des règles contextuelles et phonologiques, reflétant la connaissance que l'on a des perturbations possibles dûes à la co-articulation, aux différences de prononciation, et aux absences ou insertions excessives d'élément la reflétant ;

- comparant la partie de la chaine en entrée avec toutes les variantes des représentations de chaque mot-candidat et déterminer la plus proche.

Ces trois opérations sont fort liées au fait que l'on travaille directement sur le signal (paramétré), ou sur une représentation phonétique.

A titre d'exemple, une méthode consiste à représenter chaque mot du lexique par une suite d'états acoustiques (phonétiques) correspondant à des formes spectrales (des macro-classes phonétiques), puis à établir un réseau de transition d'états représentant les perturbations possibles de la suite des états de départ, et réalisé à l'aide de règles de type phonologiques.
L'algorithme de matching est ensuite une mise en oeuvre de la méthode de programmation dynamique (voir ci-dessus) pour la détermination d'un chemin optimal à travers un ensemble fini d'états.

2.5.1.2.2 Hypothétiseur de mots.

L'objectif de l'hypothétiseur de mots est de produire, à partir d'une partie de la chaine en entrée et d'une description acoustique ou phonétique des mots du vocabulaire, une liste de mots-hypothèses susceptibles d'y être présents. Elle correspond à une approche de type "bottom-up", du signal vers le lexique.
Cette fonction trouve surtout son utilité lorsque le langage devient relativement élaboré, le facteur de branchement moyen, c'est-à-dire une mesure de la complexité de ce langage correspondant au nombre moyen de mots-candidats en un point donné de l'analyse, étant assez élevé, les informations de bas niveau peuvent être utilisées comme contraintes de la recherche.
L'objectif de cette fonction est donc bien de rendre la reconnaissance plus facile ou plus rapide, mais pas meilleure.

Une des méthodes les plus utilisées consiste à s'appuyer sur une structure de données combinant les descriptions de tous les mots du vocabulaire pour identifier la partie de la chaine en entrée. Pendant la recherche, un "matching" est réalisé sur des petites parties des mots, pour arriver finalement à la liste des mots-hypothèses.
Grosso modo, cette méthode revient à accéder à un sous-ensemble du lexique, via un index (la structure de données), et sur base d'une clé complexe (la partie de la chaine à identifier).

2.5.1.3 L'analyse syntaxique ou syntaxico sémantique [LEA 80][PIERREL 81].

La mise en oeuvre de la définition du langage peut être utilisée à 2 niveaux :

- d'une part, l'établissement de la structure de surface et de la structure profonde de l'énoncé, déterminant les relations grammaticales entre ses différents composants, sont des éléments importants pour la détermination de sa signification ;

- d'autre part, étant donné le caractère incertain de l'analyse lexicale, elle peut jouer le rôle de contrainte sur l'agencement des mots.

Pour cela, des modèles de définition et de représentation du langage sont adoptés. Certains sont purement syntaxiques, d'autres intègrent des connaissances de type sémantique ou pragmatique. C'est la raison pour laquelle on parle aussi d'analyse syntaxico-sémantique.

La façon dont l'analyse syntaxique est réalisée dépend fortement du modèle choisi. Retenons cependant que plusieurs stratégies peuvent être adoptées :

- gauche-droite : en commençant au premier mot de la phrase et en construisant l'arbre syntaxique, tout en vérifiant les contraintes d'acceptabilité suivant l'ordre dans lequel les mots se présentent ;

- du milieu vers les côtés : en commençant cette fois à partir d'ilôts de confiance correspondant à de bons scores de reconnaissance, et en réalisant l'analyse en se dirigeant des 2 côtés.

2.5.1.3.1 Les modèles purement syntaxiques.

Les plus employés sont les grammaires hors contexte et les grammaires transformationnelles de Chomsky.

- Grammaires hors-contextes de Chomsky.

  Ces grammaires sont composées :

  (a) d'un alphabet contenant des symboles terminaux ;

  (b) d'un ensemble de symboles non terminaux, avec un symbole "distingué";

  (c) d'un ensemble de règles de grammaire, de la forme :
  X->t
  où X est un non terminal et t est la concaténation de symboles terminaux et/ou non terminaux,
  exprimant une façon possible de transformer un non terminal en une séquence d'un ou plusieurs symboles (terminaux ou non terminaux).

  de manière générative, ces règles permettent d'imposer des contraintes sur une partie de l'énoncé à reconnaitre ;
  de manière analytique, elles permettent de tester la grammaticalité des phrases et, par effet de bord, de construire son arbre syntaxique.

  - Grammaires transformationnelles de Chomsky.

    Basées sur le même principe, elles tiennent compte en plus du fait qu'une phrase ayant même structure profonde peut se manifester avec des structures de surface différentes.
    Elles permettent donc de ramener plusieurs énoncés ayant une structure de surface distincte à une forme "normale".

  Devant les incertitudes liées au processus de reconnaissance, les concepteurs décident parfois d'intégrer des contraintes sémantiques dans la syntaxe. C'est la raison d'être des modèles syntaxico-sémantiques.

2.5.1.3.2 Les modèles syntaxico-sémantiques.

Nous présentons ici excessivement brièvement les grammaires sémantiques, les grammaires par cas de Fillmore, et les grammaires systémiques.

- Les grammaires sémantiques.

  Il s'agit de grammaires de type hors-contexte, mais où les terminaux désignent des classes sémantiques et non plus des catégories syntaxiques [PIERREL 81].

- Les grammaires par cas de Fillmore.

  Ce modèle s'appuye sur une constatation de Fillmore selon laquelle il n'est plus nécessaire de déterminer l'arbre syntaxique complet d'une phrase, mais plutôt de s'attacher aux éléments fondamentaux et aux relations existant entre ces éléments sous forme d'attributs syntaxico-sémantiques (les cas).

- Les modèles systémiques (Halliday).

  Ces modèles font apparaitre dans une sorte de graphe, les différents cas possibles (les "systèmes"), et les décisions à prendre en compte lors de l'analyse d'une phrase.

  Nous retiendrons les Augmented Transition Networks de Woods, et les Réseaux à noeuds Procéduraux de J-M Pierrel.

  (a) Un A.T.N. de Woods se présentent comme un réseau de transition du même type que ceux utilisés pour représenter les grammaires hors-contexte, mais avec en plus des procédures rajoutées sur les arcs de transition, permettant de construire la structure de la phrase et prenant en compte le côté transformationnel de Chomsky.

  (b) Les R.N.P de J-M. Pierrel sont basés sur le même principe. Nous en ferons une présentation un peu plus détaillée dans le chapitre 4.

2.5.1.4 Analyse sémantico-pragmatique [LEA 80,HATON 85,PIERREL 82]

Nous regroupons ici le traitement sémantique et pragmatique en un seul car les informations sur lesquelles ils se basent sont très proches.
D'une part, les connaissances sémantiques portent sur la détermination du sens d'une phrase en général, à partir de la signification des mots qu'elle contient et des relations entre ces éléments, et d'autre part, les connaissances pragmatiques définissent le contexte d'énonciation dans le cadre plus particulier d'un dialogue et d'une application donnée.
Il est certain que ces trois éléments sont conjointement nécessaires pour la détermination du sens d'un énoncé, le dernier étant simplement plus spécifique que les 2 premiers

[HATON 85]. Enfin, rappelons aussi que nous nous intéressons ici plutôt à la compréhension de phrases, la spécificité du dialogue sera présentée en détail plus loin.

Notre connaissance du domaine de la compréhension est très faible et très incomplète. Les solutions apportées sont donc très ponctuelles et dépendantes des applications visées.

On remarque cependant que, en général, le traitement sémantico-pragmatique assure 2 fonctions :

- l'établissement de la représentation sémantique de l'énoncé, ainsi que son interpretation.
  Ceci présuppose d'abord la définition sémantique des éléments du lexique, et un formalisme de représentation des significations qui leur sont rattachées.
  Egalement, un formalisme de représentation des relations sémantiques entre les différents constituants de l'énoncé est nécessaire.
  Un certain nombre de modèles ont été mis au point, dont le plus connu est sans doute celui des dépendances conceptuelles de R. Schank.

  L'interprétation nécessite également une certaine représentation du modèle du monde dans lequel on se place.
  Dans le cas d'un dialogue, comme on le verra, il s'agit également de pouvoir réaliser un certain nombre de déductions venant compléter l'historique des échanges et le modèle du locuteur. Ces 2 derniers points font partie du contexte d'énonciation d'un énoncé.

- une contrainte supplémentaire sur des séquences de mots syntaxiquement valables, et le rejet des interprétations syntaxiquement correctes, mais sémantiquement inacceptables.

2.5.1.5 La prosodie [LEA 80].

S'il est établi que la prosodie peut être une aide appréciable pour le processus de reconnaissance et de compréhension, les réalisations dans ce domaine sont par contre plutôt rares.
Nous nous limiterons donc à une présentation succinte des notions de base.

La prosodie constitue une source de connaissance assez différente des autres en ce sens qu'elle est indépendante des résultats des autres niveaux de traitement, et donc a priori plus fiable. Il est en effet difficile de produire des résultats corrects à partir de données relativement

fortement erronées, ce qui est malgré tout caractéristique des autres analyses.

Elle constitue essentiellement une source d'informations appréciable au niveau de la détermination de la structure syntaxique et sémantique d'un énoncé oral.

- D'une part, on remarque la présence d'accents toniques sur les syllabes des mots importants de la phrase. Ces accents sont tels que la structure phonétique est plus proche de la structure phonémique sous-jacente. Ainsi, ces éléments constituent-ils des ilôts de confiance à ce niveau d'analyse. De plus, ils sont souvent associés à des structures syntaxiques spécifiques. On observe en effet que les mots "fonctionnels", tels que les prépositions, articles, etc... sont souvent non accentués, alors que les noms et la plupart des verbes et adjectifs le sont. Enfin, il semblerait aussi que les éléments de propositions subordonnées le sont rarement.

- D'autre part, la durée des unités phonétiques, ainsi que les longueurs des intervalles de temps entre 2 syllabes accentuées sont des indicateurs de fin de proposition et de vitesse de prononciation.
  Ceci suppose évidemment que la normalisation temporelle soit faite de manière prudente.

- Enfin, l'intonation, et plus particulièrement les "vallées" (i.e. les fortes diminutions suivies de fortes augmentations) peuvent être utilisées comme indicateurs de frontière entre propositions et entre constituants syntaxiques, ainsi que comme indicateur du type de phrase (interrogative, affirmative,...).

Les informations prosodiques peuvent être évaluées à partir de l'énergie et de la fréquence du signal acoustique, idéalement à l'intérieur de bandes bien spécifiques.

Au niveau des réalisations, lorsqu'elle est prise en compte, la prosodie est souvent rajoutée aux systèmes existants pour vérifier ou modifier les hypothèses provenant des autres niveaux d'analyse. C'est le cas, par exemple, pour le système HWIM, dont on reparlera plus loin.

2.5.2 Les stratégies de contrôle [LEA 80].

S'il est établi que les différentes sources de connaissances doivent communiquer et coopérer, on ne sait pas encore de manière précise comment cela doit

effectivement et efficacement se réaliser.

Le problème du contrôle regroupe l'interaction des sources de connaissances, leur activation, et la méthode qu'elles utilisent.
Il s'agit d'un problème très important car l'expérience montre que la solution adoptée peut avoir une influence directe sur les performances (vitesse et pertinence) du système.

2.5.2.1 Modèles d'interaction des sources de connaissances.

Plusieurs modèles d'interaction des différentes sources de connaissances ont été utilisés.

Retenons les principales :

- le modèle hiérarchique (ou ascendant) : dans lequel toutes les communications se font de manière "bottom-up", d'un niveau à l'autre, en partant du niveau acoustique vers le niveau sémantique. Si cette méthode est simple au niveau du contrôle, elle présente l'inconvénient d'imposer à un niveau de traitement une seule représentation du signal (celle du niveau directement inférieur), et rend le rattrapage des erreurs très difficile, si pas impossible.

- le modèle "goal-oriented" (ou descendant) : dans lequel les niveaux supérieurs réalisent un certain nombre d'hypothèses qui sont complétées et partiellement évaluées par les niveaux inférieurs, et finalement confrontées au signal par le niveau acoustique. Le mode de communication est donc de type "top-down". Pour une grammaire peu contraignante et un lexique assez étendu, cette méthode se heurte à un problème d'explosion combinatoire.

  Plusieurs systèmes utilisent une combinaison de ces 2 premières approches, comme par exemple MYRTILLE II dont on reparlera plus loin.

- Le modèle "hétérarchique" : dans lequel chaque niveau de connaissances peut communiquer à tout moment avec n'importe quel autre. Le contrôle de la recherche est alors très lourd à gérer.

- Le modèle "blackboard" : dans lequel chaque source de connaissances communique avec les autres via une structure de données. Chaque niveau évalue le contenu de cette structure, puis évalue les hypothèses qui s'y trouvent en fonction de ses connaissances propres, et éventuellement y dépose les siennes.
  Il présente également l'inconvénient d'être très lourd

à gérer.

2.5.2.2 Activation des sources de connaissances.

Etant donné la spécificité du problème, le but d'un système de compréhension de la parole est de trouver l'interprétation la plus plausible par rapport aux connaissances à tous les niveaux. Le principe généralement suivi dans les systèmes opérationnels est celui de génération d'hypothèses à chaque niveau, puis de test de ces hypothèses à d'autres niveaux ("generate and test paradigm").
Trouver l'interprétation la plus plausible d'un énoncé revient donc à un problème de recherche dans un espace constitué des diverses interprétations partielles ou totales de cet énoncé.

Plusieurs stratégies sont possibles. Nous présentons ici les principales d'entre elles.

- En profondeur d'abord.
  Le principe est d'examiner une hypothèse à fond avant de passer à une autre, c'est-à-dire de regarder ses implications à tous les niveaux avant d'en essayer une autre.

- En largeur d'abord.
  Il s'agit alors d'examiner toutes les hypothèses à un niveau donné avant de passer aux autres.
  Cette méthode comporte le risque de l'explosion combinatoire.

- Les recherches statistiques.
  Ces méthodes supposent que les alternatives soient triées à chaque niveau en fonction d'un certain score de reconnaissance.

  On distingue :

  [3.1] Le meilleur d'abord. On examine, par une stratégie en profondeur d'abord, la meilleure alternative.

  [3.2] La recherche en faisceau. On examine par une stratégie en largeur d'abord les hypothèses les meilleures.

2.5.2.3 Les méthodes.

Les méthodes les plus couramment utilisées sont :

- de gauche à droite : le signal est étudié de manière séquentielle depuis sont début, et vers la droite ;

- du milieu vers les cotés : en se basant sur des ilôts de confiance correspondant à des parties de l'énoncé pour lesquels les hypothèses ont de bons scores, et en se dirigeant vers la gauche et la droite.

2.6 Conclusion.

En raison de l'absence de solutions générales aux problèmes qu'elle pose, la compréhension de la parole est actuellement un processus de décision non déterministe prenant place dans un environnement caractérisé par l'incertitude et la présence d'erreurs.

Les techniques utilisées en reconnaissance des mots isolés ne sont pas généralisables à la parole continue.
Une approche possible consiste à développer un système composé de toutes les connaissances disponibles (acoustiques, phonétiques, phonologiques, prosodiques, lexicales, syntaxiques, sémantiques, et pragmatiques), communiquant et coopérant pour établir l'interpretation la plus plausible du signal vocal.
La recherche de cette solution se fait alors par des stratégies de recherche de solutions dans un espace composé des différentes interprétations partielles et totales. Ces systèmes relèvent à part entière du domaine de l'Intelligence Artificielle. Le système de gestion de dialogue développé à Nancy s'appuye sur un mécanisme de compréhension de phrase de ce type. Avant de passer à l'étude du dialogue qui nous intéresse principalement, nous allons illustrer les principes généraux de la compréhension de phrases sur quelques systèmes existants.

3 Illustration des principes généraux.

Différents efforts de recherche ont été réalisés çà et là depuis vingt ans en matière de reconnaissance et de compréhension de la parole.

On ne peut évidemment passer sous silence l'effort tout particulier réalisé aux Etats Unis au cours des années '70 sous l'implusion de l'Agence pour les Projets de recherche avancée (ARPA), l'effort qui se traduisit par l'émergence d'une série de systèmes tels que Hwim, Hearsay II, Harpy, Dragon, etc ...

Notre objectif n'étant pas d'établir un panorama très large à défaut d'être complet, nous nous bornerons à illustrer cet effort de recherche en ne présentant qu'un seul système américain, Hwim, réalisé par Bolt Beranek et Newman (BBN). Ce système offre l'avantage d'être basé sur un design dont bien d'autres concepteurs se sont inspirés par la suite pour leur architecture, même si les performances et les résultats de Hwim furent loin d'être les meilleurs parmi ceux de tous les autres systèmes de la même époque. Celles-ci sont principalement dues au caractère fort peu restreint du langage admis se traduisant par un facteur de branchement élevé.

L'importance de l'effort consenti par les américains ne doit néanmoins pas effacer les recherches menées ailleurs en Europe et au Japon par exemple. Mentionnons à cet effet l'effort francais, concrétisé par des systèmes tels que Esope [Mariani 82], Keal [Quinton 82], Myrtille I et II [Pierrel 81]. Nous présentons une description de ces deux derniers sytèmes tant pour illuster une fois encore les principes généraux qui ont été développés que pour introduire le lecteur à la philosophie du système dans le quel s'insèrent nos travaux.

Le système Hwim que nous allons maintenant présenter, s'inscrit dans le cadre de recherches menées aux Etats-Unis dans les années 70. Les objectifs de ce programe de recherche étaient clairement définis dans un cahier des charges [Newell] et indiquaient que les systèmes devraient

- accepter la parole continue.
- accepter plusieurs locuteurs coopératifs (non simultanément).

- travailler dans de bonnes conditions accoustiques (dans une pièce calme et avec un microphone de bonne qualité)
- pouvoir comprendre un vocabulaire de 1000 mots.
- pouvoir répondre dans un temps relativement court lorsqu'ils seraient implantés sur des machines de 100 Mips et plus.

C'est le projet ARPA qui est à l'origine de la distinction faite par la suite entre les appellations de reconnaissance et de compréhension de la parole, différence qui a été évoquée ci-dessus.

3.1 Le système HWIM.

3.1.1 Introduction.

HWIM, pour Hear What I Mean, a été implémenté sur PDP 10 tournant sur sytème Tenex, (adressage virtuel et temps partagé). Chacun de ses modules était mis en oeuvre par un processus particulier fonctionnant de façon autonome sur la machine, ceci pour différentes raisons.

- La réalisation des différents modules devait pouvoir se faire dans des langages différents.
- La capacité d'adressage d'un seul processus était insuffisante par rapport aux besoins de l'entièreté du système.
- Le système devait pour des raisons qui seront expliquées par la suite jouir d'une grande modularité.
- Cela semblait plus facile aux concepteurs.

Chacun des modules renfermait des connaissances propres.

3.1.2 Description de l'architecture du système HWIM.

Cette description a pour objet de donner une idée des composants de HWIM et du rôle que joue chacun d'eux au sein du système. Elle se veut essentiellement indicative et n'a pas la prétention d'être exhaustive. Pour plus de détails le lecteur pourra se réferrer à [Lea 80] ou à [Woods].

3.1.2.1 Composants de saisie et de digitalisation du signal. (RTIME) et (PSA

Ces deux composants ont pour rôle l'acquisition et la digitalisation de l'énoncé. De plus ils extraient du signal obtenu une série de paramètres qui seront utiles ensuite au module acoustico-phonétique. Ces paramètres sont l'énergie totale, les énergies dans les basses, moyennes et hautes fréquences.

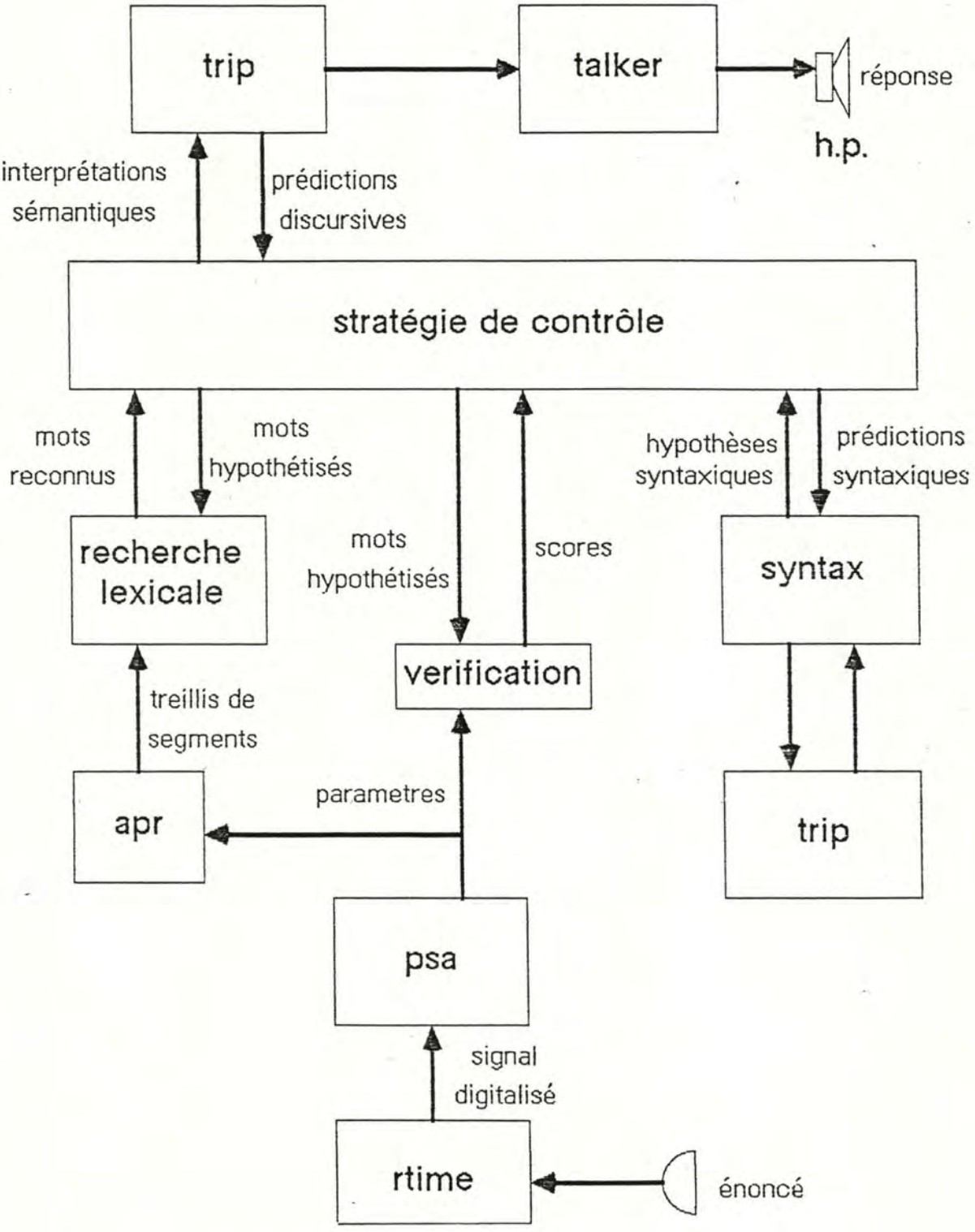


Figure 3.1: Schéma de l'architecture générale de HWIM.

3.1.2.2 Composant de reconnaissance acoustico-phonétique. (APR)

Ce composant est le module du système en charge du traitement acoustico-phonétique. Il tavaille sur la représentation de l'énoncé qui lui est fournie par les modules RTIME et PSA ainsi que sur les différents paramètres fournis par ceux-ci.

Il réalise trois tâches :

- Une tâche de segmentation dont l'objectif est de déterminer les limites des segments phonétiques qui composent l'énoncé.
- Une tâche d'étiquettage qui vise à caractériser phonétiquement de façon un peu grossière chacun des segments repérés.
- Une tâche d'évaluation dont l'objectif est l'attribution à chaque segment d'une description quantitative ou d'un certain score.

Les deux premières tâches s'effectuent conjointement en utilisant un ensemble de plus ou moins 35 règles intégrées dans un programme. Ces règles procèdent par raffinements successifs en réalisant tout d'abord une segmentation et un étiquetage très grossiers; les règles suivantes travaillent sur les résultats fournis par les prècédentes en modifiant éventuellement les limites des segments, en en créant éventuellement de nouveaux ou en attribuant de nouvelles valeur aux étiquettes.

Le résultat de cette segmentation et de cet étiquettage constitue un treillis phonétique. Certains segments n'ayant pu être identifiés de manière univoque, il en résulte la possibilité de plusieurs suites de segments possibles correspondant au signal.

A chaque étiquette de segment, il est attribué un ratio de vraisemblance. Ces ratios sont calculés durant la troisième tâche effectuée par le module : la tâche d'évaluation. De même que pour l'étiquettage des segments, le système procède pour le calcul de ces ratios par raffinements successifs.

3.1.2.3 composant de recherche lexicale. (LR)

Ce composant intègre les sources de connaissances lexicales et phonologiques. C'est lui qui va permettre la comparaison entre les mots du dictionnaire du système et le

contenu du treillis de phonèmes.

Le dictionnaire utilisé par LR possède une représentation un peu paticulière pour des motifs d'efficacité dans les recherches. En effet, iln'est pas acceptable d'avoir à considérer tous les mots précédant un mot donné avant de pouvoir trouver ce dernier. C'est pourquoi une structure d'arbre a été choisie pour rendre le plus optimales possibles les recherches menées au sein de ce dictionnaire.

De plus, comme il a déjà été signalé dans les chapitres précédents, la prononciation d'un mot s'altère plus ou moins au cours d'un discours continu. Les mutations pouvant ainsi s'opérer sont décrites par un certain nombre de règles. Ces règles sont utilisées lors du chargement du dictionnaire afin d'intégrer à l'arbre les mutations pouvant intervenir.

Ce composant permet donc la recherche d'un mot dans le dictionnaire, mais il permet également des recherches parmi différents sous-ensembles de celui-ci.

3.1.2.4 Composant de vérification. (VR)

La méthode incrémentale qu'ont suivi les concepteurs de HWIM pour mener à bien sa réalisation a révélé l'intérêt de pouvoir vérifier à tout moment l'hypothèse de la présence d'un certain mot dans l'enoncé. Cette vérification doit s'opérer sur le treillis phonétique.

Une analyse de bas en haut du treillis phonétique s'avéra être particulièrement délicate du fait, une fois encore, des phénomènes de coarticulation ainsi que du caractère génératif des transformations phonologiques pouvant intervenir.

Par contre une analyse du haut vers le bas s'avera, elle, être plus aisée puisqu'elle pouvait intégrer certaines informations contextuelles.

Ces deux raisons ont justifié l'adjonction au système de ce composant permettant la vérification ou la plausibilité de présence d'un mot donné au sein de l'énoncé traité.

3.1.2.5 Le composant syntaxique. (SYN)

Ce composant intègre différents types de connaissances. Des connaissances syntaxiques, des connaissances sématiques et des connaissances pragmatiques., c'est à dire plus spécifiques au domaine d'application.

L'analyseur syntaxique est construit autour d'un réseau de transitions augmenté (ATN). Ce réseau est dit pragmatique car il englobe non seulement les connaissances syntaxiques et sémantiques, mais il contient également des connaissances pragmatiques. Celui-ci n'accepte que des énoncés corrects syntaxiquement, significatifs dans le domaine de la gestion de budgets de voyages et qui, de plus, devront être adressé à la machine. Ceci afin que la machine ne doivent pas pouvoir comprendre des énoncés entre deux locuteurs externes. Il est exigé de plus que les énoncés soient en rapport avec ce qui a été dit auparavant.

La grammaire générale fournie par les ATN a été restreinte à des constituants liés à la tâche traitée. Ceci afin d' augmenter la capacité prédicative des réseau en permettant le rejet de phrases non appropriées, bien que syntaxiquenment correctes et composées de mots du vocabulaire de la tâche.

Les trois fonctions de l'analyseur sont

- juger de la qualité grammaticale d'une séquence de mots
- prédire les extensions possibles d'une séquence de mots proposée.
- construire une représentation sémantique formelle de l'énoncé.

Une remarque à propos de l'analyseur syntaxique est sa capacité de démarrer son travail à n'importe quel endroit de l'énoncé et de poursuivre celle-ci dans les deux sens (vers la droite et/ou vers la gauche) à partir de là. On retiendra également que dans ce système, c'est ce composant-ci qui réalise l'interprétation de l'énoncé.

3.1.2.6 Le composant d'accès à la base de données. (TRIP)

Ce composant joue le rôle de composant central du système, c'est lui qui réalise les accès à la base de données, c'est lui qui intègre les sources de connaissances

relatives au discours et aux faits.

Comme cela a été dit précédemment, c'est au moyen d' un réseau sémantique que sont modélisés les énoncés à représenter. Lors d'une requête, TRIP appelle le composant de stratégie de contrôle en lui fournissant certaines hypothèses sur l'énoncé émis; en retour il reçoit ce qui constitue en même temps une représentation et une interprétation sémantique de l' énoncé.

Cette interprétation est ensuite manipulée par TRIP de façon à en extraire l'information qui permettra la satisfaction de la demande du locuteur.

3.2 Les stratégies de contrôle.

Les stratégies de contôle régissant le processus de création des hypothèses et de leur extension ont été directement dérivées de la méthode incrémentale évoquée plus haut. Dans HWIM, le composant de contrôle utilise les autres composants un peu comme des sous-routines dans la mesure ou c'est lui qui prend les décisions relatives aux différentes hypothèses. Décisions par les quelles les ressources sont allouées à telle hypothèse plutôt qu'à telle autre. Différentes stratégies de contrôle ont été implémantées, chacune correspondant à des méthodes diiférentes d'énumération des choix des évènements de base, de raffinement des hypothèses ou à des fonctions de priorité dont l'objectif était de déterminer quel événement prendre en compte à un instant donné. Néanmoins globalement, on peut distinguer une certaine constance dans ces différentes stratégies mises au point.

[1] Lorsque l'énoncé a été saisi et paramétré, que le treillis de phonèmes a été constitué, le composant de recherche lexicale est utilisé afin de tenter de localiser dans tout ou partie du treillis certains mots qui matcheraient bien avec leur modèle de référence, et ce de façon indépendante du contexte. Chacun des mots ainsi découvert constitue un événement qui devra être exploité par la suite et qui est placé à cet effet dans une file des événements à traiter. Les événements y sont classés selon leur score(a).

[2] Ensuite, l'evénement de plus haut score est sélectionné et une hypothèse en est tirée pour être présentée en tant qu'une théorie à l'analyseur syntaxique.

(a) Le lecteur soucieux d'obtenir plus de détails sur les différentes techniques de scoring mises au point et utilisées dans HWIM se référera à [lea 80].

[a] Si cette théorie est complète, c-à-d qu'elle couvre l'ensemble de l'énoncé, alors l'analyseur retourne l'interprétation sémantique de cet énoncé. Cette interprétation est considérée comme la meilleure meilleure possible et en principe le système pourrait continuer sa recherche et trouver une autre interprétation, puis une troisième etc, chacune d'elle moins "bonne" que la précédente.

[b] Si cette théorie est grammaticallement innacceptable, alors elle est tout simplement rejetée et l'étape 2 est répétée.

[c] Dans tous les autres cas, c-à-d lorsque les théories sont incomplètes, mais grammaticallement correctes, l'analyseur fait un ensemble de propositions pour chaque mot ou chaque catégorie de mots qui pourrait se trouver à chaque extrémité de la théorie proposée.

[3] Les différentes propositions sont alors fournies au module de recherche lexicale pour qu'il effectue des recherches sur base de celles-ci, à partir de chacune des extrémités de la théorie. Sont intégrées aux recherhes les altérations phonologiques possibles dues au voisinage de l'un ou l'autre mot se trouvant à l'une des extrémité de la théorie. Pour chacun des mots ainsi trouvé, un nouvel événement est créé et rangé dans la file des événements à une place déterminée par le score dont celui-ci est gratifié. Ce nouvel événement constitue une extension de la théorie qui en est à la base.

[4] Vient ensuite une phase de maintenance de la file des événements au cours de laquelle certains d'entre eux seront supprimés, d'autres seront ajoutés ou encore, d'autres verront leur score recalculé.

[a] Si un mot qui vient d'être ajouté à une extrémité d'une théorie se retrouve également à une extrémité opposée d'une autre théorie, on dit alors qu'il y a collision d'événements. Celle-ci donne lieu à la création d'un nouvel événement issu de la réunion des deux autres qui à son tour est rangé dans la file. Les deux événements en collision sont retirés de la file.

[b] Si le système se rend compte à ce moment que l'une des théories a probablement atteint une extrémité de l'énoncé,un événement de fin d'énoncé est créé qui signale l'hypothèse faite de la fin de l'énoncé en cet endroit. Un score de vraissemblance est bien sûr ajouté à cet événement.

[5] A moins qu'une des limites des ressources ne soit atteinte, auquel cas le système s'arrêtera après avoir signalé l' absence de toute interprétation, il reprendra à l'etape [2].

3.2.1 Evaluation de la stratégie de contrôle.

Plusieurs des stratégies mises au point dans le cadre de ce système étaient des stratégies dites admissibles(a). Il est possible de démontrer formellement que, dans HWIM, l'interprétation de l'énoncé qui va être trouvée sera la meilleure possible sans pour autant avoir dû rechercher préalablement toutes les autres interprétations possibles. Mais ceci dépasse le cadre de notre exposé et devrait plutôt s'inscrire dans le cadre d'une recherche approffondie sur les techniques de scoring.

Néanmoins, pour des raisons de limitations de temps et de ressources, le prototype final de HWIM a été testé en utilisant une stratégie approximative. De plus, lors de la mise en commun de tous les composants, ceux-ci se situaient bien au dessous des performances que leur prétaient leurs concepteurs. Ils avaient en effet subi déjà grand nombre d'améliorations au cours des derniers mois du projet et d'autres améliorations importantes se profilaient encore à l'horizon. Malheureusement ces dernières améliorations ne purent être intégrées en temps utile au système. En particulier le rôle de la prosodie n'a pas pu être testé ni, a fortiori, raffiné.

Ceci a eu pour conséquence de ne pas combler tous les espoirs qui avaient été fondés sur lui. Néanmoins, les recherches menées ont permis de mettre à jour des outils et des techniques performants qui pourront être réutilisés car ils constituent des progrès indéniables dans les technologies de traitement de la parole par ordinateur.

Ce sont principalement

[1] la méthode générale de scoring permettant de combiner des scores issus de différentes sources de connaissances.

(a) Le concept de stratégie admissible s'oppose à celui de stratégie approximative. Celles du premier type garantissent la découverte d'une solution optimale même en l'absence d'un parcours exhaustif de l'ensemble de l'espace des solutions possibles. Par contre les stratégies dites approximatives ne garantissent pas que la solutions trouvée sera toujours la meilleure.

[2] Une technique efficace permettant d'incorporer des règles phonologiques génératives des altérations potentielles dans le dictionnaire du vocabulaire du système.

[3] l'utilisation d'un ATN pragmatique permettant de combiner des sources de connaissances diversifiées; essentiellement des connaissances syntaxiques, sémantiques, pragmatiques et prosodiques.

[4] un algorithme d'analyse syntaxique pouvant fonctionner dans les deux sens, tant de la gauche vers la droite que de la droite vers la gauche.

3.3 Le système Myrtille I.

3.3.1 Introduction.

Le système Myrtille I issu du Centre de Recherche en Informatique de Nancy (CRIN) a été l'un des premiers système tentant d'utiliser les apports potentiels d'une information contextuelle en reconnaissance et en compréhension de la parole. cette volonté amena son concepteur, J.M. Pierrel, à réaliser dans un premier temps un système qui avait pour but de traiter un langage dit artificiel, répondant à la description qui en a été faite précédemment.

3.3.2 L'architecture du système.

Les différents modules qui entrent en action dans ce système sont :

- un module de traitement acoustique.
- un module de segmentation et d'identification de la représentation acoustique du signal. En sortie de ce module, on trouvera une pseudo chaine de phonèmes représentant l'énoncé.
- un analyseur lexicographique dont le but est la reconnaissance de mots par comparaison de chaines phonétiques de références avec des pseudo chaines fournies par le module de segmentation et d'identification.
- Un module d'analyse syntaxique chargé de construire une représentation syntaxique de l'énoncé. Son rôle est également d'émettre des hypothèses quand aux mots - ou familles de mots - contenus dans l'énoncé.
- un module de gestion des hypothèses validées.
- un module de dialogue dont l'objectif est soit de déclencher l'action qui satisfera la demande du locuteur, soit de déclencher une demande de

confirmation relative à une partie de l'énoncé du locuteur, soit, enfin, de déclencher une demande de répétition de son énoncé par le locuteur.

- un module de synthèse de la parole permettant au système de générer, de façon audible pour le locuteur, l'énoncé fourni en sortie par le module dialogue.

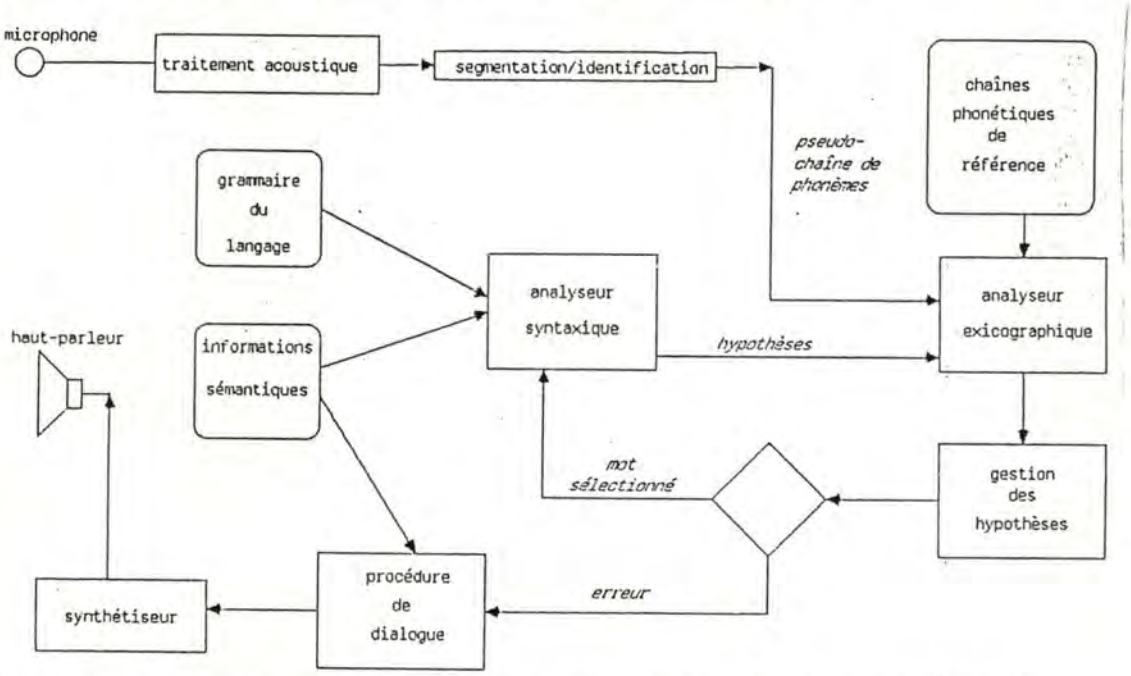


Figure 3.2: Schéma de l'architechture générale de Myrtille I.

Nous ne nous attarderons ici que sur les modules d'analyse syntaxique et d'analyse lexicographique car ils représentent à la fois les modules principaux et les plus intéressants du système.

3.3.2.1 L'analyseur syntaxique.

La réalisation du projet Myrtille I a été guidée par différents objectifs dont l'un était la volonté de mettre en évidence le rôle important d'un composant syntaxique pour la reconnaissance de la parole continue. D'où l'idée d'un système exploitant une stratégie descendante. L'analyse syntaxique y jouerait en quelque sorte le rôle d'un meneur émettant des hypothèses sur les mots à reconnaitre en tenant compte du contexte syntaxique déjà reconnu. L'algorithme d'analyse syntaxique fonctionnant sur le système Myrtille I est dérivé des algorithmes classiques qui fonctionnent avec lecture en avant d'un mot. Néanmoins, les caractéristiques du signal de la parole sont telles qu'il existe un important indéterminisme quant aux terminaux composant l'énoncé. Il était donc nécessaire d'adapter ces algorithmes de façon à les rendre applicables à ce contexte indéterministe.

Les algorithmes classiques fonctionnent selon le mode suivant : quand ils se trouvent en un point donné de l'énoncé, ils effectuent le choix d'une règle exprimant la syntaxe admissible afin de partitionner l'énoncé en un ensemble de parties sur lesquelles sera appliquée une autre règle et ainsi de suite. L'originalité de l'analyseur du système Myrtille I consiste en le remplacement du choix des règles par l'émission d'hypothèses sur les différents terminaux possibles. Le traitement de l'indéterminisme dû à la continuité de la parole est donc postposé. C'est le module d'analyse lexicographique qui se chargera de celui-ci. Ce dernier ayant choisi l'une des hypothèses lui ayant été fournies par le module syntaxique renverra celle-ci afin qu'elle serve de point d'ancrage et de point de reprise pour la suite de l'analyse.

3.3.2.2 L'analyseur lexicographique.

Le but de l'analyseur lexicographique est le test et la validation des hypothèses fournies par l'analyseur syntaxique. Celles-ci sont traitées à ce niveau puis rangées dans une pile, la pile des hypothéses, selon un ordre fonction des scores de reconnaissance obtenus par chacune d'elles. L'hypothèse ayant le meilleur score se trouve donc au sommet de la pile à l'issue du traitement de l'analyseur lexicographique. C'est cette dernière qui sera prise en considération pour la poursuite de l'analyse de l'analyse syntaxique. Si aucune des hypothèses n'a pu être validée, le sommet de la pile correspond alors à une hypothèse qui y avait été placée précédemment. La prise en compte de cette nouvelle hypothèse par la syntaxe provoque un retour arrière dans l'analyse. Il se peut qu'à un moment la pile soit vide, ce qui correspond à un échec du processus de reconnaissance.

L'algorithme d'analyse lexicographique fonctionne selon trois phases successives.

- la tentative de reconnaissance des terminaux donnés comme hypothèses par le module syntaxiques. Cette reconnaissance est effectuée grâce à une procédure spécifique qui procède par comparaison des chaines phonétiques du treillis (celles qui n'ont pas encore été explorées par l'analyseur) et de celles correspondant aux mots hypothétisés dont les représentations phoétiques sont connues du système.
- Les terminaux (ou groupe de terminaux) qui ont été reconnus sont ensuite triés selon leur score de reconnaissance. Ce score est fourni par la procédure de

reconnaissance évoquée en [1].

- La liste des terminaux (ou groupes de terminaux) triés est ensuite rangées dans la pile des hypothèses validées, ce qui permettra à l'analyseur syntaxique des les prendre en compte par la suite.

3.3.3 Apports du système Myrtille I.

Outre son fonctionnement propre, le système Myrtille I a le mérite d'avoir permi la validation d'une séries d'idées qui ont pu servir de support pour des recherches et des systèmes ultérieurs, notamment le système Myrtille II. En voici les principales :

- mise en exergue de l'importance et du rôle des informations contextuelles dans un processus de reconnaissance du type Hypothèses et Tests. Dans le système Myrtille I, les informations contextuelles sont de type syntaxiques, elles ont pour effet de réduire le plus possible le nombre de mots à tester.

- Mise au point de stratégies heuristiques permettant d'optimaliser les recherches. En particulier la stratégie consistant à exploiter en premier lieu les hypothèses gratifiées du meilleur score de reconnaissance.

- Le module de dialogue, sur lequel nous n'avons pas insisté ici fut l'un des premiers du genre.

Ce système s'adapte trés bien aux langages de type artficiels dont la syntaxe est fortement contrainte et qui ne permet que peu de possibilité d'élocution tant son facteur(a) de branchement est faible (au plus quelques dizaines de mots). Néanmois, comme nous l'avons déjà mentionné précédemment ce type de langage offre de bonnes possibilités dans divers domaines.

(a) Nous rappelons au passage que le système Hwim présentait lui un facteur de branchement assez élevé, de l'ordre de 133 mots.

3.4 Le sytème MYRTILLE II [Pierrel 81,Pierrel 82].

Le système Myrtille II est la suite du projet Myrtille I présenté dans la section précédente.

3.4.1 Objectifs du système.

Le but du système Myrtille II est de reconnaitre et de comprendre des phrases d'un langage pseudo-naturel dont les caractéristiques essentielles sont :

- une syntaxe décrivant un sous-ensemble assez vaste du francais et suffisamment stable que pour être valable pour toute application dy type centre de renseignements ;
- un vocabulaire de plus de 350 mots ;
- une sémantique et une pragmatique restreinte à un univers particulier.

Il s'agit d'un système paramétrable constituant un outil de recherche en vue de tester l'apport de divers types d'informations et la mise en oeuvre de différentes stratégies pour aboutir au résultat souhaité.

Ce résultat souhaité correspond à une représentation syntaxico-sémantique de l'énoncé oral traité. Il est obtenu dans une optique plutôt de compréhension que de reconnaissance, ce qui explique que certains résultats, bien qu'incomplets peuvent néanmoins être admis.
D'une manière tout à fait générale, il doit comporter toutes les informations nécessaires pour permettre la synthèse de l'énoncé reconnu, et son interpretation en vue du déclenchement de l'action ou de l'évaluation de la réponse correspondante.

Pour obtenir ce résultat, Myrtille II se base sur 2 types de données : la définition du langage (définition des structures du langage, déffinition syntaxico-sémantique des mots du lexique, description phonétique et phonologique des mots), et la représentation acoustico-phonétique de l'énoncé (treillis de phonèmes, diverses informations prosodiques).

Enfin, l'application-test est un centre de renseignements meteorologiques.

3.4.2 Les diverses informations prises en compte.

Les informations prises en compte dans MYRTILLE II sont les informations classiques (acoustiques, phonétiques, phonologiques, prosodiques, lexicales, syntaxiques, sémantiques, et pragmatiques), classées suivant qu'il s'agit :

- d'informations liées à la structure du langage, c'est-à-dire des informations essentiellement syntaxiques, mais aussi lexicales, prosodiques et sémantiques ;

- d'informations liées à l'application, principalement de type lexicales et pragmatiques, mais aussi syntaxiques (liste des fonctions syntaxiques possibles des mots dans le cadre restreint de l'application), sémantiques (liaisons sémantiques possibles entre les différents mots, toujours dans le cadre de l'application visée), et acoustico-phonétiques (représentation des mots, règles d'altérations possibles des différents mots, etc...) ;

- d'informations liées au locuteur, essentiellement phonologiques (rendant compte des liaisons propres au locuteur) et prosodiques (sur les mots et les syntagmes).

MYRTILLE II respecte cette classification et constitue ainsi un système paramétrable à 3 niveaux :

- le premier type d'informations devrait être unique ;

- le second est à redéfinir pour toute application ;

- le troisième est également à redéfinir pour toute classe de locuteurs.

Ces informations sont utilisées par 3 sources d'informations (STRUCTURE, LEXIQUE, PROPHON), dont on reparlera plus loin, et qui sont organisée de façon pseudo-parallèle : la détermination de la hiérarchie est réalisée dynamiquement par un superviseur en fonction de critères eux-mêmes hiérarchisés.

3.4.3 Modèles de représentation des informations linguistiques.

3.4.3.1 Les réseaux à noeuds procéduraux.

Déjà évoqués lors de l'exposé des principes généraux, l'objectif des réseaux à noeuds procéduraux (RNP) est de décrire une structure de langage, sous-ensemble assez vaste du francais, et :

- indépendante de l'application ;
- porteuse d'information sur la position relative entre les mots ;
- permettant, en cours de reconnaissance, de construire un arbre syntaxico-sémantique de même type que ceux fournis par une grammaire hors-contexte ou des ATN ;
- et adaptée au traitement de la parole (gérant le mieux possible l'indéterminisme, et intégrant des traitements particuliers pour les mots de liaison sur lesquels l'analyse ne peut pas trop s'appuyer).

Le modèle se présente comme un réseau de transition, représentant une grammaire hors-contexte à terminaux de type classes syntaxiques, ainsi que des procédures associées à chaque noeud (noeuds procéduraux).
Le but de ces procédures, qui sont par ailleurs pour la plupart indépendantes de l'application choisie, est de déterminer une hiérarchie des sorties possibles du noeud en se basant sur les entrées et sur un certain nombre de tests prosodiques, phonétiques, et syntaxico-sémantiques portant sur les parties d'énoncé déjà reconnues. Elles permettent donc d'implémenter un certain nombre de restrictions locales tout en restant dans le cadre des grammaires hors-contexte.

Bien qu'assez proches des ATN de Woods, les RNP s'en distinguent principalement par le rôle assigné aux procédures.
Dans les ATN, elles gèrent la représentation syntaxico-sémantique de la phrase. Dans les RNP, ce rôle est réservé à un module d'analyse extérieur au réseau, les noeuds procéduraux ayant pour fonction de trier les sorties possibles.

3.4.3.2 Définition syntaxico sémantique des mots : le lexique.

Afin de garder la distinction entre l'aspect définition du langage et les aspects propres à l'application et à la parole d'une part, et, d'un point de vue plus opératoire, de permettre la réalisation des 3 grands types de traitement (restriction des hypothèses émises grace à l'analyse du réseau syntaxico-sémantique, reconnaissance et compréhension

de phrases non grammaticales, mise à disposition des informations nécessaires à la reconnaissance phonétique des mots) d'autre part, le lexique se présente comme une structure à double entrée reprenant des informations de type syntaxiques (restrictions syntaxiques sur l'usage des mots dans le cadre d'une application donnée, précisions sur les constructions permises autour d'un mot, ...), de type syntaxico-sémantique (liaisons syntaxico-sémantiques possibles entre des mots, par exemple les sujets possibles d'un tel verbe), et enfin de type phonétiques et phonologiques.

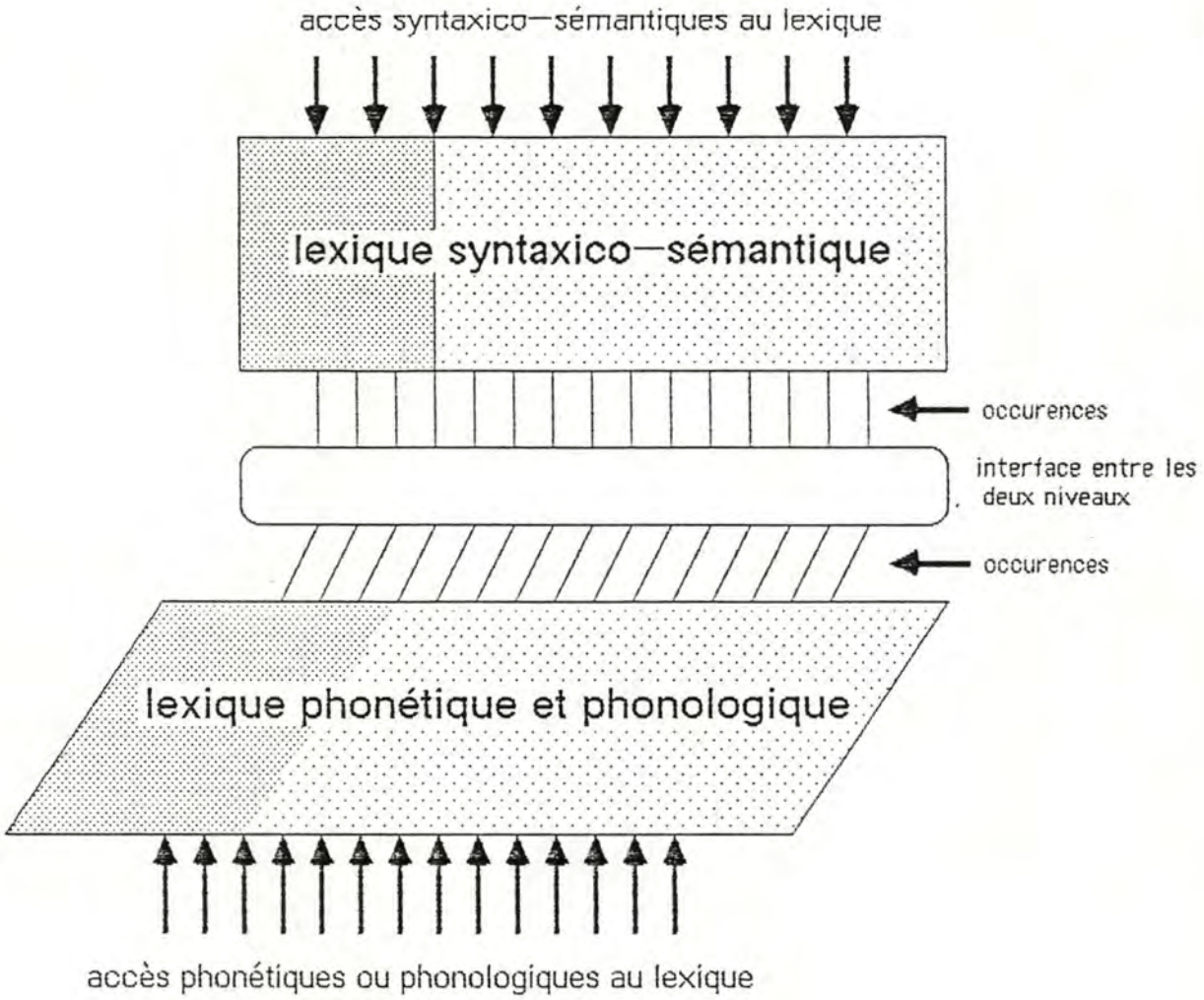


Figure 3.3: organisation du lexique.

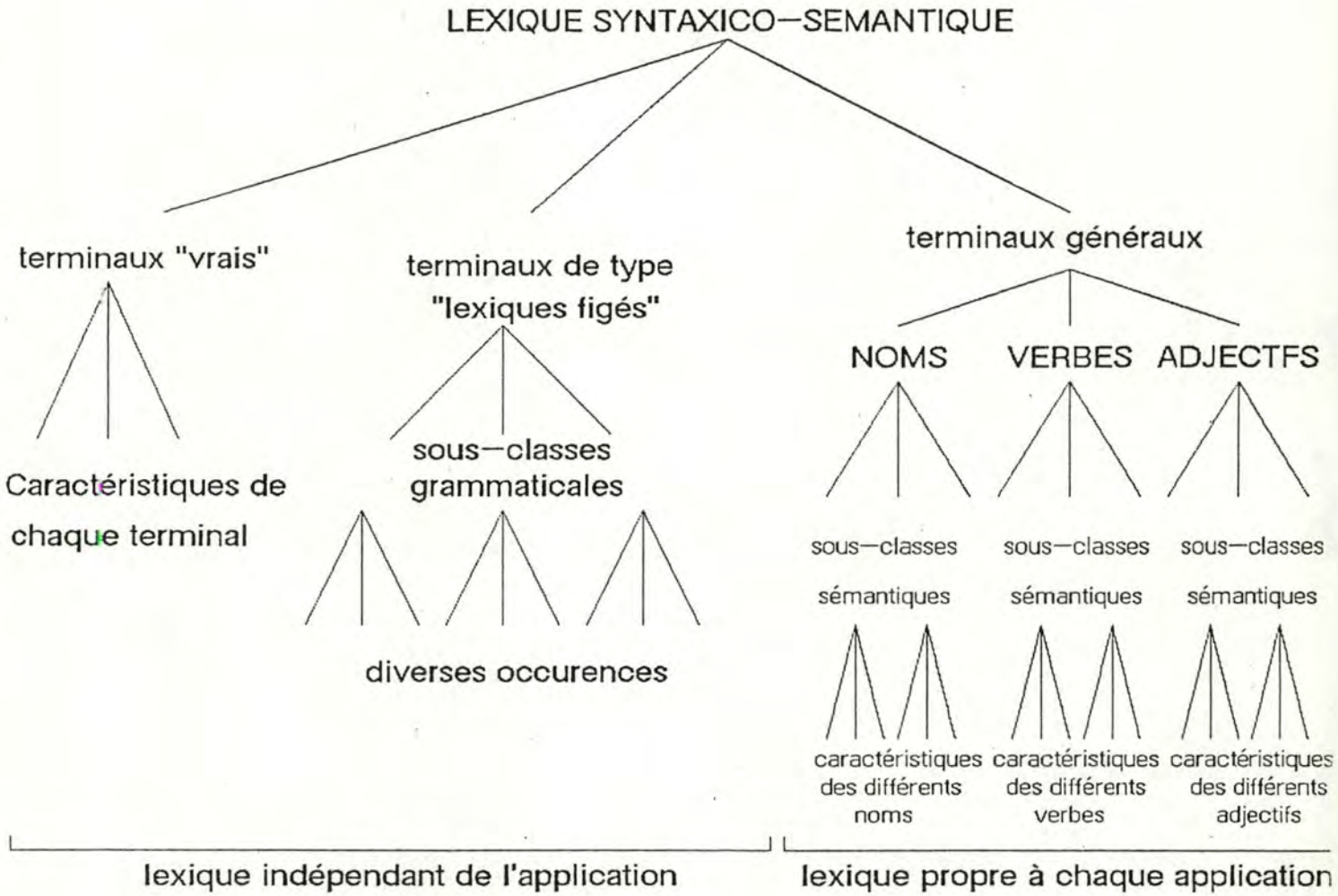


Figure 3.4: organisation hiérarchique du lexique syntaxico-sémantique.

Le lexique syntaxico-sémantique est structure de manière hiérarchique. A un premier niveau, il distingue les mots grammaticaux des mots liés à l'application. Ces derniers sont regroupés par sous-classes sémantiques réparties par classes grammaticales.

Les procédures attachées au lexique sont du type :

- sélection des adjectifs pouvant qualifier un nom ;
- sélection des différents noms pouvant être sujets d'un verbe ;
- etc...

L'experience a montré qu'un tel lexique permet effectivement de restreindre considérablement les hypothèses émises à chaque pas.

3.4.4 Fonctionnement général du système.

Le système suivant un schéma d' "hypothèse-test" combinant une méthode ascendante (s'appuyant sur les niveaux linguistiques) et descendante (partant du signal vocal et de sa transcription phonétique).
La compréhension d'une phrase revient à une recherche de solution dans un espace de réalisations partielles correspondant à l'ensemble des phrases et sous-phrases possibles compte tenu des 3 contraintes (structure du langage, lexique, treillis phonétique d'entrée).
La stratégie utilisée est essentiellement de type "meilleur d'abord", en se rapprochant dans certains cas d'une stratégie " par ilôts de confiance" pour éviter des résultats trop peu nuancés du genre "échec" ou "bonne reconnaissance".

* Au début, l'ensemble des hypothèses correspond à l'ensemble des mots du lexique.
* La prise en compte d'une source d'informations permet de restreindre ces hypothèses uniquement aux mots compatibles avec elle.
  - STRUCTURE restreint cet ensemble en fonction de la définition du langage et du contexte de la phrase lors du parcours pas à pas des RNP ;
  - LEXIQUE s'appuye, lui, sur la définition syntaxico-sémantique des mots pour éliminer celles non compatibles avec les dépendances conceptuelles acceptées par la partie de l'énoncé déjà traité et

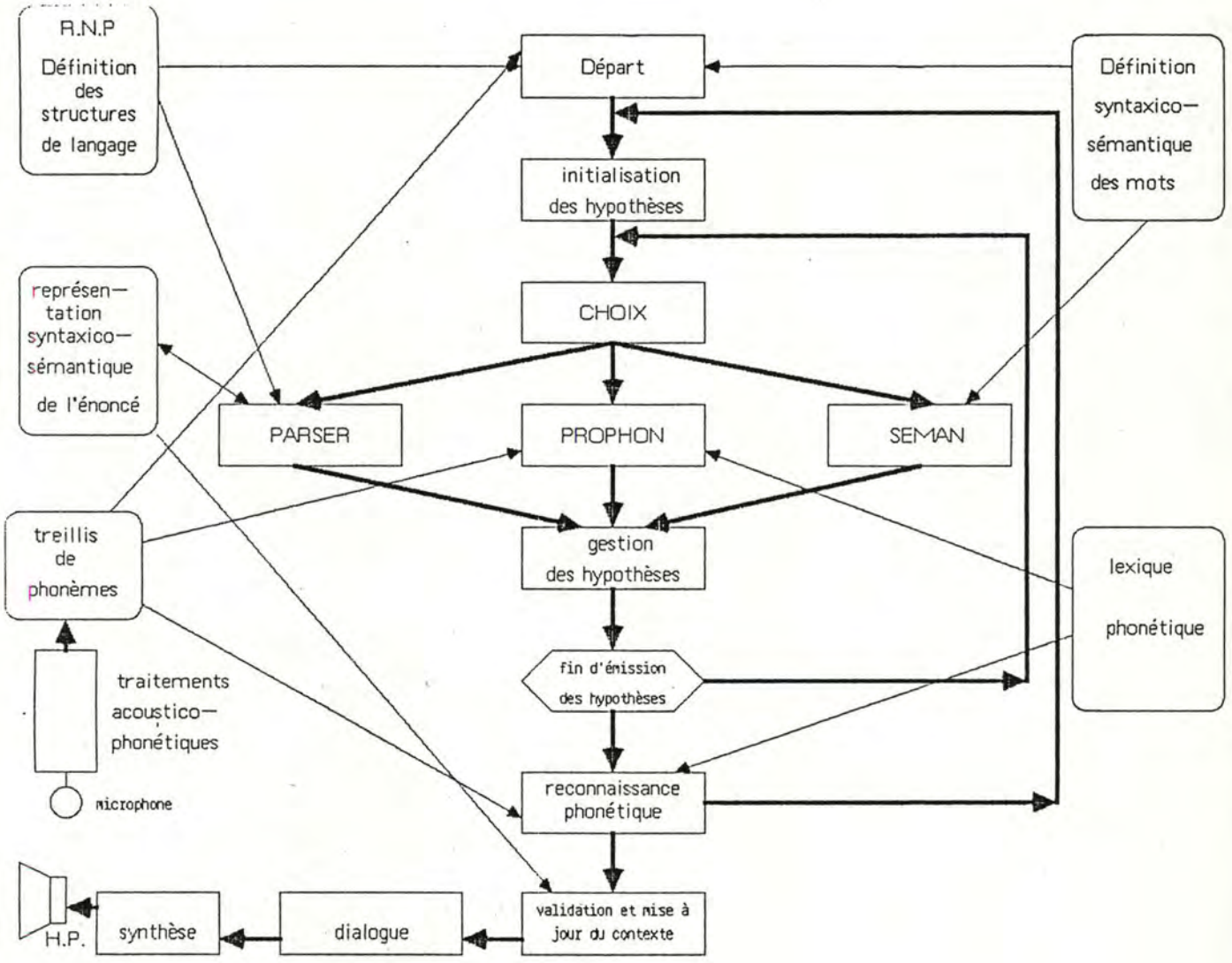


Figure 3.5: fonctionnement général de Myrtille II.

la structure reconnue ;

- PROPHON, enfin, pondère les hypothèses retenues sur base des traits prosodiques et la structure phonémique du contexte à reconnaitre.

* La reconnaissance phonétique correspond à l'étape de test des hypothèses (comparaison de la

représentation phonétique de référence avec la partie du treillis en entrée).

* VALIDATION a pour fonction de choisir une hypothèse sur base des résultats de reconnaissance, complète la représentation syntaxico-sémantique de la partie de l'énoncé qui a déjà été traitée, et détermine le point de reprise du processus de comphréhension en fonction de la stratégie de recherche adoptée (meilleur d'abord).

* Enfin, la procédure de dialogue a pour fonction d'exploiter les résultats obtenus afin de réaliser l'interpretation sémantique de l'énoncé oral.

3.4.5 Evaluation.

Les résultats des tests réalisés ont permis de montrer avant tout la validité et la robustesse des réseaux à noeuds procéduraux comme modèle de définition d'un langage pseudo-naturel.

De l'avis même des concepteurs, il faut cependant souligner [Carbonell 85-a]:

- la non prise en compte d'informations pragmatiques lors de la phase d'émission des hypothèses ;

- le manque d'interaction entre le décodage phonétique et les autres niveaux de traitement ;

- et enfin l'absence de dialogue permettant une véritable communication homme-machine.

Entre autres choses, ces quelques remarques sont à la base du projet de développement du système de gestion de dialogues oraux finalisés pour lequel nous avons été amenés à travailler lors de notre stage à Nancy. Avant d'en effectuer une présentation détaillée, il faut, nous semble-t-il, s'attarder quelque peu sur un certain nombre de considérations un peu plus générales.
Elles font l'objet du chapitre suivant.

CHAPITRE II : PROBLEMES LIES A LA CONCEPTION D'UN SYSTEME DE DIALOGUE ORAL HOMME-MACHINE.

1 Spécificité du dialogue oral.

1.1 Rappel de la définition de dialogue.

Nous avons évoqué les différentes fonctions mises en oeuvre dans un dialogue. Par la suite, lorsque nous parlerons de dialogues , il s'agira toujours de dialogues oraux finalisés.
Rappelons la définition de dialogue : Il s'agit de 2 discours qui se développent en interaction l'un vis-à-vis de l'autre.
L'aspect finalisé indique que ces dialogues prennent naissance et se déroulent avec pour objectif la réalisation d'une tâche précise, et sont donc pleinement déterminés par celle-ci.

1.2 Généralités sur le dialogue.

1.2.1 Introduction.

Le passage de la compréhension de phrases isolées à la gestion d'un dialogue finalisé introduit un certain nombre de problèmes nouveaux.
En effet, celle-ci passe nécessairement par la compréhension du dialogue, qui repose lui-même en partie sur la compréhension des énoncés qui le composent. La modélisation d'une sémantique du dialogue se heurte au difficile problème de l'adoption d'une représentation adéquate de ce dialogue et d'une analyse des énoncés vis-à-vis de ce dernier, tandis que la compréhension des énoncés doit pouvoir s'appuyer sur un modèle de langage que l'on veut le naturel et le plus "large" possible.
L'analyse de ces différents impératifs met en évidence la nécessité de la définition, de la modélisation, et de la mise en oeuvre d'un certain nombre de nouvelles connaissances et de nouvelles fonctions.

1.2.2 Compréhension du dialogue.

Un dialogue n'étant finalement constitué que de 2 discours interagissant, il est normal d'aborder le problème de sa compréhension en s'appuyant sur la sémantique du discours.
Si l'interprétation d'énoncés isolés constitue déjà un phénomène complexe et encore mal connu, celle du discours présente encore plus de difficultés.

Volontairement, nous ne rentrerons pas dans une étude approfondie de la sémantique du discours. Elle sort, nous semble-t-il, du cadre de ce mémoire, et de toute façon de celui de notre compétence. Le lecteur intéressé par ce sujet peut consulter entre autres [NEF] [KAMP].
Il est cependant nécessaire de présenter une approche du problème, à savoir celle présentée dans [NEF]. Nous l'avons choisie car le système développé à Nancy s'appuye sur les principes qui la sous-tendent. Elle est basée sur celle de Kamp exposée dans [KAMP 1981, KAMP 1984, KAMP & ABLER 1986].

Un des problèmes clé ici est l'adoption d'une représentation du discours, correspondant à une reconstruction symbolique des représentations mentales que l'on a relativement à ce discours. En effet, elle constitue un des 3 éléments fondamentaux qui rentrent dans le processus de compréhension d'un discours, avec les connaissances attachées aux entités mises en jeu, et celles qui peuvent être impliquées à partir de ce discours par inférence.

Par exemple, la compréhension du discours (élémtaire !)

- "Cet ouvrage est un mémoire"

passe nécessairement par :

[1] la compréhension de la relation existant entre les 2 entités "ouvrage" et "mémoire" sur base d'une représentation de cette relation, et dans le cadre du discours (compréhension dans un contexte d'énonciation);

[2] la compréhension des entités elles-mêmes, dans le contexte de ce discours toujours;

[3] et la compréhension de tout ce que l'on peut en déduire, jusqu'à un certain niveau (implications au niveau du discours lui-même, au niveau des entités mises en jeu, etc...).

Seule, la relation (point [1]) est d'ordre purement discursif. Si, dans l'exemple très simple ci-dessus, la représentation des schémas mentaux correspondant à cette relation ne pose pas de gros problèmes, il n'en va pas de même lorsque le discours est plus long, et qu'il faut non seulement représenter l'énoncé, mais aussi identifier son rôle vis-à-vis de ce discours, et finalement l'intégrer dans sa représentation.

La plupart des auteurs sont d'accord sur l'importance de la dimension dynamique et partielle du discours, et donc de sa représentation.
L'aspect dynamique provient du fait que les représentations discursives ne sont pas fournies d'un bloc par le locuteur,

mais de manière progressive au fil des énoncés, pour enfin aboutir à une représentation complète finale.
L'aspect partiel, quant à lui, repose sur le fait que l'interprétation sémantique ne nécessite pas de postuler l'ensemble des mondes possibles [KRIPKE, voir principes généraux], mais seulement un ensemble de propositions, extraites de celui-ci (ceux-ci), et elles seules étant pertinentes dans le cadre qui nous intéresse.

Le point de départ de cette sémantique du discours est que chaque énoncé comporte des expressions référentielles devant être intégrées dans la structure de représentation discursive (SRD), et des expressions non référentielles précisant la façon de les intégrer.
L'analyse d'un énoncé modifie donc la SRD du discours courant, jusqu'à l'obtention de sa version définitive.

En transposant tout ceci dans le cadre d'un dialogue, les "instructions" constituées par les expressions non référentielles constituent en fait des "instructions" adressées par le locuteur au destinataire l'invitant à construire ou à modifier sa SRD.

ex: il ne fait pas beau, ou plutôt, il fait franchement mauvais.

Les mots "ou plutôt" jouent ici un rôle d'indicateur de correction par rapport à la SRD qui pouvait être déduite de la première partie de l'énoncé.

Ceci est d'autant plus vrai dans le cadre des dialogues coopératifs finalisés, où l'hypothèse de l'existence de ce mécanisme de construction/révision des SRD prend une signification plus particulière encore.

Dans ce modèle, la gestion et la compréhension d'un dialogue finalisé passe donc, pour la réalisation de son objectif premier de résolution de la tâche visée, par l'établissement d'une SRD "commune" aux 2 parties, et pouvant être interprétée de manière unique et précise.
Elle devra être constituée d'informations sur le dialogue en cours, sur l'utilisateur, et sur ses intentions vis-à-vis de la tâche à accomplir.
Toute ambiguité nécessitera un nouvel échange et une adaptation réciproque des SRD.

Des fonctions de raisonnement et de gestion de ce dialogue s'avèrent également nécessaires, la première pour assurer un certain nombre de déduction à partir des énoncés en vue d'identifier la "requête" et de s'assurer que l'on possède tous les éléments pour y répondre, la seconde pour mener le dialogue à son terme en fonction des résultats de la première.

Enfin, ce modèle présuppose encore l'établissement d'une représentation sémantique de chaque énoncé.
Cette étape est a priori plus complexe que pour la compréhension de phrases isolées dont nous avons parlé jusqu'à présent, si l'on tient compte du type d'énoncés auxquels on peut s'attendre dans un dialogue que l'on désire aussi naturel que possible, et des inévitables références à la partie antérieure de ce dialogue.
Elle nécessite donc le développement d'un outil de compréhension de phrases du type de ceux présentés au chapitre précédent, ainsi que d'un outil d'interprétation visant à relier sémantiquement enoncés, tâche, et contexte local au dialogue.
Nous reviendrons longuement sur ce dernier point, qui constitue un des deux domaines que nous avons approfondi.

Avant d'aller plus loin, il faut se demander si la volonté de voir le dialogue se dérouler dans un langage aussi naturel que possible n'impose pas, pour un dialogue finalisé toujours, la définition et la modélisation du langage tout à fait général.
Cette question est fondamentale au niveau des structures de langages à définir et du champ sémantique à envisager.

1.2.3 Les langages spécialisés ou opératifs.

Plusieurs études menées par des linguistes et des ergonomes ont clairement indiqué que "les mêmes individus parleront très différemment selon qu'ils sont en train de discuter un problème technique au travail, de jouer à un jeu, ou de boire de la bière" [ARGYLE].
Ces formes de langage qui se développent dans des situations bien spécifiques sont fonction d'un certain nombre de facteurs dont le mode de communication utilisé, le caractère formel ou non de la situation, etc...

Les effets de la tâche dans laquelle les interlocuteurs sont impliqués sont, eux aussi, fort importants. Le fait qu'un certain nombre de choses soient partagés par les deux parties permet d'alléger assez fort le discours en rendant un maximum de choses implicites [FALZON].

Plusieurs définitions de ces sous-langages existent, sans qu'aucune ne fasse tout à fait l'hunanimité. Moskovitch ( ), parlant de cette notion, distingue les trois points de vue suivants : certains considèrent qu'il ne s'agit pas de sous-langages à proprement parlé, mais plutôt de style de langue ; d'autres les définissent à partir des lexiques qui leur sont spécifiques (sous-langages lexicaux) ; et d'autres enfin admettent leur existence en tant que style fonctionnel d'une part et comme langage propre, en tant que système sémiotique spécifique d'autre part.

Quelque soit la position que l'on adopte et la relation que l'on admet entre ces sous-langages et le langage "général" (qu'il faudrait pour bien faire commencer par définir...), il est indéniable que tout dialogue oral se déroulant dans le cadre d'une tâche donnée (et donc a fortiori les dialogues finalisés qui nous intéressent ici) s'effectue au moyen d'un langage plus ou moins spécialisé - "opératif" pour citer Falzon [FALZON op. cit.] - possédant un certain nombre de caractéristiques spécifiques.

[1] D'abord, différentes études ont montrés que ces langages apparaissent de façon spontanée et progressive [FALZON, op. cit.].

[2] Ils s'établissent en adaptation avec les connaissances (réelles ou supposées) de l'interlocuteur. Suivant que celui-ci est expert ou novice, le langage sera différent [FALZON op. cit.].

[3] Ils sont quelque peu restreints d'un point de vue lexical. En effet, d'une part, le lexique utilisé est lui-même restreint, et d'autre part, les mots sont caractérisés par la monosémie [FALZON op. cit., GUILBERT]. La tâche joue donc un rôle de filtre tant au niveau du nombre de mots utilisés qu'au niveau des différentes significations que l'on peut leur attribuer.

[4] Ils sont également caractérisés par un certain nombre de restrictions syntaxiques : conventionnalité des formes d'expression vis-à-vis du contexte d'énonciation [GIBBS], certaines d'entres elles devenant donc privilégiées, et structures de langage à la fois restreintes et spécifiques par rapport au langage général [KITTREDGE 1979].

[5] Enfin, leur raison d'être implique nécessairement des restrictions d'ordre pragmatique, en ce sens qu'ils ne sont utilisables qu'à l'intérieur de leur contexte, et que toute situation qui en sort nécessite un retour à l'utilisation du langage général [AMALBERTI 1985].

Ces caractéristiques, valables pour les langages opératifs en général, le sont aussi pour les dialogues opératifs qui en forment un sous-ensemble.

Pour ces derniers cependant, on observe également :

[1] qu'ils épousent la structure de la tâche à accomplir [DEUTSCH 1984], ce qui signifie, par analogie, qu'étudier l'histoire structurée d'un tel dialogue revient à étudier un schéma de résolution de la tâche ;

[2] que les actes de dialogue sont soit orientés par le but à atteindre, et donc porteurs d'informations

pertinentes par rapport à la tâche à réaliser, soit qu'ils correspondent à des actes de contrôle de dialogue. Ils peuvent, les uns et les autres, fournir ou demander de l'information.

La conclusion de tout ceci est que la conception d'un système de gestion de dialogue oral finalisé en langage aussi naturel que possible ne postule pas nécessairement la modélisation du langage "général".
Elle peut s'appuyer plutôt sur celle du langage spécialisé lié à la tâche à accomplir ainsi que du locuteur type (un tel langage étant fonction de l'interlocuteur), et dont les caractéristiques peuvent être extraites à partir d'une analyse détaillée de corpus reprenant les résultats d'un certain nombre de simulations.

Enfin, une modélisation de la tâche à accomplir et de l'univers de l'application, constituant ensemble le contexte d'énonciation, s'avère nécessaire aussi bien pour l'analyse de la structure du dialogue que pour son interprétation sémantique, comme on l'a vu dans la section précédente.

1.2.4 Le problème des références.

Le passage à la compréhension d'un dialogue ou d'un discours pose le difficile problème de la résolution des références. Sans entrer dans des considérations linguistiques, il nous paraît important de faire un certain nombre de remarques.
Nous envisageons ici successivement les références anaphoriques et les références elliptiques. elliptiques.
Nous entendons par anaphore dans un énoncé donné tout syntagme contenu dans cet énoncé visant à remplacer un autre présent dans un des énoncés antérieurs. Il s'agit bien d'une référence car la signification qu'il faut lui donner est celle de celui qu'il remplace.

exemple : énoncé précédent : " Tu veux une bière ?"

énoncé anaphorique : " J'en ai déjà une, merci."

Nous entendons par ellipse dans un énoncé donné toute omission de mots non indispensables pour sa compréhension en raison des échanges précédents. Il s'agit également d'une référence à la partie antérieure du discours car la compréhension d'un tel énoncé n'est possible que sur base de la signification des énoncés précédents.

exemple : énoncé précédent : "Où allez-vous ?"

énoncé elliptique : " A Namur ! "

La résolution des références anaphoriques n'est pas un processus encore pleinement résolu. Pour citer Nef, c'est encore un problème débattu que de savoir si l'assignation d'un antécédent à un pronom peut avoir un caractère autre que probabiliste. Deutsch (1974) fait remarquer cependant que la stucture hiérarchique d'un dialogue opératif (voir section précédente) peut apporter une réponse à cette question. Il considère que tout comme la tâche à accomplir se décompose en sous-tâches distinctes, le dialogue lui aussi se décompose en sous-dialogues correspondants (adaptation à la structure de la tâche, voir section précédente), et les anaphores opèrent en général au sein du même sous-dialogue que celui où elles apparaissent.

L'analyse des corpus résultant d'une étude réalisée par le groupe "dialogue homme-machine" du GRECO et simulant un centre de renseignements SNCF tendrait à montrer que les références anaphoriques sont, dans le cadre de dialogues finalisés, généralement non ambigues. Le référé se trouve le plus souvent dans le dernier énoncé de la machine ou celui du locuteur, et à défaut dans la partie antérieure du dialogue [PIERREL contact].
Retenons encore que les anaphoriques rencontrés sont soit les anaphoriques généraux (pronoms, etc...), soit des syntagmes spécifiques à la tâche (le prochain, le suivant,...).

Au niveau des références elliptiques, la même étude montre qu'elles sont également généralement non ambigues, et qu'elles se produisent dans 2 types de contexte :

[1] soit en réponse à une question de la machine ;

[2] soit après une information qu'elle a fournie. Dans ce cas, sous forme :

[2.1] d'un seul acte énonciatif (ex: le suivant);

[2.2] de 2 actes énonciatifs, le premier soit étant un marqueur de négation, soit reprenant une partie de l'information que la machine vient juste de fournir.

D'une manière générale donc, il semble que le problème de la résolution des références soit un tant soit peu simplifié dans le cas des dialogues finalisés, pour autant que l'on garde trace des échanges antérieurs.

Le problème demeure cependant complexe.

2 Conception d'un système de gestion de dialogue.

Sur base des conclusions qui ont été tirées de la sémantique du dialogue finalisé, du langage qui est utilisé et des références qui doivent être résolues, la conception d'un système de gestion de dialogues oraux finalisés nécessite la modélisation d'un certain nombre de connaissances, et la mise en oeuvre d'un certain nombre de fonctions.
Avant de les passer rapidement en revue, il faut s'attarder quelque peu sur des situations de dialogue auxquelles il devra pouvoir faire face.
Les 3 sous-sections suivantes sont empruntées à J.M. Pierrel dans [JMP...].

2.1 Les situations de dialogue.

Lors de la gestion d'un dialogue oral finalisé, le système peut se trouver dans diverses situations.
Ainsi, il doit pouvoir faire face à :

- des énoncés incompréhensibles ou non reconnus, en demandant une répétition complète ;

- des messages incomplets ou partiellement reconnus, soit en demandant une répétition (ce qui risque d'allourdir considérablement le dialogue), soit en faisant usage de valeurs par défaut et en demandant confirmation explicitement ou implicitement ;

- des messages ambigus, en posant une question d'éclaircissement ou en choisissant l'interprétation la plus plausible en fonction du contexte courant, ce qui nécessiterait également une demande de confirmation ;

- des messages complètement reconnus, mais mal interprétés (le seul moyen de s'en préserver est de demander une confirmation explicitement à chaque pas du dialogue...!!!), incohérents (par rapport à la tâche, ou aux énoncés antérieurs du locuteur), ou encore inutilisables car en dehors du champ d'expertise.

La définition du comportement du système et de ses réactions dans les diverses situations doivent être prises en compte, car elles seules définissent le type et la qualité du dialogue visé.

2.2 Les connaissances utilisables et leur mise en oeuvre.

A la lumière des différentes conclusions établies au fil des sections précédentes, nous pouvons déterminer les connaissances dont il faut disposer dans un système de gestion de dialogue.

Elles sont de type statique ou dynamique.

2.2.1 Les connaissances statiques.

Il s'agit de toutes les connaissances stables pour un type de dialogues et une classe d'applications déterminés.

on distingue :

[1] le modèle du langage, définissant le langage opératif admis (syntaxe, lexique) et établi à partir des corpus.

[2] Le modèle de la tâche, contenant la définition sémantico-pragmatique des entités et relations entre ces entités dans le cadre de la tâche, ainsi qu'une définition des buts et des sous-buts qui peuvent être atteint dans l'application visée.

[3] Le modèle du dialogue, contenant toutes les informations nécessaires pour le contrôle du dialogue et sa conduite.

[4] le modèle de l'utilisateur, définissant l'utilisateur type (ou les utilisateurs type) tant au niveau de ses intentions que de ses croyances vis-à-vis de la tâche, etc... [PIERREL,ergo].

2.2.2 Les connaissances dynamiques.

Il s'agit ici de toutes les connaissances qui sont amenées à se modifier soit d'un dialogue à l'autre, soit au sein d'un même dialogue.

On distingue :

[1] l'univers de la tâche, complément indispensable du modèle de la tâche, regroupe le même type d'informations, à la différence qu'elles sont susceptibles d'être modifiées.

[2] l'historique du dialogue, dont on reparlera plus loin, et qui constitue l'ensemble des échanges antérieurs au sein d'un même dialogue.

[3] Et finalement, un ensemble de solutions partielles ou temporaires constituées à une étape du dialogue.

Toutes ces connaissances sont utilisées de manière particulière pour la réalisation des fonctions spécifiques au traitement du dialogue oral finalisé.

2.2.3 Les fonctions nécessaires.

Les principales fonctions spécifiques au traitement d'un dialogue oral sont :

[1] le contrôle des canaux de communication gérant la "ligne" en début et en fin de dialogue, ainsi que lorsque de trop longs silences se produisent (envois de messages particuliers, etc...).

[2] La compréhension de phrases, établissant une représentation sémantique des énoncés en s'appuyant sur le modèle du langage, et éventuellement sur celui de la tâche (partie sémantique).

[3] L'interprétation contextuelle, qui fait le lien entre les énoncés, la tâche, et le contexte courant du dialogue, en s'appuyant sur l'historique de ce dialogue, et le modèle de la tâche (partie sémantique).

[4] le gestionnaire du dialogue, assurant son bon déroulement et sa progression, bien évidemment sur base de son modèle

[5] le raisonneur, véritable coeur du système, déduisant, à partir des interventions du locuteur (historique du dialogue), du modèle et de l'univers de la tâche, de celui de l'utilisateur et de l'ensemble des solutions partielles, les intentions de celui-ci (i.e. sa "requête"), ainsi que les particularités qui constituent cette requête, et évaluant, toujours sur base de ces mêmes données, son procédé de résolution, c'est-à-dire les informations nécessaires à sa réponse, celles manquantes qu'il faut encore obtenir, et finalement la réponse elle-même;

[6] et enfin la génération des interventions de la machine, utilisant le modèle du langage, quelque peu "en sens inverse".

Avant d'étudier en profondeur la conception du système de gestion de dialogues finalisés en développement à Nancy, nous pensons qu'il faut insister sur un certain nombre de points particuliers.

3 Questions spéciales.

Dans cette section, nous désirons insister sur l'importance de la représentation de l'univers de la tâche, de l'historique du dialogue, et de la cohérence.
En effet, ils prennent une importance toute particulière dans le cas de dialogues finalisés, ce qui justifié qu'on les approndisse un peu.

3.1 Importance de la représentation de l'univers de la tâche.

Par univers de la tâche, nous entendons ici aussi bien le modèle de la tâche (partie statique), que l'univers dont nous avons parlé dans les connaissances à mettre en oeuvre (partie dynamique).
Conjointement, ils constituent une description complète de l'application, et jouent, à ce titre, un rôle fondamental au moins à 2 égards :

[1] Ils constituent d'une part une description du "monde" dans lequel l'application se place. Cet élément est capital au niveau de l'interprétation sémantique des énoncés.
En effet, comme on l'a vu, la signification d'un énoncé dépend simultanément des mots qu'il contient et de leur signification, des relations entre ses éléments, et du contexte d'énonciation dans lequel il est produit.
Dans le cadre d'un dialogue finalisé, le "monde" de la tâche, décrit par les entités et leurs relations possibles, ainsi que les différentes situations et actions qui peuvent exister à un moment donné et à un endroit donné, constitue la majeure partie de ce contexte d'énonciation.

[2] D'autre part, ils comprennent la description proprement dite de la tâche en termes de buts à atteindre et de sous-buts qu'il faut préalablement accomplir pour les satisfaire. A ce titre, ils correspondent à des procédés de résolution de cette tâche et sont donc des informations de base pour le raisonneur (voir ci-dessus, section 2.2.3), informations sans lesquelles il ne pourrait déterminer la façon de satisfaire les

requêtes des utilisateurs.

3.2 Importance de l'historique du dialogue.

Nous avons vu que la compréhension du discours passait par une reconstruction dynamique et symbolique des schémas mentaux que l'on a relativement à ce discours. Ces constructions, mises ensemble, nous amène progressivement à bâtir une représentation de ce qui a été dit.

Intuitivement vient alors l'idée d'une structure complexe dans laquelle chaque élement d'un énoncé prend sa place tout en élargissant en même temps cette structure.
Ainsi, dans cette structure, chaque entité évoquée dans le discours aura sa place et les relations sémantiques existant entre ces entités y seront représentées.

Cette structure a une fonction de stockage qui est rendue nécessaire parce qu'il y a une tâche à réaliser. Cette tâche suppose la connaissance d'un certain nombre d'informations qui doivent être délivrées par l'usager. Or celui-ci ne les fournit pas en une seule fois. Il faut donc les conserver au fur et à mesure qu'elles sont données.
D'autre part, pour que le dialogue puisse aboutir il est nécessaire que le locuteur ne puisse pas mener le système en bateau ', c'est-à-dire que le système doit pouvoir confronter ce qui lui est dit par le locuteur avec ce que celui-ci lui a déjà dit, évitant ainsi de tourner en rond si l'usager ne cesse de se contredire.

Cette structure sera l'historique du dialogue. Celui-ci peut être envisagé selon plusieurs approches.

[1] On peut le concevoir comme une collection des représentations syntaxiques et/ou sémantiques des énoncés du locuteur.
Ce type d'approche est cependant peu satisfaisant car si une conservation séquentielle de ces structures syntaxiques peut permettre de résoudre sans trop de difficultés les références, elle n'est guère opératoire d'un point de vue sémantique. En effet, les énoncés que nous produisons contiennent pas mal de redondances. Dès lors, si plusieurs expressions ou énoncés font référence à une même entité, il serait plus efficace et plus proche de la réalité de réaliser une certaine 'épuration' de ce qui a été dit pour n'en conserver que les entités essentielles et l'ensemble des traits ou carcatéristiques y afférant.

[2] Une représentation purement sémantique de l'ensemble des énoncés du locuteur ne peut pas non plus être suffisante. En effet, si elle répond au besoin de structure reliant les entités mises en oeuvre, elle néglige les aspects syntaxiques qui sont bien souvent indispensables pour la compréhension des énoncés. La compréhension passe inmanquablement par la résolution des ellipses et anaphores et nécessite donc des informations syntaxiques sur les énoncés précédents.

[3] Une représentation conservant les deux aspects semble donc la plus indiquée. Néanmoins, tenant compte des conclusions de Falzon et de Deutsh évoquées précédement, il semble que la conservation de l'ensemble des structures syntaxiques ne soit pas indispensable. Il suffit, si l'on se réfère à ces auteurs, de ne conserver que les énoncés relatifs à la sous-tâche ou au sous-dialogue en cours.

3.3 Importance de la cohérence.

La cohérence joue un rôle important à plus d'un niveau dans un système de dialogue oral homme-machine; elle y est même indispensable.
Elle doit être garantie

[1] entre la tâche et les énoncés du locuteur,
[2] entre ce que dit le locuteur à un moment donné et ce qui a déjà été compris auparavant par le système,
[3] au niveau de la description du modèle et de l'univers de la tâche.

3.3.1 Cohérence par rapport à la tâche.

Dans le cadre d'un langage opératif, il importe beaucoup que le locuteur s'en tienne à l'univers dans lequel se situe la tâche à accomplir, et ce pour les raisons suivantes

[1] Etant donné le caractère simple et limité de la tâche, il est donc nécéssaire de pouvoir déterminer, lorsqu'un usager effectue une requête auprès d'un tel système, si sa demande se situe bien dans l'univers en cause.
Par exemple, si la tâche est l'obtention d'informations météorologiques, la question "Quand part le premier train pour Paris ?" sera incohérente avec la tâche, puisque celle-ci ne consiste nullement à donner des

3.3.3 Cohérence de la description de la tâche.

Le modèle et la decription de l'univers de la tâche sont importants, on l'a vu. Ils correspondent à une source de connaissances et doivent dès lors posséder les caractéristiques suivantes :

- complétude et
- cohérence.

Le caractère complet de cette source de connaissances sera validé ou invalidé en cours de tests, voire même en cours d'utilisation du système.
D'autre part, pour autant que le modèle sous-jacent offre suffisamment de souplesse, cette description pourra faire l'objet de modifications.
A partir du moment ou il devient possible d'accroitre et de compléter cette base de connaissances, il devient nécessaire de prendre un certain nombre de précautions :

- n'y a t il pas de contradictions induites par les nouveaux éléments en regard des anciens ?
- les nouveaux éléments ne sont-ils pas simplement redondants avec les anciens ?

Autant d'aspects de la cohérence auxquels il y a lieu d'être sensible dans la conception d'un système de dialogue.

Dans un chapitre ultérieur, nous aborderons spécifiquement l'un des aspects de la cohérence de la description de la tâche.

4 Conclusion.

Après avoir quelque peu situé la communication orale homme-machine, nous avons présenté dans le chapitre I les principes généraux sous-tendant la compréhension de la parole continue. Nous avons mis en évidence le caractère non déterministe de ce processus.
Le passage à la gestion de dialogues oraux finalisés introduit un certain nombre de problèmes nouveaux dont la compréhension de ces dialogues sur base d'une certaine sémantique de dialogue ainsi que sur base de la définition du langage admis, la résolution des références, et la confrontation à diverses situations en fonction de types d'échanges caractéristiques.
Leur résolution nécessite la définition et la mise en oeuvre de nouvelles sources de connaissances et de nouvelles fonctions.

De manière tout à fait générale, un système de gestion de dialogues oraux finalisés devra pouvoir intégrer des sources de connaissances statiques et dynamiques. Les connaissances statiques correspondent au modèle du langage, au modèle de la tâche, au modèle du dialogue et au modèle de l'utilisateur. Les connaissances dynamiques décrivent l'univers de la tâche, l'historique du dialogue, et un certain nombre de solutions partielles ou temporaires.
Sur bas de ces sources de connaissances, il doit pouvoir intégrer des fonctions de contrôle des canaux de communication, de compréhension de phrases telle que celles décrites au chapitre I, d'interprétation contextuelle, de gestion de dialogue proprement dit, de raisonnement et enfin de génération des interventions de la machine.

Certains éléments méritent d'être mis en évidence.

[1] Le modèle et l'univers de la tâche sont fondamentaux au niveau de la détermination du sens des énoncés et des procédés de résolution des requêtes des utilisateurs.

[2] L'historique est indispensable à la compréhension proprement dite du dialogue.

[3] En enfin la cohérence joue un rôle important et doit être garantie entre la tâche et les énoncés du locuteur, entre ce que dit le locuteur à un moment donné et ce qui a déjà été compris au paravant par le système, et enfin au niveau de la description du modèle et de l'univers de la tâche.

Les résultats des analyses présentées dans ces deux premiers chapitres constituent les bases "théoriques" sur

lesquelles repose le système de gestion de dialogues oraux finalisés développé à Nancy, système sur lequel nous avons été amenés à travailler.
Nous décrivons ce système en détail dans le chapitre suivant. Nous aborderons ensuite nos contributions personnelles.

CHAPITRE III : LE SYSTEME DE GESTION DE DIALOGUES ORAUX FINALISES DEVELOPPE A NANCY.

1 Présentation du système de dialogue oral finalisé.

1.1 Introduction et contexte du projet

Bénéficiant des apports des projets Myrtille I et Myrtille II, mais également conscients des limites de ceux-ci, la volonté des chercheurs de Nancy s'est naturellement tournée vers une étape ultérieure dans la recherche en compréhension de la parole : la mise au point d'un véritable système de dialogue oral homme machine. Ce système est actuellement en cours d'élaboration dans les laboratoires du CRIN à Nancy. Nous présenteront ici l'état de son développement.

Le cahier des charges du système [Carbonell 85-a] mentionnait les objectifs suivants : il fallait construire un système informatique susceptible de constituer un interface oral homme-machine qui soit à la fois efficace et confortable du point de vue de l'utilisateur. Ce système destiné à une utilisation par le grand public et non par des spécialistes de la linguistique ou d'un autre domaine devrait donc :

- être capable, dans une certaine mesure, de gérer et de structurer un dialogue oral finalisé.

- disposer d'une expertise dans un domaine relativement limité et simple.

- être capable de comprendre et de satisfaire des requêtes en provenance d'utilisateurs ou d'usagers s'exprimant dans un langage naturel sans restriction ni entrainement locutoire préalable.

Ce type de système offre un large éventail de possibilités commerciales et industrielles allant de la consultation, voire de la mise à jour, de bases de données relativement simples à la commande automatique d'articles sur base de catalogue (type vente par correspondance actuellement) en passant par la mise à jour d'agenda vocaux electroniques ou de plannings.

A ces objectifs opératoires, s'ajoutaient diverses orientations et lignes de conduites.

Premièrement, afin d'offrir un maximum de confort et de naturel au locuteur, il fallait à tout prix eviter de produire un système qui, bien que fonctionnant à la voix, reviendrait à un simple système de menus. La stratégie adoptée ne pouvait donc pas se résumer à une énumération des propositions et possibilités du système suivie des choix effectués par l'usager. Le contrôle du déroulement du dialogue se devait d'être partagé entre le système et le locuteur; tout au moins au cours des premiers échanges, le système pouvant par la suite prendre assez vite le contrôle du dialogue afin de l'orienter par des questions précises permettant ainsi de tendre le plus vite et le plus efficacement possible vers la satisfaction de la requête du locuteur.

Les expériences antérieures avaient démontré qu'il ne fallait pas espérer obtenir une compréhension exhaustive des énoncés dans un tel contexte. Les raisons de ceci sont assez claires

- le nombre élevé de locuteurs potentiels,
- la liberté relative d'élocution qui leur est laisséee.

D'un autre côté, la démarche de compréhension ne pouvait se limiter à un processus de recherche et de reconnaissance de certains mots-clés.

De plus, nous situant dans le cadre de la gestion d'un dialogue oral finalisé, il était nécessaire, comme on l'a vu dans le chapitre précédent, de mettre en oeuvre des sources de connaissances et de fonctions spécifiques, en plus des connaissances "traditionnelles".
Ces informations pouvaient également être utilisées pour pallier les lacunes de la phase de compréhension des énoncés.

Un problème crucial pour la mise en oeuvre de telles connaissances était leur représentation, nous l'aborderons ultérieurement.

Une dernière ligne directrice du projet était l'importance conférée au module dialogue du système. Déjà au cours des projets Myrtille I et Myrtille II, les concepteurs [Pierrel 81] avaient souligné l'importance d'un composant dialogue dans un système de compréhension de la parole.

En effet, y a-t-il communication entre plusieurs personnes, s'il n'existe pas de dialogue entre elles ? Il est pour le moins évident qu'en l'absence de ce dernier, la

communication sera insatisfaisante pour ceux qui y prendront part. Nous avons de plus déjà évoqué les différentes fonctions importantes que joue le dialogue dans la communication orale. C'est cette importance qu'a voulu mettre en évidence l'équipe de J.M. Pierrel en confiant au composant dialogue un rôle prépondérant au niveau des structures de contrôle de tout le système.

La présentation du système que nous allons faire ici semblera peut-être incomplète ou imprécise à certains égards.
Les raisons en sont les suivantes :

- le système se base pour une bonne part sur les acquis des systèmes Myrtille I et II [cfr chapitre 2]. Nous préférons eviter certaines répétitions et renvoyer le lecteur à ces descriptions lorsque de plus amples informations s'avèrent utiles.

- Le système suppose le traitement préalable du signal acoustique par un système expert qui a fait l'objet d'une recherche spécialisée au CRIN. les résultats de cette recherche sont présentés en détails dans [Fohr 85].

- Ce système est toujours en cours de développement et dès lors certains points sont encore à l'étude, n'ayant pas fait l'objet d'une décision à l'heure où nous écrivons ces lignes, ou étant éventuellement susceptibles d'être modifiés.

Nous voulons également insister sur le caractère scientifique de la démarche suivie, dans laquelle l'application traitée ne tient lieu que de support afin de démontrer la faisabilité de tels systèmes.
Dès lors, nous présentons, dans les pages qui suivent, les principes d'un système de dialogue valables pour une famille d'applications de type "entre de renseignements", en les illustrant par des exemples empruntés à l'application test. Celle-ci est constituée par les renseignements administratifs contenus dans les pages roses de l'annuaire téléphonique des P.T.T. en France.

1.2 Les sources de connaissances.

Classiquement, les systèmes de compréhension de la parole prennent en compte deux types de connaissances. D'une part celles liées aux énoncés, c'est à dire la syntaxe, la sémantique, la pragmatique et la prosodie, d'autre part, celles liées aux mots utilisés, c'est-à-dire le lexique, la phonétique et la phonologie. Cette classification est cependant peu opératoire [Carbonell 85-a], lorsqu'il s'agit de modéliser un univers réduit, spécialement dans le cadre d'un langage opératif [Falzon 85].

Il existe aussi des connaissances qui sont susceptibles d'évoluer plus ou moins souvent, que ce soit au cours du cycle de vie du système ou au cours d'un même dialogue. Nous les rappelons brièvement ici, il s'agit

- des informations liées à l'application elle-même, ce sont certaines données de la description de la tâche qui varient plus ou moins régulièrement, par exemple dans le cas qui nous concerne le prix d'un timbre fiscal ou la couleur d'une carte d'identité.

- des informations liées au dialogue en cours; l'historique du dialogue, c'est-à-dire tout ce qui a été dit par le locuteur et qui est considéré comme vrai par le système; l'état courant du dialogue (celui-ci est en partie décrit par l'historique, mais ne serait pas complet si l'on y trouvait pas mention du stade auquel se trouve le dialogue); c'est ce type d'information qui permettra au système de relancer le dialogue ou d'orienter celui-ci sachant où il faut en arriver et ce qui est déjà connu.

- des informations liées au phases du dialogue ainsi que mentionné dans le chapitre précédent.

1.3 Les composants du système.

Ce système est constitué de cinq composants autonomes les uns par rapport aux autres. Chacun d'eux correspond à une ou plusieurs des sources de connaissances évoquées précédemment et nécessaires dans le processus de compréhension de la parole et la gestion de dialogues finalisés. Celles-ci s'étendent, comme le lecteur aura déjà pu s'en rendre compte, du niveau le plus bas, c'est à dire

celui le plus proche du signal acoustique, au niveau le plus haut, c'est à dire les connaissance sur l'application elle-même, ainsi que sur les différentes stratégies possibles de dialogue.

- APHON est le composant de traitement acoustico-phonétique qui réalise la segmentation du signal ainsi que l'étiquetage phonétique des différents segments. De façon plus générale, on peut dire qu'il construit une représentation de l'énoncé de l'usager sous forme d'un treillis phonétique(a).

- PROSO est le composant chargé du traitement des aspects prosodiques de l'énoncé. Il doit détecter sur le signal certains marqueurs spécifiques indiquant des fins ou des débuts de mots ou de syntagmes. Il peut aussi aider à déterminer si un énoncé est de type interrogatif, affirmatif, ou contestatif.

- LEX est le composant lexical du système. Comme tout composant lexical d'un système de compréhension de la parole, il joue deux rôles complémentaires :

  - d'une part il permet la vérification d'hypothèses concernant des unités lexicales émises par d'autres niveaux de traitement,

  - d'autre part, il réalise lui-même l'émission d'hypothèses lexicales.

  Il assure le passage entre le treillis de phonèmes et un treillis lexical; pour ce faire, il utilise différents types d'accès et de représentations phonétiques et phonologiques des mots du vocabulaire du système.

- SYN-SEM est le module d'analyse syntaxico-sémantique; son rôle est l'élaboration, sur base du treillis lexical associé à l'énoncé en cours de traitement, d'une ou de plusieurs représentations syntaxico-sémantiques de cet énoncé. Ce composant délivre une représentation syntaxique et une représentation sémantique de l'énoncé sous forme de réseaux à noeuds procéduraux mis au point par Pierrel et dont nous avons déjà parlé lors de la présentation du système Myrtille II (voir chapitre 2).

- Enfin le module de dialogue : DIAL. Pour chaque énoncé du locuteur, la représentation fournie par SYN-SEM fera

---

(a) Les chercheurs du CRIN ont proposé une spécification formelle de la notion de treillis phonétique, celle-ci est reprise en annexe.

l'objet d'une interprétation contextuelle avant d'être traitée par DIAL. Cette interprétation a deux objectifs principaux, d'une part compléter la représentation de l'énoncé livrée par SYN-SEM en y intégrant ce qui a déjà été dit (résolution des ellipses, résolution des anaphores, etc...), d'autre part transformer la représentation de l'énoncé fournie par l'analyseur syntaxico-sémantique en une représenation de la signification de celui-ci dans le cadre de l'application et du dialogue courant, représentation exploitable par le module DIAL. Cette interpretation sera réalisée en fonction de la phase courante du dialogue, de l'historique du dialogue et de l'univers de la tâche.

Ayant reçu cette nouvelle représentation de l'énoncé, DIAL générera une réponse appropriée visant à se rapprocher de la satisfaction de la requête du locuteur. Les réponses du système pourront être destinées soit à obtenir un complément d'informations de la part de l'usager (fonction de transfert d'information), soit à obtenir une confirmation (fonction de validation), soit à requérir du locuteur une meilleure articulation ou une meilleure prononciation lorsqu'il parle.

De façon générale, on peut dire que c'est ce module qui a en charge tout la gestion du dialogue ainsi que la réalisation proprement dite de la tâche. Cela l'oblige à "raisonner" de façon complexe et à gérer de manière très rigoureuse l'historique du dialogue.
Le rôle de meneur du dialogue s'appuiera sur des "scénarios" de dialogue décrits dans la base de connaissances et intégrant le résultat de tests réalisés sous des conditions semblables à celles dans lesquelles travaillerait une machine dans le cadre d'un centre de renseignements grand public. Un scénario doit posséder :

- des conditions de validité des énoncés émis. C'est à dire qu'il doit pouvoir être à même de déterminer si un énoncé émis par le locuteur en réponse à une question du système est valable ou non dans le contexte courant. Si le système a demandé "c'est bien aux Etats-Unis que vous voulez allez ?", la réponse du locuteur pourra par exemple être "oui, c'est ça " ou bien "pas du tout". Ces deux réponses sont valides. Par contre des réponses du genre "A partir de quel aéroport faut-il partir ?" ne seront pas valides dans le contexte courant;

- des hypothèses sur les suites possibles du dialogue;

- des actions faisant appel au raisonement, sélectionnant d'autres scénarios et permettant l'émission d'une réponse à destination du locuteur.

1.4 Flux d'information entre les composants.

Chacun des composants dispose de certaines connaissances statiques.
Par exemple, au composant lexical a été associé un dictionnaire permettant d'obtenir pour chaque mot qui y est repris des informations phonologiques et syntaxiques en plus de la description phonétique du mot.
De plus, de l'information peut être échangée en cours de processus de reconnaissance entre tous les composants du systéme.

Les informations dynamiques qui peuvent faire l'objet d'échange entre les composants peuvent être de quatre types : il peut s'agir de prédictions, d'hypothèses, de validations (ou d'invalidations) et de résultats.

- Les prédictions sont faites par un composant P à destination d'un autre composant P'. Elles portent sur une partie ou sur l' ensemble de l'énoncé en cours de traitement et qui n'a pas encore été traité par P'. Ces prédictions ont pour but de réduire le champ d'investigation de P' lorsqu' il devra traiter cet énoncé.

- Les hypothèses sont émises par le composant P et doivent être confirmées par le composant P' avant d'être complètement admises par P.

- Les validations ou invalidations sont en quelque sorte les réponses faites par le composant P' au composant P après que ce dernier lui ait envoyé, soit une prédiction, soit une hypothèse.

- Les résultats du traitement d'un énoncé par P', dans le cadre d'une analyse de gauche à droite ou du milieu

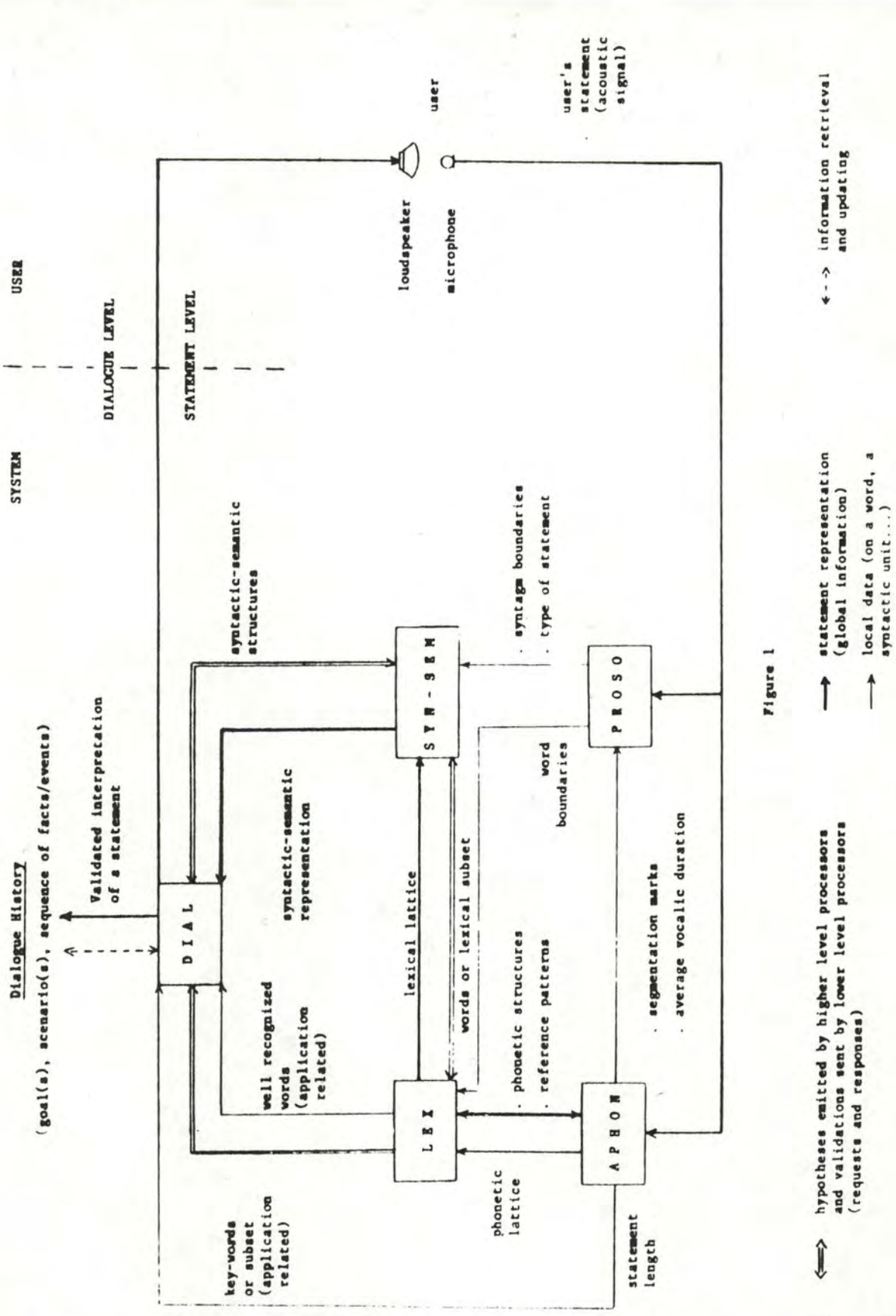


Figure 1.1: Echange d'informations dynamiques entre les composants.

vers les côtés.
Par exemple, si P est le composant syntaxico-sémantique et que P' est le composant lexical, alors les résultats

sont constitués par le treillis lexical généré par P' pour une partie déterminée de l'énoncé.

Une des caractéristiques de ce type de système est donc la richesse et la diversité de l'ensemble des informations dynamiques pouvant circuler entre les différents composants. Il est bien évident que la diversité des entités sur lesquelles travaillent chacun des composants nécessite l'établissement d'interfaces sophistiqués entre eux afin de leur permettre de communiquer; l'interpréteur contextuel constitue l'un de ces interfaces, puisqu'il opère entre le module d'analyse syntaxico-sémantique et le module de dialogue. Les différents flux d'informations pouvant survenir entre les composants du système sont illustrés à la figure 1.1

1.5 Fonctionnement et stratégies du système.

Tout en définissant les fonctions des différents processeurs, les concepteurs du système ont tâché de leur conserver un maximum d'indépendance et d'autonomie afin de conserver les potentialités d'un éventuel traitement sur des architectures parallèlles.

Actuellement les stratégies suivantes sont implémentées sur le système :

- une analyse de bas en haut (ascendante) à partir du niveau acoustico-phonétique vers le niveau lexical.

- Une analyse de haut en bas (descendante) à partir des niveaux les plus hauts vers le niveau lexical. De plus, est prévue la possibilité d'une analyse de haut en bas à partir du niveau lexical, se basant sur des ilots de confiance, ceci afin de pouvoir progresser en cas d'échec de l'analyse de haut en bas et de pouvoir reprendre cette dernière le plus rapidement possible.

Des flux d'informations entre les modules ont lieu entre SYN-SEM et PROSO, LEX et PROSO, APHON et LEX, APHON et LEX. Le type de relations existant entre ces modules est principalement du type producteur/consommateur, ces deux rôles étant occupés tantôt par l'un tantôt par l'autre module impliqué dans l'échange.

APHON et PROSO jouent le plus souvent le rôle de producteurs de prédictions à destination des autres modules. Toutefois, ils jouent également le rôle de consommateurs

lorsqu'ils doivent évaluer certaines hypothèses qui leurs sont fournies par LEX ou SYN-SEM.

Les relations existant entre SYN-SEM et LEX sont également du type producteur/consommateur. Dans une analyse dont la stratégie est orientée de haut en bas, SYN-SEM sera producteur d'hypothèses, alors que LEX les consommera; dans une stratégie orientée de bas en haut, ces rôles seront inversés.

On aura remarqué également les échanges entre DIAL et SYN-SEM, le second fournissant au premier des représentations syntaxico-sémantiques des énoncés en classifiant celles-ci selon des scores dans lesquels sont intégrés des connaissances émanant des modules lexical, phonologique et syntaxico-sématique.

1.6 Implémentation du système.

Les concepteurs du système se sont tournés, en ce qui concerne l'implémentation, vers un système à connaissances multiples, respectant l'analyse qui précède.
Cela favorisait la réalisation des buts suivants :

- La paramétrisation dy système du point de vue de l'application traitée.

  L'objectif principal était, rappelons le, de réaliser un système général et surtout adaptable à différentes applications de la famille "centre de renseignements grand public". Il était dès lors indispensable, afin de permettre le passage sans trop de problèmes d'une application à l'autre, que la description de l'univers de la tâche et la modélisation de celle-ci soit externe au système.

- La modularité du système.
  On a vu que le système se composait de cinq modules différents. Il peuvent être l' objet de traitements parallèles. Il convenait donc d'attribuer à chacun d'eux des connaissances qui leurs soient spécifiques.

- Une définition incrémentale du système.
  Tous les concepteurs, et en particulier ceux du système HWIM [LEA] (voir chapitre 2), se sont trouvés confrontés à l'absence d'un modèle de références bien

structuré et complet des mécanismes régissant la compréhension de la parole. Dans le cadre du nôtre nous ne disposions pas d'une théorie générale et rigoureuse de la représentation des informations liées aux langages opératifs ou aux dialogues oraux finalisés. Il fallait donc procéder de manière empirique pour la définition des connaissances nécéssaires. Le fait que les différentes connaissances soient représentées de manière externe aux processus qui les mettent en oeuvre en permet une définition plus souple, en autorisant une grande interactivité entre le concepteur et le système. De cette manière la définition d'une base de connaissances peut s'effectuer de manière incrémentale correspondant à une suite de raffinements successifs, et se basant, entre autres, sur l'analyse de corpus.

2 Le module dialogue.

2.1 Introduction.

Dans la suite, nous nous limiterons à une description plus détaillée du module de dialogue, et ce pour les raisons suivantes :

- il représente l'une des nouveautés principales du système de par l'importance qu'il y occupe.
- il occupe une place centrale dans l'architecture du système.
- c'est essentiellement autour de lui que s'articulent nos travaux.

Il se compose de trois parties : l'interpréteur, le gestionnaire de dialogue proprement dit et le raisonneur.

Le rôle de l'interpréteur est d'interpréter les énoncés du locuteur, c'est à dire de leur donner une représentation de leur signification dans le cadre de la tâche, de la phase de dialogue en cours et, de l'historique du dialogue.

Ce composant fait l'objet d'un chapitre particulier, mais l'on peut d'ores et déjà dire qu'il fonctionne en appliquant à une représentation syntaxique et une représentation sémantique de l'énoncé un certain nombre de règles dites d'interprétation qui ont été collectées et stockées dans une base de connaissances spécifique. Pour réaliser son objectif, il a accès à diverses informations telles que la description de l'univers de la tâche, l'historique du dialogue et même le lexique pour y obtenir des informations de type sémantique.

Le gestionnaire du dialogue a sous sa responsabilité le contrôle du dialogue. Il reçoit du raisonneur des indications sur les informations à obtenir de la part du locuteur et à partir de ces indications, il génère les questions adéquates.
De plus, il possède un certain nombre de connaissances relatives au déroulement de dialogues oraux finalisés. Elles sont représentées par des scénarios de dialogue qui décrivent des situations de dialogues (cas de contestations,

demandes de reformulation, accusés de reception, satisfaction, etc...) de façon à pouvoir les détecter et les traiter de manière appropriée.

De plus, lorsque cela s'avèrera utile, le gestionnaire du dialogue devra également pouvoir relancer le dialogue. Ce module est encore à l'étude et doit encore faire l'objet de nombreuses décisions, notamment quant au mode de représentation des scénarios. Plusieurs solutions candidates se dessinent en effet pour ce problème; nous nous limiterons à citer les "frames" [] et les réseaux à noeuds procéduraux [Pierrel 81].

Le but du système est donc dans une première étape de saisir et de comprendre la requête émise par un usager. Dans une seconde étape, il doit mener un dialogue avec cet usager afin de satisfaire celle-ci.

Le raisonneur aura donc à jouer le rôle que nous faisons inconsciemment jouer à notre esprit lorsque nous dialoguons, à savoir stocker les informations qui nous sont transmises, déduire un certain nombre de 'choses' à partir de celles-ci. C'est lui qui devra donner certaines indications au gestionnaire du dialogue pour les énoncés aux locuteurs. Enoncés ayant pour but d'obtenir des informations suffisantes permettant de satifaire la requête faite ou ayant pour but la validation de certaines informations émises par le locuteur ou encore des énoncés destinés à demander aux usagers de mieux articuler ou de parler moins vite par exemple.

2.2 Le modèle et l'univers de la tâche.

Nous avons déjà évoqué l'importance du modèle de la tâche et de la description de l'univers de l'application (cfr. chapitre 3). Dans le système développé à Nancy, c'est le composant de raisonneur qui exploite le plus cette source de connaisances.Il importe de la présenter ici.

Cette source de connaissances se caractérise par le fait qu'elle regroupe toutes les informations spécifiques à l'application. Ces informations sont réparties en différentes rubriques.

- les concepts : il s'agit de la définition d'un lexique des concepts utilisés pour cette application.

- les relations : il s'agit de la déclaration des relations existant entre les concepts.

- les renseignements : la traduction en termes de concepts des renseignements contenus dans les pages roses de l'annuaire, autrement dit, ce sur quoi porteront les requêtes qui seront adressées au système par les usagers.

- les faits par défaut : il s'agit de l'expression d'un certains nombre de généralités liées au domaine; celles-ci sont supposées être vraies lorsque rien n'indique qu'elles ne sont pas valables.

- la base de données : il s'agit d'une définition des réalités de l'application. Ces réalités sont vraies indépendamment de tout dialogue.

Avant d'évoquer le principe de fonctionnement du raisonneur, nous allons détailler quelque peu chacune des rubriques énumérées ci-dessus.

2.2.1 Les concepts.

La définition des concepts est destinée à exprimer l'univers de la compréhension du système. Les concepts sont représentatifs des objets, actions ou états impliquées dans l'univers de l'application. A chacun d'eux il correspond un concept.

Le concept est représenté par une entité lexicale qui l'identifie, il lui est aussi associé un profil qui détermine déjà un premier type de relation avec les autres concepts. Ainsi un concept peut porter sur un ou plusieurs autres concepts.
Par exemple, le concept SEXE porte sur le concept PERSONNE et l'on obtient le profil suivant :

sexe(personne)

De plus, chaque concept définit une classe de concepts, celle-ci pouvant être vide. Cette classe permettra ou interdira les instanciations des concepts. Elle permet également l'héritage des profils; en effet, chaque concept hérite du profil du concept représentant la classe dont il est membre.
Ainsi, le concept SEXE définit une classe qui contient les concepts MASCULIN et FEMININ. On pourra donc trouver des formules du type :

masculin(personne)

féminin(personne)

Tout concept apparaissant dans un profil peut être remplacé

par l'un des concepts de la classe qu'il définit.
Ainsi, le concept personne définit la classe suivante : LOCUTEUR, SYSTEME, TIERS.

personne[locuteur,système,tiers]

On pourra donc trouver des formules élémentaires telles que

féminin(locuteur)

ou encore

masculin(tiers)

qui seront valides.

Il est également possible de combiner les formules élémentaires entre elles. Pour ce faire, des opérateurs sont utilisés. Ces opérateurs sont

- la disjonction, 'ou' représentée par le symbole '|'.
- la conjonction, 'et' représentée par le symbole '&'.
- la négation, 'non' représentée par le symbole '-'.

Les formules élémentaires pourront donc être combinées de la manière suivante

&(f(a),g(b))

|(a,b)

-g(b)

&(-f(a),|(g(b),f(a)))

Ces combinaisons de formules élémentaires sont appelées des formules.

De plus, afin de disposer d'une représentation canonique des formules, facilitant ainsi les traitements, deux règles d'interprétation seront appliquées selon les cas.

&(a,|(b,c))

sera interprétée comme

|(&(a,b),&(a,c))

De même,

f(a|b) et f(a&b)

seront respectivement interprétées comme

|(f(a),f(b)) et &(f(a),f(b)).

Ces règles d'interprétation peuvent sans aucun doute être mises en défaut de nombreuses fois, mais elles sont justifiées par le fait qu'elles correspondent assez bien à la réalité traitée. Par exemple

fils(manu,&(joseph,marie))

peut s'interpréter sans mal comme

&(fils(manu,joseph),fils(manu,marie))

2.2.2 Les relations ou règles de réécriture.

Nous avons indiqué que les profils associés aux concepts établissaient déjà un premier type de relation entre ceux-ci.
Néanmoins, un univers tel que celui qui nous préoccupe contient des relations très complexes entre les entités qui le composent. Ce sont ces relations entre entités dont nous faisons usage de façon inconsciente lorsque nous déduisons certaines informations à partir d'autres.
Ainsi, si nous savons que P est le père de J, nous pourrons en déduire premièrement que J est un enfant de P, et deuxièmement que P est du sexe masculin.
De même si nous savons que A et E sont mariés et que E est de nationalité française, alors A peut obtenir la nationalité française.
Les relations de ce type traduisent tant la réalité de l'application qu'une certaine réalité plus commune.

Ces relations pourront être exprimées grâce aux formules, sous la forme de règles de réécriture pouvant s'apparenter à des règles de production de la forme

‹antécédent› => ‹conséquent›

Les règles seront écrites de la façon suivante

f1 => f2

où f1 et f2 sont des formules.

Soient les formules

f(x,y,z)

et

g(w,x,y)

dans la règle

f(x,y,z) => g(w,x,y)

les concepts x et y désigneront chacun une même entité de l'univers.
De même, dans la formule

f(x,x,y)

x désigne une seule et même entité.
Prenons par exemple le concept PERE, dont le profil est défini de la façon suivante

père(personne,personne)

ce qui signifie que ce concept porte sur deux occurences d'une entité de la classe définie par le concept PERSONNE.
Dans les règles, l'utilisation de la formule suivante

père(locuteur,locuteur)

provoquerait la confusion entre les deux entités. Il en résulterait que les deux occurences du concept LOCUTEUR désignerait une même entité. Pour éviter cela, il faut différencier les deux occurences de LOCUTEUR en utilisant un suffixe numérique.
On obtiendra donc la formule suivante

père(locuteur1,locuteur2)

Par défaut, le système considère que toute occurence d'un concept est muni du suffixe 0.

Les règles de déduction précédemment citées s'écriront donc de la manière suivante en terme de formules

père(personne1,personne2) =>

&(enfant(personne2,personne1),masculin(personne1))

et

&(époux(personne1,personne2),nationalité(personne1,france)) =>

nationalité(personne2,france).

Il faut cependant faire une restriction sur les formules utilisables dans les membres de droite des règles.

Le membre de droite ne peut en aucun cas contenir de formule contenant une disjonction ('ou', '|'). En effet, ces règles vont servir à effectuer des déductions et celles-ci doivent être certaines. Or l'introduction d'une disjonction introduit une certaine incertitude au niveau du raisonnement de par les différentes possibilités qu'elle offre.
De plus, il apparait que l'utilisation d'une disjonction dans le membre de droite d'une règle de réécriture correspond à l'énumération des éléments constitutifs d'une nouvelle classe de concepts.
Exemple : la règle

```
père(personne1,personne2) =>
        &(enfant(personne2,personne1),masculin(personne1))
```

aurait pu être écrite de la manière suivante :

```
père(personne1,personne2) =>
        &(|(fils(personne2,personne1),
            fille(personne2,personne1)),
          masculin(personne1))
```

Ce type de règle introduirait bien une certaine incertitude dans le raisonnement puisqu'il faudrait gérer les deux possibilités.
De plus il faut se poser la question de savoir si la règle décrirait bien la réalité, c'est-à-dire de savoir si le fait que P soit le père de J doit permettre d'inférer une information quelconque sur le sexe de J : a priori non.

2.2.3 Les renseignements.

Il s'agit de la description de l'objet de la tâche à accomplir, c'est-à-dire de ce sur quoi les requêtes des usagers pourront porter.
Pour l'application test plus particulièrement, ils correspondent au contenu des pages roses de l'annuaire téléphonique français.
Les requêtes pourront être du genre :

- "quel âge faut-il pour voter en France ?"

- "comment faire pour obtenir une carte d'identité ?"

il s'agit de modéliser des renseignements du type :
" Pour faire X, dans les conditions Y, il faut faire Z, sinon il faut faire Z'."
La structure d'un renseignement est donc la suivante

```
 but : < >
cond : < >
alors : < >
sinon : < >
```

Le renseignement "pour obtenir une carte d'identité, si l'on est majeur, alors il suffit de se présenter à la mairie, sinon, il faut se présenter à la mairie en compagnie d'un parent qui est majeur " peut se modéliser de la façon suivante :

```
      but : obtenir(personnel,carte_d_identité)
     cond : majeur(personnel)
    alors : aller(personnel,mairie)
    sinon : &(aller(personnel,mairie),
              accompagner(personnel,personne2),
              parent(personne2,personnel),
              majeur(personne2)
             )
```

Des conditions plus complexes peuvent également être exprimées, ainsi, le renseignement suivant

```
      but : obtenir(personnel,carte_d_identité)
     cond : nationalité(personnel,france)
    alors : cond : majeur(personnel)
            alors : aller(personnel,mairie)
            sinon : &(aller(personnel,mairie),
                      accompagner(personnel,personne2),
                      parent(personne2,personnel),
                      majeur(personne2)
                     )
    sinon :
```

exprimera la même chose que le précedent en indiquant de plus que seuls les gens de nationalité française peuvent obtenir une carte d'identité.

2.2.4 La base de données.

En plus des renseignements que nous venons de décrire, il existe un certain nombre d'informations qui ne font pas l'objet d'une action ou d'un but à effectuer sous certaines conditions. Toutefois, elles sont suceptibles de faire l'objet de certaines requêtes.
Par exemple, dans le cadre de l'application test :

- "Combien de temps faut-il pour obtenir un passeport ? "
- "Quelle est la couleur du permis de pêche sous-marine ?"

Ces questions sont proches de celles que l'on retrouve dans un query de base de données.
Les informations seront stockées dans une base de données et seront exprimées à l'aide des formules.
" Dix jours sont nécessaires pour l'obtention d'un passeport."

délai(obtenir(personne,passeport),10)

"Un permis de pêche sous-marine est de couleur arc en ciel."

couleur(permis_pêche_sous_marine,arc_en_ciel)

2.2.5 Les faits par défaut.

Il s'agit d'informations qui, si elles ne sont pas contredites, par l'usager seront considérées comme étant vraies. Elles seront utilisées par le raisonneur comme s'il s'agissait d'informations fournies par le locuteur. Si cela s'avère nécéssaire en cours de dialogue, elles feront éventuellement l'objet d'une demande de confirmation implicite ou explicite.

Dans l'application test, on supposera par exemple que la ville qui est en question est Nancy si le locuteur ne précise rien à ce sujet; de même, on supposera que celui-ci est de nationalité française.

2.3 Données du raisonneur.

L'ensemble de la base de connaissance décrite dans le point précédent est manipulée par le raisonneur, mais celui-ci manipule également un certains nombre d'autres données qui varient avec chaque dialogue traité.

2.3.1 La base de faits.

Il s'agit des informations comprises par le système à partir des énoncés de l'usager avec lequel le dialogue s'est établi.
Cette base de faits contient également l'ensemble des informations déduites par le raisonneur à partir de celles déjà connues et à l'aide des règles de déduction précédemment présentées.

Dans l'application test, si l'usager a dit " j'ai 19 ans et j'habite Paris. ", pour autant que la phrase ait été bien comprise, le système enrégistrera les informations suivantes :

- âge(locuteur,19)

- résider(locuteur,Paris)

De plus, du fait

âge(locuteur,19)

il sera déduit, à l'aide de la règle

âge(personne,sup(18)) => majeur(personne)

que le locuteur est majeur, ce qui se traduira par l'ajout d'un fait dans la base de fait

majeur(locuteur)

Plusieurs fois déjà, nous avons souligné les limites de la reconnaissance de la parole; il est donc possible qu'en cours de dialogue une information (un fait) qui avait été stocké s'avère être erronée. La responsabilité de ceci n'incombe d'ailleurs pas d'office au système, car il est très possible que ce soit le locuteur qui se soit mal exprimé, causant ainsi l'incompréhension.
Dès lors, certains faits devront, soit être retirés de la base de faits, soit être invalidés. Ces faits seront en premier lieu ceux résultant directement de la mauvaise compréhension de l'énoncé, mais aussi ceux qui auront été déduits à partir de ces derniers.
A cette fin, à chaque fait sera attribué un statut indiquant s'il s'agit d'un fait énoncé, d'un fait déduit, d'un fait par défaut ou s'il s'agit d'un fait qui a été invalidé.
Chaque fait se verra également attribuer un score, lequel devant rendre compte de sa fiabilité ou de son degré de vraissemblance en propageant les résultats des traitements effectués par les différents modules du système.

2.3.2 Le but.

C'est dans cette structure de données que sera stockée la requête du locuteur. Ce but sera confronté aux différents renseignements dont dispose le système pour déterminer les informations nécessaires à la satisfaction de la requête afin de pouvoir guider le composant de dialogue dans ses questions aux locuteurs.

Pour l'application test toujours, si le locuteur a demandé comment il devait faire pour obtenir une carte d'identité, le but enrégistré par le système devrait être :

obtenir(locuteur,carte_d_identite)

2.3.3 La question.

Toute demande d'information ne faisant pas nécessairement l'objet d'un but, il faut pouvoir représenter une question simple. C'est à cette fin que la structure a été créée.
Dans le cas de la requête précédente, une demande sera enrégistrée, celle-ci signifiera " Que dois-je faire ? " et sera enrégistrée sous la forme :

?action

Des requêtes comme " Quelle est la couleur d'un passeport ?" seront enrégistrées sous la forme

couleur(passeport,?)

alors que la requête " La couleur d'un passeport est-elle bien la couleur bleue ?" sera enrégistrée comme

couleur(passeport,?bleue)

2.4 Principe de fonctionnement du raisonneur.

La base de faits constitue simultanément l'historique du dialogue et celui du raisonnement. Elle est initialisée à partir de la base de données et des faits par défaut. Puis, au fur et à mesure que les énoncés sont compris, elle se trouve enrichie par leur interprétation et par les déductions qui peuvent en être faites à l'aide des règles de réécriture.

Le raisonneur applique la stratégie suivante :

[1] il intègre l'interprétation de l'énoncé à la base de faits et effectue la mise à jour en réalisant des inférences et en veillant à conserver sa cohérence. Nous indiquerons par la suite les différentes stratégies applicables en cas de détection d'incohérence avec le contenu de l'historique.

[2] Il vérifie ensuite si une réponse à la question posée peut être fournie. Une réponse pourra être fournie si une décision peut être prise pour l'ensemble des préconditions correspondant au renseignement sur lequel porte la question. Si tel est le cas, le raisonneur

fournit la réponse à l'usager via le gestionnaire du dialogue
Sinon, il recherche une précondition pour laquelle aucune décision n'a pu être prise et, toujours via le gestionnaire du dialogue, il provoque une question à l'usager, question dont la réponse devrait permettre de prendre une décision quant à la précondition et donc d'avancer vers la satisfaction de la requête.

3 Conclusion.

La plupart des travaux en traitement de la parole continue se sont toujours attachés à comprendre des phrases, négligeant l'établissement d'un dialogue entre l'homme et la machine. Outre les résultats qu'ils ont pu fournir, ils ont le mérite d'avoir présenté un certain nombre de solutions - ou tout au moins d'ébauches de solution - à l'ensemble des difficultés se posant, que ce soit au niveau des informations à prendre en compte ou au niveau des structures de contrôle de ces systèmes.

Sur ces acquis, il semble maintenant possible de greffer des techniques nouvelles visant à mettre en oeuvre un dialogue naturel entre l'homme et la machine dans un contexte à la fois simple et général. Nous avons essayé de présenter l'état actuel des recherches du groupe dirigé par J.M. Pierrel et J.P. Haton. Eux-mêmes reconnaissent qu'ils sont encore loin d'un système opérationnel.
Néanmoins, outre l'amélioration de l'existant, leur objectif réside essentiellement dans la preuve de la faisabilité d'un tel système.

CHAPITRE IV : L'INTERPRETATION CONTEXTUELLE D'ENONCES DANS UN SYSTEME DE DIALOGUE.

1 Définition de l'interprétation contextuelle.

1.1 Définition.

Pour bien mettre en évidence ce à quoi correspond l'interprétation contextuelle dans le système développé à Nancy, il faut la resituer par rapport à d'autres fonctions dont elle dépend.

Comme on l'a vu ci-dessus, l'objectif de l'analyse syntaxico-sémantique (fonction SYN-SEM) est principalement d'établir 2 représentations de l'énoncé courant: l'une syntaxique, l'autre sémantique.
La première, résultat de la mise en oeuvre des réseaux à noeuds procéduraux, met en évidence les catégories et les relations grammaticales qui unissent chacun des constituants.
La seconde représente le type de liens sémantiques qui existent entre ces divers constituants, toujours au sein de l'énoncé courant. Il faut souligner qu'elle résulte d'une première interprétation par rapport au contexte local de cet énoncé.
Ces 2 représentations sont avant tout des représentations descriptives, en ce sens qu'elles se bornent à décrire les types des éléments présents et leurs relations.

Le module de dialogue, quant à lui, doit disposer d'une représentation opératoire de la signification que prend cet énoncé dans le cadre du dialogue courant et de l'application visée. C'est sur elle qu'il doit s'appuyer pour "raisonner", et à partir de là générer les questions complémentaires éventuelles en vue de préciser la ou les requêtes de l'utilisateur, et ultimement les réponses la ou les satisfaisant.

Nous appelons interprétation contextuelle la fonction qui assure le passage de ces représentations descriptives à cette représentation opératoire.
En d'autres mots, c'est elle qui effectue le lien entre signification "locale" des énoncés, dialogue en cours, et application , en vue d'une utilisation (représentation opératoire) par le module de dialogue.
Il s'agit bien d'une interprétation en ce sens qu'elle doit attribuer une signification aux énoncés; elle est contextuelle car cette signification est attribuée en rapport avec le dialogue et la tâche. Cette fonction fait partie intégrante du module de dialogue.

1.1.1 Interprétation contextuelle par rapport au dialogue.

L'interprétation est donc contextuelle par rapport au dialogue en ce sens qu'elle doit établir la signification de l'énoncé dans le dialogue courant.
Cela signifie qu'elle doit d'une part établir son rôle dans le cadre du dialogue, et d'autre part s'appuyer sur un historique des interventions pour la détermination du sens de certains éléments qu'il contient, voire même de sa signification globale.

Le rôle d'un énoncé peut être très divers. Il peut constituer un acte de contrôle de dialogue, ou bien un acte totalement orienté par la tâche, comme l'expression d'une requête, un apport d'information précisant cette requête ou la personnalité du locuteur, etc... (voir chapitre II).
Il est important de bien déterminer ce rôle car le traitement devant être appliqué à la représentation résultant de l'interprétation de l'énoncé sera différent suivant qu'il appartient à l'une ou l'autre des catégories. De plus, il faut souligner que cette distinction ne correspond pas à une simple dichotomie, certains énoncés étant de composition mixte.

L'interprétation doit pouvoir également s'appuyer sur le dialogue courant, et plus précisément sur l'historique de ce dialogue, pour la détermination du sens de certains de ses constituants, voire de sa signification complète.
il s'agit ici plus particulièrement des réferences anaphoriques et elliptiques. En effet, comment attribuer un sens aux interventions du locuteur telles que :

- "j'en ai déjà une "
- "à 6 heures"

sans savoir à quoi "une" correspond, et quelle est la question qui a comme réponse "à 6 heures" ?
L'interprétation d'énoncés comportant ce genre de références passera nécessairement par leur résolution, et est donc, à ce titre, pleinement contextuelle par rapport au dialogue.

1.1.2 Interprétation contextuelle par rapport à la tâche.

Le modèle de la tâche et l'univers de l'application sont décrits à l'aide de concepts et de relations possibles entre ces concepts.
L'interprétation est contextuelle par rapport à la tâche dans la mesure ou elle doit établir une représentation opératoire de la signification que prend cet énoncé dans le

cadre de la tâche, c'est-à-dire en termes d'occurences de concepts et de relations entre elles.
Cela signifie qu'elle doit pouvoir identifier la signification des termes du vocabulaire admis pour l'application visée (le lexique), ce qui revient, dans le modèle utilisé, à identifier les concepts auxquels ils correspondent.

En outre, elle doit mettre en évidence le rôle que jouent les constituants de l'énoncé dans sa signification générale en gardant en arrière-plan la tâche à accomplir.

ex: "je suis mineur et je voudrais savoir si je peux avoir une carte d'identité".

Le sous-ensemble "je suis mineur" joue un rôle particulier: il constitue un apport d'informations précisant les caractéristiques spécifiques de la requête du locuteur.

L'interprétation de cet énoncé est également contextuelle par rapport à la tâche dans la mesure ou elle fournit ce genre d'informations en plus de la représentation de sa signification dans le cadre de l'application traitée.

1.2 Objectifs.

Pour la réalisation de la fonction d'interprétation contextuelle, nous nous sommes fixés un certain nombre d'objectifs généraux:

[1] La situation que nous venons de décrire correspond quelque peu à une situation "idéale".
En effet, nous avons implicitement adopté l'hypothèse selon laquelle le module syntaxico-sémantique fournissait une représentation syntaxique et sémantique complète et exacte. Nous avons déjà longuement insisté dans le chapitre concernant la compréhension de phrases sur le caractère non déterministe du processus et la présence presqu'inévitable d'erreurs.
La précarité de cette hypothèse requiert donc que l'interprétation contextuelle soit à même de traiter des structures syntaxiques et sémantiques incomplètes. Dans le pire des cas, elle devrait même pouvoir s'exécuter sur une simple structure lexicale, lorsqu'aucune autre structure n'aura pu être reconnue.

Il faut donc que cette fonction soit aussi robuste que possible, quelles que soient les entrées qui lui sont fournies.

[2] L'interprétation contextuelle doit constituer un outil aussi général que possible, ce qui signifie qu'elle doit être aussi indépendante que possible des

représentations sur lesquelles elle travaille, de la version interprétée qu'elle doit générer, et de l'application.
Elle doit donc être très paramétrable.
Ceci rejoint d'ailleurs un des objectifs premiers de l'équipe de Nancy: développer un système valable pour une classe d'applications.

[3] Enfin, elle devra constituer un module souple, c'est-à-dire facilement modifiable et extensible, autant pour sa mise au point que pour la réalisation de l'objectif [2] décrit ci-dessus.
Cette souplesse s'avère également indispensable pour l'intégration de la fonction dans le système complet de gestion de dialogues oraux finalisés. En effet, il ne faut pas perdre de vue que nous travaillons dans le cadre d'un projet de recherche, et un certain nombre de choses sont susceptibles d'être modifiées et améliorées au cours de l'évolution des travaux. Cette qualité est donc nécessaire.

Le cadre général étant posé, nous pouvons maintenant aborder une analyse plus détaillée.

2 Mise en oeuvre de l'interprétation contextuelle.

Dans cette section, nous analysons d'abord plus en détails l'interprétation contextuelle.
Nous verrons ensuite sur quels types d'informations elle peut s'appuyer pour réaliser son objectif ultime.
Enfin, nous présentons le type de solution que nous proposons.

2.1 Présentation plus détaillée.

Pour préciser quelque peu la fonction de l'interprétation contextuelle, il faut d'abord revenir plus en détails sur les informations qu'elle reçoit en entrée, et celles qu'elle doit produire en sortie.

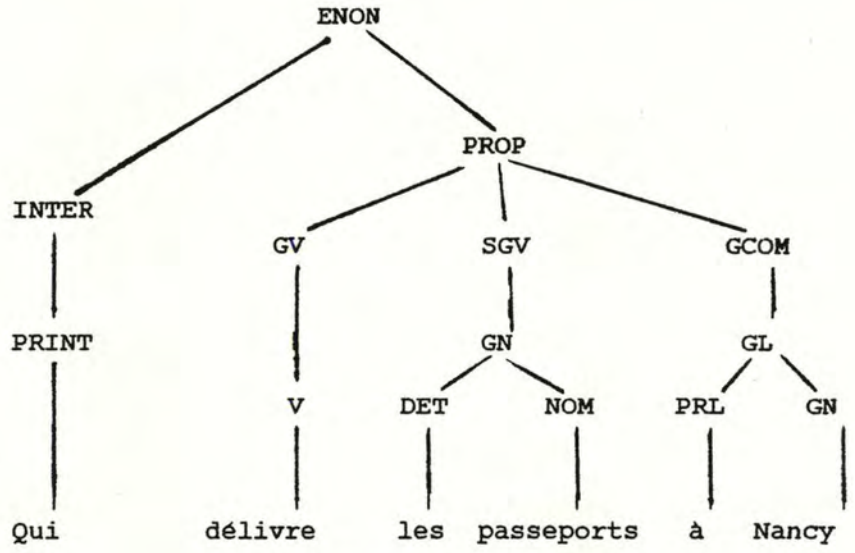
2.1.1 Les informations en entrée.

L'analyse syntaxico-sémantique fournit une représentation syntaxique et une représentation sémantique de l'énoncé du locuteur.

[1] La représentation syntaxique décrit la structure de surface de l'énoncé en entrée. Produite par l'analyse pas à pas des RNP décrivant le langage admis, elle se ramène à une arborescence dont les noeuds correspondent aux non-terminaux de la grammaire, et les feuilles aux éléments terminaux de la grammaire repérés dans l'énoncé.

Pour plus de détails sur les résultats fournis par les RNP, voir [Pierrel 81].

"Qui délivre les passeports à Nancy ?"



ENON = énoncé global
INTER = groupe interrogatif
PRINT = pronom interrogatif

PR = proposition
GV = groupe verbal
SGV = sous groupe verbal
NOMP = nom propre

GCOM = groupe complément
GN = groupe nominal
GL = groupe circonstanciel de lieu
PRL = préposition de lieu
DET = déterminant
NOM = nom

Figure 2.1: Exemple de représentation syntaxique.

[2] La représentation sémantique de l'énoncé exprime les relations de type sémantique existant dans l'énoncé entre ses différents constituants.
Elle est exprimée dans un formalisme de grammaire de cas.
Selon ce formalisme, une phrase consiste en une structure avec un prédicat (verbe, nom, ou adjectif) dérivé d'une primitive et un certain nombre d'arguments de cette primitive (groupes nominaux).
Les termes fonctionnant comme arguments de cette primitive sont spécifiés au moyen de fonctions sémantiques ou cas.
Un cas est donc l'expression d'une relation sémantique entre un terme et une primitive.
Une structure de cas d'une primitive est une séquence de cas permis pour cette primitive (obligatoires ou optionnels) [Deville 86].

exemple : "il me donne un passeport"

la primitive et sa structure de cas qui représente la structure sémantique de cet énoncé peut être schématisée par :

primitive	cas obligatoires		
EXCHPROD	(AGT	OBJ	BENEF)
donner	il	passeport	me

où EXCHPROD (échange production) est la primitive référant aux verbes transitifs qui décrivent une ACTION impliquant au moins 3 entités où l'une est physiquement ou mentalement transférée d'une entité animée à une autre entité animée ;

AGT (agent) est le cas correpondant à une entité animée contrôlant une ACTION ;

OBJ (objet) est le cas correspondant à l'entité animée ou non affectée par une ACTION (ou un PROCESSUS) ou impliquée dans un ETAT.

BENEF (bénéficiaire) est le cas correspondant à l'entité animée à qui une autre entité est physiquement ou mentalement transférée, ou à qui une ACTION est appliquée.

nb : les mots en lettres majuscules correspondent à des classes de primitives (voir ci-dessus).

Selon le formalisme qui a été arrêté pour le système développé à Nancy, la structure sémantique d'un énoncé peut être décrite de la manière suivante [Deville 86] :

< énoncé> -> <modalité> <proposition>

< modalité> -> <temps> | <voix> | <mode> | <forme>
| <modal> | <type>

<temps> -> présent | passé | futur

<voix> -> active | passive

<mode> -> infinitif | indicatif

<forme> -> positive | négative

<type> -> déclaratif | interrogatif direct
| interrogatif indirect

<modal> -> obligation | permission-capacité

<proposition> -> <primitive> <structure_cas>

<primitive> -> <verbe> | <nom> | <adjectif>

<verbe> -> dire | obtenir | etc...

<nom> -> demande | etc...

<adjectif> -> belge | etc...

<structure_cas> -> (<cas> {<gn> | <énoncé>})*n

<gn> -> (<prép>) <det> (<adjectif>) (<nom>) <nom>
(<gn> | <énoncé>)

<cas> -> agent | objet | patient | bénéficiaire |
source | destination | location | instrument |
mesure | but | temporel | condition.

<prép> -> par | etc...

<det> -> le | etc...

où

<xxxx> désigne un non terminal de la grammaire
xxxxx désigne un terminal de la grammaire

( ) indique quelque chose d'optionnel
| ou exclusif

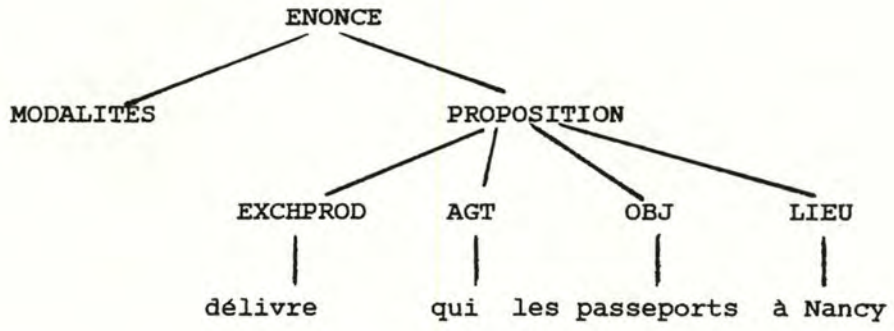
( )*n signifie que ce qui est entre parenthèses peut être répété n fois.

Les primitives sont réparties en 3 classes :n

* ETAT
* PROCESSUS
* ACTION.

Pour une définition précise de ces termes et une présentation plus détaillée, voir [Deville op. cit.].

"qui délivre les passeport à Nancy ?"



où EXCHPROD (échange production) est la primitive
réferant aux verbes transitifs qui décrivent
une ACTION impliquant au moins 3 entités
où l'une est physiquement ou mentalement
transférée d'une entité animée à
une autre entité animée ;

AGT (agent) est le cas correpondant à une entité
animée contrôlant une ACTION ;

OBJ (objet) est le cas correspondant à l'entité
animée ou non, affectée par une ACTION (ou un
PROCESSUS) ou impliquée dans un ETAT ;

LIEU (lieu) est le cas désignant la dimension spatiale
d'une entité dans un ETAT, ou la dimension
spatiale d'une ACTION ou d'un PROCESSUS.

Les modalités ne nous paraissant pas pertinentes dan cet exemple, nous n'en avons pas tenu compte.

Figure 2.2: exemple de structure de cas.

[3] Eventuellement, une structure lexicale lorsque rien d'autre n'a pu être reconnu. Il s'agit alors d'une simple chaine de mots reconnus.

ex: [...je...identité...]

2.1.2 Les informations en sortie.

Comme nous l'a souligné plus haut, l'objectif à atteindre est une interprétation contextuelle par rapport au dialogue et par rapport à la tâche à accomplir. En raison de ce double contexte à prendre en compte, les informations à fournir en sortie sont de 2 types :

[1] le type d'énoncé dont il est question. Vu la présence d'énoncés mixtes, les différents types d'énoncés possibles sont classés suivant qu'ils portent essentiellement sur la tâche ou sur le dialogue.

Le modèle adopté à Nancy est le suivant :

[1.1] les énoncés portant sur la tâche correspondent

* soit à des demandes d'informations

ex: Où dois-je aller ?

* soit à des productions d'informations, qu'elles soient explicites (ex: "je suis majeur et vacciné") ou implicites (ex: "oui, c'est ça").

[1.2] les énoncés portant sur le dialogue correspondent, quant à eux :

* soit à des contestations

ex: je n'ai pas dit ça !!

* soit à des demandes de répétitions explicites (ex:je n'ai pas compris où je dois aller) ou implicites (ex: pardon ?)

* soit enfin à des accusés de réception ou de satisfaction

ex: d'accord !, etc...

[2] Une suite de formules (concepts et relations entre ces concepts) décrivant la signification que prend l'énoncé

dans le cadre de la tâche à accomplir.

Pour la définition précise de la syntaxe de ces formules, voir annexe I.

Cette suite de formules se présente sous la forme d'un triplet

- but :
- information :
- question :

suivant que les différentes formules la constituant correspondent à une motivation du locuteur (but), à une information dégagée de l'énoncé (information), ou encore à une demande de renseignements.

Cette classification permet d'isoler le rôle joué par des sous-ensembles de l'énoncé dans sa signification globale, ce dont on a déjà parlé plus haut.

2.1.3 Rôle de l'interprétation contextuelle.

Les informations en entrée et en sortie étant clairement définies, nous pouvons maintenant donner une définition un peu plus précise de la fonction que doit remplir l'interprétation contextuelle.

Elle devra donc :

[1] identifier le type d'énoncé courant et le replacer dans le modèle des énoncés dont on a parlé plus haut ;

[2] identifier les occurences de concepts présents dans l'énoncé.
Le sous-ensemble du français accepté étant assez large, rien ne dit qu'ils se présenteront sous la même forme symbolique que celle qu'ils possèdent dans la base de connaissances du module de dialogue. Cela signifie qu'une occurence d'un concept peut être présent dans l'énoncé sous plusieurs formes extérieures, plusieurs mots pouvant, dans un contexte donné, signifier la même chose, ou plus précisément faire référence au même concept.

ex: avoir <--> obtenir
<--> posséder

etc...

Cette ambiguité, levée par le contexte, est inévitable lorque l'on désire garantir un maximum de naturel au dialogue.
Il n'existe donc pas a priori de correspondance bi-univoque entre mots du lexique (forme extérieure) et concepts.

Cette phase comprend également la résolution des références anaphoriques.

[3] identifier les relations entre les occurences de concepts présents dans l'énoncé. En effet, il est bien évident qu'elles n'apparaissent jamais comme "indépendantes" de la signification d'une phrase. Elles sont reliées sémantiquement, et ce sont ces relations qui donnent sa signification globale à l'énoncé.

[4] identifier le rôle de sous-structures composées d'occurences de concepts interdépendants dans la signification générale de l'énoncé.
Ceci correspond à identifier les formules correspondant à des buts, des informations ou des questions.
Cette étape nécessitera éventuellement la résolution des références elliptiques.

Nous allons maintenant voir sur quelles informations le module d'interprétation contextuelle peut se baser pour générer cette représentation de l'énoncé.

2.2 Les sources de connaissances à prendre en compte.

Comme nous l'avons déjà répété plusieurs fois, 3 choses rentrent en compte pour la détermination du sens d'un énoncé :

[1] la signification des éléments minimaux le constituant ;

[2] le type de relation(s) qui les unissent ;

[3] le contexte d'énonciation.

De manière plus précise, le premier constitue la dimension sémantique des connaissances de type lexicale.
Le second peut être appréhendé par la syntaxe et la sémantico-pragmatique, car toutes deux sont indicatrices des relations existant au sein d'un énoncé, l'une en terme de

liens grammaticaux, l'autre en terme de liens sémantiques possibles (ou nécessaires).
Enfin, le troisième aspect relève de la pragmatique et du dialogue courant.

Ce sont donc ces quatres sources de connaissances que nous allons envisager.

[1] La syntaxe.

La syntaxe renseigne sur les structures du langage admis, c'est-à-dire aussi bien les types de constructions générales que l'on est susceptible de rencontrer que les types de constructions particulières comme les structures stéréotypées ("oui, c'est ça", etc...). Elle met en évidence la façon dont les constituants sont organisés et les types de relations qui les unissent pour constituer ces structures.
Elle représente, à ce titre, une aide précieuse pour la détermination du type d'énoncé (interrogative, énonciative,...) et de la signification à donner aux relations mises en présence.

De plus, elle peut servir de support à un schéma d'analyse de l'énoncé. En effet, intuitivement, la façon dont l'interprétation devra procéder sur un énoncé donné dépend de sa structure.

Par exemple, l'intervention

"je voudrais obtenir une carte d'identité"

ne devra pas être abordée de la même façon que

"oui, c'est ça".

De même, la catégorie grammaticale des constituants qu'elle met en évidence joue un rôle central dans le déroulement de leur interprétation respective. On n'abordera pas une proposition de la même façon qu'un groupe nominal.

Enfin, dans certains cas, la structure syntaxique contribue à lever l'ambiguité relative à la signification de l'un ou l'autre mot de l'énoncé.

[2] Le lexique.

Pour réaliser l'interprétation d'un énoncé, il faut

bien évidemment une certaine connaissance des mots qui sont susceptibles de s'y présenter ainsi que des traits sémantiques caractérisant leur signification. C'est indispensable pour pouvoir identifier les concepts de l'application auxquels ils correspondent.

[3] La sémantico-pragmatique.

La sémantico-pragmatique inclut plusieurs éléments.
D'une part, la connaissance des concepts de l'application et des traits sémantiques s'y rapportant est bien-sûr fondamentale pour leur identification dans un énoncé donné.
D'autre part, la connaissance des cas possibles (obligatoires ou facultatifs) d'un prédicat ou d'une classe d'équivalence sémantique de prédicats est nécessaire, conjointement avec la syntaxe, comme schéma d'analyse de l'énoncé et des structures qui sont subordonnées au prédicat principal. Elle peut également servir pour la détermination de la signification de ces structures au sein de l'énoncé, ainsi que des relations entre elles et vis-à-vis des prédicats autour desquels elles s'articulent.

[4] L'historique du dialogue.

L'historique du dialogue est indispensable pour la résolution des références anaphoriques et elliptiques.
De plus, il peut fournir les informations sous-tendant des prédictions sur la structure ou la signification de l'énoncé courant.
Par exemple, si le dernier énoncé généré par la machine correspond à une question, il est raisonnable de prévoir que la structure de l'intervention suivante du locuteur sera une structure de réponse.
Ce genre d'hypothèses peut éventuellement rendre l'interprétation plus facile, plus rapide, ou même meilleure, dans le cas où aucune structure syntaxique ou sémantique n'a pu être reconnue.

2.3 Choix d'un type de solution.

Ayant défini l'interprétation contextuelle de manière précise, nous pouvons maintenant envisager un type de solution.

Les linguistes ont quelque peu négligé, jusqu'il y a peu, le domaine de la sémantique. Une des raisons de cet état de choses est qu'elle constitue un domaine d'étude très complexe par le fait qu'elle met en jeu des phénomènes très

divers de type linguistique, mais aussi philosophique, psychologique, logique, etc...
Bien que, depuis quelques temps, des études semblent à nouveau se développer dans le domaine, "personne n'a encore présenté ni même ébauché de théorie sémantique satisfaisante" [Lyons 70].
Cette constatation est, pour nous, fondamentale, car elle signifie qu'il n'existe pas de théorie générale de l'interprétation.
Notre solution ne peut donc pas s'appuyer sur une telle base théorique. Cependant, nous sommes dans une situation très particulière d'un dialogue (restreint) orienté par une tâche à accomplir. Dans ce cadre assez limité, développer le module d'interprétation contextuelle revient en quelque sorte à mettre au point un modèle d'interprétation valable uniquement pour les énoncés issus de tels dialogues.

Cela étant établi, les constatations suivantes s'imposent :

[1] ce modèle sera fortement lié à l'application ;

[2] son élaboration ne peut se faire qu'expérimentalement, et de manière incrémentale.

[3] il doit pouvoir se "débrouiller" non seulement dans la situation optimale, mais aussi en présence de structures incomplètes, voire même de simples chaines lexicales. Il s'agira donc d'un modèle essentiellement pragmatique.

A la lumière de la confrontation de l'absence de théorie générale (et des 3 constatations qui en découlent) avec les objectifs de robustesse, de généralité, et de souplesse, il nous a semblé qu'une solution purement procédurale n'était pas, à ce stade, envisageable.
Nous nous sommmes donc orientés vers une solution quelque peu plus "déclarative".

Plusieurs formalismes permettant une solution plus déclarative existent actuellement en Intelligence Artificielle. Qu'il s'agisse des Frames de Minsky, de la programmation logique, des langages orientés objets, ou des systèmes à base de règles de production, les uns et les autres présentent des avantages et des inconvénients.
Le lecteur intéressé peut à ce sujet consulter [Kayser 85].

Pour notre part, nous nous sommes orientés, en accord avec le groupe de travail de Nancy, vers un système de type "règles de production".
Il offre l'avantage d'être relativement simple à mettre en oeuvre, d'avoir déjà fait ses preuves, et enfin d'offrir une solution compatible d'une part avec la définition du problème à résoudre et d'autre part avec les objectifs

généraux que nous nous étions fixés.

Une seconde conséquence de cet état de choses est que notre participation aux travaux effectués à Nancy se limite au développement d'un outil à partir duquel il sera possible d'implémenter un modèle d'interprétation contextuelle d'énoncés.
Nous n'avons pas mis au point ce modèle.
Il nécessite en effet une étude approfondie des corpus de simulation ainsi que des connaissances linguistiques qui sortent à la fois du cadre de ce mémoire et de notre compétence.
Nous nous sommes donc limités à définir un outil suffisamment puissant et manipulable pour être utilisé ultérieurement pour la définition de ce modèle.

2.3.1 Les systèmes à règles de production.

Rappelons brièvement en quoi consiste un système à base de règles de production.
Il s'agit d'un système construit autour d'un ensemble de règles de production et d'un interpréteur de ces règles.

Une règle de production est une expression de la forme

‹conditions› -> ‹actions›

[1] Le membre de gauche (les conditions) décrit une certaine situation représentée dans un formalisme adéquat vis-à-vis du problème à résoudre.

[2] Le membre de droite donne les actions à réaliser lorsque la situation associée est détectée. Dans beaucoup de systèmes, ces actions correspondent en fait à des déductions. Dans ce cas, on parle d'un système de déduction.

Ces règles correspondent à des fragments de connaissances indépendantes, c'est-à-dire que l'on interdit qu'elles puissent s'appeler l'une l'autre.

Un système de production contient également une base de faits, constituées de propositions vraies, statiques, et une mémoire de travail, contenant des propositions décrivant l'état courant de l'exécution (énoncé du problème, déductions réalisées, etc...).
Certains auteurs ne font pas de distinction entre base de faits et mémoire de travail. Ils considèrent que la base de faits est uniquement la mémoire dynamique du système.

La présentation que nous en faisons ici semble plus générale car elle admet l'existence de connaissances d'un autre type que les règles (les faits), et qui ne sont pas à mettre sur le même pied que celles présentes en mémoire de travail en raison de leur caractère statique, i.e. valables pour toute la durée de vie du système indépendemment d'une exécution particulière.

Revenons aux règles.
Leurs conditions sont évaluées par un mécanisme de "pattern matching" avec les éléments de la mémoire de travail.
Une règle dont la prémisse, c'est-à-dire l'ensemble des conditions, est vérifiée est dite activable.

L'interpréteur, partie clé du système, correspond au mécanisme de mise en oeuvre de ces connaissances. Il fonctionne suivant un cycle de base (le "recognize/act cycle") qui correspond à la répétition des 3 fonctions suivantes :

[1] déterminer les règles activables en fonction des éléments présents en mémoire de travail (établissement du "conflict set") ;

[2] si plus d'une règle est activable, sélectionner celle à exécuter (résolution de conflit - "conflict resolution") ;

[3] exécuter la règle choisie en [2].

jusqu'à ce qu'il rencontre une action correspondant à un ordre explicite de fin d'exécution, ou bien qu'il soit bloqué car il n'existe plus de règles activables..

Le principe de raisonnement repose sur le modus ponens :

$$(P \ \& \ (P \Rightarrow Q)) \Rightarrow Q$$

Ces cycles sont initialisés le plus souvent par le positionnement d'un élément particulier en mémoire de travail.

La résolution du conflit entre les règles activables est réalisée en fonction de certaines stratégies qui dépendent fortement du problème à résoudre. On peut consulter à ce sujet le chapitre 5, [Winston 77] et [Mc Dermott 78].

Une technique utilisée pour le contrôle de l'exécution du

système est l'utilisation de méta-règles correspondant à des morceaux de connaissance sur la façon d'utiliser les règles proprement dites. Elle a le gros avantage de rendre la logique de raisonnement, d'implicite à l'intérieur de l'interpréteur, explicite dans une forme semblable aux règles elles-mêmes [Lauriere 82].

Le lecteur intéressé par plus de détails peut consulter [Winston 77] [Lauriere 82] [Hayes-roth 85].

2.3.2 Avantages de cette solution.

Le formalisme des règles de prodution a l'avantage d'être applicable à la définition du problème à résoudre dans le cadre des objectifs fixés.

En effet, d'après notre avis et celui du groupe de travail de Nancy, la logique des propositions sur laquelle un tel système repose devrait être suffisamment puissante pour exprimer un modèle d'interprétation contextuelle d'énoncés dans le cadre de dialogues orientés par une tâche bien précise.

De plus, ce formalisme permet de développer :

[1] un système robuste. Il est possible de développer plusieurs groupes de règles correspondant aux connaissances que l'on a sur la façon d'interpréter les énoncés en fonction de leur structure, l'un de ces groupes étant applicable à un moment donné et pour une situation donnée.

[2] un système aussi général que possible. L'impératif de généralité pour une fonction qui, par définition, est aussi fortement dépendante de l'application, signifie qu'il faut que le système qui l'implémente soit le plus paramétrable possible.
Un gros avantage des systèmes à règles de production est qu'il distingue la connaissance que l'on a du domaine (les règles et les faits), du mécanisme de mise en oeuvre de cette connaissance (l'interpréteur de règles). De plus, à cette modularité de départ s'ajoute la modularité intrinsèque aux règles elles-mêmes. Cela signifie qu'un tel système peut être paramétrable en quelque sorte "à 2 niveaux".
D'une part, un changement d'application ou de représentation nécessite uniquement la définition de nouvelles règles et leur intégration dans le système, les modalités d'utilisation de ces règles restant invariantes.

D'autre part, il est possible de distinguer facilement des règles "générales" de règles tout à fait dépendantes de l'application, ces dernières seulement devant faire l'objet d'une modification en cas de mise en oeuvre d'une nouvelle application.
Ainsi, ce formalisme assure-t-il une indépendance maximale du fonctionnement du module d'interprétation vis-à-vis des représentations qu'il manipule et de l'application visée par le système complet.

[3] un système aussi souple que possible. La modularité des systèmes de production et des règles elles-mêmes confère également au système un maximum de souplesse. Sa mise au point peut facilement être réalisée de manière expérimentale et incrémentale, en rajoutant de nouvelles règles ou en modifiant les règles qui n'apparaissent pas pertinentes.

2.3.3 Inconvénients de cette solution.

L'adoption du formalisme des règles de production pour la modélisation de l'interprétation contextuelle présente cependant quelques inconvénients.

D'abord, l'analyse ne peut pas être dirigée dynamiquement sur l'un ou l'autre élément de la structure en entrée.
En effet, dans un système à base de règles de production, les règles qui sont exécutées sont celles dont la prémisse est satisfaite, et qui "passent" à travers les différents filtres de la phase de résolution de conflits.
Les stratégies qui gouvernent cette sélection sont des stratégies "statiques", c'est-à-dire invariantes quelque soient la situation et l'énoncé courants. Or, nous avons vu que la structure de cas d'un prédicat, conjointement avec la structure syntaxique, peut servir de plan d'analyse de l'énoncé qui la possède. Dans certaines situations, il serait intéressant de pouvoir "orienter" l'interpréteur vers le traitement de telle partie plutôt que de telle autre. Ceci ne peut être évalué que de manière dynamique, en fonction de l'énoncé en entrée, de sa structure, ou simplement des résultats de l'interprétation d'un de ses constituants.

Bien sûr, le problème peut être contourné en utilisant le principe des contextes. Selon cette méthode, on peut restreindre le nombre de règles activables à un moment donné aux règles d'un contexte donné (le contexte actif) par le positionnement d'un élément particulier en mémoire de travail, et ainsi de manière indirecte limiter temporairement l'analyse à un aspect déterminé.

Cette solution ne semble cependant pas très satisfaisante car elle conduit inévitablement à compliquer fortement l'écriture des règles et va par conséquent quelque peu à l'encontre de l'objectif de souplesse et de modifiabilité du système.

Une autre solution serait de subordonner le fonctionnement de l'interpréteur aux résultats de l'évaluation de méta-règles rendant compte des stratégies d'analyse à suivre dans des situations données. Cette solution ne semble également pas optimale,le nombre de situations à envisager nécessitant le développement de beaucoup de règles, ce qui alourdirait considérablement le système, tant au niveau de sa mise au point qu'au niveau de son fonctionnement.

Un second inconvénient de cette solution découle de la spécificité de l'interprétation contextuelle. Elle requiert en effet des types d'actions particulières, non "standard" dans un système de production.
Par exemple, une des actions que l'on devrait pouvoir intégrer dans les règles est l'accès à l'historique du dialogue pour la résolution des références.

Ces 2 inconvénients majeurs nous ont orientés vers une solution basée sur le développement de notre propre système. Ce système reste un système à base de règles de production dans sa philosophie, mais présente un certain nombre de caractéristiques spécifiques qui en font un outil le plus général possible, mais spécialisé dans la réalisation de l'interprétation contextuelle.
Cette dernière remarque ne doit pas étonner ; l'objectif de départ n'était pas de développer un système faisant n'importe quoi suivant la façon dont il est paramétré, mais bien de développer un système aussi général et paramétrable que possible dans la tâche qui lui est dévolue, à savoir l'interprétation contextuelle d'énoncés dans le cadre de dialogues finalisés.

Ce système est présenté en détail dans le chapitre suivant.

3 Développement d'un outil spécifique.

Le système que nous proposons pour l'interprétation contextuelle est assez conséquent. En raison du temps dont nous disposions, nous n'avons pas été à même de l'implémenter complètement. Nous désirons cependant le présenter dans son entièreté ; nous décrirons ensuite le sous-système réalisé.

3.1 Le système d'interprétation contextuelle.

3.1.1 Présentation générale.

Basé sur le formalisme des règles de production, le système d'interprétation contextuelle se compose :

[1] d'une base de règles, que l'on désignera par la suite comme règles d'interprétation, correspondant aux morceaux de connaissance portant sur la façon de réaliser cette interprétation.
Le conséquent de ces règles correspond à des actions à effectuer dans des situations identifiées par la prémisse. Ces actions vont de la déduction d'informations pertientes sur l'énoncé à la détermination de morceaux d'interprétation proprement dite, en passant par des accès à l'historique du dialogue.
Ces règles correspondent à une représentation des connaissances syntaxique et sémantico-pragmatique (pour les relations sémantiques) dont nous avons parlé plus haut.

[2] d'une base de faits correspondant aux connaissances élémentaires portant sur des termes du lexique ou des concepts de l'application. Il s'agit donc des connaissances de type lexicale et sémantico-pragmatique (restreinte à la définition des concepts).

[3] d'une mémoire de travail contenant les représentations de l'énoncé à traiter, des informations déduites à partir de l'exécution des règles, et un certain nombre d'hypothèses portant sur l'interprétation à donner à l'énoncé en fonction de la dernière phase du dialogue locuteur/machine.
Elle correspond à la mémoire à court terme du système, et est clairement de type dynamique en ce sens qu'elle

contient des informations qui ne sont vraies uniquement que pour l'énoncé courant.

[4] d'une structure de données contenant l'interprétation de la partie d'énoncé qui a déjà été réalisée, et sa version définitive en fin d'exécution.

[5] et enfin, d'un interpréteur de règles dont l'objectif est de réaliser le cycle de reconnaissance/action (recognize/act cycle), c'est-à-dire :

* sélection de l'ensemble des règles activables en fonction des éléments présents en mémoire de travail et en base de faits;

* sélection de la règle à activer parmi cet ensemble de règles instanciées ;

* exécution de cette règle.

jusqu'à ce que l'on soit :

- arrivé à l'interprétation de l'énoncé

- ou bloqué (par exemple, lorsque l'interprétation n'est pas terminée et qu'il n'existe plus de règles activables).

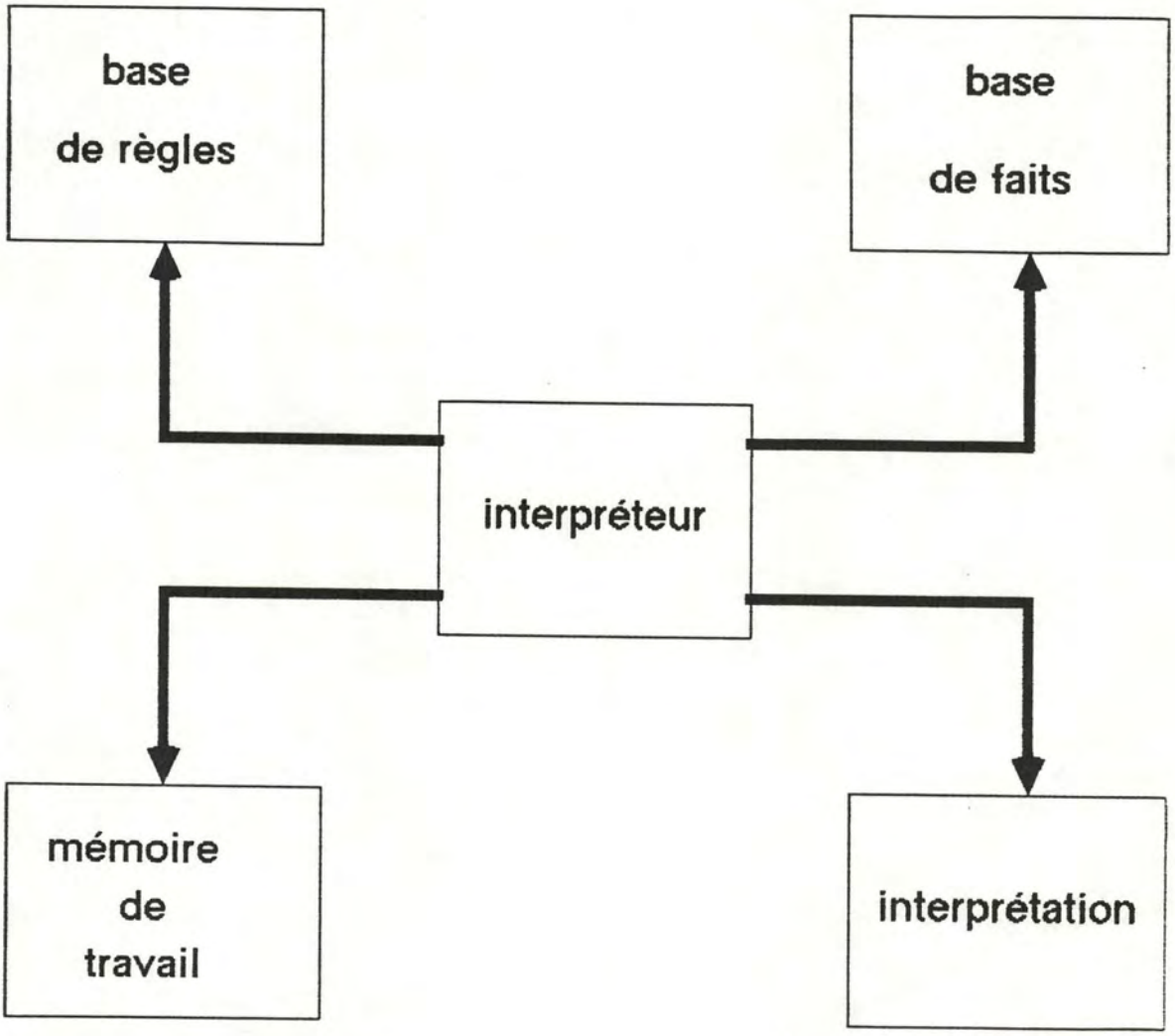


Figure 3.1: Schéma général du système.

L'intégration de ce module dans le système de gestion de dialogues finalisés nécessite en outre un certain nombre

d'interfaces particuliers, tant au moment de sa mise au point qu'au moment de son fonctionnement proprement dit.

[1] Au moment de la mise au point du système.
Deux types d'outils seront nécessaires. D'une part, un outil d'aide à l'écriture des règles et des faits, permettant au modélisateur de les définir facilement dans un mode de représentation adapté. D'autre part, un outil de compilation de ces règles et de ces faits, assurant le passage de cette représentation externe acceptable par l'homme à une représentation plus pertinente d'un point de vue informatique.

[2] Au moment du fonctionnement proprement dit, les 2 fonctions suivantes s'avèrent également nécessaires :

* un préprocesseur d'énoncé établissant la représentation interne de l'énoncé en mémoire de travail,

* un générateur d'hypothèses ayant pour fonction l'analyse de la dernière phase de dialogue et la génération d'un certain nombre d'hypothèses correpondant à des prédictions sur l'énoncé courant,

De manière tout à fait générale, voici comment le système fonctionnnera :

[1] traduction de l'énoncé par le préprocesseur d'énoncé ;

[2] génération des hypothèses sur l'énoncé ;

[3] tant qu'on est pas arrivé à une interprétation cohérente ou qu'on est pas bloqué, exécution du cycle de reconnaissance/action ;

[4] si on n'est pas arrivé à une interprétation cohérente, suppression des hypothèses en mémoire de travail et exécution du cycle de reconnaisance/action sans plus en tenir compte.

3.1.2 Formalisme de représentation.

Une étape importante dans la conception d'un tel système est la définition du formalisme dans lequel les informations manipulées vont être représentées, qu'il s'agisse de faits, d'éléments de mémoire de travail, ou des règles d'interprétation.
Nous nous attacherons d'abord aux faits et aux éléments de mémoire de travail; nous parlerons ensuite des règles d'interprétation.

3.1.2.1 Les faits et les éléments de mémoire de travail.

Toutes les informations élémentaires utilisées par le système d'interprétation, c'est-à-dire l'énoncé, les informations qui en sont déduites, les faits, et les hypothèses sont représentées à partir de relations.
Elles constituent la base du formalisme de représentation.

La définition que nous en donnons s'inspire fortement du modèle relationnel.

Considérant les définitions suivantes :

[1] un domaine Di est un ensemble de valeurs ;

[2] un n-uplet est un ensemble (d1,d2,...,dn) où di Di avec i {1,..,n} et où Di (1>=i>=n) est un domaine ;

nous dirons :

[1] qu'une relation est un sous-ensemble du produit cartésien d'un ou plusieurs domaine(s);

[2] qu'une relation finie est une relation ayant un ensemble fini de n-uplets;

[3] qu'un attribut Ai (ou constituant) de la relation est un identificateur auquel est associé un domaine Di. Il peut être vu comme un argument variable de la relation, prenant ses valeurs sur Di;

[4] qu'une instance de relation est un n-uplet pris parmi l'ensemble des n-uplets qu'elle contient;

[5] qu'un schéma de relation R(A1,A2,...,An) est une relation R avec ses attributs et ses contraintes;

[6] qu'une relation est définie en intention par son schéma, et en extension par l'ensemble de ses instances.

exemples : domaine V={v|v est un verbe du français}

schéma : mouvement(V)
/* signifie que v est un
/* verbe qui
/* exprime un mouvement.

instances : mouvement(aller)
mouvement(bouger)
mouvement(venir)

etc...

Etant donné ces définitions, un fait ou une information élémentaire sur l'énoncé ou sur l'historique sera représenté sous forme d'instance de relations finies.

L'adoption de ce formalisme de représentation peut être justifié par les considérations suivantes :

[1] il permet d'exprimer facilement les informations utilisées, car le concepteur du modèle d'interprétation définit lui-même les relations qu'il veut utiliser, ainsi que leur sémantique.

[2] ces informations peuvent ensuite être facilement manipulées (voir règles d'interprétation ci-dessous) par le fait de leur "banalisation", malgré leur différence de type.

[3] il est simple mais possède la puissance de la logique des propositions, ce qui paraît suffisant pour l'utilisation que l'on veut en faire.

Il présente cependant les inconvénients suivants :

[1] il est particulièrement lourd à manipuler par l'utilisateur humain ;

[2] il crèe des informations redondantes par rapport à l'ensemble du système de gestion de dialogues.
Ceci n'est cependant pas gênant, car on peut très bien s'en satisfaire dans un premier temps, mettre le système au point, puis éventuellement remplacer un certain nombre de tests sur les relations par des primitives d'accès à des structures de données partagées par plusieurs modules.

Avant de passer à l'étude des règles d'interprétation, nous allons voir comment tout cela est mis en oeuvre dans la mémoire de travail et la base de faits. Nous étudierons successivement les informations portant sur l'énoncé, l'historique du dialogue, les mots du lexique, et les concepts de l'application.

3.1.2.2 L'énoncé en entrée.

Comme on l'a vu ci-dessus, le module syntaxico-sémantique peut délivrer plusieurs représentations de l'énoncé: soit des représentations syntaxique et sémantique, éventuellement incomplètes, soit une chaine lexicale lorsqu'aucune autre structure n'a pu être reconnue.

Le préprocesseur d'énoncé assure le passage de ces représentations de l'énoncé à sa représentation em mémoire de travail sous forme d'une suite d'instances de relations finies mettant en évidence ses caractéristiques syntaxiques et sémantiques.

Ces instances devront correspondre à des relations prédéfinies.

exemple :

soit le domaine suivant :

T={ t | t est terme du vocabulaire }

soient les relations définies par les schémas suivants :

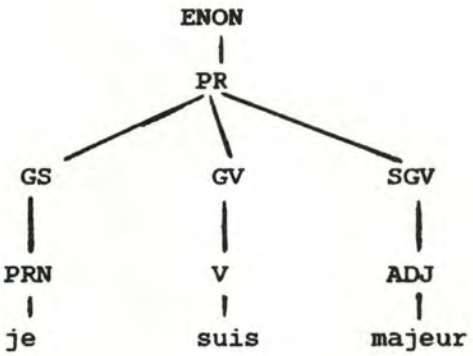
prédicat(t) où t∈T
/* signifie que t, terme du vocabulaire, est

```
     /* un prédicat (au sens grammaire de cas)
     /* de l'énoncé courant.


objet(t,t')    où t,t' ∈ T
     /* signifie que t', terme du vocabulaire, est
     /* mis en correspondance avec le cas objet
     /* (au sens grammaire de cas) de la
     /* primitive dont t est issu.
     /* Par définition de l'objet, une contrainte
     /* impose que prédicat(t) existe.


agent(t,t')    où t,t' ∈ T
     /* signifie que le terme du lexique t' est
     /* mis en correspondance avec le cas agent
     /* (au sens grammaire de cas) de la
     /* primitive  dont t est dérivé.
     /* Tout comme pour l'objet,
     /* la définition même de l'agent impose
     /* que la relation prédicat(t) existe.


présent(t)  où t∈T
     /* signifie que la proposition dont t
     /* est prédicat est du temps présent.


active(t) où t∈T
     /* signifie que la proposition dont le terme t
     /* est prédicat est à la voix active.

indicatif(t) où t∈ T
     /* signifie que la proposition dont le terme t
     /* est prédicat est au mode indicatif.

déclaratif(t) où t∈T
     /* signifie que la proposition dont le terme t
     /* est prédicat est de type déclaratif.

interr_directe(t) où t∈T
     /* signifie que la proposition dont le terme t
     /* est prédicat est interrogative directe.

positif(t) où t∈ T
     /* signifie que la proposition dont le terme t
     /* est prédicat est positive.

adjectif(t) où t∈ T
     /* signifie que le terme t du vocabulaire
     /* est un adjectif au sens grammatical.

pronom(t) où t∈T
     /* signifie que le terme t du vocabulaire
     /* est un pronom au sens grammatical.
```

```
pron_int(t) où t∈ T
    /* signifie que le terme t du vocabulaire
    /* est un pronom interrogatif au sens grammatical.

nom(t) où t∈T
    /* signifie que le terme t du vocabulaire
    /* est un nom au sens grammatical.
```

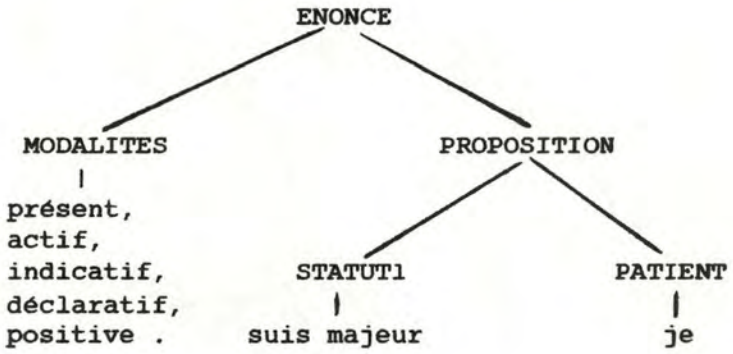
l'énoncé "je suis majeur"

représenté par



où ENON = énoncé
PR = proposition
GS = groupe sujet
SGV = suite verbal
PRN = pronom
V = verbe
ADJ = adjectif
GV = groupe verbal

et par



où STATUT1 est la primitive correspond à un verbe copule qui n'exprime pas un mouvement et qui représente un ETAT dans lequel une entité animée acquiert la propriété exprimée par ce prédicat, et où aucune dimension spatiale n'est spécifiée.

PATIENT correspond à une entité animée affectée par un PROCESSUS ou liée à un ETAT.

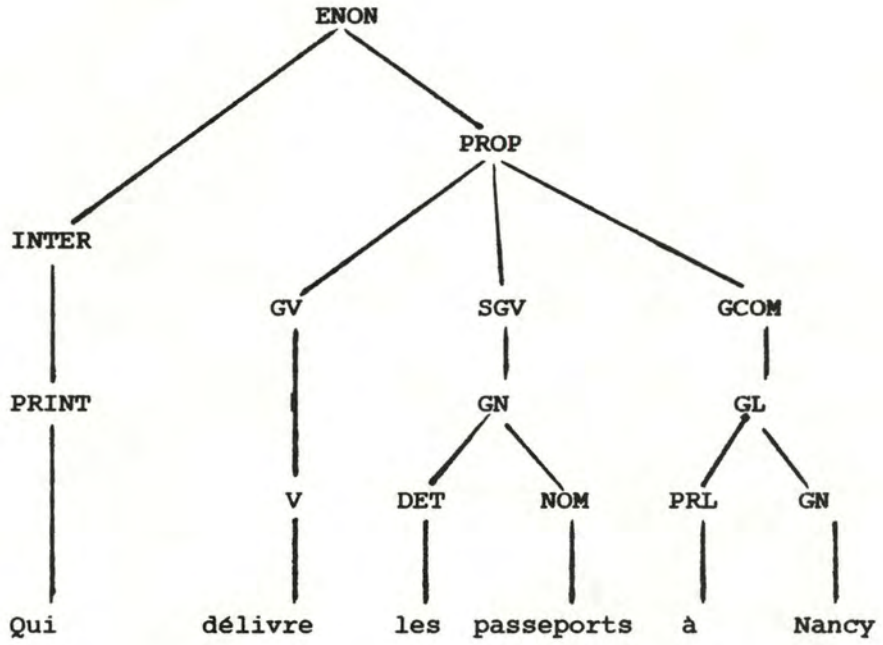
sera représenté en mémoire de travail par

```
prédicat(majeur)
patient(majeur,je)
adjectif(majeur)
pronom(je)

présent(majeur)
active(majeur)
indicative(majeur)
déclarative(majeur)
positif(majeur)
```

De même, l'énoncé suivant :

"Qui délivre les passeports à Nancy ?"

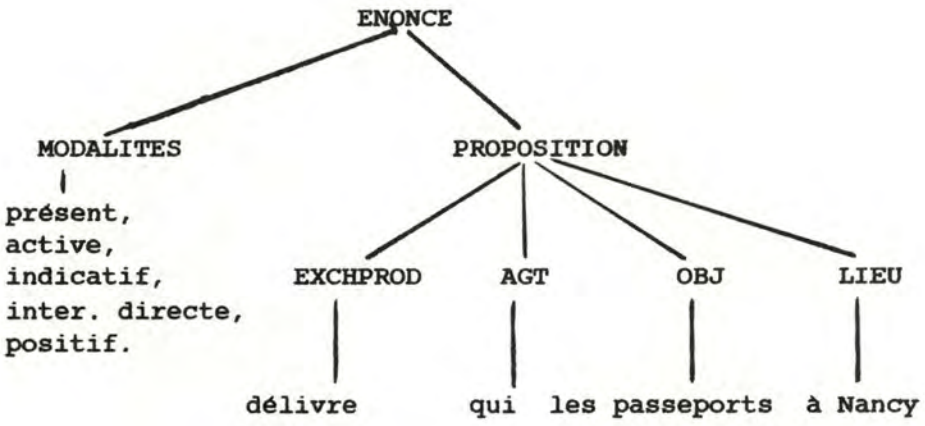
représenté par



ENON = énoncé global
INTER = groupe interrogatif
PRINT = pronom interrogatif
PR = proposition
GV = groupe verbal
SGV = suite verbal
NOMP = nom propre
GCOM = groupe complément
GN = groupe nominal
GL = groupe circonstanciel lieu
PRL = préposition de lieu
DET = déterminant
NOM = nom
V = verbe

comme représentation syntaxique,

et également représenté par



où EXCHPROD (échange production) est la primitive
réferant aux verbes transitifs qui décrivent
une ACTION impliquant au moins 3 entités
où l'une est physiquement ou mentalement
transférée d'une entité animée à
une autre entité animée ;

AGT (agent) est le cas correpondant à une entité
animée contrôlant une ACTION ;

OBJ (objet) est le cas correspondant à l'entité
animée ou non, affectée par une ACTION (ou un
PROCESSUS) ou impliquée dans un ETAT ;

LIEU (lieu) est le cas désignant la dimension spatiale
d'une entité dans un ETAT, ou la dimension
spatiale d'une ACTION ou d'un PROCESSUS.

comme représentation sémantique,

sera représenté de manière interne comme

```
prédicat(délivre)
exch_prod(délivre)
agent(délivre,qui)
objet(délivre,passeport)
lieu(délivre,Nancy)
pron_int(qui)
nom(passeports)
nom(Nancy)

présent(délivre)
voix_active(délivre)
indicatif(délivre)
interr_directe(délivre)
positive(délivre)
```

REMARQUE : il faut souligner que les propositions sont désignées par le "noeud" dont elles dépendent, c'est-à-dire leur prédicat. Ainsi, toutes les informations portant sur ces propositions et représentées par des instances de relations ont ce prédicat comme argument.

3.1.2.3 L'historique du dialogue.

Comme on l'a vu, l'historique du dialogue est utile dans l'interprétation au moins à 2 niveaux :

[1] il apporte des informations sur le type d'énoncés attendus. Suivant le dernier échange locuteur/machine, il est possible de prévoir le genre d'énoncé qui va suivre.

exemple : machine :"êtes-vous majeur ?"
[?majeur(locuteur)]

on s'attendra à une réponse du genre de :

"oui ", "bien-sûr", "évidemment", etc...

et il est probable que l'interprétation de l'énoncé suivant soit :

```
majeur(locuteur)          (cas réponse affirmative)
~majeur(locuteur)         (cas réponse négative)
```

Ces informations, ayant force d'hypothèses, peuvent éventuellement être utilisées par l'interpréteur.
Ells seront exprimées en mémoire de travail sous forme d'instances de relations prédéfinies.

```
exemple : question(dernier_énoncé_machine)

          /* le dernier énoncé de la machine était une
          /* question.

          réponse_simple(énoncé_attendu)

             /* l'énoncé attendu est de type réponse
             /* simple.

          etc...
```

[2] Il apporte des éléments d'interprétation contextuelle en cas de références et de réponses stéréotypées.
Ces informations ne seront pas présentes en mémoire de travail, mais accessibles via des actions spécialisées (voir règles d'interprétation).

3.1.2.4 Mots du lexique et concepts: les faits.

L'interprétation contextuelle doit immanquablement s'appuyer sur une certaine connaissance des mots du lexique et des concepts de l'application.
De manière plus précise, ce sont les traits sémantiques se rapportant aux uns et aux autres qui peuvent être à la base de leur mise en correspondance dans le cadre de l'énoncé courant.
Ces traits sémantiques seront considérés, dans notre système, comme des faits. L'ensemble de ces traits sémantiques constituera donc la base de faits.

Un fait sera représenté par une instance de relation finie.

exemple :

```
schémas : état(v)
              /* signifie que le verbe v exprime un
              /* état.

          lieu(l)
             /* signifie que le mot du lexique l
             /* correspond à un lieu.

          personne(t)
             /* signifie que le mot du lexique t
             /* représente une personne

faits :   état(être)
          lieu(Nancy)
          personne(Pierre)
          lieu(bureau)

              etc...
```

Il est important de bien voir que les faits correspondent à des connaissances statiques, "à long terme", par opposition aux connaissances de la mémoire de travail qui sont dynamiques, propres à un énoncé particulier.
C'est essentiellement cette distinction qui justifie leur représentation séparée au sein du système. Une représentation commune peut s'avérer dangereuse à bien des égards.

ex: prenons le verbe "avoir". Suivant le contexte, on peut lui associer deux traits sémantiques incompatibles au sens de l'application.

requête(avoir): Il s'agit d'une requête au sens de l'application ("obtenir").

possession(avoir) : Il s'agit d'une information au sens de l'application.

Ces 2 traits sémantiques existent au niveau de la base de faits. Un seul sera vérifié au niveau de l'énoncé. Il est facile d'imaginer l'état de la base de faits si, pour un énoncé, on déduit la négation de l'un des 2, puis met la base de faits à jour...!
De plus, dans un certain contexte, un verbe peut avoir un sens bien spécifique. Si on veut pouvoir déduire ce genre d'informations, il ne faut pas qu'elle rentre en concurrence avec la signification générale de ce terme.

3.1.2.5 Informations déduites par les règles.

Comme on le verra plus loin, l'exécution de certaines actions de règles permet de générer de nouvelles informations sur l'énoncé. Ces informations, de par leur définition, sont propres à l'énoncé courant, et seront donc contenues dans la mémoire de travail.
Elles seront également représentées sous forme d'instances de relations finies.

3.1.2.6 Les règles d'interprétation.

Les règles d'interprétation renferment les morceaux de connaissance portant sur la façon de réaliser l'interprétation des énoncés. Conformément à la philosophie des règles de production, elles sont indépendantes, c'est-à-dire qu'il n'existe pas de liens explicites entre elles.

De manière générale, une règle est un couple (C,A) où C est un ensemble non vide de conditions, et A un ensemble non vide d'actions.
Ces 2 ensembles sont reliés par le symbole "=>".

$$(c\_1)\ \&\ (c\_2)\ \&...\&\ (c\_n) \Rightarrow (a\_1)\ \&\ (a\_2)\ \&...\&\ (a\_m)$$

$(n>=1)$ $(m>=1)$

Il s'agit de règles de production, c'est-à-dire que le symbole "=>" signifie :

SI les conditions c_1,...,c_n sont simultanément vérifiées,
i.e. si la situation qu'elles caractérisent est détectée,

ALORS les actions a_1,...,a_m PEUVENT éventuellement être réalisées.

[1] Une condition est un schéma de relation.

ex: prédicat(#x)
personne(#y)

l'ensemble des conditions d'une règle constitue sa prémisse.
Nous verrons dans la partie consacrée à l'interpréteur la façon dont une prémisse est évaluée.

[2] Une action est une expression de la forme :

<id_act> (<arg_1>,...,<arg_n>)

où <id_act> est une chaine de caractères identifiant une des actions reconnues par l'interpréteur,
et où <arg_1>,...<arg_n> est la suite d'arguments de cette action.
Le nombre ainsi que le type de ces arguments sont fixés de manière statique en fonction de l'action envisagée.

Les différentes actions possibles sont :

[1] ajouter(x) ;

[2] supprimer(x) ;

[3] en_machine(e) ;

[4] hist-référé( ) ;

[5] interpreter(x) ;

[6] ajout_int(x).

Voyons à quoi ces actions correspondent.

[1] ajouter(x).
Cette action a pour effet de rajouter l'instance de relation x à l'ensemble des instances de relations déjà présentes dans la mémoire de travail, si elle ne s'y

trouve pas encore.
Si elle s'y trouve déjà, elle n'a alors aucun effet.

[2] supprimer(x).
Elle a pour objectif de supprimer l'instance de relation x de la mémoire de travail si elle s'y trouve. Si elle ne s'y trouve pas, elle n'a alors aucun effet.

Ces 2 premières actions correpondent en quelque sorte à des opérations de déduction.

[3] en_machine(e).
Cette action a pour effet d'accéder à l'historique du dialogue, d'en extraire le dernier énoncé de la machine (qui doit nécessairement correspondre à une question), et de remplacer dans cet énoncé le point d'interrogation matérialisant l'objet de la question (et son éventuel qualificatif) par l'interprétation du mot terminal du lexique représenté par la variable e. Si cette variable est précédée d'un guillemet ", elle doit être considérée comme une constante et c'est la chaine e elle-même qui devra être placé dans l'énoncé.
Dans un cas comme dans l'autre, le résultat ainsi obtenu est chargé dans la structure de données correspondant au résultat de l'interprétation.
Cette action contient également un ordre implicite de fin d'exécution, l'énoncé étant complètement interprété.

exemples :

1) dernier énoncé machine : aller(locuteur,?lieu)

l'exécution de

en_machine(#x) où #x = "Nancy"

chargera dans la structure de données correspondant au résultat de l'interprétation :

aller(locuteur,"Nancy")

2) dernier énoncé machine : ?majeur(locuteur)

l'exécution de

en_machine("~)

chargera dans la structure de données :

~majeur(locuteur)

ce qui correspond à l'interprétation d'une réponse négative de la part du locuteur.

[4] hist_référé( ).
L'objectif de cette action sans paramètre est d'accéder à tous les groupes nominaux du dernier échange locuteur/machine susceptibles d'être désignés par les références pronominales contenues dans l'énoncé courant, et de les charger en mémoire de travail.
La détermination de la bonne référence doit être faite par l'exécution de règles appropriées au sein de l'interpréteur lui-même.

exemple :

échange précédent :

locuteur: "je veux une carte d'identité"
machine : "êtes-vous majeur ?"

le résultat de l'exécution de :

hist_référé( )

sera de rajouter en mémoire de travail les relations:

référé(carte d'identité)
référé(majeur)

[5] interpréter(x).
Cette action correspond à un ordre d'interprétation.
Un ordre d'interprétation est un appel récursif à l'interpréteur sur la sous-structure de l'énoncé passée en paramètre.
La sous-structure est soit un terminal pour lequel il faut trouver l'équivalent au niveau des concepts, soit une structure plus complexe (par exemple, un prédicat et sa structure de cas). Dans ce dernier cas, la variable correspondant au point central de la structure (le prédicat) sera alors passée en paramètre. Il est donc nécessaire de différencier un ordre d'interprétation sur un prédicat en tant que terminal avec un ordre d'interprétation sur la structure à laquelle il correspond. Pour ce faire, dans le second cas, on précédera le nom de la variable à laquelle il est lié du symbole "s/" précisant ainsi qu'il s'agit bien de la sous-structure dont il est élément principal.
Quels que soient les paramètres, l'interpréteur n'aura

plus accès qu'au sous-ensemble des informations contenues dans la mémoire de travail et portant sur ces paramètres.
Il faut bien voir que ce mécanisme dirige en quelque sorte l'interpréteur vers l'interprétation de cette sous-structure, en lui évitant ainsi de se disperser trop dans sa recherche d'une solution. Ceci correspond bien à ce que nous voulions intégrer dans le système de production (voir plus haut - choix d'un type de solution).
Enfin, un ordre d'interprétation contient un ordre implicite de fin d'exécution qui sera exécuté lorsqu'il aura terminé sa tâche.

[6] ajout_int(x).
Cette dernière action a pour fonction de rajouter à la fin de la structure de données correspondant à l'interprétation d'une partie de l'énoncé soit la constante x (dans ce cas, x est précédé d'un guillemet "), soit la valeur de la variable x.
Dans le premier cas, il s'agira soit d'un concept de l'application, soit d'un élément prédéfini intervenant dans la formulation de l'interprétation de l'énoncé (but,information,question, "(",")",",", ET logique, OU logique), soit d'une formule. Dans le second cas, il s'agira d'une constante présente dans l'énoncé et invariante en ce sens qu'il ne lui correspond pas d'interprétation.

exemple : "Nancy"

Les divers constituants possibles des règles étant définis, nous pouvons maintenant les illustrer sur des exemples simples.

exemples :

1) Si prédicat(#x)
& objet(#x,#o)
& requête(#x)
& obligation(#x)

=> ajouter(requête(#x))
& ajouter(objet_requête(#o)) ;

signifie que si on a un prédicat qui exprime une une requête dans un énoncé ayant une modalité d'obligation ; que la primitive dont il est dérivé possède un cas objet qui peut être mis en correspondance avec un élément de l'énoncé, alors on peut déduire que la proposition

exprime une requête, et que l'objet exprime
l'objet de cette requête.

2) SI prédicat(#x)
& prédicat(#y)
& objet(#x,#y)
& requête(#x)
& objet_requête(#y)

=> ajout_int("but:)
& interpréter(s/#o) ;

signifie que si on a un prédicat dont la primitive
admet un objet dans sa structure de cas, et
que l'on a pu le mettre en correspondance avec
un élément de l'énoncé, élément qui est
lui-même un prédicat ; que le premier exprime une
requête et le second un objet de requête, alors
ajouter "but:" à la partie d'interprétation réalisée
et appel récursif à l'interpréteur sur la sous-
structure dépendant du second prédicat.

3) SI prédicat(#x)
& objet_requête(#x)
& exchobt(#x)

=> ajout_int("obtenir)

signifie que si on a un prédicat exprimant un objet
de requête, et que ce prédicat est dérivé
de la rpimitive "exchobt", il s'interprète
comme le concept "obtenir".
(règle peut-être un peu trop simple...).

3.1.3 L'interpréteur de règles.

Le rôle de l'interpréteur est, en un mot, d'utiliser les règles d'interprétation, la base de faits, l'historique du dialogue et la mémoire de travail pour établir l'interprétation contextuelle de l'énoncé.

De manière un peu plus précise, il devra remplir au moins les 3 fonctions suivantes :

[1] exécution des règles, c'est-à-dire :

* sélection des règles activables ;
* sélection de la règle à exécuter ;
* exécution de cette règle.

[2] contrôle d'exécution de ces règles ;

[3] et gestion d'une trace d'exécution.

Il faut souligner que ces fonctions sont très liées les unes aux autres.

3.1.3.1 Exécution des règles.

La première étape à réaliser est la sélection des règles activables, c'est-à-dire celles dont la prémisse est vraie.
D'une manière générale, la précondition (prémisse) d'une règle est vraie si et seulement si les conditions qui la composent sont simultanémant vraies.

Une condition C est :

* vraie s'il existe un jeu de substitution des arguments variables du schéma de relation C par des mots du lexique, jeu tel que l'instance de C ainsi obtenue soit présente en mémoire de travail ou en base de faits.

* fausse sinon.

exemple :

informations en mémoire re travail :

```
prédicat(vouloir)
prédicat(avoir)
```

informations en base de faits :

```
document(passeport)
personne(je)
volonte(vouloir)
```

les conditions suivantes sont vraies :

```
prédicat(#x)          avec #x="avoir"
prédicat(#x)          avec #x="vouloir"
document(#x)             avec #x="passeport"
personne(#x)             avec #x="je"
```

les conditions suivantes sont fausses :

```
prédicat(#x)          avec #x="passeport"
personne(#x)             avec #x="avoir"
lieu(#x)
ville(#x)

etc...
```

Sur base de cette définition, on dira que la prémisse d'une règle est :

* vraie s'il existe un jeu de substitution unique des arguments variables des différents schémas de relations par des mots du lexique de l'application, jeu tel que toutes les conditions soient simultanément vraies.
  Ce jeu de substitution doit être unique, c'est-à-dire qu'il doit attribuer la même valeur à chaque variable de même identificateur.

* fausse sinon.

exemple :

informations en mémoire de travail :

```
modalité_volonté(vouloir)
prédicat(vouloir)
objet(vouloir,avoir)
prédicat(avoir)
groupe_nominal(carte_identité)
```

la précondition suivante est vraie :

```
  prédicat(#x)
& objet(#x,#o)
& prédicat(#o)
```

avec le jeu de substitution : { #x="vouloir",
#o="avoir" } ;

la précondition suivante est fausse :

```
  prédicat(#x)
& objet(#x,#o)
& groupe_nominal(#o)
```

car il n'existe aucun jeu de substitution unique rendant les conditions simultanément vraies.

Il faut ensuite déterminer, parmi l'ensemble des règles instanciées (i.e. activables), celle qui va être exécutée.

Plusieurs critères peuvent être utilisés. Nous avons retenus les suivants, dans l'ordre :

[1] un critère de généralité, privilégiant les règles qui sont des cas particuliers d'autres.
Une règle est dite plus générale qu'une autre si et seulement si sa prémisse est strictement incluse dans la prémisse de la seconde.
Ce critère nous a semblé valable dans la mesure où il parait logique d'exécuter des règles plus précises, et donc correspondant mieux à la situation dans laquelle on se trouve.

[2] un critère de plus forte contrainte, donnant la préférence aux règles dont le nombre de condition est le plus élevé.

[3] un critère d'âge, donnant la préférence aux règles les plus jeunes dans le conflict set, c'est-à-dire celles instantiées par le fait de la déduction d'informations récentes, et donc à priori plus pertinentes.

[4] enfin, s'il reste des règles concurrentes, la première règle est sélectionnée arbitrairement.

Pour terminer, les actions de la règle sélectionnée doivent être exécutées de manière séquentielle et de gauche à droite.

3.1.3.2 Gestion de la trace d'exécution.

Le but de cette deuxième fonction de l'interpréteur est de tenir à jour un suivi des différentes règles qui ont été exécutées, ainsi que des jeux de substitution qui les ont rendues activables.

Cette fonction se justifie au moins à 2 égards :

[1] au niveau de l'interpréteur, pour détecter, autant que faire se peut, un éventuel cyclage dans l'exécution des règles.

[2] Au niveau de l'utilisateur, afin de lui permettre de suivre et de comprendre le cheminement réalisé par le programme pour arriver à la solution ou à un éventuel échec.

3.1.3.3 Contrôle d'exécution.

L'objectif de cette dernière fonction est de générer les hypothèses d'interprétation à partir de l'état de la dernière phase de dialogue, de traduire l'énoncé, et d'assurer le contrôle des itérations de la fonction d'exécution des règles, jusqu'à ce que l'on soit :

[1] bloqué parce qu'il n'existe plus de règles activables ;

[2] ou arrivé à la fin de l'interprétation de l'énoncé courant.

Cette fonction comprend entre autres la reprise en cas d'échec causé par les hypothèses.
Elle utilise le préprocesseur d'énoncé et le générateur d'hypothèses.

3.1.4 Classification des règles d'interprétation.

D'un point de vue théorique, il n'existe qu'un seul type de règles. Cependant, d'un point de vue opératoire, il peut être utile d'en distinguer plusieurs, et ce en fonction des actions qu'elles réalisent.

Nous proposons d'organiser les règles suivant qu'il s'agit

[1] de règles d'identification ou de déduction ;

[2] ou de règles d'interprétation proprement dite.

3.1.4.1 Règles d'identification.

Nous entendons par règles d'identification toute règle dont la partie "actions" contient exclusivement des accès à la mémoire de travail. Il s'agit donc, en quelque sorte, de règles qui dégagent des informations de l'énoncé, c'est-à-dire qui déduisent un certain nombre d'informations à partir de celles que l'on connait déjà, en vue de le caractériser de manière un peu plus précise.

```
exemple : Si prédicat(#x)
            & objet(#x,#o)
            & requête(#x)
            & obligation(#x)

          => ajouter(requête(#x))
           & ajouter(objet_requête(#o)) ;
```

signifie que si on a un prédicat qui exprime une une requête dans un énoncé ayant une modalité d'obligation ; que la primitive dont il est dérivé possède un cas objet qui peut être mis en correspondance avec un élément de l'énoncé, alors on peut déduire que la proposition exprime une requête, et que l'objet exprime l'objet de cette requête.

3.1.4.2 Règles d'interprétation proprement dite.

Nous entendons par règles d'interprétation proprement dite toute règle dont la partie "actions" contient une ordre d'interprétation, éventuellement couplé à un accès à l'historique du dialogue, et/ou à un accès au résultat partiel de l'interprétation.

exemples :

```
1)   SI  prédicat(#x)
       & prédicat(#y)
       & objet(#x,#y)
       & requête(#x)
       & objet_requête(#y)

     =>  ajout_int("but:)
       & interpréter(s/#o) ;
```

signifie que si on a un prédicat dont la primitive admet un objet dans sa structure de cas, et que l'on a pu le mettre en correspondance avec un élément de l'énoncé, élément qui est lui-même un prédicat ; que le premier exprime une requête et le second un objet de requête, alors ajouter "but:" à la partie d'interprétation réalisée et appel récursif à l'interpréteur sur la sous-structure dépendant du second prédicat.

```
2)  SI  prédicat(#x)
      & objet_requête(#x)
      & exchobt(#x)

    => ajout_int("obtenir)
```

signifie que si on a un prédicat exprimant un objet de requête, et que ce prédicat est dérivé de la rpimitive "exchobt", il s'interprète comme le concept "obtenir".
(règle peut-être un peu trop simple...).

```
3) SI  question(dernier_énoncé_machine)
     & réponse_simple(énoncé_attendu)
     & structure_stéréotypée(#y)
     & réponse_simple(#y)
     & négative(#y)

   =>  ajout_int("information:)
       & en_machine("~) ;
```

signifie que si le dernier énoncé de la machine était une question demandant une réponse simple, et que l'énoncé en entrée correspond à une structure stéréotypée exprimant une réponse simple (du genre "oui", "non", "c'est ça",etc...), et enfin que cet énoncé est négatif, l'interprétation qui lui correspond est une information dont la valeur est la négation du dernier énoncé machine.

3.1.4.3 Intérêt de la distinction.

L'intérêt de la distinction est, il faut le répéter, uniquement opératoire.
Elle permet de scinder la phase d'interprétation en 2 étapes :

[1] dans un premier temps, on rassemble le maximum d'informations concernant l'énoncé en entrée, ces informations étant déduites par des règles d'identification ;

[2] dans un second temps, on utilise ces informations pour déterminer le type général d'énoncé auquel on a affaire, puis on l'interprète suivant un schéma déterminé. Ce sont les règles d'interprétation proprement dite qui, d'une part, dans leur partie "conditions", spécifient le type d'énoncé, et d'autre part, dans leur partie "actions", imposent leur schéma d'interprétation.

Cela revient donc à déterminer 2 contextes, possédant chacun ses règles propres.
Dans la première étape, une certaine souplesse est admise, le but étant de récolter un maximum d'informations. Le choix des règles à exécuter, ainsi que l'ordre dans lequel elles le seront, est assez peu limité.
Dans la seconde étape cependant, il faut que l'interprétation se fasse de manière unique en fonction du type général d'énoncé, et suivant un schéma bien défini.

L'avantage de ce mode d'organisation est donc triple :

[1] il permet d'abord d'ordonnancer l'approche générale du problème en la découpant en 2 étapes séquentielles ;

[2] il permet ensuite de garder un maximum d'ouverture et de souplesse pour l'analyse des énoncés, où il n'existe pas de règles simples systématiques.

[3] il offre enfin la rigidité voulue au moment de l'interprétation proprement dite, en permettant la décomposition en sous-tâches à exécuter séquentiellement via les appels récursifs matérialisés par les ordres d'interprétation.

C'est ici que réside, à notre avis, l'intérêt majeur du système proposé. En effet, l'opération délicate est, nous semble-t-il, l'analyse de l'énoncé. Une fois que celui-ci est identifié à un type général, les règles à appliquer devraient être, dans la plupart des cas, relativement

systématiques.

3.2 Présentation du sous-système réalisé.

Vu le temps dont nous disposions, nous n'avons pas été à même de réaliser le système complètement. Nous nous sommes attaché au développement d'un sous-système utile dont la présentation fait l'objet de cette section.
Nous insistons sur le fait que l'objectif de cette présentation n'est pas de le définir de manière formelle et précise, mais bien de donner au lecteur une bonne idée de ce en quoi il consiste. Les spécifications complètes et précises, les textes des programmes, et l'un ou l'autre exemple d'exécution se trouvent en annexe.

Le sous-système utile s'articule principalement autour de l'interpréteur de règles. Les interfaces nécessaires dont on a parlé à la section 3.1.1 ont été quelque peu laissés de côté.

Il se compose actuellement de 3 programmes :

[1] "gestion_br" est un programme élémentaire permettant l'écriture et la compilation des règles d'interprétation. De nature beaucoup plus simple que les outils prévus, il s'est avéré cependant indispensable pour le développement et le test de l'interpréteur. A terme, il est donc appelé à être remplacé par l'outil d'aide à l'écriture des règles et le compilateur prévus.

[2] "gestion_bf" est le programme qui joue provisoirement le rôle des outils d'édition et de compilation des faits présentés plus haut. Il est également appelé à être remplacé ultérieurement.

[3] enfin, "interp" est le programme qui correspond à l'interpréteur de règles. Mis à part l'un ou l'autre point, il respecte la définition qui en a été faite en section 3.1.

3.2.1 Le programme gestion_br.

Ce programme assure provisoirement la gestion de la base de règles. Il permet, via un menu, d'accéder aux fonctions suivantes :

[1] insertion de règles dans la base.

Cette fonction permet d'introduire une ou plusieurs règles dans la base. Elles sont introduites sous une forme externe appréhendable par l'homme et sont immédiatement traduites en représentation interne.
Une vérification de cohérence est assurée au niveau de l'introdution des conditions (une au moins) et de l'introduction des actions (au moins une également, vérification de leur existence et de la validité de leurs paramètres).

[2] Supression d'une règle de la base.

Etant donné un identificateur de règle, cette fonction permet de supprimer de la base la règle qui lui correspond.

[3] Modification d'une règle.

De nature très élémentaire, cette fonction permet à l'utilisateur de modifier une règle, c'est-à-dire de la supprimer de la base et d'en introduire une autre correspondant à sa version modifiée.

[4] Affichage du contenu de la base.

Cette fonction permet de consulter à l'écran l'ensemble des règles sous forme externe ainsi que leur identificateur.

[5] Impression du contenu de la base.

Elle simule l'impression sur papier des règles contenues dans la base.

[6] Destruction de la base.

Cette fonction permet de détruire complètement la base de règles.

3.2.2 Le programme gestion_bf.

Ce programme assure temporairement la gestion de la base de faits. Il permet, via un menu également, d'accéder aux fonctions suivantes :

[1] Insertion de faits dans la base.

[2] Supression d'un fait de la base.

[3] Modification d'un fait de la base.

[4] Affichage du contenu de la base.

[5] Impression du contenu de la base.

[6] Destruction de la base.

Outre le fait qu'elles travaillent sur des faits et non des règles, ces fonctions ont la même définition que leur homologue du programme de gestion de la base de règles.

3.2.3 Le programme interp.

"interp" est le programme qui correspond à l'interpréteur de règles.
Il respecte la définition qui en a été faite au moment de la présentation générale du système, sauf au niveau :

* du préprocesseur d'énoncé qui n'est que simulé. C'est en effet à l'utilisateur d'introduire de manière interactive la représentation de l'énoncé dans la mémoire de travail.

* du générateur d'hypothèses, qui est prévu mais non implémenté.

* de certaines fonctions qui n'étaient pas indispensables au fonctionnement du sous-système utile, et dont nous reparlerons plus loin.

En s'appuyant sur la base de règles et la base de faits qui peuvent être développées par les 2 programmes précédents, il réalise donc :

[1] l'interprétation proprement dite des règles ;

[2] le contrôle d'exécution ;

[3] la gestion d'une trace d'exécution.

Suivant l'option qu'il prend, l'utilisateur peut suivre son fonctionnement en interactif, ou disposer de la trace d'exécution dans un fichier annexe.

3.2.3.1 Interprétation des règles.

L'interprétation proprement dite des règles revient à répéter les 3 fonctions suivantes :

[1] sélection du conflict set.

Le conflict set est la liste des instances de règles qui sont activables à un moment donné. Une instance de règle est un couple (R,S) ou R est une règle et S un jeu de substitution attribuant une valeur déterminée à

chaque variable de la règle.
La construction de ce conflict set n'est pas réalisée en parcourant exhaustivement la base de règles et en vérifiant si chaque règle est activable, mais bien en s'appuyant sur sa version du cycle précédent (s'il existe) et sur les éléments qui viennent d'être rajoutés en mémoire de travail.
Le conflict set du cycle c est donc construit à partir de celui du cycle c-1.
D'abord, en sont extraites toutes les règles dont une ou plusieurs conditions ne sont plus vérifiées en raison de l'ajout d'un élément en mémoire de travail.
Ensuite, en partant des "nouveaux" éléments de mémoire de travail, la prémisse des règles de la base de règles ayant au moins une condition portant sur l'un de ces éléments est évaluée. Les règles dont la prémisse s'est avérée vraie sont alors rajoutées au conflict set restant.
Cette stratégie, évitant le parcours exhaustif de la mémoire de travail, de la base de règles, et de la base de faits, est motivée par les réflexions suivantes :

[1.1] Une règle activable au cycle c-1 reste activable au cycle c si et seulement si aucune de ces conditions n'est rendue fausse par un des éléments rajoutés en mémoire de travail.

[1.2] Les seules règles non activables au cycle c-1 qui ont une probabilité non nulle d'être activables au cycle c sont celles dont une condition au moins porte sur un élément qui vient d'être rajouté en mémoire de travail.

Enfin, un filtre se basant sur une trace d'exécution est implémenté afin d'extraire du conflict set les instances de règles qui ont déjà été activées. Ceci est bien évidemment prévu pour éviter un cyclage éventuel.

[2] Conflict resolution.

Etant donné l'ensemble des instances de règles activables à un moment donné, un choix doit être fait pour déterminer celle qui va être activée.
Les statégies utilisées pour ce choix sont celles qui ont été présentées ci-dessus, à l'exception du critère d'âge qui reste prévu, mais non implémenté.

[3] Activation des règles.

L'instance de règle sélectionnée est ensuite activée. Ses différentes actions sont exécutées séquentiellement de gauche à droite. Seule l'action de suppression d'un

élément en mémoire de travail n'a pas été implémentée. Les actions d'accès à l'historique du dialogue pour la résolution des anaphores et des ellipses sont simulées par l'intervention de l'utilisateur en interactif.

3.2.3.2 Contrôle d'exécution.

Le préprocesseur d'énoncé et le générateur d'hypothèses n'ayant pas été implémentés, le contrôle d'exécution revient simplement pour le moment au lancement de l'interpréteur et à l'affichage des messages appropriés suivant les résultats de son exécution.

3.2.3.3 Trace d'exécution.

La gestion de la trace d'exécution permet d'une part d'éviter le cyclage provenant de l'activation de la même instance de règle, et d'autre part de fournir les informations nécessaires au suivi de l'exécution.

Un filtre est implémenté pour assurer le premier objectif, filtre qui supprime du conflict set toutes les instances de règles "dangereuses".

Pour le deuxième objectif, les informations recueillies portent sur l'activation des instances de règles et l'exécution de chacune de leurs actions : règle courante à activer, jeu de substitution, action courante à exécuter, diagnostic d'exécution, etc...
Ces informations, présentées de manière particulière, sont soit affichées à l'écran, soit transférées dans un fichier annexe.

4 Evaluation et développements ultérieurs.

Il est actuellement trop tôt pour pouvoir évaluer complètement la solution adoptée. En effet, c'est au moment du développement et de la mise au point du modèle d'interprétation proprement dit que se révéleront pleinement ses forces et ses faiblesses.
Nous pouvons cependant déjà mettre en évidence quelques points précis qui nous sont apparus soit avec le recul du temps, soit au moment des tests du systèmes sur des exemples simples. Nous pensons plus particulièrement à une certaine lourdeur du formalisme de relations, aux scores et aux explications.

4.1 Relative lourdeur du formalisme.

Bien que nous en étions déjà conscients au moment de la conception du système, force nous est d'admettre que le formalisme s'est avéré quelque peu lourd à manipuler au moment de l'écriture des règles qui ont sous-tendus les tests effectués.
Cette lourdeur provient essentiellement de 2 choses: d'une part, le fait que les relations s'expriment "en français" de manière non abrégée, et d'autre part le fait que la représentation est "horizontale", par opposition à des représentations "arborescentes" comme celles utilisées pour exprimer la structure syntaxique et sémantique de l'énoncé.

Le premier point ne nous paraît pas être un problème en soi. En effet, le concepteur a la liberté de définir les relations qu'il désire mettre en oeuvre, à l'exception de celles qui représentent l'énoncé en entrée et les référés des pronoms. Ces dernières devraient en effet être arrêtées une fois pour toutes. Nous avons opté pour un mode d'expression très significatif afin de garder beaucoup de clarté dans l'énoncé des règles ; nous aurions aussi bien pu identifier les relations par des numéros. Nous estimons qu'une solution satisfaisante peut être trouvée entre ces deux extrêmes. Le choix reste de toute manière toujours possible.

Le second point nous semble par contre plus gênant.
D'abord, le fait d'avoir réduit la représentation à une seule dimension nous force à utiliser beaucoup de relations pour exprimer une structure qui pourrait être représentée plus succinctement. Les exemples donnés ci-dessus pour

illustrer la représentation de l'énoncé en entrée sont assez révélateurs de cet inconvénient qui rejaillit inévitablement sur le reste du système. Peut-être faudrait-il faire quelque chose à ce sujet. Il nous semble cependant qu'une étude préalable s'impose. En effet, nous ne sommes pas entièrement convaincus que ce soit très fondamental dans le cadre qui nous intéresse. A part peut-être les premières répliques, les interventions constituant un dialogue finalisé semblent être assez courtes et d'une structure relativement simple. Si le formalisme dans lequel elles sont représentées peut alourdir dans une certaine mesure le système lorsque celles-ci sont complexes, la répercussion est moins forte quand elles sont plus simples.
D'autre part, le fait d'avoir "applati" toute représentation nous oblige ensuite à "situer" tout élément subordonné par rapport aux éléments centraux de la structure dans l'énoncé des préconditions. Ceci est surtout vrai pour les règles d'identification qui travaillent par définition sur l'énoncé complet. Au moment des ordres d'interprétation, les seules informations disponibles sont celles qui se rapportent à la structure passée en paramètre. Dans ce cas, cette dernière remarque se justifie moins.

4.2 Mécanisme d'explication.

Actuellement, la compréhension du fonctionnement du système ne peut s'appuyer que sur les informations fournies par la trace d'exécution. Il peut être utile de développer un mécanisme d'explication des résultats. Il peut s'avérer très intéressant, tant pour la mise au point du modèle d'interprétation que pour le suivi de son fonctionnement proprement dit.
Nous pensons tout particulièrement à l'explication du "pourquoi" et du "comment" des résultats.
Le "pourquoi" doit pouvoir éclairer le modélisateur ou l'utilisateur sur les raisons pour lesquelles le système est arrivé à une telle interprétation de l'énoncé.
Le "comment", lui, correspond à l'explication de la façon dont l'interpréteur a abouti à ce résultat.
Nous pensons qu'il y a place ici pour un développement ultérieur.

4.3 Les scores.

Les représentations de l'énoncé sont actuellement considérées dans le système comme exactes à 100%. De même, les déductions et les morceaux d'interprétation réalisés par l'interpréteur ne souffrent aucune nuance.

Ceci n'est peut-être pas très réaliste.
Sans doute faudrait-il intégrer un mécanisme d'évaluation qui, combinant les scores des représentations en entrée avec ceux des déductions et des interprétations, attribue au résultat de l'interpréteur une certaine pondération.
Un tel mécanisme n'est cependant pas simple à mettre en oeuvre. En effet, la pondération à donner aux conséquents des règles d'interprétation n'est déjà pas facile à évaluer. On imagine par conséquent la difficulté de mise en oeuvre de la propagation des scores de reconnaissance des structures syntaxiques et sémantiques, propagation qui se fait en combinant les résultats de départ avec ces pondérations.

Il nous semble cependant qu'un mécanisme d'évaluation constitue un élément important si l'on désire obtenir un résultat pleinement satisfaisant.

5 Conclusion.

Une de nos contributions aux recherches menées à Nancy s'est située au niveau du développement de l'interprétation contextuelle des énoncés.
Cette fonction établit une représentation opératoire de la signification que prennent ces énoncés dans le cadre du dialogue courant et de la tâche à accomplir.
Pour ce faire, elle doit s'appuyer sur des représentations de l'énoncé et des connaissances prédéfinies. Le module d'analyse syntaxico-sémantique lui fournit une représentation syntaxique et sémantique éventuellement incomplète, ou une chaine lexicale lorsqu'aucune autre structure n'a pu être détectée. Les connaissances à mettre en oeuvre sont de type lexical, syntaxique, sémantico-pragmatique, et pragmatique (historique du dialogue).
Cette fonction doit constituer un modèle d'interprétation des énoncés issus de dialogues oraux finalisés, modèle pragmatique et expérimental en raison de l'absence de théorie générale sur laquelle il pourrait s'appuyer.

Nous avons opté pour le développement d'un outil spécifique reposant sur le formalisme des règles de production et dont l'objectif est de permettre la définition et la mise en oeuvre de ce modèle. En raison de la difficulté du problème et des notions linguistiques qu'il nécessite et que nous ne possédons pas, nous n'avons pas réalisé ce modèle. Nous nous sommes limités à la définition d'un outil que nous voulions robuste et aussi souple et général que possible, et dont nous avons implémenté un sous-système utile.
L'intérêt premier de la solution que nous proposons réside dans le fait qu'elle allie la puissance et la modularité du formalisme des règles de production à des caractéristiques propres au problème de l'interprétation contextuelle.

Il est encore trop tôt que pour évaluer pleinement la solution que nous proposons. Celle-ci ne révélera ses forces et ses faiblesses qu'au moment de la définition et de la mise en oeuvre du modèle d'interprétation, ainsi qu'au cours de son fonctionnement.
Cependant, l'adjonction d'un mécanisme d'explication des résultats et l'introduction de scores semblent être d'ores et déjà des développements ultérieurs particulièrement intéressants.

CHAPITRE V : ASPECTS DE LA COHERENCE DANS UN SYSTEME DE DIALOGUE.

1 Détection d'incohérence dans les règles de réécriture.

1.1 Introduction.

Nous avons avons eu l'occasion souligner le rôle et l'importance de la description de la tâche et de son univers dans le cadre des systèmes experts.
Il importe donc, lors de la création d'un tel système, de pouvoir s'assurer que les connaissances décrites sont de qualité. Ce besoin de qualité justifie l'existence et l'utilisation d'outils permettant à l'expert de valider, dans une certaine mesure, les connaissances qu'il décrit.

La réalisation que nous allons maintenant présenter, est un outil dont l'objectif est de répondre, au moins partiellement, à ce besoin.

Rappelons brièvement les trois composants principaux qu'intègrent la plupart des systèmes experts.

[1] Une base de connaissance rasssemblant une certaine représentation de tout le savoir d'un expert humain.
Ce savoir est constitué non seulement de connaissances spécifiques à un domaine précis (détection et diagnostic d'infections sanguines, code de la route, réalisation de configurations informatiques), mais aussi d'un ensemble de connaissances purement logiques et proches du sens commun.

[2] une base de faits contenant les données relatives au domaine exploré et liées à l'exécution en cours.
Son contenu pourrait être l'expression de faits tels que :

- " la température ambiante dans le four est de 100 degrés ",
- " le capitaine est âgé de trente et un ans ",
- " cet animal possède des griffes et des yeux frontaux ".

[3] Un moteur d'inférence, dont le rôle est d'appliquer les connaissances générales du domaine (se trouvant dans le composant décrit en [1]) aux informations particulières contenues dans la base de faits.

Jusqu'à présent, les préoccupations principales des chercheurs en Intelligence Artificielle (I.A.) étaient centrées sur les mécanismes de raisonnement. L'amélioration des performances des systèmes intelligents reposait essentiellement sur ceux-ci, ce qui avait pour conséquence de rendre les mécanismes mis en oeuvre de plus en plus complexes.
Sans doute la qualité du raisonnement joue-t-elle un rôle dans les performances des systèmes. Il n'en reste pas moins que la qualité des structures et des bases de connaissances est également déterminante dans ce domaine.
D'où l'effort de recherche consenti ces dernières années afin de découvrir des modèles performants de représentation.

Les formalismes permettant de représenter les connaissances sont nombreux [Jackson 86]. Citons

- les frames [Minsky 75]
- les règles de production [Davis 77],
- les graphes, arbres et réseaux [Woods 72],
- la logique des prédicats [Kowalski 79].

Parmi tous ces formalismes, la représentation sous forme de règles de production est l'une des plus souvent utilisées, d'une part parce qu'elle est apte à exprimer des informations variées et à traiter des situations complexes et d'autre part parce que leur formalisme est attrayant et relativement simple [Davis op. cit.]

Nous ne reviendrons pas sur la description détaillée de celles-ci car elle a été faite au chapitre précédent. Néanmoins, le chapitre 4 a évoqué, sans le developper, un aspect sur lequel nous souhaiterions revenir : la résolution du conflit entre les différentes règles applicables à un instant donné.

Le choix de la règle à appliquer constitue un problème délicat. En effet, il va influencer toute la suite du déroulement de la recherche de la solution par le système.
C'est pourquoi les chercheurs en I.A. se sont fréquemment penchés sur ce problème. Ils y ont apporté différents types de solutions, chacune d'elles consistant en une stratégie de choix plus ou moins élaborée.
Sans nous étendre longuement sur celles-ci, nous citons les plus fréquentes :

ordonancement des règles : l'ensemble des règles est classé selon une priorité déterminée par le concepteur du système. Lorsque plusieurs règles sont en conflit, celle qui a la priorité la plus importante est appliquée et les autres sont ignorées;

ordonancement des données : un coéfficient d'importance est attribué à chaque aspect des situations décrites, de façon à obtenir une liste de priorités. La règle choisie est celle qui obtient la priorité la plus grande en fonction des aspects de situations qu'elle envisage;

contrainte maximale : la règle appliquée est celle qui est la plus contraignante, ce qui se traduit bien souvent par la liste de prémisses la plus longue;

caractère récent d'utilisation : la règle qui est appliquée est soit celle qui a été la plus récemment appliquée, soit celle qui a été la moins récemment appliquée;

limitation du contexte : les règles sont réparties en sous-groupes dont seuls certains sont activables à un instant donné. Cela a pour effet de réduire fortement le nombre de règles pouvant entrer en conflit;

La réalisation de l'action décrite dans le conséquent peut aller du simple ajout d'un fait dans la base de faits à un traitement beaucoup plus complexe de type procédural.

1.2 Systèmes de déductions.

Supposons que les conditions de nos règles décrivent des combinaisons de faits connus.

Supposons également que leurs actions soient des faits nouveaux à déduire directement des combinaisons de faits des conditions.

Sous ces contraintes, nous pouvons dire que le système de règles de production devient un système de déduction.

Dès lors, les parties "condition" de nos règle peuvent être appelées des antécédents et les parties "action" de ces mêmes règles des conséquents.

Un tel système possède un certain nombre de faits réputés connus et il en déduit un certain nombre de conclusions.

Supposons que nous souhaitions identifier des animaux à l'aide de telles règles.
Nous pourrions écrire une règle qui décrirait chaque animal. Les antécédents de chaque règle énuméreraient les caractéristiques d'un animal et les conséquents donneraient le nom de l'animal répondant à l' ensemble des caractéristiques. Les caractéristiques devraient être suffisament étoffées pour pouvoir rejeter à coup sûr toute identification erronée d'un animal.

Une autre possibilité s'offre. Il s'agirait de créer des règles, mais chaque règle n'aurait plus l'ensemble des caractéristiques d'un seul animal. Elles permettraient plutôt de déduire un certain nombre de caractéristiques intermédiaires afin de procéder selon un mode de raisonnement plus élaboré que la simple comparaison exhaustive d'un grand nombre de caractéristiques.

L'avantage des ces règles est qu'elles permettent d'utiliser des antécédents

- limités quant au nombre de prémisses,
- faciles à comprendre,
- faciles à manipuler.

Dès lors, la procédure de raisonnement crée une chaine de déductions entre les faits de la base de faits.

Un tel système de déduction peut fonctionner selon deux modes. Jusqu'à présent, nous avons supposé implicitement qu'il fonctionnait à partir des faits qu'il connaissait en en déduisant de nouveaux.
Néanmoins, le mécanisme inverse est également possible pour un système de déduction. Il lui est poosible de valider ou d'invalider une hypothèse qui lui est fournie en utilisant les règles pour créer d'autres hypothèses.

Prenons en exemple les règles

R0 : a => b
R1 : x => c
R2 : b => d
R3 : c => e
R4 : d => f

et le seul fait 'a' contenu dans la base de faits.

Vérifier l'hypothèse 'f', revient à vérifier, selon nos règles, l'hypothèse 'd'. De même, vérifier 'd' revient à vérifier 'b' qui, lui-même, revient à vérifier 'a'. Or 'a' est vérifié puisqu'il se trouve dans la base de faits.
Donc notre hypothèse de départ est vérifiée également.

Le même raisonnement pour l'hypothése 'e' nous amènerait à dire que celle-ci n'est pas vérifiée.

Sachant que les deux modes de raisonnement peuvent être appliqués, la question se pose de savoir lequel des deux est préférable.

La réponse à cette question dépend de l'objectif poursuivi.

Si le but est de déduire un maximum d'informations nouvelles à partir d'un ensemble de faits initiaux, il est clair que le chaînage avant sera bien plus performant. Par contre, si le but est la vérification d'une hypothése ou son invalidation, l'utilisation du mécanisme de chaînage avant se solderait par un énorme gaspillage de ressources si le nombre de règles est relativement important et cela à cause de l'explosion combinatoire se produisant lorsque plusieurs règles peuvent s'appliquer simultanément et que différentes possibilités sont à envisager.
Dans ce cas le mécanisme du chaînage arrière sera préféré.

1.3 Objectif poursuivi.

Le but de notre travail est d'aider le concepteur d'une base de connaissances, exprimée sous forme de règles de déduction, à garantir la cohérence des règles qu'il écrit. De nombreux tests, dits de cohérence, peuvent être réalisés sur un tel ensemble de règles.

P. Le Beux [Le Beux 86] nous en cite les principaux :

- vérification d'absence de contradictions entre les règles,
- vérification d'absence de redondances,
- vérification d'absence de bouclages.

En ce qui nous concerne, nous nous somme bornés à vérifier l'absence de contradiction entre les règles de réécriture.

Un ensemble de règles sera considéré comme incohérent s'il permet de générer des faits contradictoires à partir d'un ou plusieurs mêmes faits.

Deux faits sont contradictoires si l'un est la négation de l'autre. Les faits

a et -a

que l'on lit ' a ' et ' non a ', sont contradictoires.

Nous détecterons deux types de contradictions, et donc deux types d'incohérences.

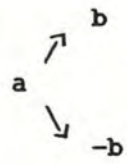
[1] Lorsque l'application successive de plusieurs règles distinctes à un fait initial provoque la réécriture d'un fait contradictoire au fait initial. Ce qui peut se représenter de la manière suivante

a --> b --> -a

C'est à dire que 'a' se réécrit comme 'b', qui lui-même se réécrit comme '-a'. Nous aurions pu écrire ceci de la façon suivante

```
a => b      }
            }       a => -a
b => -a     }
```

[2] Lorsque l'application de plusieurs règles distinctes à un fait initial provoque la réécriture de deux faits contradictoires. Ce qui peut se représenter de la manière suivante :



Cela aurait pu se représenter comme

a => b

a => -b

Nous avons tenu à scinder ces deux cas pour la raison suivante :

La logique classique nous aurait plutôt incités à les confondre, en effet, elle nous enseigne que les expressions

a => b

et

-b => -a

sont équivalentes.

Dès lors, l'incohérence [2]

a => b

a => -b

aurait pu se réécrire

a => b

b => -a

nous ramenant ainsi à l'incohérence [1].

Néanmoins, il nous paraît intéressant de ne pas nous confiner uniquement dans le cadre de la logique pure. En effet, l'utilisation du symbôle '=>' comme connecteur de deux formules ne doit être poussé trop loin en ce sens.

Soit la règle

majeur(x) => -mineur(x)

exprimant que si x est majeur alors x n'est pas mineur.

Ce n'est pas le rôle de la logique d'exprimer que la règle

mineur(x) => -majeur(x)

existe également. Il est du ressort du concepteur du système d'exprimer cette règle s'il estime qu'elle fait partie de la description des connaissances nécessaires.

1.4 Convention d'écriture.

Par la suite, pour la simplicité des écritures et pour faciliter la compréhension, nous prendrons les conventions suivantes :

[1] L'antécédent d'une règle ne contient pas de disjonction. Donc il ne pourra s'y trouver que des propositions élémentaires et des conjonctions (ET).

On ne trouvera donc que des antécédents du type

griffes(x) =>

ou du type

griffes(x) et yeux_frontaux(x) =>

[2] Le conséquent d'une règle ne contiendra que des propositions élémentaires, donc aucune conjonction ni disjonction.

Les conséquents des règles seront donc uniquement du type

carnivore(x)

Ceci n'est guère gênant. En effet :

- nous avons déjà effectué et justifié une restriction du même type au cours du chapitre 4; cette restriction est encore valable ici;

- d'autre part, la suppression de la conjonction dans le conséquent d'une règle ne pose pas ou peu de problèmes. Tout règle contenant une conjonction dans son conséquent peut en effet se reformuler à l'aide de plusieurs règles dont chacune ne contient qu'une proposition élémentaire dans son membre de droite.

Par exemple, la règle

f(a) => g(a) ET h(a)

peut se reformuler à l'aide des deux règles.

f(a) => g(a)
f(a) => h(a)

De façon générale, toute règle possédant la conjonction de n propositions élémentaires dans son membre de droite peut se reformuler à l'aide de n règles. Les membres de gauche (antécédents) sont identiques à ceux de la règle reformulée et les membres de droite sont constitués chaque fois d'une proposition élémentaire du conséquent de la règle qui est reformulée;

- l'absence de disjonction dans l'antécédent des règles peut être contournée de la même manière. En effet, la règle

f(a) OU g(a) OU h(a) => x(a)

peut se reformuler à l'aide des trois règles.

f(a) => x(a)
g(a) => x(a)
h(a) => x(a)

Ceci peut également être exprimé de manière générale en disant que toute règle dont l'antécédent est exprimé à l'aide de la disjonction de m propositions (élémentaires ou conjonctives) peut être reformulée par m règles distinctes.

Leurs conséquents sont identiques à celui de la règle qui est reformulée et leurs antécédents sont constitués de chacune des m propositions qui constituaient la disjonction de la règle initiale.

1.5 Présentation des solutions possibles.

Nous allons présenter ici deux solutions envisageables pour le problème de la cohérence de règles de déduction. L'une est tirée d'un formalisme de représentation des règles en un réseau et qui est présentée dans [Le Beux 86]; l'autre est issue de nos propres travaux.

1.5.1 Représentation des règles sous forme de réseau.

1.5.1.1 Introduction.

Ce principe de représentation est basé sur l'observation que bien souvent, le conséquent d'une règle est repris dans l'antécédent d'un certain nombre de règles de la base.

Par exemple

R1 : mammifère(x) ET sabots(x) ==> ongulé(x)
R2 : ongulé(x) ET long_cou(x) ==> girafe(x)

Dès lors, l'idée proposée est de chaîner les règles entre elles en utilisant la relation successeur / prédécesseur.

Soient R1 et R2 deux règles.

On dira que R2 est successeur de R1 si et seulement si l'action de R1 constitue une prémisse de R2.

On dira également que R1 est un prédécesseur de R2.

Cette relation se notera de la façon suivante :

R1 -> R2.

Une règle qui ne possède pas de prédécesseur est dite initiale.

Si ce principe de chainage est étendu à l'ensemble de la base de règles, on obtiendra un réseau de type ET / OU.

Par exemple, avec les règles

R1 : p1 ET p2 ET p3 ==> a1
R2 : a1 ET p4 ==> a2
R3 : p5 ET p6 ==> a2

On obtiendra un réseau tel que celui qui est présenté à la figure [1.1].

Il est à noter que, selon [Le Beux op. cit.], la représentation sous forme de réseau doit se construire au fur et à mesure que les règles sont introduites. Néanmoins, il nous semble que la construction de ce réseau peut s'effectuer une fois que l'ensemble des règles ont été décrites, pour autant que la prise en compte de ces dernières se fasse de façon séquentielle.

De plus, toujours selon [Le Beux op. cit.], il convient de scinder l'ensemble des règles en deux sous-ensembles, dont il sera tiré deux réseaux distincts.

Le premier, dit réseau expert, regroupe l'ensemble des connaissances spécifiques à l'expert.
Dans le cadre de l'application décrite au chapitre 4, il pourrait s'agir de régles exprimant par exemple les conditions de nationalité d'une personne.

Par exemple

marié(p1,p2) et nationalité(p1,france) ==> nationalité(p2,france)

Cette règle exprime qu'une personne obtient toujours la nationalité française si elle est mariée à une personne de nationalité française.

le second est appelé le réseau adjoint. Il consiste en l'expression d'un certain nombre de règles qui ne sont pas à proprement parler du ressort de l'expert, car elles ne renferment pas de connaissances qui lui soient propres. Ce sont des règles qui décrivent une certaine réalité plus commune et de nature purement logique.

Par exemple, la règle

père(p1,p2) ==> enfant(p2,p1) ET masculin(p1)

n'est pas une règle qui est propre à une expertise sur les renseignements administratifs, elle renferme une

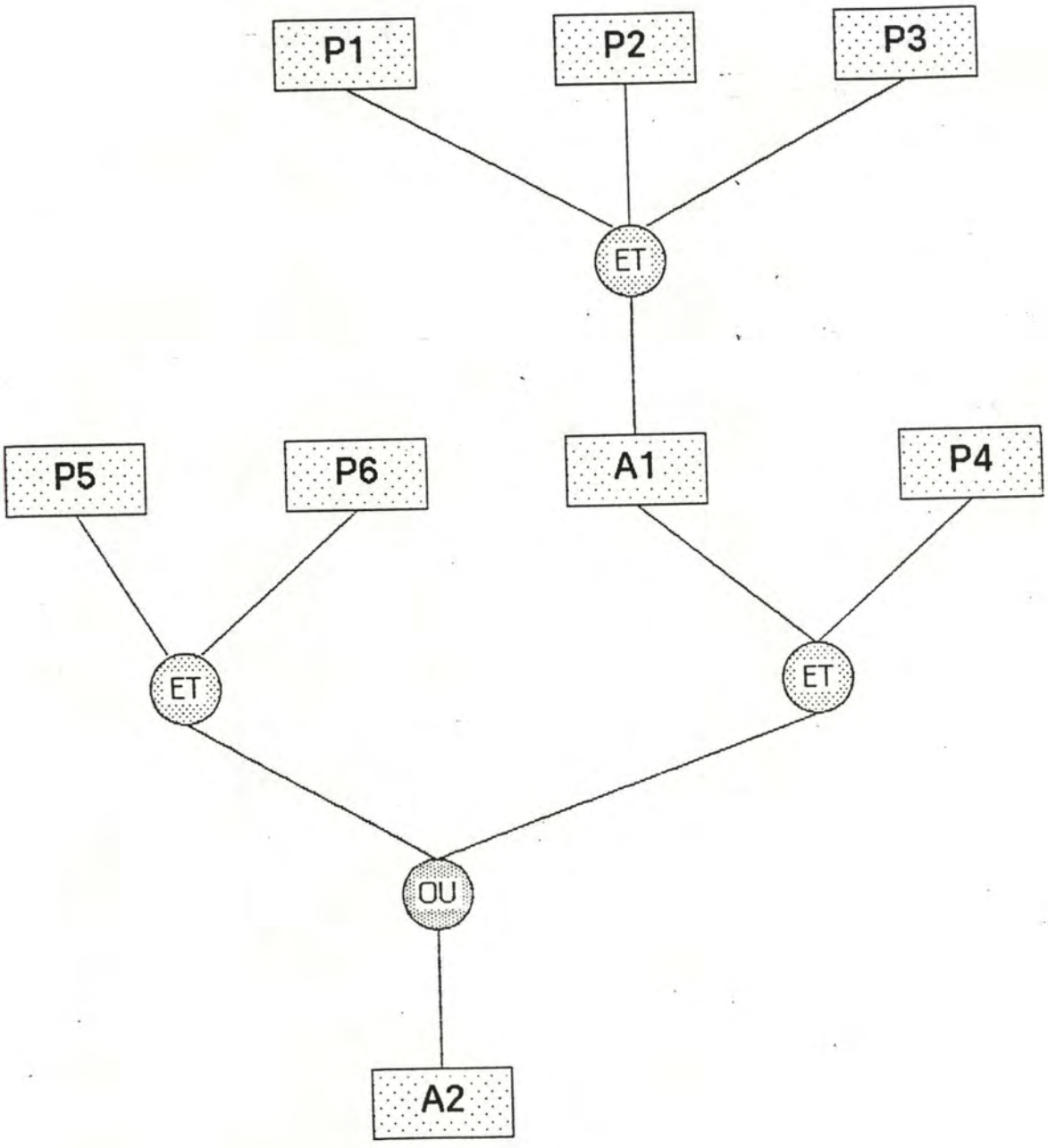


Figure 1.1: réseau de type ET / OU.

connaissance connue de tous, mais qui est pourtant nécessaire dans le cadre d'une telle application.

De notre point de vue, cette distinction n'est pas fondamentale, elle s'avère utile lorsqu'un véritable expert est amené à définir une base de connaissances. A ce moment, il ne conçoit peut-être pas que certaines réalités doivent être exprimées. Celles-ci doivent donc pouvoir être exprimées par ailleurs

Ces règles adjointes ont également certaines caractéristiques communes avec les méta-règles, en ce sens qu'elles peuvent servir, notamment, à la réduction de l'ensemble des règles applicables (conflict set) à un instant donné.

1.5.1.2 Moyens et principes.

Dans la suite, les règles du réseau expert seront notées

Ri [i : 1..n]

et celles du réseau adjoint seront notées

Sj [j : 1..m]

De plus, l'ensemble des prémisses d'une règle Ri sera désigné par l'expression

SET Ri.

Soit la règle

```
R1 : griffe(x)          ET
     yeux_frontaux(x)   ET
     dents_pointues(x) ==> carnivore(x),
```

l'expression SET R1 désignera,

{ griffe(x) , yeux_frontaux(x) , dents_pointues(x) }

1.5.1.3 Caractéristiques certaines et éventuelles.

Soient E et P deux propositions d'une base de connaissances.

Définition 1 : P est une caractéristique certaine de E, si et seulement si la réalisation de E implique la réalisation de P.
On dira qu'une proposition P est réalisée lorsque P constitue une assertion.

Définition 2 : Un trajet lié à une proposition E est toute suite de règles

Ri -> Rk -> Rj -> ..... -> Rl

où l'action de Rl est E et où Ri est une règle initiale.
Ce trajet est noté T(E).

Si une proposition P n'apparaît qu'en tant que prémisse dans une base de connaissances, alors il n'y aura aucun trajet qui lui sera associé;
par contre, si une proposition apparaît comme action d'une règle, on pourra éventuellement lui associer plusieurs trajets Tk(E) [k : 1..q].

Cette définition de la notion de trajet permet de redéfinir la notion de caractéristique certaine d'une proposition de la manière suivante :

Définition 3 : P est une caractéristique certaine de E si et seulement si l'une des trois conditions suivantes est remplie,

[1] ∀ k ∈ [1..q], ∃ i ∈ [1..n] / Ri ∈ Tk(E) et P ∈ SET Ri.

[2] Il existe une règle du réseau adjoint Sj dont l'une des actions est P et dont toutes les prémisses sont des carcatéristiques certaines de E.

[3] Il existe une règle du réseau adjoint Sj dont la seule prémisse serait non(P) et d'action A où non(A) est une caractéristique certaine de E.

Donc à chaque proposition E apparaissant dans la base de connaissances, il est possible d'associer un ensemble de propositions :

{ P / P est caractéristique certaine de E }.

Cet ensemble est noté

certain(E).

A l'instar des caractéristiques certaines, il est possible de définir les caractéristiques éventuelles d'une proposition.

Définition 4 : une proposition P est carcatéristique éventuelle de la proposition E si et seulement si la

proposition P contribue à la réalisation de la proposition E.
Il n'y a donc pas d'implication directe entre P et E, simplement, la vérification de P joue-t-elle en faveur de la vérification de E.

De même que la définition 1 avait été exprimée de manière plus formelle, la définition 4 peut l'être elle aussi, ce qui donne la définition 5.

Définition 5 : P est une caractéristique éventuelle de E si et seulement si l'une des trois conditions suivantes est vérifiée ;

[1] ∃ k ∈ [1..q], ∃ i ∈ [1..n] / Ri ∈ Tk(E) et P ∈ SET Ri.

[2] Il existe une règle du réseau adjoint Sj dont l'une des actions est P et dont toutes les prémisses sont des caractéristiques certaines de E.

[3] Il existe une règle du réseau adjoint Sj dont la seule prémisse serait non(P) et d'action A où non(A) est une caractéristique certaine de E.

De même que pour les caractéristiques certaines, on peut associer à une propostion E l'ensemble des propositions P qui en sont caractéristiques éventuelles.
Cet ensemble est noté

éventuel(E).

La confrontation des définitions 3 et 5 nous permet de déduire que

certain(E) C éventuel(E).

1.5.1.4 Principe d'insertion des règles dans le réseau.

Soient deux règles, R1 et R2, dont les actions sont respectivement A1 et A2 et dont les prémisses sont respectivement désignées par SET R1 et SET R2.
Supposons de plus que R1 soit une ancienne règle et que R2 en soit une nouvelle. Différentes situations peuvent se présenter :

[a] SET R1 = SET R2 et A1 = A2

Nous nous trouvons ici face à un cas typique de redondance entre deux règles.
La nouvelle règle, R2 en l'occurence, ne doit donc pas être insérée dans le réseau.

[b] SET R1 = SET R2 et A1 ≠ A2

Ce cas se présentera fréquemment dans un univers indéterministe et, de plus, il rend sans doute compte de la restriction qui a été faite à propos de l'absence de conjonction dans l'action d'une règle.
Différentes stratégies sont applicables à ce cas, soit la condition d'une des deux règles est reformulée, soit elle est laissée telle quelle.
De plus, c'est ici que doit s'effectuer le contrôle de cohérence entre les actions de chacune des deux règles.

[c] SET R1 ⊂ SET R2 et A1 = A2

Dans ce cas, l'ensemble des prémisses de R2 exprime des contraintes plus fortes que celles exprimées dans R1.
Ici aussi deux stratégies peuvent être envisagées, d'une part la confirmation de la règle R1 accompagnée du rejet de R2 considérée comme trop contraignante, d'autre part, la suppression de R1 et l'insertion de R2.

[d] SET R1 C SET R2 et A1 ≠ A2

Dans ce cas-ci, la règle R1 sera maintenue et la règle R2 sera insérée après la réécriture de ses prémisses en fonction de R1.
Par exemple

R1 : a => b
R2 : a ET c => d

donnera lieu à la reformulation des prémisses de R2 sous la forme

R2 : b ET c => d

[e] A1 C SET R2

Si l'intersection des ensembles des prémisses de chacune des règles est vide, alors, la règle R2 est simplement ajoutée au réseau. Dans le cas contraire, la règle R1 est confirmée et les prémisses de R2 sont réduites de la manière suivante :

SET R2 = SET R2 SET R1.

Ces différentes situations vont être appliquées de façon plus gérérale.

Soit la règle R à insérer dans un réseau dans lequel ont déjà été insérées les règles Ri [i : 1..n].
Le processus d'insertion est le suivant :

[e 1] Cette étape consiste en une "mise en forme" des prémisses de R.

[1] La règle R est comparée à chacune des règles Ri se trouvant dans le réseau afin de déterminer de façon exhaustive l'ensemble des prémisses possibles de R. Ceci est réalisé de la manière suivante :

∀ Rj, si SET Rj C SET R et si Aj ∈ SET R
alors SET R = SET R U { Aj }.

Exemple, soient les règles

R1 : a ET b => c
R2 : b ET d => e
R3 : f ET h => i

et la règle

R : a ET b ET d ET f => g

dont SET R est égal à l'ensemble

{ a , b , d , f }

Après le traitement [1], SET R est devenu égal à l'ensemble

{ a , b , d , f , c , e }

[2] Ensuite, SET R doit être réduit de sorte que ne seront plus conservées que les prémisses vraiment

indispensables.

Soit SET R = { p1 , p2 , p3 , ... , pn } après [1].

Deux formules de réduction sont proposées.
La première

$$SET\ R' = SET\ R - \bigcup_{i=1}^{n} \text{éventuel}(Pi)$$

L'ensemble SET R se trouve donc réduit et l'expression la plus simple possible de l'antécédent de R est ainsi obtenue. Seules les prémisses logiquement indispensables ont été conservées.
La seconde

$$SET\ R'' = SET\ R - \bigcup_{i=1}^{n} \text{certain}(Pi)$$

Dans cette optique-ci, seules les prémisses logiquement inutiles ont été supprimées. Remarquons que le concepteur de la base de connaissances a le choix entre les deux formulations de la règle.

[3] R, ainsi réduite, est de nouveau comparée à l'ensemble des règles ayant mêmes prémisses afin d'envisager les situations [a] et [b] pour éventuellement rejeter les règles redondantes et vérifier la cohérence des différentes actions ou conséquents.

[e 2] Une fois le processus de réduction terminé, l'insertion proprement dite dans le réseau peut s'effectuer.
Dans le cas qui nous préoccupe, seule la situation [b] doit être envisagée; c'est le cas où

A2 = non(A1)

Si c'est le cas, l'incohérence doit être signalée et l'une des deux règles en conflit rejetée.

Si c'est la nouvelle règle qui est rejetée, aucun problème ne se pose; par contre si c'est l'ancienne règle qui doit être rejetée, celle qui était déjà insérée dans le réseau, il y a lieu de réorganiser le réseau.

1.5.2 Exécution symbolique.

1.5.2.1 Introduction.

Nous avons indiqué que notre seul critère de détection d'incohérence était de pouvoir trouver deux faits contradictoires dans l'ensemble des déductions effectuées à partir d'une série de faits.
Remarquons que l'ordre dans lequel sont faites ces déductions nous importe peu.

Illustrons ceci par deux exemples.
Soit les règles

R1 : a => b
R2 : b => c
R3 : c => non(a)

D'un fait a' vérifiant la condition a de R1, il sera déduit un nouveau fait b', par application de cette règle. Ensuite, c' et non(a') seront respectivement déduits par application de R2 et R3.
Ceci remplit notre critère d'incohérence puique les faits a' et non(a') sont contradictoires et dès lors, l'ensemble des règles ayant permis cette déduction doit être déclaré incohérent.

De même, si l'on considère l'ensemble des règles

R1 : a => b
R2 : b => c
R3 : c => d
R4 : b => e
R5 : e => non(d)

La vérification de l'antécédent de R1 par un fait a' provoquera après application des règles,

R1 sur a', dont il sera déduit b'

R2 sur b', dont il sera déduit c'

R3 sur c', dont il sera déduit d'

R4 sur b', dont il sera déduit e'

R5 sur e', dont il sera déduit non(d')

Comme dans le cas précédent, notre critère d'incohérence est satisfait, ce qui amène à déclarer incohérent l'ensemble des règles.

1.5.2.2 Principe de détection.

Il n'est évidemment pas envisageable d'énumérer l'ensemble des faits représentables et d'essayer de les réécrire un à un de toutes les manières possibles en vérifiant à chaque déduction l'absence de contradictions avec les faits déjà déduits.

Rappelons, à ce sujet, que nous avons mentionné au cours du chapitre 3 que chaque concept définissait une classe de concepts.

Par exemple, la définition du concept 'animé' provoque la création d'une classe des concepts (les concepts animés) qui comprend les concepts 'personne', 'animal' et 'administration'. Ces derniers définissent eux-mêmes de nouvelles classes de concepts.

```
animé [ personne       [.......],
        animal         [.......],
        administration [.......]
      ]
```

Cette notion de classe peut être étendue aux formules écrites à l'aide des concepts.
A chaque concept, il est également associé un profil définissant la structure des formules qui l'incluent; une formule écrite à l'aide des concepts définit et représente une classe de formules. Cette classe regroupe l'ensemble des formules qui peuvent être obtenues en remplaçant l'un des concepts de la formule représentant la classe par un concept appartenant à la classe que celui-ci définit.

Par exemple, le concept personne définit une classe des concepts

```
personne [ locuteur,
           tiers,
           machine
```

]

Le concept 'document', lui, définit une classe de concepts reprenant les concepts 'passeport', 'carte_d_identité',

Le concept 'obtenir' définit une classe de concepts vide mais posséde le profil

obtenir(personne,document).

Cette formule représente donc une classe de formules. Ce sont les formules obtenues en remplaçant soit le concept personne, soit le concept document par l'un des concepts appartenant aux classes qu'il définissent respectivement.

Voici, à titre d'exemple, quelques unes des formules qui appartiendront à cette classe :

obtenir(personne,carte_d_identité)

obtenir(locuteur,document)

obtenir(tiers,passeport)

obtenir(locuteur,permis_de_chasse)

Remarquons que chacune des classes ainsi définies contient un certain nombre de sous-classes. Ainsi la sous-classe représentée par la formule

obtenir(personne,carte_d_identité)

contient les formules

obtenir(locuteur,carte_d_identité)

et

obtenir(tiers,carte_d_identité)

et

obtenir(machine,carte_d_identité)

Nous dirons par la suite que toute formule constitue une instance de la formule représentant la classe à laquelle elle appartient.

Chaque antécédent des règles est écrit à l'aide d'une ou plusieurs formules. Les faits étant représentés par des formules, chacune d'elle est représentative de l'ensemble des faits qui vérifieront l'antécédent de la règle.

La formule majeur(personne) est représentative de la formule majeur(locuteur) qui traduit le fait que le locuteur est majeur.

L'idée de cette méthode de détection d'incohérence est de passer en revue l'ensemble des représentants des formules qui peuvent être réécrites. Leur nombre est fini puisque ces représentants sont les différentes formules qui interviennent dans les antécédents des règles de réécriture.

Chaque antécédent de règle sera donc considéré comme constituant un ou plusieurs faits. A partir de ceux-ci, nous essayerons de faire toutes les déductions (ou réécritures) possibles en vérifiant, lors de chaque déduction, que celle-ci ne constitue pas une contradiction avec un fait déjà déduit.

Pour ce faire, il nous suffit de pouvoir disposer d'un moteur d'inférence tel que celui qui réalisera les déductions dans le système opérationnel. Il est également nécessaire de pouvoir accéder à la base de faits pour y trouver le dernier fait inféré et pouvoir le confronter aux autres précedemment inférés.

Soit l'ensemble de règles

```
R0 : a => b
R1 : e => f
R2 : b => c
R3 : f => g
R4 : c => d
R5 : g => -e
```

La règle R0 sera d'abord prise en compte et un fait sera tiré de la prémisse et ajouté à la base de faits.
Ce qui donnera

BF = { a }

La seule règle applicable est la règle R1. Après l'avoir appliquée, on appliquera successivement R3 et R2, ce qui donnera

BF = { a b }

Ensuite, R4 sera appliquée ce qui laissera une base de faits

BF = { a b c }

Constatons que jusqu'ici, la base de faits ne contient pas de faits contradictoires.
Comme plus aucune règle de réécriure ne peut être appliquée, une nouvelle règle sera prise en compte, la règle R1. La base de faits sera réinitialisée à l'aide des prémisses de cette nouvelle règle.

Ce qui donnera

BF = { e }.

L'application de la règle R1 laissera la base de faits dans l'état suivant :

BF = { e f }

Ensuite, la règle appliquée sera R5 qui provoquera l'ajout du fait

-e

à la base de fait. Or celle-ci contient déjà le fait

e,

dès lors une incohérence dans les règles devra être signalée.

Grossièrement, l'algorithme utilisé pourrait se résumer comme suit :

```
pour chaque règle de l'ensemble des règles de réécriture

    |   initialiser la base des faits en
    |   fonction de la prémisse de la règle
    |   à traiter.
    |
    |   tant que des nouveaux faits viennent d'être
    |   ajoutés a la base des faits
    |
    |       |   INFER
    |       |
    |       |   vérifier la cohérence de la base
    |       |   de faits.
    |       |
    |       |   si la base de faits est incohérente
    |       |
    |       |     |   impression du diagnostic d'incohérence.
    |       |   fin_si
    |       |
```

```
    |      fin_tant_que
    |
fin_pour_chaque
```

INFER représente l'appel au moteur d'inférence au sujet duquel, on supposera simplement qu'il fonctionne en chaînage avant.
Néanmoins, tel quel, il présente l'inconvénient de ne pas permettre le suivi de l'application des règles. En effet, l'objectif poursuivi est d'offrir un outil d'aide à la conception pour le concepteur. Or l'algorithme tel qu'il est présenté permet tout au plus d'avoir accès aux différents états successifs de la base de faits. En cas de détection d'incohérence, il est nécessaire de pouvoir déterminer quelles règles sont entrées en conflit.

Si le nombre de règles est important, il sera difficile de pouvoir retrouver les règles qui ont été appliquées au seul vu des états successifs de la base de faits. Dès lors, pour un diagnostic utile de l'incohérence détectée, il paraît indipsensable d'offrir un suivi des règles appliquées et des conditions de leur application.

Par exemple si la règle

```
R6 :  majeur(personne)=> -mineur(personne)
```

a été appliquée au fait

```
majeur(locuteur),
```

il faudrait pouvoir par la suite déterminer que c'est cette règle qui a été appliquée pour passer d'un état de la base de faits à l'état suivant.
Nous supposerons donc que le moteur d'inférence gère cette trace d'exécution.

1.5.2.3 Règles applicables.

Afin de spécifier le moteur d'inférence, il est nécessaire de déterminer le critère d'applicabilté d'une règle.
Une règle sera applicable, lorsque son antécédent sera vérifié.
L'antécédent d'une règle est vérifié si les trois conditions suivantes sont satisfaites simultanément :

[1] si pour chaque formule qui compose l'antécédent de la règle, il existe un fait avec lequel elle "matche",

[2] si chacun des concepts définis par les faits vérifiant l'antécédent peut être unifié avec les concepts de l'antécédent de la règle, définissant ainsi un jeu de substitution.

[3] si la trace ne mentionne pas que cette règle a déjà été appliquée pour le jeu de substitution défini en [2].

Pour rappel, la formule f2 matche avec la formule f1 si et seulement si l'une des deux conditions suivantes est satisfaite :

[1] la formule f2 est identique à la formule f1

majeur(personne)

matche avec

majeur(personne)

[2] la formule f2 constitue une instance de la formule f1 ou, en d'autres termes, si la formule f2 appartient à une classe de formules dont f1 est représentative.

Prenons deux exemples afin d'illustrer ceci :
[ex. a] Soit la règle

R0 : nationalité(personne,france)
ET résider(personne,france) => ...

et la base de faits contenant les faits

f1 : nationalité(locuteur,france)
et
f1 : résider(locuteur,france)

La première condition est bien vérifiée, en effet pour chacune des formules de l'antécédent, il existe un fait qui matche avec elle.

nationalité(locuteur,france)

matche avec

nationalité(personne,france)

et

résider(locuteur,france)

matche avec

résider(personne,france)

La seconde condition est elle aussi vérifiée, en effet les concepts des deux faits f1 et f1 peuvent s'unifier.
Les deux occurences de 'locuteur' peuvent s'unifier comme le veut la double occurrence du concept 'personne' dans l'antécédent de la règle.
Il est de plus évident que les deux concepts 'france' peuvent s'unifier selon la double occurrence de ce même concept dans la règle.

[ex. b] Soit la même règle R0 que dans l'exemple précédent et les faits

f1 : nationalité(locuteur,france)

et

f1 : résider(tiers,france)

Dans cet exemple la première de nos deux conditions reste vérifiée; en effet

nationalité(locuteur,france)

matche avec

nationalité(personne,france)

et

résider(tiers,france)

matche avec

résider(personne,france)

Mais l'occurrence du concept ' locuteur ' ne peut s'unifier avec l'occurrence du concept ' tiers ' comme le voudrait la double occurrence du concept personne de la règle qui, rappelons-le, désigne une seule et même entité.En effet de manière exclusive dans la définition des concepts.

1.5.2.4 Initialisation de la base de faits.

Nous avons dit plus haut que la base de faits devait être initialisée au début de chaque traitement de règles en fonction des prémisses de celles-ci.

Il ne suffit pourtant pas de générer autant de faits qu'il n'y a de prémisses à la règle. en effet si nous prenons les règles

R1 : a => b
R2 : a => -c
R3 : b ET d => c

les traitements des règles R1 et R2 provoqueraient l'initialisation de la base de faits par le fait

a

et celui de R3 par les faits

b et d

ce qui, après les différentes réécritures possibles, laisserait la base de faits dans un état ne permettant pas de détecter d'incohérence selon notre critère. Néanmoins, rien ne permet d'affirmer que la base de faits ne contiendra jamais simultanément les faits

a et d,

auquel cas R1 pourrait être appliquée ce qui permettrait de déduire

b.

Ensuite R2 pourrait être appliquée pour la même raison que R1, permettant la déduction de

c.

Puis R3 pourrait l'être également puisque

b

avait été déduit précédemment. Ce qui amènerait la base de faits dans l'état suivant :

BF = { a , d , b , c , -c }

satisfaisant au critère d'incohérence.

De ceci, nous pouvons conclure que, lors de l'initialisation de la base de faits, les faits qui devront y être insérés correspondront non seulement aux prémisses de

la régle à traiter, mais aussi aux prémisses de toute règle dont le conséquent est un fait déjà inséré dans la base de faits.

1.5.2.5 Règles à traiter.

Traiter une règle consiste en l'initialisation d'une recherche d'incohérence à partir de cette règle et en la réalisation de toutes les déductions possibles à partir de l'ensemble de faits ainsi initialisé.
Une question se pose néanmoins, toutes les règles doivent-elles faire l'objet d'un tel traitement ?

Examinons un exemple.

Soient les règles :

R0 : a => b
R1 : b => c
R2 : c => d

Supposons que nous appliquions l'algorithme décrit précédemment et que les règles soient traitées dans un ordre purement séquentiel.

La règle R0 sera la première traitée, de telle sorte que la base de faits sera initialisée en fonction de ses prémisses.

Cela donnera

BF = { a }

Les règles qui seront appliquées successivement seront R0, R1 puis R2, donnant des états successifs de la base de faits

BF = { a , b }
puis
BF = { a , b , c }
et
BF = { a , b , c , d }

Remarquons qu'aucune incohérence n'a été détectée.

Ensuite, le traitement de la règle R1 devrait être envisagé avec initialisation de la base de faits à

BF = { b }

suivie de l'application successive des deux règles R1 et R2. Les états successifs de la base de faits seront

BF = { b , c }

et, aprés l'application de R1,

BF = { b , c , d }.

Ceux-ci ne sont donc que des sous-ensembles des états successifs obtenus lors du traitement de la rège R0.

Dés lors, si ceux-ci ne se sont pas trouvés dans un état incohérent, leurs sous-ensembles ne pourront jamais s'y trouver. Ce qui nous permet de dire que les règles à traiter sont celles qui n'ont jamais été appliquées au cours d'un traitement précédent.

1.5.2.6 Le moteur d'inférence.

Nous avons précisé lors de l'introduction de ce chapitre que l'un des composants de tout système expert était son moteur d'inférence.
De plus, nous avons indiqué que le type de moteur à utiliser était fonction de l'objectif poursuivi. Il est clair que, dans le cas qui nous préoccupe, l'objectif est de réaliser un maximum de réécritures (déductions) sur base d'un certain nombre de faits supposés connus afin de voir si des faits contradictoires ne riquent pas d'être déduits.
Le moteur d'inférence que nous utiliserons sera donc un moteur fonctionant en chainage avant.

Nous avions également mentionné qu'en cas de conflit entre les règles à appliquer, il fallait en choisir une selon une certaine stratégie afin de le résoudre.
Nous avons choisi de solutionner ce problème d'une manière à la fois originale et simple qui consiste à ne pas résoudre le conflit, mais à appliquer "simultanément" toutes les règles applicables à un instant donné.

Ce choix peut se justifier de la façon suivante :

- un maximum de déductions sont à réaliser afin de détecter au plus vite les incohérences, il nous a donc semblé plus efficace de réaliser directement toutes les déductions possibles.

- Nous situant dans un système de déduction, l'effet de l'application d'une règle ne peut être que l'ajout d'un fait à la base de faits. Donc, un fait y appartenant et participant à la satisfaction des conditions d'une règle ne peut pas être supprimé ou invalidé par l'application d'une autre; il est dès lors acceptable de les appliquer simultanément.

- Néanmoins, en supposant qu'il soit capital de n'appliquer qu'une seule règle à la fois, l'insertion d'une procédure de résolution du "conflict set" ne poserait aucun problème.

C'est le moteur d'inférence qui se charge de gérer la trace d'exécution. Dans cette trace sont stockés les règles appliquées et les conditions de leur application, c'est à dire un identifiant de la règle et le jeu de substitutions pour lequel elle a été appliquée.
La raison d'être de cette trace est la nécéssité éventuelle de retour arrière dans les déductions, ainsi que la possibilité de suivre le fonctionnement du moteur pour pouvoir éventuellement par la suite expliquer la provenance d'une incohérence.

En appliquant ce qui vient d'être dit à l'algorithme présenté initialement, on peut en tirer un nouveau :

```
pour chaque règle de l'ensemble des règles de réécriture

   |  si la règle doit encore être traitée
   |
   |       |    initialiser la base des faits en
   |       |    fonction de la prémisse de la règle
   |       |    à traiter.
   |       |
   |       |    tant que des nouveaux faits viennent d'être
   |       |    ajoutés a la base des faits
   |       |
   |       |        |     INFER
   |       |        |
   |       |        |     vérifier la cohérence de la base
   |       |        |     de faits.
   |       |        |
   |       |        |     si la base de faits est incohérente
   |       |        |
   |       |        |      |  impression du diagnostic d'incohérence.
   |       |        |     fin_si
   |       |        |
   |       |    fin_tant_que
   |       |
   |   fin_si
   |
 fin_pour_chaque
```

1.5.2.7 Remarque.

Dans ce qui précède, il a été dit que les règles de réécritures dont le membre de droite contiendrait une disjonction ne serait pas prises en compte. Neânmoins, il semble possible d'avoir un jour à traiter ce genre de règles. Etant dès lors obligés de les accepter, il faut vérifier leur cohérence.

Afin de pouvoir utiliser la même méthode que celle qui est présentée ici, nous proposons de procéder de la manière suivante : chacun des OU d'un membre de droite sera considéré comme un OU exclusif.
Pour chacun d'eux, il sera généré une règle qui possède l'une des propositions alternatives comme membre de droite. Chacune des règles ainsi générées aura mêmes prémisses que la règle originale.

Ceci peut être explicité de façon plus formelle de la manière suivante :

Soit la règle

R : c1 ET c2 ET ... ET cn => a1 OU a2 OU ... OU am

A partir de cette règle, il en sera généré m ayant chacune

c1 ET c2 ET ... ET cn

comme antécédent et

ai

comme conséquent. Etant alternatives, ces règles ne peuvent se trouver dans un même ensemble. Donc autant d'ensembles seront générés que de règles n'auront été créées.

Dès lors, d'un ensemble contenant p propositions alternatives dans les membres de droite de ses règles, il sera généré

$\prod_{i}$ #ACT(Ri) avec i C [1..p]

où p est le nombre de règles
et ACT(Ri) désigne l'ensemble des
conséquents de la règle Ri.

ensemble ne contenant aucune règle
dont le membre de droite serait une disjonction.

Soient les règles

R0 : a => b OU c
R1 : c => d OU e.

La décomposition donnera lieu à la création des 4 (2 * 2) ensemble de règles suivants :

E1 : { R0' : a => b
R1' : c => d }

E1 : { R0' : a => b
R1'' : c => e }

E1 : { R0'' : a => c
R1' : c => d }

E1 : { R0'' : a => c
R1'' : c => e }

Le traitement des réécritures symboliques sera alors appliqué successivement à chacun des ensembles, et si des incohérences sont alors découvertes, il s'agira d'incohérences potentielles. En effet, face à une règle contenant une disjonction dans le conséquent, il ne peut être déterminé a priori laquelle des propositions disjointes sera effectivement choisie.

Un tel traitement représente bien évidemment une lourde charge en temps de calcul et en place mémoire consommée, mais celle-ci reste toutefois tolérable dans la mesure où le traitement n'est exécuté qu'une seule fois de manière statique en dehors du fonctionnement du système utilisant les règles.

1.6 Conclusion

Les principes, méthodes et algorithmes exposés ici peuvent permettre de soulager l'expert dans le travail fastidieux que représente l'expression de ses connaissances.

Dans les deux méthodes présentées, les conflits entre les règles sont détectés. Le premier système présenté permet en outre la détection d'autres pièges dans lesquels pourrait tomber l'expert. Il présente toutefois, selon nous, l'incovénient de reformuler de façon automatique les règles, ce qui pourrait sembler moins naturel à l'expert.

Bien que différentes, ces deux méthodes présentes toutefois des points communs :

- la hiérarchisation de l'ensemble des règles
- l'utilisation de concepts communs tel que la désigantion des ensembles certain( ) et éventuel( ).

Aucune des deux méthodes ne prétend résoudre l'ensemble des problèmes posés par l'acquisition des connaissances par un système expert, ceux-ci restent nombreux, diversifiés et fortement liés au formalisme de représentation utilisé. Dans le cas présent, ces méthodes offrent une assistance à la conception d'un système à base de règles de déduction ou de réécriture.

2 Les contraintes dans le système développé à Nancy.

2.1 Introduction.

En informatique, lorsqu'on parle de cohérence, on pense bien souvent aux contraintes d'intégrité utilisées dans les bases de données.

Une base de données est une collection d'articles qui sont organisés de telle sorte qu'il soit aisé d'y accéder. La structure de l' organisation liant les articles rend compte des liens sémantiques qui relient les articles stockés dans la base.

L'historique du dialogue constitue, lui aussi, en un certain sens une collection d'éléments. Ils se distinguent quelque peu des articles classiques par le fait que leur structure est sans doute moins précise. Les éléments de l'historique peuvent, eux aussi, faire l'objet de contraintes et il nous a semblé intéresant d'essayer d'établir le lien entre les contraintes d'intégrité au sens où elles sont utilisées dans les bases de données et au sens où elles peuvent être utilisées dans le cadre d'un ensemble de faits.

Un système tel que celui présenté au chapitre 3 prend en compte deux type de contraintes d'intégrité :

[1] des contraintes statiques

[2] des contraintes qui sont susceptibles d'évoluer, d'apparaitre ou de disparaitre en cours d'exécution. Elle pourraient être nommées contraintes dynamiques; néanmoins, pour ne pas entrer en conflit avec la littérature existante et portant sur les bases de données, nous les nommerons contraintes évolutives.

2.2 Les contraintes statiques.

Les contraintes statiques que l'on peut définir en rapport avec le contenu de la base de faits sont tout à fait semblables à celles qui peuvent être définies sur le contenu d'une base de données classique.

Il s'agit de propriétés qui ne sont pas représentées par les concepts de base du modèle, mais qui doivent toutefois être satisfaites par les informations qui sont stockées dans la mémoire du système. A priori, la syntaxe offerte (cfr. chapitre 4) pour la définition des concepts ne permet pas de définir de telles relations sur les concepts du domaine d'application.

Néanmoins, la définition même des concepts et les règles de réécriture vont nous permettre de contourner cette apparente difficulté.

Supposons que nous souhaitions exprimer une contrainte portant sur l'âge des personnes.
Par exemple, nous voulons exprimer que l'âge de toute personne doit être compris entre les entiers 0 et 100.

Ceci peut être réalisé lors de la définition du concept 'âge' :

```
mesure [ .....,
         âge(personne,int0_100),
         .....
       ]
```

accompagné de la définition d'un concept 'int0_100' :

```
int0_100 [ 0..100]
```

Ceci exprime bien la contrainte souhaitée, même si ce mode d'expression présente quelques lourdeurs.

Supposons, d'autre part que nous souhaitions exprimer que l'âge d'une personne mineure ne peut excéder 17 ans. L'expression de cette contrainte peut se faire à l'aide des règles de réécriture de la manière suivante :

```
âge(personne,int0_100) ET inf(int0_100,17) => mineur(personne)
et
âge(personne,int0_100) ET sup(int0_100,17) => majeur(personne)
```

Ce qui peut se traduire de la façon suivante dans le formalisme présenté au chapitre 3 :

&(âge(personne,int0_100) , inf(int0_100,17)) => mineur(personne)

&(âge(personne,int0_100) , sup(int0_100,17)) => majeur(personne)

Ceci indique donc que le modèle disponible pour la définition du domaine d'application permet d'exprimer un certain nombre de contraintes d'intégrité liant les concepts entre eux.

Il est toutefois évident que l'expression des contraintes statiques à l'aide du modèle utilisé représente un certain alourdissement de la définition de la tâche.

2.3 Les contraintes évolutives.

Les contraintes évolutives se distinguent des précédentes par le fait qu'elles n'ont pas le même poids sur les faits que celles qui viennent d'être décrites sur les faits qui doivent être intégrés à la base de faits.

Quelles sont ces contraintes ?

Lorsque le locuteur prononce une phrase qui, après avoir été reconnue, analysée et interprétée, est passée sous forme de formule(s) au module de gestion de dialogue, celui-ci va vérifier, outre la non violation des éventuelles contraintes statiques, que le fait ou les faits représentés par cette formule ne sont pas en contradiction avec les faits déjà contenus dans la base de faits.
Ainsi les faits déjà intégrés à la base de faits constituent-ils, dans une certaine mesure, des contraintes sur les faits qui pourraient y être intégrés par la suite.

La différence avec les contraintes classiques se situe au niveau de la réaction du système en cas de violation de l'une d'elles par un fait.
En effet, si la violation d'une contrainte d'intégrité classique doit provoquer le rejet pur et simple du fait (ainsi que dans toute base de données), la violation d'une contrainte évolutive doit amener à remettre en cause tant le fait que la contrainte violée par celui-ci.

Deux raisons peuvent, en effet, être évoquées :

[1] d'une part les lacunes de la reconnaissance du signal peuvent avoir conduit à une mauvaise compréhension d'un énoncé produit par le locuteur

[2] et d'autre part, il reste toujours possible que ce soit le locuteur lui-même qui se soit mal exprimé.

Prenons un exemple; soit l'ensemble des faits contenant le fait

f1 : majeur(locuteur)

Soit la formule

f2 : mineur(locuteur)

déduit de la phrase "j'ai 16 ans" prononcée par le locuteur, puis reconnue et interprétée. Cette déduction a été réalisée grâce à la règle de réécriture

&(âge(personne,int0_100),inf(int0_100,17) => mineur(personne)

L'utilisation d'une autre règle de réécriture

mineur(personne) => -majeur(personne)

permet de détecter la présence d'une incohérence dans la base de faits.

Il ne peut être question de rejeter purement et simplement f2 sous prétexte qu'il viole la contrainte exprimée par f1. Cette contrainte nous a permis de détecter l'incohérence et celle-ci doit être levée pour pouvoir éventuellement satisfaire la requête émise par le locuteur.

Cette incohérence doit être levée par une procédure spécialisée.
Plusieurs stratégies peuvent s'appliquer à cette fin :

[a] Ne tenir compte que du premier énoncé, ce qui, dans le cas présenté reviendrait à considérer l'interprétation de tout énoncé, correcte ou erronée, comme une contrainte statique.

[b] Ne tenir compte que du dernier énoncé, ce qui à terme risque de permettre au locuteur de "jouer avec les pieds" du système en affirmant des faits contradictoires. Une autre conséquence de ceci serait que les mauvaises interprétations de ce qu'il a dit ne seraient pas aisément perçues par le locuteur.

[c] Envisager l'utilisation des scores propagés par les différents modules du système pour ne retenir que le fait gratifié du meilleur score. Toutefois, au vu des performances actuelles de la reconnaissance du signal cette stratégie nous apparaît comme peu fiable.

[d] Enclencher une phase de validation / invalidation des faits en conflit, via le gestionnaire du dialogue. Cette startégie nous paraît intéressante car elle semble correspondre au comportement naturel de tout qui, confronté à une situation incertaine, veut s'en sortir.

Le système pourrait poser explicitement la question suivante

" N'avez vous pas dit être majeur ? "

Selon la réponse faite par le locuteur, sur laquelle des hypothèses assez strictes pourraient être émises,

le raisonneur pourrait remettre sa base de faits dans un état cohérent.

2.4 Conclusion.

Bien qu'ayant été bref sur la question, il nous semble en avoir dit suffisament pour conclure que les procédures destinées à assurer la cohérence de l'historique du dialogue pourraient être semblables à celles utilisées en matière de cohérence de base de données. Elles devraient bien sûr intégrer certaines particularités telles que les déductions réalisées à partir des règles de réécriture ainsi que les possibilités de retour arrière lorsqu'un fait s'avère erroné.

CONCLUSION.

L'ordinateur envahit de plus en plus tous les secteurs d'activités humaines.
Le mode de communication entre l'homme et la machine prend dès lors une importance considérable.
L'utilisation de la parole semble être une solution très sésuisante, bien qu'elle pose un certain nombre de problèmes complexes.

Jusqu'à présent, les recherches dans le domaine se sont surtout centrées sur l'étude de la reconnaissance et de la compréhension de phrases "isolées" dans des contextes très limités.
Toutes aboutissent à peu près à la même conclusion : il s'agit de processus non déterministes prenant place dans un environnement contenant inévitablement des erreurs.

Une des méthodes les plus utilisées s'appuye sur l'interaction de connaissances linguistiques et acoustiques, et relève du domaine de l'Intelligence Artificielle.

Actuellement, il semble que les recherches s'orientent plutôt vers de réels systèmes de dialogues caractérisés par l'utilisation d'un langage pseudo-naturel.
Le passage à de tels systèmes pose un certain nombre de problèmes nouveaux et très spécifiques, et leur résolution nécessite l'intégration de nouvelles sources de connaisssances et de nouvelles fonctions.

A Nancy, nous avons été amenés à travailler sur un système de gestion de dialogues oraux finalisés conçu pour une classe d'applications "grand public" de type "centre de renseignements".
Ce système relève également du domaine de l'Intelligence Artificielle de par son objectif premier et les principes généraux qui sous-tendent sa conception.
Cette dernière repose d'une part sur l'interaction des sources de connaissances nécessaires à la compréhension des énoncés et à la gestion du dialogue, et d'autre part sur l'utilisation de stratégies de recherche de solution dans un espace.

Le composant central de ce système est le module de dialogue. Il regroupe les fonctions d'interprétation contextuelle, de gestion de dialogue proprement dite, de raisonnement, et de génération des interventions de la machine.

Notre participation au développement de ce système s'est située au niveau de l'interprétation contextuelle des énoncés et au niveau de certains aspects de la cohérence.

L'interprétation contextuelle a pour objectif d'établir une représentation opératoire de la signification que prennent les énoncés dans le cadre d'un dialogue et d'une application donnée. Pour ce faire, elle s'appuye sur des représentations syntaxique et sémantique ou éventuellement lexicale fournies par le module d'analyse syntaxico-sémantique.

La réalisation de cet objectif nécessite la mise en oeuvre de connaissances lexicales, syntaxiques, sémantico-pragmatiques, et pragmatiques (historique du dialogue).

Elle doit constituer un modèle d'interprétation des énoncés issus de dialogues oraux finalisés, modèle pragmatique et expérimental en raison de l'absence de théorie générale sur laquelle il peut s'appuyer.
Ce modèle doit en outre satisfaire à des exigences de robustesse, de généralité ("paramétrabilité") et de souplesse.

Nous avons opté pour le développement d'un outil spécifique basé sur le formalisme des systèmes de production, et ayant comme objectif de permettre la définition de ce modèle sous forme de règles de production et d'assurer sa mise en oeuvre via un interpréteur de règles.
Nous n'avons pas réalisé ce modèle, principalement en raison des connaissances linguistiques qu'il nécessite pour sa mise au point et que nous ne possédons pas.
Nous nous sommes surtout centrés sur la définition de cet outil et sur l'implémentation d'un sous-système utile axé autour de l'interpréteur de règles.

Il est encore trop tôt pour pouvoir évaluer correctement la solution que nous proposons.
En effet, c'est au moment de la définition du modèle d'interprétation et de sa mise en oeuvre dans le système complet qu'apparaîtront ses forces et ses points faibles.
Cependant, il nous semble d'ores et déjà que le formalisme de relations utilisé présente certains inconvénients, et que l'ajout d'un mécanisme d'explication des résultats ainsi que l'introduction de scores les pondérant peuvent constituer deux développements ultérieurs intéressants.

Nous avons tenté de montrer le rôle et l'importance de la cohérence dans un système de dialogue homme-machine.

Nous avons évoqué ses différents rôles

- cohérence des énoncés par rapport à la tâche;
- cohérence des énoncés par rapport à l'historique du dialogue et du raisonnement;

- cohérence de la description de la tâche et plus spécialement des règles de réécriture.

Nous nous sommes tout particulièrement attardés sur ce dernier point.

Rappelons que les règles de réécriture (ou encore règles de déduction) sont utilisées par le raisonneur du module dialogue afin de progresser vers la satisfaction de la requête de l'utilisateur à partir des affirmations de celui-ci.

L'intérêt d'une telle vérification de cohérence dépasse le cadre du système développé à Nancy et présenté en détails au chapitre 3.
La méthode présentée est donc générale et peut dés lors s'appliquer à n'importe quel ensemble de règles de déduction, pour autant que celles-ci satisfassent à la définition qui en est donnée au chapitre 5.

Elle représente également un pas vers l'amélioration des techniques d'acquisition des connaissances qui jouent un rôle important en intelligence artificielle et continuent à poser un certains nombre de problèmes.

Les deux méthodes présentées possèdent un point commun important : elles aboutissent toutes deux à une hiérarchisation des règles.

La première présente une hiérarhie effective puisqu'elle passe par la structuration de l'ensemble des règle en un réseau.

Dans la seconde, la hiérarchisation des règles est également présente dans la mesure où les règles ne sont pas traitées de façon indifférente selon qu'elles ont déjà été appliquée ou non ainsi que dans la mesure où à partir d'un fait généré sur base d'une règle déterminée, toutes les déductions possibles sont effectuées.

Cette seconde méthode présente l'avantage de ne pas avoir à construire de structures complexes et spéciales autres que celles employées habituelelement par les systèmes à base de règles.

Elle présente un incovénient non négligeable qui est de ne pouvoir détecter aisément d'incohérence autres que celles du type "contradictions". La première méthode permet, elle, de détecter d'autres incohérences telles les redondances et les bouclages.

REFERENCES BIBLIOGRAPHIQUES.

[Argyle 81]

M. Argyle, A. Furnham et J. Graham.
"J.A. Social Situations."
Cambridge University Press, Cambrigde, England, 1981.

[Carbonell 85-a]

N. Carbonell, F. Charpillet, J.P. Haton, B. Mangeol, P. Mousel, J.M. Pierrel, A. Roussanaly.
Dialogue oral homme-machine : bilan du projet MYRTILLE et perspectives.

[Carbonell 85-b]

N. Carbonell, J.P. Haton, B. Mangeol, P. Mousel, J.M. Pierrel, A. Roussanaly.
Les sources de connaissance dans un système de dialogue oral Homme-Machine.
Actes du congrès "Reconnaissance des formes et Intelligence Artificielle", AFCET.
Grenoble, France, Novembre 1985

[Davis 77]

R. Davis
"Production rules as a representation for a knowledge base consultation system."
Artificial Intelligence, 8, 1977, p 15-45.

[Deutsh 74]

B.G. Deutsh.
"The structure of task oriented dialogs"
Proceedings of the I.E.E.E. symposium on speech recognition, 1974.

[Deville 86]

G. Deville & H. Paulussen .
"A case grammar as an original linguistic model for the semantic representation of utterances in a man-machine dialog system".
Mémoire de linguistique informatique
Universitaire Instelling Antwerpen, 1986.

[Falzon 85]

P. Falzon.
"Les langages opératifs"
Actes du séminaire GRECO-GALF "Dialogue homme-machine à composante orale"
Nancy, France, 1985

[Guilbert 73]

L. Guilbert.
"La spécificité des termes scientifiques et techniques."
La langue française, n. 17, Février 1973.

[Fohr 85]

D. Fohr, N. Carbonell, J.P. Haton.
"SYSTEXP, un système expert pour le décodage acoustico-phonétique".
Actes des cinquièmes journées internationales sur les systèmes experts et leurs applications.
Avignon, France, Mai 1985.

[Haton 85]

J.P. Haton
"Intelligence artificielle en compréhension automatique de la parole : état des recherches et comparaison avec la vision par ordinateur."
T.S.I., vol. 4, n. 3, 1985, p 265-287.

[Hayes-Roth 85]

F. Hayes-Roth.
"rule-based systems".
Comm. ACM, vol. 28, n. 9, 1985, p 921-932.

[Jackson 85]

P. Jackson
"knowledge representation."
Actes de la chaire IBM.
Namur, Belgique, Avril 1985

[Kamp 81]

H. Kamp.
"Evènements, représentations discursives et référence temporelle."
Langages, n. 64, p 39-64.

[Kamp 84]

H. Kamp.
"A Theory of truth and semantic representation".
in " Truth, interpretation and information",
Groenendijk and alii,
Foris, Groningen, 1984, p. 1-42.

[Kayser 85]

D. Kayser.
"Examen des diverses méthodes utilisées en représentation des connaissances."
Congrès "Reconnaissance des formes et Intelligence Artificielle".
AFCET, Grenoble 27-29 novembre 85.

[Kittredge 79]

R. Kitredge
"Textual Cohésion within Sublanguages : implications for automatic analysis and synthesys."
Actes du colloque INRIA-LISH "représentation des connaissances et raisonnement dans les sciences de l'homme."
St Maximin, France, 17-19 septembre 1979.

[Kowalski 79]

Kowalski R.
"Logic for Problem Solving."
Artificial Intelligence Series, The Computer Library
North Holland, New-York, U.S.A., 1979

[Kripke 61]

S. Kripke.
"Semantical consideration on modal logic."
L.Linsky editor.
Oxford University Press, 1961, p 63-72.

[Laurière 82]

J.L. Laurière.
"Représentation et urilisation des connaissances".
TSI, vol. 1, n. 1, 1982, p 25-42.

[Lea 80]

Wayne A. Lea and all
"Trends in speech recognition."
Engelwood Cliffs, Prentice Hall Inc., 1980

[Le Beux 86]

P. Le Beux, D. Fontaine.
un système d'acquisition des connaissances pour systèmes experts.
T.S.I., vol 5, n. 1, 1986, p 7-19.

[Liberman 70]

A.M. Liberman.
"The grammars of speech and language."
Cognitive Psychology,n. 1, 1970, 301-323.

[Lyons 70]

J. Lyons.
"Linguistique générale"
Librairie LAROUSSE, Paris, 1970.

[Marouzeau 31]

J. Marouzeau.
"Lexique de la terminologie linguistique".
Paris, Geuthner, 1931.

[Martinet 56]

A. Martinet
"Eléments de linguistique générale".
Paris,Colin, 1960.

[Mariani 82]

J. Mariani
"The ESOP Continuous Speech Understanding System."
Proceedings of the IEEE-ICASSP
Paris,France,1982

[Mc Dermott 78]

J. Mc Dermott, A. Newell, et J. Moore.
"The efficiency of certain production systems implementations"
in "Pattern-directed inference systems"
Academic Press, 1978, Carnegie-Mellon Univ.

[Meloni 83]

H. Meloni.
"Traitement des contraintes linguistiques en reconnaissance de la parole."
T.S.I., Vol. 2, n. 5, 1983, p 349-363.

[Minsky 75]

Minsky M.
"A Framework for Represnetin Knowledge."
in "The psychology of Computer Vision." edité par Winston P.H., p 211-217

[Nef 86]

F. Nef
"Construction et révision des représentations discursives."
Colloque "Langue naturelles et argumentation"
Rayaumont, Frace, 5-7 Mars 1986.

[Pierrel 81]

Pierrel J.M.
"Etude et mise en oeuvre de contraintes linguistiques en compréhension automatique du discours continu."
Thèse d'état, Université de Nancy I.
Nancy, France, Mars 1981.

[Pierrel 82]

Pierrel J.M.
"Utilisation de contraintes linguistiques en compréhension automatique de la parole continue : le système MYRTILLE II."
TSI , 1982, p 404-421.

[Pierrel 85 a]

J.M. Pierrel
"Aspects of man-machine voice dialog."
Actes du congrès INRIA, Mai, Juin 1985, p. 253-278.

[Pierrel 85 b]

J.M. Pierrel
"Dialogue oral homme-machine en situation orientée par l'action."
Actes du sèminaire " Dialogue homme-machine à composante orale"
Nancy, France, 11 - 12 Mai 1985, p 91 - 120.

[Pierrel 85 c]

J.M. Pierrel
"Aspects of man-machine voice dialog."
Actes du séminaires INRIA "Principes de la communication homme-machine"
Parole, Vision et Langage Naturel.
Vresailles, France, 28 Mai - 7 Juin 85.

[Quinton 82]

P. Quinton
"Utilisation de Contraintes Syntaxiques pour la Reconnaissance de la Parole Continue."
T.S.I. vol.1; n.3; 1982; p403-421.

[Weizenbaum 84]

J. Weizenbaum.
"Computer Power and Human Reason, from Judgment to Calculation."
London, Penguin Books, 1984

[Winston 77]

Winston
"Artificial Intelligence"
Addison-Wesley, New-York,1977

[Woods 72]

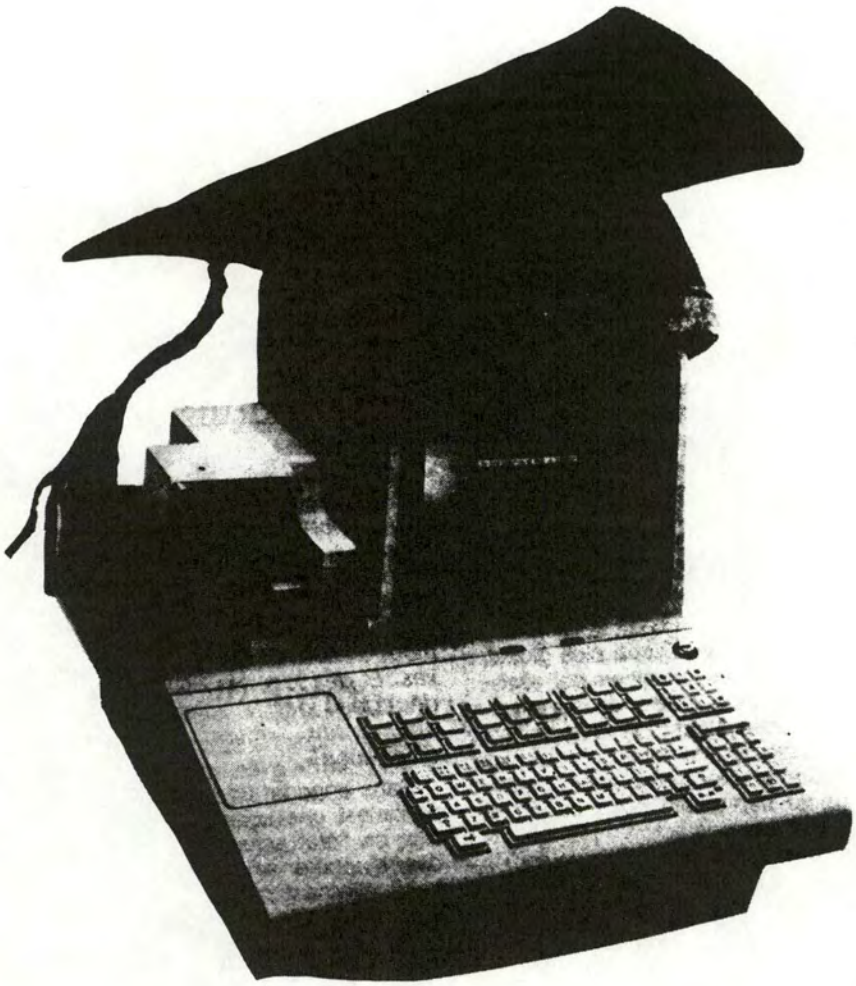
Woods W.A., Kaplan R.M. and Nash-Weber B.
The LUR Sciences Natural language Information system : Final Report
B.B.N. Report N. 2378, Bolt Beranek and Newman, Cambridge, Massachussets,U.S.A. , 1972.

TABLE DES MATIERES(*).

(*) Le lecteur trouvera une table des matières plus détaillée en fin de chaque chapitre.

RESPONSABILITE DES CHAPITRES

Introduction	Manu Lorant et serge Simonet.
Chapitre I, section 1	Manu Lorant.
Chapitre I, section 2	Serge Simonet.
Chapitre I, section 3	Manu Lorant et. Serge Simonet.
Chapitre II	Serge Simonet.
Chapitre III	Manu Lorant.
Chapitre IV	Serge Simonet.
Chapitre IV	Manu Lorant.
Conclusion	Manu Lorant et Serge Simonet.



Facultés Universitaire Notre-Dame de la Paix

Institut d'informatique

Rue Grandgagnage, 21

B - 5000 Namur.

CONTRIBUTIONS A UN SYSTEME DE DIALOGUE HOMME-MACHINE A COMPOSANTE ORALE : ANNEXES

Mémoire présenté par

Manu Lorant

et

Serge Simonet

en vue de l'obtention

du titre de

Licencié et Maitre en Informatique

Année académique 1985-1986

TABLE DES MATIERES DES ANNEXES.

ANNEXE I : FORMALISME DE REPRESENTATION DES CONNAISSANCES MANIPULEES PAR LE RAISONNEUR.

1 Définition des terminaux.

Les termes soulignés désignent les terminaux de la grammaires.

^ désigne une chaîne vide.

VI désigne une virgule, ',' .

PV désigne un point virgule, ';' .

MK désigne un marqueur, '%%' .

TERME désigne une chaîne de caractères de l'alphabet (majuscules et minuscules).

VAL désigne une valeur de l'univers de la tâche.

PO désigne une paranthèse ouvrante, '(' .

PF désigne une paranthèse fermante, ')' .

CO désigne un crochet ouvrant, '[' .

CF désigne un crochet fermant, '[' .

ET désigne un crochet fermant, '&' .

OU désigne un crochet fermant, '|' .

IM désigne une 'implication', '=>' .

2 Définition de la grammaire des concepts.

Nous avons présenté dans les chapitres 3 et 5 du mémoire le rolê qu'avaient à jouer les concepts dans la définition de l'univers de la tâche. Nous n'y reviendrons pas ici.

L'introduction des concepts s'effectue par éditeur de texte, emacs par exemple. Elle doit satisfaire à la grammaire suivante :

```
concepts : %% def %%

def : cpt vi_cpt;

vi_cpt : vi_cpt VI cpt | ^;

cpt : cpt_el s_cpt;

s_cpt : CO def CF | ^;

cpt_el : TERME s_cpt_el;

s_cpt_el : PO arg0 PF | ^;

arg0 : idf0_arg vi_arg0 ;

vi_arg0 : vi_arg0 VI idf0_rg | ^ ;

idf0_arg : TERME | VAL ;
```

3 Exemple.

Cet exemple correspond à une partie de la dèfinition des concepts tels qu'ils pourraient l'être dans le cadre de l'application "renseignements administratifs".

```
%%
anime [personne [tiers,
                 locuteur,
                 système
                ],
       animal [domestique[chien,
                          chat,
                          poule,
                          canard
                         ],
               sauvage[lion,
                       singe,
                       autruche
                      ]
              ],
       administration
      ],
inanime [ lieu [ administration [impôts,
                                 mairie,
                                 commissariat
                                ],
                  pays [europe [ cee [france,
                                      belgique,
                                      italie,
                                      espagne
                                     ],
                                 yougoslavie,
                                 pologne
                                ],
                        amerique,
                        afrique,
                        oceanie,
                        asie
                       ]
                 ],
          document [ carte_d_identité,
                     passeport,
                     permis_de_séjour,
                     timbre_fiscal,
                     permis_de_pêche_sous_marine
                   ]
         ],
etat [ âge(personne, INT_100),
       sexe(personne)[ masculin,
                       féminin
                     ],
```

```
        couleur(inanime) [ bleu,
                           rouge,
                           blanc,
                           vert
                         ]
        lien_fam(personne1,personne2)
           [  parent [ père,
                       mère
                     ],
              fr_sr [ frère,
                      soeur
                    ]
              enfant [ fils,
                       fille
                     ]
           ],
        en_règle(personne),
        nationalité(personne,pays),
        état_civil(personne)[ marié,
                              célibataire,
                              veuf,
                              divorcé
                            ]
      ],
action [ aller(personne,lieu),
         obtenir(personne,document),
         rempli(personne,document),
         acompagner(personne1,personne2)
       ]
%%
```

4 La grammaire des formules.

Nous l'avons également précisé précédemment, la combinaison des concepts permet l'écriture de formules. L'écriture d'une formule doit satisfaire à la grammaire suivante :

```
fm : PO fm PF | idfl_cpt s_fm | NON fm | opbin s_fm | CHAINE
;

s_fm : PO fm vi_fm PF | ^

vi_fm : vi_fm VI fm | ^ ;

idfl_cpt : TERME s_idfl_cpt ;

s_idf_cpt : ENTIER | ^ ;

opbin : ET | OU ;
```

5 La grammaire de la base de données.

Il existe un certain nombre de faits qui sont prédéfinis pour le système. Ces faist constituent les faits par défaut qui sont considérés comme vrais jusqu'à preuve du contraire.

Il sont définis à l'aide des formules selon la grammaire suivante :

```
bd : fm_bd pv_bd ;

pv_bd : pv_bd PV fm_bd ;

fm_bd : fm ;
```

6 La grammaire des règles de réécriture.

Les règles de réécriture dont l'utilité a été décrites précédemment peuvent donc s'écrire en satisfaisant à la grammaire suivante :

```
base_rg : %% bs_rg %% ;

bs_rg : rg PT_VI bs_rg | ^ ;

rg : fm IM fm ;
```

7 Exemple de règles de réécriture.

Les règles présentées ici, sont données à titre d'exemple dans le cadre de l'application test développée actuellement à Nancy. Elles ne sont pas tirées directement d'une version opérationnelle du système et dès lors elles peuvent présenter certaines lacunes par rapport à la réalités. Cela ne nous paraît pas fondamental dans la mesure où il ne s'agit ici que d'exemples relatifs au formalisme et non au contenu des règles.

```
posséder(personne,carte_d_identité)
                    =>
                      nationalité(personne,france);

& (père(personne1,personne2),nationalité(personne1,france))
                    =>
                      nationalité(personne2,france);

&( -nationalité(personne,france),
    résider(personne,france),
    posséder(personne,permis_de_séjour))
                    =>
                     en_règle(personne)


père(personne1,personne2)
                    =>
                      masculin(personne1);

père(personne1,personne2)
                    =>
                      enfant(personne2,personne1)
```

```
&(frère(personne1,personne2),masculin(personne2))
                    =>
                      frère(personne2,personne1);

&(frère(personne1,personne2),féminin(personne2))
                    =>
                      soeur(personne2,personne1);

&(soeur(personne1,personne2),masculin(personne2))
                    =>
                      frère(personne2,personne1);

&(soeur(personne1,personne2),féminin(personne2))
                    =>
                      soeur(personne2,personne1);
%%
```

8 Définition de la grammaire des renseignements.

Les renseignement sont définis, eux aussi, à l'aide des formules et en respectant la grammaire suivante :

rens : rens_el pv_rens;

pv_rens : pv_rens PV rens_el | ^.

rens_el : BUT fm action condit ;

action : fm | ^ ;

condit : SI fm ALORS action condit SINON action condit | ^.

9 Exemples de renseignements.

De même que pour les règles de réécriture, le contenu des renseignements donné ici est purement exemplatif, le lecteur ne s' attardera donc pas inutilement aux questions de contenu.

Supposons que l'on souhaite exprimer une procédure simplifiée permettant l'obtention d'une carte d'identité en France.
Pour une personne majeure de nationalité française, il suffit de se présenter à la mairie.
Pour une personne mineure de nationalité française, il faut qu'elle se rende à la mairie en compagnie de son père ou de sa mère.
Pour toute personne ne possédant pas la nationalité française, il n'y a pas de solution.

```
BUT : obtenir(personne1,carte_d_identité)

SI : nationalité(persone1,france)

ALORS SI : majeur(personne1)

      ALORS : aller(personne1,mairie)

      SINON : &(aller(personne2,mairie),
                parent(personne2,personne1),
                accompagner(personne1,personne2)
               )
SINON :
```

Supposons que l'on veuille exprimer le fait que toute personne voulant accéder aux Etats-Unis et qui ne soit pas de nationalité américaine doivent posséder un visa valable.

```
BUT : accéder(personne,usa)

SI : -nationalité(personne,usa)

ALORS : ok

SINON : &(posséder(personne,visa),
          valable(visa),
         )
```

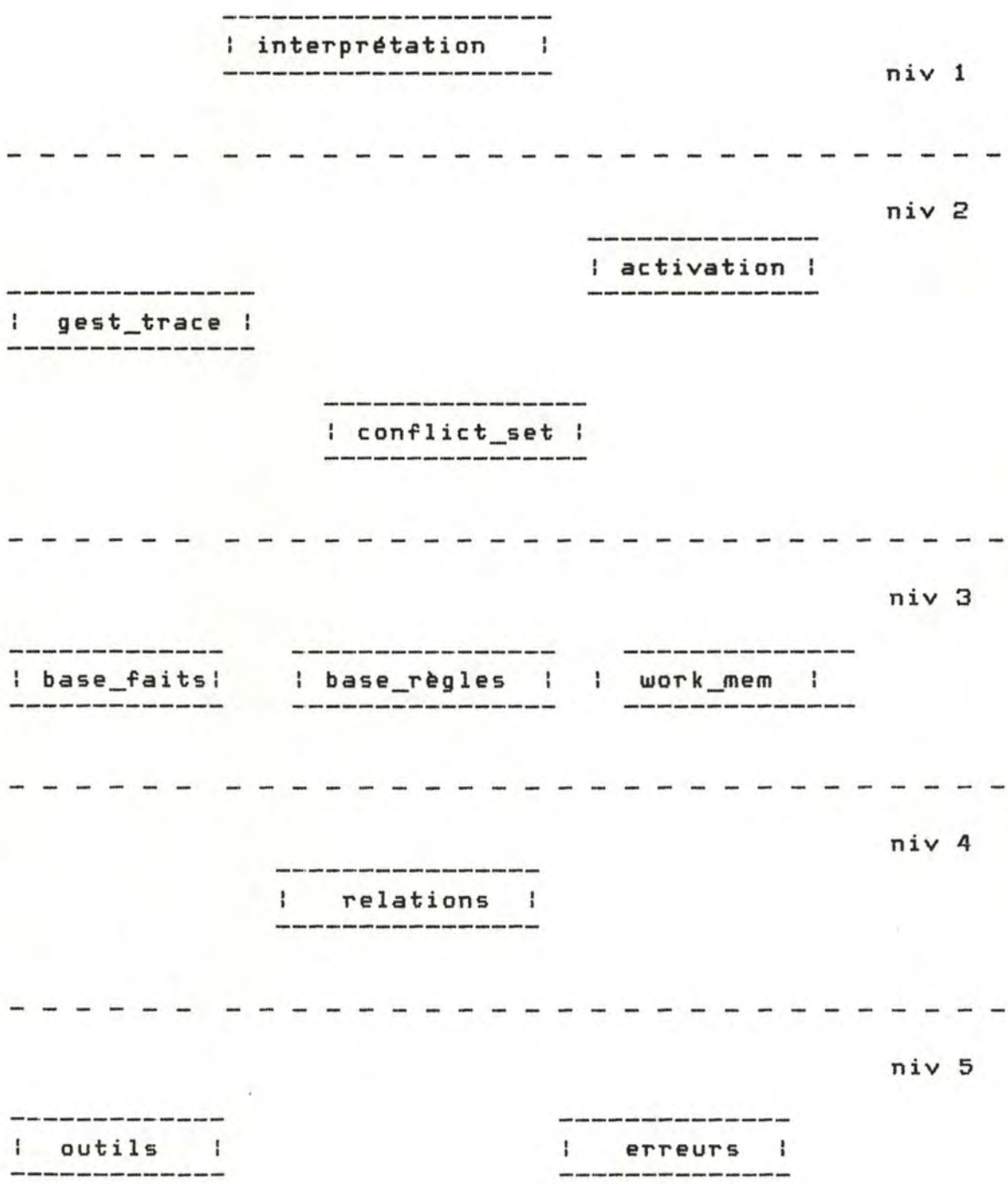
ANNEXE II : PROGRAMMES LIES A L'INTERPRETATION CONTEXTUELLE.

PARTIE II.1 : SPECIFICATIONS DES PROGRAMMES.

SPECIFICATIONS DU PROGRAMME INTERP.

Le programme "interp" correspond à l'interpréteur de règles.

Il est conçu comme un ensemble de modules organisés suivant une quasi-hiérarchie de type "utilise".



Nous parlons de "quasi-hiérarchie" car le problème lui-même n'est structurable hiérarchiquement qu'au prix d'artifices complexes, et ce en raison de la présence des appels récursifs à l'interpréteur au niveau des actions. A cette exception près, nous avons tenu à suivre ce mode d'organisation.

Pour la découpe en niveaux, nous avons tenté autant que possible de respecter un certain niveau d'abstraction.

Le niveau 5 regroupe 2 modules utilitaires.

Au dessus du niveau 4, on parle de relations uniquement.

Au dessus du niveau 3, les objets connus et manipulés sont la base de règles et les règles qu'elle contient, la base de faits et ses faits, et finalement la mémoire de travail et ses éléments. Les relations ne sont connues qu'indirectement en ce sens qu'elle sous-tendent tous ces objets, mais ne sont jamais référencées ou utilisées comme telles. D'ailleurs, on n'a aucune idée sur la façon dont elles sont accessibles.

Au dessus du niveau 2, on connait uniquement le conflict set ou l'activation d'instances de règles, et la trace d'exécution.

Enfin, l'utilisateur du module de niveau 1 ne connait du système que la fonction de contrôle de l'interpréteur.

La découpe en modules, elle, est basée sur un critère d'objets manipulés pour les niveaux inférieurs, et sur un citère de fonction pour les niveaux 2 et 1. A titre d'exemple, le module "relations" regroupe toutes les fonctions qui gèrent la représentation et l'utilisation des relations, tandis que le module activation est composé de fonctions coopérant pour la réalisation de l'activation d'une instance de règle.

Nous avons choisi ces critères car il nous semble que si des modifications doivent être apportées au système dans les niveaux inférieurs, c'est plutôt au niveau des objets que le système manipule que des fonctions qu'il réalise.

Pour les niveaux supérieurs, c'est la constatation inverse qui est vraie.

Par souci de clarté, nous présentons les modules et leur spécification de bas en haut, en remontant les différents niveaux d'abstraction.
Pour ce faire, nous utilisons un certain nombre de primitives ayant les significations suivantes :

* identificateur (obj, ens)

    cette fonction désigne l'identificateur de l'objet "obj" dans l'ensemble "ens".

* identifié(id, ens)

    cette fonction désigne l'objet d'identificateur "id" dans l'ensemble "ens".

* vide (obj)

    cette fonction signifie que l'objet "obj" est vide

* not(expr)

    cette fonction signifie que la négation de l'expression "expr" est vraie.

De même nous définissons une fois pour toutes le type LISTE.

- nom : LISTE
- type : liste

- contenu : - L : longueur de la liste
  type : entier

  - elem [1..?] : table des éléments.

Dans certains cas, nous utilisons d'autres fonctions; elles sont définies à l'endroit même où elles sont utilisées, ou elles correspondent à des fonctions du système proprement dit, et le lecteur est invité alors à consulter leur spécification précise.

Enfin, nous utilisons les constantes suivantes :

```
VRAI = 1
FAUX = 0

ECHEC = -1
ERREUR = 0
OK = 1
FIN = 2
RUNNING = 3
VIDE = 4

NMAXELEM = 100      nombre max d'éléments
NMAXCOND = 10       nombre max de conditions
NMAXACT = 10        nombre max d'actions
NMAXVAR = 5         nombre max de variables
```

```
NMAXRG = 100          nombre max de règles
NMAXARG = 5           nombre max d'arguments
NTYACT = 8            nombre max de type d'actions
NMAXSUB = 20          nombre max de substitution

OFF-LINE = 0
ON-LINE = 1

CONST = 0             indicateur de constante
VARIABLE = 1          indicateur de variable.
```

Pour la présentation des spécifications, nous avons respecté la démarche suivante : nous présentons d'abord l'interface du module, c'est-à-dire les objets et les spécifications abstraites des fonctions qui le constitue, puis la représentation interne des objets manipulés et les spécifications concrètes des fonctions.
On observera que, d'une manière générale, la représentation interne des objets est centrée sur les mêmes types que ceux utilisés pour la définition de l'interface.
Nous avons procédé de la sorte uniquement pour des raisons de facilité, et ceci n'enlève absolument rien à "l'étanchéité" des modules ou à leur capacité de cacher l'information autour de laquelle ils s'articulent.
De même, le lecteur remarquera certainement qu'il n'existe pas automatiquement de spécifications concrètes pour chaque fonction de l'interface.
Dans certains cas, il nous a paru en effet inutile de les écrire, les spécifications abstraites étant pleinement suffisantes pour la réalisation de leur développement.

UTILITAIRES.

1 Le module erreurs.

Ce module ne contient que des fonctions de type interface, c'est-à-dire des fonctions qui peuvent être utilisées par les modules supérieurs dans la quasi-hiérarchie.

1.1 Fonction erreur1().

* entrées : aucune.

* résultats : aucun

* fonction : affiche au terminal le message d'erreur correspondant à l'introduction d'un argument de relation non significatif.

1.2 Fonction erreur2().

* entrées : aucune.

* résultats : aucun

* fonction : affiche au terminal le message d'erreur correspondant à un échec de l'interprétation.

1.3 Fonction erreur3().

* entrées : aucune.

* résultats : aucun

* fonction : affiche au terminal le message d'erreur correspondant à une erreur ayant forcé une terminaison anormale de l'interpréteur.

1.4 Fonction erreur4().

* entrées : aucune.

* résultats : aucun

* fonction : affiche au terminal le message d'erreur correspondant à la production d'un diagnostic d'exécution inattendu.

1.5 Fonction erreur5().

* entrées : aucune.

* résultats : aucun

* fonction : affiche au terminal le message d'erreur correspondant à une erreur dans l'introduction d'un identificateur de règle.

1.6 Fonction erreur6().

* entrées : aucune.

* résultats : aucun

* fonction : affiche au terminal le message d'erreur correspondant à une incompréhension totale du message introduit par l'utilisateur.

2 Le module outils.

Ce module ne contient également que des fonctions de type interface. Il s'agit uniquement de fonctions outils utilisées par presque tous les autres modules.

2.1 Fonction search_T(tabc,l,chaine).

* entrées : tabc : tableau de chaines de caractères
  l : entier : longueur de ce tableau
  chaine : une chaine de caractères

* préconditions : 0 <= l <= 100

* résultats : search_T : entier

* postconditions : si il existe i tel que tabc[i]=chaine,
  search_T = i ;
  sinon, search_T = -1.

* fonction : recherche la chaine "chaine" dans la table "tabc". Si elle y est, retourne son indice. Sinon, retourne -1.

2.2 Fonction nom(tabc,ind).

* entrées : tabc : tableau de chaines de caractères
  ind : entier : indice d'une chaine dans ce tableau.

* préconditions : 0 <= ind <= 100

* résultats : nom : chaine de caractères.

* postconditions : nom=tabc[ind]

* fonction : retourne la chaine de caractères d'indice "ind" dans le tableau "tabc".

2.3 Fonction sort(t).

* entrées : t : liste d'entiers.

* préconditions : not (vide(t))

* résultats : t'

* postconditions : pour tout i (0<=i<=longueur(t)-1), on a t[i]<=t[i+1].

* fonction : trie un tableau d'entiers "t" par ordre alphabétique.

2.4 Fonction inclusion(a,b).

* entrées : a, b : listes d'entiers.

* préconditions : a,b <> VIDE.

* résultats : inclusion : entier

* postconditions : inclusion > 0 ssi a est strictement inclu dans b,
inclusion = 0 sinon.

* fonction : détermine si la liste d'entiers a est strictement incluse dans b. Retourne un entier nul sinon.

2.5 Fonction pres int(t,ent).

* entrées : t : liste d'entiers
ent : un entier

* préconditions : not (vide(t)).

```
*     résultats : pres_int : entier

*     postconditions : si il existe i tel que t[i]=ent,
                              pres_int=i ;
                       sinon, pres_int=-1.

*
      fonction :
      recherche si l'entier "ent" est présent dans la liste
      d'entiers "t".
      Si oui, retourne son indice dans cette liste.
      Sinon, retourne -1.
```

LE MODULE RELATIONS.

Ce module regroupe toutes les fonctions qui gèrent la représentation et l'utilisation des relations.

1 L'interface.

Pour les utilisateurs de ce module, les différents objets qu'ils peuvent manipuler sont des types suivants :

* un schéma de relation :

  _ nom : REL
  _ type : record

  _ contenu : _ NOM : nom identifiant la relation
                type : chaine de caractères.

              _ NB_ARG : nombre d'arguments
                type : entier positif.

* Une valeur d'un argument de relation :

  _ nom : VALEUR
  _ type : chaine de caractères.

* Une relation sous une forme complète :

  _ nom : REL_EXT
  _ type : record

  _ contenu : _ NOM : nom identifiant la relation
              type: chaine caractères

            _ NB_ARG: nombre d'arguments
              type : entier positif

            _ ARG [1..NMAXARG] : tableau des noms des arguments
              type : tableau de chaines de car.

* Une référence à la table des schémas de relations :

  _ nom : SCH_REL

_ type : pointeur

_ contenu : cette table contient a tout moment tous les schémas de relations connus du systèmes.

Il n'est pas nécessaire de connaitre sa structure pour l'utilisation de ce module.

* Une référence à la table de valeurs d'arguments de relations :

_ nom : VAL
_ type : pointeur

_ contenu : cette table contient à tout moment toutes les valeurs des arguments des relations.

L'utilisation de ce module ne nécessite pas de connaitre la structure de cette table.

Les fonctions suivantes sont également définies :

* ouvrir_rel()
* fermer_rel()
* aj_sch_rel()
* pres_sch_rel()
* schema_rel()

* anal_rel()

* ouvrir_val()
* fermer_val()
* aj_val()
* pres_val()
* nom_val()

1.1 Fonction ouvrir rel().

* entrées : aucune

* résultats : sch_rel : pointeur vers la table des schémas de relations.

* postconditions : ouvert(*sch_rel)

* fonction : ouverture de la table des schémas de relations.

1.2 Fonction fermer_rel().

* entrées : sch_rel : pointeur vers la table des schémas de relations

* préconditions : ouvert(*sch_rel)

* résultats : aucun.

* postconditions : aucune

* fonction : fermeture et sauvetage de la table des schémas de relations.

1.2.1 Fonction aj_sch_rel().

* entrées : r : une structure de type REL.
  sch_rel : pointeur vers la table des schémas de relations

* préconditions : ouvert(*sch_rel)

* résultats : *sch_rel'

  id : entier

* postconditions : *sch_rel' = *sch_rel + r
  id = identificateur(r,*sch_rel)

* fonction : ajouter le schéma de relation r dans la table des schémas de relation.
  Retourner l'identificateur de ce schéma dans la table.

1.3 Fonction pres_sch_rel().

* entrées :

  sch_rel : pointeur vers la table des schémas de relations
  nom_rel : chaine de caractères : nom de relation

nb_arg : entier : nombre d'arguments de la relation

* <u>préconditions</u> : ouvert(*sch_rel)

0 <= nb_arg <= NMAXARG

* <u>résultats</u> : pres_sch_rel : entier

* <u>postconditions</u> :

Si r, la relation identifiée par (nom_rel,nb_ar
appartient à la table des schémas de
relation,

alors pres_sch_rel=i
où i=identificateur(r,*sch_rel).

Sinon, pres_sch_rel=-1.

* <u>fonction</u> : Si la relation définie par "nom_rel" et "nb_arg" appartient à la table des schémas de relation, retourne son identificateur.
Sinon, retourne -1.

1.4 <u>Fonction schema_rel().</u>

* <u>entrées</u> :

sch_rel : pointeur vers la table des schémas de relations
id_rel : entier : identificateur d'une schéma de relation

* <u>préconditions</u> : ouvert(sch_rel)
id_rel >= 0

* <u>résultats</u> : schéma_rel = chaine de caractères.

* <u>postconditions</u> :

schéma_rel= nom(identifié(id_rel,*sch_rel))

* <u>fonction</u> : retourne le nom sous forme de chaine de caractères du schéma de relation identifié par id_rel dans la table référencée par sch_rel.

1.5 Fonction ouvrir val().

* entrées : aucune

* préconditions : aucune

* résultats : val : référence à la table des valeurs de relations

* postconditions : ouvert(*val)

* fonction : ouverture et chargement de la table des valeurs de relation.

1.6 Fonction fermer val().

* entrées : val : référence à la table des valeurs de relations

* préconditions : ouvert(*val)

* résultats : aucun.

* postconditions : aucune

* fonction : fermeture et sauvetage de la table des valeurs de relations.

1.7 Fonction aj val().

* entrées :

val : référence à la table des valeurs de relations
valeur : chaine de charactères : une valeur de relations.

* préconditions : ouvert(*val)

* résultats : aj_val = entier
*val'

* postconditions : *val' = *val + valeur
aj_val = identificateur (valeur,*val)

* fonction : ajouter la valeur de relation "valeur" dans la table référencée par "val" et retourner son identificateur dans cette table.

1.8 Fonction pres val().

* entrées :

val : référence à la table des valeurs de relations
v : chaine de caractères : une valeur de relation

* préconditions : ouvert(*val)

* résultats : pres_val = entier

* postconditions :

si v appartient à *val,

alors pres_val=identificateur(v,val),
sinon, pres_val=-1.

* fonction : vérifie si la valeur de relation "v" apparitent à la table référencée par "val".
Si oui, retourne son identificateur dans cette table.
Sinon, retourne -1.

1.9 Fonction nom val().

* entrées :

val : référence à la table des valeurs de relations
id_val : entier : identificateur de valeur de

relation.

* préconditions : ouvert(*val)
  id_val >=0

* résultats : nom_val = chaine de caractères

* postconditions : nom_val = identifié(id_val,*val)

* fonction : étant donné un identificateur de valeur "id_val", retourne cette valeur sous forme d'une chaine de caractères.

1.10 Fonction anal rel().

* entrées : arg : chaine de caractères : une relation complète sous forme de chaine de charactères.

* préconditions : aucune.

* résultats : rx : une référence à une relation de type REL_EXT.

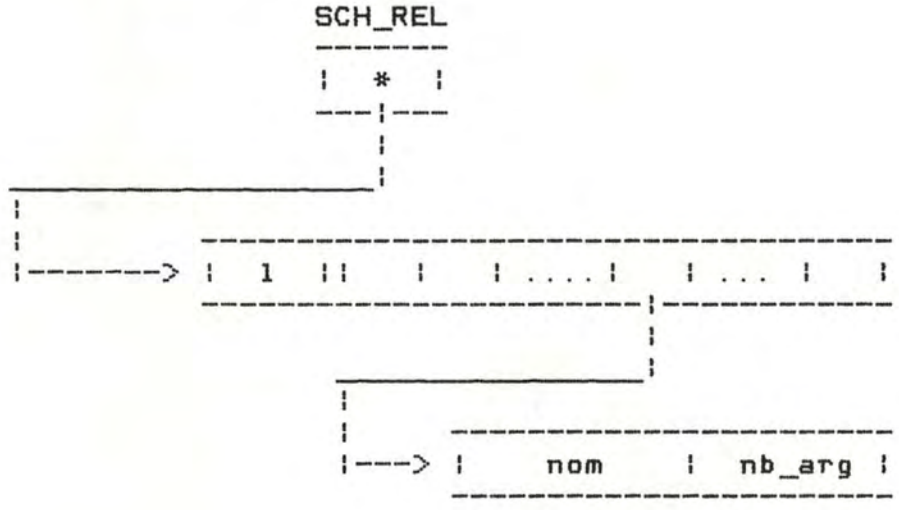
* postconditions : rx <> NULL

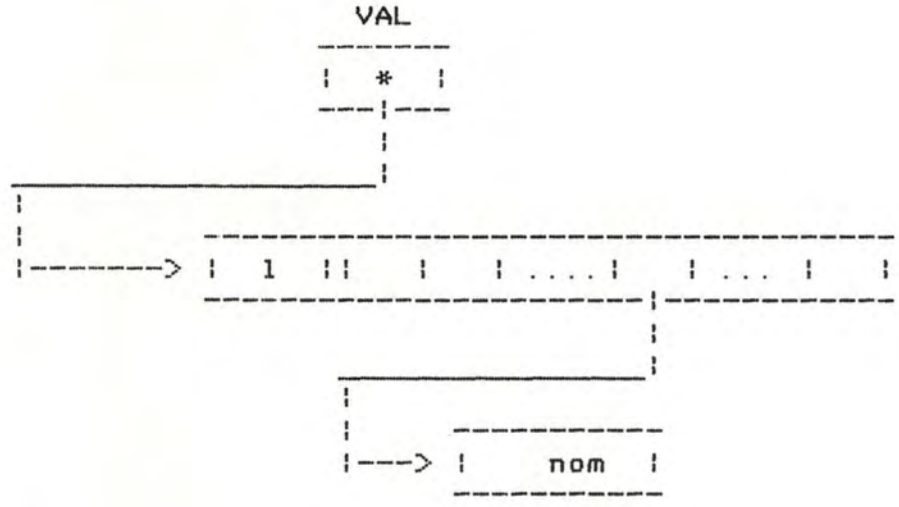
* fonction : transforme la chaine de caractères "arg" en un objet de type REL_EXT et retourne un pointeur vers cet objet.

2 Spécifications concrètes.

2.1 Représentation interne.

La représentation interne est très proche de celle de l'interface.





* structure de la table de schémas de relations.

- nom : RELAT
- type : liste de REL

- référence : SCH_REL

* structure de la table de valeurs d'arguments de relation.

_ nom : VALEURS
_ type : liste de VAL

- référence : VAL

* structure du fichier "relations.file" :

- nom : RELATIONS.FILE
- type des éléments : REL.

* structure du fichier "valeurs.file" :

- nom : VALEURS.FILE
- type des éléments : chaîne de caractères.

2.2 Fonction ouvrir_rel().

* entrées : aucune

* résultats : sch_rel : pointeur vers la table des schémas de relations.

* postconditions : ouvert(*sch_rel),
sch_rel<>NULL,
vide(*sch_rel) => sch_rel->l=0,
*sch_rel contient tous les schémas de relation définis dans le système.

* fonction : Création de la table des schémas de relations, et chargement du contenu du fichier "relations.file" dans cette table.
Si ce fichier n'existe pas encore, le créer.
Mise à jour de sch_rel->l.

2.3 Fonction fermer_rel().

* entrées : sch_rel : pointeur vers la table des schémas de relations

* préconditions : ouvert(*sch_rel)

* résultats : aucun.

* postconditions : aucune

* fonction : fermeture et sauvetage de la table des schémas de relations dans le fichier "relations.file".

2.4 Fonction aj_sch_rel().

* entrées : sch_rel : pointeur vers la table des schémas de relations;
r : une structure de type REL.

* préconditions : ouvert(*sch_rel)

* résultats : *sch_rel'
id: entier

* postconditions : *sch_rel' = *sch_rel + r
sch_rel->ptr_rel[i]=r => id=i

* fonction : ajouter le schéma de relation r dans la table des schémas de relation.
Retourne son identificateur dans cette table.

2.5 Fonction pres_sch_rel().

* entrées : sch_rel : pointeur vers la table des schémas de relations
nom_rel : chaine de caractères : nom de relation
nb_arg : entier : nombre d'arguments de la relation

* préconditions : ouvert(*sch_rel)
0 <= nb_arg <= NMAXARG

* résultats : pres_sch_rel : entier

* postconditions :

Si r, la relation identifiée par (nom_rel,nb_arg) appartient à la table des schémas de relation,

alors pres_sch_rel=i
où i= est tel que sch_rel->ptr_rel[i]=r.

Sinon, pres_sch_rel=-1.

* fonction : Si la relation définie par "nom_rel" et "nb_arg" appartient à la table des schémas de relation, retourne son identificateur.
Sinon, retourne -1.

2.6 Fonction schema_rel().

* entrées : sch_rel : pointeur vers la table des schémas de relations
id_rel : entier : identificateur d'une schéma de relati

* préconditions : ouvert(*sch_rel)
id_rel >= 0

* résultats : schéma_rel = chaine de caractères.

* postconditions :
schéma_rel= sch_rel->ptr_rel[id_rel]->nom

* fonction : retourne le nom sous forme de chaine de caractères du schéma de relation identifié par id_rel dans la table référencée par sch_rel.

2.7 Fonction ouvrir val().

* entrées : aucune

* préconditions : aucune

* résultats : val : référence à la table des valeurs de relations,

* postconditions : ouvert(*val)

  val<> NULL,

  vide(val) => val->l=0,

  *val contient toutes les valeurs de relations définies dans le système.

* fonction : ouverture et chargement de la table des valeurs de relation à partir du contenu du fichier "valeurs.file".
  Création de ce fichier s'il n'existe pas encore.
  mise à jour de val->l

2.8 Fonction fermer val().

* entrées : val : référence à la table des valeurs de relations

* préconditions : ouvert(*val)

* résultats : aucun.

* postconditions : aucune

* fonction : fermeture et sauvetage de la table des valeurs de relations dans le fichier "valeurs.file".

2.9 Fonction aj_val().

* entrées : val : référence à la table des valeurs de relations
valeur : chaine de charactères : une valeur de relation

* préconditions : ouvert(*val)

* résultats : aj_val = entier
*val'

* postconditions : *val' = *val + valeur

aj_val = i tel que val->ptr_val[i]=valeur

* fonction : ajouter la valeur de relation "valeur" dans la table référencée par "val" et retourner son identificateur dans cette table.

2.10 Fonction pres_val().

* entrées : val : référence à la table des valeurs de relations
v : chaine de caractères : une valeur de relations

* préconditions : ouvert(*val)

* résultats : pres_val = entier

* postconditions :

pres_val = i s'il existe i tel que val->ptr_val[i]=valeur;

pres_val=-1 sinon.

* fonction : vérifie si la valeur de relation "v" apparitent à la table référencée par "val".
Si oui, retourne son identificateur dans cette table.
Sinon, retourne -1.

2.11 Fonction nom_val().

* **entrées** : val : référence à la table des valeurs de relations
id_val : entier : identificateur de valeur de relation

* **préconditions** : ouvert(*val)
id_val >=0

* **résultats** : nom_val = chaine de caractères

* **postconditions** : nom_val = val->ptr_val[id_val]

* **fonction** : étant donné un identificateur de valeur "id_val", retourne cette valeur d'argument de relation sous forme de chaine de caractères.

LE MODULE WORKMEM.

Ce module contient toutes les fonctions de représentation et d'utilisation de la mémoire de travail.

1 L'interface.

Les objets concernant la mémoire de travail ont, pour l'utilisateur de ce module, les formats suivants :

* un élément de mémoire de travail, premier format :

  - nom : EL_EXTERNE
  - type : record
  - contenu : - R : référence à un schéma de relation.
    type : pointeur vers une REL.
    - ARG[1..NMAXARG] : table de références aux valeurs des arguments.
    type : table de pointeurs vers des chaines de caractères.

* un élément de mémoire de travail, deuxième format:

  - nom : EL_WM
  - type : record
  - contenu : - ID_REL : identificateur de la relation dont il est l'instance ;
    type : entier positif
    - AGE : l'âge de cet élément en mémoire de travail, i.e. le nombre de cycles de l'interpréteur qui se sont déroulés depuis son apparition.
    type : entier positif
    - NB_ARG : le nombre d'arguments de la relation.
    type : entier positif
    - ARG[1..NMAXARG] : table d'identificateurs des valeurs des arguments.

* une liste d'éléments de mémoire de travail :

- nom : L_EL_WM
- type : liste

- contenu : une suite d'éléments de type
  pointeur vers des EL_WM.

* une liste d'identificateurs d'éléments de mémoire
  de travail :

- nom : L_ID_EL
- type : liste

- contenu : une suite d'identificateurs d'éléments
  de mémoire de travail.
  type de l'identificateur : entier positif.

La mémoire de travail est référencée de la manière suivante :

- nom : WM
- type : pointeur vers la mémoire de travail.

Les fonctions suivantes sont définies :

* ouvrir_wm()
* aj_wm()
* aj_wm_ext()
* ac_wm()
* ac_nouv_el_wm()
* id_classe()
* ac_wm_val()
* pres_el_wm()
* structure()
* maj_wm()
* aff_l_el_wm()
* aff_wm()
* int_el_wm()

1.1 Fonction ouvrir wm().

* entrées : aucune.

* préconditions : aucune.

* résultats : wm : référence à la mémoire de travail.

* postconditions : ouvert(*wm)

* fonction : ouverture de la mémoire de travail.

1.2 Fonction aj_wm().

* entrées : wm : référence à la mémoire de travail.
el : EL_WM : un élément de mémoire de travail

* préconditions : ouvert(*wm)

* résultats : *wm'
id : entier

* postconditions : *wm' = *wm + el
id = identificateur(el,*wm)

* fonction : ajouter l'élément "el", deuxième format, en mémoire de travail.
Retourner son identificateur dans cette mémoire de travail.

1.3 Fonction aj_wm_ext().

* entrées : wm : référence à la mémoire de travail.
el : EL_EXT : un élément de mémoire de travail

* préconditions : ouvert(*wm)

* résultats : *wm'
id : entier

* postconditions : *wm' = *wm + el
id = identificateur(el,*wm)

* fonction : ajouter l'élément "el", premier format, en mémoire de travail.
Retourner son identificateur dans cette mémoire de

travail.

1.4 Fonction ac wm().

* entrées : wm : référence à la mémoire de travail.
id_rel : entier : identificateur de relation

* préconditions : ouvert(*wm)
id_rel >=0

* résultats : l : pointeur vers une liste d'éléments de mémoire de travail de type L_EL_WM.

* postconditions :

Pour tout el appartenant à l :
id_classe(el)=id_rel.

vide(*l) => l=NULL.

* fonction : crèe une liste de tous les éléments de mémoire de travail ayant comme identificateur de relation "id_rel".
Retourne un pointeur vers cette liste.

1.5 Fonction ac nouv el wm().

* entrées : wm : référence à la mémoire de travail.
cycle : entier

* préconditions : ouvert(*wm)
cycle >=0

* résultats : l : un pointeur vers une liste d'identificateurs d'éléments de mémoire de travail de type L_ID_EL.

* postconditions : Pour tout el appartenant à l :
identifié(el,*wm)->age = cycle-1.

vide(*l) => l=NULL.

* fonction : crèe une liste de tous les éléments de mémoire de travail ayant un age égal à "cycle"-1. Retourne un pointeur vers cette liste.

1.6 Fonction id classe().

* entrées : wm : référence à la mémoire de travail.
el : entier : identificateur d'élément de mémoire de travail

* préconditions : ouvert(*wm)
el >=0

* résultats : id : entier

* postconditions : identifié(el,*wm)->id_rel=id

* fonction : retourne l'identificateur de classe, c'est-à-dire l'identificateur de relation de l'élément de mémoire de travail identifié par "el".

1.7 Fonction ac wm val().

* entrées : wm : référence à la mémoire de travail.
v : identificateur d'une valeur de relation

* préconditions : ouvert(*wm)
v >=0

* résultats : l : référence à une liste d'éléements de mémoire de travail de type L_EL_WM.

* postconditions :

Pour tout el appartenant à l,
il existe i (0<=i<=NMAXARG) tel que

el->arg[i]=v.

vide(l) => l->l=0.

* fonction : Création d'une liste de tous les éléments de mémoire de trvail ayant la valeur identifiée par "v" comme une de leurs arguments.
Retourner son identificateur dans cette mémoire de travail.

1.8 Fonction pres el wm().

* entrées : wm : référence à la mémoire de travail.
el : EL_WM : un élément de mémoire de travail

* préconditions : ouvert(*wm)

* résultats : id : entier

* postconditions : Si el appartient à *wm,
id = identificateur(el,*wm).
Sinon, id = -1.

* fonction : recherche si l'élément "el" appartient à la mémoire de travail référencée par "wm".
Si oui, retourne son identificateur dans cette mémoire de travail.
Sinon, retourne -1.

1.9 Fonction structure().

* entrées : wm : référence à la mémoire de travail.
v : entier : identificateur de valeur de relation.

* préconditions : ouvert(*wm)
v >= 0

* résultats : l : pointeur vers une liste d'identificateurs de valeurs.

* postconditions : Pour tout id appartenant à l :
id identifie une valeur de relation appartenant à la structure dont la valeur identifiée par

v est racine.

vide(l) => l->l = 0.

* fonction : recherche toutes les valeurs, càd les mots du lexique qui sont dépendants de celle identifiée par "v" dans l'énoncé courant contenu en mémoire de travail.
Retourne un pointeur vers la liste d'identificateurs de ces valeurs. en mémoire de travail.

1.10 Fonction maj wm().

* entrées : wm : référence à la mémoire de travail.
str : booléen : indicateur de structure.
v : entier : identificateur de valeur.

* préconditions : ouvert(*wm)
v >= 0

* résultats : *wm'

* postconditions :

pour tout el appartenant à *wm',
il existe i tel que :

(el->arg[i]=v)

OU [(el->arg[i] appartient à structure(v))
ET (str=vrai)]

* fonction : extrait de la mémoire de travail tout élément qui n'a pas comme argument la valeur identifiée par v si "str" est faux ou tout élément n'ayant pas cette valeur comme argument ou une valeur qui appartient à la structure dépendante d'elle si "str" est vrai.

1.11 Fonction aff_l_el_wm().

* entrées : fich : un fichier de type texte.
l : un pointeur vers une liste d'éléments de mémoire de travail de type L_EL_WM

* préconditions : l <> NULL
vide(l) => l->l=0

* résultats : fich'

* postconditions : pour tout el appartenant à l , el = chaine de caractères correspondant à un et un seul élément de la liste "l".

* fonction : transformations de tous les éléments de la liste "l" en chaines de caractères, et stockage dans le fichier "fich" avec, en regard de chaque élément, leur âge et leur identificateur.

1.12 Fonction aff_wm().

* entrées : wm : référence à la mémoire de travail.
fich : un fichier texte.

* préconditions : ouvert(*wm)

* résultats : fich'

* postconditions : fich' contient tous les éléments appartenant à la mémoire de travail sous forme de chaines de caractères, ainsi que leur âge respectif et leur identificateur.

* fonction : transformation de tous les éléments de mémoire de travail en chaines de caractères, et stockage dans le fichier "fich" avec en regard leur âge et leur identificateur.

1.13 Fonction int el wm().

* entrées : wm : référence à la mémoire de travail.

* préconditions : ouvert(*wm)

* résultats : *wm'

* postconditions : *wm' = *wm + les éléments de mémoire de travail introduits au terminal et transformés sous une forme interne.

* fonction : Introduction d'éléments de mémoire de travail au terminal : saisie de ces éléments au terminal sous forme de chaine de caractères, transformation et ajout dans la mémoire de travail avec mise à jour de leur âge (âge=0).

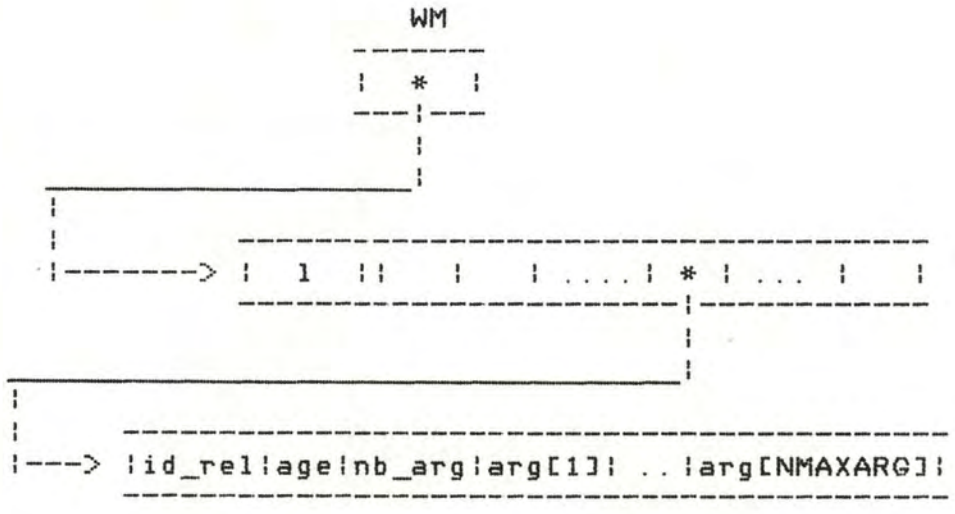
2 Spécifications concrètes.

2.1 Représentation interne.

Actuellement, la représentation interne de la mémoire de travail correspond à un objet de type L_EL_WM défini dans l'interface.

- nom : mémoire de travail
- type : L_EL_WM

- référence : WM



2.2 Fonction ouvrir_wm().

* entrées : sch_rel , val

* préconditions : aucune.

* résultats : wm : référence à la mémoire de travail.

* postconditions : ouvert(*wm) ET ouvert(*sch_rel) ET ouvert(*val).

* fonction : ouverture de la mémoire de travail, et ouverture de la table des schémas de relations référencée par "sch_rel" et de la table des valeurs référencée par "val", si ce n'est déjà fait.

2.3 Fonction aj_wm().

* entrées : wm : référence à la mémoire de travail.
el : EL_WM : un élément de mémoire de travail

* préconditions : ouvert(*wm)

* résultats : *wm'
id : entier

* postconditions : *wm' = *wm + (wm->elem[wm->l]=el);
*wm'->l = *wm->l + 1;
id= wm->l.

* fonction : ajouter l'élément "el", deuxième format, en mémoire de travail.
Retourner son identificateur dans cette mémoire de travail.

2.4 Fonction aj_wm_ext().

* entrées : wm : référence à la mémoire de travail.
el : EL_EXT : un élément de mémoire de travail

* préconditions : ouvert(*wm)

* résultats : *wm'
id : entier

* postconditions : *wm' = *wm + wm->elem[wm->l];
*wm'->l = *wm->l + 1;
id= wm->l.

* fonction : ajouter l'élément "el", premier format, en mémoire de travail, après l'avoir transformé en second format.
Retourner son identificateur dans cette mémoire de

travail.

2.5 Fonction ac nouv el wm().

* entrées : wm : référence à la mémoire de travail.
  cycle : entier

* préconditions : ouvert(*wm)
  cycle >=0

* résultats : l : un pointeur vers une liste d'identificateurs d'éléments de mémoire de travail de type L_ID_EL.

* postconditions : Pour tout el appartenant à l :

  el=entier positif;
  wm->elem[el]->age=cycle-1;
  vide(*l) => l=NULL.

* fonction : crée une liste de tous les éléments de mémoire de travail ayant un age égal à "cycle"-1. Retourne un pointeur vers cette liste.

2.6 Fonction id classe().

* entrées : wm : référence à la mémoire de travail.
  el : entier : identificateur d'élément de mémoire de travail

* préconditions : ouvert(*wm)
  el >=0

* résultats : id : entier

* postconditions : wm->elem[el]->id_rel=id.

* fonction : retourne l'identificateur de classe, c'est-à-dire l'identificateur de relation de l'élément de mémoire de travail identifié par "el".

2.7 Fonction pres_el_wm().

* entrées : wm : référence à la mémoire de travail.
el : EL_WM : un élément de mémoire de travail

* préconditions : ouvert(*wm)

* résultats : id : entier

* postconditions : Si el appartient à *wm,

id est tel que wm->elem[id]=el

Sinon, id = -1.

* fonction : recherche si l'élément "el" appartient à la mémoire de travail référencée par "wm".
Si oui, retourne son identificateur dans cette mémoire de travail.
Sinon, retourne -1.

LE MODULE BASE FAITS.

Le module "base_faits" regroupe toutes les fonctions de gestion de la représentation et de l'utilisation de la base de faits.

1 L'interface.

Pour les utilisateurs de ce module, les différents objets qu'ils peuvent manipuler possèdent un des types suivants:

* un fait :

  - nom : FAIT
  - type : record

  - contenu : - ID_REL : l'identificateur du schéma de relation dont il est une instance.
    type : entier positif

    - ARG : identificateur de l'argument de cette relation
      type : entier positif

* une liste de faits :

  - nom : L_FAITS
  - type : liste

  - contenu : une suite d'éléements de type pointeur vers un FAIT.

La base de faits sera référencée de la manière suivante :

- nom : BF
- type : pointeur vers la base de faits

- contenu : la base de faits contient tous les faits définis dans le système.

Il n'est pas nécessaire de connaitre le mode de représentation de la base de faits pour l'utilisation de ce module.

Les fonctions suivantes sont également définies :

* ouv_bf()
* ferm_bf()
* ac_bf()
* ac_bf_val()
* aff_bf()
* intro_fait()
* suppr_fait()
* modif_fait()
* delete_bf()

1.1 Fonction ouv bf().

* entrées : aucune.

* préconditions : aucune.

* résultats : bf: référence à la base de faits.

* postconditions : ouvert(*bf)

* fonction : ouverture et chargement de la base de faits.

1.2 Fonction ferm bf().

* entrées : bf : référence à la base de faits.

* préconditions : ouvert(*bf)

* résultats : aucun.

* postconditions : aucune.

* fonction : fermeture et sauvetage de la base de faits.

1.3 Fonction ac_bf().

* entrées : bf : référence à la base de faits.
  id_rel : entier : identificateur de relation.

* préconditions : ouvert(*bf)

* résultats : l : une référence à une liste de faits.

* postconditions : Pour tout el appartenant à l :
  el->id_rel=id_rel.

  vide(l) => l=NULL.

* fonction : Retourne un pointeur vers une liste de faits ayant un identificateur de relation identique à "id_rel" passé en paramètre.

1.4 Fonction aff_bf().

* entrées : bf : référence à la base de faits.
  fich : un fichier de type texte.

* préconditions : ouvert(*bf)

* résultats : fich'

* postconditions : fich' contient tous les éléments de la base de faits sous forme de chaines de caractères

* fonction : transformation de tous les faits de la base de faits sous forme de chaines de caractères, et stockage dans le fichier "fich" avec en regard leur identificateur.

1.5 Fonction intro fait".

* entrées : bf : référence à la base de faits.

* préconditions : ouvert(*bf)

* résultats : bf'

* postconditions : bf' = bf + tous les faits tapés au terminal et transformés sous forme interne.

* fonction : Introduction des faits au terminal.
Lecture de ces faits sous forme de chaines de caractères, transformation en représentation interne, et ajout à la base de faits.

1.6 Fonction suppr fait().

* entrées : bf : référence à la base de faits.
id_fait : entier : identificateur de faits.

* préconditions : ouvert(*bf)

* résultats : bf'
status : entier.

* postconditions : si identifié(id_fait,*bf) existe :

bf' = bf - identifié(id_fait,*bf)
status=OK.

sinon bf' = bf
status = ERREUR.

* fonction : S'il existe un fait identifié par "id_fait" dans la base de faits, suppression de ce fait de la base et reourner "status"=OK.
Sinon, la base reste inchangée et "status"=ERREUR.

1.7 Fonction modif fait().

* entrées : bf : référence à la base de faits.
id_fait : entier : un identificateur de fait.

* préconditions : ouvert(*bf)

* résultats : bf'
status : entier

* postconditions : Si identifié(id_fait,*bf) existe :

bf' = bf où suppr_fait(id_fait) et ajout
d'un nouveau fait introduit
au terminal.
status = OK

sinon, bf' = bf
status = ERREUR.

* fonction : S'il existe un fait identifié par "id_fait", alors modification de ce fait en le supprimant et en saisissant un nouveau au terminal puis en le rajoutant en base de faits. Dans ce cas, "status" = OK.
Sinon, la base de faits reste inchangée et "status"=ERREUR.

1.8 Fonction delete bf().

* entrées : bf : référence à la base de faits.

* préconditions : ouvert(*bf)

* résultats : *bf'

* postconditions : vide(*bf)

* fonction : destruction du contenu de la base de faits.

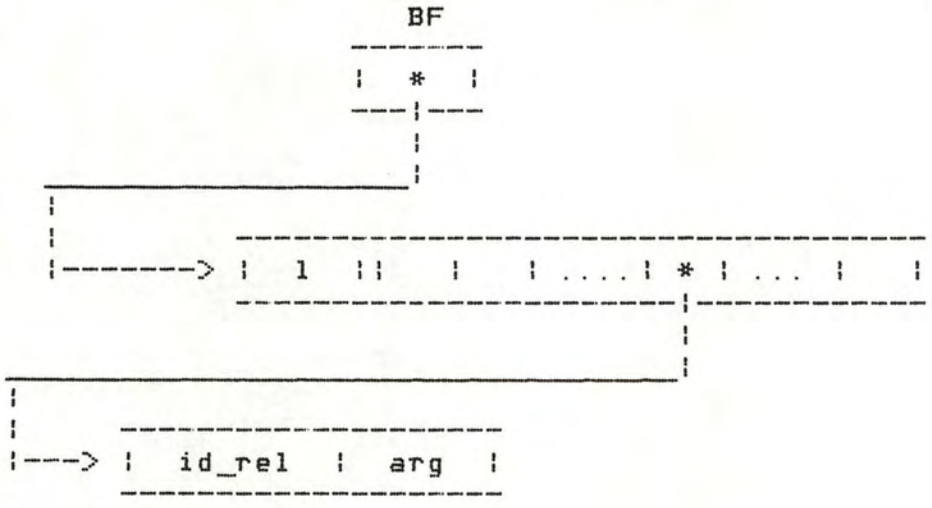
2 Spécifications concrètes.

2.1 Représentation interne.

Actuellement, la base de faits est représentée comme une structure de type L_FAITS.

- nom : base de faits
- type : L_FAITS

- référence : BF



* structure du fichier "b_faits.file" :

- nom : B_FAITS.FILE
- type des éléments : FAIT

2.2 Fonction ouv_bf().

* entrées : sch_rel, val

* préconditions : aucune.

* résultats : bf: référence à la base de faits.

* postconditions : ouvert(*bf) ET ouvert(*sch_rel) ET ouvert(*val)
*bf contient l'ensemble des faits définis dans le système.

* fonction : ouverture et chargement de la base de faits à partir du contenu du fichier "b_faits.file", et mise à jour de bf->l.
Si la table des schémas de relations et la table des valeurs ne sont pas encore ouvertes, les ouvrir.

2.3 Fonction ferm_bf().

* entrées : bf : référence à la base de faits.

sch_rel : référence à la table des schémas de relations.

val : référence à la table des valeurs de relations.

* préconditions : ouvert(*bf)

* résultats : aucun.

* postconditions : aucune.

* fonction : fermeture de la base de faits et sauvetage de son contenu dans le fichier "b_faits.file".
Fermeture de la table des schémas de relations et de la table des valeurs de relations, si ce n'est déjà fait.

2.4 Fonction suppr fait().

* entrées : bf : référence à la base de faits.
id_fait : entier : identificateur de faits.

* préconditions : ouvert(*bf)

* résultats : bf'
status : entier.

* postconditions : si (id_fait<bf->l)ET(id_fait>=0

bf' = bf - bf->elem[id_fait]
status=OK.

sinon bf' = bf
status = ERREUR.

* fonction : S'il existe un fait identifié par "id_fait" dans la base de faits, suppression de ce fait de la base et reourner "status"=OK.
Sinon, la base reste inchangée et "status"=ERREUR.

2.5 Fonction modif fait().

* entrées : bf : référence à la base de faits.
id_fait : entier : un identificateur de fait.

* préconditions : ouvert(*bf)

* résultats : bf'
status : entier

* postconditions :

si (id_fait<bf->l)ET(id_fait>=0)

bf' = bf où suppr_fait(id_fait) et ajout
d'un nouveau fait introduit
au terminal dans bf->elem[bf->l]

bf'->l = bf->l + 1

status = OK

sinon, bf' = bf

status = ERREUR.

* fonction : S'il existe un fait identifié par "id_fait", alors modification de ce fait en le supprimant et en saisissant un nouveau au terminal puis en le rajoutant en base de faits. Dans ce cas, "status" = OK.
Sinon, la base de faits reste inchangée et "status"=ERREUR.

2.6 Fonction aj bf().

* entrées : bf : référence à la base de faits.
f : une référence à un objet de type FAIT.

* préconditions : ouvert(*bf)
f<>NULL

* résultats : bf'

* postconditions : bf' = bf + (bf->elem[bf->l]=f)
bf'->l = bf->l + 1.

* fonction : ajouter le fait référencé par "f" dans la base de faits.

2.7 Fonction ac bf val().

* entrées : bf : référence à la base de faits.
v : entier : identificateur de valeur de relation.

* préconditions : ouvert(*bf)
v >= 0

* résultats : l : une référence à une liste de faits.

* postconditions : Pour tout el appartenant à l :

el->arg = v.

vide(l) => l=NULL.

* fonction : Retourne un pointeur vers une liste de faits ayant un identificateur de valeur de relation identique à "v" passé en paramètre.

2.8 Fonction int fait().

* entrées : bf : référence à la base de faits.

* préconditions : ouvert(*bf)

* résultats : bf'

* postconditions :

bf' = bf + (bf->elem[bf->l]=représentation interne d'un fait lu au terminal).

bf'->l = bf->l + 1

* fonction : lit au terminal un fait sous forme de chaine decaractères, le transforme en représentation interne, et le rajoute en base de faits.

LE MODULE BASE REGLES.

Le module "base_règles" contient toutes les fonctions assurant la gestion de la représentation de la base de règles et des règles qu'elle contient.

1 L'interface.

Pour l'utilisateur du module, les différents objets manipulés ont un des types suivants :

* une condition d'une règle, premier format :

  - nom : COND_EXTERNE
  - type : record
  - contenu : - NOM : nom identifiant un type de condition.
    type : chaine de caractères.

    - NB_ARG : nombre d'arguments de la condition.
      type : entier positif

    - TAB_VAR : table des arguments de cette condition
      type : table de chaines de caractères.

* une condition de règle, second format :

  - nom : COND
  - type : record
  - contenu : - ID_REL : un identificateur de relation identifiant le type de condition.
    type : entier positif.

    - L_VAR : une table d'identificateurs de variables présentes dans la condition.
      type : table d'entiers positifs.

* une action de règle, premier format :

  - nom : ACT_EXTERNE
  - type : record
  - contenu : - NOM_ACT : nom identifiant l'action
    type : chaine de caractères.

    - ARG : argument de l'action
      type : chaine de caractères.

* une action de règle, second format :

- nom : ACT
- type : record

- contenu : - ID_ACT : identificateur de l'action.
type : entier positif

- L_ARG : longueur de l'argument
type : entier positif

- ARG : référence à l'argument
type : pointeur.

* un argument d'un ordre d'interprétation :

- nom : ARG_INTERP
- type : record

- contenu : - STRC : indicateur de structure ou d'élément terminal. Il sert à indiquer si l'action porte sur la structure dépendant de la variable, ou sur la variable elle-même en tant qu'élément terminal.
type : booléen.

- VAR : identificateur de la variable concernée.
type : entier positif

* un argument d'une action sur la mémoire de travail :

- nom : ARG_WM
- type : record

- contenu : - ID_REL : identificateur de la relation dont l'argument constitue une instance.
type : entier positif

- NB_ARG : nombre d'argument de cette relation.
type : entier positif.

- ARGREL : table des identificateurs des arguments de la relation
type : table d'entiers positifs.

* un argument d'un ajout d'interprétation :

- nom : ARG_IN
- type : record

- contenu : - T_ARG : indicateur du type d'argument : variable

ou constante.
type : booléen.

- V : identificateur de la variable concernée
type : entier

OU - CONST : la constante concernée.

* un argument d'accès à l'historique pour les ellipses :

- nom : ARG_ELLIPSE
- type : record

- contenu : - T_ARG : indicateur du type d'argument : variable
type: booléen.

- V : identificateur de la variable concernée
type : entier positif

OU - CONST : la constante concernée.

* une liste de conditions :

- nom : LS_COND
- type : liste de COND

* une liste d'actions :

- nom : LS_ACT
- type : liste de ACT

* une liste d'identificateurs de règles :

- nom : L_ID_REGLES
- type : liste

- contenu : une suite d'identificateurs de règles
type des identificateurs : entier positif

La base de règles est référencée de la manière suivante :

- nom : BR
- type : pointeur vers la base de règles.

- contenu : la base de règles contient toutes les
règles définies dans le système.

Les fonctions suivantes sont également définies :

* ouvrir_br()
* fermer_br()
* aff_br()
* premisse()
* consequent()
* accès_rg_cdts()
* généralité()
* max_contrainte()
* intro_règles()
* suppr_règles()
* modif_règles()
* delete_br()

1.1 Fonction ouvrir br().

* entrées : aucune.

* préconditions : aucune.

* résultats : br: référence à la base de règles.

* postconditions : ouvert(*br)

* fonction : ouverture et chargement de la base de règles.

1.2 Fonction fermer br().

* entrées : br : référence à la base de règles.

* préconditions : ouvert(*br)

* résultats : aucun.

* postconditions : aucune.

* fonction : fermeture et sauvetage de la base de règles.

1.3 Fonction aff br().

* entrées : br : référence à la base de règles.
fich : un fichier de type texte.

* préconditions : ouvert(*br)

* résultats : fich'

* postconditions : fich' contient toutes les règles de la base de règles sous forme de chaines de caractères.

* fonction : transformation de toutes les règles de la base de règles sous forme de chaines de caractères, et stockage dans le fichier "fich" avec en regard leur identificateur.

1.4 Fonction nombre var().

* entrées : id : entier : identificateur de règle.

* préconditions : id >= 0

* résultats : nb_var : entier : nombre de variables

* postconditions :

nb_var = le nombre de variables de la règle identifiée par id.

* fonction : retourne le nombre de variables de la règle identifiée par "id".

1.5 Fonction var regle().

* entrées : id_rg : entier : identificateur de règle.

* préconditions : id >=0 .

* résultats : l : une liste de variables de type VALEURS (voir module relations).

* postconditions :

Si identifié(id_rg,*br) existe, alors :

Pour tout el appartenant à l :

el est une variable de identifié(id_rg,*br)

l->l = nombre_var(id_rg)

sinon, vide(l).

* fonction : retourne la liste de toutes les variables de la règle identifiée par "id_rg" dans la base de règles, si une telle règle existe.
Retourne une liste "l" vide sinon.

1.6 Fonction prémisse().

* entrées : br : référence à la base de règles.
r : entier : identificateur de règles.

* préconditions : ouvert(*br)
r >= 0

* résultats : l : une liste de conditions de type LS_COND.

* postconditions :

Si identifié(r,*br) existe, alors :

Pour tout el appartenant à l :

el appartient à prémisse(identifié(r,*br)).

sinon, vide(l).

* fonction : retourne la liste de toutes les conditions formant la prémisse de la règle identifiée par "r" dans la base de règles, si une telle règle existe.

Retourne une liste "l" vide sinon.

1.7 Fonction conséquent().

* entrées : br : référence à la base de règles.
r : entier : identificateur de règles.

* préconditions : ouvert(*br)
r >= 0

* résultats : l : une liste d'actions de type LS_ACT.

* postconditions :

Si identifié(r,*br) existe, alors :

Pour tout el appartenant à l :

el appartient à conséquent(identifié(r,*br)).

sinon, vide(l).

* fonction : retourne la liste de toutes les actions formant le conséquent de la règle identifiée par "r" dans la base de règles, si une telle règle existe.
Retourne une liste "l" vide sinon.

1.8 Fonction accès rg cdts().

* entrées : br : référence à la base de règles.
id_cond : entier : un identificateur de condition,
càd un identificateur de relation.

* préconditions : ouvert(*br)
id_cond >= 0

* résultats : l : une liste d'identificateurs de règles de type LS_ID_REGLE.

* postconditions : Pour tout el appartenant à l :

el=identificateur(r,*br) tel que r a une condition identifiée par id_cond.

* fonction : Retourne une liste des identificateurs de toutes les règles ayant une condition identifiée par "id_cond".

1.9 Fonction généralité().

* entrées : br : référence à la base de règles.
r1, r2 : entier : 2 identificateurs de règles.

* préconditions : ouvert(*br)
r1, r2 >= 0.

* résultats : généralité = entier

* postconditions :

```
Si identifié(r1,*br) plus général
   que identifié(r2,*br)

  alors généralité < 0;

  sinon Si identifié(r2,*br) plus général
           que identifié(r1,*br)

           alors généralité > 0
           sinon généralité = 0
```

* fonction : Détermine laquelle des règles identifiée par "r1" ou "r2" est la plus générale.
Si "r1" plus général que "r2", alors retourne un entier < 0.
Si "r2" plus général que "r1", alors retourne un entier > 0.
Sinon, retourne un entier nul.

1.10 Fonction max contrainte().

* entrées : br : référence à la base de règles.
r1, r2 : entier : 2 identificateurs de règles.

* préconditions : ouvert(*br)
r1, r2 >= 0.

* résultats : max_contrainte = entier

* postconditions :

```
Si identifié(r1,*br) plus contraignant
   que identifié(r2,*br)

  alors max_contrainte < 0;

  sinon Si identifié(r2,*br) plus contraignant
           que identifié(r1,*br)

           alors max_contrainte > 0
           sinon max_contrainte = 0
```

* fonction : Détermine laquelle des règles identifiée par "r1" ou "r2" est la plus contraignante.
Si "r1" plus contraignant que "r2", alors retourne un entier < 0.
Si "r2" plus contraignant que "r1", alors retourne un entier > 0.
Sinon, retourne un entier nul.

1.11 Fonction intro règle".

* entrées : br : référence à la base de règles.

* préconditions : ouvert(*br)

* résultats : *br'

* postconditions : *br' = *br + toutes les règles tapés au terminal et transformés sous forme interne.

* fonction : Introduction des règles au terminal.
Lecture de ces règles sous forme de chaines de caractères, transformation en représentation interne, et ajout à la base de règles.

1.12 Fonction suppr règle().

* entrées : br : référence à la base de règles.
id_règle : entier : identificateur de règles.

* préconditions : ouvert(*br)

* résultats : *br'
status : entier.

* postconditions : si identifié(id_règle,*br) existe :
*br' = *br - identifié(id_règle,*br)
status=OK.

sinon *br' = *br
status = ERREUR.

* fonction : S'il existe une règle identifiée par "id_règle" dans la base de règles, suppression de cette règle de la base et retourner "status"=OK.
Sinon, la base reste inchangée et "status"=ERREUR.

1.13 Fonction modif règle().

* entrées : br : référence à la base de règles.
id_rg : entier : un identificateur de règle.

* préconditions : ouvert(*br)

* résultats : *br'
status : entier

* postconditions : Si identifié(id_rg,*br) existe :
*br' = *br où suppr_règle(id_règle) et ajout d'une nouvelle règle introduite au terminal.
status = OK

sinon, *br' = *br
status = ERREUR.

* fonction : S'il existe une règle identifiée par "id_rg", alors modification de cette règle en la supprimant et en en saisissant une nouvelle au terminal puis en la rajoutant en base de règles. Dans ce cas, "status" = OK.
Sinon, la base de règles reste inchangée et "status"=ERREUR.

1.14 Fonction delete br().

* entrées : br : référence à la base de règles.

* préconditions : ouvert(*br)

* résultats : *br'

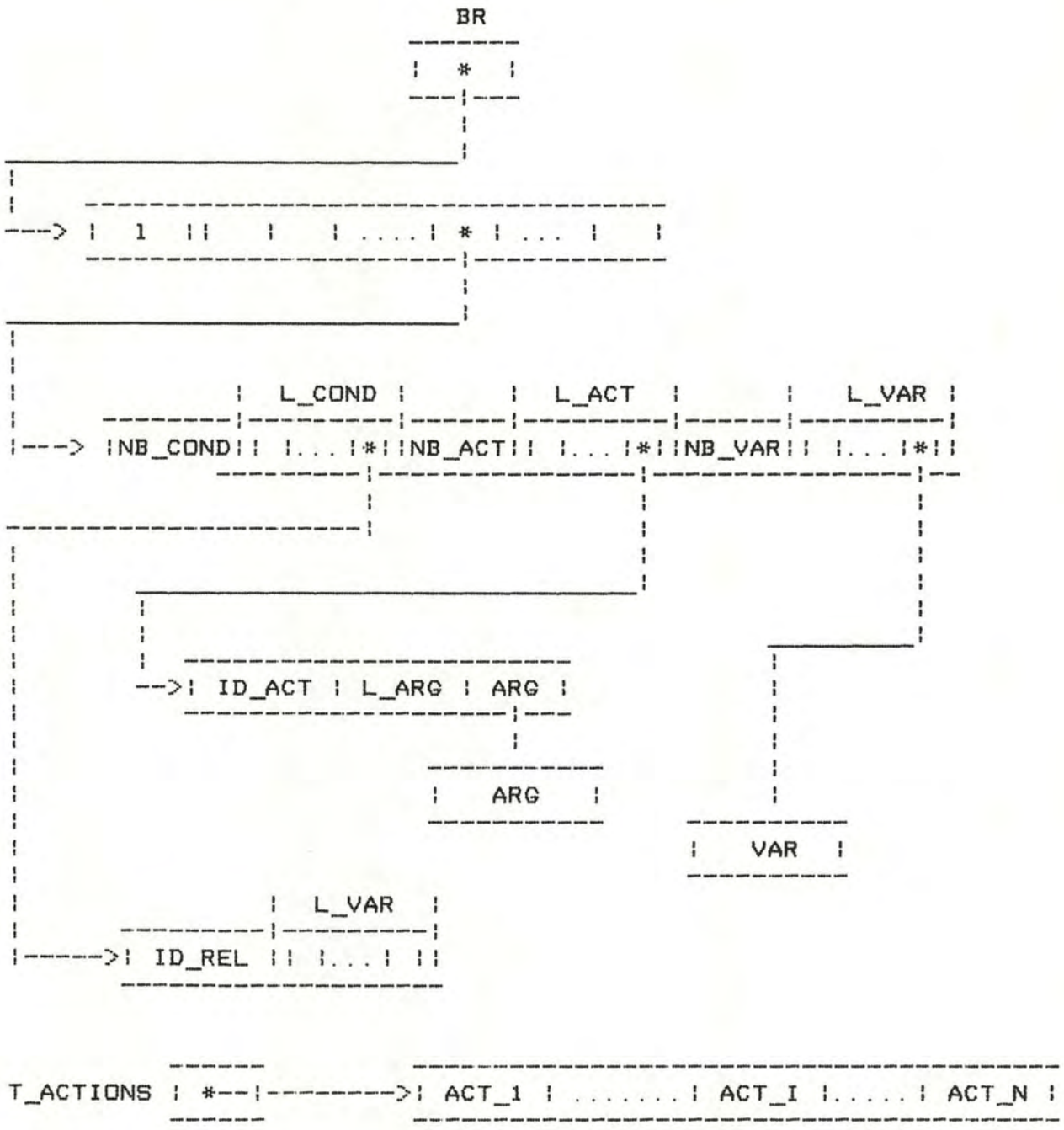
* postconditions : vide(*br)

* fonction : destruction du contenu de la base de règles.

2 Spécifications concrètes.

2.1 Représentation interne.

La représentation interne des règles est actuellement la suivante :



* une règle :

- nom : REGLE
- type : record

- contenu : - NB_COND : le nombre de conditions de la règle.
  type : entier positif

- L_COND[1..NMAXCOND] : table de références des conditions.
  type de réf : pointeur.

- NB_ACT : nombre d'actions de la règle.
  type : entier positif

- L_ACT[1..NMAXACT] : table des références des actions.
  type de réf : pointeur.

- NB_VAR : nombre de variables contenues dans la règle.

- L_VAR[1..NMAXVAR] : table des références des variables.
  type de réf : pointeur.

* la base de règles :

- nom : BASE_DE_REGLES
- type : liste de règles de type REGLE

- référence : BR

* la table des actions :

- nom : TAB_ACTION
- type : tableau de chaines de caractères.

- référence : T_ACTIONS

* structure du fichier "b_règles.file" :

- nom : B_REGLES.FILE
- type des éléments : REGLE

* structure du fichier "actions.file" :

- nom : ACTIONS.FILE
- type des éléments : chaîne de caractères.

2.2 Fonction ouvrir br().

* entrées : sch_rel, val, T_actions

* préconditions : aucune.

* résultats : br: référence à la base de règles.

* postconditions : ouvert(*br) ET ouvert(sch_rel) ET ouvert(val) ET ouvert(T_actions).

* fonction : ouverture de la base de règles et de la table des schémas de relations, de la table des valeurs de relations, et de la table des actions, si ce n'est déjà fait.
Chargement de la base de règles avec le contenu du fichier "b_règles.file" et mise à jour de br->l.

2.3 Fonction fermer br().

* entrées : br : référence à la base de règles.
T_actions : référence à la table des actions.

sch_rel : référence à la table des relations.

val : référence à la table des valeurs.

* préconditions : ouvert(*br)

* résultats : aucun.

* postconditions : aucune.

* fonction : fermeture et sauvetage de la base de règles dans le fichier "b_règles.file".
Fermeture de la table des actions, des schémas de relations et des valeurs de relations si ce n'est déjà

fait.

2.4 Fonction prémisse().

* entrées : br : référence à la base de règles.
r : entier : identificateur de règles.

* préconditions : ouvert(*br)
r >= 0

* résultats : l : une liste de conditions de type LS_COND.

* postconditions :

Si (r >= 0) ET (r < br->l) :

alors l = br->ptr_rg[r]->l_cond[0..NMAXCOND].

sinon, vide(l), i.e. l->l=0.

* fonction : retourne la liste de toutes les conditions formant la prémisse de la règle identifiée par "r" dans la base de règles, si une telle règle existe.
Retourne une liste "l" vide sinon.

2.5 Fonction conséquent().

* entrées : br : référence à la base de règles.
r : entier : identificateur de règles.

* préconditions : ouvert(*br)
r >= 0

* résultats : l : une liste d'actions de type LS_ACT.

* postconditions :

Si (r >= 0) ET (r < br->l)

alors l = br->ptr_rg[r]->l_act[0..NMAXACT].

sinon, l->l=0.

* fonction : retourne la liste de toutes les actions formant le conséquent de la règle identifiée par "r" dans la base de règles, si une telle règle existe. Retourne une liste "l" vide sinon.

2.6 Fonction accès rg cdts().

* entrées : br : référence à la base de règles.
  id_cond : entier : un identificateur de condition, càd un identificateur de relation.

* préconditions : ouvert(*br)
  id_cond >= 0

* résultats : l : une liste d'identificateurs de règles de type LS_ID_REGLE.

* postconditions :

  Pour tout el appartenant à l :

  il existe i tel que

  br->ptr_rg[el]->l_cond[i]->id_rel = id_cond.

* fonction : Retourne une liste des identificateurs de toutes les règles ayant une condition identifiée par "id_cond".

2.7 Fonction suppr règle().

* entrées : br : référence à la base de règles.
  id_règle : entier : identificateur de règles.

* préconditions : ouvert(*br)

* résultats : *br'
  status : entier.

* postconditions :

```
Si ( id_règle >= 0 ) ET ( id_règle < br->l )

        *br' = *br - br->ptr_rg[id_règle]
        status=OK.

    sinon *br' = *br
          status = ERREUR.
```

* fonction : S'il existe une règle identifiée par "id_règle" dans la base de règles, suppression de cette règle de la base et retourner "status"=OK.
Sinon, la base reste inchangée et "status"=ERREUR.

2.8 Fonction modif règle().

* entrées : br : référence à la base de règles.
id_rg : entier : un identificateur de règle.

* préconditions : ouvert(*br)

* résultats : *br'
status : entier

* postconditions : Si (id_règle >= 0) ET (id_règle < br->l)

```
                    *br' = *br - br->ptr_rg[id_règle]
                               + une règle introduite au terminal
                                 et rajoutée en fin de la base.

                    status=OK.

                sinon *br' = *br
                      status = ERREUR.
```

* fonction : S'il existe une règle identifiée par "id_rg", alors modification de cette règle en la supprimant et en en saisissant une nouvelle au terminal puis en la rajoutant en base de règles. Dans ce cas, "status" = OK.
Sinon, la base de règles reste inchangée et "status"=ERREUR.

2.9 Fonction delete br().

* entrées : br : référence à la base de règles.

* préconditions : ouvert(*br)

* résultats : *br'

* postconditions : vide(*br)

* fonction : destruction du contenu de la base de règles.

2.10 Fonction ouv act().

* entrées : aucune

* préconditions : aucune.

* résultats : T_actions : référence à la table des actions.

* postconditions : ouvert(T_actions).

* fonction : ouverture de la table des actions et chargement de son contenu à partir du contenu du fichier "actions.file".

2.11 Fonction aj act().

ATTENTION : cette fonction n'est utile que pour la mise au point du système et une éventuelle modification.

* entrées : T_actions : référence à la table des actions.
action : chaine de caractères : une nouvelle action.

* préconditions : ouvert(*T_actions) ET not (vide(action)).

* résultats : *T_actions'
id : entier.

* postconditions : *T_actions = append(*T_actions, action)

id tel que T_actions->val[id]=action.

* fonction : ajoute un nouveau type d'actions dans le système et retourne son identificateur, càd son indice dans la table des actions.

2.12 Fonction pres act().

* entrées : T_actions : référence à la table des actions.
action : chaine de caractères : une nouvelle action.

* préconditions : ouvert(*T_actions).

* résultats : pres_act : entier.

* postconditions :

s'il existe i tel que T_Actions->val[i]=action

alors pres_act=i
sinon pres_act=-1

* fonction : Recherche si l'action "action" est présente dans la table des actions référencée par T_actions.
Si oui, retourne son indice dans la table.
Sinon, retourne -1.

2.13 Fonction fermer_T_actions().

* entrées : T_actions : référence à la table des actions.

* préconditions : ouvert(*T_actions).

* résultats : aucun.

* postconditions : aucune.

* fonction : Fermeture de la table des actions et sauvetage de son contenu dans le fichier "actions.file".

2.14 Fonction aj_rg().

* entrées : br : référence à la base de règles.
r : référence à une règle de type REGLE.

* préconditions : ouvert(*br) ET r <> NULL.

* résultats : *br'

* postconditions :
*br' = *br + (br->ptr_rg[br->l]=r).

*br'->l = *br->l + 1.

* fonction : Ajoute la règle référencée par "r" à la base de règles.

2.15 Fonction tr_cdt_xi().

* entrées :
cx : COND_EXTERNE : une condition.

ri : une règle de type REGLE.

ind : ENTIER : un indice de condition.

* <u>préconditions</u> :

not(vide(cx)) ET (0 <= ind <= NMAXCOND).

* <u>résultats</u> : ri'
status : entier.

* <u>postconditions</u> :

(status=OK)
ET (ri'=ri où ri->l_cond[ind]=transformé(cx))

où transformé(cx) retourne la condition cx exprimée sous forme d'un objet de type COND.

OU

(status=ECHEC)
ET (ri'=ri)

si le nombre d'arguments de la relation correspondant à la condition n'est pas le même que le nombre d'argument que la condition.

* <u>fonction</u> : Transforme la condition externe "cx" en une condition de type COND.
Recherche la relation correspondant à cette condition;
Si elle n'existe pas encore, la rajouter dans la table des schémas de relations.
Sinon, vérifier si le nombre d'arguments de la condition est le même que le nombre d'arguments de la relation.
Si c'est le cas, "status"=OK et rajoute la condition de type COND à "ri->l_cond" à l'indice "ind".
Sinon, "status" = ECHEC et "ri" ,reste inchangé.

2.16 <u>Fonction tr cdt ix().</u>

* <u>entrées</u> :

ri : une règle de type REGLE.

id : entier : identificateur de condition.

.

* préconditions : not(vide(ri)) ET (0 <= id <= NMAXCOND).

* résultats : cx : une condition de type COND_EXTERNE.

* postconditions :
cx = transformé(ri->l_cond[id]).

* fonction : Retourne une condition de type COND_EXTERNE correspondant à la transformation de la condition de type COND "ri->l_cond[id]".

2.17 Fonction interp strc().

* entrées : arg : chaine de caractères.

* préconditions : not(vide(arg)).

* résultats : strc : booléen : indicateur de structure.
arg'

* postconditions :

```
SI arg=concat("s/",ch)

   alors arg'=ch
         strc=VRAI

   sinon arg'=arg
         strc=faux.
```

* fonction : Détermine si l'argument "arg" correspond à une structure ou pas, càd s'il est composé des symboles "s/" suivis d'une chaine de caractères "ch".
Si oui, "strc"=VRAI et la fonction retourne "ch".
Sinon, "strc"=faux et la fonction retourne "arg".

2.18 Fonction ident arg().

* entrées : arg : chaine de caractères.

* préconditions : not(vide(arg)).

* résultats : t_arg : booléen : indicateur type d'argument.

arg'

* postconditions :

si arg=concat('"',ch)

alors t_arg = CONST
arg' = ch

sinon t_arg = VARIABLE
arg' = arg.

* fonction : Détermine si l'argument "arg" correspond à une constante ou à une variable, càd s'il est composé du symbole '"' suivi d'une chaine de caractères ou pas.
Si oui, retourne t_arg = CONST et arg' = ch.
Sinon, retourne t_arg = VARIABLE et arg' = arg.

2.19 Fonction tr act xi().

* entrées : ri : règle de type REGLE

ax : une action de type ACT_EXTERNE.

ind : entier : identificateur d'une action.

T_actions : référence à la table des actions.

* préconditions :

not(vide(ri))
ET not(vide(ax))
ET (0 <= ind <= NMAXACT)
ET ouvert(T_actions)

* résultats : ri'
status : entier.

*

postconditions :

(status=OK) ET (ri'=ri où ri->l_act[ind]=transformé(ax))

où transformé(ax) retourne la transformation
de ax en un objet de type ACT,

avec id_act = pres_act(ax->nom)

arg = transformé(ax->arg) qui renvoie :

soit un objet de type ARG_INTERP
si ax->nom="interpréter"

soit un objet de type ARG_WM
si ax->nom="ajouter" ou "supprimer"

soit un objet de type ARG_IN
si ax->nom="ajout_int"

soit un objet de type ARG_ELLIPSE
si ax->nom="en_machine"

arg=NULL sinon.

l_arg=taille(arg) en nombre d'octets.

OU

(status=ECHEC) ET (ri'=ri)

* fonction : Transforme l'action "ax" en un objet de type ACT en mettant à jour son argument et le pointeur vers cet argument en fonction du type d'action.
Si le nombre ou le type d'argument n'est pas valable, retourne "status"=ECHEC et ri' = ri.
Sinon, retourne "status"=OK et ri->l_act[ind]=transformé(ax).

2.20 Fonction tr_act_ix().

* entrées : ri : une règle de type REGLE.

id : entier : identificateur d'une action de cette règle.

T_actions : référence à la table des actions.

* préconditions :

not(vide(ri))

ET (0 <= id <= NMAXACT)
ET ouvert(T_actions)

* résultats : ax : une action de type ACT_EXTERNE.

* postconditions :

ax->nom=nom(t->actions->val, ri->l_act[id]->id_act)

ax->arg = transformé(ri->l_act[id]->arg).

* fonction : Retourne l'action "ax" de type ACT_EXTERNE correspondant à la transformation de l'action d'indice "id" dans la liste des actions de la règle "ri".

2.21 Fonction aff cdt().

* entrées : cx : une condition de type COND_EXTERNE.
fich : un fichier de type texte.

* préconditions : ouvert(fich) ET not(vide(cx)).

* résultats : fich'

* postconditions : fich' = append (fich, une chaine de caractères correspodant à cx).

* fonction : Copie à la fin du fichier "fich" la condition cx sous forme de chaines de caractères, en rajoutant des parenthèses pour introduire les arguments, et en les séparant par des virgules.

2.22 Fonction aff act().

* entrées : ax : une action de type ACT_EXTERNE.
fich : un fichier texte.

* préconditions : ouvert(fich) ET not(vide(ax)).

* résultats : fich'

* postconditions : fich' = append(fich, chaine de caractères correspondant à "ax").

* fonction : Copie en fin du fichier "fich" l'action "ax" sous forme d'une seule chaine de caractères, en rajoutant des parenthèses pour isoler les arguments, et des virgules pour les séparer.

2.23 Fonction ext tp cdts().

* entrées : lc : une liste de conditions de type LS_COND.

* préconditions : not(vide(lc)).

* résultats : l : une liste d'identificateurs de type de conditions, càd de schémas de relations.

* postconditions :

Pour tout el appartenant à l :

il existe c appartenant à lc tel que
c->ide_rel=el.

vide(l) => l->l = 0.

* fonction : Retourne une liste "l" des identificateurs de conditions correspodant aux conditions de la liste de conditions "lc".

2.24 Fonction intro cond().

* entrées : r : une règle de type REGLE.

* préconditions : vide (r->l_cond[0..NMAXCOND]).

* résultats : r'

* postconditions : r' = r où r->l_cond contient toutes les conditions lues au terminal sous forme de chaines de caractères et transformées sous forme de COND.

* fonction : Introduction des conditions d'une règle.
Saisie au terminal de chaque condition sous forme de chaines de caractères.
Transformation sous forme de condition COND en vérifiant le nombre d'arguments admis et en redemandant la condition tant que ce nombre n'est pas valable.
Ajout de chaque condition ainsi transformée dans r->l_cond[0..NMAXCON].
Mise à jour de r->nb_cond.

2.25 Fonction intro act().

* entrées : r : une règle de type REGLE.

* préconditions : vide (r->l_act[0..NMAXCOND]).

* résultats : r'

* postconditions : r' = r où r->l_act contient toutes les actions lues au terminal sous forme de chaines de caractères et transformées sous forme de ACT.

* fonction : Introduction des actions d'une règle.
Saisie au terminal de chaque action sous forme de chaines de caractères.
Transformation sous forme d'action ACT en vérifiant le nombre d'arguments admis et leur type, et en redemandant l'action tant que ce nombre n'est pas valable.
Ajout de chaque action ainsi transformée dans r->l_act[0..NMAXCON].
Mise à jour de r->nb_act.

LE MODULE CONFLICT SET.

Ce module regroupe toutes les fonctions de gestion et d'utilisation du conflict_set.

1 L'interface.

Pour l'utilisateur de ce module, les objets utilisés sont du type suivant :

* une instance de règle :

  - nom : INST_REGLE
  - type : record

  - contenu : - ID_RG : un identificateur de règles.
                type : entier.

              - L_VAL_VAR[1..NMAXVAR] : liste des identificateurs des valeurs des variables. (id de valeur dans la table des valeurs.)
                type : entier.

* une liste d'instances de règles :

  - nom : L_INST_REGLES
  - type : liste

  - contenu : une liste d'éléments de type INST_REGLE.

* référence du conflict_set :

  - nom : CONSET
  _ type : pointeur vers un objet de type L_INST_REGLES.

Les fonctions suivantes sont également définies :

* init_conflict_set()
* clot_conflic_set()
* slct_conflict_set()
* resol_conflict_set()

1.1 Fonction init conflict set().

* entrées : aucune.

* préconditions : aucune.

* résultats :
  br : référence à la base de règles ;
  bf : référence à la base de faits ;
  wm : référence à la mémoire de travail ;

* postconditions : ouvert(*br)
  ouvert(*bf)
  ouvert(*wm).

* fonction : Réalisation des opérations d'initialisation avant utilisation des fonctions de gestion du conflict set : ouverture de la base de règles, de la base de faits, et de la mémoire de travail.

1.2 Fonction clot conflict set().

* entrées :
  br : référence à la base de règles ;
  bf : référence à la base de faits ;
  wm : référence à la mémoire de travail ;

* préconditions : ouvert(*br)
  ouvert(*bf)
  ouvert(*wm).

* résultats : aucun.

* postconditions : aucune.

* fonction : Réalisation des opérations de clôture après utilisation du conflict set.

1.3 Fonction slct conflict set().

* entrées :
br : référence à la base de règles ;
bf : référence à la base de faits ;
wm : référence à la mémoire de travail ;

* préconditions : ouvert(*br)
ouvert(*bf)
ouvert(*wm).

* résultats : conset : référence au conflict_set

* postconditions :

*conset = { ir t.q. ir = instance de règle,
et activable(ir->id_rg) }

activable (r) => il existe un jeu de substitution unique des variables de r par des constantes, jeu rendant prémisse(r) vraie.

* fonction : Sélection du conflict set, càd de l'ensemble des règles activables en fonction des règles disponibles, des éléments présents en mémoire, et du contenu de la base de faits.

1.4 Fonction resol conflict set().

* entrées : conset : référence au conflict set courant.

* préconditions : conset <> NULL

* résultats : ir : une instance de règle de type INST_REGLE.

* postconditions : ir est telle que :

1) activable(ir->rg)

2) pour toute ir' appartenant à *conset :

* généralité(ir, ir') > 0
OU * max_contrainte(ir, ir') < 0
OU * âge(ir) <= âge(ir')

OU ir a été choisie au hasard parmi toutes les instances de règles du conflict set ayant le même "âge" dans ce conflict set et étant aussi générales et aussi contraignantes.

* <u>fonction</u> : résolution du conflict set :

détermine, parmi toutes les instances de règles appartenant au conflict set, celle qui est activable en fonction des critères suivants, par ordre décroissant d'importance :

* préférence aux règles les moins générales ;

* préférence aux règles les plus contraignantes ;

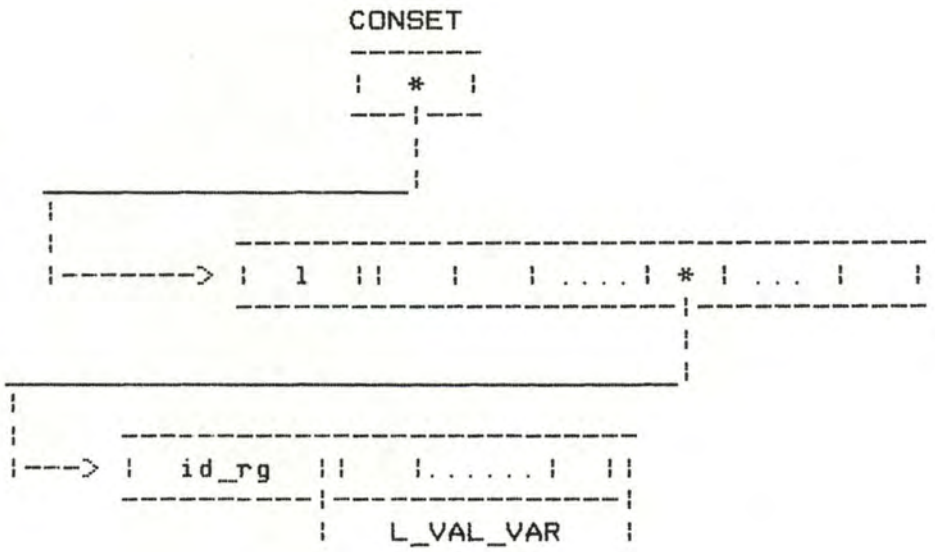
* préférence aux règles les plus jeunes dans le conflict set.

Si plusieurs instances de règles restent en compétition, choix de la première arbitrairement.

2 Spécifications concrètes.

2.1 Représentation interne.

La représentation interne est la même que celle de l'interface.



En outre, ce module utilise un type de données annexe : le type substitution.

- nom : J_SUB
- type : liste
- contenu : liste de paires (i, j) où

"i" est identificateur de variable (entier)

et "j" est identificateur de valeur (entier)

et une liste de jeu de jubstitution :

nom : L_SUB
type : liste de J_SUB.

2.2 Fonction slct conflict set().

* entrées :

br : référence à la base de règles ;
bf : référence à la base de faits ;
wm : référence à la mémoire de travail ;

conset : référence au conflict set.

* préconditions : ouvert(*br)
ouvert(*bf)
ouvert(*wm).

* résultats : *conset'

* postconditions :

*conset' = *conset

- { ir appart. à *conset telles
que not(activable(ir->id_rg)), i.e.

celles rendues non activables par
la suppression d'un élément
en mémoire de travail
au cycle précédent. }

+ { ir appart. pas à *conset telles
que activable(ir->id_rg), i.e.

celles rendues activables par
l'ajout d'éléments
en mémoire de travail
au cycle précédent. }

* fonction : Sélection du conflict set, càd de l'ensemble des règles activables en fonction des règles disponibles, des éléments présents en mémoire, et du contenu de la base de faits.
Ce conflict set est obtenu à partir du conflict set du cycle précédent en y supprimant les instances de règles qui ne sont plus activables en raison de la suppression d'un élément en mémoire de travail, et en y rajoutant celles qui sont le devenues également par l'ajout d'éléments en mémoire de travail.

2.3 Fonction identif cond().

* entrées : cond : entier : identificateur d'une condition

* préconditions : cond >= 0

* résultats : id : entier identificateur de type de condition.

* postconditions :

id = identificateur du type de la condition, càd identificateur de la relation à laquelle elle correspond.

* fonction : Retourne l'identificateur du type de la condition "cond", càd l'identificateur de la relation à laquelle elle correspond.

2.4 Fonction pres conset().

* entrées : id_rg : entier : identificateur de règle.
conset : référence au conflict set.

* préconditions : conset <> NULL.

* résultats : id : entier

* postconditions :

S'il existe i tel que conset->elem[i]->id_rg=id_rg

alors id = i

sinon id = 0.

* fonction : recherche s'il existe une instance de la règle d'identificateur "id_rg" dans le conflict set.
Si oui, retourne son indice dans ce conflict set.

Sinon, retourne 0.

2.5 Fonction maj conset().

* entrées : conset : référence au conflict set.

* préconditions : conset <> NULL

* résultats : *conset'

* postconditions :

*conset' = *conset

- { ir appart. à *conset telles
  qu'il existe une condition c
  appartenant à prémisse(ir->id_rg),
  condition portant sur el appart. à *wm
  et telle que l'action supprimer(el)
  a été réalisée au cycle
  précédent . }

* fonction : Mise à jour du conflict set , càd suppression de toutes les instances de règles étant rendues non activable par la suppression d'un élément en mémoire de travail au cycle précédent.

2.6 Fonction ad conset().

* entrées : conset : référence au conflict set.

ir : un pointeur vers une instance de règle de type INST_REGLE.

* préconditions : (conset <> NULL)ET(ir <> NULL).

* résultats : *conset'

* postconditions :

```
*conset' = *conset + (conset->elem[conset->l]=ir)

conset'->l = conset->l + 1.
```

* fonction : Ajoute l'instance de règle référencée par "ir" dans le conflict set courant référencé par "conset".

2.7 Fonction cp val var().

* entrées :

tab_1 : tableau d'entiers de taille iférieure à NMAXVAR.
s : un jeu de substitution de type J_SUB.

* préconditions : vide(tab_1).

* résultats : tab_1'

* postconditions :

```
tab_1[i] = j ssi il existe (i, j) dans s.
         = ? sinon.
```

* fonction : Transforme le jeu de substitution "s" en un tableau d'entiers "tab_1" en gardant la liaison entre les variables et leur valeur par le biais des indices. Ainsi, le jeu de substitution (i, j) attribuant la valeur identifiée par "j" à la variable identifiée par "i" sera représenté dans la table "tab_1" de la façon suivante : tab_1[i]=j.
Retourne ce tableau "tab_1" modifié.

2.8 Fonction nom val var().

* entrées :

l_v_v : tableau d'entiers : liste d'identificateurs de valeurs de variables.
id : entier : indice d'une valeur dans cette table.

* préconditions : 0 <= id < longueur(l_v_v).

* résultats : nom : chaine de caractères.

* postconditions : nom = nomval(l_v_v[id]) (voir module relations.).

* fonction : Etant donné une table d'identificateurs de valeurs de variables et un indice dans cette table, retourne le nom de la valeur sous forme de chaine de caractères.

2.9 Fonction sup conset().

* entrées : conset : référence au conflict set.

id_ir : entier : identificateur d'instance de règle.

* préconditions : (conset <>NULL) ET (id_rg > 0)

* résultats : *conset'

* postconditions :

```
Si (0 <= id_ir <= conset->l)
        *conset' = *conset - conset->elem[id_ir]
        conset'->l = conset->l - 1.
  sinon *conset'=conset.
```

* fonction : Si "id_ir" identifie une instance de règle dans le conflict set, supprime cette instance du conflict set.
Sinon, cette fonction n'a aucun effet.

2.10 Fonction vrai().

* entrées :

wm : référence à la mémoire de travail ;
bf : référence à la base de faits ;
cdt : une condition de type COND (cfr base règles).

* préconditions : not(vide(cdt))
ouvert(*wm)
ouvert(*bf)

* résultats : l : une liste de substitution de type L_SUB.

* postconditions :

vide(l) => il n'existe pas de jeu de substitution rendant la condition "cdt" vraie ; (l=NULL)

Sinon, pour tout el appartenant à l, el est jeu de substitution des variables de la condition par des constantes tel la condition "cdt" est vraie.

* fonction : Evalue la condition "cdt".
Si cette condition est vraie, retourne une liste des jeux de substitution qui la rendent vraie.
Sinon, retourne une liste vide.

2.11 Fonction fusion().

* entrées :

sb_1, sb_2 : 2 jeux de substitution de type J_SUB.

* préconditions : sb_1, sb_2 <> NULL.

* résultats : sb : un jeu de substitution de type J_SUB.

* postconditions :

Pour tout couple (var,val) appartenant à sb_1, (var,val) appartient à sb.

Pour tout couple (var,val) appartenant à sb_2, (var,val) appartient à sb.

Pour tout couple (var,val) appartenant à sb , une des 3 propositions suivantes doit être vérifiée :

soit (var,val) appartient à sb_1 et sb_2 ;

soit (var,val) appartient à sb_1 et pas à sb_2 ; alors, il n'existe pas de couple (var, val') appartenant à sb_2.

soit (var,val) appartient à sb_2 et pas à sb_1 ;
alors, il n'existe pas de couple (var, val')
appartenant à sb_1.

vide(sb) => sb_1 et sb_2 sont incompatibles.

* fonction : Fusion des 2 jeux de substitution "sb_1" "sb_2" en un troisième "sb".

2.12 Fonction inter().

* entrées :
ls_1, ls_2 : 2 listes de jeux de substitution de type L_SUB.

* préconditions : aucune.

* résultats : ls : une liste de jeu de substitution de type L_SUB.

* postconditions :

Si vide(ls_1) alors ls = ls_2;
Si vide(ls_2) alors ls = ls_1;

Si not(vide(ls_1)) et not(vide(ls_2))
alors pour tout el appartenant à ls :
il existe el_1 appartenant à ls_1 et
il existe el_1 appartenant à ls_1 tels que

el = fusion(el_1,el_2).

* fonction : Fusion des 2 listes de jeux de substitution "ls_1" et "ls_2" en une troisième "ls" contenant des jeux de substitution correspondant à la fusion de 2 jeux appartenant respectivement à "ls_1" et "ls_2".

2.13 Fonction évalue().

* entrées :

wm : référence à la mémoire de travail ;
bf : référence à la base de faits ;

l_c : une liste de conditions de type LS_COND;

(voir base de règles).

* préconditions : l_c->l <= 1.

* résultats : ls : une liste de jeux de substitution de type L_SUB.

* postconditions :

Si l_c->l = 1

alors évalue = vrai(l_c->elem[0])

sinon
évalue=inter(vrai(l_c->elem[0]),
évalue(l_c->elem[1..l_c->l]

* fonction : Evalue la liste de conditions "l_c".
Retourne la liste des jeux de substitution qui la rendent vraie.
Retourne une liste vide si elle n'est pas vérifiée.

2.14 Fonction activable().

* entrées :
rg : entier : identificateur de règle.

wm : référence à la mémoire de travail ;
bf : référence à la base de faits ;

* préconditions :

il existe r appartenant à la base de règles tel que identificateur(r,*br) = rg.

ouvert(*wm)
ouvert(*bf)

* résultats : lir : une liste d'instances de règle de type L_INST_REGLES.

* postconditions :

si not(vide(évalue(prémisse(rg))))

alors pour tout i (0 <= i <= lir->l)

lir->elem[i] = une instance de règle composée avec un des jeux de substitution.

sinon vide(lir) (i.e. lir = NULL).

* fonction : Vérifie si une règle d'identificateur "rg" est activable ou pas.
Si oui, retourne une liste d'instances de cette règle composées avec les jeux de substitution qui rendent sa prémisse vraie.
Retourne une liste vide sinon.

2.15 Fonction pres_lr().

* entrées : id : un identificateur de règle.
lr : une liste d'identificateurs de règles de type L_ID_REGLES.

* préconditions : aucune.

* résultats : res : entier.

* postconditions :

S'il existe i tel que lr->elem[i] = id

alors res = i
sinon res = -1

* fonction : Recherche si l'identificateur de règle "id" est présent dans la liste des identificateurs de règles "lr".

Si oui, retourne son indice dans cette liste.
Sinon, retourne -1.

2.16 Fonction filtre().

* entrées : br : référence à la base de règles ;
wm : référence à la mémoire de travail ;

* préconditions : ouvert(*br)
ouvert(*wm).

* résultats : lr : une liste d'identificateurs de règles de type L_ID_REGLES.

* postconditions :

pour tout id_rg appartenant à lr :

il existe el appartenant à *wm
c appartenant à prémisse(id_rg)

tel que c porte sur el et el appartient à ac_nouv_el_wm().
(voir module work_mem).

* fonction : retourne une liste de tous les identificateurs de règles de la base de règles ayant une condition qui porte sur un élément qui vient d'être rajouté en mémoire de travail au cycle précédent.

2.17 Fonction compare inst().

* entrées : ir_1, ir_2 : 2 intances de règles de type INST_REGLE.

* préconditions : not(vide(ir_1))
not(vide(ir_2))

* résultats : res : entier.

* postconditions :

* si ir_1 = ir_2

```
            alors res = 0 ;

        * si    (ir_1->id_rg = ir_2->id_rg)
             ET (ir_1->l_val_var <> ir_2->l_val_var)

            alors res = 1 ;

        * si    (ir_1->id_rg <> ir_2->id_rg)
             ET (ir_1->l_val_var <> ir_2->l_val_var)

           alors res = 2.
```

* fonction : Compare les 2 instances de règles "ir_1" et "ir_2".
Si elles sont identiques, retourne la valeur 0 ;
si elles sont deux instances différentes de la même règle, retourne la valeur 1 ;
si elles n'ont rien en commun, retourne la valeur 2.

2.18 Fonction aff_1_inst_rg().

* entrées : fich : fichier de type texte ;
lir : liste d'instances de règles de type L_INST_REGLES ;
br : référence à la base de règles.

* préconditions : ouvert(*br)
not(vide(lir))
ouvert(fich)
fich = standard output ou fichier logique imprimante.

* résultats : fich'

* postconditions :

fich' = append(fich, contenu de "lir" transformé sous forme externe).

* fonction : Copie de la liste d'instances de règles "lir" en fin du fichier "fich" et sous forme externe pour affichage à l'écran (fich = standard output) ou impression (fich=fichier logique de l'imprimante).

2.19 Fonction strat généralité().

* entrées : lir : une liste d'insatances de règles de type L_INST_REGLES.

* préconditions : lir->l > 1.

* résultats : lir'

* postconditions :

lir' = lir - { x appartenant à lir tels que
il existe y appartenant à lir et

généralité(x->id_rg, y->id_rg) < 0 }.

* fonction : Etant donné une liste d'instances de règles "lir", supprime de cette liste toutes les instances qui correspondent à des règles plus générales que celles auxquelles d'autres instances de cette liste correspondent.
Retourne la liste ainsi modifiée.

2.20 Fonction strat contrainte().

* entrées : lir : une liste d'instances de règles de type L_INST_REGLES.

* préconditions : lir->l > 1.

* résultats : lir'

* postconditions :

lir' = lir - { x appartenant à lir tels que
il existe y appartenant à lir et

max_contrainte(x->id_rg, y->id_rg) > 0 }.

* fonction : Etant donné une liste d'instances de règles "lir", supprime de cette liste toutes les instances qui correspondent à des règles moins contraignantes que

celles auxquelles d'autres instances de cette liste correspondent.
Retourne la liste ainsi modifiée.

LE MODULE GEST TRACE.

Ce module regroupe toutes les fonctions de gestion de la trace d'exécution.

1 L'interface.

L'interface avec les modules de niveau supérieurs est constitué d'une référence à une structure correspondant à la trace d'exécution, d'une référence à un fichier, et d'un certain nombre de fonctions.

En fait, 2 traces d'exécution sont gérées : l'une, interne au système, lui permet de faire un certain nombre de vérifications, comme par exemple pour éviter un cyclage de l'interpréteur ; la seconde est dédiée exclusivement à l'utilisateur et est de nature quelque peu plus explicite.
La première est représentée sous forme d'une structure de données, tandis que la seconde est destinée dans un fichier en fonction de l'option prise au moment démarrage du programme.

* Référence à la trace d'exécution :

  - nom : TRACE
  - type : pointeur vers une structure qui correspond à la trace, et de type TRC_TRACE.

L'utilisation de ce module ne nécessite pas la définition de la structure.

* Référence au fichier trace-utilisateur :

  - nom : trc_pf
  - type : pointeur vers un fichier de type texte.

Les fonctions suivantes sont définies :

* trc_creation()
* trc_ant_maj()
* trc_d_interp()
* trc_f_interp()
* trc_aj_wm()
* trc_aj_int()
* trc_référé()
* trc_diag_maj()
* trc_présence()

1.1 Fonction trc creation().

* entrées : option : booléen : spécifie si la trace doit être affichée à l'écran ou bien copiée dans un fichier.

* préconditions : (option=ON_LINE) OU (option=OFF_LINE)

* résultats : trace : référence trace d'exécution.
trc_pf : référence au fichier texte qui recevra la trace d'exécution.

* postconditions :

ouvert(*trace)
ouvert(*trc_pf)

option=ON_LINE => *trc_pf = standard output ;
option=OFFLINE => *trc_pf = "trace.out" ;

* fonction : Création et ouverture de la trace d'exécution.
Mise à jour d'une référence vers un fichier destinataire en fonction de "option".

1.2 Fonction trc ant maj().

* entrées : trace : référence trace d'exécution ;
trc_pf : référence fichier trace-utilisateur ;
cycle : entier : cycle interpréteur ;

ir : une instance de règle de type INST_REGLE.

* préconditions : ouvert(*trace)
ouvert(*trc_pf)

* résultats : *trace'
*trc_pf'

* postconditions :
*trace' = *trace + ir

*trc_pf' = *trc_pf modifié par

aff_l_inst_rg(trc_pf, ir)

(voir conflict set pour cette fonction)

* fonction : Mise à jour de la trace d'exécution référencée par "trace" et par "trc_pf" avant l'activation de l'instance de règle "ir".

1.3 Fonction trc d interp().

* entrées : arg : argument de l'action de type ARG_INTERP

trc_pf : référence fichier trace-utilisateur ;

* préconditions : ouvert(*trc_pf)

* résultats : *trc_pf'

* postconditions :

*trc_pf' = *trc_pf modifié par l'ajout d'informations concernant l'appel récursif, et ce avant son déclenchement :

* identificateur de variable concernée,
* type d'argument (structure ou terminal)

* fonction : Mise à jour de la trace d'exécution référencée par "trc_pf" avant l'activation de l'action d'ordre d'interprétation de la règle courante.

1.4 Fonction trc f interp().

* entrées : status : entier : diagnostic d'exécution.
trc_pf : référence fichier trace-utilisateur ;
cycle : entier : cycle interpréteur ;

* préconditions : ouvert(*trc_pf)

* résultats : *trc_pf'

* postconditions :

*trc_pf' = *trc_pf modifié par l'ajout d'informations concernant la fin d'exécution de l'action d'appel récursif et sur ses résultats, càd le status.

* fonction : Mise à jour de la trace d'exécution référencée par "trc_pf" après l'exécution d'un ordre d'inteprétation en lui ajoutant le diagnostic d'exécution.

1.5 Fonction trc_aj_wm().

* entrées :

trc_pf : référence fichier trace-utilisateur ;

e : un élément de type EL_WM ;

* préconditions : ouvert(*trc_pf)

* résultats : *trc_pf'

* postconditions :

*trc_pf' = *trc_pf modifié par l'ajout d'informations concernant une action d'ajout d'un élément en mémoire de travail:

* l'élément rajouté
* le diagnostic d'exécution (OK)

* fonction : Mise à jour de la trace d'exécution référencée par "trc_pf" après l'exécution d'une action d'ajout d'un élément en mémoire de travail : rajout des informations cncernant l'élément rajouté et le diagnostic d'exécution.

1.6 Fonction trc_aj_int().

* entrées :

trc_pf : référence fichier trace-utilisateur ;
arg : argument de type chaine de caractères.

* préconditions : ouvert(*trc_pf)

* résultats : *trc_pf'

* postconditions :

*trc_pf' = *trc_pf modifié par l'ajout d'informations concernant l'action d'ajout d'une partie d'interprétation dans la structure de données :

* argument de l'action "arg"
* diagnostic d'exécution (OK).

* fonction : Mise à jour de la trace d'exécution référencée par "trc_pf" après l'exécution d'une action d'ajout d'une partie d'interprétation "arg".

1.7 Fonction trc_référés().

* entrées :
trc_pf : référence fichier trace-utilisateur ;
status : entier : diagnostic d'exécution.

* préconditions : ouvert(*trc_pf)

* résultats : *trc_pf'

* postconditions :

*trc_pf' = *trc_pf modifié par l'ajout d'informations concernant l'action d'accès à l'historique pour la résolution des références, à savoir le diagnostic d'exécution.

* fonction : Mise à jour de la trace d'exécution référencée par "trc_pf" après l'exécution d'une action d'accès à l'historique pour la résolution des références.

1.8 Fonction trc diag ma,().

* entrées : status : entier : diagnostic exécution
trc_pf : référence fichier trace-utilisateur ;

* préconditions : ouvert(*trc_pf)

* résultats : *trc_pf'

* postconditions :

*trc_pf' = *trc_pf modifié par l'ajout du diagnostic d'exécution global status de la règle qui vient d'être activée.

* fonction : Mise à jour de la trace d'exécution référencée par "trc_pf" après l'activation de l'instance de règle courante : ajout de son diagnostic d'exécution global "status".

1.9 Fonction trc présence().

* entrées :
trace : référence trace d'exécution.
ir : instance de règle de type INST_REGLE.

* préconditions : ouvert(*trace)

* résultats : trc_présence : entier.

* postconditions :

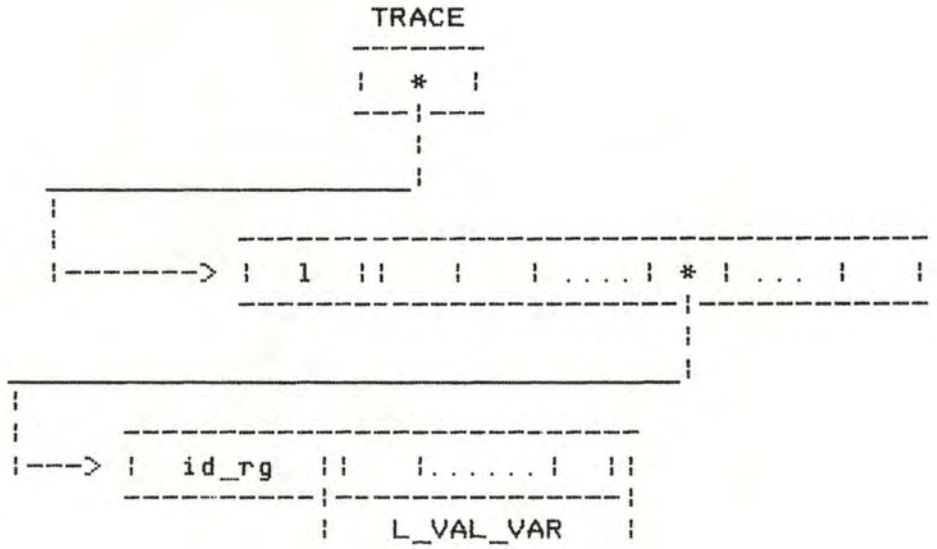
trc_présence = 1 si ir appartient à *trace ;
= 0 sinon.

* fonction : Recherche si l'instance de règle "ir" se trouve dans la trace d'exécution référencée par "trace".
Si oui, retourne l'entier 1.
Sinon, retourne 0.

2 Spécifications concrètes.

2.1 Représentation interne.

La trace d'exécution est organisée de la manière suivante :



La trace est organisée comme un objet de type L_INST_REGLES. Pour le détail de ce type de structure, voir module "conflict_set".

2.2 Fonction trc creation().

* entrées : option : booléen : spécifie si la trace doit être affichée à l'écran ou bien copiée dans un fichier.

* préconditions : (option=ON_LINE) OU (option=OFF_LINE)

* résultats : trace : référence trace d'exécution.
  trc_pf : référence au fichier texte qui recevra la trace d'exécution.

* postconditions :

trace <> NULL et trace->l = 0.

ouvert(*trc_pf)

option=ON_LINE => *trc_pf = standard output ;
option=OFFLINE => *trc_pf = "trace.out" ;

* fonction : Création et ouverture de la trace d'exécution.
Mise à jour d'une référence vers un fichier destinataire en fonction de "option".

2.3 Fonction trc_ant_maj().

* entrées : trace : référence trace d'exécution ;
trc_pf : référence fichier trace-utilisateur ;
cycle : entier : cycle interpréteur ;

ir : une instance de règle de type INST_REGLE.

* préconditions : ouvert(*trace)
ouvert(*trc_pf)

* résultats : *trace'
*trc_pf'

* postconditions :

*trace' = *trace + (trace->elem[trace->l]=ir) ;

trace'->l = trace->l ;

*trc_pf' = *trc_pf modifié par
aff_l_inst_rg(trc_pf, ir)

* fonction : Mise à jour de la trace d'exécution référencée par "trace" et par "trc_pf" avant l'activation de l'instance de règle "ir".

LE MODULE ACTIVATION.

Le module "activation" regroupe toutes les fonctions chargées de l'activation des règles, c'est-à-dire de l'exécution de leurs actions.

1 L'interface.

Le seul interface avec les modules de niveaux supérieurs est matérialisé par la structure de données correspondant à la partie d'interprétation déjà réalisée et par la fonction "active()".

* référence à la partie d'interprétation déjà réalisée :

  - nom : RESULT_INT
  - type : pointeur vers un tableau de caractères.

* référence courante dans cette structure :

  - nom : CUR_INT
  - type : pointeur vers un endroit dans le tableau référencé par RESULT_INT

1.1 Fonction active().

* entrées : ir : une instance de règle de type INST_REGLE

  wm : référence à la mémoire de travail ;

  br : référence à la base de règles ;

  br : référence à la base de faits ;

  result_int : référence à la structure de données correspondant à l'interprétation partielle déjà réalisée.

  cur_int : pointeur courant vers cette structure.

* préconditions : not(vide(ir))
  ouvert(*wm)

```
                    ouvert(*br)
                    ouvert(*bf)
                    cur_int et result_int <> NULL.

*   résultats : status
                *wm'
                *result_int'
                cur_int'

*   postconditions :

    * S'il existe act appartenant à conséquent(ir->id_rg)
      tel que nom(act)="interpréter" :

      *result_int' = append(*result_int, résultat d'interprétation
                      de l'appel récursif à l'interpréteur) ;

      cur_int' = cur_int + taille de ce résultat en nombre de
                 caractères;

      *wm' = *wm ;

    * S'il existe act appartenant à conséquent(ir->id_rg)
      tel que nom(act)="ajouter" :

      *result_int' = *result_int ;

      cur_int' = cur_int ;

      *wm' = *wm modifiée par aj_wm(e)

            où e = transformation de l'argument
                     arg en un objet de type EL_WM ;

    * S'il existe act appartenant à conséquent(ir->id_rg)
      tel que nom(act)="supprimer" :

      *result_int' = *result_int ;

      cur_int' = cur_int ;

      *wm' = *wm moins e

            où e = transformation de l'argument
                     arg en un objet de type EL_WM ;

      *wm'->l = *wm->l - 1 ;

    * S'il existe act appartenant à conséquent(ir->id_rg)
      tel que nom(act)="ajout_int" :
```

```
  *result_int' = append(*result_int, x)

              où x = nom_val_var(lvv, arg->v)
                              si arg->t_arg = VARIABLE
                 et x = arg->const
                              si arg->t_arg = CONST ;

  cur_int' = cur_int + taille (x)

  *wm' = *wm ;


* S'il existe act appartenant à conséquent(ir->id_rg)
  tel que nom(act)="hist_référé" :

  *result_int' = *result_int ;

  cur_int' = cur_int ;

  *wm' = *wm modifié par aj_wm(e)
              pour tout e, objet de type EL_WM
              construit à partir d'un élément
              de la liste des valeurs v_1,...,v_n
              recueillies dans l'historique et
              susceptibles d'être le référé.

      A ces éléments "e" correspond la
      forme externe :

          référé(v_i)


* S'il existe act appartenant à conséquent(ir->id_rg)
  tel que nom(act)="en_machine" :

  *result_int' = append(*result_int, x)

         où x est le dernier énoncé de la machine
               repéré dans l'historique et où
               le point d'interrogation est remplacé par
               l'interprétation de act->arg (si
               act->arg->t_arg = VARIABLE) ou par act->arg
               lui-même ( si act->arg->t_arg = CONST)

  cur_int' = cur_int + taille(x)

  *wm' = *wm


Dans tous les cas :

  status = ERREUR si une des actions n'a pu être
                  exécutée, les autres actions du
                  conséquent de la règle sont alors
                  ignorées.
```

status = ECHEC si l'interpréteur n'a pas pu aller au bout d'un appel récursif, les autres actions étant également ignorées ;

status = OK si toutes les actions de la règle ont été exécutées avec succès ;

status = FIN idem, mais une des actions exécutée contenait un ordre implicite de fin d'exécution.

* <u>fonction</u> : Activation proprement dite d'une règle, c'est-à-dire exécution séquentielle des différentes actions contenues dans son conséquent.
Retourne un diagnostic d'exécution (voir postconditions).

2 Spécifications concrètes.

2.1 Représentation interne.

La représentation de la structure de données correspondant au résultat de l'interprétation est identiquement la même que celle définie en interface ci-dessus.

Pour la gestion des appels récursifs à l'interpréteur, une pile est définie pour assurer le sauvetage et la restitution de la mémoire de travail.

```
nom : STACK
type : record

contenu : - PTR : pointeur de pile
                  type : entier.

          - ELEM [1..5] : liste d'éléments de
                          la pile.
                  type de ces éléments : L_EL_WM
```

2.2 Fonction push().

* entrées : pile : une pile de type STACK
list : une liste d'éléments de mémoire de travail de type L_EL_WM.

* préconditions : aucune.

* résultats : pile'
status : entier.

* postconditions :

```
si pile->ptr <= 4

   alors pile'->ptr = pile->ptr + 1
         pile'->elem[pile->ptr] = list
         status = OK ;
```

sinon pile'=pile
status = ECHEC.

* fonction : Charge la liste d'instance de règles "list" sur le sommet de la pile "pile" si c'est possible.
Si oui, retourne "status" = OK.
Sinon, retourne "status"=ECHEC.

2.3 Fonction pop().

* entrées : pile : une pile de type STACK.

* préconditions : pile->ptr >= 1.

* résultats : pile'
list : une liste d'éléments de mémoire de travail de type L_EL_WM.

* postconditions :

```
list = pile->elem[pile->ptr] ;

pile'->ptr = pile->ptr - 1 ;

pile'->elem[1..5] = pile->elem[1..5].
```

* fonction : Retourne la liste d'éléments de mémoire de travail "list" qui se trouve au sommet de la pile et met à jour ce sommet.

2.4 Fonction ordre interp().

* entrées :

arg : l'argument de type ARG_INTERP

lvv : tableau des identificateurs des valeurs des variables (L_ID_VAL).

result_int : référence à la structure de données correspondant à l'interprétation partielle déjà réalisée.

cur_int : pointeur courant dans cette structure.

cycle : entier : cycle de l'interpréteur.

wm : référence à la mémoire de travail.

conset : référence au conflict set.

pile : pile de type STACK.

* préconditions : not(vide(arg))
result_int et cur_int <> NULL.
ouvert(*wm)

* résultats : cur_int'
*result_int'
*wm'
conset'
cycle'
status : entier.

* postconditions :

*wm' = *wm
cycle' = cycle
conset' = NULL

* status = FIN

=> *result_int' = append(result_int, chaine de caracte`res
correspondant à l'interprétation de
l'argument)

cur_int' = cur_int + le nombre de caractères
de l'interprétation de l'argument.

* status = ERREUR si l'interpréteur est tombé
sur une erreur lors de l'exécution
d'une action d'une règle activable ;
=> *result_int' = *result_int
cur_int' = cur_int

* status = ECHEC si le sauvetage de la mémoire de
travail n'a pu se faire ou si l'intepréteur
a échoué dans l'établissement
d'une interprétation complète
de l'argument.

=> *result_int' = *result_int
cur_int' = cur_int

* fonction : Réalisation d'un appel récursif sur une sous-structure de l'énoncé en entrée :
sauvetage du contenu de la mémoire de travail sur la pile "pile";
mise à jour de la *wm en fonction de l'argument (voir maj_wm() du module de gestion de la mémoire de travail);
destruction du conflict set ;
sauvetage du cycle de l'interpréteur ;
lancement de l'interpréteur ;
quand il a terminé, restitution du contenu de la mémoire de travail et du cycle interpréteur, et destruction du conflict set courant.

Retourne "status" = ERREUR, ECHEC ou FIN.

2.5 Fonction aj_el_wm().

* entrées : arg : l'argument de type ARG_WM ;
lvv : liste des identificateurs des valeurs des variables (L_ID_VAL).

wm : référence à la mémoire de travail ;

âge : entier : âge de l'élément à ajouter.

* préconditions : not(vide(arg)).
ouvert(*wm)

* résultats : status
*wm'

* postconditions :

*wm' = *wm avec aj_wm(e)

où e = transformation de l'argument arg en un objet de type EL_WM.

status = OK.

* fonction : Exécution d'un ajout d'un élément en mémoire de travail "arg" après l'avoir transformé en objet de type EL_WM.
Retourne "status" = OK .

2.6 Fonction sup el wm().

* entrées : arg : l'argument de type ARG_WM ;
lvv : liste des identificateurs des valeurs des variables (L_ID_VAL).

wm : référence à la mémoire de travail ;

* préconditions : not(vide(arg)).
ouvert(*wm)

* résultats : status
*wm'

* postconditions :

*wm' = *wm - e

où e = transformation de l'argument arg en un objet de type EL_WM.

*wm'->l = *wm->l - 1

status = OK.

* fonction : Exécution d'une suppression d'un élément de mémoire de travail "arg" après l'avoir transformé en objet de type EL_WM.
Retourne "status" = OK.

2.7 Fonction ajout int().

* entrées :

arg : l'argument de type ARG_IN

lvv : tableau des identificateurs des valeurs des variables (L_ID_VAL).

result_int : référence à la structure de données correspondant à l'interprétation partielle déjà réalisée.

cur_int : pointeur courant dans cette structure.

* préconditions : not (vide(arg))
result_int et cur_int <> NULL.

* résultats : cur_int'
*result_int'
status : entier.

* postconditions :

Si arg->t_arg = VARIABLE

```
*result_int' = append(*result_int, nom_val_var(lvv, arg->v));
cur_int' = cur_int + taille(nom_val_var(lvv, arg->v));
status = OK;
```

Si arg->t_arg = CONST

```
*result_int' = append(*result_int, arg->const)
cur_int' = cur_int + taille(arg->const));
status = OK.
```

* fonction : Exécution d'une action d'ajout d'une partie d'interprétation.
Si arg->t_arg=VARIABLE, recherche de la valeur de la variable "arg->v" dans "lvv" et ajout de cette valeur sous forme de chaine de caractères dans la structure de données référencée par "result_int", et mise à jour de "cur_int".
Si arg->t_arg=CONST, ajout de "arg->const" dans cette structure de données, et mise à jour également de "cur_int".

2.8 Fonction h_référés().

* entrées : wm : référence à la mémoire de travail ;

* préconditions : ouvert (*wm).

* résultats : *wm'

* postconditions :

*wm' = *wm modifié par aj_wm(e)

pour tout e, objet de type EL_WM
construit à partir d'un élément

de la liste des valeurs v_1,...,v_n
recueillies dans l'historique et
susceptibles d'être le référé.

A ces éléments "e" correspond la
forme externe :

référé(v_i)

* fonction : Etant donné une liste d'instances de règles "lir", supprime de cette liste toutes les instances qui correspondent à des règles moins contraignantes que celles auxquelles d'autres instances de cette liste correspondent.

2.9 Fonction en machine().

* entrées :

arg : l'argument de type ARG_ELLIPSE

lvv : tableau des identificateurs des valeurs des variables (L_ID_VAL).

result_int : référence à la structure de données correspondant à l'interprétation partielle déjà réalisée.

cur_int : pointeur courant dans cette s tructure.

cycle : entier : cycle de l'interpréteur.

wm : référence à la mémoire de travail.

conset : référence au conflict set.

pile : pile de type STACK.

* préconditions : not (vide(arg))
result_int et cur_int <> NULL.

* résultats : cur_int'
*result_int'
*wm'
conset'
cycle'
status : entier.

* **postconditions** :

Si arg->t_arg = VARIABLE

*wm' = *wm

cycle' = cycle

conset' = NULL

*result_int' = append(*result_int, x)

où x est le dernier énoncé de la machine
repéré dans l'historique et où
le point d'interrogation est remplacé par
l'interprétation de la valeur associée
à la variable arg->v, interprétation
obtenue par appel récursif à l'interpréteur
avec arg->v transformé en objet de type ARG_INTERP
et lvv comme paramètres.

cur_int' = cur_int + taille(x)

status = ERREUR si l'interpréteur est tombé
sur une erreur.

status = ECHEC si le sauvetage de la mémoire de
travail n'a pu se faire;

status = FIN sinon.

Si arg->t_arg = CONST

*wm' = *wm

cycle' = cycle

conset' = NULL

*result_int' = append(*result_int, x)
où x est le dernier énoncé de la machine
repéré dans l'historique et où
le point d'interrogation est remplacé par
la valeur de arg->const.

cur_int' = cur_int + taille(x)

status = FIN.

* fonction : Exécution de l'action d'accès à l'historique pour la résolution des références ; remplacement du point d'interrogation dans le dernier énoncé machine par l'interprétation ou la valeur de l'argument "arg" passé en paramètre, le tout étant ensuite chargé en fin de la structure de données référencée par "result_int".

LE MODULE INTERPRETATION.

Le module "interprétation" constitue le module principal du système.
Il contient les fonctions qui utilisent les modules de niveau inférieurs pour implémenter l'interpréteur de règles de production.

1 L'interface.

L'interface est constitué de la seule fonction "contrôle().

1.1 Fonction contrôle().

* entrées :

  option : booléen : identificateur de l'option prise pour la trace-utilisateur (on- ou off-line).

  fich : le fichier contenant les représentations syntaxique et sémantique de l'énoncé.

* préconditions : not(vide(fich)).

* résultats :

  res_interp : le fichier de résultat de l'interprétation de l'énoncé réalisée par l'interpréteur de règles.

  Il s'agit d'un fichier de type texte.

* postconditions : "res_interp" contient le résultat de l'inteprétation de l'énoncé par l'intepréteur de règles.

* fonction : Contrôle de l'interprétation d'un énoncé :
  initialisation du système
  traduction de l'énoncé sous forme de relations à partir de ses représentations contenues dans "fich" ;
  sauvetage de la mémoire de travail ;
  génération des hypothèses à partir de l'historique ;
  lancement de l'intepréteur de règles ;

En cas d'échec, restitution de la mémoire de travail (sans les hypohtèses) et re-lancement de l'intepréteur de règles.
affichage du diagnostic d'exécution et copie du résultat dans le fichier "res_interp".

2 Spécifications concrètes.

2.1 Fonction anti cyclage().

* entrées :

  trace : référence trace d'exécution ;

  lir : une liste d'instances de règle de type L_INST_REGLES.

* préconditions : ouvert(*trace)
  not(vide(lir)).

* résultats : lir'

* postconditions :

  lir' = lir - {ir t.q. trc_présence(ir) = 1 }.

* fonction : Extrait de la liste d'instances de règles "lir" toutes les instances de règles qui sont présentes dans la trace d'exécution.
  Retourne la liste ainsi modifiée.

2.2 Fonction initialisation().

* entrées : aucune

* préconditions : aucune

* résultats :

  br : référence à la base de règles

  bf : référence à la base de faits

  wm : référence à la mémoire de travail

  result_int : réf. à la structure de données correspondant au résultat partiel

de l'interprétation.

cur_int : pointeur courant dans cette structure.

* postconditions :

ouvert(*br)
ouvert(*bf)
ouvert(*wm)
result_int <> NULL et vide(result_int)
cur_int <> NULL

* fonction : Réalisation des opérations d'initialisation.

2.3 Fonction trad énoncé().

* entrées :

fich : un fichier contenant les représentations de l'énoncé (syntaxique et sémantique)

type ????

wm : référence à la mémoire de travail

* préconditions : ouvert(*wm)

* résultats : *wm'

* postconditions :

*wm' : *wm modifié par aj_wm(e) pour tout e construit à partir des représentations de l'énoncé contenues dans "fich" et ayant pour objectif de le représenter en mémoire de travail.

* fonction : Traduction des représentations syntaxique et sémantique de l'énoncé courant contenues dans "fich" sous forme d'instances de relations et ajout de ces instances en mémoire de travail.

2.4 Fonction genere_hyp().

* entrées :

  wm : référence à la mémoire de travail

* préconditions : ouvert(*wm)

* résultats : *wm'

* postconditions :

  *wm' : *wm modifié par aj_wm(e) pour tout e construit à partir des hypothèses faites sur l'énoncé courant en fonction du dernier énoncé de la machine contenu dans l'historique.

* fonction : Génération des hypothèses sur l'énoncé à traiter en fonction du dernier énoncé de la machine contenu dans l'historique, et chargement de ces hypothèses en mémoie de travail sous forme d'instances de relations.

2.5 Fonction aff_interp().

* entrées :

  fich : un fichier de type texte

  pf_res : fichier de résultat de l'interprétation

  result_int : référence à la structure de données correspondant à l'interprétation partielle de l'énoncé

* préconditions : ouvert(fich)
  ouvert(*pf_res)

* résultats : fich'
  *pf_res'

* postconditions :

fich' = append(fich, result_int)

*pf_res' = append(*pf_res, result_int)

* fonction : Affiche le résultat de l'interprétation contenu dans la structure référencée par "result_int" dans le fichier "fich" et dans le fichier résultat référencé par "pf_res".

2.6 Fonction cloture().

* entrées :

br : référence à la base de règles.
bf : référence à la base de faits.

* préconditions : ouvert(*bf)
ouvert(*br)

* résultats : aucun

* postconditions : aucune

* fonction : Réalisation des opérations de clôture, càd fermeture et sauvetage de la base de règles et de la base de faits.

2.7 Fonction interpréteur().

* entrées :

br : référence à la base de règles ;
bf : référence à la base de faits;
wm : référence à la mémoire de travail;

result_int : référence à la structure de données correspondant à l'interprétation partielle déjà réalisée.

cur_int : pointeur courant dans cette structure.

* préconditions : ouvert(*wm)
ouvert(*br)
ouvert(*bf)
vide(result_int)
result_int <> NULL
cur_int <> NULL

* résultats : *result_int'
cur_int'
*wm'
status : entier : diagnostic d'exécution.

* postconditions :

*wm' = *wm modifiée par l'exécution des différentes actions d'ajout et de suppression d'éléments, actions appartenant aux règles qui ont été activées.

status=OK => *result_int' contient le résultat de l'activation des règles, càd une interprétation de l'énoncé contenu dans la mémoire de travail.
cur_int' = cur_int + taille de cette interprétation.

status=ECHEC => l'interpréteur n'a pu aboutir à une interprétation complète de l'énoncé

*result_int' = ???
cur_int' = ???

status=ERREUR => l'intepréteur est tombé en erreur lors de l'activation d'une règle.

*result_int' = ???
cur_int' = ???

status=VIDE => l'intepréteur ne trouve plus de règles activables.
Si les règles sont bien écrites, cela revient au même que si status=OK.

* fonction : Implémentation de l'interpréteur proprement dit : réalisation du "recognize/act cycle" jusqu'à ce que l'on soit arrivé à un ordre de fin d'exécution, ou bien qu'il n'y ait plus de règles activables, ou bien encore que l'interpréteur soit bloqué en raison d'une erreur.

SPECIFICATIONS DU PROGRAMME GESTION BF.

1 Approche générale.

Le programme "gestion_bf" est le programme chargé de la gestion de la base de faits.

Il se greffe sur la hierarchie des modules du programme "interp" pour utiliser le module "base_faits", et donc indirectement tous les modules que ce dernier utilise.

Il utilise plus particulièrement les fonctions :

* ouv_bf()
* ferm_bf()
* intro_faits()
* modif_fait()
* suppr_fait()
* aff_bf()
* delete_bf()

Les types des données qu'il manipule sont celles définies dans l'interface du module "base_faits".

2 Spécifications de ce programme.

* entrées : aucune.

* préconditions : aucune.

* résultats : aucun

* postconditions : aucune

* fonction : gestion de la base de faits :
  ouverture de la base de faits ;
  affichage d'un menu des différentes opérations possibles ; exécution des opérations demandées par l'utilisateur en utilisant :

  * "intro_fait()" pour l'introduction de faits dans la base ;
  * "modif_fait()" pour la modification d'un fait dans la base ;
  * "suppr_fait()" pour la supression d'un fait dans la base ;

* "aff_bf()" pour l'affichage de la base de faits ;
* "delete_bf()" pour la desctruction du contenu de la base de faits ;
* "aff_bf()" pour l'impression du contenu de la base de faits dans un fichier. Pour cette dernière opération, le fichier cible est "b_faits.out".

Enfin, fermeture et sauvetage de la base de faits.

SPECIFICATIONS DU PROGRAMME GESTION_BR.

1 Aproche générale.

Le programme "gestion_br" est le programme chargé de la gestion de la base de règles.

Il se greffe sur la hierarchie des modules du programme "interp" pour utiliser le module "base_règles", et donc indirectement tous les modules que ce dernier utilise.

Il utilise plus particulièrement les fonctions :

* ouvrir_br()
* fermer_br()
* intro_règles()
* modif_règle()
* suppr_règle()
* aff_br()
* delete_br()

Les types des données qu'il manipule sont celles définies dans l'interface du module "base_règles".

2 Spécifications de ce programme.

* entrées : aucune.

* préconditions : aucune.

* résultats : aucun

* postconditions : aucune

* fonction : gestion de la base de règles :
  ouverture de la base de règles ;
  affichage d'un menu des différentes opérations possibles ; exécution des opérations demandées par l'utilisateur en utilisant :

  * "intro_règles()" pour l'introduction de règles dans la base ;
  * "modif_règle()" pour la modification d'un règle dans la base ;
  * "suppr_règle()" pour la supression d'un règle dans la base ;

* "aff_bf()" pour l'affichage de la base de règles ;
* "delete_br()" pour la desctruction du contenu de la base de règles ;
* "aff_br()" pour l'impression du contenu de la base de règles dans un fichier. Pour cette dernière opération, le fichier cible est "b_règles.out".

Enfin, fermeture et sauvetage de la base de règles.

PARTIE II.2 : MANUELS D'UTILISATION DES PROGRAMMES.

1 Le programme gestion_br.

Le programme "gestion_br" est le programme qui est chargé provisoirement de la gestion de la base de règles.

En raison de son caractère très provisoire, il est (vraiment) très rudimentaire, surtout au niveau de l'interface utilisateur.
Nous avons en effet estimé inutile de développer un programme très élaboré qui est destiné de toute façon à être remplacé dans un avenir proche.

1.1 Pré-requis.

Son exécution correcte nécessite la présence des fichiers "b_règles.file" et "relations.file" dans la directory courante, du moins s'ils existent.
S'ils n'existent pas, ils seront crées par le programme.

* "b_règles.file" est le fichier qui contient la base de règles sauvegardée depuis la dernière utilisation du programme ;

* "relations.file" est le fichier contenant les schémas de relations définis jusqu'à présent.

  Pour rappel, ce dernier fichier est également nécessaire car les conditions des règles sont basées sur le formalisme des relations.

1.2 Lancement du programme.

Le lancement du programme est réalisé par la commande :

```
@ gestion_br
```

1.3 Exécution du programme.

Son fonctionnement est basé sur un menu.
Après avoir tapé la commande de lancement du programme, l'utilisateur voit s'afficher le menu suivant à l'écran :

```
-----------------------------------------------------------------

    GESTION DE LA BASE DE REGLES.

  (i) insertion de règles dans la base ;
  (m) modification d'une règle de la base ;
  (s) suppression d'une règle de la base ;
  (d) destruction de la base de règles ;
  (a) affichage des règles contenues dans la base;
  (i) impression des règles contenues dans la base ;
  (f) fin ;

 >

-----------------------------------------------------------------
```

Après l'exécution de chaque commande (sauf la dernière), il reviendra à ce menu principal.

1.3.1 Insertion de règles dans la base.

S'il tape "i" comme réponse au menu principal, l'utilisateur déclenche la fonction d'insertion de règles dans la base.

Les conditions des règles sont d'abord introduites, une à une, puis les actions.

Le premier message que l'utilisateur verra à l'écran est le suivant :

```
-----------------------------------------------------------------

introduction des conditions.....

schéma >

-----------------------------------------------------------------
```

Ce message l'invite à introduire le schéma de relation correspondant à la première condition.

Le schéma de relation correspond ici au nom identificateur de la relation.

exemple : condition : verbe (x)

=> l'utilisateur tape

schéma > verbe <RET>

Le format est très libre(chaine de caractères quelconque, mais sans "blanc"), et il n'est pas nécessaire que la relation soit déjà définie dans le système.

Le schéma étant introduit, l'utilisateur sera invité à donner les arguments de la relation (ses variables) l'une après l'autre, un par ligne, chaque fois que le message " variable >" apparait .
La fin de la liste est signalée par l'utilisateur en tapant directement <RET> sur la ligne suivante directement après avoir introduit son dernier argument.

exemple :

---

introduction des conditions.....

schéma > verbe <RET>

variable > arg_1 <RET>

variable > arg_2 <RET>

variable > <RET>

schéma >

Si le schéma de relation était déjà défini dans le système, une vérification du nombre d'arguments est réalisée. Si le nombre est différent de celui connu par le programme, l'utilisateur est invité à retaper complètement la condition.

---

erreur dans l'introduction de la condition.....
veuillez recommencer, s.v.p.

schéma >

Sinon, le schéma de relation correspondant à la condition est ajouté à ceux déjà connus du système.

Après avoir introduit la dernière variable et signifié la fin des arguments par <RET>, l'utilisateur est invité à tapé la condition suivante par le message "schéma >".

Le processus se répète jusqu'à ce que l'utilisateur ait introduit toutes ses conditions. Il annonce la fin de la liste en tapant <RET> en réponse au dernier message " schéma >".

Après avoir introduit les conditions, le message suivant prévient l'utilisateur qu'il peut introduire ses actions :

---

introduction des actions.....

action >

L'utilisateur tape alors le nom de l'action.
Le message "argument >" l'invite ensuite à introduire l'argument en une seule fois sous forme de chaine de caractères (en une seule fois même si l'argument est une relation).

---

introduction des actions.....

action > ajouter <RET>
         ----

argument > verbe(x) <RET>
           ----

action > <RET>

Si le nom d'action tapé au terminal ou le type d'argument n'est pas valable, l'utilisateur est prié de réintroduire toute l'action.

---

erreur dans l'introduction de l'action....
veuillez recommencer, s.v.p.

action >

Après avoir tapé une action et son argument, l'utilisateur est invité à taper la suivante si elle existe par le message " action >" qui s'affiche sur la dernière ligne.

Le processus se répète ainsi pour toutes les actions, et l'utilisateur signifie la fin de la liste en tapant <RET> en réponse au dernier message "action >".

La règle étant complètement introduite, l'utilisateur aura à spécifier s'il désire introduire une autre règle en répondant au message :

---

action > <RET>
         -----

introduction d'une autre règle (<RET>=non) ?

S'il tape un caractère quelconque, tout le processus décrit dans cette section recommence pour une autre règle.
Sinon, s'il tape <RET>, l'utilisateur se voit revenir au menu principal.

1.3.2 Modification d'une règle.

En tapant "m" comme réponse au menu principal, l'utilisateur déclenche la fonction de modification d'une règle.

Il est invité à introduire l'identifiant de la règle qu'il désire modifier par le message :

```
-------------------------------------------------------------

   identificateur de la règle à modifier ?

-------------------------------------------------------------
```

L'identifiant de la règle qui doit être introduit est un entier.
Si l'utilisateur tape autre chose (quoi que ce soit), ou bien un entier n'identifiant aucune règle de la base, il en est averti par le message d'erreur :

```
-------------------------------------------------------------

  identifiant de règle non valable

-------------------------------------------------------------
```

et est automatiquement renvoyé au menu principal.

Sinon, la règle ainsi identifiée est supprimée de la base de règles et l'utilisateur est invité à en introduire une autre suivant les même modalités que celles décrites dans la section précédente.

Après introduction de cette règle correspondant à la version modifiée de la règle supprimée, l'utilisateur est renvoyé au menu principal.

1.3.3 Suppression d'une règle.

Après avoir répondu "s" au menu principal, l'utilisateur déclenche la fonction de suppression d'une règle de la base.

Il est invité à introduire l' identifiant de la règle qu'il désire supprimer par le message :

---

identificateur de la règle à supprimer ?

S'il tape autre chose qu'un entier référençant une règle de la base, l'utilisateur est en est averti et est renvoyé au menu principal.

---

identifiant de règle non valable

Sinon, la règle ainsi identifiée est supprimée de la base de règles et retour au menu principal.

1.3.4 Destruction de la base.

S'il tape "d" en réponse au menu principal, l'utilisateur se déclenche la fonction de destruction de la base de règles.

Dans ce cas, son contenu sera complètement et irrémédiablement détruit.

Après cette opération, le programme retourne au menu principal.

1.3.5 Affichage du contenu de la base.

En répondant "a" au menu principal, l'utilisateur accède à la fonction d'affichage de la base de règles.

Cette dernière sera donc affichée au terminal de la manière suivante :

```
-------------------------------------------------------------------

  -------------BASE DE REGLES-------------

   identificateur de règle : id_1

           cdt_1_1
         & cdt_1_2
           .
           .
         & cdt_1_m

      =>  act_1_1
        & act_1_2
           .
           .

        & act_1_n

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

next >

-------------------------------------------------------------------
```

Pour l'affichage de la règle suivante, l'utilisateur tape n'importe quel caractère en réponse au message "next >". Ce mécanisme (rudimentaire !!!) permet de contrôler le déroulement de l'écran.

Après affichage de la base de règles, le programme retourne au menu principal.

1.3.5.1 Impression du contenu de la base.

S'il tape "p" comme réponse au menu principal, l'utilisateur déclenche l'exécution d'impression du contenu de la base de règles dans le fichier "b_règles.out".

Ce fichier de type texte contiendra la base de règles sous la forme suivante :

```
-------------BASE DE REGLES-------------

  identificateur de règle : id_1

          cdt_1_1
        & cdt_1_2
          .
          .
        & cdt_1_m

     =>  act_1_1
       & act_1_2
          .
          .

       & act_1_n

---------------------------------------
            .
            .
            .
---------------------------------------

  identificateur de règle : id_i

          cdt_i_1
        & cdt_i_2
          .
          .
        & cdt_i_o

     =>  act_i_1
       & act_i_2
          .
          .

       & act_i_p

--------------------------
```

L'utilisateur est averti de la fin de l'exécution de cette opération par l'affichage du message :

Les règles sont dans b_règles.out.

Après affichage de ce message, le programme retourne au menu principal.

1.3.6 Fin d'exécution.

S'il tape "f" en réponse au menu principal, l'utilisateur commande la fin du programme "gestion_br".

Le contenu de la base de règles est alors sauvé dans le fichier "b_règles.file" pour une utilisation ultérieure.
Le fichier "relations.file" est également mis à jour.

2 Le programme gestion bf.

Le programme "gestion_bf" est le programme qui est chargé provisoirement de la gestion de la base de faits.

En raison de son caractère très provisoire, il est (vraiment) très rudimentaire, surtout au niveau de l'interface utilisateur, tout comme "gestion_br".

2.1 Pré-requis.

Son exécution correcte nécessite la présence des fichiers "b_faits.file", "relations.file", et "valeurs.file" dans la directory courante, du moins s'ils existent.
S'ils n'existent pas, ils seront crées par le programme.

* "b_faits.file" est le fichier qui contient la base de faits sauvegardée depuis la dernière utilisation du programme ;

* "relations.file" est le fichier contenant les schémas de relations définis jusqu'à présent.

* "valeurs.file" est le fichier contenant les valeurs des arguments de relations.

Pour rappel, ces 2 derniers fichiers sont également nécessaires car les faits sont basés sur le formalisme des relations.

2.2 Lancement du programme.

Le lancement du programme est réalisé par la commande :

```
@ gestion_bf
```

2.3 Exécution du programme.

Son fonctionnement est basé sur un menu.
Après avoir tapé la commande de lancement du programme, l'utilisateur voit s'afficher le menu suivant à l'écran :

```
-----------------------------------------------------------------

    GESTION DE LA BASE DE FAITS.

  (i) insertion de faits dans la base ;
  (m) modification d'un fait de la base ;
  (s) suppression d'un fait de la base ;
  (d) destruction de la base de faits ;
  (a) affichage des faits contenus dans la base;
  (i) impression des faits contenus dans la base ;
  (f) fin ;

 >

-----------------------------------------------------------------
```

Après l'exécution de chaque commande (sauf la dernière), il reviendra à ce menu principal.

2.3.1 Insertion de faits dans la base.

S'il tape "i" comme réponse au menu principal, l'utilisateur déclenche la fonction d'insertion de règles dans la base.

Le premier message que l'utilisateur verra à l'écran est le suivant :

```
-----------------------------------------------------------------

 .....introduction des faits.....

schéma >

-----------------------------------------------------------------
```

Ce message l'invite à introduire le schéma de relation correspondant au premier fait.

Tout comme pour les conditions des règles, le schéma de relation correspond ici au nom identificateur de la relation.

exemple : fait : document (passeport)

=> l'utilisateur tape

schéma > document <RET>

Le format est très libre(chaine de caractères quelconque, mais sans "blanc"), et il n'est pas nécessaire que la relation soit déjà définie dans le système.

Le schéma étant introduit, l'utilisateur sera invité à donner l' argument de la relation par le message " argument >".

exemple :

.....introduction des faits.....

schéma > document <RET>

argument > arg <RET>

schéma >

Si le schéma de relation était déjà défini dans le système, une vérification du nombre d'arguments est réalisée. Si le nombre connu par le programme est différent de 1 , le fait n'est pas accepté et l'utilisateur est invité à en taper un autre.

---

erreur dans l'introduction du fait.
veuillez recommencer, s.v.p.

schéma >

Sinon, le schéma de relation correspondant au fait est ajouté à ceux déjà connus du système.

Après avoir introduit l'argument, l'utilisateur est invité à

préciser s'il désire introduire un nouveau fait en répondant à la question :

---

introduction d'un autre fait (<RET>=non) ?

S'il tape un caractère quelconque, le processus décrit dans cette section recommence pour un autre fait.
Sinon, s'il tape <RET>, l'utilisateur se voit revenir au menu principal.

2.3.2 Modification d'un fait.

En tapant "m" comme réponse au menu principal, l'utilisateur déclenche la fonction de modification d'un fait.

Il est invité à introduire l'identifiant de ce fait qu'il désire modifier par le message :

---

identificateur du fait à modifier ?

L'identifiant du fait doit être un entier.
Si l'utilisateur tape autre chose (quoi que ce soit), ou bien un entier n'identifiant aucun fait de la base, il en est averti par le message d'erreur :

---

identifiant de fait non valable

---

et est automatiquement renvoyé au menu principal.

Sinon, le fait ainsi identifié est supprimé de la base de

faits et l'utilisateur est invité à en introduire un autre suivant les même modalités que celles décrites dans la section précédente.

Après introduction du fait correspondant à la version modifiée du fait supprimé, l'utilisateur est renvoyé au menu principal.

2.3.3 Suppression d'un fait.

Après avoir répondu "s" au menu principal, l'utilisateur déclenche la fonction de suppression d'un fait de la base.

Il est invité à introduire l' identifiant du fait qu'il désire supprimer par le message :

```
-----------------------------------------------------------------

   identificateur du fait à supprimer ?

-----------------------------------------------------------------
```

S'il tape autre chose qu'un entier référençant un fait de la base, l'utilisateur est en est averti et est renvoyé au menu principal.

```
-----------------------------------------------------------------

  identifiant de fait non valable

-----------------------------------------------------------------
```

Sinon, le fait ainsi identifié est supprimé de la base de faits et retour au menu principal.

2.3.4 Destruction de la base.

S'il tape "d" en réponse au menu principal, l'utilisateur se déclenche la fonction de destruction de la base de faits.

Dans ce cas, son contenu sera complètement et irrémédiablement détruit.

Après cette opération, le programme retourne au menu principal.

2.3.5 Affichage du contenu de la base.

En répondant "a" au menu principal, l'utilisateur accède à la fonction d'affichage de la base de faits.

Cette dernière sera donc affichée au terminal de la manière suivante :

```
-----------------------------------------------------------------

  -------------base de faits-------------

  id : id_1      fait_1
  id : id_2      fait_2
   .               .
   .               .
   .               .
  id : id_n      fait_n


 >

-----------------------------------------------------------------
```

Pour arrêter de consulter cette base, l'utilisateur tape n'importe quel caractère en réponse au message " >".

Après affichage de la base de faits, le programme retourne au menu principal.

2.3.5.1 Impression du contenu de la base.

S'il tape "p" comme réponse au menu principal, l'utilisateur déclenche l'exécution d'impression du contenu de la base de faits dans le fichier "b_faits.out".

Ce fichier de type texte contiendra la base de règles sous la forme suivante :

```
-------------base de faits-------------

id : id_1      fait_1
id : id_2      fait_2
 .              .
 .              .
 .              .
id : id_n      fait_n
```

L'utilisateur est averti de la fin de l'exécution de cette opération par l'affichage du message :

Les faits sont dans b_faits.out.

Après affichage de ce message, le programme retourne au menu principal.

2.3.6 Fin d'exécution.

S'il tape "f" en réponse au menu principal, l'utilisateur commande la fin du programme "gestion_bf".

Le contenu de la base de faits est alors sauvé dans le fichier "b_faits.file" pour une utilisation ultérieure.
Les fichiers "relations.file" et "valeurs.file" sont également mis à jour.

3 Le programme interp.

Le programme "interp" est celui qui implémente l'interpréteur de règles de production.

Son utilisation est très simple, car l'aide de l'utilisateur n'est demandé que pour la réalisation des opérations qui ne sont pas encore implémentées : traduction de l'énoncé et chargement en mémoire de travail, et simulation des accès à l'historique pour la résolution des références (action "hist_référé") et des ellipses (action "en_machine").

3.1 Pré-requis.

Son exécution correcte nécessite la présence des fichiers "b_règles.file", "b_faits.file", "relations.file", et "valeurs.file" dans la directory courante, du moins s'ils existent.
Elle requiert impérativement la présence du fichier "actions.file".

* "b_règles.file" est le fichier qui contient la base de règles telle qu'elle a été sauvegardée lors de la dernière utilisation des programmes ;

* "b_faits.file" est le fichier qui contient la dernière version de la base de faits ;

* "relations.file" est le fichier contenant les schémas de relations définis jusqu'à présent.

* "valeurs.file" est le fichier contenant les valeurs des arguments de relations.

* "actions.file" est le fichier contenant la table des actions définies sur le système.
  On comprend donc aisément pourquoi sa présence est indispensable.

3.2 Lancement du programme.

Le lancement du programme est réalisé par la commande :

```
@ interp
```

Une option est possible pour la trace d'exécution. Si l'utilisateur désire ne pas voir la trace d'exécution en

interactif, il tapera :

```
@ interp -s
```

ce qui aura pour effet de limiter l'interaction entre le programme et l'utilisateur aux seuls éléements nécessaires à son fonctionnement.
Dans ce cas, une version de la trace est disponible dans le fichier "trace.out".

3.3 Exécution

L'interaction entre l'utilisateur et le programme se limite à la trace d'exécution (si telle est l'option prise) et à la simulation des fonctions non encore implémentées.

Dans cette section, nous ne nous étendrons pas sur des exemples de fonctionnement, la partie suivante consacrée aux exemples d'exécution nous larraissant largement suffisante.

Après avoir tapé la commande de lancement du programme, l'utilisateur se verra invité à introduire le contenu initial de la mémoire de travail, c'est-à-dire la représentation interne de l'énoncé en entrée.

La traduction de l'énoncé n'étant pas implémentée, le contenu de la mémoire de travail doit être introduit à la main.

schéma >

La procédure à suivre pour l'introduction des éléements de mémoire de travail est rigoureusement la même que celle utilisée pour l'introduction des conditions dans le programme "gestion_br".
Nous renvoyons donc le lecteur à cette section pour plus d'informations.

Une fois que le contenu de la mémoire de travail est introduit, le programme s'exécute tout seul, en permettant à l'utilisateur de suivre son travail si l'option "-s" n'a pas été choisie.
Pour ce faire, il s'arrête avant l'activation de chaque instance de règles sur le message "next >".
L'utilisateur lui rendra la main en tapant n'importe quel caractère.

Pour chaque instance activée, il affichera l'identifiant de la règle, le jeu de substitution, et le suivi pas à pas de l'exécution de ses actions.

Lorsqu'il rencontre une action de type "en_machine", le programme invite l'utilisateur à communiquer la dernier énoncé machine sous forme de formule.

Simulation de l'accès à l'historique....

tapez le dernier énoncé machine s.v.p.

> aller(locuteur,?lieu)

Lorsqu'il rencontre une action de type "hist_référé", il invite alors l'utilisateur à introduire les groupes nominaux des derniers échanges susceptibles d'être référencés par les syntagmes anaphoriques présents dans l'énoncé.

Simulation de l'accès à l'historique...

Introduisez successivement les différents groupes nominaux susceptibles d'être référencés par le pronom traité.

next >

L'utilisateur les tapera alors sous forme de chaine de caractères. Il spécifiera la fin de la liste en tapant directement <RET> en réponse au dernier message "next >".

Lorsque l'interpréteur a terminé, le résultat de son exécution est affiché au terminal, et sauvé dans le fichier "res_interp", qui est un simple fichier de type texte.

PARTIE II.3 : EXEMPLES D'EXECUTION DES PROGRAMMES.

CONTENU DE LA BASE DE FAITS :

```
--------------base de faits---------------
id : 0   type_oui_non (oui)
id : 1   type_oui_non (non)
id : 2   prem_personne (je)
id : 3   singulier (je)
id : 4   id_national (carte_identite)
id : 5   reponse_aff (oui)
id : 6   reponse_neg (non)
id : 7   reponse (oui)
id : 8   reponse (non)
id : 9   lieu (Nancy)
```

CONTENU DE LA BASE DE REGLES :

```
---------------BASE DE REGLES---------------

identificateur de regle : 0

   predicat(#x)
 & obtention(#x)
 & cas_objet(#x,#o)
 & groupe_nominal(#o)
 & cas_benef(#x,#b)

 => ajouter(str_simple(#x,#b,#o))
  & ajouter(requete(#x))
  & ajouter(objet_requete(#o))
-----------------------------

   identificateur de regle : 1

   interp_predicat(#x)
 & requete(#x)
 & obtention(#x)

 => ajout_int("obtenir)
-----------------------------

   identificateur de regle : 2

   interp_personne(#x)
 & prem_personne(#x)
 & singulier(#x)

 => ajout_int("locuteur)
-----------------------------

   identificateur de regle : 3

   str_simple(#a,#b,#c)
 & requete(#a)
 & objet_requete(#c)

 => ajout_int("TYP-ENON: requete /)
  & ajout_int(" BUT :)
  & ajouter(interp_predicat(#a))
  & interpreter(#a)
  & ajout_int("()
  & ajouter(interp_personne(#b))
  & interpreter(#b)
  & ajout_int(",)
  & ajouter(interp_objet(#c))
  & interpreter(#c)
  & ajout_int("))
  & ajout_int("/ INFO: null / QUESTION: ?action /)
-----------------------------
```

```
   identificateur de regle : 4

   interp_objet(#x)
 & objet_requete(#x)
 & id_national(#x)

 => ajout_int("carte_identite)
-----------------------------

   identificateur de regle : 5

   question(dern_en_machine)
 & reponse(#x)
 & str_stereotypee(#x)
 & type_oui_non(#x)

 => ajouter(reponse_simple(#x))
-----------------------------

   identificateur de regle : 6

   question(dern_en_machine)
 & reponse(#x)
 & str_stereotypee(#x)

 => ajout_int("TYP-ENON: apport-info /)
  & ajout_int(" BUT:null / )
  & ajout_int("INFO:)
  & en_machine(#x)
  & ajout_int(" / QUESTION:null /)
-----------------------------

   identificateur de regle : 7

   groupe_nominal(#x)
 & ~predicat(#x)

 => ajouter(str_stereotypee(#x))
-----------------------------

   identificateur de regle : 8

   reponse_simple(#x)
 & reponse_neg(#x)

 => ajout_int("~)
-----------------------------

   identificateur de regle : 9

   groupe_nominal(#x)
 & lieu(#x)

 => ajout_int(#x)
-----------------------------

   identificateur de regle : 10

   reponse_simple(#x)
 & reponse_aff(#x)

 => ajout_int(")
-----------------------------
```

1) EXEMPLE NUMERO 1.

ENONCE EN ENTREE : " JE DESIRE UNE CARTE D'IDENTITE"

---------- TRACE D'EXECUTION ----------

la traduction de l'enonce n'etant pas implementee, le
contenu de la memoire de travail doit etre introduit a la main.

```
schema >
predicat
argument > desire
argument >
schema > obtention
argument > desire
argument >
schema > cas_objet
argument > desire
argument > carte_identite
argument >
schema > groupe_nominal
argument > carte_identite
argument >
schema > cas_benef
argument > desire
argument > je
argument >
schema >

 --------------memoire de travail---------------
predicat (desire)  0
obtention (desire)  0
cas_objet (desire, carte_identite)  0
groupe_nominal (carte_identite)  0
cas_benef (desire, je)  0

*****************************************************

 cycle : 1


 debut d'execution de la regle :

  id de regle : 0

substitution :
#x  -> desire
#o  -> carte_identite
#b  -> je
```

```
 next >
-----------------------------------------------


 ajout de l'element suivant en WM :
str_simple (desire, je, carte_identite)  1

 diagnostic d'execution :  O.K.

 ajout de l'element suivant en WM :
requete (desire)  1

 diagnostic d'execution :  O.K.

 ajout de l'element suivant en WM :
objet_requete (carte_identite)  1

 diagnostic d'execution :  O.K.
fin d'execution de la regle courante

diagnostic d'execution :  O.K.
-----------------------------------------------


***************************************************

 cycle : 2


 debut d'execution de la regle :

  id de regle : 3

substitution :
#a  -> desire
#b  -> je
#c  -> carte_identite


 next >
-----------------------------------------------


 ajout de la partie d'interpretation suivante :
TYP-ENON: requete /
 diagnostic d'execution :  O.K.

 ajout de la partie d'interpretation suivante :
 BUT :
 diagnostic d'execution :  O.K.

 ajout de l'element suivant en WM :
interp_predicat (desire)  2

 diagnostic d'execution :  O.K.

 execution d'un appel recursif a l'interpreteur.
identificateur de variable concernee : O
element terminal
```

```
**********************************************************

 cycle : 1


 debut d'execution de la regle :

  id de regle : 1

substitution :
#x  -> desire


 next >
----------------------------------------------


 ajout de la partie d'interpretation suivante :
obtenir
 diagnostic d'execution :  O.K.
fin d'execution de la regle courante

diagnostic d'execution :  O.K.
----------------------------------------------


 plus de regles activables...

fin de l'appel recursif du cycle 2
diagnostic d'execution de cette action : O.K.

 ajout de la partie d'interpretation suivante :
(
 diagnostic d'execution :  O.K.

 ajout de l'element suivant en WM :
interp_personne (je)  2

 diagnostic d'execution :  O.K.

 execution d'un appel recursif a l'interpreteur.
identificateur de variable concernee : 1
element terminal

**********************************************************

 cycle : 1


 debut d'execution de la regle :

  id de regle : 2

substitution :
#x  -> je


 next >
----------------------------------------------
```

```
 ajout de la partie d'interpretation suivante :
locuteur
 diagnostic d'execution :  O.K.
fin d'execution de la regle courante

diagnostic d'execution :  O.K.
-----------------------------------------------


 plus de regles activables...

fin de l'appel recursif du cycle 2
diagnostic d'execution de cette action : O.K.

 ajout de la partie d'interpretation suivante :
,
 diagnostic d'execution :  O.K.

 ajout de l'element suivant en WM :
interp_objet (carte_identite)  2

 diagnostic d'execution :  O.K.

 execution d'un appel recursif a l'interpreteur.
identificateur de variable concernee : 2
element terminal

*****************************************************

 cycle : 1


 debut d'execution de la regle :

  id de regle : 4

substitution :
#x  -> carte_identite


 next >
-----------------------------------------------


 ajout de la partie d'interpretation suivante :
carte_identite
 diagnostic d'execution :  O.K.
fin d'execution de la regle courante

diagnostic d'execution :  O.K.
-----------------------------------------------


 plus de regles activables...
```

```
fin de l'appel recursif du cycle 2
diagnostic d'execution de cette action : O.K.

 ajout de la partie d'interpretation suivante :
)
 diagnostic d'execution :  O.K.

 ajout de la partie d'interpretation suivante :
/ INFO: null / QUESTION: ?action /
 diagnostic d'execution :  O.K.




fin d'execution de la regle courante

diagnostic d'execution :  FIN
-------------------------------------------

/* resultat de l'interpretation */

 TYP-ENON: requete /
 BUT : obtenir(locuteur, carte_identite)/
 INFO: null /
 QUESTION: ?action /



--------------memoire de travail---------------
predicat (desire)  0
obtention (desire)  0
cas_objet (desire, carte_identite)  0
groupe_nominal (carte_identite)  0
cas_benef (desire, je)  0
str_simple (desire, je, carte_identite)  1
requete (desire)  1
objet_requete (carte_identite)  1
interp_predicat (desire)  2
interp_personne (je)  2
interp_objet (carte_identite)  2
```

2) EXEMPLE NUMERO 2.

ENONCE EN ENTREE : "NON"

DERNIER ENONCE DE LA MACHINE : "ETES-VOUS MAJEUR?"

---------- TRACE D'EXECUTION ----------

la traduction de l'enonce n'etant pas implementee, le
contenu de la memoire de travail doit etre introduit a la main.

```
schema > question
argument > dern_en_machine
argument >
schema > reponse
argument > non
argument >
schema > groupe_nominal
argument > non
argument >
schema > ~predicat
argument > non
argument >
schema >

--------------memoire de travail--------------
question (dern_en_machine)  0
reponse (non)  0
groupe_nominal (non)  0
~predicat (non)  0

*****************************************************

 cycle : 1


 debut d'execution de la regle :

  id de regle : 7

substitution :
#x  -> non


 next >
-----------------------------------------------
```

```
 ajout de l'element suivant en WM :
str_stereotypee (non)  1

 diagnostic d'execution :  O.K.
fin d'execution de la regle courante

diagnostic d'execution :  O.K.
-----------------------------------------


*****************************************************

 cycle : 2


 debut d'execution de la regle :

  id de regle : 5

substitution :
dern_en_machine  -> dern_en_machine
#x  -> non


 next >
-----------------------------------------


 ajout de l'element suivant en WM :
reponse_simple (non)  2

 diagnostic d'execution :  O.K.
fin d'execution de la regle courante

diagnostic d'execution :  O.K.
-----------------------------------------


*****************************************************

 cycle : 3


 debut d'execution de la regle :

  id de regle : 6

substitution :
dern_en_machine  -> dern_en_machine
#x  -> non


 next >
-----------------------------------------


 ajout de la partie d'interpretation suivante :
TYP-ENON: apport-info /
 diagnostic d'execution :  O.K.
```

```
 ajout de la partie d'interpretation suivante :
 BUT: null /
 diagnostic d'execution :  O.K.

 ajout de la partie d'interpretation suivante :
INFO:
 diagnostic d'execution :  O.K.

 execution d'un appel recursif a l'interpreteur.
identificateur de variable concernee : 1
element terminal

*********************************************************

 cycle : 1

 debut d'execution de la regle :

  id de regle : 8

substitution :
#x  -> non

 next >
-----------------------------------------

 ajout de la partie d'interpretation suivante :
~
 diagnostic d'execution :  O.K.
fin d'execution de la regle courante

diagnostic d'execution :  O.K.
-----------------------------------------

 plus de regles activables...

fin de l'appel recursif du cycle 2
diagnostic d'execution de cette action : O.K.

 Simulation de l'acces a l'historique...

 introduisez le dernier enonce_machine, s.v.p.
 >? majeur(locuteur)

 ajout de la partie d'interpretation suivante :
 / QUESTION: null /
 diagnostic d'execution :  O.K.
fin d'execution de la regle courante

diagnostic d'execution :  O.K.
-----------------------------------------

 plus de regles activables...

 /* resultat de l'interpretation */
```

```
TYP-ENON: apport-info /
BUT: null /
INFO: ~ majeur(locuteur) /
QUESTION: null /


---------------memoire de travail---------------
question (dern_en_machine)  0
reponse (non)  0
groupe_nominal (non)  0
~predicat (non)  0
str_stereotypee (non)  1
reponse_simple (non)  2
```

3) EXEMPLE NUMERO 3.

ENONCE EN ENTREE : "A NANCY".

DERNIER ENONCE MACHINE : "OU ALLEZ-VOUS ?"

---------- TRACE D'EXECUTION ----------

la traduction de l'enonce n'etant pas implementee, le
contenu de la memoire de travail doit etre introduit a la main.

```
schema > question
argument > dern_en_machine
argument >
schema > reponse
argument > Nancy
argument >
schema > groupe_nominal
argument > Nancy
argument >
schema > ~predicat
argument > Nancy
argument >
schema >

--------------memoire de travail--------------
question (dern_en_machine)  0
reponse (Nancy)  0
groupe_nominal (Nancy)  0
~predicat (Nancy)  0

*****************************************************

 cycle : 1


 debut d'execution de la regle :

  id de regle : 7

substitution :
#x  -> Nancy


 next >
```

```
 ajout de l'element suivant en WM :
str_stereotypee (Nancy)  1

 diagnostic d'execution :  O.K.
fin d'execution de la regle courante

diagnostic d'execution :  O.K.
-----------------------------------------------


**************************************************

 cycle : 2


 debut d'execution de la regle :

  id de regle : 6

substitution :
dern_en_machine  -> dern_en_machine
#x  -> Nancy


 next >
-----------------------------------------------


 ajout de la partie d'interpretation suivante :
TYP-ENON: apport-info /
 diagnostic d'execution :  O.K.

 ajout de la partie d'interpretation suivante :
 BUT:null /
 diagnostic d'execution :  O.K.

 ajout de la partie d'interpretation suivante :
INFO:
 diagnostic d'execution :  O.K.

 execution d'un appel recursif a l'interpreteur.
identificateur de variable concernee : 1
element terminal

**************************************************

 cycle : 1


 debut d'execution de la regle :

  id de regle : 9

substitution :
#x  -> Nancy


 next >
-----------------------------------------------
```

```
 ajout de la partie d'interpretation suivante :
Nancy
 diagnostic d'execution :  O.K.
fin d'execution de la regle courante

diagnostic d'execution :  O.K.
-----------------------------------------------


 plus de regles activables...

fin de l'appel recursif du cycle 2
diagnostic d'execution de cette action : O.K.

 Simulation de l'acces a l'historique...

 introduisez le dernier enonce_machine, s.v.p.
 >aller(locuteur,?lieu)

 ajout de la partie d'interpretation suivante :
 / QUESTION: null /
 diagnostic d'execution :  O.K.
fin d'execution de la regle courante

diagnostic d'execution :  O.K.
-----------------------------------------------


 plus de regles activables...


  /* resultat de l'interpretation */

 TYP-ENON:  apport-info /
 BUT: null /
 INFO: aller(locuteur,Nancy) /
 QUESTION: null /


--------------memoire de travail---------------
question (dern_en_machine)  0
reponse (Nancy)  0
groupe_nominal (Nancy)  0
~predicat (Nancy)  0
str_stereotypee (Nancy)  1
```

PARTIE II.4 : TEXTES DE PROGRAMMES.

```
/*****************************************************************/
/*                    fichier des déclarations                   */
/*****************************************************************/


#include "const.h"
#include "conflict_set.h"
#include "relations.h"
#include "base_regles.h"
#include "workmem.h"
#include "base_faits.h"
#include "gest_trace.h"
#include "outils.h"
#include "activation.h"
```

```
/*****************************************************************/
/* fichier des declarations pour le module activation.           */
/*****************************************************************/

struct stack { int ptr;
               struct l_el_wm *elem[5];
             } *pile;
```

```
/*****************************************************************/
/* fichier des declarations pour le module base_faits.           */
/*****************************************************************/


struct fait_externe { struct rel *r;
                      char *arg;
                    };

struct fait { int id_rel;
              int arg;
            };

struct l_faits { int l;
                 struct fait *elem[NMAXELEM];
               } *bf;
```

```
/*****************************************************************/
/* fichier des declarations pour le module base_regles.      */
/*****************************************************************/


struct tab { int l;
             char *val[NTYPACT];
           } *T_actions;

struct cond { int id_rel;
              int l_var[NMAXARG];
            };

struct act { int id_act;
             int l_arg;
             int *arg;
           };

struct regle { int nb_cond;
               struct cond *l_cond[NMAXCOND];
               int nb_act;
               struct act *l_act[NMAXACT];
               int nb_var;
               char *l_var[NMAXVAR];
             };

struct l_regles { int l;
                  struct regle *ptr_rg[NMAXRG];
                } *br;

struct cond_externe { char *nom;
                      int nb_arg;
                      char *tabvar[NMAXVAR];
                    };

struct act_externe { char *nom_act;
                     char * arg;
                   };

struct arg_interp { int strc;
                    int var;
                  };

struct arg_in { int t_arg;
                union {
                        int v;
                        char const[30];
                      } uval;
              };

struct arg_ellipse { int t_arg;
                     union {
                             int v;
                             char const[30];
                           } uvaleur;
                   };

struct arg_wm { int id_rel;
                int nb_arg;
                int argrel[NMAXARG];
              };

struct rg_externe { int nb_cond;
                    struct cond_externe *tabcond[NMAXCOND];
                    int nb_act;
```

```
                    struct act_externe *tabact[NMAXACT];
                  };

struct ls_cond { int l;
                 struct cond *el[NMAXCOND];
               };

struct ls_act { int l;
                struct act *el[NMAXACT];
              };

struct inst_regle { int id_rg;
                    int l_val_var[NMAXVAR];
                  };

struct l_inst_regles { int l;
                       struct inst_regle *elem[NMAXSUB];
                     };
struct l_id_regles { int l;
                     int elem[NMAXRG];
                   };
```

```
/*****************************************************************/
/* fichier des declarations pour le module conflict_set.         */
/*****************************************************************/


struct sub { int l;
             int elem[NMAXVAR][2];
           };

typedef struct ls_sub { int l;
                        struct sub *elem[NMAXSUB];
                      } SUB;
```

```
/*****************************************************************/
/*                    fichier des constantes.                    */
/*****************************************************************/


#define VRAI                    1
#define FAUX                    0

#define ECHEC                  -1
#define ERREUR                  0
#define OK                      1
#define FIN                     2
#define RUNNING                 3
#define VIDE                    4

#define NMAXELEM              100
#define NMAXCOND               20
#define NMAXACT                20
#define NMAXVAR                 5
#define NMAXRG                100
#define NMAXARG                 5
#define NTYPACT                 8
#define NMAXSUB                20

#define OFF_LINE                0
#define ON_LINE                 1

#define CONST                   0
#define VARIABLE                1
```

```
/*************************************************************/
/* fichier des declarations pour le module gest_trace.       */
/*************************************************************/

struct trc_trace { int l;
                   struct trc_ir *elem[NMAXRG];
                 } *trace;

struct trc_ir { int cycle;
                struct inst_regle ir;
              };

FILE *trc_pf;
```

```
/*****************************************************************/
/* fichier des declarations pour le module outils.               */
/*****************************************************************/

struct tabint { int l;
                int elem[NMAXRG];
              };
```

```
/*****************************************************************/
/* fichier des declarations pour le module relations.            */
/*****************************************************************/

struct rel { char *nom;
             int nb_arg;
           };

struct rel_ext { char *r;
                 int nb_arg;
                 char *arg[NMAXARG];
               };


struct relat { int l;
               struct rel *ptr_rel[100];
             } *sch_rel;


struct valeurs { int l;
                 char *ptr_val[100];
               } *val;
```

```
/*****************************************************************/
/* fichier des declarations pour le module workmem.              */
/*****************************************************************/


struct el_externe
        { struct rel *r;
          char *arg[NMAXARG];
        };

struct el_wm
        { int id_rel;
          int age;
          int n_arg;
          int arg[NMAXARG];
        };

struct l_el_wm
        { int l;
          struct el_wm *elem[NMAXELEM];
        } *wm;

struct l_id_el { int l;
                 int elem[NMAXELEM];
               };

struct l_id_val { int l;
                  int elem[NMAXELEM];
                };
```

```
/*****************************************************************/
/*                        MODULE ERREURS                         */
/*****************************************************************/


#include <stdio.h>


erreur1()

{ printf("\n argument non significatif \n");
  printf("\n requete ignoree...  \n");
};

erreur2()

{
 printf("echec de l'interpreteur...consultez la trace.\n");
};

erreur3()

{
 printf("interpreteur interrompu par une erreur...");
 printf("consultez la trace.\n");
};

erreur4()

{
 printf("interpreteur renvoie un diag inattendu...");
 printf("erreur de programmation !!!\n");
};


erreur5()

{ printf("identificateur de regle non valable.\n");
};

erreur6()

{ printf("desole, je ne comprends pas.Recommencez, s.v.p.");
};
```

```
/*                                                                          */
/*                          MODULE OUTILS.C                                 */
/*                                                                          */
/****************************************************************************/
/*                                                                          */
/* Ce module contient les fonctions "outils" suivantes :                    */
/*                                                                          */
/*     . search_T()                                                         */
/*     . nom()                                                              */
/*     . sort()                                                             */
/*     . inclusion()                                                        */
/*     . pres_int()                                                         */
/*                                                                          */
/* Pour les specifications completes et precises, voir outils.spec.         */
/*                                                                          */
/****************************************************************************/


#include <stdio.h>
 include "declarations.h"


/****************************************************************************/
/* recherche si la chaine 'chaine' se trouve dans le tableau de chaines     */
/* de caracteres 'tabc', de longueur 'long'.                                */
/* Si c'est le cas, retourne son adresse relative dans ce tableau,          */
/*                  sinon retourne '-1'.                                    */
/****************************************************************************/


search_T(tabc,l,chaine)

char *tabc[100];                        /* tabl. de chaines de car.     */
int l;                                  /* longueur de ce tableau       */
char *chaine;                           /* ptr vers la chaine recherchee*/

{ int i;

  for (i=0;i<l;i++)
     { if (!(strcmp(tabc[i],chaine)))
         return(i);
     };
  return(-1);
};


/****************************************************************************/
/* retourne la chaine de caracteres dont l'adresse relative dans 'tabc' */
/* est 'ind'.                                                               */
/****************************************************************************/


char *nom(tabc,ind)

char *tabc[100];                        /* tabl. de chaines de car.     */
int ind;                                /* indice de la chaine recherche*/

{
return (tabc[ind]);
};


/****************************************************************************/
```

```
/* trie une table d'entiers par ordre croissant.                          */
/************************************************************************/

sort(t)

struct tabint *t;                          /* ptr vers le tableau d'entiers*/

{ int i,j,k,el;

  i=0;
  j=t->l-1;

  while (i<j)
       { k=i;
         while (k<j)
            { if (t->elem[k]<t->elem[i])
                 { el=t->elem[i];
                   t->elem[i]=t->elem[k];
                   t->elem[k]=el;
                 }
              else if (t->elem[k]>t->elem[j])
                      { el=t->elem[j];
                        t->elem[j]=t->elem[k];
                        t->elem[k]=el;
                      };
              k++;
            };
         i++;
         j--;
       };
 }



 ************************************************************************/
/* determine si le tableau d'entiers A est inclu dans le tableau B.       */
/* Si oui, retourne une valeur >0 ;                                       */
 * sinon retourne une valeur =0.                                         */
 ************************************************************************/

 nt inclusion(a,b)

struct tabint *a,*b;

  int i,j,trouve;

  trouve=1;
  i=0;
  j=0;

  while ((trouve==1)&&(i<a->l))
    { if (a->elem[i]==b->elem[j])
        { i++;
          j++;
        }
       else if ((a->elem[i]>b->elem[j])&&(j<b->l-1))
                    j++;
            else trouve=0;
    };

  return(trouve);


/************************************************************************/
 * recherche d'un element dans un tableau d'entiers non trie.            */
```

```
/* fonction retourne -1.                                                    */
/**************************************************************************/

int pres_int(t,ent)

struct tabint *t;
int ent;

{int i;

  for (i=0;i<t->l;i++)
     { if (t->elem[i]==ent)
         return(i);
     };

  return(-1);
};
```

```
/*****************************************************************/
/*                                                               */
/*                MODULE DES SCHEMAS DE RELATIONS.               */
/*                                                               */
/*****************************************************************/
/*                                                               */
/*                                                               */
/*                                                               */
/*  ce module regroupe les fonctions de gestion et de            */
/*  representation des schemas de relation.                      */
/*                                                               */
/*                                                               */
/*****************************************************************/



#include <stdio.h>
#include "declarations.h"



/*****************************************************************/
/* ouverture de la table des schemas de relations et chargement, */
/* dans cette table, du contenu du fichier relations.file.       */
/*****************************************************************/


ouvrir_rel ()

{ FILE *pf_rel;
  extern struct relat *sch_rel;
  int i,count;

/* initialisations des pointeurs */

 sch_rel=(struct relat*)malloc(sizeof(struct relat));
 sch_rel->ptr_rel[0]=(struct rel *)malloc(sizeof(struct rel));
 sch_rel->ptr_rel[0]->nom=(char *)malloc(30);
 sch_rel->l=0;

 pf_rel=fopen("relations.file","r");

 /* si le fichier relations.file n'existe pas, on le cree */

 if (pf_rel==NULL)
    { pf_rel=fopen("relations.file","w");
      fclose(pf_rel);
      return(OK);
    };

/* transfert des schemas de relations du fichier relations.file */
/* vers la table des schemas.                                   */

 i=0;
 while (count=fread(sch_rel->ptr_rel[i]->nom,sizeof(char),30,pf_rel)!=0)
       { fread(&sch_rel->ptr_rel[i]->nb_arg,sizeof(int),1,pf_rel);

         sch_rel->ptr_rel[++i]=(struct rel *)malloc(sizeof(struct rel));
         sch_rel->ptr_rel[i]->nom=(char *)malloc(30);
       };

 fclose(pf_rel);

/* le dernier espace alloue n'etant pas utilise, on peut le */
/* liberer.                                                 */
```

```
 free(sch_rel->ptr_rel[i]);

 sch_rel->l=i;
 return(OK);
};

/*************************************************************************/
/* ajouter un schema de relations dans la table des schemas deja   */
/* existants.                                                              */
/* cette fonction retourne l'adresse relative du nouveau schema    */
/* dans cette table.                                                       */
/*************************************************************************/

aj_sch_rel(r)

 struct rel *r;

{extern struct relat *sch_rel;

 sch_rel->ptr_rel[sch_rel->l]=(struct rel *)malloc(sizeof(struct rel));
 sch_rel->ptr_rel[sch_rel->l]->nom=(char *)malloc(30);
 strcpy(sch_rel->ptr_rel[sch_rel->l]->nom,r->nom);
 sch_rel->ptr_rel[sch_rel->l]->nb_arg=r->nb_arg;
 return(sch_rel->l++);
};

/*************************************************************************/
/* verifie si le schema de relation dont le nom est donne en entree*/
/* se trouve deja dans la table des relations.                     */
/* Cette fonction retourne son adresse relative s'il est present,  */
/* ou la valeur "-1" s'il ne s'y trouve pas encore.                */
/*************************************************************************/

pres_sch_rel(nom_rel,nb_arg)

char  *nom_rel;
int nb_arg;

{ extern struct relat *sch_rel;
  int i;

  for (i=0;(i<sch_rel->l)&&(strcmp(sch_rel->ptr_rel[i]->nom,nom_rel));i++);

  if (i!=sch_rel->l)
    { if (nb_arg==sch_rel->ptr_rel[i]->nb_arg)
          return(i);
        else return(-2);
    };
  return(-1);
 };


/*************************************************************************/
/* a partir d'un identifiant de schema de relation, cette fonction */
/* ce schema de relation                                                   */
/*************************************************************************/


struct rel *schema_rel(id_rel)

int id_rel;

{ extern struct relat *sch_rel;
  struct rel *r;
```

```
  r=(struct rel *)malloc(sizeof(struct rel));
  r->nom=(char *)malloc(30);
  strcpy(r->nom,sch_rel->ptr_rel[id_rel]->nom);
  r->nb_arg=sch_rel->ptr_rel[id_rel]->nb_arg;
  return(r);
};


/*************************************************************************/
/* fermeture de la table des schemas de relations et sauvetage de  */
/* son contenu dans le fichier relations.file.                           */
/*************************************************************************/

fermer_rel()

{extern struct relat *sch_rel;
 FILE *pf_rel;

 int i,count;

 pf_rel=fopen("relations.file","w");

 for(i=0;i<sch_rel->l;i++)

       {count=fwrite(sch_rel->ptr_rel[i]->nom,sizeof(char),30,pf_rel);
        count=fwrite(&sch_rel->ptr_rel[i]->nb_arg,sizeof(int),1,pf_rel);
       };

 fclose(pf_rel);
 sch_rel=NULL;

};

/*************************************************************************/
/* affichage du contenu de la table des schemas de relations.      */
/*************************************************************************/

aff_sch_rel ()

 {extern struct relat *sch_rel;

  int i;
  printf("---------------schemas de relations--------------- \n");
  for (i=0;i<sch_rel->l;i++)
      {printf("%s",sch_rel->ptr_rel[i]->nom);
       printf("    ");
       printf("%d \n",sch_rel->ptr_rel[i]->nb_arg);
      };
  printf("\n");
 };


/*************************************************************************/
/*                                                                       */
/*                      VALEURS DE RELATIONS.                            */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
/*                                                                       */
/* gestion et representation des valeurs de relations.                   */
/*                                                                       */
/*                                                                       */
/*                                                                       */
/*************************************************************************/
```

```
/*************************************************************************/
/* ouverture de la table des valeurs de relations et chargement,    */
/* dans cette table, du contenu du fichier valeurs.file.            */
/*************************************************************************/

ouvrir_val ()

{
FILE *pf_val;
extern struct valeurs *val;
int i;

/* initialisations des pointeurs */

 val=(struct valeurs*)malloc(sizeof(struct valeurs));
 val->ptr_val[0]=(char *)malloc(20);
 val->l=0;

 pf_val=fopen("valeurs.file","r");

 /* si le fichier valeurs.file n'existe pas, on le cree */

 if (pf_val==NULL)
    { pf_val=fopen("valeurs.file","w");
      close(pf_val);
      return(OK);
    };

/* transfert des valeurs de relations du fichier valeurs.file    */
/* vers la table des valeurs.                                    */

 i=0;
 while (fread(val->ptr_val[i],sizeof(char),20,pf_val)!=0)
       { val->ptr_val[++i]=(char *)malloc(20);
       };

 fclose(pf_val);

/* le dernier espace alloue n'etant pas utilise, on peut le */
/* liberer.                                                 */

 free(val->ptr_val[i]);

 val->l=i;

 return(OK);
};

/*************************************************************************/
/* ajouter une valeur de relations dans la table des valeurs deja  */
/* existantes.                                                     */
/* cette fonction retourne l'adresse relative de la nouvelle valeur*/
/* dans cette table.                                               */
/*************************************************************************/

aj_val(valeur)

 char *valeur;

{extern struct valeurs *val;

 val->ptr_val[val->l]=(char *)malloc(20);
 strcpy(val->ptr_val[val->l],valeur);
 return(val->l++);
};
```

```
/*************************************************************************/
/* verifie si la valeur en parametre se trouve deja dans la table  */
/* des valeurs.                                                          */
/* cette fonction retourne l'adresse relative de la valeur si elle */
/* se trouve dans la table,                                              */
/* sinon, elle retourne "-1".                                            */
/*************************************************************************/

pres_val(v)

char *v;

 {  extern struct valeurs *val;
    int i;

   for (i=0;i<val->l;i++)
      { if (!(strcmp(val->ptr_val[i],v)))
           return(i);
      };

   return(-1);
 };

/*************************************************************************/
/* etant donne un identifiant de valeur de relation, cette fontion */
/* retourne la chaine de carateres correpondant a cette valeur.    */
/*************************************************************************/

char *nomval(id_val)

int id_val;

{extern struct valeurs *val;
 char *ch;

 ch=(char *)malloc(30);
 strcpy(ch,val->ptr_val[id_val]);
 return(ch);
};


/*************************************************************************/
/* fermeture de la table des valeurs de relations et sauvetage de  */
/* son contenu dans le fichier valeurs.file.                             */
/*************************************************************************/

fermer_val()


{ extern struct valeurs *val;
  FILE *pf_val;
  int i,count;

 pf_val=fopen("valeurs.file","w");

 for(i=0;i<val->l;i++)

        count=fwrite(val->ptr_val[i],sizeof(char),20,pf_val);

 fclose(pf_val);
 val=NULL;
};

/*************************************************************************/
/* affichage du contenu de la table des valeurs de relations.      */
```

```
/*************************************************************************/

aff_val ()


 {extern struct valeurs *val;
  int i;

  printf("----------------valeurs de relations---------------- \n");
  for (i=0;i<val->l;i++)
       printf("%s \n",val->ptr_val[i]);
  printf("\n");
 };

/*************************************************************************/
/* analyse une relation sous forme de chaine de caracteres et la   */
/* transforme en objet de type rel_ext.                            */
/*************************************************************************/

struct rel_ext *anal_rel(arg)

char *arg;

{ struct rel_ext *rx;
  char *ch;
  int i;

  rx=(struct rel_ext *)malloc(sizeof(struct rel_ext));

  rx->r=(char *)malloc(30);
  ch=rx->r;
  i=0;
  while (( *arg!='(')&&(*arg!='\0'))
     { *ch = *arg;
        ch++;
        arg++;
     };

  if (*arg=='\0')
    {rx->arg[0]=(char *)malloc(30);
     *rx->arg[0]= *arg;
     rx->nb_arg=0;
     return(rx);
    };

  *ch='\0';
  arg++;



  while (*arg!=')')
    { rx->arg[i]=(char *)malloc(30);
      ch=rx->arg[i];

      while ((*arg!=',')&&(*arg!=')'))
        { *ch = *arg;
          arg++;
          ch++;
        };
      if (*arg!=')')
            arg++;
      i++;
    };

  rx->nb_arg=i;
```

```
  return(rx);
};
```

```
/*****************************************************************************/
/*                        MODULE WORKING MEMORY.                             */
/*****************************************************************************/
/*                                                                           */
/*                                                                           */
/* module de gestion et de representation de la memoire de travail.          */
/*                                                                           */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include "declarations.h"


extern ouvrir_rel();
extern fermer_rel();
extern int aj_sch_rel();
extern int pres_sch_rel();
extern struct rel *schema_rel();

extern ouvrir_val();
extern int aj_val();
extern int pres_val();
extern char *nomval();
extern fermer_val();

/*****************************************************************************/
/* trace                                                                     */
/*****************************************************************************/

struct l_id_el *trc_ac_act(c,n)

int c,n;

{ struct l_id_el *li;

 li=(struct l_id_el *)malloc(sizeof(struct l_id_el));
 li->l=0;
 return(li);
};


/*****************************************************************************/
/* ouverture de la memoire de travail.                                       */
/*****************************************************************************/

ouvrir_wm()

{ extern struct l_el_wm *wm;
  extern struct valeurs *val;
  extern struct relat *sch_rel;

  wm=(struct l_el_wm *)malloc(sizeof(struct l_el_wm));
  wm->l=0;
  wm->elem[0]=(struct el_wm *)malloc(sizeof(struct el_wm));

  if (sch_rel==NULL)
        ouvrir_rel();

  if (val==NULL)
        ouvrir_val();

};
```

```
/*****************************************************************************/
/* ajoute un element, representation interne, a la memoire de travail.  */
/* Cette fonction retourne l'adresse relative de ce nouvel element dans */
/* la table representant la memoire de travail.                          */
/*****************************************************************************/

aj_wm(el)

struct el_wm *el;

{extern struct l_el_wm *wm;
 extern int cycle;
 int i;

 wm->elem[wm->l]=(struct el_wm *)malloc(sizeof(struct el_wm));

 wm->elem[wm->l]->age=cycle;
 wm->elem[wm->l]->id_rel=el->id_rel;
 wm->elem[wm->l]->n_arg=el->n_arg;
 for (i=0;i<el->n_arg;wm->elem[wm->l]->arg[i]=el->arg[i++]);

 return(wm->l++);
};

/*****************************************************************************/
/* ajoute un element, representation externe, dans la memoire de travail*/
/* Cette fonction retourne l'adresse relative de ce nouvel element.      */
/*                                                                       */
/* La seule difference entre cette fonction et la precedente est que la */
/* premiere travaille directement avec une representation interne d'un  */
/* element (avec des identifiants plutot que des valeurs), tandis que la*/
/* seconde travaille avec une representation externe d'un enonce ( avec */
/* des valeurs plutot que des identifiants.                              */
/*****************************************************************************/

int aj_wm_ext(el_ext)

struct el_externe *el_ext;

{ extern struct l_el_wm *wm;
  int i;
  struct el_wm *el;

  el=(struct el_wm *)malloc(sizeof(struct el_wm));

  /* recherche de l'identifiant de la relation. */

  if ((el->id_rel=pres_sch_rel(el_ext->r->nom,el_ext->r->nb_arg))==-1)
       el->id_rel=aj_sch_rel(el_ext->r);

    else if (el->id_rel==-2)     /* nbre d'arg incompatibles...*/
           return(-1);

  /* recherche de l'identifiant des arguments de la rel.*/

  for (i=0;i<el_ext->r->nb_arg;i++)
     { if ((el->arg[i]=pres_val(el_ext->arg[i]))==-1)
            el->arg[i]=aj_val(el_ext->arg[i]);
     };
  el->n_arg=i;

  return(aj_wm(el));
};

/*****************************************************************************/
```

```
/* acces a tous les elements de la memoire de travail ayant comme ident. */
/* de relation celui fourni en parametre.                                 */
/* Cette fonction retourne un pointeur vers la liste des elements de      */
/* memoire de travail ayant cet identificateur de relation.               */
/* Si cette liste est vide, elle retourne un pointeur NULL.               */
/**************************************************************************/

struct l_el_wm *ac_wm(id_rel)

 int id_rel;

{ extern struct l_el_wm *wm;
  extern struct relat *sch_rel;
  struct l_el_wm *l_el;
  int i,j;


  l_el=(struct l_el_wm *)malloc(sizeof(struct l_el_wm));
  l_el->l=0;

  i=sch_rel->l-1;

  if (id_rel>i)
    { free(l_el);
      l_el=NULL;
      return(l_el);
    };

  j=0;

  for (i=0;i<wm->l;i++)
     { if (wm->elem[i]->id_rel==id_rel)
           l_el->elem[j++]=wm->elem[i];
     };
  l_el->l=j;

  if (l_el->l==0)
   { free(l_el);
     l_el=NULL;
   };

  return(l_el);
};


/**************************************************************************/
/* acces aux elements de la memoire de travail les plus recents, i.e.     */
/* ceux qui viennent d'etre ajoutes au cycle precedent.                   */
/**************************************************************************/

struct l_id_el *ac_nouv_el_wm()

{ struct l_id_el *t;
  extern struct l_el_wm *wm;
  extern int cycle;
  int el,c;

  c=cycle-1;
  t=(struct l_id_el *)malloc(sizeof(struct l_id_el));
  t->l=0;

  for (el=0;el<wm->l;el++)
     if (wm->elem[el]->age >= c)
       t->elem[t->l++]=el;

  if (t->l==0)
```

```
    free(t);
      t=NULL;
    };

  return(t);
};

/*************************************************************************/
/* retourne le type de l'element de la memoire de travail el             */
/*************************************************************************/

 int id_classe(el)

 int el;

 { extern struct l_el_wm *wm;

   return(wm->elem[el]->id_rel);

 };


/*************************************************************************/
/* acces aux elements de la memoire de travail dont un argument a la     */
/* valeur donnee en parame`tre.                                          */
/* Si structure=vrai, alors la valeur donnee en parametre est a          */
/* considerer comme la structure imbriquee dont elle est la racine.      */
/* Tous les elements portant sur chaque membre de cette structure seront*/
/* donc selectione's.                                                    */
/* Si structure=faux, il s'agit de la "feuille" de nom valeur.           */
/*************************************************************************/

struct l_el_wm *ac_wm_val(v)

int v;

{ extern struct l_el_wm *wm;
  extern struct valeurs *val;
  struct l_el_wm *l_el;
  int i, j, k;


  j=0;
  l_el=(struct l_el_wm *)malloc(sizeof(struct l_el_wm));
  l_el->l=0;

  i=val->l-1;

  if (v>i)
      return(l_el);

  for (i=0; i<wm->l; i++)
     { for (k=0; k<wm->elem[i]->n_arg; k++)
          { if (wm->elem[i]->arg[k]==v)
                  l_el->elem[j++]=wm->elem[i];
          };
     };

  l_el->l=j;

  return(l_el);
};

/*************************************************************************/
/* verifie si un element de la memoire de travail se trouve dans une     */
/* liste non triee.                                                      */
```

```
/* Si oui, retourne son adresse relative dans cette liste, sinon la     */
/* fonction retourne -1.                                                */
/************************************************************************/

int pres_el_wm(list,el)

struct l_el_wm *list;
struct el_wm *el;

{ int i;

  for (i=0;i<list->l;i++)
    /* les elements sont les memes s'ils ont meme adresse. */
     if (list->elem[i]==el)
       return(i);

  return(-1);
};


/************************************************************************/
/* retourne une liste des identificateurs des valeurs qui sont elements */
/* de la structure dont 'v' est la racine.                              */
/************************************************************************/

struct l_id_val *structure(v)

int v;                                  /* id de la racine */

{ extern struct l_el_wm *wm;
  static struct l_id_val *liv;
  int i,j,l;

  if (liv==NULL)
   {  liv=(struct l_id_val *)malloc(sizeof(struct l_id_val));
      liv->l=0;
   };

  for (i=0;i<wm->l;i++)

    { for (l=0;l<wm->elem[i]->n_arg;l++)
        /* si l'element courant porte sur le terminal v */
        { if (wm->elem[i]->arg[l]==v)
            /* tous les arguments suivants sont a prendre...*/
           { l++;
             while (l<wm->elem[i]->n_arg)
                 /* s'il n'est pas encore dedans */
                { if (pres_int(liv,wm->elem[i]->arg[l])==-1)
                    /* on le rajoute */
                    liv->elem[liv->l++]=wm->elem[i]->arg[l];
                  /* analyse des sous_struct eventuelles...*/
                   liv=structure(wm->elem[i]->arg[l]);
                   l++;
                }
           }
        }
    };

  return(liv);
};


/************************************************************************/
/* mise a jour de la memoire de travail.                                */
/* de la memoire de travail, on extrait tous les elements portant sur v,*/
/* et sur les valeurs membres de sa sous_structure si 'str'=VRAI.       */
```

```
/* La memoire de travail est inchangee.                                    */
/*************************************************************************/

struct l_el_wm *maj_wm(str,v)

int str;                        /* booleen */
int v;                          /* identificateur du terminal */

{ struct l_el_wm *ls1,*ls2;
  struct l_id_val *liv;
  int i,j;

  ls1=ac_wm_val(v);

  /* si ce n'est que le terminal qui nous interesse, it's over...*/

  if (str==FAUX)
     return(ls1);

  /* sinon, on s'occupe de la structure.*/

  liv=structure(v);

  for (i=0;i<liv->l;i++)
     { ls2=ac_wm_val(liv->elem[i]);
       for (j=0;j<ls2->l;j++)
          if (pres_el_wm(ls1,ls2->elem[j])==-1)
              ls1->elem[ls1->l++]=ls2->elem[j];
     };

  return(ls1);
};



/*************************************************************************/
/* affiche a l'ecran le contenu de la liste d'elements de wm             */
/*************************************************************************/

aff_l_el_wm(fich,liste_el)

FILE *fich;
struct l_el_wm *liste_el;

{
  struct el_externe *el_ext;
  int i,j;

  el_ext=(struct el_externe *)malloc(sizeof(struct el_externe));

  for (i=0;i<liste_el->l;i++)
     { el_ext->r=schema_rel(liste_el->elem[i]->id_rel);
       fprintf(fich,"%s %c",el_ext->r->nom,'(');
       for (j=0;j<liste_el->elem[i]->n_arg;j++)
          { el_ext->arg[0]=nomval(liste_el->elem[i]->arg[j]);
            fprintf(fich,"%s",el_ext->arg[0]);
            if (j!=liste_el->elem[i]->n_arg - 1)
                    fprintf(fich,"%c",',');
          };
       fprintf(fich,"%c",')');
       fprintf(fich,"  %d \n",liste_el->elem[i]->age);
     }
};
```

```
/*****************************************************************************/
/* affiche a l'ecran le contenu de la memoire de travail                     */
/*****************************************************************************/

aff_wm(fich)

FILE *fich;

{ extern struct l_el_wm *wm;
  struct el_externe *el_ext;
  int i,j;

  el_ext=(struct el_externe *)malloc(sizeof(struct el_externe));
  fprintf(fich,"---------------memoire de travail--------------- \n");
  for (i=0;i<wm->l;i++)
     { el_ext->r=schema_rel(wm->elem[i]->id_rel);
       fprintf(fich,"%s %c",el_ext->r->nom,'(');
       for (j=0;j<wm->elem[i]->n_arg;j++)
          { el_ext->arg[0]=nomval(wm->elem[i]->arg[j]);
            fprintf(fich,"%s",el_ext->arg[0]);
            if (j!=wm->elem[i]->n_arg - 1)
                    fprintf(fich,"%c",',');
          };
       fprintf(fich,"%c",')');
       fprintf(fich,"  %d \n",wm->elem[i]->age);
     }
};

/*****************************************************************************/
/* lit au terminal une instance de relation, representation externe, et */
/* l'ajoute dans la memoire de travail.                                      */
/*****************************************************************************/

int_el_wm()


{extern struct l_el_wm *wm;

 struct el_externe *el_ext;
 int i;

  el_ext=(struct el_externe *)malloc(sizeof(struct el_externe));
  el_ext->r=(struct rel *)malloc(sizeof(struct rel));
  el_ext->r->nom=(char *)malloc(30);
  for (i=0;i<5;i++)
         el_ext->arg[i]=(char *)malloc(30);

  printf("\nschema > ");
  gets(el_ext->r->nom);

  while (strcmp(el_ext->r->nom,"")!=0)
       { printf("argument > ");
         i=0;
         gets(el_ext->arg[i]);
         while (strcmp(el_ext->arg[i],"")!=0)
              { printf("argument > ");
                gets(el_ext->arg[++i]);
              };
         el_ext->r->nb_arg=i;

         if (aj_wm_ext(el_ext)==-1)
            printf("\n nbre d'arg non valable. recommencez, s.v.p. \n");

         printf("schema > ");
         gets(el_ext->r->nom);
       }
```

```
/*****************************************************************************/
/*                      MODULE BASE DE FAITS .                               */
/*****************************************************************************/
/*                                                                           */
/*                                                                           */
/* module de representation et de gestion des faits.                         */
/*                                                                           */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/


#include <stdio.h>
#include "declarations.h"


/*****************************************************************************/
/*                      FONCTIONS EXTERNES UTILISEES                         */
/*****************************************************************************/


/* module relations.c */

extern ouvrir_rel();                /* ouverture de la table des schemas  */
                                    /* de relations.                      */
extern fermer_rel();                /* fermeture de la table des schemas  */
                                    /* de relations.                      */
extern int aj_sch_rel();            /* ajout d'un schema de relation dans */
                                    /* la table des schemas deja existants*/
extern struct rel *schema_rel();    /* retourne le schema de relation dont*/
                                    /* l'identificateur est fourni en     */
                                    /* parametre.                         */
extern int pres_sch_rel();          /* etant donne un nom de schema de    */
                                    /* relation, retourne son adresse     */
                                    /* relative dans la table s'il s'y    */
                                    /* trouve, sinon -1.                  */

extern ouvrir_val();                /* ouverture de la table des valeurs  */
extern fermer_val();                /* fermeture de la table des valeurs  */
extern int aj_val();                /* ajoute une valeur dans la table des*/
                                    /* valeurs.                           */
extern int pres_val();              /* etant donne un nom de valeur,      */
                                    /* retourne son adresse relative dans */
                                    /* la table si elle s'y trouve, sinon */
                                    /* retourne -1.                       */
extern char *nomval();              /* etant donne un identificateur de   */
                                    /* valeur, retourne son nom.          */


/*****************************************************************************/
/* ouverture de la base de faits.                                            */
/*****************************************************************************/

ouv_bf()

{ FILE *pf_bf;                          /* ptr vers le fichier des faits */
  extern struct l_faits *bf;            /* ptr vers la base des faits    */
  extern struct valeurs *val;
  extern struct relat *sch_rel;

  int i;

  if (val==NULL)
```

```
        ouvrir_val();
  if (sch_rel==NULL)
        ouvrir_rel();

  bf=(struct l_faits *)malloc(sizeof(struct l_faits));
  bf->l=0;
  bf->elem[0]=(struct fait *)malloc(sizeof(struct fait));

  pf_bf=fopen("b_faits.file","r");

  /* si le fichier est vide, on le cree... */

  if (pf_bf==NULL)
    { pf_bf=fopen("b_faits.file","w");
      fclose(pf_bf);
      return;
    };
  i=0;

  while (fread(&bf->elem[i]->id_rel,sizeof(int),1,pf_bf)!=0)
       { fread(&bf->elem[i]->arg,sizeof(int),1,pf_bf);
         bf->elem[++i]=(struct fait *)malloc(sizeof(struct fait));
       };

  bf->l=i;
  fclose(pf_bf);
};


/*****************************************************************************/
/*   fermeture de la base de faits.                                          */
/*****************************************************************************/

ferm_bf()

{ FILE *pf_bf;
  extern struct l_faits *bf;
  extern struct valeurs *val;
  extern struct relat *sch_rel;

  int i;

  pf_bf=fopen("b_faits.file","w");

  for (i=0;i<bf->l;i++)
    { fwrite(&bf->elem[i]->id_rel,sizeof(int),1,pf_bf);
      fwrite(&bf->elem[i]->arg,sizeof(int),1,pf_bf);
    };

  fclose(pf_bf);
  if (val!=NULL)
        fermer_val();
  if (sch_rel!=NULL)
        fermer_rel();

};



/*****************************************************************************/
/* ajoute un element, representation interne, a la base de faits.            */
/* Cette fonction retourne l'adresse relative de ce nouvel element dans      */
/* la table representant la base de faits.                                   */
/*****************************************************************************/
```

```
aj_bf(f)

struct fait *f;

{extern struct l_faits *bf;

 bf->elem[bf->l]=(struct fait *)malloc(sizeof(struct fait));

 bf->elem[bf->l]->id_rel=f->id_rel;
 bf->elem[bf->l]->arg=f->arg;
 return(bf->l++);
};


/*************************************************************************/
/* ajoute un element, representation externe, dans la base de faits    l*/
/* Cette fonction retourne l'adresse relative de ce nouvel element.     */
/*                                                                       */
/* La seule difference entre cette fonction et la precedente est que la */
/* premiere travaille directement avec une representation interne d'un  */
/* element (avec des identifiants plutot que des valeurs), tandis que la*/
/* seconde travaille avec une representation externe d'un element (avec */
/* des valeurs plutot que des identifiants.)                             */
/*************************************************************************/

aj_bf_ext(fx)

struct fait_externe *fx;

{ extern struct l_faits *bf;
  struct fait *f;

  f=(struct fait *)malloc(sizeof(struct fait));

  /* recherche de l'identifiant de la relation. */

  if ((f->id_rel=pres_sch_rel(fx->r->nom,1))==-1)
       f->id_rel=aj_sch_rel(fx->r);

  /* recherche de l'identifiant des arguments de la rel.*/

  if ((f->arg=pres_val(fx->arg))==-1)
            f->arg=aj_val(fx->arg);

  return(aj_bf(f));
};

/*************************************************************************/
/* acces a tous les elements de la base de faits ayant comme identifiant*/
/* de relation celui fourni en parametre.                                */
/* Cette fonction retourne un pointeur vers la liste des elements de    */
/* la base de faits   ayant cet identificateur de relation.              */
/* Si cette liste est vide, elle retourne un pointeur NULL.              */
/*************************************************************************/

struct l_faits *ac_bf(id_rel)

 int id_rel;

{ extern struct l_faits *bf;
  extern struct relat *sch_rel;
  struct l_faits *lf;
  int i,j;

  i=sch_rel->l-1;
  if (id_rel>i)
```

```
      return (NULL);

  lf=(struct l_faits *)malloc(sizeof(struct l_faits));
  j=0;

  for (i=0;i<bf->l;i++)
     { if (bf->elem[i]->id_rel==id_rel)
           lf->elem[j++]=bf->elem[i];
     };
  lf->l=j;

  if (lf->l==0)
    { free(lf);
      lf=NULL;
    };

  return(lf);
};

/*****************************************************************************/
/* acces aux elements de la base de faits dont un argument a la          */
/* valeur donnee en parame`tre.                                          */
/* Si structure=vrai, alors la valeur donnee en parametre est a          */
/* considerer comme la structure imbriquee dont elle est la racine.      */
/* Tous les elements portant sur chaque membre de cette structure seront*/
/* donc selectione's.                                                    */
/* Si structure=faux, il s'agit de la "feuille" de nom valeur.           */
/*****************************************************************************/

struct l_faits *ac_bf_val(v)

int v;

{ extern struct l_faits *bf;
  extern struct valeurs *val;
  struct l_faits *lf;
  int i,j;

  i=val->l-1;
  if (v>i)
      return(NULL);

  j=0;
  lf=(struct l_faits *)malloc(sizeof(struct l_faits));

  for (i=0;i<bf->l;i++)
     { if (bf->elem[i]->arg==v)
           lf->elem[j++]=bf->elem[i];
     };

  lf->l=j;
  return(lf);
};


/*****************************************************************************/
/* affiche a l'ecran le contenu de la liste d'elements de bf             */
/*****************************************************************************/

aff_lf(lf)

struct l_faits *lf;

{
  struct fait_externe *fx;
  int i;
```

```
  fx=(struct fait_externe *)malloc(sizeof(struct fait_externe));
  printf("--------------elmts de la liste--------------- \n");
  for (i=0;i<lf->l;i++)
     { fx->r=schema_rel(lf->elem[i]->id_rel);
       printf("%s %c",fx->r->nom,'(');
       fx->arg=nomval(lf->elem[i]->arg);
       printf("%s%c \n",fx->arg,')');
     };
};


/*************************************************************************/
/* affiche a l'ecran le contenu de la base de faits.                     */
/*************************************************************************/

aff_bf(fich)

FILE *fich;

{ extern struct l_faits *lf;
  struct fait_externe *fx;
  int i,count;
  char ch;

  fx=(struct fait_externe *)malloc(sizeof(struct fait_externe));
  fprintf(fich,"----------------base de faits--------------- \n");
  for (i=0,count=0;i<bf->l;i++,count++)
     { fx->r=schema_rel(bf->elem[i]->id_rel);
       fprintf(fich,"id : %d    ",i);
       fprintf(fich,"%s %c",fx->r->nom,'(');
       fx->arg=nomval(bf->elem[i]->arg);
       fprintf(fich,"%s%c \n",fx->arg,')');
       if (fich==stdout)
          if ((count==20)||(i==bf->l-1))
            {printf("\n >");
             gets(&ch);
             count=0;
            }
     }
};

/*************************************************************************/
/* lit au terminal une instance de relation, representation externe, et */
/* l'ajoute dans la base de travail.                                     */
/*************************************************************************/

int_fait()


{extern struct l_faits *bf;

 struct fait_externe *fx;

  fx=(struct fait_externe *)malloc(sizeof(struct fait_externe));
  fx->r=(struct rel *)malloc(sizeof(struct rel));
  fx->r->nom=(char *)malloc(30);
  fx->arg=(char *)malloc(30);
  printf("\nschema> ");
  gets(fx->r->nom);

  printf("argument > ");
  gets(fx->arg);
  fx->r->nb_arg=1;
```

```
  aj_bf_ext(fx);

  free(fx);
};



/*************************************************************************/
/* introduction de faits dans la base de faits.                          */
/*************************************************************************/

intro_faits()

{char *ch;
 int fin = FAUX;

 ch=(char *)malloc(5);

 printf("\n ...introduction des faits...\n\n");

 while (fin==FAUX)
      { int_fait();
        printf("\n introduction d'un autre fait ? (<RET>=non)");
        gets(ch);
        if (strcmp(ch,"")==0)
          fin=VRAI;
      }
 free(ch);
};

/*************************************************************************/
/* suppression d'un fait dans la base de faits.                          */
/*************************************************************************/

suppr_fait(id_fait)

int id_fait;

{extern struct l_faits *bf;
 int i,status;

 if ((bf->l>id_fait)&&(id_fait>=0))
   { free(bf->elem[id_fait]);
     for (i=id_fait+1;i<bf->l;bf->elem[i-1]=bf->elem[i++]);
     bf->l--;
     status=OK;
   }
  else { erreur5();
         status=ERREUR;
       };
  return(status);

};



/*************************************************************************/
/* modification d'un fait de la base de faits.                           */
/*************************************************************************/

modif_fait(id_fait)

int id_fait;

{
 if (suppr_fait(id_fait)==OK)
```

```
    int_fait();
};


/*****************************************************************************/
/* destruction de la base de faits.                                          */
/*****************************************************************************/

delete_bf()

{ extern struct l_faits *bf;

  bf->l=0;
};
```

```
/*****************************************************************************/
/*                        MODULE BASE DE REGLES.                             */
/*****************************************************************************/
/*                                                                           */
/*                                                                           */
/*         module de gestion et de representation des regles.                */
/*                                                                           */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/


#include <stdio.h>
#include "declarations.h"


/*****************************************************************************/
/*                           FONCTIONS EXTERNES UTILISEES                    */
/*****************************************************************************/


/* module 'relations'    */

extern ouvrir_rel();                    /* ouverture de la table des   */
                                        /* schemas de relations        */
extern fermer_rel();                    /* fermeture et sauvetage de la*/
                                        /* table des schemas de relat. */
extern int pres_sch_rel();              /* verifie si une relation est */
                                        /* presente dans la table.     */
extern struct rel *schema_rel();        /* retourne le schema de relat.*/
                                        /* d'apres son identificateur. */
extern struct rel_ext *anal_rel();      /* decomposition d'une chaine  */
                                        /* de caracteres en relation   */
                                        /* externe.                    */
extern ouvrir_val();                    /* ouverture de la table des   */
                                        /* valeurs.                    */

/* module 'outils'       */

extern int search_T();                  /* recherche d'une chaine de   */
                                        /* caracteres dans une table   */
extern char *nom();                     /* retourne la chaine de car.  */
                                        /* d'une table dont l'id. est  */
                                        /* donne.                      */

extern sort();                          /* tri d'une liste d'entiers   */
extern int inclusion();                 /* verifie si une liste        */
                                        /* d'entiers est incluse dans  */
                                        /* une autre.                  */


/*****************************************************************************/
/* ouverture de la table des actions et chargement, dans cette               */
/* table, du contenu du fichier actions.file.                                */
/*****************************************************************************/

ouv_act()

{extern struct tab *T_actions;
 FILE *pf_act;
 int i;

/* initialisations des pointeurs */
```

```
T_actions=(struct tab *)malloc(sizeof(struct tab));
T_actions->val[0]=(char *)malloc(30);
T_actions->l=0;

pf_act=fopen("actions.file","r");

/* si le fichier actions.file n'existe pas, on le cree */

if (pf_act==NULL)
   { pf_act=fopen("actions.file","w");
     close(pf_act);
     return(OK);
   };

/* transfert des actions du fichier actions.file vers la table des     */
/* actions                                                             */

i=0;
while (fread(T_actions->val[i],sizeof(char),30,pf_act)!=0)
      { T_actions->val[++i]=(char *)malloc(30);
      };

fclose(pf_act);

/* le dernier espace alloue n'etant pas utilise, on peut le */
/* liberer.                                                 */

free(T_actions->val[i]);

T_actions->l=i;

return(OK);
};

/*****************************************************************/
/* ajouter une action dans la table des actions deja existantes.  */
/* cette fonction retourne l'adresse relative de la nouvelle action*/
/* dans cette table.                                              */
/*****************************************************************/


aj_act(action)

 char *action;

{extern struct tab *T_actions;

 T_actions->val[T_actions->l]=(char *)malloc(30);
 strcpy(T_actions->val[T_actions->l],action);
 return(T_actions->l++);
};


/*****************************************************************/
/* verifie si l'action en parametre se trouve deja dans la table  */
/* des actions.                                                   */
/* cette fonction retourne l'adresse relative de l'action  si elle */
/* se trouve dans la table,                                       */
/* sinon, elle retourne "-1".                                     */
/*****************************************************************/

pres_act(action)

char *action;
```

```
{  extern struct tab *l_actions;
   int i;

  for (i=0;i<T_actions->l;i++)
     { if (!(strcmp(T_actions->val[i],action)))
          return(i);
     };

  return(-1);
};

/*************************************************************************/
/* fermeture de la table des actions et sauvetage de son contenu   */
/* dans le fichier T_actionseurs.file.                                   */
/*************************************************************************/

fermer_T_actions()


{ extern struct tab *T_actions;
  FILE *pf_act;
  int i,count;

 pf_act=fopen("actions.file","w");

 for(i=0;i<T_actions->l;i++)

        count=fwrite(T_actions->val[i],sizeof(char),30,pf_act);

 fclose(pf_act);

 if (i!=T_actions->l) printf("erreur de sauvetage...\n");
};

/*************************************************************************/
/* affichage du contenu de la table des actions.                         */
/*************************************************************************/

aff_T_actions ()


 {extern struct tab *T_actions;
  int i;

  printf("----------------TABLE DES ACTIONS---------------- \n");
  for (i=0;i<T_actions->l;i++)
       printf("%s \n",T_actions->val[i]);
  printf("\n");
 };


/*************************************************************************/
/* ouverture et chargement de la base de regles a partir du fichier  */
/* b_regles.file, s'il existe. Sinon, la base de regles reste vide.  */
/*************************************************************************/

ouvrir_br ()

{extern struct l_regles *br;
 FILE *pf_rg;
 extern struct tab *T_actions;
 extern struct relat *sch_rel;
 extern struct valeurs *val;

 int i,j,k,count,z;
 char *ch;
```

```
/* initialisation des pointeurs*/

br=(struct l_regles *)malloc(sizeof(struct l_regles));
br->ptr_rg[0]=(struct regle *)malloc(sizeof(struct regle));
br->l=0;

/* ouverture de la table des schemas de relations et de la table*/
/* des actions.                                                 */

if (sch_rel==NULL)
     ouvrir_rel();
if (val==NULL)
     ouvrir_val();

ouv_act();

/* si le fichier n'existe pas, on le cree. */

if ((pf_rg=fopen("b_regles.file","r"))==NULL)
      { pf_rg=fopen("b_regles.file","w");
        fclose(pf_rg);
        return(OK);
      };

/* transfert du contenu du fichier b_regles.file */

j=0;

while (count=fread(&br->ptr_rg[j]->nb_cond,sizeof(int),1,pf_rg)!=0)

    /* transfert des conditions de la regle courante */

    { for (i=0;i<br->ptr_rg[j]->nb_cond;i++)
       { br->ptr_rg[j]->l_cond[i]=(struct cond *)malloc(sizeof(struct
cond));
         fread(&br->ptr_rg[j]->l_cond[i]->id_rel,sizeof(int),1,pf_rg);
         fread(br->ptr_rg[j]->l_cond[i]->l_var,sizeof(int),NMAXARG,pf_rg);
       };

    /* transfert des actions de la regle courante */

      fread(&br->ptr_rg[j]->nb_act,sizeof(int),1,pf_rg);

      for (i=0;i<br->ptr_rg[j]->nb_act;i++)

       { br->ptr_rg[j]->l_act[i]=(struct act *)malloc(sizeof(struct act));
         fread(&br->ptr_rg[j]->l_act[i]->id_act,sizeof(int),1,pf_rg);
         fread(&br->ptr_rg[j]->l_act[i]->l_arg,sizeof(int),1,pf_rg);
         k=br->ptr_rg[j]->l_act[i]->l_arg;
         br->ptr_rg[j]->l_act[i]->arg=(int *)malloc(k);

         fread(br->ptr_rg[j]->l_act[i]->arg,1,k,pf_rg);
       };

    /* transfert des variables de la regle courante */

      fread(&br->ptr_rg[j]->nb_var,sizeof(int),1,pf_rg);

      for (i=0;i<br->ptr_rg[j]->nb_var;i++)
         { br->ptr_rg[j]->l_var[i]=(char *)malloc(30);
           fread(br->ptr_rg[j]->l_var[i],sizeof(char),30,pf_rg);
         };

      br->ptr_rg[++j]=(struct regle *)malloc(sizeof(struct regle));
```

```
      };

 /* le dernier espace alloue n'etant pas utilise, on le libere*/

 free(br->ptr_rg[j]);
 br->l=j;

 return(OK);
};

/*************************************************************************/
/* ajoute la regle (representation interne) passee en parametre dans */
/* la base de regle.                                                     */
/*************************************************************************/

aj_rg(r)

struct regle *r;

{ extern struct l_regles *br;
  int i, j, k;

  br->ptr_rg[br->l++]=r;

};



/*************************************************************************/
/* fermeture de la base de regles et recopiage de son contenu dans le*/
/* fichier b_regles.file.                                                */
/*************************************************************************/

fermer_br()


{ extern struct l_regles *br;
  extern struct tab *T_actions;
  extern struct relat *sch_rel;

  FILE *pf_rg;
  int i, j, k;

  pf_rg=fopen("b_regles.file", "w");

  /* pour chaque regle de la base, transfert de ses conditions, de */
  /* ses actions, puis de ses variables.                           */

  for (i=0; i<br->l; i++)
   { fwrite(&br->ptr_rg[i]->nb_cond, sizeof(int), 1, pf_rg);

     /* transfert des conditions de la regle courante */

     for (j=0; j<br->ptr_rg[i]->nb_cond; j++)
      { fwrite(&br->ptr_rg[i]->l_cond[j]->id_rel, sizeof(int), 1, pf_rg);
        fwrite(br->ptr_rg[i]->l_cond[j]->l_var, sizeof(int), NMAXARG, pf_rg);
      };

     /* transfert des actions de la regle courante */

     fwrite(&br->ptr_rg[i]->nb_act, sizeof(int), 1, pf_rg);

     for (j=0; j<br->ptr_rg[i]->nb_act; j++)
      { fwrite(&br->ptr_rg[i]->l_act[j]->id_act, sizeof(int), 1, pf_rg);
        fwrite(&br->ptr_rg[i]->l_act[j]->l_arg, sizeof(int), 1, pf_rg);
        k=br->ptr_rg[i]->l_act[j]->l_arg;
```

```
        fwrite(br->ptr_rg[i]->l_act[j]->arg,1,k,pf_rg);
      };

    /* transfert des variables de la regle courante */

    fwrite(&br->ptr_rg[i]->nb_var,sizeof(int),1,pf_rg);
    for (j=0;j<br->ptr_rg[i]->nb_var;j++)
        fwrite(br->ptr_rg[i]->l_var[j],sizeof(char),30,pf_rg);

   };

  fclose(pf_rg);

  /* fermeture de la table des schemas de relations et de la */
  /* table des actions.                                      */

  if (sch_rel!=NULL)
          fermer_rel();
  if (val!=NULL)
          fermer_val();

  fermer_T_actions();

};

/*************************************************************************/
/* transforme la condition sous forme externe passee en parametre en */
/* forme interne, et la rajoute a la regle donnee a l'endroit indice */
/* par "ind"                                                         */
/*************************************************************************/


tr_cdt_xi (cx,ri,ind)

struct cond_externe *cx;
struct regle *ri;
int ind;

{ struct rel *r;
  int i,j,k;

  ri->l_cond[ind]=(struct cond *)malloc(sizeof(struct cond));

  if ((ri->l_cond[ind]->id_rel=pres_sch_rel(cx->nom,cx->nb_arg))==-1)
    { r=(struct rel *)malloc(sizeof(struct rel));
      r->nom=cx->nom;
      r->nb_arg=cx->nb_arg;
      ri->l_cond[ind]->id_rel=aj_sch_rel(r);
    };

  if (ri->l_cond[ind]->id_rel==-2)
    return(ECHEC);

  /* pour chaque variable de la condition, recherche de cette */
  /* variable dans la liste des variables de la regles.       */
  /* si elle n'y est pas, on l'ajoute                         */

  for (i=0;i<cx->nb_arg;i++)
     { if ((k=search_T(ri->l_var,ri->nb_var,cx->tabvar[i]))==-1)
         { ri->l_var[ri->nb_var]=(char *)malloc(30);
           strcpy(ri->l_var[ri->nb_var],cx->tabvar[i]);
           k=ri->nb_var++;
         };
       ri->l_cond[ind]->l_var[i]=k;
     };
};
```

```
/*****************************************************************/
/* transforme la condition d'identificateur 'id' de la regle     */
/* referencee par 'ri', forme interne, sous une forme externe.   */
/* (operation inverse de 'tr_cdt_xi').                           */
/*****************************************************************/

struct cond_externe *tr_cdt_ix(ri,id)

struct regle *ri;
int id;

{ struct rel *r;
  struct cond_externe *cx;
  int i;

  cx=(struct cond_externe *)malloc(sizeof(struct cond_externe));

  r=schema_rel(ri->l_cond[id]->id_rel);

  cx->nom=r->nom;
  cx->nb_arg=r->nb_arg;

  for (i=0;i<r->nb_arg;i++)
      cx->tabvar[i]=nom(ri->l_var,ri->l_cond[id]->l_var[i]);

  return(cx);
};


/*****************************************************************/
/* determine si un ordre d'interpretation porte sur une structure ou */
/* un terminal de l'enonce en entree.                            */
/*****************************************************************/

int interp_strc(arg)

char **arg;

{ char *ch;

  ch = *arg;

  if ( *ch =='s')
    { ch++;
      if ( *ch == '/')
        { ch++;
          *arg = ch;
          return(1);
        };
    };

  return(0);
};


/*****************************************************************/
/* identifie le type d'argument d'une action d'ajout d'une partie */
/* d'interpretation, cad determine s'il s'agit d'une variable ou */
/* d'une constante.                                              */
/*****************************************************************/

int ident_arg(arg)

char **arg;
```

```
{ int t_arg;
  char *str;

  str = *arg;

  while (*str==' ') str++;

  if (*str=='"')
     { t_arg=CONST;
       *arg = ++str;
     }
   else  t_arg=VARIABLE;

  return(t_arg);
}


/*************************************************************************/
/* transforme l'action, forme externe, en representation interne, et */
/* la rajoute a la liste des actions de la regle, forme interne, a   */
/* l'adresse relative 'ind'.                                         */
/*************************************************************************/

tr_act_xi(ax,ri,ind)

struct act_externe *ax;
struct regle *ri;
int ind;

{ extern struct tab *T_actions;
  struct rel_ext *rx;
  struct rel *r;
  int t_arg,k,l;

  ri->l_act[ind]=(struct act *)malloc(sizeof(struct act));
  ri->l_act[ind]->id_act=search_T(T_actions->val,T_actions->l,ax->nom_act);

  if (ri->l_act[ind]->id_act==-1)
    return(ECHEC);

  switch (ri->l_act[ind]->id_act) {

    case 0:
      ri->l_act[ind]->l_arg=sizeof(struct arg_interp);
      ri->l_act[ind]->arg=(struct arg_interp *)malloc(sizeof(struct arg_int
      ri->l_act[ind]->arg->strc=interp_strc(&ax->arg);
      if ((k=search_T(ri->l_var,ri->nb_var,ax->arg))!=-1)
          ri->l_act[ind]->arg->var=k;
      else return(ECHEC);
      break;

    case 1:

    case 2:
      ri->l_act[ind]->l_arg=sizeof(struct arg_wm);
      ri->l_act[ind]->arg=(struct arg_wm *)malloc(sizeof(struct arg_wm));
      rx=anal_rel(ax->arg);

      if ((ri->l_act[ind]->arg->id_rel=pres_sch_rel(rx->r,rx->nb_arg))==-1)
        { r=(struct rel *)malloc(sizeof(struct rel));
          r->nom=rx->r;
          r->nb_arg=rx->nb_arg;
          ri->l_act[ind]->arg->id_rel=aj_sch_rel(r);
        };

      if (ri->l_act[ind]->arg->id_rel==-2)
```

```
        return(ECHEC);

      for (k=0;k<rx->nb_arg;k++)
        { if ((l=search_T(ri->l_var,ri->nb_var,rx->arg[k]))==-1)
           { if ((l=pres_val(rx->arg[k]))==-1)
               l=aj_val(rx->arg[k]);
           };

          ri->l_act[ind]->arg->argrel[k]=l;
        };

        ri->l_act[ind]->arg->nb_arg=rx->nb_arg;
        break;

      case 3:
        ri->l_act[ind]->arg=(struct arg_in *)malloc(sizeof(struct arg_in));
        t_arg=ident_arg(&ax->arg);

        if (t_arg==VARIABLE)
          { ri->l_act[ind]->l_arg=(sizeof(struct arg_in))+(sizeof(int));
            ri->l_act[ind]->arg->t_arg=VARIABLE;

            if ((l=search_T(ri->l_var,ri->nb_var,ax->arg))==-1)
              return(ECHEC);
            ri->l_act[ind]->arg->uval.v=l;
          }
         else { ri->l_act[ind]->l_arg=(sizeof(struct arg_in))+(strlen(ax->ar
                ri->l_act[ind]->arg->t_arg=CONST;
                strcpy(ri->l_act[ind]->arg->uval.const,ax->arg);
              }
         break;

      case 4:
        ri->l_act[ind]->l_arg=0;
        break;

      case 5:

       ri->l_act[ind]->arg=(struct arg_ellipse *)malloc(sizeof(struct arg_el
       t_arg=ident_arg(&ax->arg);

       if (t_arg==VARIABLE)
         { ri->l_act[ind]->l_arg=(sizeof(struct arg_ellipse)+sizeof(int));
           ri->l_act[ind]->arg->t_arg=VARIABLE;

           if ((k=search_T(ri->l_var,ri->nb_var,ax->arg))!=-1)
               ri->l_act[ind]->arg->uvaleur.v=k;
            else return(ECHEC);
         }
        else { ri->l_act[ind]->l_arg=(sizeof(struct arg_ellipse)+(strlen(ax-
               ri->l_act[ind]->arg->t_arg=CONST;
               strcpy(ri->l_act[ind]->arg->uvaleur.const,ax->arg);
             }
    }

};


/*****************************************************************/
/* transforme l'action d'identificateur 'id' de la regle referencee  */
/* par 'ri', forme interne, sous une forme externe.                  */
/* (operation inverse de tr_act_xi.).                                */
/*****************************************************************/

struct act_externe *tr_act_ix(ri,id)
```

```
struct regle *ri;
int id;

{ struct act_externe *ax;
  char *ch,*ch2,*ch3;
  struct rel *r;
  int i;

  ax=(struct act_externe *)malloc(sizeof(struct act_externe));

  ax->nom_act=nom(T_actions->val,ri->l_act[id]->id_act);
  ax->arg=(char *)malloc(30);

  switch (ri->l_act[id]->id_act) {

    case 0:
      ch=ax->arg;
      if (ri->l_act[id]->arg->strc!=0)
        { *ch='s';
          ch++;
          *ch='/';
          ch++;
        };
      ch2=nom(ri->l_var,ri->l_act[id]->arg->var);
      strcpy(ch,ch2);
      break;

    case 1:

    case 2:
      ch=ax->arg;

      r=schema_rel(ri->l_act[id]->arg->id_rel);
      while (( *ch++ = *r->nom++) !='\0');
      *--ch='(';
      ch++;

      for (i=0;i<r->nb_arg;i++)
        { if (ri->l_act[id]->arg->argrel[i]<ri->nb_var) /* si c'est une var
             ch2=nom(ri->l_var,ri->l_act[id]->arg->argrel[i]);
           else ch2=nomval(ri->l_act[id]->arg->argrel[i]);

          while (( *ch++ = *ch2++) !='\0');

          if (i!=r->nb_arg-1)
            { *--ch=',';
              ch++;
            }
        };
      if (i==0)
          *ch=')';
        else *--ch=')';
      break;

    case 3:
      if (ri->l_act[id]->arg->t_arg==VARIABLE)
            ax->arg=nom(ri->l_var,ri->l_act[id]->arg->uval.v);
       else { ch=ax->arg;
              *ch++='"';
              strcpy(ch,ri->l_act[id]->arg->uval.const);
            };
      break;

    case 4:
      strcpy(ax->arg,"");                  /* pas d'argument a cette act*/
      break;
```

```
    case 5:
      if (ri->l_act[id]->arg->t_arg==VARIABLE)
            ax->arg=nom(ri->l_var,ri->l_act[id]->arg->uvaleur.v);
       else { ch=ax->arg;
              *ch++='"';
              strcpy(ch,ri->l_act[id]->arg->uvaleur.const);
            };

   };

  return(ax);
};



/*************************************************************************/
/* affiche au terminal la condition cx, forme externe.                   */
/*************************************************************************/

aff_cdt(fich,cx)

FILE *fich;
struct cond_externe *cx;

{ int i;

  fprintf(fich,"%s",cx->nom);
  fprintf(fich,"%c",'(');

  for (i=0;i<cx->nb_arg;i++)
     { fprintf(fich,"%s",cx->tabvar[i]);
       if (i!=cx->nb_arg-1)
         fprintf(fich,"%c",',');
     };
  fprintf(fich,"%c",')');
};

/*************************************************************************/
/* affiche au terminal l'action 'ax', forme externe.                     */
/*************************************************************************/

aff_act(fich,ax)

FILE *fich;
struct act_externe *ax;

{ fprintf(fich,"%s",ax->nom_act);
  fprintf(fich,"%c",'(');
  fprintf(fich,"%s",ax->arg);
  fprintf(fich,"%c",')');
};

/*************************************************************************/
/* affiche a l'ecran le contenu de la base de regles.                    */
/*************************************************************************/

aff_br(fich)

FILE *fich;

{ extern struct l_regles *br;
  struct cond_externe *cx;
  struct act_externe *ax;
  int i,j;
  char *c;
```

```
  c=(char *)malloc(10);

  fprintf(fich,"\n\n---------------BASE DE REGLES--------------- \n\n\n\n");


  for (i=0;i<br->l;i++)
     { fprintf(fich,"identificateur de regle : %d \n\n",i);
       fprintf(fich,"    ");

       /* affichage des conditions */

       for (j=0;j<br->ptr_rg[i]->nb_cond;j++)
          { cx=tr_cdt_ix(br->ptr_rg[i],j);
            aff_cdt(fich,cx);
            if (j!=br->ptr_rg[i]->nb_cond-1)
                 fprintf(fich,"\n & ");
          };

       fprintf(fich,"\n\n => ");

       /* affichage des actions */

       for (j=0;j<br->ptr_rg[i]->nb_act;j++)
          { ax=tr_act_ix(br->ptr_rg[i],j);
            aff_act(fich,ax);
            if (j!=br->ptr_rg[i]->nb_act-1)
              fprintf(fich," \n  & ");
          };
       fprintf(fich,"\n------------------------------\n \n");

       if (fich==stdout)
           { fprintf(fich," next > ");
             gets(c);
           };
       fprintf(fich,"    ");
     }
};

/*************************************************************************/
/* acces a la premisse d'une regle dont l'identifiant est donne          */
/*************************************************************************/

struct ls_cond *premisse(r)

int r;

{ extern struct l_regles *br;
  struct ls_cond *lc;
  int i;

  lc=(struct ls_cond *)malloc(sizeof(struct ls_cond));
  lc->l=br->ptr_rg[r]->nb_cond;

  for (i=0;i<lc->l;lc->el[i]=br->ptr_rg[r]->l_cond[i++]);

  return(lc);
};

/*************************************************************************/
/* acces aux actions d'une regle dont l'identifiant est donne.           */
/*************************************************************************/

struct ls_act *consequent(r)

int r;
```

```
{ extern struct l_regles *br;
  int i;
  struct ls_act *la;

  la=(struct ls_act *)malloc(sizeof(struct ls_act));
  la->l=br->ptr_rg[r]->nb_act;

  for (i=0;i<la->l;la->el[i]=br->ptr_rg[r]->l_act[i++]);

  return (la);

};

/*************************************************************************/
/* acces a toutes les regles ayant une condition donnee.                 */
/*************************************************************************/

struct l_id_regles *acces_rg_cdts(id_cond)

int id_cond;

{ extern struct l_regles *br;
  struct l_id_regles *lr;
  int i,j;

  lr=(struct l_id_regles *)malloc(sizeof(struct l_id_regles));

  lr->l=0;

  /* pour chaque regle de la base de regles */

  for (i=0;i<br->l;i++)

    /* pour chaque condition de la regle courante */

    { for (j=0;j<br->ptr_rg[i]->nb_cond;j++)
        if (br->ptr_rg[i]->l_cond[j]->id_rel==id_cond)
          { lr->elem[lr->l++]=i;
            break;
          };
    };

  return(lr);
};



/*************************************************************************/
/* extrait d'une liste de conditions une liste des types des differentes*/
/* conditions.                                                           */
/* retourne un pointeur vers cette liste.                                */
/*************************************************************************/

struct tabint * ext_tp_cdts(lc)

struct ls_cond *lc;

{ struct tabint *t;
  int i;

  t=(struct tabint *)malloc(sizeof(struct tabint));
  t->l=0;

  for (i=0;i<lc->l;t->elem[i]=lc->el[i++]->id_rel);
```

```
  t->1=lc->1;

  return(t);
};

/*************************************************************************/
/* determine si la regle r1 est une generalite de la regle r2.           */
/* Si oui, retourne une valeur <0;                                       */
/* si r2 est plus general que r1, une valeur >0;                         */
/* sinon, une valeur =0.                                                 */
/*************************************************************************/

int generalite(r1,r2)

int r1,r2;

{ struct tabint *t1,*t2;
  struct ls_cond *lc1,*lc2;

  int status,id;

  lc1=premisse(r1);
  lc2=premisse(r2);

  if (lc1->1==lc2->1)
    return(0);

  if (lc1->1<lc2->1)
    { id=1;
      t1=ext_tp_cdts(lc1);
      t2=ext_tp_cdts(lc2);
    }
   else { id=2;
          t1=ext_tp_cdts(lc2);
          t2=ext_tp_cdts(lc1);
        };

  sort(t1);
  sort(t2);

  status=inclusion(t1,t2);

  if (status!=0)
      if (id==1)
         return(-1);
       else return(1);

  return(0);
};


/*************************************************************************/
/* compare 2 regles et determine laquelle des 2 a la plus forte          */
/* contrainte.                                                           */
/*************************************************************************/

int max_contrainte (id_r1,id_r2)

int id_r1,id_r2;

{ extern struct l_regles *br;

  return (br->ptr_rg[id_r2]->nb_cond - br->ptr_rg[id_r1]->nb_cond);
};
```

```
/*************************************************************************/
/* retourne le nombre de regles d'une variable.                          */
/*************************************************************************/

 int nombre_var(id)

  int id;

{ return(br->ptr_rg[id]->nb_var);
};


/*************************************************************************/
/* retourne une liste  des varaibles d'une regle                         */
/*************************************************************************/

 struct valeurs *var_regle(id_rg)

 int id_rg;


{ struct valeurs *l;
  int i;

  l=(struct valeurs *)malloc(sizeof(struct valeurs));
  l->l = br->ptr_rg[id_rg]->nb_var;

  for (i=0;i<l->l;l->elem[i]=br->ptr_rg[id_rg]->l_var[i++]);

  return(l);
};


/*************************************************************************/
/* introduction des conditions de la regle courante.                     */
/*************************************************************************/

intro_cond(r)

 struct regle *r;

{ int c,v;
  struct cond_externe *cx;

  /* initialisations. */
  cx=(struct cond_externe *)malloc(sizeof(struct cond_externe));
  cx->nom=(char *)malloc(30);
  for (c=0;c<NMAXVAR;cx->tabvar[c++]=(char *)malloc(30));

  printf("\n introduction des conditions.........\n");
  printf("schema > ");
  gets(cx->nom);
  c=0;

  while (strcmp(cx->nom,""))
       { printf("variable > ");
         gets(cx->tabvar[0]);
         v=0;

         while (strcmp(cx->tabvar[v],""))
              { printf("variable > ");
                gets(cx->tabvar[++v]);
              };

         cx->nb_arg=v;
```

```
        if (tr_cdt_xi(cx,r,c)==ECHEC)
          { printf("\n erreur dans le nbre d'arg. de la relation...\n");
            printf("veuillez recommencer, s.v.p.\n");
          }
         else c++;

        printf("schema > ");
        gets(cx->nom);
      };

  r->nb_cond=c;

  for (c=0;c<NMAXVAR;free(cx->tabvar[c++]));
  free(cx->nom);
  free(cx);
};

/*************************************************************************/
/* introduction des actions de la regle courante.                        */
/*************************************************************************/

intro_act(r)

 struct regle *r;

{ int c;
  struct act_externe *ax;

  /* initialisations */
  ax=(struct act_externe *)malloc(sizeof(struct act_externe));
  ax->nom_act=(char *)malloc(30);
  ax->arg=(char *)malloc(30);

  printf("\n introduction des actions.....\n");
  printf("action >");
  gets(ax->nom_act);
  c=0;

  while (strcmp(ax->nom_act,""))
       { printf("argument > ");
         gets(ax->arg);
         if (tr_act_xi(ax,r,c)==ECHEC)
           { printf("\n erreur dans l'introduction de l'action...\n");
             printf("veuillez recommencer, s.v.p.\n");
           }
          else c++;

         printf("action > ");
         gets(ax->nom_act);
       };

  r->nb_act=c;

  free(ax->arg);
  free(ax->nom_act);
  free(ax);

};


/*************************************************************************/
/* destruction de la base de regles toute entiere.                       */
/*************************************************************************/

delete_br()
```

```
{ extern struct l_regles *br;

  br->l=0;
};


/*************************************************************************/
/* introduction de regles dans la base de regles.                        */
/*************************************************************************/

intro_regles()

{ extern struct l_regles *br;
  char *ch;
  int fin = FAUX;
  struct regle *r = NULL;

  ch=(char *)malloc(5);
/*------*/


  while (fin==FAUX)
    {
      r=(struct regle *)malloc(sizeof(struct regle));
      r->nb_var=0;
      intro_cond(r);
      if (r->nb_cond==0)
         {printf("\n une condition au moins s.v.p....\n");
          printf("\n recommencez...\n");
          continue;
         };
      intro_act(r);
      if (r->nb_act==0)
         {printf("\n une action au moins s.v.p....\n");
          printf("\n recommencez...\n");
          continue;
         };
      aj_rg(r);
      printf("\n introduction d'une autre regle ? (<RET>=non) ");
      gets(ch);
      if (strcmp(ch,"")==0)
        fin=VRAI;
    };

/*  free(r); */

};

/*************************************************************************/
/* supprimer une regle de la base de regle.                              */
/*************************************************************************/


suppr_regle(id_rg)

 int id_rg;

{ extern struct l_regles *br;
  int i,status;

  if ((br->l>id_rg)&&(id_rg>=0))
    { free(br->ptr_rg[id_rg]);
      for (i=id_rg+1;i<br->l;br->ptr_rg[i-1]=br->ptr_rg[i++]);
      br->l--;
      status=OK;
```

```
    }
   else { erreur5();
          status=ERREUR;
        };
  return(status);
};

/*************************************************************************/
/* modification d'une regle de la base (suppression puis insertion)*/
/*************************************************************************/

modif_regle(id_rg)

 int id_rg;

{ extern struct l_regles *br;
  struct regle *r = NULL;

  if (suppr_regle(id_rg)==OK)
    { r=(struct regle *)malloc(sizeof(struct regle));
      r->nb_var=0;

      intro_cond(r);
      while (r->nb_cond==0)
         {printf("\n une condition au moins s.v.p....\n");
          printf("\n recommencez...\n");
          intro_cond(r);
         };
      intro_act(r);
      while (r->nb_act==0)
         {printf("\n une action au moins s.v.p....\n");
          printf("\n recommencez...\n");
          intro_act(r);
         };
      aj_rg(r);
    };
  if (r!=NULL)
      free(r);

};
```

```
/*****************************************************************************/
/*                                                                           */
/*                          MODULE CONFLICT_SET                              */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*                                                                           */
/* contient toutes les fonctions utilisees pour la constitution du          */
/* conflict set.                                                             */
/*                                                                           */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/


#include <stdio.h>
#include "declarations.h"


/******** FONCTIONS EXTERNES UTILISEES *********/


/* -- module base_regles.c -- */

extern struct l_id_el *trc_ac_act();
extern struct l_id_regles *acces_rg_cdts();
extern struct ls_cond *premisse();

/* -- module workmem.c -- */

extern struct l_id_el *ac_nouv_el_wm();
extern struct l_el_wm *ac_wm();
extern int id_classe();

/* -- module base_faits.c -- */


extern struct l_faits *ac_bf();


/* -- module relations.c -- */

extern char *nomval();


/* -- module outils.c -- */

extern char * nom();

/*****************************************************************************/
/* retourne l'identifiant de la condition portant sur l'element dont         */
/* l'identificateur el est donne en parametre.                               */
/*****************************************************************************/

int identif_cond(el)

int el;

{ return(id_classe(el));
};


/*****************************************************************************/
/* verifie si la regle d'identificateur id_rg se trouve dans le              */
/* conflict set. Si oui, retourne son adresse relative, sinon retourne       */
```

```
/* O.                                                                    */
/************************************************************************/

int pres_conset(id_rg)

int id_rg;

{ extern struct l_inst_regles *conset;
  int pres;
  int i;

  pres=0;
  i=0;

  while ((i<conset->l)&&(pres==0))
       pres=(conset->elem[i++]->id_rg!=id_rg) ? 0 : --i;

  return(pres);
};


/************************************************************************/
/* mise-a-jour du conflict set.                                         */
/* cette fonction a pour objectif d'adapter le conflict set du cycle    */
/* precedent en fonction des nouveaux elements en memoire de travail,   */
/* c'est-a-dire d'en extraire toutes les regles dont une condition est  */
/* rendue fausse par l'ajout d'un element en memoire de travail         */
/************************************************************************/

maj_conset()

{ extern struct l_inst_regles *conset;
  extern int cycle;
  struct l_id_el *l_el;
  struct l_id_regles *lr;
  int i, j, k;

  if (conset==NULL)
     {conset=(struct l_inst_regles *)malloc(sizeof(struct l_inst_regles));
      conset->l=0;
     };

  l_el=trc_ac_act(cycle-1, 1);


  if (l_el==NULL)
    { free(l_el);
      l_el=NULL;
      return;
    };

  for (i=0; i<l_el->l; i++)
     { lr=acces_rg_cdts(identif_cond(l_el->elem[i]));

       for (j=0; j<lr->l; j++)
          { if (k=pres_conset(lr->elem[j])!=0)
                sup_conset(k);
          }
     }
};


/************************************************************************/
/* ajoute une instance de regle dans le conflict set.                   */
/************************************************************************/
```

```
ad_conset(ir)

struct inst_regle *ir;

{ extern struct l_inst_regles *conset;

  conset->elem[conset->l++]=ir;
};


/*************************************************************************/
/* copie une liste de valeurs de variables dans une table apres les      */
/* avoir prealablement triees.                                           */
/*************************************************************************/

cp_val_var(t1,t2)

int *t1;
struct sub *t2;

{ int i,j;

  for (i=0;i<t2->l;i++)
     { for (j=0;j<t2->l;j++)
         if (t2->elem[j][0]==i)
             break;

       *t1=t2->elem[j][1];
       t1++;
     };
};


/*************************************************************************/
/* etant donne une liste de val de variables d'une instance de regle     */
/* et une reference a une  variable de cette liste, cette fonction       */
/* renvoie le nom de la valeur de cette variable pour l'instance de      */
/* regle donnee.                                                         */
/*************************************************************************/

int nom_val_var(l_val_var,id)

int l_val_var[NMAXARG];
int id;

{ return (nomval(l_val_var[id]));
}


/*************************************************************************/
/* supprime la regle d'identificateur id_rg du conflict set              */
/*************************************************************************/

sup_conset(id_rg)

int id_rg;

{ extern struct l_inst_regles *conset;
  int i;

  for (i=id_rg;i<conset->l-1;i++)
     conset->elem[i]=conset->elem[i+1];

  conset->l--;
};
```

```
/*****************************************************************************/
/* evalue une condition.                                                     */
/* retourne la liste de substitution de variables par des constantes         */
/* rendant cette condition vraie.                                            */
/*****************************************************************************/

/*struct ls_sub*/ SUB *vrai(cdt)

struct cond *cdt;

{ struct ls_sub *lis;

  struct l_el_wm *lel;
  struct l_faits *lf;
  int i, j;

  lis=(struct ls_sub *)malloc(sizeof(struct ls_sub));
  lis->l=0;

  lel=ac_wm(cdt->id_rel);

  if (lel!=NULL)

    { for (i=0; i<lel->l; i++)
       { lis->elem[lis->l]=(struct sub *)malloc(sizeof(struct sub));

         for (j=0; j<lel->elem[i]->n_arg; j++)
            { lis->elem[lis->l]->elem[j][0]=cdt->l_var[j];
              lis->elem[lis->l]->elem[j][1]=lel->elem[i]->arg[j];
            };

         lis->elem[lis->l]->l=lel->elem[i]->n_arg;

         lis->l++;
       };
       return(lis);
    };

  free(lel);

  lf=ac_bf(cdt->id_rel);

  if (lf==NULL)
    { free(lis);
      lis=NULL;
      return(lis);
    };


  for (i=0; i<lf->l; i++)
         { lis->elem[lis->l]=(struct sub *)malloc(sizeof(struct sub));
           lis->elem[lis->l]->elem[0][0]=cdt->l_var[0];
           lis->elem[lis->l]->elem[0][1]=lf->elem[i]->arg;
           lis->elem[lis->l++]->l=1;
         };


  return(lis);
};

/*****************************************************************************/
/* fusion de 2 jeux de substitution, i.e. creation d'un 3e reprenant         */
/* les 2 autres. Il sera vide s'il y a incompatibilite.                      */
/*****************************************************************************/
```

```
struct sub *fusion(s1,s2)

struct sub *s1,*s2;

{ struct sub *s;
  int i,i1,i2,trouve;

  s=(struct sub *)malloc(sizeof(struct sub));
  s->l=0;

  for (i1=0;i1<s1->l;i1++)
     { for (i2=0;i2<s2->l;i2++)
          { if (s1->elem[i1][0]==s2->elem[i2][0])
              if (s1->elem[i1][1]==s2->elem[i2][1])
                  break;
               else { free(s);
                      s=NULL;
                      return(s);              /* incompatible */
                    }
          };

        s->elem[s->l][0]=s1->elem[i1][0];
        s->elem[s->l++][1]=s1->elem[i1][1];
      };

  for (i2=0;i2<s2->l;i2++)
      { i=0;
        trouve=0;

        while ((trouve==0)&&(i<s->l))
            (s->elem[i][0]==s2->elem[i2][0]) ? trouve=1 : i++;

        if (trouve==0)
          { s->elem[s->l][0]=s2->elem[i2][0];
            s->elem[s->l++][1]=s2->elem[i2][1];
          }
      };
  return(s);
};


/*************************************************************************/
/* intersection de 2 listes de substitutions                             */
/* idem que "fusion" mais maintenant pour 2 listes de substitution.      */
/*************************************************************************/


struct ls_sub *inter(ls1,ls2)

struct ls_sub *ls1,*ls2;

{ struct ls_sub *ls;
  struct sub *sb;
  int s1,s2;

  if ((ls1==NULL)||(ls2==NULL))
       return(NULL);

  ls=(struct ls_sub *)malloc(sizeof(struct ls_sub));
  ls->l=0;

  for (s1=0;s1<ls1->l;s1++)
    { for (s2=0;s2<ls2->l;s2++)
        { sb=fusion(ls1->elem[s1],ls2->elem[s2]);
          if (sb!=NULL)
```

```
                ls->elem[ls->l++]=sb;
            }
        };
    return(ls);
};

/*****************************************************************************/
/* evaluation d'une liste de condition                                       */
/*****************************************************************************/

struct ls_sub *evalue(lc)

struct ls_cond *lc;

{ struct cond *cd;
  struct ls_sub *ls;
  int i;

  if (lc->l==1)
     { ls=vrai(lc->el[0]);
       return(ls);
     }
   else { cd=lc->el[0];
          for (i=0;i<lc->l-1;i++)
               lc->el[i]=lc->el[i+1];
          lc->l--;
          ls=vrai(cd);
          return(inter(ls,evalue(lc)));
        };
};

/*****************************************************************************/
/* teste si la regle rg est activable.                                       */
/* Si oui, retourne la liste des instances de cette regle.                   */
/*****************************************************************************/

struct l_inst_regles *activable(rg)

int rg;

{ struct l_inst_regles *lir;
  struct ls_sub *ls;
  int id_sub;

  lir=(struct l_inst_regles *)malloc(sizeof(struct l_inst_regles));
  lir->l=0;


  ls=evalue(premisse(rg));

  if (ls==NULL)
    { free(lir);
      lir=NULL;
      return(lir);
    };

  for (id_sub=0;id_sub<ls->l;id_sub++)
    { lir->elem[lir->l]=(struct inst_regle *)malloc(sizeof(struct inst_regl
      lir->elem[lir->l]->id_rg=rg;
      cp_val_var(lir->elem[lir->l++]->l_val_var,ls->elem[id_sub]);
    };

  return(lir);
};

/*****************************************************************************/
```

```
/* recherche si une regle est presente dans une liste                    */
/************************************************************************/

int pres_lr(lr,id)

struct l_id_regles *lr;
int id;

{ int i;

  for (i=0;i<lr->l;i++)
    { if (lr->elem[i]==id)
        return(i);
    };

  return(-1);
};


/************************************************************************/
/* extrait de la base de regles celles qui ont une condition portant    */
/* sur l'un des elements rajoutes en memoire de travail au cycle        */
/* precedent.                                                           */
/************************************************************************/

struct l_id_regles *filtre()

{ struct l_id_regles *lr;
  struct l_id_regles *l_rg;
  struct l_id_el *lel;
  int el,id_rg;

  lr=(struct l_id_regles *)malloc(sizeof(struct l_id_regles));
  lr->l=0;

  lel=ac_nouv_el_wm();
  if (lel==NULL)
     return(lr);

  for (el=0;el<lel->l;el++)
    { l_rg=acces_rg_cdts(identif_cond(lel->elem[el]));

      for (id_rg=0;id_rg<l_rg->l;id_rg++)
         { if (pres_lr(lr,l_rg->elem[id_rg])==-1)
             lr->elem[lr->l++]=l_rg->elem[id_rg];
         };

      free(l_rg);
    };

  free(lel);

  return(lr);
};



/************************************************************************/
/* compare 2 instances de regles et determine si elles sont             */
/* equivalentes.                                                        */
/* Si elles sont identiques retourne la valeur 0;                       */
/* si elles sont 2 instances differentes de la meme regle, retourne 1*/
/* si elles n'ont aucun point commun, retourne 2.                       */
/************************************************************************/
```

```
int compare_inst(ir1,ir2)

struct inst_regle *ir1,*ir2;

{ int i;
  int nb_var;

  if (ir1->id_rg!=ir2->id_rg)
    return(2);

  /* vu qu'il s'agit d'instances de la meme regle, */
  /* reste a comparer les valeurs des variables.   */
  nb_var=nombre_var(ir1->id_rg);
  for (i=0;i<nb_var;i++)
     if (strcmp(ir1->l_val_var[i],ir2->l_val_var[i])!=0)
       return(1);

  return(0);
};

/*****************************************************************************/
/* affiche une liste d'instances de regles.                                  */
/*****************************************************************************/

aff_l_inst_rg(fich,lir)

struct l_inst_regles *lir;
FILE *fich;

{ extern struct l_regles *br;
  struct regle *r;
  char *ch,*next;
  int i,j;
  struct valeurs *l;


  next=(char *)malloc(30);

  for (i=0;i<lir->l;i++)
    { fprintf(fich,"\n  id de regle : %d \n \n",lir->elem[i]->id_rg);

      l=var_regle(lir->elem[i]->id_rg);
      if (l->l==0)
        { fprintf(fich,"pas de variables->pas de substitution\n");
          continue;
        };
      fprintf(fich,"substitution : \n");
      for (j=0;j<l->l;j++)
        {
          fprintf(fich,"%s  -> ",l->ptr_val[j]);
          ch=nomval(lir->elem[i]->l_val_var[j]);
          fprintf(fich,"%s \n",ch);
        };
      if (fich==stdout)
        { printf("\n \n next > ");
          gets(next);
        }
    }
};


/*****************************************************************************/
/* initialisation du conflict set                                            */
/*****************************************************************************/

init_conflict_set()
```

```
  { ouvrir_br();
    ouvrir_wm();
    ouv_bf();
  };


/*************************************************************************/
/* operations de clotures du conflict set.                               */
/*************************************************************************/

clot_conflict_set()


 {

   fermer_br();
   ferm_br();

};

/*************************************************************************/
/* selection du conflict set courant.                                    */
/*************************************************************************/

int slct_conflict_set()

{ extern struct l_inst_regles *conset;
  struct l_id_regles *lr = NULL;
  struct l_inst_regles *lir = NULL;
  int id_rg,id_inst;

  maj_conset();

  lr=filtre();

  if (lr->l!=0)

    { for (id_rg=0;id_rg<lr->l;id_rg++)

        { lir=activable(lr->elem[id_rg]);

          if (lir!=NULL)

          for (id_inst=0;id_inst<lir->l;ad_conset(lir->elem[id_inst++]));
        }
    };

  if (lir!=NULL)
       free(lir);

  if (conset->l==0)
     return (VIDE);

  return(OK);
};

/*************************************************************************/
/* elimine dans une liste d'instances de regles, celles qui sont une  */
/* generalisation d'autres.                                              */
/*************************************************************************/

struct l_inst_regles *strat_generalite(lir)

struct l_inst_regles *lir;
```

```
{ struct l_inst_regles *l;
  int i,k;

  l=(struct l_inst_regles *)malloc(sizeof(struct l_inst_regles));
  l->l=0;

  i=0;

  while (i<lir->l)
    { k=0;
      while ((k<l->l)&&(generalite(lir->elem[i]->id_rg,l->elem[k]->id_rg)=
           k++;

      if (k==l->l)
        l->elem[l->l++]=lir->elem[i++];

       else { if (generalite(lir->elem[i]->id_rg,l->elem[k]->id_rg)<0)
                 i++;
               else l->elem[k]=lir->elem[i++];
            };
     };


  return(l);
};


/*************************************************************************/
/* elimine de la liste d'instance de regles celles qui sont moins        */
/* contraignantes que d'autres.                                          */
/* Cette fonction retourne l'adresse de la nouvelle liste.               */
/* (qui ne sera jamais vide...)                                          */
/*************************************************************************/

struct l_inst_regles *strat_contrainte(lir)

struct l_inst_regles *lir;

{ struct l_inst_regles *l;
  struct inst_regle *ir;
  int i,j,k;

  l=(struct l_inst_regles *)malloc(sizeof(struct l_inst_regles));
  l->l=0;

  i=0;
  j=lir->l-1;

  while (i<j)
     { if ((k=max_contrainte(lir->elem[i]->id_rg,lir->elem[j]->id_rg))>0)
           i++;
        else if (k==0)
               { j--;
                 ir=lir->elem[j];
                 lir->elem[j]=lir->elem[i];
                 lir->elem[i]=ir;
               }
              else { j=lir->l-1;
                     ir=lir->elem[j];
                     lir->elem[j]=lir->elem[i];
                     lir->elem[i]=ir;
                     i++;
                   };
      };
```

```
  for (k=j; k<lir->l; l->elem[l->l++]=lir->elem[k++]);

  return(l);
};

/*****************************************************************************/
/* resolution du conflict set.                                               */
/*****************************************************************************/

struct inst_regle *resol_conflict_set()

{ extern struct l_inst_regles *conset;
  struct l_inst_regles *lir,*liir;
  int strategie,fin;
  int i;

  lir=(struct l_inst_regles *)malloc(sizeof(struct l_inst_regles));
  lir->l=0;

  for (i=0;i<conset->l;lir->elem[i]=conset->elem[i++]);
  lir->l=conset->l;

  fin=FAUX;
  strategie=1;

  while (fin==FAUX)

    { switch (strategie) {

        case 1:
          liir=strat_generalite(lir);
          break;

        case 2:
          liir=strat_contrainte(lir);
          break;
     };
/*        case 3: */
/*          liir=strat_age(lir); */
/*      }; */

      if ((liir->l>1)&&(strategie<3))
        { lir=liir;
          strategie++;
        }

      else { fin=VRAI;
             if (liir->l==0)
               liir->elem[0]=lir->elem[0];
           };
    };

  return(liir->elem[0]);
};
```

```
/*                                                                    */
/*              MODULE DE GESTION DE LA TRACE D'EXECUTION.            */
/*                                                                    */
/**********************************************************************/


#include <stdio.h>
#include "declarations.h"


/**********************************************************************/
/*                    FONCTION EXTERNES UTILISEES.                    */
/**********************************************************************/


/* module 'base_regles.c' */

 extern aff_l_inst_rg();
extern int compare_inst();


/* module 'workmem.c' */

 extern aff_wm();
 extern aff_l_el_wm();



/**********************************************************************/
/* creation de la trace, vide.                                        */
 **********************************************************************/

trc_creation()

  extern struct trc_trace *trace;
  extern FILE *trc_pf;
  extern int option;


  trace=(struct trc_trace *)malloc(sizeof(struct trc_trace));
  trace->l=0;

  if (option==OFF_LINE)
       trc_pf=fopen("trace.out","w");
    else trc_pf=stdout;


  fprintf(trc_pf,"\n--------------- TRACE D'EXECUTION ---------------\n\n\n");
  fprintf(trc_pf,"\n\n\n\n");


 **********************************************************************/
/* affichage d'un diagnostic d'execution.                             */
/**********************************************************************/

ff_diag(status)

int status;

{ extern FILE *trc_pf;
  int i;
```

```
        case ERREUR   : fprintf(trc_pf," ERREUR \n");
                        break;
        case ECHEC    : fprintf(trc_pf," ECHEC \n");
                        break;
        case FIN      : fprintf(trc_pf," FIN \n");
                        break;
        case OK       : fprintf(trc_pf," O.K. \n");
                        break;
        case RUNNING  : fprintf(trc_pf," RUNNING \n");
                        break;
        case VIDE     : fprintf(trc_pf," RUNNING \n");
                        break;

    }
};


/*****************************************************************************/
/* mise a jour de la trace d'execution avant execution de la regle           */
 * courante. De cette maniere, on sait a tout moment ou on en est..... */
/*****************************************************************************/


trc_ant_maj(ir)

struct inst_regle *ir;

{ extern struct trc_trace *trace;
  extern int cycle;
  extern FILE *trc_pf;
  struct l_inst_regles *lir;
  int i;

  /* maj de la trace interne du systeme */

  trace->elem[trace->l]=(struct trc_ir *)malloc(sizeof(struct trc_ir));
  trace->elem[trace->l]->cycle=cycle;
  trace->elem[trace->l]->ir.id_rg=ir->id_rg;


  for (i=0;i<NMAXVAR;i++)
    trace->elem[trace->l]->ir.l_val_var[i]=ir->l_val_var[i];
  trace->l++;

  /* maj de la trace utilisateur */

  fprintf(trc_pf,"\n*********************************************************\n");
  fprintf(trc_pf,"\n cycle : %d \n\n\n",cycle);
  fprintf(trc_pf," debut d'execution de la regle : \n");

  /* copiage de l'instance de regle qui s'execute.*/

  lir=(struct l_inst_regles *)malloc(sizeof(struct l_inst_regles));
  lir->l=1;
  lir->elem[0]=ir;
  aff_l_inst_rg(trc_pf,lir);
  free(lir);
  fprintf(trc_pf,"-----------------------------------------\n\n");
};


/*****************************************************************************/
/* maj de la trace d'execution avant execution d'un appel recursif a         */
/* l'interpreteur.                                                           */
/*****************************************************************************/
```

```
trc_d_interp(arg)

 truct arg_interp *arg;


  extern FILE *trc_pf;

  fprintf(trc_pf,"\n execution d'un appel recursif a l'interpreteur.\n");
  fprintf(trc_pf,"identificateur de variable concernee : %d\n",arg->var);
  if (arg->strc==VRAI)
    fprintf(trc_pf,"concerne egalement la sous-structure qui en depend\n");
   else fprintf(trc_pf,"element terminal\n");
 ;


'*************************************************************************/
 * maj de la trace de'excution apres execution d'un appel recursif.     */
/*************************************************************************/

 rc_f_interp(cycle,status)

int cycle;
 nt status;

{ extern FILE *trc_pf;

  if (status==VIDE)
    { fprintf(trc_pf,"\n plus de regles activables...\n\n");
      status=OK;
    };

  fprintf(trc_pf,"fin de l'appel recursif du cycle %d\n",cycle);
  fprintf(trc_pf,"diagnostic d'execution de cette action :");
  aff_diag(status);
};


 *************************************************************************/
/* maj de la trace utilisateur par suite d'une action de type ajout     */
 * d'un element en memoire de travail.                                  */
 * Contrairement a l'appel recursif ci-dessus, il n'y aura qu'une seule*/
/* fonction, une erreur etant par definition exclue.                    */
/*************************************************************************/

trc_aj_wm(e)

 truct el_wm *e;

{ extern FILE *trc_pf;
  struct l_el_wm *lel;

  fprintf(trc_pf,"\n ajout de l'element suivant en WM : \n");

  lel=(struct l_el_wm *)malloc(sizeof(struct l_el_wm));
  lel->l=1;
  lel->elem[0]=e;
  aff_l_el_wm(trc_pf,lel);
  free(lel);

  fprintf(trc_pf,"\n diagnostic d'execution : ");
  aff_diag(OK);
};
```

```
 * maj de la trace utilisateur par suite de l'execution d'une action   */
/* d'ajout d'interpretation partielle.                                  */
/* Comme pour l'action precedente, une seule fonction est necessaire.  */
 ***********************************************************************/

trc_aj_int(arg)

 har *arg;

 extern FILE *trc_pf;


  fprintf(trc_pf,"\n ajout de la partie d'interpretation suivante :\n");
  fprintf(trc_pf,arg);
  fprintf(trc_pf,"\n diagnostic d'execution : ");
  aff_diag(OK);
 ;



 ***********************************************************************/
,* maj de la trace utilisateur lors de l'execution de l'action d'acces */
/* a tous les referes possibles d'un pronom.                            */
 ***********************************************************************/

trc_referes ()

  fprintf(trc_pf,"\n execution d'un acces a tous les referes possibles \n");
  fprintf(trc_pf,"d'un pronom...\n\n");

  fprintf(trc_pf,"diagnostic d'execution : ");
  aff_diag(OK);
  return(OK);
 ;



'***********************************************************************/
/* mise a jour de la trace apres execution de la regle courante...      */
/* ( diagnostic de son execution)                                       */
'***********************************************************************/

trc_diag_maj(status)

int status;

{ extern FILE *trc_pf;

  fprintf(trc_pf,"fin d'execution de la regle courante\n\n");

  fprintf(trc_pf,"diagnostic d'execution : ");
  aff_diag(status);

  fprintf(trc_pf,"-----------------------------------------\n\n");

};


/***********************************************************************/
/* teste la presence d'une instance de regle dans la trace d'execution */
/***********************************************************************/

trc_presence(ir)

struct inst_regle *ir;
```

```
extern struct trc_trace *trace;
int trouve, i;

i=0;
trouve=0;

while ((i<trace->l)&&(trouve==0))
  { if (compare_inst(ir,&trace->elem[i]->ir)==0)
      trouve=1;
    i++;
  };

return(trouve);
```

```
/*****************************************************************/
/*                                                               */
/*                  MODULE D'ACTIVATION DES REGLES.              */
/*                                                               */
/*****************************************************************/


#include <stdio.h>
#include "declarations.h"


extern struct ls_act *consequent();
extern maj_wm();
extern struct l_el_wm *maj_wm();
extern interpreteur();

/*****************************************************************/
/* gestion du stack de la memoire de travail pour les appels recursifs:*/
/*  PUSH                                                         */
/*****************************************************************/

int push(list)

struct l_el_wm *list;

{ extern struct stack *pile;

  if (pile==NULL)
    { pile=(struct stack *)malloc(sizeof(struct stack));
      pile->ptr=0;
    };

  if (pile->ptr==5)
    return(ECHEC);

  pile->elem[pile->ptr++]=list;
};

/*****************************************************************/
/* gestion du stack de la memoire de travail pour les appels recursifs:*/
/*  POP                                                          */
/*****************************************************************/

struct l_el_wm *pop()

{ extern struct stack *pile;

  return(pile->elem[--pile->ptr]);
};



/*****************************************************************/
/* execution d'un ordre d'interpretation.                        */
/*****************************************************************/

int ordre_interp(arg,l_val_var)

struct arg_interp *arg;
int l_val_var[NMAXVAR];


{ extern struct l_el_wm *wm;
  extern struct l_inst_regles *conset;
  extern int cycle;
```

```
  int old_cycle;

  if (push(wm)==ECHEC)
    return(ECHEC);

  wm=maj_wm(arg->strc, l_val_var[arg->var]);

  free(conset);
  conset=NULL;

  old_cycle=cycle;

  /* gestion de la trace d'execution*/
  trc_d_interp(arg);

  status=interpreteur();

  /* gestion de la trace d'execution (-->resultat de l'ap. rec.)*/
  trc_f_interp(cycle, status);

  free(conset);
  conset=NULL;

  wm=pop();
  cycle=old_cycle;
  return(status);
};

/*****************************************************************************/
/* action d'ajout d'un element en working memory.                            */
/*****************************************************************************/

int aj_el_wm(arg, l_val_var)

struct arg_wm *arg;
int l_val_var[NMAXVAR];

{ extern int cycle;
  struct el_wm *e;
  int i;

  e=(struct el_wm *)malloc(sizeof(struct el_wm));

  e->id_rel=arg->id_rel;
  e->n_arg=arg->nb_arg;
  e->age=cycle;

  for (i=0; i<arg->nb_arg; i++)
      e->arg[i]=l_val_var[arg->argrel[i]];

  /* gestion de la trace d'execution */

  trc_aj_wm(e);

  aj_wm(e);

  return(OK);
};

/*****************************************************************************/
/* action de supression d'un elememt de la memoire de travail.               */
/*****************************************************************************/

int sup_el_wm(arg, l_val_var)

struct arg_wm *arg;
```

```
int l_val_var[NMAXARG];

{

};


/*****************************************************************************/
/* action d'ajout d'une interpretation partielle.                            */
/*****************************************************************************/

int ajout_int(arg, l_val_var)

char *arg;
int l_val_var[NMAXARG];


{ extern char *cur_int;
  char *str;

  if (arg->t_arg==VARIABLE)
        str=nom_val_var(l_val_var, arg->uval.v);
   else str=arg->uval.const;

  /* gestion de la trace d'execution */
  trc_aj_int(str);

  while((*cur_int++ = *str++)!='\0');
  cur_int--;

};


/*****************************************************************************/
 * acces a tous les referes possibles d'un pronom dans l'historique    */
/* Cette fonction est actuellement simulee...                                */
/*****************************************************************************/


h_referes()

  struct el_externe *ex;

  ex=(struct el_externe *)malloc(sizeof(struct el_externe));
  ex->r=(struct rel *)malloc(sizeof(struct rel));
  ex->r->nom=(char *)malloc(30);
  strcpy(ex->r->nom, "refere");
  ex->r->nb_arg=1;
  ex->arg[0]=(char *)malloc(30);

  printf("\n Simulation de l'acces a l'historique... \n");
  printf("Introduisez successivement les differents groupes nominaux");
  printf(" susceptibles d'etre\n");
  printf("references par le pronom traite : \n");
  printf("\n next > ");

  gets(ex->arg[0]);

  while (strcmp(ex->arg[0], ""))
       { aj_wm_ext(ex);
         printf("\n next > ");
         gets(ex->arg[0]);
       };

  trc_referes();
```

```
/*****************************************************************************/
/* simulation de l'acees a l'historique, pour la resolution des              */
/* anaphores:                                                                */
/*  . ordre d'interpretation sur l'argument ;                                */
/*  . saisie au terminal du dernier enonce de la machine ;                   */
/*  . remplacement du point d'interrogation par l'interpretation de          */
/*  . l'argument ;                                                           */
/*  . et ordre de fin d'execution de l'interpeteur.                          */
/*****************************************************************************/

int en_mach(arg, l_val_var)

struct arg_ellipse *arg;
int l_val_var[NMAXARG];

{ extern char *cur_int;
  char *ch1,*ch2,*ch3,*str,*old_cur_int;
  struct arg_interp *arg_int;
  int i, j, status;

  str=(char *)malloc(30);
  ch2=(char *)malloc(30);
  ch3=ch2;
  old_cur_int=cur_int;

  if (arg->t_arg==VARIABLE)

    { arg_int=(struct arg_interp *)malloc(sizeof(struct arg_interp));
      arg_int->strc=FAUX;
      arg_int->var=arg->uvaleur.v;

      status=ordre_interp(arg_int, l_val_var);

      if ((status==ECHEC)||(status==ERREUR))
         return(status);

        i=0;   /* sauvetage de l'interp acquise avant */
        for (ch1=old_cur_int; ch1!=cur_int; *ch2++ = *ch1++, i++);

    };


  printf("\n Simulation de l'acces a l'historique...\n");
  printf("\n introduisez le dernier enonce_machine, s.v.p.\n");
  printf(" > ");
  gets(str);

  /* recopiage de la partie de l'enonce de la machine avant le '?'*/
  /* (de l'interpretation de la reponse fournie par le locuteur.   */

  for (ch1=old_cur_int; (*str!='?')&&(*str!='\0'); *ch1++ = *str++);

  if (*str=='\0')
    return(ERREUR);                    /* ce n'est pas une question*/

  /* on passe, dans l'enonce machine, le '?' et son eventuel */
  /* qualificatif.                                           */

  while ((*str!=',')&&(*str++ !=')')&&(*str!=' '));
  str--;
  if (*str=='?')
    str++;
```

```
  /* recuperation de l'interpretation de la reponse du locteur */
  /* qui vient maintenant completer l'enonce machine.          */

  if (arg->t_arg==VARIABLE)
      for (j=0; j<i; j++)
       *ch1++ = *ch3++;

  if (arg->t_arg==CONST)
       for (ch3=arg->uvaleur.const;*ch3!='\0';*ch1++ = *ch3++);

  /* on garde le reste de l'enonce machine */

  while ((*ch1++ = *str++)!='\0');
  ch1--;
  cur_int=ch1;

  return(OK);
};



/*************************************************************************/
/* active l'instance ir, c'est-a-dire execute l'ensemble de ses actions*/
/*************************************************************************/

int active(ir)

struct inst_regle *ir;

{ int fin = FAUX;
  int status,ac;
  struct ls_act *la;

  la=consequent(ir->id_rg);

  for (ac=0;ac<la->l;ac++)

   { switch (la->el[ac]->id_act) {

        case 0:
          status=ordre_interp(la->el[ac]->arg,ir->l_val_var);
          fin=VRAI;
          break;

        case 1:
          status=aj_el_wm(la->el[ac]->arg,ir->l_val_var);
          break;

        case 2:
          status=sup_el_wm(la->el[ac]->arg,ir->l_val_var);
          break;

        case 3:
          status=ajout_int(la->el[ac]->arg,ir->l_val_var);
          break;

        case 4:
          status=h_referes();
          break;

        case 5:
          status=en_mach(la->el[ac]->arg,ir->l_val_var);
          break;
```

```
    -

     if ((status==ERREUR) || (status==ECHEC))
         return(status);
   };

  if (fin==VRAI)
    return(FIN);

  return(OK);
};
```

```
/*****************************************************************************/
/*                                                                           */
/*                        MODULE D'INTERPRETATION.                           */
/*****************************************************************************/


#include <stdio.h>
#include "declarations.h"


/*****************************************************************************/
/*                      FONCTIONS EXTERNES UTILISEES.                        */
/*****************************************************************************/


/* module 'conflict_set.c' */

extern int slct_conflict_set();
extern struct inst_regle *resol_conflict_set();
extern init_conflict_set();
extern clot_conflict_set();

/* module 'activation.c' */

extern int active();

/* module 'gest_trace.c' */

extern trc_ant_maj();
extern trc_diag_maj();



/*****************************************************************************/
/*                            VARIABLES GLOBALES.                            */
/*****************************************************************************/

 int cycle;                             /* numero de cycle de l'interpreteur*/

 struct l_inst_regles *conset;          /* adresse du conflict_set          */
 char *cur_int;                         /* adresse de la position dans      */
                                        /* l'interpretation courante        */
 char *result_int;                      /* adresse du debut de              */
                                        /* l'interpretation                 */
 int option;                            /* identificateur des options prises*/

/*****************************************************************************/
/* extrait du conflict set courant les instances de regles qui ont           */
/* deja ete executees, c'est-a-dire celles qui se trouvent dans la           */
/* trace d'execution.                                                        */
/*****************************************************************************/

struct l_inst_regles *anti_cyclage(lir)

struct l_inst_regles *lir;


{ extern struct trc_trace *trace;
  struct l_inst_regles *l;
  int i;

  l=(struct l_inst_regles *)malloc(sizeof(struct l_inst_regles));
  l->l=0;

  for (i=0;i<lir->l;i++)
    { if (trc_presence(lir->elem[i])==FAUX)
```

```
        l->elem[l->l++]=lir->elem[i];
    };

  return(l);
};

/*****************************************************************************/
/* initialisations du systeme d'interpretation.                              */
/*****************************************************************************/

initialisation()

{ extern char *result_int;
  extern char *cur_int;

  init_conflict_set();
  trc_creation();
  result_int=(char *)malloc(100);
  cur_int=result_int;
};


/*****************************************************************************/
/* traduction de l'enonce a interpreter et chargement de cette               */
/* traduction dans la memoire de travail.                                    */
/*****************************************************************************/

trad_enonce()

{
 /* non implementee... simple lecture au terminal */

 printf("\n la traduction de l'enonce n'etant pas implementee, le \n");
 printf("contenu de la memoire de travail doit etre introduit a la main.\n");
 int_el_wm();
};


/*****************************************************************************/
/* affichage de l'interpretation de l'enonce, resultat du boulot de          */
/* de l'interpreteur.                                                        */
/*****************************************************************************/

aff_interp(pf)

FILE *pf;

{ FILE *pf_res;

  pf_res=fopen("res_interp","w");

  fprintf(pf_res,"\n %s \n",result_int);
  fprintf(pf,"\n %s \n\n\n\n",result_int);

  if (pf!=stdout)
    printf("\n %s \n\n\n\n",result_int);

};


/*****************************************************************************/
/* realisation des operations de cloture.                                    */
/*****************************************************************************/

cloture()
```

```
{
  clot_conflict_set();
};



/*****************************************************************************/
/* l'interpreteur de regles...                                               */
/*****************************************************************************/

interpreteur()

{ extern struct l_inst_regles *conset;
  extern int cycle;
  struct inst_regle *ir;
  int fin = FAUX;
  int status;

  cycle=0;

  while (fin==FAUX)

    { cycle++;

      status=slct_conflict_set();

      /*printf("\n conflict_set.....\n");*/
      /*aff_l_inst_rg(stdout,conset);*/

      if (status==OK)

        { conset=anti_cyclage(conset);
          /*printf("\n conflict set apres anti-cyclage.....\n");*/
          /*aff_l_inst_rg(stdout,conset);*/

          if (conset->l==0)
            return(VIDE);

          ir=resol_conflict_set();
          trc_ant_maj(ir);
          status=active(ir);
          trc_diag_maj(status);

          if ((status==ECHEC)||(status==ERREUR))
              return(status);
          if (status==FIN)
              fin=VRAI;
        }
      else return(status);

    };
  return(OK);
};



/*****************************************************************************/
/* module de controle de l'interpreteur de regles.                           */
/*****************************************************************************/


controle()


{ extern FILE *trc_pf;
  extern int option;
```

```
   int status;


   initialisation();

   trad_enonce();

   aff_wm(trc_pf);

   /* sauve_wm();                          */
   /* genere_hyp();                        */

   status=interpreteur();

   /* if (status==ECHEC)                   */
   /*    { reset_wm();                     */
   /*      status=interpreteur();          */
   /*    };                                */

   switch (status) {

      case ECHEC   : erreur2();
                     break;
      case ERREUR  : erreur3();
                     break;
      case VIDE    : if (trc_pf!=stdout)
                           printf("\n\n\n plus de regles activables...\n\n\n"
                     fprintf(trc_pf,"\n\n plus de regles activables...\n\n\n\
      case OK      : aff_interp(trc_pf);
                     break;
      default      : erreur4();
     };

  aff_wm(trc_pf);
};


main(argc,argv)

 int argc;
 char *argv[];

{ extern int option;

  if (argc>1)
    { if ((argv[1][0]=='-')&&(argv[1][1]=='s'))
        option=OFF_LINE;
      else { erreur1();
             return;
           }
    }
   else option=ON_LINE;

  controle();
}
```

```
/*****************************************************************************/
/*                                                                           */
/*          PROGRAMME DE GESTION DE LA BASE DE FAITS.                        */
/*                                                                           */
/*****************************************************************************/


#include <stdio.h>

#include "const.h"
#include "base_faits.h"



/*****************************************************************************/
/*                    FONCTIONS EXTERNES UTILISEES                           */
/*****************************************************************************/



/* module base_faits.c */

 extern ouv_bf();
 extern ferm_bf();
 extern intro_faits();
 extern suppr_fait();
 extern aff_bf();
 extern modif_fait();
 extern delete_bf();



/*****************************************************************************/
/* affichage du menu utilisateur.                                            */
/*****************************************************************************/

affich_menu()


{ printf("\n\n\n\n\n\n\n\n\n");
  printf("\n         GESTION DE LA BASE DE FAITS.    \n\n");
  printf("    (i) insertion de faits dans la base ; \n");
  printf("    (m) modification d'un fait de la base ; \n");
  printf("    (s) suppression d'un fait de la base ; \n");
  printf("    (d) destruction de la base de faits ; \n");
  printf("    (a) affichage des faits contenues dans la base ;\n");
  printf("    (p) impression des faits contenues dans la base;\n");
  printf("    (f) fin. \n\n");

  printf(" > ");
};



/*****************************************************************************/
/*                          PROGRAMME PRINCIPAL                              */
/*****************************************************************************/


main ()

{char *ch;
 int id_fait;
```

```
  ch=(char *)malloc(10);
  ouv_bf();

  affich_menu();

  gets(ch);

  while (strcmp(ch,"f"))
       { switch (*ch) {

            case 'i' : intro_faits();
                       break;

            case 'm' : printf("\n identificateur du fait a modifier?");
                       id_fait= -1;
                       scanf("%d",&id_fait);
                       gets(ch);
                       modif_fait(id_fait);
                       break;

            case 's' : printf(" identificateur du fait a supprimer?");
                       id_fait= -1;
                       scanf("%d",&id_fait);
                       gets(ch);
                       suppr_fait(id_fait);
                       break;

            case 'a' : aff_bf(stdout);
                       break;

            case 'd' : delete_bf();
                       break;

            case 'p' : aff_bf(fopen("b_faits.out","w"));
                       break;

            default  : erreur6();
        };

      affich_menu();
      gets(ch);
     };

   ferm_bf();
}
```

```
/*****************************************************************/
/*              PROGRAMME DE CREATION DES REGLES.                */
/*****************************************************************/

#include <stdio.h>
#include "const.h"
#include "base_regles.h"


/*****************************************************************/
/*                  FONCTIONS EXTERNES UTILISEES.                */
/*****************************************************************/

/* module 'base_regles.c'      */

extern ouvrir_br();             /* ouverture et chargement de la   */
                                /* base de regles                  */
extern fermer_br();             /* fermeture et sauvetage de la    */
                                /* base de regles.                 */
extern intro_regles();          /* introduction de regles dans la  */
                                /* base a partir du terminal.      */
extern modif_regle();           /* modification d'une regle de la  */
                                /* base.                           */
extern suppr_regle();           /* suppression d'une regle de la   */
                                /* base                            */
extern aff_br();                /* affichage du contenu de la base */
                                /* de regles.                      */
extern delete_br();             /* destruction de la base de regles*/


/*****************************************************************/
/*                        VARIABLES GLOBALES                     */
/*****************************************************************/

struct tab *T_actions;          /* adresse de la table des actions  */

struct valeurs *val;            /* adresse de la table des valeurs  */

struct relat *sch_rel;          /* adresse de la table des schemas  */
                                /* de rlations.                     */

struct l_regles *br;            /* adresse de la base de regles.    */



/*****************************************************************/
/* affichage du menu utilisateur.                                */
/*****************************************************************/

affich_menu()

{ printf("\n\n\n\n\n\n\n\n\n");
  printf("\n          GESTION DE LA BASE DE REGLES.    \n\n");
  printf("    (i) insertion de regles dans la base ; \n");
  printf("    (m) modification d'une regle de la base ; \n");
  printf("    (s) suppression d'une regle de la base ; \n");
  printf("    (d) destruction de la base de regles ; \n");
  printf("    (a) affichage des regles contenues dans la base ;\n");
  printf("    (p) impression des regles contenues dans la base;\n");
  printf("    (f) fin. \n\n");
```

```
  printf(" > ");
};


main()

{ char *ch;
  int id_rg;

  ouvrir_br();
  printf("...programme provisoire bete et mechant...");

  ch=(char *)malloc(10);

  affich_menu();

  gets(ch);

  while (strcmp(ch,"f"))

       { switch (*ch)  {

                case 'i' : intro_regles();
                           break;

                case 'm' : printf("\n identificateur de la regle a modifier?");
                           id_rg= -1;
                           scanf("%d",&id_rg);
                           gets(ch);
                           modif_regle(id_rg);
                           break;

                case 's' : printf("\n identificateur de la regle a supprimer?");
                           id_rg= -1;
                           scanf("%d",&id_rg);
                           gets(ch);
                           suppr_regle(id_rg);
                           break;

                case 'd' : delete_br();
                           break;

                case 'a' : aff_br(stdout);
                           break;

                case 'p' : aff_br(fopen("b_regles.out","w"));
                           printf("\n les regles sont dans b_regles.out\n");
                           break;

                default  : erreur6();
            };
          affich_menu();
          gets(ch);
        };

  fermer_br();
}
```

Facultés Universitaire Notre-Dame de la Paix

Institut d'informatique

Rue Grandgagnage, 21

B - 5000 Namur.

CONTRIBUTIONS A UN SYSTEME DE DIALOGUE HOMME-MACHINE A COMPOSANTE ORALE : ANNEXES (suite).

Mémoire présenté par

Manu Lorant

et

Serge Simonet

en vue de l'obtention

du titre de

Licencié et Maitre en Informatique

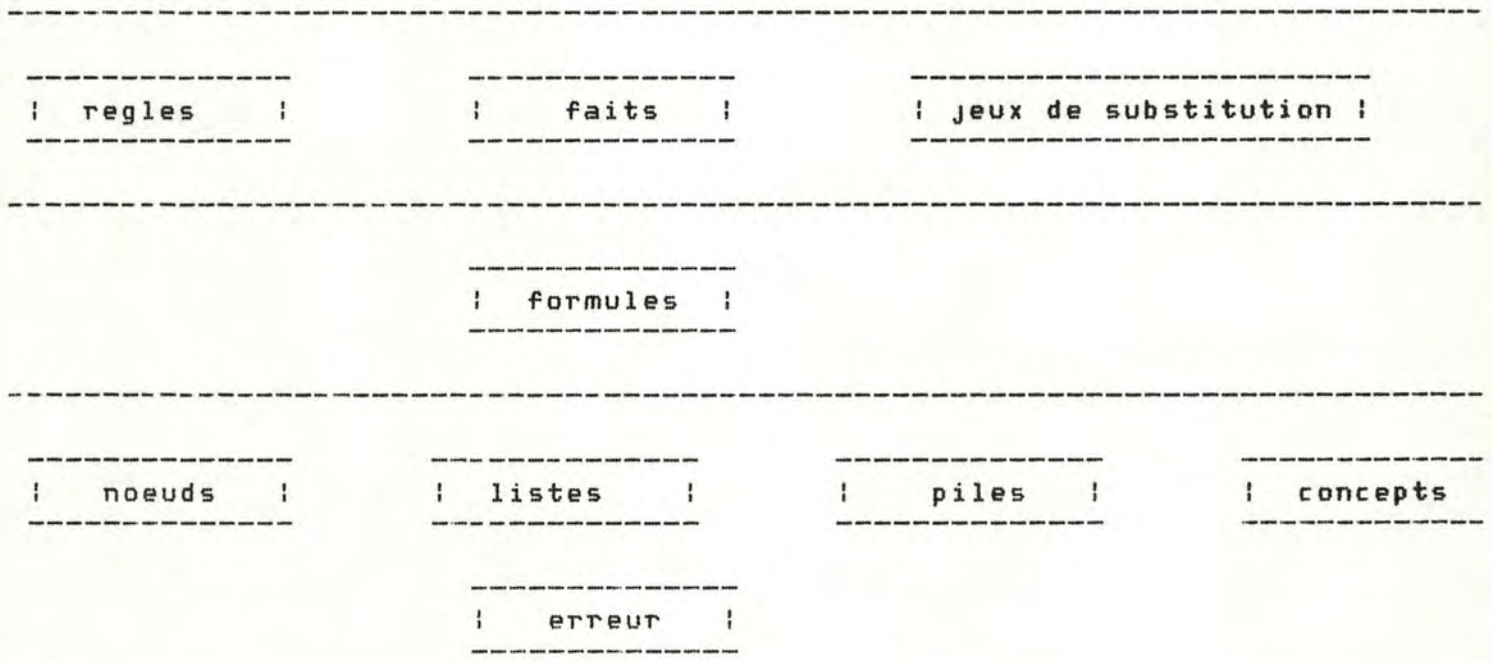
Année académique 1985-1986

ANNEXE III : PROGRAMME DE VERIFICATION DE COHERENCE.

PARTIE III.1 : SPECIFICATIONS DU PROGRAMME.

1 Spécification du programme de détection d'incohérence.

1.1 Architecture des modules.



1.2 Module erreur.

1.2.1 Spécification des fonctions.

1.2.1.1 Fonction erreur()

* Entrées : i : entier, s : une chaîne de caractères.

* Préconditions : aucune

* Résultats : aucun

* Fonction : afficher un message d'erreur correspondant à l'entier i fournit en ent'e et en y insérant la chaîne de caractères fournie en entrée.

1.3 Module liste

1.3.1 Spécification des objets.

Une liste est une collection d'objets de même type, tel qu'à partir d'un élément de la liste, il est possible d'accéder à son suivant. Une liste peut être vide ou non.

1.3.2 Spécification des fonctions.

1.3.2.1 Fonction ajq list()

* Entrées : l1, l2 : 2 listes.

* Préconditions : aucune.

* Résultats : ptrl : un pointeur vers une liste l.

* Postconditions : l1 € l et l2 € l.

* Fonction : Ajout de la liste l2 en tête de la liste l1.

1.3.2.2 Fonction ajq llist()

* Entrées : l1, l2 : 2 listes de listes.

* Préconditions : aucune.

* Résultats : ptrl : un pointeur vers une liste de listes.

* Postconditions : l1 € l et l2 € l.

* Fonction : Ajout de la liste de listes l2 en tête de la liste de listes l1.

1.3.2.3 Fonction ajt list()

* Entrées : l1 et l2 : 2 listes.

* Préconditions : aucune.

* Résultats : ptrl : un pointeur vers une liste l.

* Postconditions : l2 C l et l1 C l.

* Fonction : Ajouter la liste l2 en tête de la liste l1.

1.3.2.4 Fonction ajt llist()

* Entrées : l11, l12 : 2 listes de listes.

* Préconditions : aucune.

* Résultats : ptrl : un pointeur vers une liste de liste.

* Postconditions : l12 C l1 et l11 C l1.

* Fonction : ajouter la liste de listes l12 en ^te de la liste l11.

1.3.2.5 Fonction alloc list()

* Entrées : i, j : 2 entiers

* Préconditions : aucune

* Résultats : ptrl : un pointeur vers une paire d'entiers

* Postconditions : ptrl pointe vers la paire d'entiers (i, j).

* Fonction : allouer une paire d'entiers.

1.3.2.6 Fonction alloc llist()

* Entrées : ptrl : pointeur vers une liste

* Préconditions : aucune.

* Résultats : ptrll : un pointeur vers une liste de listes.

* Postconditions : ptrll pointe vers la liste de listes contenant la seule liste l.

* Fonction : allouer une liste.

1.3.2.7 Fonction coher list elt()

* Entrées : l : liste; i1, i2 : entiers.

* Préconditions : aucune.

* Résultats : x : entier.

* Postconditions : x ∈ [0,1,2]

* Fonction : Cette fonction a pour objectif de rechercher si la liste l conitent une paire d'entiers égale à (i1,i2), ou si elle contient une paire d'entiers égale à (i1,x) avec x différent de i2 ou encore si elle ne contient aucune paire d'entiers égale à (x,y) avec x égal à i1.
Selon le cas, l'entier renvoyé sera respectivement 0, 1 ou 2.

1.3.2.8 Fonction avance()

* Entrées : t0, t1 : tableaux de listes de liste de jeux de substitution.

* Préconditions : taille(t0) = taille(t1);

V i, 0 <= i < taille(t1) : t1[i] C t0[i];

V i, 0 <= i < taille(t1) : t1[i]->lliste est une liste d'une seule liste;

t1 constitue donc une suite de listes tirées à partir des listes des listes de t0.

* Résultats : t1' : tableau de listes de listes de jeux de substitution.

* Postconditions : ∀ i, 0 <= i < taille(t1) :
t1[i]->lliste est une liste d'une seule liste;

t1' constitue la suite suivante des liste de jeux de substitution qu'il est possible de tirer à partir de t0.

Une suite de listes est une suite S

$$(l_0, l_1, l_2, l_3, .., l_{n-1})$$

où li désigne la ième liste de jeux de subsitution. et où n = taille(tO).

La suite suivante S' de S est la suite

$$(l'_0, l'_1, l'_2, l'_3, .., l'_{n-1})$$

où l'_i

désigne la l_{i+1} ème liste de tO[i] si

$\forall j , l_i < j < n : l_i = taille(tO[i]->lliste)$

ou bien $l'_i = 0$ si $l'_{i-1} = l_{i-1} + 1$

* Fonction : Cette fonction a pour objectif de renvoyer une suite de listes qui soit la suivante d'une suite donnée de listes.

1.3.2.9 Fonction compar liste()

* Entrées : l1, l2 : 2 listes.

* Préconditions : aucune.

* Résultats : x : booléen.

* Postconditions : x = (l1 = l2).

* Fonction : Cette fonction a por objectif de comparer deux listes de paires d'entiers. Elle renvoie un booléen qui a la valeur VRAI si les deux listes sont identiques; il a la valeur FAUX si les deux listes ne sont pas identiques.

1.3.2.10 Fonction in liste0()

* Entrées : l : liste et i : entier.

* Préconditions : aucune.

* Résultats : ptrl : pointeur vers un élément de la liste l.

* Postconditions : ptrl -> val = i;.

* Fonction : déterminer si la liste l contient une paire d'entiers dont l'élément de gauche est égal à un entier donné.

1.3.2.11 Fonction in liste1()

* Entrées : l : liste et i : entier.

* Préconditions : aucune.

* Résultats : ptrl : pointeur vers un élément de la liste l.

* Postconditions : ptrl -> supp = i;.

* Fonction : déterminer si la liste l contient une paire d'entiers dont l'élément de droite est égal à un entier donné.

1.3.2.12 Fonction in liste2()

* Entrées : l : liste et i : entier.

* Préconditions : aucune.

* Résultats : ptrl : pointeur vers un élément de la liste l.

* Postconditions : ptrl -> val = i;.

* Fonction : déterminer si la liste l contient une paire d'entiers dont l'élément de gauche est égal à un entier donné.

1.3.2.13 Fonction union coher list()

* Entrées : l1, l2 : listes.

* Préconditions : aucune.

* Résultats : l : liste.

* Postconditions : si coher_list_elt(l1,l2) = 1 ,alors vide(l). si coher_list_elt(l1,l2) = 0 ou 2, alors l = l1 U l2.

* Fonction : Cette fonction a pour but de construire une liste l qui soit cohérente avec les deux listes l1 et l2. Une liste sera cohérente si elle ne contient pas deux paires d'entiers (x1,y1) et (x2,y2) où x1 est égal à x2 et x1 est différent de y2. S'il n'y a pas moyen de construire une telle liste à partir des liste l1 et l2, la liste renvoyée sera vide.

1.3.2.14 Fonction union liste()

* Entrées : d : tableau de listes.

* Préconditions : ∀ i , 0 <= i < taille(d) : d[i] est une liste.

* Résultats : l : liste.

* Postconditions : ∀ i , 0 <= i < taille(d), si ∃ j , 0 <= j < taille(d)

  avec union_coher_liste(d[i],d[j])=FAUX

  alors l = nil.

  ∀ i , 0 <= i < taille(d), si ∀ j , 0 <= j < taille(d)

  avec union_coher_liste(d[i],d[j])=VRAI

  alors l <> nil et

  ∀ (x,y) ∈ d[i] : (x,y) ∈ l et

$\forall$ (x,y) $\in$ d[j] : (x,y) $\in$ l

* Fonction : Cette fonction a pour objectif de renvoyer l'union de deux listes.

1.4 Module noeuds.

1.4.1 Spécification des objets.

Le noeud est objet qui correspond à l'occurrence d'un concept ou d'un opérateur ET ou OU.

Il possède un certain nombre d'attributs :

- son code qui détermine s'il s'agit de l'occurrence d'un concept, d'un ET ou d'un OU;
- son signe qui peut être PLUS ou MOINS;
- son suffixe;
- un pointeur vers un membre du groupe d'occurrences de concepts avec lesquels il est instancié;
- un pointeur vers le représentant du groupe d'occurrences de concepts avec lesquels il est instancié;
- un pointeur vers une liste d'occurrences de concepts qui sont ses arguments.
- un pointeur vers une liste d'occurrences de concepts dont il est argument.

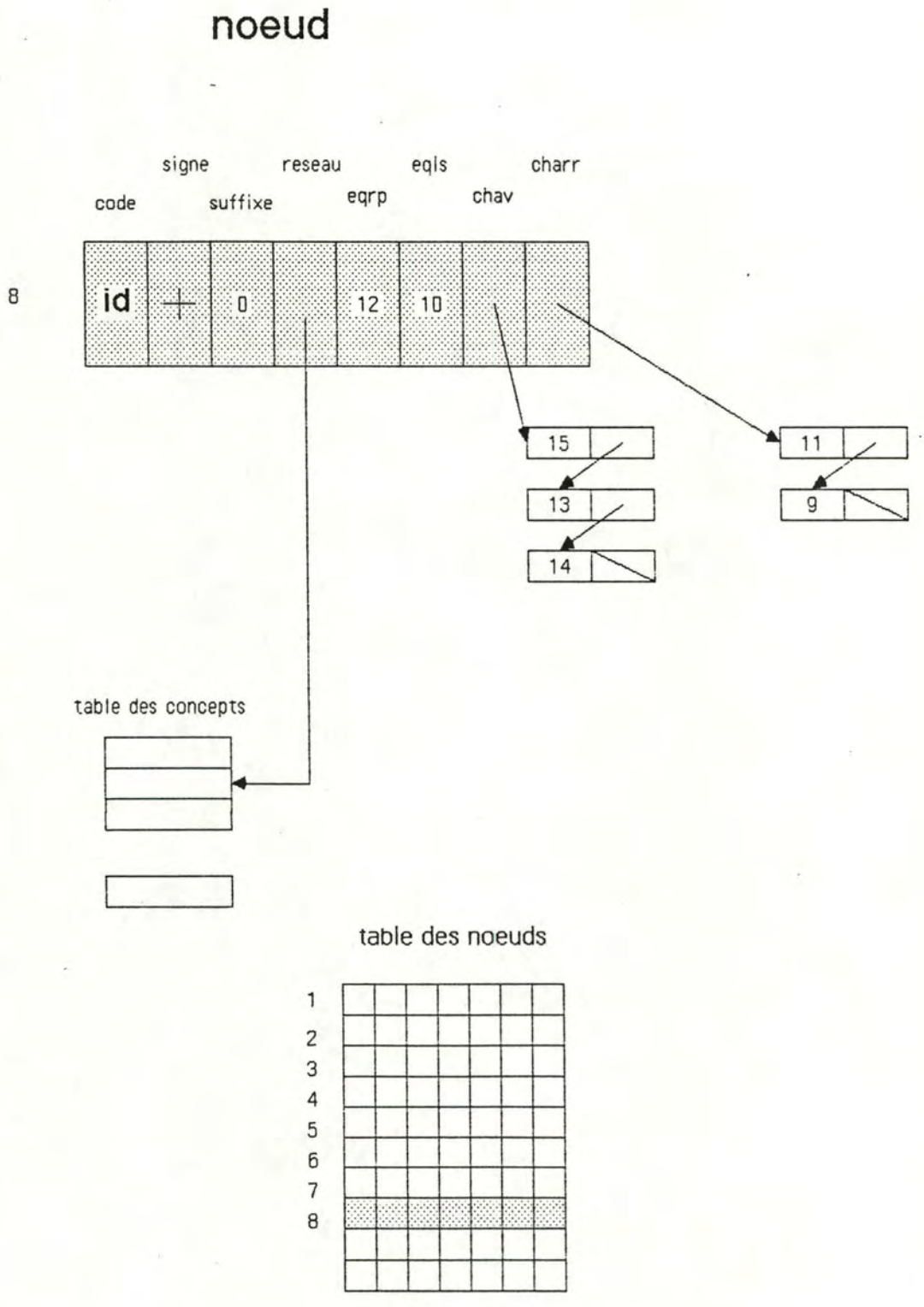


Figure 1.1: Représentation interne d'un noeud.

1.4.2 Spécification des fonctions.

1.4.2.1 Fonction copy_noeud()

* Entrées : f1, f2 : formules; l : liste de formules.

* Préconditions : f1 estla formule à recopier à la fin de la table des formules. Si f2 n'est pas vide, alors f1 est un argument de f2 et f2 a déjà été recopiée dans la table des formules. $\forall$ li $\in$ l, li doit encore être recopiée dans la table des formules.

* Résultats : aucun.

* Postconditions : aucune.

* Fonction : cette fonction a pour objectif de recopier la formule f1 à la fin de la table des formules.

1.5 Module pile.

1.5.1 Spécification des objets.

Une pile est une collection d'entiers pour laquelle il n'est possible que

- d'accéder qu'à l'entier qui est au sommet de celle-ci,
- de retirer l'entier qui se trouve à son sommet et
- d'ajouter un entier à son sommet.

1.5.2 Spécification des fonctions.

1.5.2.1 Fonction cp lst pile all()

* Entrées : P : pile; l : liste d'entiers.

* Préconditions : 0 <= P->ppile < Tnoeud.

* Résultats : P' : pile

* Postconditions : si P + long(l) < Tnoeud alors P' = P U { x : x ∈ l }

  si P + long(l) >= Tnoeud alors le message d'erreur numéro 4 est affiché.

* Fonction : fonction dont l'objectif est de copier le contenu de la liste d'entiers l au sommet de la pile P. Tous les éléments de la liste sans exception seront copiés au sommet de la pile si sa taille le permet. Si la taille de la pile est insuffisante, alors seule la partie de la liste pouvant être copiée le sera et un message d'erreur sera affiché.

1.5.2.2 Fonction cp lst pile onw()

* Entrées : P : pile; l : liste d'entiers.

* Préconditions : 0 <= P->ppile < Tnoeud.

* Résultats : P' : pile

* Postconditions : si P->ppile + #{ x : x ∈ l et x ∈ P } >= Tnoeud

  alors le message d'erreur numéro 4 est affiché.

  si P->ppile + #{ x : x ∈ l et x ∈ P } < Tnoeud

  alors P' = P U { x : x ∈ l et x ∈ p}

* Fonction : fonction dont l'objectif est de copier au sommet de la pile les éléments de la pile qui ne s'y trouvent pas encore, pour autant que la taille de la

pile le permette. Si la taille de la pile est insuffisante, alors seule la partie de la liste pouvant être copiée le sera et un message d'erreur sera affiché.

1.5.2.3 Fonction cp pile list()

* Entrées : P : pile;

* Préconditions : 0 <= P->ppile < Tnoeud;

* Résultats : l : liste d'entiers;

* Postconditions : $\forall$ x $\in$ P : x $\in$ l.

  0 <= P->ppile < Tnoeud;

* Fonction :

1.5.2.4 Fonction in pile()

* Entrées : P : pile; x : entier;

* Préconditions : 0 <= P->ppile < Tnoeud.

* Résultats : 0 ou 1.

* Postconditions : si x est contenu dans la pile, alors la valeur 1 est retournée;

  si x n'est pas contenu dans la pile, alors la valeur 0 est retournée;

* Fonction : fonction dont l'objectif est de déterminer si un entier x se trouve présent dans une pile P.

1.5.2.5 Fonction pop pile()

* Entrées : P : pile;

* Préconditions : P->ppile pointe vers le sommet de la pile (>=0)

* Résultats : x : entier; P' : pile;

* Postconditions : Si P->ppile = 0 alors rien n'est renvoyé et le message d'erreur numéro 5 est affiché.

  Si P->ppile >= 0, alors l'élément du sommet de la pile est renvoyé et P'->ppile = P->ppile-1;

* Fonction : fonction dont l'objectif est de renvoyer l'entier se trouvant au sommet de la pile P et de retirer cet entierdu sommet de la pile.

1.5.2.6 Fonction push pile()

* Entrées : P : pile; x : entier;

* Préconditions : le pointeur de P pointe vers le sommet de la pile P

* Résultats : P' : pile.

* Postconditions : Si P->ppile >= Tnoeud alors

  le message d'erreur numéro 4 est affiché et P' = P.

  Si P->ppile < Tnoeud alors

  P' = P U {x}.

  le pointeur de P' pointe au sommet de la pile

* Fonction : Fonction dont le but est de mettre l'entier x au sommet de la pile donnée en entrée.

1.5.2.7 Fonction vide pile()

* Entrées : P : pile.

* Préconditions : aucune.

* Résultats : P' : pile.

* Postconditions : sommet_pile = 0;
  ∀ x : x ∈ [0..T_NOEUD] : présent(x) = FAUX;

* Fonction : Initialiser une pile en faisant pointer le pointeur vers le sommet de la pile vers le fond de la pile et en indiquant qu'aucun entier ne s'y trouve.

1.6 Module concepts.

1.6.1 Spécification des objets.

Un concept est un objet représentant une entité lexicale du monde de l'application. Il posséde un certain nombre d'attributs.

- un libellé (libellé);
- un ordre, permettant de le situer par rapport aux autres complexes (ordre). Cet attribut est tel que

  ∀ c1, c2 : ordre (c1) > ordre (c2) OU ordre (c1) < ordre (c2);
- une liste des concepts dont il appartient aux classes (isa).
- une liste des concepts qui appartiennent à la classe qu'il définit (class);
- une liste des concepts qui sont ses arguments (arg);
- une liste des concepts dont il est argument (gra);
- une liste des formules dans lesquelles il apparaît.

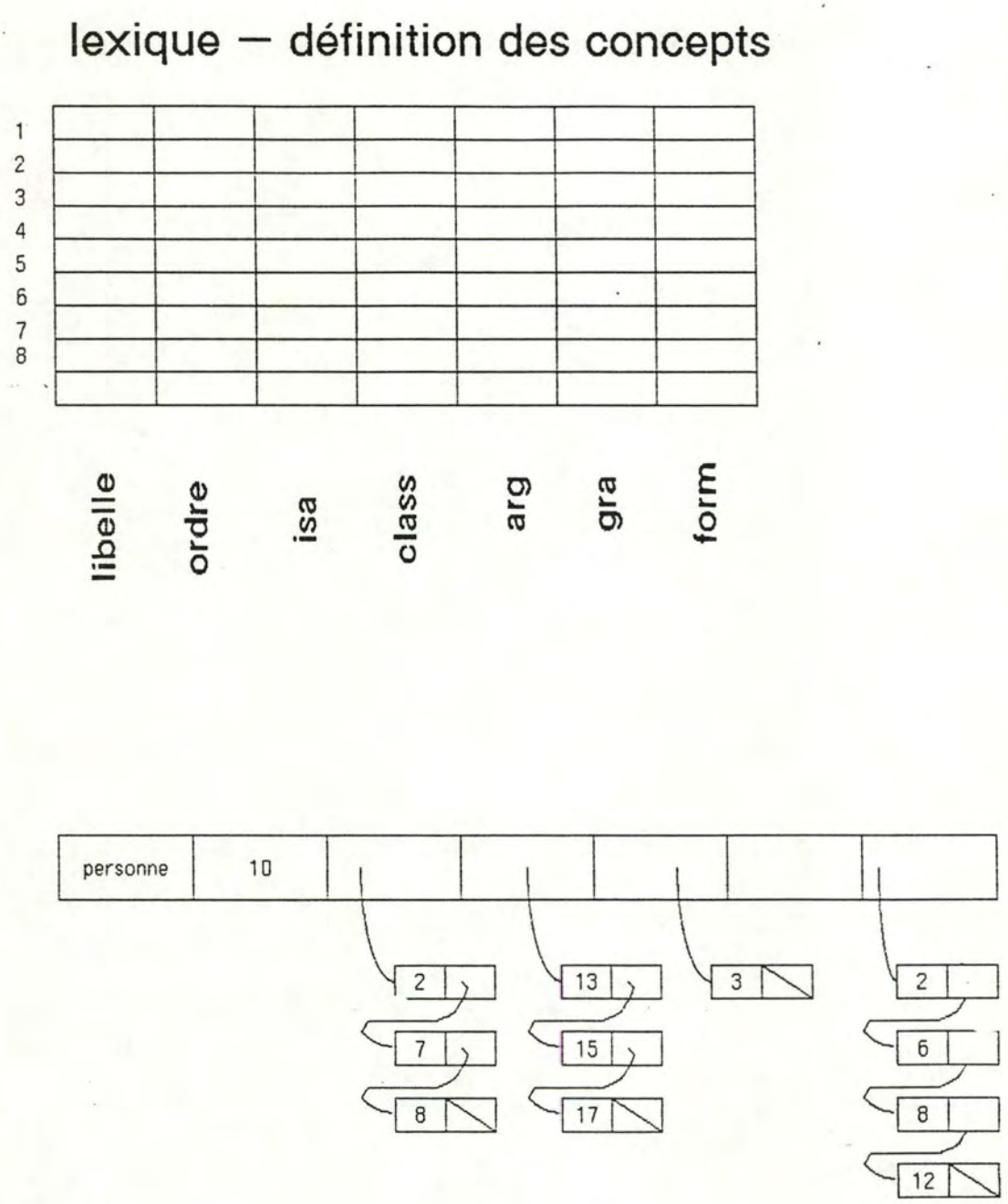


Figure 1.3: Représentation des concepts.

1.6.2 Spécification des fonctions.

1.6.2.1 Fonction instance nd()

* Entrées : c1, c2 : concept.

* Préconditions : c1 représente une occurence d'un concept C1 et c2 représente une occurence d'un concept C2.

* Résultats : b : booléen.

* Postconditions : b aura la valeur VRAI si l'occurence du concept C1 est une instance de l'occurence du concept C2 et b aura la valeur FAUX sinon.

* Fonction : Cette fonction a pour objectif de déterminer si une occurence d'un concept est instance d'une occurence d'autre concept.

1.6.2.2 Fonction isa()

* Entrées : c1, c2 : concepts.

* Préconditions : aucune.

* Résultats : b : booléen.

* Postconditions : b aura la valeur VRAI si c1 est un descendant de c2 et b aura la valeur FAUX si c1 n'est pas un descendant de c2.

* Fonction : Cette fonction a pour objectif de déterminer si un concept est fils d'un autre ou pas.

1.7 Module formules.

1.7.1 Spécification des objets.

Une formule est un objet de la forme f[(x,y,...)]. Où f, x, y, ... sont des occurences de concepts.
Tout formule correspond à une entrée dans la table des noeuds.

1.7.2 Spécification des fonctions.

1.7.2.1 Fonction in pre()

* Entrées : f : formule; c : concept.

* Préconditions : aucune.

* Résultats : ptr : pointeur vers l'endroit où le concept apparaît dans la formule.

* Postconditions : ptr est égal à nil si c n'apparaît pas dans f. ptr est différent de nil si c apparaît dans f.

* Fonction : Rechercher si un concept donné apparaît dans une formule. Si le concept y apparaît, alors un pointeur vers son occurence dans la formule est renvoyé, sinon un pointeur vide est renvoyé signalant l'absence d'occurence.

1.7.2.2 Fonction appar nd()

* Entrées : f1, f2 : formules.

* Préconditions : f2 est une instance de f1.

* Résultats : J : jeu de substitution.

* Postconditions : $\forall$ (x,y) $\in$ J : x $\in$ f1, y $\in$ f2.

* Fonction : Fonction dont le but est de determiner un jeu de substitution à partir de deux formules dont l'une est instance de l'autre.

1.7.2.3 Fonction liste instance fm()

* Entrées : f : formule.

* Préconditions : aucune.

* Résultats : ll : une liste de listes des jeux de substitutions.

* Postconditions : ∀ l ∈ ll : l contient les jeux de substitution pour la formule f.

  Si ll est vide, alors il n'y a aucune formule f' qui soit instance de f.

* Fonction : Cette fonction a pour objectif de rechercher parmi un ensemble de formules celles qui sont instance d'une formule déterminée 'f'. Pour chaque formule de l'ensemble qui constitue une instance de la formule 'f', une liste de jeux de substitution sera créé. Une liste de ces listes est renvoyée. Si aucune formule de l'ensemble n'est instance de 'f', alors la liste renvoyée est vide.

1.7.2.4 Fonction instance fm()

* Entrées : f1, f2 : formules.

* Préconditions : aucune.

* Résultats : b : booléen.

* Postconditions : b aura la valeur VRAI si la formule f1 constitue une instance de la formule f2, dans le cas contraire, b aura la valeur FAUX.

* Fonction : Cette fonction a pour objectif de vérifier si une formule constitue une instance d'une autre formule.

1.7.2.5 Fonction mk substi()

* **Entrées** : j : jeu de substitution; f : formule.

* **Préconditions** : ∀ (c1, c2) ∈ j, ∃ c ∈ f : c et c1 sont identiques.

* **Résultats** : f' : formule

* **Postconditions** : f' a le même profil que f.

  ∀ (c, c') : c et c' occupent respectivement la même position dans le profil de f et f',
  si ∃ (c1, c2) ∈ j : c1 = c
  alors c2 = c'

* **Fonction** : Cette fonction a pour objectif de substituer à chacune des occurences des concepts apparaissant dans f son correspondant de la suite de jeux de substitution.

1.8 Module faits.

1.8.1 Spécification des objets.

Un fait est une formule qui possède pour attributs

- une entrée dans la table des noeuds;
- un score de vraissemblance;
- un status indiquant s'il a été déduit, compris par le système, s'il s'agit d'un fait par défaut ou encore s'il s'agit d'un fait qui a été invalidé;
- une liste des règles qui ont permis sa déduction dans le cas où il s'agit d'un fait déduit;
- une liste des faits qui ont permis sa déduction dans le cas où il s'agit d'un fait déduit.

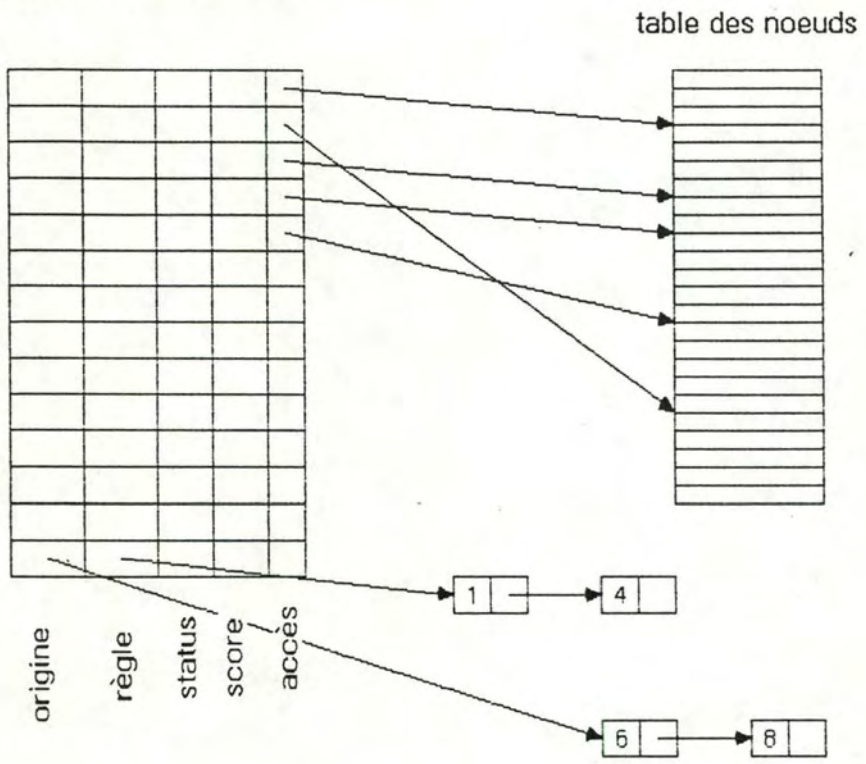


Figure 1.4: Représentation interne d'un fait.

1.8.2 Spécification des fonctions.

1.8.2.1 Fonction init base faits()

* Entrées :
  r : règle; BF la base de faits

* Préconditions : BF est vide.

* Résultats : BF' la base de faits.

* Postconditions : ∀ f, f ∈ précondition de r : f ⊂ BF'

  ∀ f' avec fait ∈ BF', si ∃ r' avec r' règle et post(r') = f'

  alors ∀ ff'' ∈ pré(r') : f'' ∈ BF.

* Fonction : Cette fonction a pour objectif d'initialiser la base de fait en fonction des prémisse d'une règle. L'ensemble des prémisses de la règle sera inséré dans la base de faits, ainsi que les prémisses de toute règle dont le conséquent serait un fait contenu dans la base de faits.

1.8.2.2 Fonction ajout fait()

* Entrées : f : fait élémentaire.
  BF : base de faits.

* Préconditions : aucune.

* Résultats : BF' : base de faits.

* Postconditions : BF' = BF ∪ { f }.

* Fonction : Cette fonction a pour objectif d'ajouter un fait à la base de faits.

1.8.2.3 Fonction eclat()

* Entrées : f : fait ou conjonction de faits. BF : base de faits.

* Préconditions : aucune.

* Résultats : BF' : base de faits.

* Postconditions : BF' = BF U { f } si f est un fait élémentaire. BF' = BF U { fi } si f est une conjonction de faits élémentaires.

* Fonction : Cette fonction a pour objectif d'ajouter un fait ou une suite de faits à la base de faits.

1.8.2.4 Fonction in bf()

* Entrées : f : fait; BF : base de faits.

* Préconditions : aucune.

* Résultats : b : booléen.

* Postconditions : b aura la valeur VRAI si f est contenu dans BF et la valeur FAUX sinon.

* Fonction :

1.9 Module règles.

1.9.1 Spécification des objets.

Une règle est un objet qui fait correspondre une formule f1 à une formule f2. Elle possède deux attributs :

- une entrée dans la table des noeuds correspondant à la formule qui constitue son antécédent;
- une entrée dans la table des noeuds correspondant à la formule qui constitue son conséquent.

1.9.2 Spécification des fonctions.

1.9.2.1 Fonction eval()

* Entrées : Tbf : la table des faits. r : règle.

* Préconditions : r est une règle de la base de règles.

* Résultats : ll : liste de listes de jeux de substitution.

* Postconditions : ∀ l ∈ ll : l est une liste de jeux de substitution pour la précondition de r.

* Fonction : Cette fonction a pour objectif de rechercher l'ensemble des jeux de substitution pour lesquels la précondition de la règle donnée en entrée est saitsfaite.

1.9.2.2 Fonction liste instance cond()

* Entrées : f : formule.

* Préconditions : f est la précondition d'une règle de la base de règles BR et est une conjonction de formules fi, où fi désigne la i ème formule qui compose la formule f.

* Résultats : t : tableau de listes de listes de jeux de substitution pour les formules fi.

* Postconditions : ∀ fi ∈ f : t[i] est un pointeur vers une liste de listes de jeux de substitution pour la formule fi.

* Fonction : Cette fonction a pour objectif de rechercher toutes les instances d'une précondition de règles qui appartiennent dans la base de faits.

1.9.2.3 Fonction msg applic rq1()

* Entrées : r : règle.

  yytrace : fichier

* Préconditions : r appartient à la base de rèles.

* Résultats : yytrace modifié.

* Postconditions : la chaîne de caractères "application de la règle", la règle r ont été écrites dans le fichier yytrace.

* Fonction : Cette fonction a pour objectif d'écrire dans le fichier yytrace que la règle r a été appliquée.

1.9.2.4 Fonction msg applic rg2()

* Entrées : r : règle,

  yytrace : fichier.

* Préconditions : r est une règle de la base de règles.

* Résultats : yytrace modifié.

* Postconditions : la chaîne de caractères "application de la règle", la règle r et la chaîne " mais le fait se trouve déjà dans la base de faits" ont été écrites dans le fichier yytrace.

* Fonction : Cette fonction a pour objectif d'écrire dans le fichier yytrace que la règle r a été appliquée mais que le fait dont elle a provoqué la déduction se trouvait déjà dans la base de faits.

1.9.2.5 Fonction msg inco det()

* Entrées : r : règle.

  l : liste de jeux de substitution.

  yytrace : fichier.

* Préconditions : r est une règle de la base de règles.

  ∀ (x, y) ∈ l : x apparaît dans la précondition de r.

* Résultats : yytrace modifié.

* Postconditions : la chaîne de caractères "incohérence détectée lors de l'application de la règle", la règle r, la chaine "avec les jeux de substitution " et la liste de jeux de substitution ont été ecrites dans yytrace.

* Fonction : Cette fonction a pour objectif d'écrire dans le fichier yytrace um message signalant qu'une incohérence a été détectée lors de l'application d'une règle pour un certain jeu de substitution.

1.10 Module trace.

1.10.1 Spécification des objets.

La trace est un ensemble de couples. Chaque couple est composé d'un numéro de règle et d'une liste de jeux de substitution. La présence d'un couple (r, lj) dans la trace indique que la règle numéro r a été appliquée avec la liste de jeux de substitution lj.

1.10.2 Spécification des fonctions.

1.10.2.1 Fonction ajouter regle trace()

* Entrées : T : la trace d'exécution.

  r : regle.

  l : liste de jeux de substitution

* Préconditions : r est une règle de la base de règles.

  le couple défini par la règle r et la liste de jeux de substitution l ne se trouve pas dans T.

* Résultats : T modifié.

* Postconditions : le couple défini par la règle, r, et la liste de jeux de substitution, l, se trouve dans T.

* Fonction :

1.10.2.2 Fonction in trace()

* Entrées : r : règle; j : jeu de substitution.

* Préconditions : ∀ (c1, c2) ∈ j : la précondition de r contient au moins une occurence de c1.

* Résultats : b : booléen.

* Postconditions : b aura la valeur VRAI si la trace d'exécution contient un couple (r, j).

* Fonction : Cette fonction a pour objectif de vérifier si la trace d'exécution contient déjà le couple (r, j), c'est à dire si la règle r a déjà été appliquée pour le jeu de substitution j. Dans tout autre cas, b aura la valeur FAUX.

1.11 Programme principal.

1.11.1 Fonction traitement()

* Entrées : r : règle

  Tfaits : base de faits.

  Trc : base de règles.

  yytrace : fichier.

* Préconditions : r est une règle de la base de règles qui n'a pas encore été traitée et qui n'a jamais été appliquée.

* Résultats : yytrace modifié.

  Tfaits : modifié.

* Postconditions : tous les faits qu'il était possible de déduire à partir de l'état initial de la base de faits ont été deduits. en d'autres termes il n'existe plus de règle qui, si elle était appliquée, permmette de déduire un nouveau fait.

* Fonction : Cette fonction a pour objectif de réaliser la recherche d'incohérence pour une règle donnée.

1.12 Module d'acqusition des données.

Les fonctions de ce module ont pour objectif d'aller extraire des fichiers issus de la compilation du fichier décrivant la tâche, les informations nécéssaires à la vérification d'incohérence.

1.12.1 ouvrir()

* Entrées : ndf : une chaîne de caractères.

* Préconditions : ndf correspond à un nom de fichier.

* Résultats : les fichiers correspondant à la chaîne donnée en entrée à laquelle a été ajoutée l'un des suffixe suivant sont ouverts.

  Les suffixes ajoutés sont
  - .nd.d
  - .ch.d
  - .isa.d
  - .class.d
  - .arg.d
  - .cond.d
  - .lex.d
  - .lien.d
  - .bd.d
  - .rc.d
  - .rs.d

* Postconditions : $\forall$ f, avec f fichier référence' par ndf auquel a été l'un des suffixes :
  f est ouvert en lecture.

  Si l'un des fichiers ainsi défini n'existe pas, alors le nom de ce fichier est affiché avec le message d'erreur numéro 3 et l'exécution se termine.

* Fonction : Cette fonction a pour objectif d'ouvrir en lecture l'ensemble des fichiers nécessaires au chargement des règles et des concepts. Si un fichier est inexistant, alors l'exécution du programme se termine avec l'affichage d'un message d'erreur précisant le nom du fichier inexistant.

1.12.2 fermer()

* Entrées : les noms des fichiers qui ont été ouverts par la fonction ouvrir.

* Préconditions : Les fichiers sont ouverts.

* Résultats : Fermetures des fichiers.

* Postconditions : ∀ f , f est un des fichiers ouvert par la fonction ouvrir : f est fermé.

* Fonction : Cette fonction a pour objectif de fermer les fichiers qui ont été ouverts pour le chargement des règles et des concepts.

1.12.3 acq lex()

* Entrées : f le fichier contenant les concepts.

* Préconditions :

  Le fichier f est ouvert.

  Le nom du fichier f se termine par le suffixe ".lex.d".

* Résultats : Tlex : table des concepts,

  Nblex : entier.

* Postconditions : ∀ c , c C : c ∈ Tlex.

  Nblex est le nombre de concepts repris dans Tlex.

* Fonction : Cette fonction a pour objectif l'acquisition des chaînes de caractères correspondant au lexique.

1.12.4

* Entrées : f : fichier des arguments des concepts.

* Préconditions : le nom de f se termine par le suffixe ".arg.d".

  Le fichier f est ouvert.

* Résultats : Tlex : table des concepts.

* Postconditions : A partir de chaque concept, il est possible d'accéder à l'ensemble de ses arguments.

* Fonction : Cette fonction a pour objectif l'acquisition des arguments des concepts.

1.12.5 acq fils()

* Entrées : f : fichier des fils des concepts selon la relation "isa".

* Préconditions : Le nom du fichier f se termine par le suffixe ".isa.d".

  le fichier f est ouvert.

* Résultats : Tlex : table des concepts.

* Postconditions : A partir de chaque concept, il est possible d'accéder à l'ensemble des concepts dont il est représentant des classes.

* Fonction : Cette fonction a pour objectif l'acquisition des fils, lien"isa", de chaque concept.

1.12.6 acq pere()

* Entrées : f : fichier des pères des concepts selon la relation "class".

* Préconditions : Le nom du fichier f se termine par le suffixe ".class.d".

  le fichier f est ouvert.

* Résultats : Tlex : table des concepts.

* Postconditions : A partir de chaque concept, il est possible d'accéder à l'ensemble des concepts dont il appartient à la classe ou à la sous_classe.

* Fonction : Cette fonction a pour objectif l'acquisition des fils, lien"isa", de chaque concept.

1.12.7 acq noeud()

* Entrées : f : fichier des noeuds.

* Préconditions : Le nom du fichier f se termine par le suffixe ".nd.d".

  le fichier f est ouvert.

* Résultats : Tnoeud : table des noeuds;

  Nbnoeud : entiers.

* Postconditions : La table ds noeuds contient Nbnoeud entrées.

  $\forall$ i, 0 <= i < Nbnoeud : Tnoeud[i].eqrp =Tnoeud[i].eqls = i.

* Fonction : Cette fonction a pour objectif l'acquisition des noeuds.

1.12.8 acq_lien()

* Entrées : f : fichier des liens.

* Préconditions : Le nom du fichier f se termine par le suffixe ".lien.d".

  le fichier f est ouvert.

* Résultats : Tnoeud : table des noeuds;

* Postconditions : $\forall$ x : 0 <= x < Nbnoeud : il est possible d'accéder au noeuds qui sont arguments de la formule représentée par x et aux noeuds dont la formule représentée par x est argument.

* Fonction : Cett fonction a pour objectif d'acquérir l'ensemble des lien existant entre les différents noeuds.

1.12.9 acq chaine()

* Entrées : f : fichier des chaînes.

* Préconditions : Le nom du fichier f se termine par le suffixe ".ch.d".

  le fichier f est ouvert.

* Résultats : Tchaine : la table des chaînes.

  Nbchaine : entier.

* Postconditions : ∀ i : 0 <= i < Nbchaine : x représente une chaine.

* Fonction : Cett fonction a pour objectif d'acquérir l'ensemble chaînes.

1.12.10 acq cond()

* Entrées : f : fichier des conditions.

* Préconditions : Le nom du fichier f se termine par le suffixe ".cond.d".

  le fichier f est ouvert.

* Résultats : Tcond : tables de conditions.

  Nbcond : entiers.

* Postconditions : ∀ x : 0 <= x < Nbcond : Tcond[x] représente une condition.

* Fonction : Cett fonction a pour objectif d'acquérir l'ensemble conditions.

1.12.11 acq bd()

* Entrées : f : fichier de la base de données.

* Préconditions : Le nom du fichier f se termine par le suffixe ".bd.d".

  le fichier f est ouvert.

* Résultats : Tbd : table de la base de données.

* Postconditions : ∀ x : 0 <= x < Nbbd : Tbd[x] représente un élément de la base de données.

* Fonction : Cett fonction a pour objectif d'acquérir l'ensemble des éément de la base de données.

1.12.12 acq rc()

* Entrées : f : fichier des règles de réécriture.

* Préconditions : Le nom du fichier f se termine par le suffixe ".rc.d".

  le fichier f est ouvert.

* Résultats : Trc : la table des règles de réécriture.

  Nbrc : entier.

* Postconditions : ∀ x : 0 <= x < Nbrc : Trc[x] représente une règle de réécriture. Trc[x].pre désigne sa précondition et Trc[x].pot désigne sa post condition.

* Fonction : Cett fonction a pour objectif d'acquérir l'ensemble des règles de réécriture.

1.12.13 init bf()

* Entrées : Tbd : ta table de la base de données.

* Préconditions : ∀ x, 0 <= x < Nbbd : Tbd[x] désigne n élément de la base de données.

* Résultats : Tbf : la table des faits ou base de faits.

  Nbbf : entier.

* Postconditions : ∀ x : 0 <= x < Nbrc : Trc[x] représente un fait de la base de faits.

  Nbf = Nbbd.

* Fonction : Cette fonction a pour objectif d'initialiser la base de faits avec le contenu de la base de données.

1.12.14 acquerir()

* Entrées : fi : l'ensemble des fichiers produits par le compilateur cpd pour une description de la tâche.

* Préconditions : aucune.

* Résultats : Ensemble des tables à partir des fichiers en entrée.

  Ensemble des entiers déterminant la taille de chacune des tables.

* Postconditions : Les postconditions sont constituées par l'ensemble des postconditions de chacune des fonctions appelées dans acquérir, à savoir :

  - acq_lex,
  - acq_arg,
  - acq_fils,
  - acq_pere,
  - acq_noeuds,
  - acq_lien,
  - acq_chaine,

- acq_bd,
- acq_rc,
- acq_cond

* Fonction : Cette fonction a pour objectif d'acquérir l'ensemble des données compilée par "cpd". et nécessaire à l'exécution du programme.

1.13 Module des utilitaires d'écran et de visualisation.

1.13.1 ok()

* Entrées : une chaîne de caractères.

* Préconditions : aucune.

* Résultats : b : booléen.

* Postconditions : Si le caractère saisi au terminal est 'o' ou 'y' b aura la valeur VRAI sinon b prendra la valeur FAUX;
* Fonction : Cette fonction a pour objectif d'afficher un message de demande de confirmation au terminal, d'attendre une réponse de l'utilisateur et en fonction de cette réponse de renvoyer la valeur VRAI ou la valeur FAUX.

1.13.2 ligne()

* Entrées : aucune

* Préconditions : aucune.

* Résultats : passage à la ligne suivante dans yytrace.

* Postconditions : aucun.
* Fonction : Cette fonction a pour objectif de passer à la ligne dans le fichier yytrace.

1.13.3 trait()

* Entrées : l : entier

x : char

* Préconditions : x est un caractère affichable à l'écran

* Résultats : le fichier yytrace modifié.

* Postconditions : aucune.
* Fonction : Cette fonction a pour objectif d'écrire une ligne composée de l fois le caractère x dans le fichier yytrace.

1.13.4 visu lex()

* Entrées : aucune

* Préconditions : aucune.

* Résultats : le fichier yytrace modifié.

* Postconditions : aucune.
* Fonction : Cette fonction a pour objectif de lister dasn le fichier yytrace le contenu de la table du lexique. Pour chaque entrée dans la table, on écrira
- un entier identifiant,
- une chaîne de caractères correspondant au libelle du concept,
- un entier correspondnat à l'ordre du concept,
- la liste des pères ou éventuellement un message signalant qu'il ne possède pas de père,
- la liste des fils ou éventuellement un message signalant qu'il ne possède pas de fils,
- la liste des arguments ou éventuellement un message signalant qu'il ne possède pas d'argument,
-

1.13.5 visu liste()

* Entrées : l : liste.

* Préconditions : aucune.

* Résultats : le fichier yytrace modifié.

* Postconditions : aucune.
* Fonction : Cette fonction a pour objectif d'écrire dans le fichier yytrace le contenu d'une liste l.

1.13.6 visu llist()

* Entrées : ll : une liste de liste.

* Préconditions : aucune.

* Résultats : le fichier yytrace modifié

* Postconditions : aucune.
* Fonction : Cete fonction a pour objectif d'écrire dans yytrace le contenu d'une liste de listes. Chacune des liste sera écrite de la même manière que pour visu_list().

1.13.7 visu list inst()

* Entrées : lj : liste de jeux de substitution.

* Préconditions : aucune.

* Résultats : le fichier yytrace modifié

* Postconditions : aucune.
* Fonction : Cette fonction a pour objectif d'écrire dans yytrace le contenu d'une liste de jeux de substitution.

1.13.8 visu rec nd()

* Entrées : n : noeud.

* Préconditions : n appartient à la table des noeuds.

* Résultats : le fichier yytrace modifié

* Postconditions : Le noeuds et tous les noeuds qui lui étaient reliés par le bief de 'chav' sont écrits dans yytrace.

* Fonction : Cette fonction a pour objectif d'écrire dans yytrace un noeud et tous ses descendants. Pour chaque noeud, on écrira son type, son signe, le libelle du concept dont il représente une occurence, son suffixe.

1.13.9 visu nd()

* Entrées : f : formule.

* Préconditions : aucune.

* Résultats : le fichier yytrace modifié

* Postconditions : aucune.

* Fonction : Cette fonction a pour objectif d'écrire la formule

1.13.10 visu tn()

* Entrées : Tn : la table des noeuds.

* Préconditions : aucune.

* Résultats : le fichier yytrace modifié

* Postconditions : le contenu de la tabe Tn a été ajouté au fichier yytrace.

* Fonction : Cette fonction a pour objectif d'écrire dans yytrace le contenu de la table des noeuds.

1.13.11 visu rg()

* Entrées : r : regle.

* Préconditions : r appartient à la table des règles.

* Résultats : le fichier yytrace modifié

* Postconditions : la règle r a été écrite dans le fichier yytrace.

* Fonction : Cette fonction a pour objectif d'écrire dans le fichier yytrace la règle r.

1.13.12 visu rc()

* Entrées : Trc : la table des règles.

* Préconditions : aucune.

* Résultats : le fichier yytrace modifié

* Postconditions : la table des règles a été écrite dans le fichier yytrace.

* Fonction : Cette fonction a pour objectif d'écrire le contenu de la table des règle dans le fichier yytrace.

1.13.13 visu ft()

* Entrées : t : type des faits à écrire. Tbf : la table des faits.

* Préconditions : t est l'un des types [F_DELETE, F_ETOILE, F_BD, F_ENONCE, F_DEDUIT]

* Résultats : le fichier yytrace modifié

* Postconditions : Si Tbf est vide, alors un message le signalant sera écrit dans yytrace.

  Si t = F_ETOILE, alors l'ensemble des faits de la base de faits ont été écrits dans le fichier yytrace.

  Si t <> F_ETOILE, alors les faits dont le statut est égal à t ont été écrits dans le fichier yytrace.

* Fonction : Cette fonction a pour objectif d'écrire les faits d'un type donné dans le fichier yytrace.

1.13.14 visu tr()

* Entrées : Trace : la trace d'exécution.

* Préconditions : aucune.

* Résultats : le fichier yytrace modifié

* Postconditions : Le contenu de la trace a été écrit dans le fichier yytrace.

* Fonction : Cette fonction a pour objectif d'écrire le contenu de la trace dans le fichier yytrace.

PARTIE III.2 : MANUEL D'UTILISATION DU PROGRAMME.

2 Manuel d'utilisation du programme inco.

2.1 Introduction.

L'objectif de ce progrmme est de soumettre un ensemble de règles de réécriture au test de détection d'incohérence présenté au chapitre 5 de ce mémoire.

Les données utilisées par le programme devront être entrées par l'utilisateur, compilées par lui et ensuite soumises au programme de détection d'incohérence. Une illustration du processus à suivre est présenté à la figure [1.1].

2.2 Introduction des concepts.

L'introduction des concepts se fera en respectant la grammaire décrite dans l'annexe précédente.

2.3 Introduction des règles.

Les règles seront décrites par l'utilisateur à l'aide des concepts définis au point précédent. La définition des règles sera également effectuée à l'aide d'un éditeur de texte, à la suite de la définition des concepts et dans le même fichier.

L'écriture des règles doit satisfaire à la grammaire présentée dans l'annexe 1.

Le lecteur trouvera dans l'annexe 1 des exemples de règles de réécriture.

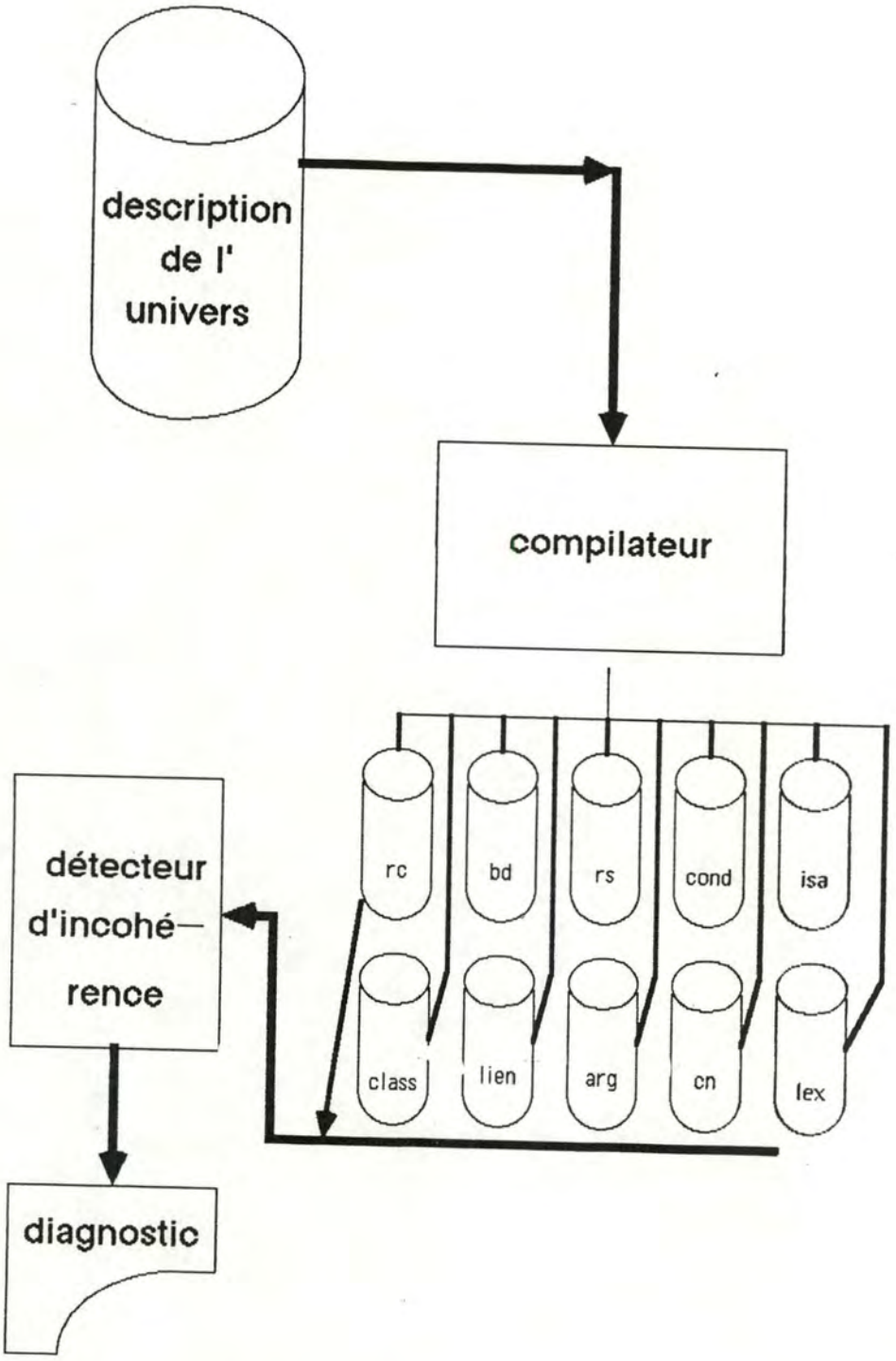


Figure 2.1: Schéma du processus de détection des incohérences.

2.4 Compilation des concepts et des règles.

Lorsque la définition des concepts et des règles est terminée, l'ensemble doit être compilé. La compilation se réalise par la commande

```
cpd non_de_fichier
```

où nom_de_fichier désigne le fichier dans lequel ont été définis les concepts et les règles.

Ce programme provoque :

[1] la lecture du fichier nom de fichier et la vérification de la correction de la syntaxe utilisée pour la définition des règles et concepts. Toute erreur provoque l'affichage d'un diagnostic à l'écran.

[2] La création d'un fichier récapitulatif où sont repris pour chacun des concepts définis
- son profil,
- ses concepts fils,
- ses concepts pères,
- les concepts qu'il possède pour arguments,
- les concepts dont il est argument.

[3] La création d'un ensemble de fichiers qui contiennent le résultat de la compilation et qui sont utilisés lors de la vérification de cohérence des règles de réécriture.

Ces fichiers sont les suivants :

* nom_de_fichier.arg.d : les arguments des concepts.

* nom_de_fichier.bd.d : la base de données, inutilisée dans notre application de détection d'incohérence.

* nom_de_fichier.class.d : les classes de concepts.

* nom_de_fichier.cond.d : les conditions de renseignements, inutilisés dans notre application de détection d'incohérence.

* nom_de_fichier.isa.d : la hiérarchie des concepts du bas vers le haut.

* nom_de_fichier.lex.d : la liste des concepts classés par ordre alphabétique croissant.

* nom_de_fichier.lien.d : les liens entre les noeuds et les concepts.

* nom_de_fichier.nd.d : la liste des noeuds (cfr la description de la représentation interne).

* nom_de_fichier.out : le résumé et diagnostics de la compilation.

* nom_de_fichier.rc.d : les règles de réécriture.

* nom_de_fichier.rs.d : les renseignements, fichier inutilisé dans le cadre de la détection d'incohérence des règles de réécriture.

Si le compilateur envoie le message "0 erreur detectee", alors la déclaration est correte, et l'utilisateur peut passer à la phase suivante, sinon il doit corriger ses éventuelles erreurs de syntaxe et recompiler son fichier.

Cette phase est répétée autant de fois que nécessaire.

PARTIE III.3 : EXEMPLES D'EXECUTION DU PROGRAMME.

3 Exemples de fonctionnement du programme.

Nous avons réalisé deux exemplesqui illustrent les capacités du programme de détection d'incohérence dans les règles de réécriture. Pour chacun de ces exemples, nous donnons

- une copie du fichier contenant les déclarations des concepts et des règles;
- la commande ayant permis l'exécution du programme de détection d'incohérence;
- une copie du résultat obtenu par ce programme.

L'un des deux exemples choisi est composé de règles qui, vraisemblablement, apparaîtront dans le la description de l'univers de la tâche du système développé à Nancy une fois qu'il sera dans sa phase opérationnelle. L'autre est un exemple plus général qui ne porte pas sur un domaine d'application particulier.

3.1 Exemple I (a).

3.1.1 Déclaration des concepts et des règles.

La déclaration est effectuée via un editeur de texte et le fichier ainsi créé se nomme "an".

```
%%
anime [ personne [ tiers, locuteur, systeme ],
        animal   [ chien, chat, poisson ]
      ],
etat(personne) [ majeur, mineur ],

famille [ parent(personne, personne)[ pere, mere ],
          enfant(personne, personne)[ fils, fille ]
        ],
sexe(personne) [ masculin, feminin]

%%

%%

majeur(personne)=>-mineur(personne);

mineur(personne)=>majeur(personne);

pere (personne1, personne2)=>masculin(personne1);

pere (personne1, personne2)=>enfant(personne2, personne1)

%%
```

3.1.2 Commandes.

Successivement, les commandes suivantes sont effectuées :

```
cpd an

inco -s an
```

3.1.3 Résultat.

Le flag s ayant été invoqué, les résultats ont été redirigés vers le fichier "an.out" dont voici le contenu.

```
regle a traiter > majeur0(personne0)=>-mineur0(personne0)

incoherence detectee lors de l'application de la regle :

            majeur0(personne0)=>-mineur0(personne0)

avec le jeu de substition

[ 1]    majeur0 -----> [ 1]    majeur0
[ 2]  personne0 -----> [ 2]  personne0

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

regle a traiter > mineur0(personne0)=>majeur0(personne0)

incoherence detectee lors de l'application de la regle :

            majeur0(personne0)=>-mineur0(personne0)

avec le jeu de substition

[17]    majeur0 -----> [ 1]    majeur0
[18]  personne0 -----> [ 2]  personne0

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

regle a traiter > pere0(personne1,personne2)=>masculin0(personne1)

 pas d'incoherence detectee.....

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

regle a traiter > pere0(personne1,personne2)=>enfant0(personne2,personne1

cette regle n'est plus a traiter car elle a deja ete appliquee

-------------------------------------------------------------------------
```

3.2 Exemple I (b).

Nous avons effectué une seconde fois la recherche d'incohérence, mais cette fois en invoquant le flag m qui permet de visualiser l'évolution de la base de faits au fur et à mesure de l'exécution du programme.

le résultat suivant a été obtenu :

```
regle a traiter > majeur0(personne0)=>-mineur0(personne0)


Etat initial de la base de faits


majeur0(personne0)
mineur0(personne0)


application de la regle : majeur0(personne0)=>-mineur0(personne0)


majeur0(personne0)
mineur0(personne0)
-mineur0(personne0)


incoherence detectee lors de l'application de la regle :

      majeur0(personne0)=>-mineur0(personne0)

avec le jeu de substition

[ 1]    majeur0 -----> [ 1]    majeur0
[ 2]  personne0 -----> [ 2]  personne0


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

regle a traiter > mineur0(personne0)=>majeur0(personne0)


Etat initial de la base de faits


mineur0(personne0)


application de la regle : mineur0(personne0)=>majeur0(personne0)


mineur0(personne0)
majeur0(personne0)
```

```
application de la regle : majeur0(personne0)=>-mineur0(personne0)


mineur0(personne0)
majeur0(personne0)
-mineur0(personne0)


incoherence detectee lors de l'application de la regle :

        majeur0(personne0)=>-mineur0(personne0)

avec le jeu de substition

[17]    majeur0 -----> [ 1]    majeur0
[18]  personne0 -----> [ 2]  personne0

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

regle a traiter > pere0(personne1,personne2)=>masculin0(personne1)


Etat initial de la base de faits


pere0(personne1,personne2)


application de la regle : pere0(personne1,personne2)=>masculin0(personne1


pere0(personne1,personne2)
masculin0(personne1)


application de la regle : pere0(personne1,personne2)=>enfant0(personne2,p


pere0(personne1,personne2)
masculin0(personne1)
enfant0(personne2,personne1)


 pas d'incoherence detectee.....

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

regle a traiter > pere0(personne1,personne2)=>enfant0(personne2,personne1

cette regle n'est plus a traiter car elle a deja ete appliquee

-------------------------------------------------------------------------
```

3.3 Exemple II (a).

3.3.1 Déclaration des concepts et des règles.

Comme dans l'exemple précédent, la déclaration est effectuée via un editeur de texte. Le fichier ainsi créé se nomme "test".

```
%%

a, b, c, d, e, x, y, f, g, h, i, j, k, z

%%
%%

b=>h;
h=>-g;
c=>d;
b=>c;
x=>z;
a=>b;
e=>x;
x=>y;
a=>f;
b=>g;
x=>i;
k=>j
%%
```

3.3.2 Commandes.
Successivement, les commandes suivantes sont effectuées :

```
cpd test

inco -s test
```

3.3.3 Résultat.

Le flag s ayant été invoqué, les résultats ont été redirigés vers le fichier "test.out" dont voici le contenu.

```
regle a traiter > b0=>h0

incoherence detectee lors de l'application de la regle : b0=>g0 avec le
substition

[ 2]          b0 -----> [22]          b0

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

regle a traiter > h0=>-g0

cette regle n'est plus a traiter car elle a deja ete appliquee

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

regle a traiter > c0=>d0

incoherence detectee lors de l'application de la regle : b0=>g0 avec le
substition

[ 8]          b0 -----> [22]          b0

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

regle a traiter > b0=>c0

cette regle n'est plus a traiter car elle a deja ete appliquee

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

regle a traiter > d0=>e0

cette regle n'est plus a traiter car elle a deja ete appliquee

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

regle a traiter > x0=>z0
```

```
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------------
regle a traiter > a0=>b0
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------------
regle a traiter > e0=>x0
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------------
regle a traiter > x0=>y0
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------------
regle a traiter > a0=>f0
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------------
regle a traiter > b0=>g0
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------------
```

3.4 Exemple II (b).

Nous avons effectué une seconde fois la recherche d'incohérence, mais cette fois en invoquant le flag m qui permet de visualiser l'évolution de la base de faits au fur et à mesure de l'exécution du programme.

le résultat suivant a été obtenu :

```
regle a traiter > bO=>hO

Etat initial de la base de faits

bO
aO

application de la regle : bO=>hO

bO
aO
hO

application de la regle : hO=>-gO

bO
aO
hO
-gO

application de la regle : bO=>cO

bO
aO
hO
-gO
cO

application de la regle : aO=>bO
mais le fait y est deja
application de la regle : aO=>fO

bO
aO
```

```
h0
-g0
c0
f0

application de la regle : b0=>g0

b0
a0
h0
-g0
c0
f0
g0

incoherence detectee lors de l'application de la regle : b0=>g0
avec le jeu de substition

[ 2]          b0 -----> [22]          b0

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

regle a traiter > h0=>-g0

cette regle n'est plus a traiter car elle a deja ete appliquee

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

regle a traiter > c0=>d0

Etat initial de la base de faits

c0
b0
a0

application de la regle : b0=>h0

c0
b0
a0
h0

application de la regle : h0=>-g0

c0
```

```
bO
aO
hO
-gO

application de la regle : cO=>dO

cO
bO
aO
hO
-gO
dO

application de la regle : bO=>cO
mais le fait y est deja
application de la regle : dO=>eO

cO
bO
aO
hO
-gO
dO
eO

application de la regle : aO=>bO
mais le fait y est deja
application de la regle : eO=>xO

cO
bO
aO
hO
-gO
dO
eO
xO

application de la regle : xO=>yO

cO
bO
aO
hO
-gO
dO
eO
```

```
xO
yO

application de la regle : aO=>fO

cO
bO
aO
hO
-gO
dO
eO
xO
yO
fO

application de la regle : bO=>gO

cO
bO
aO
hO
-gO
dO
eO
xO
yO
fO
gO

incoherence detectee lors de l'application de la regle : bO=>gO
avec le jeu de substition

[ 8]          bO -----> [22]          bO

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

regle a traiter > bO=>cO

cette regle n'est plus a traiter car elle a deja ete appliquee

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

regle a traiter > dO=>eO

cette regle n'est plus a traiter car elle a deja ete appliquee

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

regle a traiter > xO=>zO
```

```
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------
regle a traiter > a0=>b0
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------
regle a traiter > e0=>x0
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------
regle a traiter > x0=>y0
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------
regle a traiter > a0=>f0
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------
regle a traiter > b0=>g0
cette regle n'est plus a traiter car elle a deja ete appliquee
-------------------------------------------------------------------
```

PARTIE III.4 : TEXTE DU PROGRAMME.

```
/-------------------------------------------------------------------/
  Ce fichier contient les ordres de compilation et les liens entre
  les fichiers pour la construction du programme de détection
  d'incohérence.
/-------------------------------------------------------------------/

inco : inco.o acq.o delta.o erreur.o liste.o outils.o \
           pile.o recherche.o util.o visu.o trt.o
       cc -g inco.o acq.o delta.o erreur.o liste.o outils.o \
           pile.o recherche.o util.o visu.o trt.o -o inco
inco.o : const.c const2.c type.c decl.c ext.c inco.c
       cc -c -g inco.c
acq.o : const.c const2.c type.c decl.c ext.c acq.c
       cc -c -g acq.c
delta.o : const.c const2.c type.c decl.c ext.c delta.c
       cc -c -g delta.c
erreur.o :  const.c const2.c type.c decl.c ext.c erreur.c
       cc -c -g erreur.c
liste.o :  const.c const2.c type.c decl.c ext.c liste.c
       cc -c -g liste.c
outils.o :  const.c const2.c type.c decl.c ext.c outils.c
       cc -c -g outils.c
pile.o :  const.c const2.c type.c decl.c ext.c pile.c
       cc -c -g pile.c
trt.o :  const.c const2.c type.c decl.c ext.c trt.c
       cc -c -g trt.c
recherche.o :  const.c const2.c type.c decl.c ext.c recherche.c
       cc -c -g recherche.c
util.o :  const.c const2.c type.c decl.c ext.c util.c
       cc -c -g util.c
visu.o :  const.c const2.c type.c decl.c ext.c visu.c
       cc -c -g visu.c
```

```
/*****************************************************************/
/* CE FICHIER CONTIENT LA DEFINITION DES CONSTANTES POUR LE      */
/* RAISONNEUR. ON L'INCLUERA PAR LA DIRECTIVE #INCLUDE DANS      */
/* LE FICHIER CONTENANT LE PROGRAMME PRINCIPAL.                  */
/*****************************************************************/

#include <stdio.h>
#include <ctype.h>
#include <strings.h>

                        /*_____________________________________ */
#define nil   0         /* Pointeur vide                      */
#define VIDE  0         /* Effet de bord pour un noeud        */
#define VRAI  1         /* Booleen = vrai                     */
#define FAUX  0         /* Booleen = faux                     */
#define PLUS  1         /* Booleen de negation                */
#define MOINS -1        /* Booleen de negation                */
                        /*_____________________________________ */
                        /*_____________________________________ */
#define ID_CODE 1       /*                                    */
#define CH_CODE 2       /*          Codification              */
#define ET_CODE 3       /*   des noeuds d'une formule.        */
#define OU_CODE 4       /*_____________________________________ */
                        /*_____________________________________ */
#define T_LIST 16       /*Taille d'un enregistremt liste      */
#define T_LLIST 8       /*Taille d'un enregistremt lliste     */
#define T_LEX  1000     /*Taille du lexique                   */
#define LC      20      /*Longueur d'une chaine               */
#define T_NOEUD 100     /*Taille de la table des noeuds.      */
#define T_CHAINE 10     /*Taille de la table des chaines.     */
#define T_TRACE 100     /*Taille de la trace.                 */
#define T_COND 50       /*Taille des conditions.              */
#define T_BD 10         /*Taille de la base de donnees.       */
#define T_RC 5          /*Taille de la base des regles.       */
#define T_RS 500        /*Taille de la base de renseign.      */
#define T_FAIT 1000     /*Taille de la base des faits.        */
#define T_EXCL 100      /*Taille de la pile des exclus.       */
#define T_SYST 5000     /*Taille de la pile des systemes.     */
#define T_N_BIT (((T_NOEUD*(T_NOEUD-1))/2)+7)/8/*           */
                        /*_____________________________________*/
                        /*_____________________________________*/
#define MAX_SC 100      /*Score maximal.                      */
#define MIN_SC 0        /*Score minimal.                      */
                        /*_____________________________________ */
                        /*_____________________________________ */
#define F_ETOILE -1     /*                                    */
#define _DELETE 0       /*                                   */
#define F_BD     1      /*   Differents etats pour            */
#define F_ENONCE 2      /*        la Base des                 */
#define F_DEDUIT 3      /*           Faits.                   */
                        /*_____________________________________ */
                        /*_____________________________________ */
#define UNIFIE  0       /* Indice du tableau equ[] et         */
#define EXCLU   1       /* parametre des fonctions.           */
                        /*_____________________________________ */
#define MARQX   10      /* Valeur de marquage de formules     */
                        /*_____________________________________ */

/*_____________________________________________________________ */
/*_____________________les pseudo-fonctions_____________________ */
/*_____________________________________________________________ */

#define MOP(c)  if (Mop == VRAI) fprintf(yytrace,"\n===%s===",c)
#define ET(x,y) ((x) == VRAI)?(y):FAUX
```

```
#define OU(x,y) ((x) == FAUX)?(y):VRAI
#define MAX(x,y) ((x) > (y))?(x):(y)
#define MIN(x,y) ((x) < (y))?(x):(y)
#define NEGATIF(x) ((x) < 0)?VRAI:FAUX
#define POSITIF(x) ((x) > 0)?VRAI:FAUX
#define ZERO(x) ((x) == 0)?VRAI:FAUX

/*o=o=o=o=o=o=oo=o=o=o=o=o=o=o= FIN o=o=o=o==o=o=o=o=o=o=o=o=o=o=o=o=o=o=*/
```

```
#define  MAX_APP 100          /* nombre maximum d'unificateurs    */

#define  APPLICABLE 1         /* indicateur d'applicabilite d'    */
                              /* une regle.                       */
#define  NAPPLICABLE 0        /* indicateur de non applicabilite  */
                              /* d'une regle.                     */
#define  SELECT 1             /* indicateur de selection d'une    */
                              /* regle pour application.          */
#define  NSELECT 0            /* indicateur de non selection d'   */
                              /* une regle pour application.      */
#define  ATRAIT 1             /* indicateur de traitement d'une   */
                              /* regle -reste a traiter-.         */
#define  NATRAIT 0            /* indicateur de traitement d'une   */
                              /* regle -n'est plus a traiter.     */
#define  NOTPRE -1            /* valeur renvoyee quand un objet   */
                              /* recherche dans une liste n'y est */
                              /* pas trouve.                      */
```

```
/*************************************************************************/
/* CE FICHIER CONTIENT LES DEFINITIONS DE TYPES QU'ON UTILISE DANS   */
/* LE RAISONNEUR.                                                        */
/* ON L'INCLUERA DANS LE SOURCE DE CHAQUE FICHIER QU'ON COMPILE      */
/* SEPAREMENT PAR :                                                      */
/* #INCLUDE "type.c"                                                     */
/*************************************************************************/

/*_____________________________________________________________________*/
/*____________________Definition de TYPE_______________________________*/
/*_____________________________________________________________________*/

struct doublet
        {struct formule
                  {int code, noeud, indic, neg ,reseau,pi;
                   struct doublet *rev;
                   union cas
                          {struct id
                                   {struct doublet *arg, *dif;
                                    int num;
                                   } i_id;
                           struct ch
                                   {
                                    char c[20];
                                   } i_ch;
                           struct opbin
                                   {struct doublet *f;
                                   } i_opbin;
                           }info;
                   } *lien;
          struct doublet *suc;
         };

                        /* * * * */

struct liste
       {int val, supp;
        struct liste *suc;
       };

                        /* * * * */

struct lliste
       {struct liste  *liste;
        struct lliste *suc;
       };

                        /* * * * */

struct lex
       {char libelle[LC];
        int ordre;
        struct liste *isa, *class, *arg, *gra, *form;
       };
                        /* * * * */

struct noeud
       {int code, signe, reseau, suffixe, eqrp, eqls;
        struct liste *chav, *charr;
       };
                        /* * * * */

struct cond
       {int test, a1, a2, im1, im2;};
```

```
                        /* * * * * */
struct reec
       {int pre, post;};

                        /* * * * * */
struct rens
       {int req, cond;};

                        /* * * * * */

struct fait
       {int acces, status, score;
        struct liste *orig,*regle;
       };

                        /* * * * * */

struct syst
       {int ed, n1, n2, frontiere;
        struct liste *action;
       };

                        /* * * * * */

struct pile
      {int Ppile[T_NOEUD];
       int Pile[T_NOEUD];
       int ppile;
      };

                        /* * * * * */
struct appar
      {int regle;
       struct liste *liste;
      };

                        /* * * * * */

struct tab_pointer_l { struct liste *tab[T_NOEUD];
                       int taille;
                       };

                        /* * * * * */

struct tab_pointer_ll { struct lliste *tab[T_NOEUD];
                        int taille;
                       };

                        /* * * * * */


/*_____________________________________________________________________*/
/*_____________________________F_I_N___________________________________*/
/*_____________________________________________________________________*/
```

```
/*************************************************************************/
/* CE FICHIER CONTIENT LA DECLARATION DE TOUTES LES VARIABLES GLOBALES*/
/* DU PROGRAMME "RAISON". ON L'INCLUERA AVEC LA DIRECTIVE #INCLUDE    */
/* DANS LE FICHIER CONTENANT LE PROGRAMME PRINCIPAL.                  */
/*************************************************************************/

                                        /*_____________________________________*/
int Mop;                                /* Indicateur de mise au point.       */
int Sil;                                /* le silencieux.                     */
int Msg;                                /* les messages.                      */
int Vis;                                /* la visualisation des tables.       */
char Entree[35];                        /* Nom du fichier en parametre.       */
char Fichier[45];                       /* VT pour la concatenation.          */
                                        /*_____________________________________*/

/*_____________________________________________________________________________*/
/*_____________________Variables globales_____________________________________*/
/*_____________________________________________________________________________*/

                                        /*_____________________________________*/
int Nblex;                              /* Nb d'unites lexicales.             */
int Nbnoeud;                            /* Nb de noeuds                       */
int Nbchaine;                           /* Nb de chaine.                      */
int Nbcond;                             /* Nb de conditions.                  */
int Nbbd;                               /* Nb de formules ds la bd.           */
int Nbrc;                               /* Nb d' expressions dans rc.         */
int Nbrs;                               /* Nb de renseignement.               */
int Nbf;                                /* Nb de faits.                       */
int Nbrg;                               /* Nb de regles appareillees.         */
int Nbtr;                               /* Nb de souvenirs dans la trace.     */
int Ptsyst = 0;                         /* Pointeur de pile P_syst.           */
char Tchaine[T_CHAINE][LC];             /* Table des chaines.                 */
struct lex Tlex[T_LEX]   ;              /* Le lexique.                        */
struct noeud Tnoeud[T_NOEUD];           /* Description des noeuds.            */
struct cond Tcond[T_COND];              /* Description des conditions.        */
int Tbd[T_BD];                          /* La base de donnees.                */
struct appar Trace[T_TRACE];            /* La table des appareillements.      */
struct reec Trc[T_RC];                  /* La base de regles.                 */
struct rens Trs[T_RS];                  /* La base de renseignement.          */
struct fait Tfait[T_FAIT];              /* La base des faits(c historique)    */
char Texcl[T_N_BIT];                    /* La matrice de bits des exclus.     */
struct syst Psyst[T_SYST];              /* Pile des unificateurs.             */
struct formule *S_but,                  /*         Resultats                  */
               *S_fait,                 /*       de l'analyseur.              */
               *S_quest;                /*     cf gram.requete                */
int I_but,                              /*         Resultats                  */
    I_fait,                             /*        de l'interface              */
    I_quest,                            /*       saisie-raisonneur.           */
    I_qx;                               /*                                    */
int Err;                                /* Booleen de presence d'erreur.      */
                                        /*_____________________________________*/
```

```
int a_traiter[T_RC];        /* indicateur de traitement        */
                            /* des regles de reecritures       */

struct liste
        *Tapp[T_RC];        /* Table des listes d'unifica-     */
                            /* teurs. A chaque regle on est    */
                            /* une liste d'unificateur.        */
```

```
/*************************************************************************/
/* CE FICHIER CONTIENT LA DECLARATION DES FICHIERS QUI VONT ETRE     */
/* UTILLISES DANS LE PROGRAMME "RAISON". ON L'INCLUERA PAR LA        */
/* DIRECTIVE #INCLUDE DANS LE FICHIER CONTENANT LE PROGRAMME         */
/* PRINCIPAL.                                                        */
/*************************************************************************/

/*_____________________________________________________________*/
/*____________________Declaration des fichiers_________________*/
/*_____________________________________________________________*/

extern FILE *yyin;
extern FILE *yyout;
FILE *yycon = stdin;
FILE *yytrace = stdout;
                    /*_____________________________________________*/
FILE *pnoeud,       /*Repres d'un noeud d'une formule     */
     *pchaine,      /*Table-Fichier des chaines utilisees*/
     *pisa,         /*Relations isa pour les concepts     */
     *pclass,       /*Relations de classe -id-            */
     *parg,         /*Listes des arguments d'un concept   */
     *plex,         /*Lexique resultat                    */
     *plien,        /*Lien pour chaque noeud.             */
     *pcond,        /*Noeud des conditionnelles           */
     *pbd,          /*Ensemble des formules de la bd      */
     *prc,          /* --- id ---                   rc     */
     *prs;          /* --- id ---                   rs     */
                    /*_____________________________________________*/

/*o=o=o=o=o=o=o=o=o=o=o=o=o=o=o=F=I=N=o=o=o=o=o=o=o=o=o=o=o=o=o=o=o=o*/
```

```
extern struct noeud Tnoeud[];

extern struct lex Tlex[];

extern struct appar list_appar[];

extern struct appar Trace[];

extern struct reec Trc[];

extern struct fait Tfait[];

extern char Tchaine[][LC];

extern struct cond Tcond[];

extern char Texcl[];

extern struct rens Trs[];

extern int Tbd[];

extern int a_traiter[];

extern struct liste *Tapp[],
                    *alloc_list(),
                    *ajq_list(),
                    *ajt_list(),
                    *in_list0(),
                    *in_list1(),
                    *in_list2(),
                    *in_list3();

extern struct lliste *ajq_llist(),
                     *ajt_llist(),
                     *alloc_llist();
extern int Pile[],
           PPile[];

extern char Entree[],
            Fichier[];

extern FILE *yytrace;

extern int R1[],R2[],R3[],Nbbd,Ptsyst,
           Nblex,Nbf,Nbchaine,Nbrs,Nbrg,
           Nbrc,Nbnoeud,Nb_pl,Nbcond,Nbtr,
           Sil,Vis,Msg,Mop;

extern struct syst Psyst[];

extern struct formule *S_but,
                      *S_fait,
                      *S_quest;

extern int I_but,
       I_fait,
       I_quest,
       I_qx;
```

```
/*************************************************************************/
/* CE FICHIER CONTIENT LES FONCTIONS DE GESTION  D'UNE MATRICE     */
/* CARREE DE BITS ,SYMETRIQUE / DIAG. PRINCIPALE.                  */
/* CETTE MATRICE EST UN PARAMETRE. ELLE CONTIENT LES NOEUDS QUI    */
/* SONT MUTUELLEMENT EXCLUS.                                       */
/* LA TAILLE = ((T_NOEUD*(T_NOEUD-1)/2)+7)/8   )                   */
/* CE FICHIER SERA COMPILE SEPAREMENT.                             */
/*************************************************************************/

#include <stdio.h>

/*************************************************************************/
/*                      LES EXTERNES.                                    */
/*************************************************************************/

                          /*_____________________________________________*/
extern int Mop;           /* Indicateur de mise au point.                */
extern FILE *yytrace;     /* Fichier de trace.                           */
                          /*_____________________________________________*/

/*************************************************************************/
/*                      LES CONSTANTES.                                  */
/*************************************************************************/

#define VRAI 1
#define FAUX 0
#define MOP(c) if (Mop == VRAI) fprintf(yytrace,"\n===%s===\n",c)


/*************************************************************************/
/*                      LES GLOBAUX.                                     */
/*************************************************************************/

                          /*_____________________________________________*/
char Masque[8]={0x80,     /*                                             */
                0x40,     /*                Masque.                      */
                0x20,     /*                                             */
                0x10,     /* Les bits sont numerotes de la facon         */
                0x08,     /* suivante:                                   */
                0x04,     /*  _______________                            */
                0x02,     /* |0|1|2|3|4|5|6|7|                           */
                0x01};    /*  _______________                            */
                          /*_____________________________________________*/

/*************************************************************************/
/*                      LES FONCTIONS.                                   */
/*************************************************************************/

/*************************************************************************/
/* NIVEAU OCTET:                                                         */
/*              -set_x();                                                */
/*              -reset_x();                                              */
/*              -test_x();                                               */
/* NIVEAU MATRICE:                                                       */
/*              -ad_mat();                                               */
/*              -set_mat();                                              */
/*              -reset_mat();                                            */
/*              -test_mat();                                             */
/*              -init_mat();                                             */
/*              -visu_mat();                                             */
/*************************************************************************/

/*************************************************************************/
/*                     *NIVEAU OCTET.*                                   */
/*************************************************************************/
```

```
/*_____________________________________________________________________*/
/*_____________________Set dans un octet_____________________________*/
/*_____________________________________________________________________*/

set_x(ad,nbit)              char *ad;
                            short nbit;

                        /*_____________________________________________*/
                        /* Set du nbit-ieme bit de l'octet d'adresse */
                        /* ad.                                         */
                        /*_____________________________________________*/

{
 MOP("set_x");
 (*ad) = (*ad)|Masque[nbit];
 return;
}

/*_____________________________________________________________________*/
/*_____________________Reset dans un octet___________________________*/
/*_____________________________________________________________________*/

reset_x(ad,nbit)            char *ad;
                            short nbit;

                        /*_____________________________________________*/
                        /* Reset du nbit-ieme bit de l'octet           */
                        /* d'adresse ad.                               */
                        /*_____________________________________________*/

{
 MOP("reset_x");
 (*ad) = (*ad)&~(Masque[nbit]);
 return;
}

/*_____________________________________________________________________*/
/*_____________________Test dans un octet____________________________*/
/*_____________________________________________________________________*/

int test_x(ad,nbit)         char *ad;
                            short nbit;

                        /*_____________________________________________*/
                        /* Test du nbit-ieme bit de l'octet d'adresse*/
                        /* ad.                                         */
                        /*_____________________________________________*/

{char c;

 MOP("test_x");
 c = (*ad)&Masque[nbit];
 if (c == 0x00) return(FAUX);
 return(VRAI);
}

/*********************************************************************/
/*                      *NIVEAU MATRICE. *                           */
/*********************************************************************/

/*_____________________________________________________________________*/
/*_____________________Indices->adresse______________________________*/
/*_____________________________________________________________________*/

ad_mat(i,j,ind,nbit)        int i,j;
```

```
                        int *ind;
                        short *nbit;

                    /*_____________________________________________*/
                    /* Calcul de l'indice  de l'octet et du depl. */
                    /* relatif correspondant aux indices i, j.    */
                    /* Invariant: i#j.                            */
                    /*_____________________________________________*/

{int k, b;

 MOP("ad_mat");
 if (i < j)
    {k = i; i = j; j = k;}
 b = ((i * (i - 1)) / 2) + j;
 (*ind) = b / 8;
 (*nbit) = b % 8;
 return;
}

/*_____________________________________________________________________*/
/*_____________________Set matrice_____________________________________*/
/*_____________________________________________________________________*/

set_mat(texcl, i, j)     char *texcl;
                         int i, j;

                    /*_____________________________________________*/
                    /* Le concept i # du concept j.               */
                    /*_____________________________________________*/

{int ind;
 short nbit;
 char *ad;

 MOP("set_mat");
 if (i == j) return;
 ad_mat(i, j, &ind, &nbit);
 ad = texcl + ind;
 set_x(ad, nbit);
 return;
}

/*_____________________________________________________________________*/
/*_____________________reset matrice___________________________________*/
/*_____________________________________________________________________*/

reset_mat(texcl, i, j)       char *texcl;
                             int i, j;

                    /*_____________________________________________*/
                    /* Le concept i ? du concept j.               */
                    /*_____________________________________________*/

{int ind;
 short nbit;
 char *ad;

 MOP("reset_mat");
 if (i == j) return;
 ad_mat(i, j, &ind, &nbit);
 ad = texcl + ind;
 reset_x(ad, nbit);
 return;
}
```

```
/*_____________________________________________________________________*/
/*_____________________test matrice_____________________________________*/
/*_____________________________________________________________________*/

int test_mat(texcl, i, j)       char *texcl;
                                int i, j;

                    /*_____________________________________________*/
                    /* Le concept i # du concept j ?               */
                    /*_____________________________________________*/

{int ind;
 short nbit;
 char *ad;

 MOP("test_mat");
 if (i == j) return(FAUX);
 ad_mat(i, j, &ind, &nbit);
 ad = texcl + ind;
 return(test_x(ad, nbit));
}

/*_____________________________________________________________________*/
/*_____________________Init matrice____________________________________*/
/*_____________________________________________________________________*/

init_mat(texcl, t)          char *texcl;
                            int t;

                  /*_____________________________________________*/
                  /* Initialisation d'une matrice txt a zero.    */
                  /*_____________________________________________*/

{int nb_oct, i;

 MOP("init_mat");
 nb_oct = (((t * (t - 1)) / 2) + 7) / 8;
 for (i = 0;
      i <= nb_oct;
      i++)
      *(texcl + i) = 0x00;
 return;
}

/*_____________________________________________________________________*/
/*_____________________Copie de matrices_______________________________*/
/*_____________________________________________________________________*/

cp_mat(texcl1, texcl2, t)          char *texcl1, *texcl2;
                                   int t;

                  /*_____________________________________________*/
                  /* Copie d'une matrice texcl2 de taille t      */
                  /* dans une matrice texcl1.                    */
                  /* cp_mat(dest, source, taille)                */
                  /*_____________________________________________*/

{int t_real;

 MOP("cp_mat");
 t_real = (((t * (t - 1)) / 2) + 7) / 8;
 write(texcl1, texcl2, t);
 return;
}

/*_____________________________________________________________________*/
```

```
/*_____________________Visualisation matrice_____________________________*/
/*_____________________________________________________________________*/

visu_mat(texcl,t)          char *texcl;
                           int t;

                    /*_____________________________________________________*/
                    /* Visualisation d'une matrice t x t .                 */
                    /*_____________________________________________________*/

{int i,j,mem;

 MOP("visu_mat");
 mem = Mop;
 Mop = FAUX;
 for(i = 0;
     i < t;
     i++)
    {for (j = 0;
          j < i;
          j++)
          fprintf(yytrace,"%1d",test_mat(i,j));
     fprintf(yytrace,"\n");
    }
 Mop = mem;
 return;
}

/*********************************************************************/
/*                              FIN                                  */
/*********************************************************************/
```

```
#include <stdio.h>

extern FILE *yytrace;

erreur(i,s)     int i;
                char *s;

{
  switch (i)
        {case 0 : break;
         case 1 : fprintf(yytrace,"\n%s\n","***Trop de concepts! Adieu!");
                  exit(0);
                  break;
         case 2 : fprintf(yytrace,"\n%s\n","***Parametres incorrects. Adieu!
                  exit(0);
                  break;
         case 3 : fprintf(yytrace,"\n***Fichier %s manquant. Adieu!\n",s);
                  exit(0);
                  break;
         case 4 : fprintf(yytrace,"\n***Pile pleine: %s . Adieu!\n",s);
                  exit(0);
                  break;
         case 5 : fprintf(yytrace,"\n***Pile vide: %s . Adieu!\n",s);
                  exit(0);
                  break;
         case 6 : fprintf(yytrace,"\n***Cas non prevu: %s . Adieu!\n",s);
                  exit(0);
                  break;
         case 7 : fprintf(yytrace,"\n***incoherence detectee . Adieu!\n");
                  exit(0);
                  break;
         case 8 : fprintf(yytrace,"\n***Trace pleine. Adieu!\n");
                  exit(0);
                  break;
         case 9 : fprintf(yytrace,"\n***choix non prevu \n");
                  break;
         default: break;
        }
  return;
 }
```

```
# include <stdio.h>
# include "const.c"
# include "const2.c"
# include "type.c"
# include "ext.c"

/*************************************************************************/
/* ce fichier contient un certain nombre de procedures manipulant        */
/* les listes et les listes de listes.                                   */
/*************************************************************************/



/*************************************************************************/
/* manipulations de listes d'entiers.(struct liste).                     */
/*************************************************************************/

/*_____________________________________________________________________*/
/*_____________________Allocation d'un element_____________________*/
/*_____________________________________________________________________*/

struct liste *alloc_list(i,j)   int i,j;

                    /*_____________________________________________*/
                    /* allocation et initiallisation d'un         */
                    /* dans une liste de type "liste"             */
                    /*_____________________________________________*/

{struct liste *resul;

 MOP("alloc_list");
 resul = (struct liste *)malloc(T_LIST);
 resul->val = i;
 resul->supp = j;
 return(resul);
}

/*_____________________________________________________________________*/
/*_____________________Adjonction en queue_____________________*/
/*_____________________________________________________________________*/

struct liste *ajq_list(liste1,liste2)   struct liste *liste1,
                                                    *liste2;

                    /*_____________________________________________*/
                    /* adjonction en queue d'une liste ou atome*/
                    /* a une liste.                               */
                    /*_____________________________________________*/

{struct liste *vt;

 MOP("ajq_list");
 if (liste1 == nil)
    {return (liste2);
    }
 for (vt = liste1;
      vt->suc != nil;
      vt = vt->suc);
 vt->suc = liste2;
 return(liste1);
}

/*_____________________________________________________________________*/
/*_____________________Adjonction en tete_____________________*/
```

```
/*_____________________________________________________________________*/

struct liste *ajt_list(liste1,liste2)   struct liste *liste1,
                                                    *liste2;

                        /*_____________________________________________*/
                        /* adjonction en tete d'un ATOME a une        */
                        /* liste.                                      */
                        /*_____________________________________________*/

{struct liste *vt;

 MOP("ajt_list");
 if (liste1 == nil)
    {return (liste2);
    }
 liste1->suc = liste2;
 return(liste1);
}

/*---------------------------------------------------------------------*/

/*********************************************************************/
/* manipulations de listes de listes d'entiers.(struct lliste).      */
/*********************************************************************/

/*_____________________________________________________________________*/
/*_____________________________Allocation d'un element_________________*/
/*_____________________________________________________________________*/

struct lliste *alloc_llist(liste) struct liste *liste;

                        /*_____________________________________________*/
                        /* allocation et initiallisation d'un         */
                        /* dans une liste de type "liste"             */
                        /*_____________________________________________*/

{struct lliste *resul;

 MOP("alloc_llist");
 resul = (struct lliste *)malloc(T_LLIST);
 resul->liste = liste;
 return(resul);
}

/*_____________________________________________________________________*/
/*_____________________________Adjonction en queue_____________________*/
/*_____________________________________________________________________*/

struct lliste *ajq_llist(lliste1,lliste2) struct lliste *lliste1,
                                                       *lliste2;

                        /*_____________________________________________*/
                        /* adjonction en queue d'une lliste ou atome*/
                        /* a une lliste.                               */
                        /*_____________________________________________*/

{struct lliste *vt;

 MOP("ajq_llist");
 if (lliste1 == nil)
    {return (lliste2);
    }
 for (vt = lliste1;
      vt->suc != nil;
      vt = vt->suc);
```

```
 vt->suc = lliste2;
 return(lliste1);
}

/*_____________________________________________________________*/
/*_____________________Adjonction en tete_____________________*/
/*_____________________________________________________________*/

struct lliste *ajt_llist(lliste1,lliste2) struct lliste *lliste1,
                                                 *lliste2;

                        /*_____________________________________*/
                        /* adjonction en tete d'un ATOME a une */
                        /* lliste.                              */
                        /*_____________________________________*/

{struct lliste *vt;

 MOP("ajt_llist");
 if (lliste1 == nil)
    {return (lliste2);
    }
 lliste1->suc = lliste2;
 return(lliste1);
}

/*-------------------------------------------------------------*/
```

```
#include <stdio.h>
#include "const.c"
#include "const2.c"
#include "type.c"
#include "ext.c"

/*-------------------------------------------------------------------*/
/*                                                                   */
/*                  FONCTION DE MANIPULATION DES PILES               */
/*                                                                   */
/*-------------------------------------------------------------------*/


vide_pile(P) struct pile *P;

/* fonction dont le but est d'initialiser la pile */
/* P. Le pointeur de la pile pointera vers l'     */
/* element 0 de la pile, et tous les elements de  */
/* l'indicateur de presence dans la pile seront   */
/* mis a FAUX.                                    */

{ int j;

  MOP("vide_pile");
  for (j=0; j<T_NOEUD; j++) P->Ppile[j]=FAUX;
  P->ppile=0;
}

/*-------------------------------------------------------------------*/

push_pile(P,x) struct pile *P; int x;

/* fonction dont le but est de mettre l'entier x  */
/* au sommet de la pile P. Si la pile P est       */
/* pleine, alors un message le signalant sera     */
/* affiche.                                       */

{
  MOP("push_pile");
  if (P->ppile>=T_NOEUD) erreur(4);
  P->Pile[P->ppile]=x;
  P->ppile++;
}

/*-------------------------------------------------------------------*/

pop_pile(P) struct pile *P;

/* fonction dont l'objectif est de renvoyer et de */
/* depiler l'entier se trouvant au sommet de la   */
/* pile P. Si cette pile est vide, alors un       */
/* message le signalant sera.                     */

{
 MOP("pop_pile");
 if (P->ppile==0) erreur(5);
 P->ppile--;
 return(P->Pile[P->ppile]);
}

/*-------------------------------------------------------------------*/

cp_lst_pile_onw(P,l) struct pile *P; struct liste *l;
```

```
/* fonction dont le but est de copier la liste l  */
/* au sommet de la pile. Seuls les elements de la */
/* liste ne se trouvant pas encore dans la pile   */
/* y seront ajoutes. Si la pile se trouve         */
/* etre pleine en cours de copie, un message      */
/* le signalant sera affiche.                     */

{ struct liste *p;

  MOP("cp_lst_pile_onw");
  for(p=l;
      p!=nil;
      p=p->suc)
      if (!in_pile(P,p->val))  push_pile(P,p->val),
                               P->Ppile[p->val]=VRAI;
}

/*----------------------------------------------------------------------*/

cp_lst_pile_all(P,l) struct pile *P; struct liste *l;

/* fonction dont le but est de copier la liste l       */
/* au sommet de la pile. Tous les elements de la liste */
/* y seront ajoutes. Si la pile se trouve              */
/* etre pleine en cours de copie, un message           */
/* le signalant sera affiche.                          */

{ struct liste *p;

  MOP("cp_lst_pile_all");
  for(p=l;
      p!=nil;
      p=p->suc)
      {  push_pile(P,p->val),
         P->Ppile[p->val]=VRAI;
      }
}

/*----------------------------------------------------------------------*/

cp_pile_liste(l,P) struct liste *l; struct pile *P;

/* fonction dont le but est d'ajouter le contenu */
/* de la pile a la fin de la liste l.            */

{ struct liste *p;
  int i;

  MOP("cp_pile_liste");
  p=nil;
  for (i=0;
       i<P->ppile;
       i++)
       p=ajq_list(p,alloc_list(P->Pile[i],P->Pile[i]));
}

/*----------------------------------------------------------------------*/

in_pile(P,x)  struct pile *P; int x;

/* fonction dont le but est de determiner si  */
/* l'entier x se trouve dans la pile P.       */
/* Si oui, la valeur VRAI sera renvoyee,      */
/* sinon FAUX sera renvoye.                   */

{
```

```
    MOP("in_pile");
    return (P->Ppile[x]);
}

/*-------------------------------------------------------------------------*/

visu_pile(P) struct pile *P;

{int j;

 MOP("visu_pile");
 fprintf(" taille de la pile %d\n\n",P->ppile);
 for (j=P->ppile - 1;
      j>=0;
      j--)
      fprintf("%d\n",P->Pile[j]);
}
```

```
# include <stdio.h>
# include "const.c"
# include "type.c"
# include "ext.c"

/*************************************************************************/
/* CE FICHIER CONTIENT UN CERTAIN NOMBRE D'UTILITAIRES CONCERNANTS       */
/* LES PRINCIPALES STRUCTURES UTILISEES DANS LE RAISONNEUR.              */
/* CE FICHIER SERA INCLU DANS LE PROGRAMME PRINCIPAL PAR:                */
/* #INCLUDE "util.c"                                                     */
/*************************************************************************/

/*_____________________________________________________________________*/
/*_____________________Transformation de doublets_____________________*/
/*_____________________________________________________________________*/

struct liste *db_list(db)       struct doublet *db;

                    /*_____________________________________________*/
                    /* Transformation d'une liste de doublets      */
                    /* provenant de la saisie, en liste.           */
                    /*_____________________________________________*/

{struct liste *vl;
 struct doublet *vd;

 MOP("db_list");
 for (vl = nil, vd = db;
      vd != nil;
      vd = vd->suc)
     {vl = ajq_list(vl,alloc_list(Nbnoeud+(db->lien->noeud)));}
 return(vl);
}

/*_____________________________________________________________________*/
/*_____________________synchronisation_____________________*/
/*_____________________________________________________________________*/

int synch_list(l1,l2)           struct liste *l1, *l2;

                    /*_____________________________________________*/
                    /* Synchronisation de la liste l1 sur la liste l2*/
                    /* Cette synchronisation se fait sur le champ    */
                    /* supp. Le resultat de cette fonction est le    */
                    /* rang de l'atome x de l1 tq x->supp == l2->supp*/
                    /*_____________________________________________*/

{struct liste *vl;
 int cp;

 MOP("synch_list");
 cp = 0;
 if (l2 == nil) return(cp);
 for(vl = l1;
     vl != nil;
     vl = vl->suc, cp++)
     if (vl->supp == l2->supp) break;
 return(cp);
}


/*************************************************************************/
/* MANIPULATION DU RESEAU SEMANTIQUE DES CONCEPTS.                       */
/*************************************************************************/
```

```
/*_____________________________________________________________________*/
/*____________________ordonner les concepts_____________________________*/
/*_____________________________________________________________________*/

ord_cpt()
                    /*_____________________________________________________*/
                    /* Dans le but d'accelerer certains traitements       */
                    /* on va exploiter l'ordre partiel qui existe         */
                    /* dans le graphe des concepts.                       */
                    /* cette fonction ordonne ces concepts avec un        */
                    /* numero croissant.                                  */
                    /*_____________________________________________________*/

{int cpt, ordre;
 struct liste *v10, *tete;

 MOP("ord_cpt");
 for (cpt = 0;
      cpt < Nblex;
      cpt++)
     Tlex[cpt].ordre = -1;
 tete = alloc_list(0);
 for(ordre = 0;
     tete != nil;
     tete = tete->suc)
    {Tlex[tete->val].ordre = ordre++;
     for(v10 = Tlex[tete->val].class;
         v10 != nil;
         v10 = v10->suc)
         if (Tlex[v10->val].ordre < ordre)
             tete = ajq_list(tete,alloc_list(v10->val));
    }
}

/*_____________________________________________________________________*/
/*____________________relation isa_____________________________________*/
/*_____________________________________________________________________*/

int isa_cpt(c1,c2)      int c1,c2;

                    /*_____________________________________________________*/
                    /*Fonction a resultat booleen testant si l'elemt      */
                    /*c1 "est un" c2 dans le reseau des concepts.         */
                    /*_____________________________________________________*/

{struct liste *v10, *v11, *v12,
              *tfile, *qfile;/*tete et queue de file*/

 MOP("isa_cpt");
 if (Tlex[c1].ordre < Tlex[c2].ordre) return(FAUX);
 if (c1 == 0)
     {if (Mop == VRAI) fprintf(yytrace,"###-oui\n");
      return(VRAI);
     }
 if (c2 == 0)
     {if (Mop == VRAI) fprintf(yytrace,"###-non\n");
      return(FAUX);
     }
 for (tfile = qfile = alloc_list(c2);
      tfile != nil;
      v11 = tfile, tfile = tfile->suc, free(v11))
     {/*enfiler les fils de tfile*/
       for (v10 = Tlex[tfile->val].class;
            v10 != nil;
            v10 = v10->suc)
```

```
            if (v10->val != 0)
               {v12 = alloc_list(v10->val);
                qfile->suc = v12;
                qfile = qfile->suc;
               }
        if (tfile->val == c1)
           {if (Mop == VRAI) fprintf(yytrace,"-oui\n");
            return(VRAI);/*traitement de tfile*/
           }
      }
  if (Mop == VRAI) fprintf(yytrace,"-non\n");
  return(FAUX);
}

/*_____________________________________________________________________*/
/*_____________________Recherche de l'intersection_____________________*/
/*_____________________________________________________________________*/

int inter_cpt(el1,el2)  int el1,
                            el2;
                     /*_____________________________________________*/
                     /* Recherche de l'intersection de 2 concepts el1*/
                     /* et el2. Le resultat est l'intersection la    */
                     /* plus "proche" aux 2 elements.                */
                     /*_____________________________________________*/

{struct liste *vl, *tfile, *qfile;

 MOP("inter_cpt");
 if ((el1 == 0)||(el1 == el2)) return(el2);
 for (tfile = qfile = alloc_list(el1);
      tfile != nil;
      tfile = tfile->suc)
     {for (vl = Tlex[tfile->val].isa;
           vl != nil;
           vl = vl->suc)
          {if (isa_cpt(el2,vl->val) == VRAI)
              return(vl->val);
           qfile->suc = alloc_list(vl->val);
           qfile = qfile->suc;
          }
      }
}

/*_____________________________________________________________________*/
/*_____________________Inverser la relation arg________________________*/
/*_____________________________________________________________________*/

invers_arg()

                     /*_____________________________________________*/
                     /* Cette fonction permet de creer la relation   */
                     /* inverse de arg (gra) pour chaque concept     */
                     /* de l'application.                            */
                     /*_____________________________________________*/

{struct liste *vl;
 int i, j;

 MOP("invers_arg()");
 for (i = 1;
      i < Nblex;
      i++)
     {for (vl = Tlex[i].arg, j = 1;
           vl != nil;
           vl = vl->suc, j++)
```

```
            Tlex[vl->val].gra = ajt_list(alloc_list(i, j),
                                         Tlex[vl->val].gra);
        }
    return;
}

/*****************************************************************************/
/* MANIPULATION DE FORMULES.                                                 */
/*****************************************************************************/


/*___________________________________________________________________________*/
/*_____________________Relation isa pour un noeud____________________________*/
/*___________________________________________________________________________*/

int isa_fm(fm1, fm2)      int fm1, fm2;

                    /*_____________________________________________________*/
                    /* Extansion de la relation isa des concepts sur*/
                    /* les formules.                                       */
                    /*_____________________________________________________*/

{
 MOP("isa_fm");
 if (fm1 == VIDE) return(VRAI);
 if (fm2 == VIDE) return(FAUX);
 if ((Tnoeud[fm1].code == CH_CODE) || (Tnoeud[fm2].code == CH_CODE))
    if (Tnoeud[fm1].code == Tnoeud[fm2].code)
        if ((Tnoeud[fm1].reseau == 0) ||
            (Tnoeud[fm1].reseau == Tnoeud[fm2].reseau))
            return(VRAI);
        else return(FAUX);
    else return(FAUX);
 else return(isa_cpt(Tnoeud[fm1].reseau, Tnoeud[fm2].reseau));
}

/*___________________________________________________________________________*/
/*____________________Relation inter pour un noeud___________________________*/
/*___________________________________________________________________________*/

int inter_fm(fm1, fm2)    int fm1, fm2;

                    /*_____________________________________________________*/
                    /* Extansion de la relation inter des concepts  */
                    /* sur les formules.                                   */
                    /*_____________________________________________________*/

{int joint;

 MOP("inter_fm");
 if ( (Tnoeud[fm1].code == CH_CODE)
    &&(Tnoeud[fm2].code == CH_CODE))
    return(0);
 if (Tnoeud[fm1].code == CH_CODE) return(-1);
 if (Tnoeud[fm1].code == CH_CODE) return(-1);
 joint = inter_cpt(Tlex[Tnoeud[fm1].reseau], Tlex[Tnoeud[fm2].reseau]);
 if (joint == 0) return(-1);
 return(joint);
}

/*___________________________________________________________________________*/
/*_____________________Synchronisation des arguments_________________________*/
/*___________________________________________________________________________*/

struct liste *synch_fm(fm1, fm2)          int fm1,
                                              fm2;
```

```
                    /*_____________________________________________________*/
                    /* Cette fonction sert a synchroniser les             */
                    /* arguments des formules fm1 et fm2. Le resultat     */
                    /* est un pointeur sur un arguments de fm1 qui        */
                    /* verifie la synchronisation (si rien alors nil)     */
                    /* NB: On risque d'avoir des problemes dans le        */
                    /* cas  insense de non-isa(fm1,fm2).                  */
                    /* _____________________________________________________*/

{int i,cp;
 struct liste *vl;

 MOP("synch_fm");
 cp = synch_list(Tlex[Tnoeud[fm1].reseau].arg,Tlex[Tnoeud[fm2].reseau].arg);
 if (i == 0) return(nil);
 for (i = 1, vl = Tnoeud[fm1].chav;
      i < cp;
      i++, vl = vl->suc);
 return(vl);
}

/*************************************************************************/
/* MANIPULATION DE LA TABLE DES CHAINES (Constantes).                    */
/*************************************************************************/

/*_____________________________________________________________________*/
/*____________________Adjonction_______________________________________*/
/*_____________________________________________________________________*/

int aj_chaine(c)        char *c;

                    /*_____________________________________________________*/
                    /* Adjonction dans la table des chaines (ou           */
                    /* table des constantes) de la chaine de carac-       */
                    /* teres c. On recherche c dans la table de           */
                    /* facon seq. et on renvoie l'indice; sinon on        */
                    /* fait une adjonction en queue.                      */
                    /*_____________________________________________________*/

{int i;

 MOP("aj_chaine");
 if (strcmp(c,"") == 0) return(0);
 for (i = 1;
      i < Nbchaine;
      i++)
     {if (strcmp(Tchaine[i],c) == 0) return(i);}
 strcpy(Tchaine[Nbchaine],c);
 Nbchaine++;
 return(Nbchaine);
}

/*************************************************************************/
/* MANIPULATION DE LA TABLE DES NOEUDS.                                  */
/*************************************************************************/



/*************************************************************************/
/* MANIPULATIONS DE LA PILE D'UNIFICATEURS.                              */
/*************************************************************************/

/*_____________________________________________________________________*/
/*_____________________Adjonction dans le systeme______________________*/
/*_____________________________________________________________________*/
```

```
emp_syst(ed,n1,n2,f,l)  int ed,
                            n1,
                            n2,
                            f;
                        struct liste *l;

                    /*_____________________________________________________*/
                    /* Adjonction (gestion de pile) d'un element dans*/
                    /* le systeme des unificateurs. (x=y,w#z...).      */
                    /* ed = type UNIFIE ou EXCLU.                      */
                    /* n1,n2 = les 2 noeuds concernes.                 */
                    /* f = frontiere, c-a-d information necessaire     */
                    /*     pour un retour arriere.                     */
                    /*     Ceci permet d'eviter l'empilement de tout */
                    /*     l'environnement a chaque unification.       */
                    /*_____________________________________________________*/

{
 MOP("emp_syst");
 Psyst[Ptsyst].ed = ed;
 Psyst[Ptsyst].n1 = n1;
 Psyst[Ptsyst].n2 = n2;
 Psyst[Ptsyst].frontiere = f;
 Ptsyst[Psyst].action = l;
 Ptsyst++;
 if (Ptsyst >= T_SYST) erreur(4,"Psyst");
}

/*_____________________________________________________________________*/
/*_____________________________Suppression dans le systeme_____________*/
/*_____________________________________________________________________*/

dep_syst()

                    /*_____________________________________________________*/
                    /* Depiler de la pile Psyst.                       */
                    /*_____________________________________________________*/

{
 MOP("dep_syst");
 Ptsyst--;
 if (Ptsyst < 0) erreur(5,"Psyst");
 return;
}

/*********************************************************************/
/* MANIPULATIONS DE LA CLASSE D'EQUIVALENCE. (NOEUDS UNIFIES).       */
/*********************************************************************/

/*.....................................................................*/
/* REMARQUE: Dans cette partie, on fera appel aux fonctions externes*/
/*           du module delta.o.                                      */
/*.....................................................................*/

/*_____________________________________________________________________*/
/*_____________________________Test d'appartenance a la classe_________*/
/*_____________________________________________________________________*/

int test_equ(c1,c2)     int c1,
                            c2;

                    /*_____________________________________________________*/
                    /* On teste si c1,c2 sont dans la meme  classe.    */
                    /*_____________________________________________________*/
```

```
{
 MOP("test_equ");
 if  (Tnoeud[c1].eqrp == Tnoeud[c2].eqrp)
      return(VRAI);
 else
      return(FAUX);
}

/*_____________________________________________________________________*/
/*_____________________________Test de la coherence de  classe_________*/
/*_____________________________________________________________________*/

int coher_equ(c1,c2)     int c1,
                             c2;

                    /*_____________________________________________*/
                    /* On teste la coherence de 2 classes c1,c2 pour*/
                    /* l'operateur UNIFIE .                        */
                    /* En effet, on tient compte du critere suivant:*/
                    /* c1 = c2 => non(c1 # c2).                    */
                    /*_____________________________________________*/

{
 MOP("coher_equ");
 if  (test_mat(Texcl,Tnoeud[c1].eqrp,Tnoeud[c2].eqrp) == FAUX)
      return(VRAI);
 else
      return(FAUX);
}

/*_____________________________________________________________________*/
/*_____________________________Adjonction dans une classe______________*/
/*_____________________________________________________________________*/

int aj_equ(noeud,classe)          int noeud,
                                      classe;

                    /*_____________________________________________*/
                    /* Adjonction d'un noeud dans une classe en    */
                    /* unification. (UNIFIE)                       */
                    /*_____________________________________________*/

{int vqx, vqy, i;
 struct liste *l;
 int rp_noeud, rp_classe;

 MOP("aj_equ");
 rp_noeud = Tnoeud[noeud].eqrp;
 rp_classe = Tnoeud[classe].eqrp;
 if (coher_equ(noeud,classe) == FAUX) return(FAUX); /*1)Impossible  */
 if (test_equ(noeud,classe) == VRAI)                /*2)Deja fait   */
    return(VRAI);
 for (vqx = rp_noeud;                               /*3)OK:         */
      Tnoeud[vqx].eqls != vqx;
      Tnoeud[vqx].eqrp = rp_classe,                 /* -M. rp       */
      vqx = Tnoeud[vqx].eqls);
 for (vqy = classe;
      Tnoeud[vqy].eqls != vqy;                      /* -vqy=queue cl*/
      vqy = Tnoeud[vqy].eqls);
 Tnoeud[vqy].eqls = rp_noeud;
 for (i = 0, l = nil;                               /* -exclusion   */
      i <= Nbnoeud;
      i++)
     {if ( (test_mat(Texcl,i,rp_noeud) == VRAI)
         &&(test_mat(Texcl,i,rp_classe) == FAUX))
         {set_mat(Texcl,i,rp_classe);
```

```
            l = ajt_list(alloc_list(i,rp_classe),l);
          }
      }
 emp_syst(UNIFIE,noeud,classe,vqy,l);
 return(VRAI);
}

/*_____________________________________________________________________*/
/*_____________________Inversion de aj_equ_____________________________*/
/*_____________________________________________________________________*/

inv_aj_equ()

                    /*_____________________________________________*/
                    /* Fonction inverse de la precedente(aj_equ).  */
                    /* Cette fonction restaure la classe d'equivale-*/
                    /* nce en faisant un retour arriere d'UN cran, */
                    /* c-a-d la tete de la pile Psyst.             */
                    /*_____________________________________________*/

{int vqx, vqrp, vqf;
 struct liste *l;

 MOP("inv_aj_equ");
 dep_syst();
 vqf = Psyst[Ptsyst].frontiere;
 vqrp = Tnoeud[vqf].eqls;
 for (vqx = vqrp;
      vqx != Tnoeud[vqx].eqls;
      Tnoeud[vqx].eqrp = vqrp,
      vqx = Tnoeud[vqx].eqls);
 Tnoeud[vqf].eqls = vqf;
 for (l = Psyst[Ptsyst].action;
      l != nil;
      l = l->suc)
      reset_mat(Texcl,l->val,l->supp);
 return;
}

/*********************************************************************/
/* MANIPULATIONS DE LA CLASSE DES NOEUDS EXCLUS.                     */
/*********************************************************************/

/*.....................................................................*/
/* REMARQUE: Dans cette partie, on fera appel aux fonctions externes*/
/*           du module delta.o.                                      */
/*.....................................................................*/

/*_____________________________________________________________________*/
/*_____________________Test de la coherence de  classe_____________*/
/*_____________________________________________________________________*/

int coher_mat(c1,c2)     int c1,
                             c2;

                    /*_____________________________________________*/
                    /* On teste la coherence de 2 classes c1,c2 pour*/
                    /* l'operateur EXCLU .                         */
                    /* En effet, on tient compte du critere suivant:*/
                    /* c1 # c2 => non(c1 = c2).                    */
                    /*_____________________________________________*/

{
 MOP("coher_mat");
 if  (test_equ(c1,c2) == FAUX)
      return(VRAI);
```

```
 else
      return(FAUX);
}

/*_____________________________________________________________________*/
/*_____________________________Adjonction dans une classe_____________*/
/*_____________________________________________________________________*/

int aj_mat(noeud,classe)        int noeud,
                                classe;

                        /*_____________________________________________*/
                        /* Adjonction d'un noeud dans une classe en    */
                        /* exclusion .(EXCLU)                          */
                        /*_____________________________________________*/

{int rp_noeud, rp_classe;

 MOP("aj_mat");
 rp_noeud = Tnoeud[noeud].eqrp;
 rp_classe = Tnoeud[classe].eqrp;
 if (coher_mat(noeud,classe) == FAUX) return(FAUX);
 if (test_mat(Texcl,noeud,classe) == VRAI) return(VRAI);
 set_mat(Texcl,rp_noeud,rp_classe);
 emp_syst(EXCLU,noeud,classe,0,nil);
 return(VRAI);
}

/*_____________________________________________________________________*/
/*_____________________________Inversion de aj_mat_____________________*/
/*_____________________________________________________________________*/

inv_aj_mat()

                        /*_____________________________________________*/
                        /* Fonction inverse de la precedente(aj_mat).  */
                        /* Cette fonction restaure la matrice d'exclu- */
                        /* sion en faisant un retour arriere d'UN cran,*/
                        /* c-a-d la tete de la pile Psyst.             */
                        /*_____________________________________________*/

{int rp_1, rp_2;

 MOP("inv_aj_mat");
 dep_syst();
 rp_1 = Tnoeud[Psyst[Ptsyst].n1].eqrp;
 rp_2 = Tnoeud[Psyst[Ptsyst].n2].eqrp;
 reset_mat(Texcl,rp_1,rp_2);
 return;
}

/*********************************************************************/
/* INTERFACE SAISIE - RAISONNEUR.                                    */
/*********************************************************************/

/*_____________________________________________________________________*/
/*_____________________Interface pour une formule_____________________*/
/*_____________________________________________________________________*/

int itn_fm(fm)  struct formule *fm;

                        /*_____________________________________________*/
                        /* Rangement de la formule fm dans la table des*/
                        /* noeuds. Le resultat est le maximum de noeuds*/
                        /* ranges.                                     */
                        /*_____________________________________________*/
```

```
{int rel, mx, mxvt;
 struct doublet *vd;

 MOP("itn_fm");
 mx = -1;
 if (fm == nil) return(mx);
 fm->indic = MARQX;
 rel = fm->noeud;
 Tnoeud[Nbnoeud+rel].code = fm->code;
 Tnoeud[Nbnoeud+rel].signe = fm->neg;
 Tnoeud[Nbnoeud+rel].reseau = fm->reseau;
 Tnoeud[Nbnoeud+rel].suffixe = 0;
 Tnoeud[Nbnoeud+rel].eqrp = Nbnoeud + rel;
 Tnoeud[Nbnoeud+rel].eqls = Nbnoeud + rel;
 if (fm->pi == VRAI) I_qx = Nbnoeud + rel;
 Tlex[fm->reseau].form = ajq_list(Tlex[fm->reseau],alloc_list(Nbnoeud+rel));
 switch (fm->code)
        {case ID_CODE:  for (vd = fm->info.i_id.dif;
                             vd != nil;
                             vd = vd->suc)
                            aj_mat(Nbnoeud+(vd->lien->noeud),Nbnoeud+rel);
                        for (vd = fm->info.i_id.arg;
                             vd != nil;
                             vd =vd->suc)
                             {mxvt = itn_fm(vd->lien);
                              mx = MAX(mx,mxvt);
                             }
                        break;
         case CH_CODE:  Tnoeud[Nbnoeud+rel].reseau = aj_chaine(fm->info.i_ch.
                        mx = rel;
                        break;
         case ET_CODE:
         case OU_CODE:  for (vd = fm->info.i_opbin.f;
                             vd != nil;
                             vd =vd->suc)
                             {mxvt = itn_fm(vd->lien);
                              mx = MAX(mx,mxvt);
                             }
                        break;
         default:       erreur(6,"(interface)");
                        break;
        }
 return(mx);
}

/*_____________________________________________________________________*/
/*_____________________Interface_____________________________________*/
/*_____________________________________________________________________*/

itn_saisie()

                    /*_____________________________________________*/
                    /* Interface avec la saisie. Les pointeurs se  */
                    /* retrouveront dans:                          */
                    /* I_but --> noeud But,                        */
                    /* I_fait --> noeud Fait,                      */
                    /* I_quest --> noeud quest,                    */
                    /* I_qx --> objet sur lequel porte quest.      */
                    /*_____________________________________________*/

{int i1, i2, i3;

 MOP("itn_saisie");
 I_but = I_fait = I_quest = I_qx = VIDE;
 if (S_but != nil) I_but = Nbnoeud + (S_but->noeud);
```

```
 if (S_fait != nil) I_fait = Nbnoeud + (S_fait->noeud);
 if (S_quest != nil) I_quest = Nbnoeud + (S_quest->noeud);
 i1 = itn_fm(S_but);
 i2 = itn_fm(S_fait);
 i3 = itn_fm(S_quest);
 Nbnoeud = Nbnoeud + MAX(MAX(i1,i2),i3);
 Nbnoeud++;
}

/*************************************************************************/
/* MANIPULATIONS DE LA BASE DES FAITS.                                   */
/*************************************************************************/


/*=0=0=0=0=0=0=0=0=0=0=0=0=0=0=F=I=N=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0*/
```

```
# include <stdio.h>
# include "const.c"
# include "const2.c"
# include "type.c"
# include "ext.c"

/*-------------------------------------------------------------------------
/*
/* Ce fichier contient la pluspart des procedures qui sont utilisées    */
/* pour manipuler les noeuds, les formules et les règles.               */
/*
/*-------------------------------------------------------------------------

init_base_faits(i) int i;

/* fonction dont le but est l'initialisation de la  */
/* base de faits en fonction de la premisse de la   */
/* i eme regle. Les caracteristiques eventuelles    */
/* des antecedents seront ajoutes a la base de      */
/* faits.                                           */

{ int j, k;

  MOP("init_base_faits");
  Nbf=0;
  eclat(Trc[i].pre);
  for (j=0; j<Nbf; j++)
        for (k=0; k<Nbrc; k++)
            if (instance_fm(Tfait[j].acces, Trc[k].post)==VRAI)
               eclat(Trc[k].pre);
}

/*-------------------------------------------------------------------------

incoherent_bf()

/* fonction dont le but est de determiner si la     */
/* base de faits est dans un etat coherent.         */
/* C'est a dire, si elle ne contient pas deux       */
/* contradictoires.                                 */

{ int i, j;

   for (i=0; i<Nbf; i++)
      for (j=i + 1; j<Nbf; j++)
           if ((compar_form(Tfait[i].acces, Tfait[j].acces)==VRAI)&&
               (sign_opp(Tfait[i].acces, Tfait[j].acces)==VRAI))
                 return(VRAI);
   return(FAUX);
}

/*-------------------------------------------------------------------------

sign_opp(i, j)  int i, j;

/* fonction dont le but est de determiner si deux   */
/* formules sont de signes opposes ou pas.          */
/* si les deux formules donnees en entree sont de   */
/* signe oppose, la valeur VRAI sera renvoyee,      */
/* sinon la valeur FAUX sera renvoyee.              */

{ if (Tnoeud[i].signe==Tnoeud[j].signe) return(FAUX);
  return(VRAI);
}
```

```
/*-------------------------------------------------------------------------*/

compar_form(i,j)  int i,j;

/* fonction dont le but est de verifier si deux     */
/* formules sont identiques au signe pres.          */
/* si c'est le cas, la valeur VRAI sera renvoyee,   */
/* sinon la valeur FAUX sera renvoyee.              */

{ struct liste *li,
               *lj;
  if (Tnoeud[i].reseau!=Tnoeud[j].reseau) return(FAUX);
  for(li=Tnoeud[i].chav,
      lj=Tnoeud[j].chav;
      ((li!=nil)&&(lj!=nil));
      li=li->suc,
      lj=lj->suc) if (Tnoeud[li->val].reseau!=Tnoeud[lj->val].reseau)
                     return(FAUX);
  return(VRAI);
}

/*-------------------------------------------------------------------------*/

struct liste *appar_nd(form1,form2) int form1,form2;

/* fonction dont le but est de realiser le lien     */
/* entre form1 et form2 qui sont deux noeuds dont   */
/* sait que form2 est une instance de form1.        */
/* la fonction renverra une liste d'unificateurs    */
/* couples d'entiers.                               */


{ struct liste *l,
               *lp;

  struct pile P1,
              P2;

  int pp;

  MOP("appar_nd");
  l=nil;
  vide_pile(&P1);
  vide_pile(&P2);
  push_pile(&P1,form1);
  push_pile(&P2,form2);
  for (pp=0;
       pp<P1.ppile;
       pp++)

       if (!(lp=in_list0(l,P1.Pile[pp])))

          { l=ajq_list(l,alloc_list(P1.Pile[pp],P2.Pile[pp]));
            cp_lst_pile_all(&P1,Tnoeud[P1.Pile[pp]].chav);
            cp_lst_pile_all(&P2,Tnoeud[P2.Pile[pp]].chav);
          }

       else if (lp->supp!=P2.Pile[pp]) return(nil);

       return(l);
}

/*-------------------------------------------------------------------------*/

/* fonction dont le but est de rechercher parmi */
```

```
/* l'ensemble des faits de la base de faits     */
/* ceux qui sont instance de form.              */
/* la fonction renverra une liste de listes d'  */
/* unificateurs des faits en question et de     */
/* form.                                        */

struct lliste *llist_instance_fm(form) int form;

{ struct lliste *llst;
  int i;

  MOP("llist_instance_fm");
  llst=nil;
  for(i=0;
      i<Nbf;
      i++)

      if (instance_fm(Tfait[i].acces,form))

           llst=ajq_llist(llst,alloc_llist(appar_nd(Tfait[i].acces,form)));

  return(llst);
}

/*-------------------------------------------------------------------------

coher_list_elt (list,el1,el2) struct liste *list; int el1,el2;

/* list est une liste de couples d'entiers.     */
/* el1 et el2 sont deux entiers.                */
/* la fonction renverra 0 si la liste contient */
/* deja le couple (el1,el2);                    */
/* la fonction renverra  2 si la liste          */
/* contient un couple dnt l'element de gauche   */
/* est el1 mais dont l'element de droite n'est */
/* pas el2.                                     */
/* la fonction renverra 1 si la liste list ne   */
/* contient aucun couple dont l'element de      */
/* gauche est egal a el1.                       */

{ struct liste *pl;

  MOP("coher_list_elt");
  pl=in_list2(list,el1);
  if (pl==nil) return(1);
  if (pl->supp != el2) return(2);
  return(0);
}

/*-------------------------------------------------------------------------*/

struct liste *union_coher_list(lst1,lst2) struct liste *lst1,*lst2;

/* lst1 et lst2 sont deux listes de couples     */
/* d'entiers.                                   */
/* la fonction renverra une liste vide si       */
/* il existe i,j :  lst1[i].val==lst2[j].val    */
/*              et lst1[i].supp!=lst2[j].supp   */
/* sinon la fonction renverra une liste de      */
/* couple d'entiers non vide, egale a la        */
/* reunion des deux listes en entrees de la-    */
/* quelle auront ete supprimes les couples      */
/* apparaissant plus d'une fois.                */

{ struct liste *p,
               *pl;
```

```
  MOP("union_coher_list");
  if (lst1 == nil) return(lst2);
  if (lst2 == nil) return(lst1);
  pl = lst1;
  for (p=lst2;
       p!=nil;
       p=p->suc)

       switch (coher_list_elt(pl,p->val,p->supp))
         { case  1 : pl=ajq_list(pl,alloc_list(p->val,p->supp));
                     break;

           case  0 : break;

           case  2 : return(nil);
                     break;
         }
   return(pl);
}

/*-------------------------------------------------------------------------*/

struct liste *union_liste(d) struct tab_pointer_ll *d;

/* fonction dont le but est de renvoyer une liste   */
/* de l'union coherente des liste donnees dans d.   */
/* d est une table de pointeurs vers des listes de  */
/* listes d'unificateurs,                           */

{ struct liste *p;

  int i;

  MOP("union_liste");
  for (p=nil,
       i=0;
       i<d->taille;
       i++)

       if ((p=union_coher_list(p,d->tab[i]->liste))==nil) return(nil);

  return(p);
}

/*-------------------------------------------------------------------------*/

list_instance_cond(cond,res) int cond; struct tab_pointer_ll *res;

/* procedure dont le but est de rechercher */
/* l'ensemble des listes de jeux de        */
/* substitution possible pour la formule   */
/* que represente cond. Ces listes seront  */
/* livrées dans res.                    */
/* res est un tableau de pointeurs vers    */
/* les listes de jeux de subsitution.      */

{ struct liste *p;

  int i;

  MOP("list_instance_cond");
  for(i=0;i<T_NOEUD;i++) res->tab[i]=nil;
  res->taille=0;
  for(i=0,
      p=Tnoeud[cond].chav;
```

```
      p!=nil;
      p=p->suc,
      i++)

      res->tab[i]=llist_instance_fm(p->val);

  res->taille=i;
}

/*-------------------------------------------------------------------------*/

avance (lp0,lp) struct tab_pointer_ll *lp0,*lp;

{ int j,
      fin,
      fini=VRAI;

  MOP("avance");
  for(j=lp0->taille-1,
      fin=FAUX;
      (!fin)&&(j>=0); )

     if (lp->tab[j]->suc != nil)

        { lp->tab[j]=lp->tab[j]->suc;
          fin=VRAI;
        }

     else if (j > 0)

             { lp->tab[j]=lp0->tab[j];
               j--;
             }

          else return(FAUX);
  return(VRAI);
}

/*-------------------------------------------------------------------------*/

struct lliste *eval (regle) int regle;

{ struct lliste *ll,
                *lp;
  struct liste  *l,
                *p;
  struct tab_pointer_ll d0,
                        d;
  int i,
      fini,
      foutu,
      vr;

  MOP("eval");

/* si la precondition est de type ID_CODE */

  if (Tnoeud[Trc[regle].pre].code==ID_CODE)

     {  ll=llist_instance_fm(Trc[regle].pre);
        if (Vis) visu_llist(ll);
        return(ll);
     }

/* si la precondition est de type ET_CODE */
```

```
  if (Tnoeud[Trc[regle].pre].code==ET_CODE)

     {  list_instance_cond(Trc[regle].pre,&d0);
        for (i=0;i<d0.taille;i++)
           if (d0.tab[i]!=nil) d.tab[i]=d0.tab[i];
           else return(nil);
        d.taille=d0.taille;
        ll=nil;
        for(fini=FAUX;
            fini==FAUX;
            fini=!avance(&d0,&d))

            { l=union_liste(&d);
              if (l!=nil)
                 ll=ajq_llist(ll,alloc_llist(l));
            }
        return(ll);
      }

/* si la precondition est de type OU_CODE */

  if (Tnoeud[Trc[regle].pre].code==OU_CODE)

    { vr=FAUX;
      for (p=Tnoeud[Trc[regle].pre].chav;
           (p!=nil)&&(!vr);
           p=p->suc)
           switch (Tnoeud[p->val].code) {

             case ID_CODE :   ll=llist_instance_fm(p->val);
                              return(ll);
                              break;
             case ET_CODE :   list_instance_cond(p->val,&d0);
                              foutu=FAUX;
                              for (i=0;i<d0.taille;i++)
                                  if (d0.tab[i]!=nil) d.tab[i]=d0.tab[i];
                                  else foutu=VRAI;
                              if (foutu) return(nil);
                              d.taille=d0.taille;
                              for(fini=FAUX,
                                  ll=nil;
                                  fini==FAUX;
                                  fini=!avance(&d0,&d))
                                  { l=union_liste(&d);
                                    if (l!=nil)
                                       ll=ajq_llist(ll,alloc_llist(l));
                                  }
                              return(ll);
                              break;
                    }
     }
}

/*-------------------------------------------------------------------------*/

/* fonction dont le but est de verifier      */
/* si un noeud (nd1) est une instance        */
/* d'un autre noeud (nd1).                   */
/* si c'est le cas, VRAI sera renvoye,       */
/* sinon FAUX sera renvoye.                  */

instance_nd(nd1,nd2) int nd1,nd2;

{ MOP("instance_nd");
  if (((Tnoeud[nd1].reseau!=Tnoeud[nd2].reseau)&&
       !isa(Tnoeud[nd1].reseau,Tnoeud[nd2].reseau))||
```

```
      (Tnoeud[nd1].signe!=Tnoeud[nd2].signe))
      return(FAUX);
  return(VRAI);
}

/*-------------------------------------------------------------------------*/

/* fonction dont le but est de verifier     */
/* si une formule (form1) est une instance  */
/* d'une autre formule (form2).             */
/* si c'est le cas, VRAI sera renvoye,      */
/* sinon FAUX sera renvoye.                 */

instance_fm(form1,form2) int form1,form2;

{ struct liste *ptr1,
               *ptr2;
  int premier;

MOP("instance_fm");

/* si le premier membre de la formule form1 */
/* ne constitue pas une instance du premier */
/* membre de la formule form2, ce n'est pas */
/* peine de continuer.                      */

if (!instance_nd(form1,form2)) return(FAUX);

/* il faut ensuite verifier si toutes les   */
/* les formules qui composent les arguments */
/* des deux formules sont instances les     */
/* unes des autres.                         */

for (ptr1=Tnoeud[form1].chav,
     ptr2=Tnoeud[form2].chav;
     (ptr1!=nil)&&(ptr2!=nil);
     ptr1=ptr1->suc,
     ptr2=ptr2->suc)
       if (!instance_fm(ptr1->val,ptr2->val)) return(FAUX);

/* si une des deux formules a plus d'       */
/* arguments que l'autre, elles ne peuvent  */
/* instances l'une de l'autre.              */

return ((ptr1==nil)&&(ptr2==nil));
}

/*-------------------------------------------------------------------------*/

/* fonction dont le but est de verifier si le */
/* concept fils est un descendant du concept  */
/* pere. Si c'est le cas, VRAI sera renvoye,  */
/* sinon FAUX sera renvoye.                   */
/* putain, que cette fonction m'a             */
/* fait chier !                               */

isa (fils,pere) int fils,pere;

{ int i;
  struct pile  P;


MOP("isa");
if (Tlex[fils].ordre<Tlex[pere].ordre) return(FAUX);
vide_pile(&P);
push_pile(&P,fils);
```

```
for (i=0;i<P.ppile;i++)
    { if (P.Pile[i]==pere)  return(VRAI);
      cp_lst_pile_onw(&P,Tlex[P.Pile[i]].isa);
    }
return(FAUX);
}

/*-------------------------------------------------------------------------*/

comp_str_semb(nd1,nd2) int nd1,nd2;

/*************************************************/
/*                                               */
/* fonction dont le but est de verifier si       */
/* deux noeuds sont strictement semblables.      */
/* C'est a dire                                  */
/* s'il ont memes profils, memes valeurs         */
/* dans reseau et memes signes.                  */
/*                                               */
/*************************************************/

{ struct liste *ptr1,
               *ptr2;
  int semb;

MOP("comp_str_semb");
if (Tnoeud[nd1].reseau!=Tnoeud[nd2].reseau) return(FAUX);
if (Tnoeud[nd1].signe!=Tnoeud[nd2].signe) return(FAUX);
if (Tnoeud[nd1].suffixe!=Tnoeud[nd2].suffixe) return(FAUX);
for (ptr1=Tnoeud[nd1].chav,
     ptr2=Tnoeud[nd2].chav;
     ((ptr1!=nil)&&(ptr2!=nil));
     ptr1=ptr1->suc,
     ptr2=ptr2->suc) if (!comp_str_semb(ptr1->val,ptr2->val)) return(FAUX);
return((ptr1==nil)&&(ptr2==nil));
}

/*-------------------------------------------------------------------------*/
/*                    MANIPULATION DES FAITS                               */
/*-------------------------------------------------------------------------*/

/* fonction dont le but est d'ajouter un fait */
/* atomique a l'ensemble des faits.           */

ajout_fait(nd)

int nd;

{
MOP("ajout_fait");

Tfait[Nbf].acces=nd;
Nbf++;
}

/*-------------------------------------------------------------------------*/

/* fonction dont le but est de verifier si un     */
/* fait trouve  dans l'ensemble des faits de la   */
/* base.   */

in_bf(fait) int fait;

{ int i;

  MOP("in_bf");
```

```
  for (i=0;i<Nbf;i++)
    if(comp_str_semb(fait,Tfait[i].acces)) return(VRAI);
  return(FAUX);
}

/*-------------------------------------------------------------------------*/

/* fonction dont le but est d'ajouter un fait  */
/* non necessairement atomique a l'ensemble    */
/* des faits. Ce fait sera soit atomique soit  */
/* une suite de faits atomiques relies par des */
/* 'et'.                                       */

eclat (nd) int nd;

{ struct liste *ptr1;

 MOP("eclat");

 if(Tnoeud[nd].code==ET_CODE) {
     ptr1=Tnoeud[nd].chav;
     while(ptr1!=nil) {
          ajout_fait(ptr1->val);
          ptr1=ptr1->suc;
      }
   }
 else ajout_fait(nd);
}

/*-------------------------------------------------------------------------*/
/*                  FONCTIONS DE RECHERCHE DANS LA TRACE                   */
/*-------------------------------------------------------------------------*/

in_trace(regle,liste) int regle; struct liste *liste;

{ int i;

  MOP("in_trace");
  for (i=0;
       i<Nbtr;
       i++)
       if (Trace[i].regle==regle)
          if (compar_liste(Trace[i].liste,liste)) return(VRAI);
  return(FAUX);
}

/*-------------------------------------------------------------------------*/

compar_liste(liste1,liste2) struct liste *liste1,*liste2;

{ struct liste *ptr1,
               *ptr2;

  MOP("compar_liste");
  for (ptr1=liste1,
       ptr2=liste2;
       (ptr1!=nil)&&(ptr2!=nil);
       ptr1=ptr1->suc,
       ptr2=ptr2->suc)
       if ((ptr1->val!=ptr2->val)||(ptr1->supp!=ptr2->supp)) return(FAUX);

  return((ptr1==nil)&&(ptr2==nil));
}
/*-------------------------------------------------------------------------*/
/*              FONCTION  D'INITIALISATION DE LA TRACE                     */
/*-------------------------------------------------------------------------*/
```

```
init_trace()

{
 Nbtr=0;
}

/*-------------------------------------------------------------------------*/
/*                    FONCTION  D'AJOUT DANS LA TRACE                      */
/*-------------------------------------------------------------------------*/

ajouter_regle_trace(regle,liste) int regle; struct liste *liste;

{ MOP("ajouter_regle_trace");
  if (Nbtr==T_TRACE)  erreur(8,"");
  Trace[Nbtr].regle=regle;
  Trace[Nbtr].liste=liste;
  Nbtr++;
}

/*-------------------------------------------------------------------------*/
/*                  FONCTIONS D'APPLICATION D'UNE REGLE                    */
/*-------------------------------------------------------------------------*/

appliquer_regle(regle,l) int regle; struct liste *l;

{ int n;
  struct liste *lst;

  MOP("appliquer_regle");
  n=Nbnoeud;
  lst=nil;
  copy_noeud(Trc[regle].post,0,lst);
  mk_substi(n,l);
  return(n);
}

/*-------------------------------------------------------------------------*/

mk_substi(nd,l)   int nd ;   struct liste *l;

/* fonction dont le but est d'effectuer     */
/* les substitutions entre les occurences   */
/* des concepts de la precondition et ceux  */
/* du fait verifiant celle-ci avec les      */
/* concepts de la post condition.           */

{ struct liste *ptr,
               *ndpt;
  int ndb;

  MOP("mk_substi");
  if (Mop)

    { fprintf(yytrace,"travail avec la liste \n");
            visu_list_inst(l);
            trait(20,'*');
    }

  if ((ndpt=in_list3(l,nd))!=nil)

    {   Tnoeud[nd].reseau=Tnoeud[ndpt->val].reseau;
        Tnoeud[nd].suffixe=Tnoeud[ndpt->val].suffixe;
    }

   for(ptr=Tnoeud[nd].chav;
```

```
        ptr!=nil;
        ptr=ptr->suc)
                    mk_substi(ptr->val,1);
}

/*-------------------------------------------------------------------------*/


copy_noeud(nd,ndp,l) int nd,ndp; struct liste *l;

{ struct liste *p,
               *pp;
  int ndd;

  extern struct liste *in_list0();


  MOP("copy_noeud");
  if (ndp==0)
     {  ndd=Nbnoeud;
        Tnoeud[Nbnoeud].code=Tnoeud[nd].code;
        Tnoeud[Nbnoeud].signe=Tnoeud[nd].signe;
        Tnoeud[Nbnoeud].reseau=Tnoeud[nd].reseau;
        Tnoeud[Nbnoeud].suffixe=Tnoeud[nd].suffixe;
        Tnoeud[Nbnoeud].chav=nil;
        Tnoeud[Nbnoeud].charr=nil;
        Tnoeud[Nbnoeud].eqrp=Nbnoeud;
        Tnoeud[Nbnoeud].eqls=Nbnoeud;
        l=ajq_list(l,alloc_list(nd,Nbnoeud));
        Nbnoeud++;
      }
  else if ((pp=in_list0(l,nd))==nil)
         { ndd=Nbnoeud;
           Tnoeud[Nbnoeud].code=Tnoeud[nd].code;
           Tnoeud[Nbnoeud].signe=Tnoeud[nd].signe;
           Tnoeud[Nbnoeud].reseau=Tnoeud[nd].reseau;
           Tnoeud[Nbnoeud].suffixe=Tnoeud[nd].suffixe;
           Tnoeud[Nbnoeud].chav=nil;
           Tnoeud[Nbnoeud].charr=nil;
           Tnoeud[Nbnoeud].eqrp=Nbnoeud;
           Tnoeud[Nbnoeud].eqls=Nbnoeud;
           Tnoeud[ndp].chav=ajq_list(Tnoeud[ndp].chav,alloc_list(ndd,ndd));
           Tnoeud[Nbnoeud].charr=ajq_list(Tnoeud[Nbnoeud].charr,
                                          alloc_list(ndp,ndp));
           l=ajq_list(l,alloc_list(nd,Nbnoeud));
           Nbnoeud++;
         }
       else { Tnoeud[ndp].chav=ajq_list(Tnoeud[ndp].chav,
                                        alloc_list(pp->supp,pp->supp));
              Tnoeud[pp->supp].charr=ajq_list(Tnoeud[pp->supp].charr,
                                              alloc_list(ndp,ndp));
              ndd=pp->supp;
            }
  for(p=Tnoeud[nd].chav;
      p!=nil;
      p=p->suc) copy_noeud(p->val,ndd,l);

}


/*-------------------------------------------------------------------------*/

/* fonction dont le but est de determiner */
/* si un concept ayant une entree donnee  */
/* dans la table du lexique se trouve     */
/* dans une formule donnee.               */
```

```
in_pre(ncon,nfor) int ncon,nfor;

{ int ppile,i;
  struct pile P;
  struct liste *ptr1,
               *ptr2;

   MOP("in_pre");
   vide_pile(&P);
   push_pile(&P,nfor);
   for (i=0;i<P.ppile;i++)
      { if ((Tnoeud[P.Pile[i]].reseau==Tnoeud[ncon].reseau)&&
            (Tnoeud[P.Pile[i]].suffixe==Tnoeud[ncon].suffixe))

             return(P.Pile[i]);

        cp_lst_pile_onw(&P,Tnoeud[P.Pile[i]].chav);
      }
   return(NOTPRE);
}

/*-------------------------------------------------------------------------*/
```

```
#include "const.c"
#include "const2.c"
#include "type.c"
#include "ext.c"

/*-------------------------------------------------------------------------*/
/*                                                                         */
/*              RECHERCHE DANS DES LISTES                                  */
/*                                                                         */
/*-------------------------------------------------------------------------*/

struct liste *in_list0(lst,nd) struct liste *lst; int nd;

/*                                                  */
/* fonction dont le but est de determiner           */
/* si l'entier nd se trouve dans lst.               */
/* Si c'est le cas, l'adresse de l'element          */
/* de la liste contenant nd sera renvoyee,          */
/* sinon nil sera renvoye.                          */
/*                                                  */


{ struct liste *ptr;

  MOP("in_list0");
  for (ptr=lst;
       ptr!=nil;
       ptr=ptr->suc)
       if (ptr->val==nd) return(ptr);
  return(nil);
}

/*-------------------------------------------------------------------------*/

struct liste *in_list1(lst,nd) struct liste *lst; int nd;

/*                                                  */
/* fonction dont le but est de determiner           */
/* si l'entier nd se trouve dans lst.               */
/* Si c'est le cas, l'adresse de l'element          */
/* de la liste contenant nd sera renvoyee,          */
/* sinon nil sera renvoye.                          */
/*                                                  */

{ struct liste *ptr;

  MOP("in_list1");
  for (ptr=lst;
       ptr!=nil;
       ptr=ptr->suc)
       if (ptr->supp==nd) return(ptr);
  return(nil);
}

/*-------------------------------------------------------------------------*/

struct liste *in_list2(lst,nd) struct liste *lst; int nd;

/*                                                  */
/* fonction dont le but est de determiner           */
/* si un noeud, ayant meme entree que nd            */
/* dans le lexique, se trouve dans lst.             */
/* Si c'est le cas, l'adresse de l'element          */
/* de la liste contenant nd sera renvoyee,          */
```

```
/* sinon nil sera renvoye.                                    */
/*                                                            */

{ struct liste *ptr;

  MOP("in_list2");
  for (ptr=lst;ptr!=nil;ptr=ptr->suc)
      {
        if ((Tnoeud[ptr->val].reseau==Tnoeud[nd].reseau)&&
            (Tnoeud[ptr->val].suffixe==Tnoeud[nd].suffixe)) return(ptr);
      }
  return(nil);
}

/*------------------------------------------------------------------------*/

struct liste *in_list3(lst,nd) struct liste *lst; int nd;

/*                                                            */
/* fonction dont le but est de determiner                     */
/* si un noeud, ayant meme entree que nd                      */
/* dans le lexique, se trouve dans lst.                       */
/* Si c'est le cas, l'adresse de l'element                    */
/* de la liste contenant nd sera renvoyee,                    */
/* sinon nil sera renvoye.                                    */
/*                                                            */

{ struct liste *ptr;

  MOP("in_list2");
  for (ptr=lst;ptr!=nil;ptr=ptr->suc)
      {
        if ((Tnoeud[ptr->supp].reseau==Tnoeud[nd].reseau)&&
            (Tnoeud[ptr->supp].suffixe==Tnoeud[nd].suffixe)) return(ptr);
      }
  return(nil);
}

/*------------------------------------------------------------------------*/
/*                                                                        */
/*                RECHERCHE DANS UNE ARBORESCENCE DE NOEUDS.              */
/*                                                                        */
/*------------------------------------------------------------------------*/

rech_LxIs(ndo,ndt)  int ndo,ndt;

/* fonction dont le but est de rechercher le noeud    */
/* descendant de ndo ayant meme entree dans Tlex que  */
/* le noeud ndt ou dont l'entree dans Tlex soit un    */
/* descendant de l'entree de ndt dans Tlex.           */

{ int i;
  struct liste *ptr;
  struct pile  *P;

   vide_pile(P);
   push_pile(P,ndo);
   for (i=0;i<P->ppile;i++)
      { if ((Tnoeud[P->Pile[i]].reseau==Tnoeud[ndt].reseau)||
            (isa(Tnoeud[P->Pile[i]].reseau,Tnoeud[ndt].reseau)))
                                                  return(P->Pile[i]);
        cp_lst_pile_onw(P,Tnoeud[P->Pile[i]].chav);
      }
   return(NOTPRE);
}
```

```
/*-------------------------------------------------------------------------*/


rech_LxIsSi(ndo,ndt)  int ndo,ndt;

/* fonction dont le but est de rechercher le noeud   */
/* descendant de ndo ayant meme entree dans Tlex que */
/* le noeud ndt ou dont l'entree dans Tlex soit un   */
/* descendant de l'entree de ndt dans Tlex et        */
/* meme signe.                                       */

{ int i;
  struct pile  *P;

   vide_pile(P);
   push_pile(P,ndo);
   for (i=0;i<P->ppile;i++)
      { if (((Tnoeud[P->Pile[i]].reseau==Tnoeud[ndt].reseau)||
              (isa(Tnoeud[P->Pile[i]].reseau,Tnoeud[ndt].reseau)))&&
             (Tnoeud[P->Pile[i]].signe==Tnoeud[ndt].signe))
                                                  return(P->Pile[i]);
        cp_lst_pile_onw(P,Tnoeud[P->Pile[i]].chav);
       }
   return(NOTPRE);
}

/*-------------------------------------------------------------------------*/


rech_LxIsSiSu(ndo,ndt)  int ndo,ndt;

/* fonction dont le but est de rechercher le noeud   */
/* descendant de ndo ayant meme entree dans Tlex que */
/* le noeud ndt ou dont l'entree dans Tlex soit un   */
/* descendant de l'entree de ndt dans Tlex,          */
/* meme signe et meme suffixe.                       */

{ int i;
  struct pile  *P;

   vide_pile(P);
   push_pile(P,ndo);
   for (i=0;i<P->ppile;i++)
      { if (((Tnoeud[P->Pile[i]].reseau==Tnoeud[ndt].reseau)||
              (isa(Tnoeud[P->Pile[i]].reseau,Tnoeud[ndt].reseau)))&&
             (Tnoeud[P->Pile[i]].signe==Tnoeud[ndt].signe)&&
             (Tnoeud[P->Pile[i]].signe==Tnoeud[ndt].signe))
                                                  return(P->Pile[i]);
        cp_lst_pile_onw(P,Tnoeud[P->Pile[i]].chav);
       }
   return(NOTPRE);
}

/*-------------------------------------------------------------------------*
```

```
#include <stdio.h>
#include "const.c"
#include "const2.c"
#include "type.c"
#include "ext.c"

extern struct liste *ajq_list(),
                    *ajt_list(),
                    *alloc_list();

extern FILE
     *pnoeud,          /*Repres d'un noeud d'une formule    */
     *pchaine,         /*Table-Fichier des chaines utilisees*/
     *pisa,            /*Relations isa pour les concepts    */
     *pclass,          /*Relations de classe -id-           */
     *parg,            /*Listes des arguments d'un concept  */
     *plex,            /*Lexique resultat                   */
     *plien,           /*Lien pour chaque noeud.            */
     *pcond,           /*Noeud des conditionnelles          */
     *pbd,             /*Ensemble des formules de la bd     */
     *prc,             /* --- id ---              rc        */
     *prs;             /* --- id ---              rs        */
                       /*_____________________________________*/

/*************************************************************************/
/* CE FICHIER CONTIENT TOUTES LES FONCTIONS CONCERNANTS L'ACQUI-*/
/* SITION DE LA BASE DE CONNAISSANCE.                          */
/* ON CENTRALISERA L'ACQUISITION GLOBALE DANS UNE SEULE FONCTION*/
/* QU'ON APPELERA LORS DE L'INITIALLISATION DU SYSTEME.         */
/* ON INCLUERA CE FICHIER AVEC LA DIRECTIVE #INCLUDE "ACQ.C".  */
/*************************************************************************/

/*************************************************************************/
/* -ouvrir();                                                  */
/* -fermer();                                                  */
/* -acq_lex();                                                 */
/* -acq_arg();                                                 */
/* -acq_fils();                                                */
/* -acq_pere();                                                */
/* -acq_noeud()                                                */
/* -acq_lien();                                                */
/* -acq_chaine();                                              */
/* -acq_cond();                                                */
/* -acq_bd();                                                  */
/* -acq_rc();                                                  */
/* -acq_rs();                                                  */
/* _m_a_j_charr();                                             */
/* -acquerir();                                                */
/*************************************************************************/

/*_____________________________________________________________*/
/*_____________________Open des Fichiers_____________________*/
/*_____________________________________________________________*/

ouvrir()
                    /*_____________________________________________*/
                    /* Ouverture des Fichiers (cf parametres)      */
                    /*_____________________________________________*/
{
 MOP("ouvrir");
 strcpy(Fichier,Entree);
 pnoeud = fopen(strcat(Fichier,".nd.d"),"r");
 if (Mop == VRAI) fprintf(yytrace,"\n%s",Fichier);
 if (pnoeud == nil) erreur(3,Fichier);
```

```
strcpy(Fichier,Entree);
pchaine = fopen(strcat(Fichier,".ch.d"),"r");
if (Mop == VRAI) fprintf(yytrace,"\n%s",Fichier);
if (pnoeud == nil) erreur(3,Fichier);
strcpy(Fichier,Entree);
pisa = fopen(strcat(Fichier,".isa.d"),"r");
if (Mop == VRAI) fprintf(yytrace,"\n%s",Fichier);
if (pnoeud == nil) erreur(3,Fichier);
strcpy(Fichier,Entree);
pclass = fopen(strcat(Fichier,".class.d"),"r");
if (Mop == VRAI) fprintf(yytrace,"\n%s",Fichier);
if (pnoeud == nil) erreur(3,Fichier);
strcpy(Fichier,Entree);
parg = fopen(strcat(Fichier,".arg.d"),"r");
if (Mop == VRAI) fprintf(yytrace,"\n%s",Fichier);
if (pnoeud == nil) erreur(3,Fichier);
strcpy(Fichier,Entree);
plex = fopen(strcat(Fichier,".lex.d"),"r");
if (Mop == VRAI) fprintf(yytrace,"\n%s",Fichier);
if (pnoeud == nil) erreur(3,Fichier);
strcpy(Fichier,Entree);
plien = fopen(strcat(Fichier,".lien.d"),"r");
if (Mop == VRAI) fprintf(yytrace,"\n%s",Fichier);
if (pnoeud == nil) erreur(3,Fichier);
strcpy(Fichier,Entree);
pcond = fopen(strcat(Fichier,".cond.d"),"r");
if (Mop == VRAI) fprintf(yytrace,"\n%s",Fichier);
if (pnoeud == nil) erreur(3,Fichier);
strcpy(Fichier,Entree);
pbd = fopen(strcat(Fichier,".bd.d"),"r");
if (Mop == VRAI) fprintf(yytrace,"\n%s",Fichier);
if (pnoeud == nil) erreur(3,Fichier);
strcpy(Fichier,Entree);
prc = fopen(strcat(Fichier,".rc.d"),"r");
if (Mop == VRAI) fprintf(yytrace,"\n%s",Fichier);
if (pnoeud == nil) erreur(3,Fichier);
strcpy(Fichier,Entree);
prs = fopen(strcat(Fichier,".rs.d"),"r");
if (Mop == VRAI) fprintf(yytrace,"\n%s",Fichier);
if (pnoeud == nil) erreur(3,Fichier);
}

/*_____________________________________________________________________*/
/*_____________________Close des Fichiers_____________________________*/
/*_____________________________________________________________________*/

fermer()
                    /*_____________________________________________*/
                    /* fermeture des Fichiers (cf parametres)    */
                    /*_____________________________________________*/
{
  MOP("fermer");
 fclose(pnoeud);
 fclose(pchaine);
 fclose(pisa);
 fclose(pclass);
 fclose(parg);
 fclose(plex);
 fclose(plien);
 fclose(pcond);
 fclose(pbd);
 fclose(prc);
 fclose(prs);
}
```

```
/*_____________________________________________________________________*/
/*_____________________Le lexique______________________________________*/
/*_____________________________________________________________________*/

acq_lex()

                    /*_____________________________________________*/
                    /* Acquisition des chaine de caracteres du    */
                    /* lexique.                                   */
                    /*_____________________________________________*/
{int i;

 MOP("acq_lex");
 fscanf(plex,"%d\n",&Nblex);
 if (Nblex > T_LEX) erreur(1);
 strcpy(Tlex[0].libelle ,"#");
 for (i = 1;
      i < Nblex;
      i++)
     {fscanf(plex,"%s\n",Tlex[i].libelle);
     }
}

/*_____________________________________________________________________*/
/*_____________________Les arguments___________________________________*/
/*_____________________________________________________________________*/

acq_arg()

                    /*_____________________________________________*/
                    /* acquisition des arguments pour chaque      */
                    /* concept.                                   */
                    /*_____________________________________________*/

{int i, val, suppl;
 struct liste *vl;

 MOP("acq_arg");
 for (i = 0;
      i < Nblex;
      i++)
     {for(vl = nil,fscanf(parg,"%d\n%d\n",&val,&suppl);
          val != -1;
          fscanf(parg,"%d\n%d\n",&val,&suppl))
          vl = ajq_list(vl,alloc_list(val,suppl));
      Tlex[i].arg = vl;
     }
}

/*_____________________________________________________________________*/
/*_____________________Les Fils________________________________________*/
/*_____________________________________________________________________*/

acq_fils()

                    /*_____________________________________________*/
                    /* acquisition des fils(lien ISA) pour        */
                    /* chaque concept.                            */
                    /*_____________________________________________*/

{int i, val, suppl;
 struct liste *vl;

 MOP("acq_fils");
 for (i = 0;
      i < Nblex;
```

```
      i++)
     {for(vl = nil,fscanf(pisa,"%d\n%d\n",&val,&suppl);
          val != -1;
          fscanf(pisa,"%d\n%d\n",&val,&suppl))
          vl = ajq_list(vl,alloc_list(val,suppl));
      Tlex[i].isa = vl;
     }
}


/*_____________________________________________________________________*/
/*_____________________________Les peres_____________________________*/
/*_____________________________________________________________________*/

acq_pere()

                    /*_____________________________________________*/
                    /* acquisition des peres(lien CLASS)  pour   */
                    /* chaque concept.                           */
                    /*_____________________________________________*/

{int i, val, suppl;
 struct liste *vl;

 MOP("acq_pere");
 for (i = 0;
      i < Nblex;
      i++)
     {for(vl = nil,fscanf(pclass,"%d\n%d\n",&val,&suppl);
          val != -1;
          fscanf(pclass,"%d\n%d\n",&val,&suppl))
          vl = ajq_list(vl,alloc_list(val,suppl));
      Tlex[i].class = vl;
     }
}

/*_____________________________________________________________________*/
/*_____________________________Les noeuds____________________________*/
/*_____________________________________________________________________*/

acq_noeud()

                    /*_____________________________________________*/
                    /* Acquisition des noeuds dans le fichier    */
                    /* logique pnoeud.                           */
                    /*_____________________________________________*/

{int n,cd,rs,sg,sf;

 MOP("acq_noeud");
 Nbnoeud = -1;
 for (fscanf(pnoeud,"%d ",&n);
      !feof(pnoeud);
      fscanf(pnoeud,"%d ",&n))
     {fscanf(pnoeud,"%d %d %d %d\n",&cd,&rs,&sg,&sf);
      Tnoeud[n].code = cd;
      Tnoeud[n].reseau = rs;
      Tnoeud[n].signe = sg;
      Tnoeud[n].suffixe = sf;
      Tnoeud[n].eqrp = n;
      Tnoeud[n].eqls = n;
      Tlex[rs].form = ajq_list(Tlex[rs].form,alloc_list(n));
      Nbnoeud = MAX(Nbnoeud,n);
     }
 Nbnoeud++;
}
```

```
/*_____________________________________________________________________*/
/*_____________________Les liens_____________________________________*/
/*_____________________________________________________________________*/

acq_lien()

                        /*_____________________________________________*/
                        /* Acquisition des liens pour chaque noeud,*/
                        /* ces liens sont stockes dans le fichier  */
                        /* logique plien. Ce sont des liens de  */
                        /* chainage (av & ar).                     */
                        /*_____________________________________________*/

{struct liste *vl;
 int l,n;

 MOP("acq_lien");
 init_mat(Texcl,T_NOEUD);
 for (n = 1;
      ((n <= Nbnoeud) || (!feof(plien)));n++)
     {for (vl = nil, fscanf(plien,"%d\n",&l);
           l != -1;
           fscanf(plien,"%d\n",&l))
           vl = ajq_list(vl,alloc_list(l));
      Tnoeud[n].chav = vl;
      for (vl = nil, fscanf(plien,"%d\n",&l);
           l != -1;
           fscanf(plien,"%d\n",&l))
           vl = ajq_list(vl,alloc_list(l));
      Tnoeud[n].charr = vl;
      for (fscanf(plien,"%d\n",&l);
           ((l != -1)&&(!(feof(plien))));
           fscanf(plien,"%d\n",&l))
           set_mat(Texcl,n,l);
     }
}

/*_____________________________________________________________________*/
/*_____________________Les chaines___________________________________*/
/*_____________________________________________________________________*/

acq_chaine()

                        /*_____________________________________________*/
                        /* Acquisition des chaines dans le fichier  */
                        /* logique pchaine.                        */
                        /*_____________________________________________*/

{int i;

 MOP("acq_chaine");
 Nbchaine = -1;
 strcpy(Tchaine[0],"");
 for (fscanf(pchaine,"%d ",&i);
      !feof(pchaine);
      fscanf(pchaine,"%d ",&i))
     {fscanf(pchaine,"%s\n",Tchaine[i]);
      Nbchaine = MAX(Nbchaine,i);
     }
 Nbchaine++;
}

/*_____________________________________________________________________*/
/*_____________________Les conditions________________________________*/
/*_____________________________________________________________________*/
```

```
acq_cond()

                        /*_____________________________________________________*/
                        /* Acquisition des conditions dans le                  */
                        /* Logigue pcond.                                      */
                        /*_____________________________________________________*/

{int n, t, a, b, x, y;

 MOP("acq_cond");
 fscanf(pcond,"%d ",&n);
 for (Nbcond = -1;
      !feof(pcond);
      Nbcond = MAX(Nbcond,n))
     {fscanf(pcond,"%d %d %d %d %d\n",&t,&a,&x,&b,&y);
      Tcond[n].test = t;
      Tcond[n].a1 = a;
      Tcond[n].a2 = b;
      Tcond[n].im1 = x;
      Tcond[n].im2 = y;
      fscanf(pcond,"%d ",&n);
     }
 Nbcond++;
}

/*_____________________________________________________________________________*/
/*_____________________________La BD_____________________________________________*/
/*_____________________________________________________________________________*/

acq_bd()
                        /*_____________________________________________________*/
                        /* Acquisition de la bd dans le fichier                */
                        /* Logique pbd.                                        */
                        /*_____________________________________________________*/

{
 for(Nbbd = 0;
     !(feof(pbd));
     fscanf(pbd,"%d ",&(Tbd[Nbbd++])));
}

/*_____________________________________________________________________________*/
/*_____________________________La RC____________________________________________*/
/*_____________________________________________________________________________*/

acq_rc()
                        /*_____________________________________________________*/
                        /* Acquisition de la rc dans le fichier                */
                        /* Logique prc.                                        */
                        /*_____________________________________________________*/

{
 for( Nbrc = 0;
     !(feof(prc));
     fscanf(prc,"%d %d ",&(Trc[Nbrc].pre),&(Trc[Nbrc].post)), Nbrc++);
}

/*_____________________________________________________________________________*/

acq_rs()
                        /*_____________________________________________________*/
                        /* Acquisition de la rs dans le fichier                */
                        /* Logique prs.                                        */
                        /*_____________________________________________________*/
```

```
{
 for( Nbrs = 0;
      !(feof(prs));
      fscanf(prs,"%d %d ",&(Trs[Nbrs].req),&(Trs[Nbrs].cond)), Nbrs++);
}

/*_____________________________________________________________________*/
/*_____________________________Init bf_____________________________*/
/*_____________________________________________________________________*/

init_bf()

                        /*_____________________________________________*/
                        /* Initiallisation de la base des faits avec le */
                        /* contenu de la base de donnees.              */
                        /* Rappelons qu'une classe d'equivalence est   */
                        /* associee a cette base et que sa represen-   */
                        /* tation est associee a Tnoeud (initiallise   */
                        /* dans acq_noeud.                             */
                        /*_____________________________________________*/
{for (Nbf = 0;
      Nbf < Nbbd;
      Nbf++)
     {Tfait[Nbf].acces = Tbd[Nbf];
      Tfait[Nbf].score = MAX_SC;
      Tfait[Nbf].status = F_BD;
      Tfait[Nbf].orig = nil;
      Tfait[Nbf].regle = nil;
     }
 return;
}

/*_____________________________________________________________________*/
/*_____________________precompilation d'une regle_____________________*/
/*_____________________________________________________________________*/

ext_rc(regle)   int regle;

                        /*_____________________________________________*/
                        /* Precompilation d'une regle.                 */
                        /* Ex: france => -etranger   alors             */
                        /* nat(x, france) => nat(x,-etranger)...       */
                        /*_____________________________________________*/

{int mbg, mbd, gen;
 struct liste *vl;

 MOP("ext_rc");
 mbg = Trc[regle].pre;
 mbd = Trc[regle].post;
 gen = inter_fm(mbg,mbd);
 if (NEGATIF(gen) == VRAI) return;
 for (vl = Tlex[gen].gra;
      vl != nil;
      vl = vl->suc)
      creer_rc(regle,vl);
 return;
}

creer_rc(regle,vl) int regle; struct liste *vl;

{
}

/*_____________________________________________________________________*/
/*_____________________precompilation de la base des regles_____________*/
```

```
/*_____________________________________________________________________*/

ext_all_rc()

                        /*_____________________________________________*/
                        /* On cherche a precompiler la base de regle*/
                        /*_____________________________________________*/


{int i;

 MOP("ext_all_rc");
 for (i = 0;
      i < Nbnoeud;
      i++)
      ext_rc(i);
 return;
}

/*---------------------------------------------------------------------*/
/*                      Mise a jour des peres                          */
/*---------------------------------------------------------------------*/

m_a_j_charr()

{ int i;
  struct liste *ptr;

  MOP("m_a_j_char");
  for (i=0; i<Nbnoeud; i++) Tnoeud[i].charr=nil;
  for (i=0; i<Nbnoeud; i++)
      for (ptr=Tnoeud[i].chav; ptr!=nil; ptr=ptr->suc)
          Tnoeud[ptr->val].charr=ajq_list(Tnoeud[ptr->val].charr,
                                          alloc_list(i,i));
}

/*********************************************************************/
/*                    INITIALISATION DU SYSTEME.                     */
/*********************************************************************/

/*_____________________________________________________________________*/
/*_____________________Acquisition globale_____________________________*/
/*_____________________________________________________________________*/

acquerir()

                        /*_____________________________________________*/
                        /* Centralisation de toutes les lectures  */
                        /* sur les fichiers specifiques.          */
                        /*_____________________________________________*/

{acq_lex();
 acq_arg();
 acq_fils();
 acq_pere();
 acq_noeud();
 acq_lien();
 acq_chaine();
 acq_bd();
 acq_rc();
 acq_rs();
 acq_cond();
 ord_cpt();
 invers_arg();
 init_bf();
}
```

```
#include <stdio.h>
#include "const.c"
#include "const2.c"
#include "type.c"
#include "ext.c"

/*-----------------------------------------------------------------------*/
/*                                                                       */
/*      Ce fichier contient la fonction principale du programme de       */
/*                  détection d'incohérence.                             */
/*                                                                       */
/*-----------------------------------------------------------------------*/

traitement(j) int j;

{  int fin,
       n,
       rg;

   struct lliste *ll,
                 *ptrll,
                 *eval();

   struct liste *l;

  init_base_faits(j);
  if (Msg)
     { fprintf(yytrace,"\n\nEtat initial de la base de faits\n\n");
       visu_ft(F_ETOILE);
     }
  fin=FAUX;
  init_trace();
  while (fin!=VRAI)
   {  fin=VRAI;
      for(rg=0;rg<Nbrc;rg++)
        {   ll=eval(rg);
            for (ptrll=ll;ptrll!=nil;ptrll=ptrll->suc)
            if (in_trace(rg,ptrll->liste)==FAUX)
                {  fin=FAUX;
                   n=appliquer_regle(rg,ptrll->liste);
                   ajouter_regle_trace(rg,ptrll->liste);
                   if (in_bf(n)==FAUX)
                      { if (Msg) msg_applic_rg1(rg);
                        eclat(n);
                        a_traiter[rg]=FAUX;
                        if (Msg) visu_ft(F_ETOILE);
                        if (incoherent_bf()==VRAI)
                           { msg_inco_det(rg,ptrll->liste);
                             return;
                           }
                      }
                   else if (Msg) msg_applic_rg2(rg);
                }
        }
     }
    trait(79,'*');ligne();
}

/*-----------------------------------------------------------------------*/

msg_applic_rg1(i) int i;

{ ligne();
  fprintf(yytrace,"application de la regle : ");
  visu_rg(i);
```

```
  ligne();
}

/*-------------------------------------------------------------------------*/

msg_applic_rg2(i) int i;

{ ligne();
  fprintf(yytrace,"application de la regle : ");
  visu_rg(i);
  fprintf(yytrace,"mais le fait y est deja");
}

/*-------------------------------------------------------------------------*/

msg_inco_det (i,l) int i; struct liste *l;

{ ligne();
  fprintf(yytrace,"incoherence detectee lors de ");
  fprintf(yytrace,"l'application de la regle : ");
  visu_rg(i);
  fprintf(yytrace,"avec le jeu de substition \n ");
  visu_list_inst(l);
  ligne();
}

/*-------------------------------------------------------------------------*/
```

```
#include <stdio.h>
#include "const.c"
#include "const2.c"
#include "type.c"
#include "ext.c"

/*************************************************************************/
/* CE FICHIER CONTIENT LES FONCTIONS DE VISUALISATIONS DES               */
/* DIFFERENTES STRUCTURES DE DONNEES UTILISEES POUR LA                   */
/* REPRESENTATION DES OBJETS SUIVANTS:                                   */
/* - LE LEXIQUE ( Tlex )                                                 */
/* - LES CHAINES ( Tchaine )                                             */
/* - LA B.D.  ( Tbd )                                                    */
/* - LES REGLES (Trc )                                                   */
/* - LES RENSEIGNEMENTS ( Trs )                                          */
/* ON L'INCLUERA DANS LE FICHIER main.c AVEC LA COMMANDE:                */
/* #INCLUDE "visu.c"                                                     */
/*************************************************************************/

/*-----------------------------------------------------------------------*/
/*                          LES EXTERNES.                                */
/*-----------------------------------------------------------------------*/

/*************************************************************************/
/* ok(s);                                                                */
/* ligne();                                                              */
/* trait(i,c);                                                           */
/* visu_lex();                                                           */
/* visu_list();                                                          */
/* visu_llist();                                                         */
/* visu_list_inst();                                                     */
/* visu_rec_nd();                                                        */
/* visu_nd();                                                            */
/* visu_tn();                                                            */
/* visu_ch();                                                            */
/* visu_bd();                                                            */
/* visu_rc();                                                            */
/* visu_ft();                                                            */
/* visu_rg();                                                            */
/* visu_tr();                                                            */
/*************************************************************************/

/*************************************************************************/
/* FONCTIONS DE MANIPULATION DE TEXTES ET D'ECRANS.                      */
/*************************************************************************/

/*_______________________________________________________________________*/
/*_____________________________OK?_______________________________________*/
/*_______________________________________________________________________*/

int ok(s)       char *s;

                /*_____________________________________________________*/
                /* Protocole de confirmation.                          */
                /*_____________________________________________________*/

{char buff[2];

 printf("\n%s (o/n):",s);
 scanf("%2s",buff);
 return(((buff[0] == 'o')||(buff[0] == 'y'))?VRAI:FAUX);
}

/*_______________________________________________________________________*/
```

```
/*_____________________RC_____________________*/
/*____________________________________________*/

ligne()

                    /*_____________________________________*/
                    /* Retour chariot.                     */
                    /*_____________________________________*/
{
 fprintf(yytrace,"\n");
}

/*____________________________________________*/
/*_____________________Trait__________________*/
/*____________________________________________*/

trait(i,x)      int i; char x;

                    /*_____________________________________*/
                    /* Trace d'un trait de longueur i      */
                    /*_____________________________________*/
{int j;

 for (j = 1;
      j <= i;
      j++)
     fprintf(yytrace,"%c",x);
 fprintf(yytrace,"\n");
}

/*********************************************/
/*              LE LEXIQUE                   */
/*********************************************/

/*____________________________________________*/
/*_____________________Visu_lex_______________*/
/*____________________________________________*/

visu_lex()

{struct liste *ptr;
 int  i,j,n;
 fprintf(yytrace," \n\n\n");
 fprintf(yytrace,"  Visualisation du lexique\n");
 fprintf(yytrace,"  ------------------------\n");
 fprintf(yytrace," \n\n\n");
 for (i = 0;
      i < Nblex;
      i++)
    {fprintf(yytrace," identifiant : %4d  nom : %30s \n",i,Tlex[i].libelle);
     fprintf(yytrace," ordre %4d \n",Tlex[i].ordre);
     ptr=Tlex[i].isa;
     if (ptr==nil) fprintf(yytrace,"  pas de concept pere. \n");
     else { fprintf(yytrace," concept(s) pere(s) : ");
            while (ptr!=nil) { fprintf(yytrace,"%s ",Tlex[ptr->val].libelle
                               fprintf(yytrace," (%d)   ",ptr->val);
                               ptr=ptr->suc;
                             }
            fprintf(yytrace," \n");
          }
     ptr=Tlex[i].class;
     if (ptr==nil) fprintf(yytrace," pas de concept fils\n");
     else { fprintf(yytrace," concept(s)   fils  : ");
            while (ptr!=nil) { fprintf(yytrace,"%s ",Tlex[ptr->val].libelle
                               fprintf(yytrace," (%d)  ",ptr->val);
                               ptr=ptr->suc;
```

```
                                }
            fprintf(yytrace," \n");
          }
      fprintf(yytrace,"\n");
      ptr=Tlex[i].arg;
      if (ptr==nil) fprintf(yytrace," pas d'argument.");
      else { fprintf(yytrace," %s ","argument(s) : ");
             while (ptr!=nil) { fprintf(yytrace,"%s ",Tlex[ptr->val].libelle
                                fprintf(yytrace," (%d)   ",ptr->val);
                                ptr=ptr->suc;
                              }
             fprintf(yytrace," /");
           }
      fprintf(yytrace,"\n");
      trait(75,'*');
     }
     trait(75,'*');
}

/*_____________________________________________________________________*/
/*_____________________visu_liste_____________________________________*/
/*_____________________________________________________________________*/

visu_list(liste)        struct liste *liste;

                    /*_____________________________________________*/
                    /* Visualisation d'une liste d'entier.         */
                    /*_____________________________________________*/

{struct liste *vt;

   fprintf(yytrace," < ");
    for(vt = liste;
        vt != nil;
        vt = vt->suc)
        fprintf(yytrace,"%d->",vt->val);
    fprintf(yytrace,"%s ",">");
}

/*---------------------------------------------------------------------*/

visu_llist(ll)  struct lliste *ll;

{ struct lliste *l;
  int i;

  for (i=0,
       l=ll;
       l!=nil;
       i++,
       l=l->suc)
       { if (Mop) fprintf("%d \n",i);
         visu_list_inst(l->liste);
         fprintf("\n");
         trait(10,'-');
       }
}

/*********************************************************************/
/*                        UNE LISTE D'INSTANCES                      */
/*********************************************************************/

visu_list_inst(list) struct liste *list;

{ struct liste *listl;
  listl=list;
```

```
  while(list1!=nil)
  { fprintf(yytrace," [%2d]  %s%1d -----> [%2d]  %s%1d\n",
                    list1->val,Tlex[Tnoeud[list1->val].reseau].libelle,
                    Tnoeud[list1->val].suffixe,
                    list1->supp,Tlex[Tnoeud[list1->supp].reseau].libelle,
                    Tnoeud[list1->supp].suffixe);
    list1=list1->suc;
  }
}

/*************************************************************************/
/*                          UNE FORMULE                                  */
/*************************************************************************/

/*_______________________________________________________________________*/
/*_________________Visualisation recursive_______________________________*/
/*_______________________________________________________________________*/

visu_rec_nd(noeud)      int noeud;

                        /*_____________________________________________*/
                        /* Visualisation d'un noeud et appel recursif  */
                        /* sur le lien chainage-avant (chav).          */
                        /* Cette fonction est appelee dans visu_nd qui */
                        /* est plus "jolie".                           */
                        /*_____________________________________________*/

{struct liste *vl;
 char op;

 MOP("visu_rec_nd");
 if (noeud == VIDE) return;
 switch (Tnoeud[noeud].code)
        {case ID_CODE:  fprintf(yytrace,"%s%s%d ",
                                ((Tnoeud[noeud].signe == MOINS)?"-":""),
                                Tlex[Tnoeud[noeud].reseau].libelle,
                                Tnoeud[noeud].suffixe);
                        if (Tnoeud[noeud].chav != nil)
                           fprintf(yytrace,"(");
                        vl=Tnoeud[noeud].chav;
                        while (vl != nil)
                        { visu_rec_nd(vl->val);
                          vl=vl->suc;
                          if (vl != nil) fprintf(yytrace," , ");
                        }
                        if (Tnoeud[noeud].chav != nil)
                            fprintf(yytrace,")");
                        break;
         case CH_CODE:  fprintf(yytrace,"%s",Tchaine[Tnoeud[noeud].suffixe]);
                        break;
         case OU_CODE:
         case ET_CODE:  op = (Tnoeud[noeud].code == ET_CODE)?'&':'|';
                        fprintf(yytrace,"%c(",op);
                        vl=Tnoeud[noeud].chav;
                        while (vl != nil)
                        { visu_rec_nd(vl->val);
                          vl=vl->suc;
                          if (vl != nil) fprintf(yytrace," , ");
                        }
                        fprintf(yytrace,") ");
                        break;
        default:        erreur(6,"visualisation");
                        break;
       }
 return;
}
```

```
/*_____________________________________________________________________*/
/*_____________________Visualisation___________________________________*/
/*_____________________________________________________________________*/

visu_nd(fm)      int fm;

                 /*_____________________________________________________*/
                 /* visualisation d'une formule fm.                     */
                 /*_____________________________________________________*/

{
 MOP("visu_nd");
 visu_rec_nd(fm);
 ligne();
 return;
}

/*********************************************************************/
/*                    LA TABLE DES NOEUDS                            */
/*********************************************************************/

/*_____________________________________________________________________*/
/*_____________________________Visualisation___________________________*/
/*_____________________________________________________________________*/

visu_tn()

                  /*_____________________________________________________*/
                  /* Trace de la table des noeuds pour debuggage.        */
                  /*_____________________________________________________*/

{int i;

 MOP("visu_tn");
 ligne();
 fprintf(yytrace,"%4s%2s%2s%19s%5s%5s",
                 "no","code","sgn","concept","eqrp","eqls");
 ligne();
 for (i = 0;
      i < Nbnoeud;
      i++)
     {fprintf(yytrace,"%4d",i);
      fprintf(yytrace,"%2d",Tnoeud[i].code);
      fprintf(yytrace,"%2s",((Tnoeud[i].signe == PLUS)?"+":"-"));
      switch (Tnoeud[i].code)
             {case ET_CODE:  fprintf(yytrace," &&&&&&&&&&&&&&&&&&");
                             break;
              case OU_CODE:  fprintf(yytrace," |||||||||||||||| ");
                             break;
              case ID_CODE:  fprintf(yytrace,"%19s",
                                     Tlex[Tnoeud[i].reseau].libelle);
                             break;
              case CH_CODE:  fprintf(yytrace,"%19s",
                                     Tchaine[Tnoeud[i].suffixe]);
                             break;
              default      : erreur(0);
                             break;
             }
      fprintf(yytrace,"%5d %5d ",Tnoeud[i].eqrp,Tnoeud[i].eqrp);
      visu_list(Tnoeud[i].chav);
      visu_list(Tnoeud[i].charr);
      fprintf(yytrace,"\n");
     }
}
```

```
/*****************************************************************************/
/*                      LA TABLE DES CHAINES                                 */
/*****************************************************************************/

/*___________________________________________________________________________*/
/*_____________________________Visualisation_____________________________*/
/*___________________________________________________________________________*/

visu_ch()

                    /*_____________________________________________________*/
                    /* Trace de la table des chaines pour debuggage.      */
                    /*_____________________________________________________*/

{int i;

 MOP("visu_ch");
 ligne();
 fprintf(yytrace,"Visualisation de la table des chaines\n");
 trait(75,'*');
 for (i = 0;
      i < Nbchaine;
      i++)
     {fprintf(yytrace,"%3d %s\n",i,Tchaine[i]);
     }
 trait(75,'*');
 trait(75,'*');
 return;
}

/*****************************************************************************/
/*                        LA BASE DE DONNEES                                 */
/*****************************************************************************/

/*___________________________________________________________________________*/
/*_____________________________Visualisation_____________________________*/
/*___________________________________________________________________________*/

visu_bd()
                    /*_____________________________________________________*/
                    /* Visualisation de la base de donnees. (Tbd)         */
                    /*_____________________________________________________*/

{int  i;

 MOP("visu_bd");
 fprintf(yytrace,"Visualisation de la base de donnees\n");
 trait(75,'*');
 for (i = 0;
      i < Nbbd;
      i++)
     {fprintf(yytrace,"\n|%d| ",i);
      visu_nd(Tbd[i]);
     }
 fprintf(yytrace,"\n");
 trait(75,'*');
 trait(75,'*');
 return;
}

/*****************************************************************************/
/*                        LA BASE DE REGLES                                  */
/*****************************************************************************/

/*___________________________________________________________________________*/
/*_____________________________Visualisation_____________________________*/
```

```
/*_____________________________________________________________________*/

visu_rg(i) int i;

                    /*_____________________________________________*/
                    /* Visualisation d'une regle.                  */
                    /*_____________________________________________*/
{
 MOP("visu_rg");
 visu_rec_nd(Trc[i].pre);
 fprintf(yytrace," => ");
 visu_rec_nd(Trc[i].post);
 ligne();
}

/*_____________________________________________________________________*/


visu_rc()

                    /*_____________________________________________*/
                    /* Visualisation de la base des regles.        */
                    /*_____________________________________________*/

{int i;

 MOP("visu_rc");
 fprintf(yytrace,"Visualisation des regles de reecriture\n");
 for (i = 0;
      i < Nbrc;
      i++)
     {fprintf(yytrace,"|%3d| ",i);
      visu_rg(i);
     }
 return;
}

/*********************************************************************/
/*                        LA BASE DE FAITS                           */
/*********************************************************************/

/*_____________________________________________________________________*/
/*_____________________________Visualisation___________________________*/
/*_____________________________________________________________________*/

visu_ft(code)   int code;

                    /*_____________________________________________*/
                    /* Visualisation de la base des faits.         */
                    /*_____________________________________________*/

{int i,n;

 MOP("visu_ft");
 if (Nbf==0) { fprintf(yytrace," base de faits vide\n");
               return;
             }
 n=0;
 ligne();
 for (i=0;
      i<Nbf;
      i++)
     {if ( (Tfait[i].status == code)
         ||(code == F_ETOILE))
         {visu_nd(Tfait[i].acces);
          n++;
```

```
        }
    }
 if (n==0) fprintf(yytrace,"pas de faits de ce code\n");
 ligne();
 return;
}


visu_lst_appar(l,n) struct appar l[]; int n;

{ int i;
  for(i=0;
      i<n;
      i++) {fprintf(yytrace,"%d    ",l[i].regle);
            visu_list_inst(l[i].liste);
            ligne();
           }
}

/*-------------------------------------------------------------------*/
/*                  VISUALISATION DE LA TRACE                        */
/*-------------------------------------------------------------------*/

visu_tr()

{ int i;

  fprintf(yytrace,"trace d'execution");
  if (Nbtr==0) { fprintf(yytrace," vide\n");
                 return;
               }
  else fprintf(yytrace," \n");
  for(i=0;i<Nbtr;i++)

      { fprintf(yytrace,"regle %d\n",Trace[i].regle);
        visu_list_inst(Trace[i].liste);
      }
}

/*****************************F*I*N*************************************/
```