PROLOG II

MANUEL D'EXEMPLES

Henry KANOUI

Mars 1982

PROLOG II est un système qui a été développé par:

A. Colmerauer, H. Kanoui et M. Van Caneghem

Groupe d'Intelligence Artificielle ERA CNRS 363
Faculté des Sciences de Luminy Case 901
70, Route Léon Lachamp
13288 MARSEILLE Cedex 9
tel: (91) 41 01 40

AVANT-PROPOS

Ce manuel fait partie d'une documentation complète décrivant le système Prolog II et qui comprend en outre :

- Le modèle théorique de Prolog II par Alain Colmerauer.
- Le manuel d'utilisation de Prolog II par Michel van Caneghem.
- Le manuel de référence du langage Candide, par Henry Kanoui.
- Le manuel décrivant la machine Micromegas par Henry Kanoui.

Il consiste en une collection de programmes dont plusieurs ont été empruntés à A. Colmerauer, M. van Caneghem et R. Pasero.

Les premiers exemples présentés sont très simples et les explications détaillées qui les accompagnent peuvent constituer une introduction à la programmation en Prolog.

La deuxième partie du manuel est faite d'exemples plus complets, mais avec moins de commentaires, qui montrent quelques applications interessantes de Prolog.

Enfin, quelques courts programmes illustrent les nouvelles possibilités de Prolog.II.

Bien entendu, tous les exemples de ce manuel ont tourné sur la version Apple II de PROLOG II, le plus souvent dans des temps très raisonnables. La plupart d'entre eux se trouvent sur le disque PRO-EX.

1 QUELQUES INFORMATIONS SUR PROLOG

1.1 LES DEBUTS DE PROLOG

Le développement des techniques nées de l'Intelligence Artificielle, leur utilisation dans des domaines d'intéret genéral comme la compréhension des langues naturelles, les systèmes experts, les banques de données se sont faites au travers de langages de programmation spécialisés qui offrent les ressources nécessaires à l'axiomatisation et la résolution des problèmes posés.

C'est ainsi que LISP, langage basé sur le lambda-calcul est né au Etats-Unis au début des années 60. Depuis lors, une somme de travail considérable a été consacrée non seulement au langage lui-meme, mais aussi à son environnement (matériel et logiciel) par les universitaires américains.

Dix ans plus tard A. Colmerauer, dans le but de créer un nouveau langage servant à programmer les problèmes d'analyse et de compréhension de la langue naturelle a été amené à utiliser la Logique du Premier Ordre et la démonstration automatique de théorèmes. La Logique du Premier Ordre sous forme clausale, munie d'une règle d'inférence convenable (principe de résolution de Robinson), est un langage de programmation efficace qui s'est appelé PROLOG. Un programme y est constitué d'un ensemble de relations, et son exécution revient à démontrer une nouvelle relation à partir de celles qui constituent le programme. Le formalisme obtenu est naturel et élégant. Il est en meme temps très puissant et présente des possibilités intrinsèques (non-déterminisme, unification,...) qui, essentielles dans les problèmes d'I.A., ne se retrouvent ni dans les langages classiques, ni dans ceux de la famille de LISP.

PROLOG a été utilisé dans diverses applications d'I.A.:
- communication en langue naturelle avec un ordinateur
- calcul formel
- construction de plans en robotique
- ecriture de compilateurs
- banques de données,
- C.A.O.

....
pour lesquelles il s'est révélé parfaitement adapté.

Le premier interpréteur de PROLOG a été écrit en 1973 dans notre laboratoire par P. Roussel et a eu une forte influence sur ses successeurs.

Il utilisait le principe de non recopie des termes, et de fortes restrictions (démonstration linéaires, ordonnancement des littéraux d'une clause, controle du non-déterminisme, unification sans test d'occurence de variable) l'ont rendu utilisable comme langage de programmation. Ecrit en Fortran, il a été installé sur la plupart des matériels existants et s'est répandu un peu partout (France, Angleterre, Portugal, Espagne, Etats-Unis, Canada, Pologne, Hongrie,...).

Parmi les autres implantations de PROLOG, il faut citer un

compilateur écrit pour DEC-10 par D.Warren. Et puis, fait important, PROLOG tourne sur des micro-ordinateurs: une version précédente a été installée sur un Exorciser (M6800) par nous memes et sur un Sorcerer par F.McCabe.

1.2 PROLOG II

Très vite les programmes écrits en PROLOG sont devenus très gros et complexes. Les limites des implantations existantes ont été atteintes très rapidement, bien sur en ce qui concerne les notions de place mémoire, de facilité d'utilisation et de modularité mais aussi pour certains concepts de base.

Nous avons donc entrepris l'étude d'une nouvelle version de PROLOG qui essaye d'apporter une solution à ces différents problèmes.

On peut mentionner tout particulièrement les points suivants:

PORTABILITE: Grace à l'utilisation d'une machine virtuelle, le nouveau système Prolog est portable sur des ordinateurs de toutes tailles, y compris les 'micros'.

INTERACTIVITE: Un éditeur de clauses est intégré à PROLOG, qui permet de développer entièrement une application sous PROLOG. Il faut remarquer que cet éditeur est écrit en PROLOG.

MODULARITE: L'espace des clauses est organisé en une hiérarchie de sous-mondes structurée en arbre. On dispose également de commandes permettant la manipulation de ces sous-mondes.

EXTENSIBILITE: L'utilisateur a la possibilité d'ajouter des prédicats évaluables qui sont écrits dans le langage de la machine hote. Cette possibilité ne concerne que certains types de prédicats (arithmétique, mise en page, controle de périphériques,...)

FIABILITE: Un gros effort a été fait pour transmettre correctement les erreurs. Toute erreur peut etre récupérée par l'utilisateur, quel que soit le niveau auquel elle s'est produite.

NOUVEAUX CONCEPTS:

i) un type de coroutine par le biais des prédicats geler (qui permet de retarder l'évaluation d'un littéral tant qu'une variable n'est pas instanciée) et dif.

ii) unification étendue aux arbres infinis, ce qui permet de régler d'une certaine facon le test d'occurence.

iii) controle de l'execution par l'introduction du concept de bloc.

iv) structuration compacte des données pour les chaines et les n-uplets.

1.3 IMPLANTATION

Pour pouvoir atteindre notre objectif de portabilité, nous avons défini et réalisé:

-Un langage de haut niveau CANDIDE qui a été spécialement crée pour écrire l'interpréteur PROLOG.

-Une machine virtuelle MICROMEGAS qui execute le code généré

par le compilateur CANDIDE.

L'interpréteur PROLOG est entièrement écrit en CANDIDE (70 pages environ), le code machine produit est donc transportable sur toute machine supportant MICROMEGAS. Un compilateur de CANDIDE a été réalisé sur APPLE II en PASCAL.

Pour pouvoir transporter le système PROLOG sur une autre machine il faut donc écrire:
-un interpréteur ou un compilateur de la machine virtuelle MICROMEGAS en langage machine.
-une gestion des entrées/sorties sur la machine hote, écrite en langage évolué, avec éventuellement une gestion de mémoire virtuelle.
Ceci étant réalisé, il suffit de charger le code de l'interpréteur PROLOG, puis le superviseur PROLOG (écrit en PROLOG), pour avoir un système qui fonctionne.

Une version complète de tout le système a été implantée sur APPLE II.

Cette version bien qu'un peu lente, reste malgré tout très utilisable grace à sa grande interactivité. Une première utilisation de ce système nous a montré que de disposer de PROLOG sur un micro-ordinateur est très stimulant pour écrire des programmes dans ce langage. Tous les exemples donnés ce manuel (et bien d'autres) ont été programmés sur l'Apple.

En plus du développement de programmes PROLOG (avec deux disques on peut écrire 40 pages de PROLOG), cette version permettra aussi de faire enfin un enseignement correct de PROLOG.

2 PRESENTATION INFORMELLE A TRAVERS UN EXEMPLE.

Prolog est un langage de programmation fait pour representer et utiliser les connaissances que l'on a sur un certain domaine. Plus exactement, le domaine est un ensemble d'objets et la connaissance est materialisée par un ensemble de relations qui décriront à la fois les propriétés de ces objets et leurs interactions.

Un ensemble de règles décrivant ces objets et ces relations constitue un programme Prolog.

Par exemple, dire "Jean est le père de Paul" n'est rien d'autre qu'affirmer qu'une relation (etre père) lie deux objets (designés par leur nom: Jean et Paul), et qu'on pourrait ecrire:

est-père-de(Paul,Jean)

De meme, une question du genre : "Qui est le père de Paul ?" revient à chercher si la relation "est-père-de" lie Paul à un autre objet qui sera la réponse à notre question.

Remarquons que si on définit une relation entre des objets, l'ordre dans lequel les objets sont donnés est significatif :

est-père-de(Paul,Jean) est different de est-père-de(Jean,Paul).

Pour exprimer la relation précédente, nous avons utilisé des identificateurs pour nommer les objets et la relation qui les lie.

Le nom de la relation (est-père-de) est appelé "predicat", les objets sur lesquels porte la relation sont les "arguments".

2.1 PREMIER EXEMPLE: AU RESTAURANT.

Pour illustrer la facon dont les choses se passent en Prolog, nous allons considérer un premier exemple qui décrit la Carte d'un restaurant.

Les objets qui nous interessent sont les mets que l'on peut y consommer, et une première série de relations donne la classification de ces mets en hors d'oeuvres, plat à base de viande ou de poisson, et desserts.

Cette carte constitue une petite banque de données qui s'écrit comme suit:

```
" la carte "

hors-d-oeuvre(Artichauts-Melanie) ->;
hors-d-oeuvre(Truffes-sous-le-sel) ->;
hors-d-oeuvre(Cresson-oeuf-poche) ->;

viande(Grillade-de-boeuf) ->;
viande(Poulet-au-tilleul) ->;

poisson(Bar-aux-algues) ->;
poisson(Chapon-farci) ->;

dessert(Sorbet-aux-poires) ->;
dessert(Fraises-chantilly) ->;
dessert(Melon-en-surprise) ->;
```

Les relations qui y sont définies introduisent à la fois les objets (les mets proposés) et leur classification. Par exemple :

hors-d-oeuvre(Cresson-oeuf-poche) -> ;

indique que le Cresson-oeuf-poche est un hors d'oeuvre, et rien de plus.

En réalité, ce premier type de règle se résume à énoncer de simples "assertions".
Dès que l'on a une telle collection d'assertions, on peut poser des questions qui les concernent.
Une question du genre :

"est-ce que le Cresson-oeuf-poche est un hors d'oeuvre ?"

se traduira par :

hors-d-oeuvre(Cresson-oeuf-poche) ;

On cherche alors si cette assertion fait partie de celles qui sont connues : la réponse est oui.

Par contre,

hors-d-oeuvre(Salade-de-tomates) ;

recevra une réponse négative, puisque la banque ne contient pas une telle assertion.

Supposons maintenant que l'on désire savoir quelles sont les hors d'oeuvres que l'on peut consommer. Il serait fastidieux de poser la suite des questions:

```
hors-d-oeuvre(Salade-de-tomates) ;
...
hors-d-oeuvre(Artichauts-Melanie) ;
...
hors-d-oeuvre(Fraises-chatilly) ;
...
```

et d'attendre une réponse oui ou non à chaque fois. On préférerait demander :

Quels sont les hors d'oeuvres ?

ou mieux: Quels sont les objets "e" qui sont des hors d'oeuvres?

sans pour cela connaitre l'objet que "e" représente.
Ici le nom "e" ne désigne pas un objet particulier, mais tout objet appartenant à l'ensemble (eventuellement vide) de ceux qui possèdent la propriété d'etre un hors d'oeuvre et que l'on demande au programme de determiner.

On dit que "e" est une "variable".

Dans notre cas la question se traduit par:

```
hors-d-oeuvre(e) ;
```

et Prolog repond par :

```
e=Artichauts-Melanie
e=Truffes-sous-le-sel
e=Cresson-oeuf-poche
```

qui n'est rien d'autre que l'ensemble des objets que peut désigner la variable "e" pour que l'assertion soit vérifiée.

A partir des relations qui constituent la banque de données initiale, on peut construire des relations plus complexes et plus générales. Par exemple, à partir des relations "viande( )" et "poisson( )" qui expriment que l'argument est un plat à base de viande ou de poisson, on peut définir la relation "plat( )" qui dit que "un plat est un plat à base de viande ou de poisson" et qu'on écrit :

```
plat(p) -> viande(p) ;
plat(p) -> poisson(p) ;
```

et qu'on lit : p est un plat si p est un plat à base de viande ;
p est un plat si p est un plat à base de poisson.

Ici encore, on utilise une variable, "p", qui dans chacune des deux règles désigne respectivement tous les plats à base de viande et tous les plats à base de poisson.

Précisons que la portée d'une variable est restreinte à la règle dans laquellle elle est définie, et donc que la variable "p" de la première règle n'est pas liée à la variable "p" de la deuxième règle.

Dans notre exemple, la question "quels sont les plats ?" traduite par:

```
plat(p) ;
```

provoque les réponses :

```
p=Grillade-de-boeuf
p=Poulet-au-tilleul
p=Bar-aux-algues
p=Chapon-farci
```

Intéressons nous maintenant à la composition d'un repas : suivant les règles habituelles, un repas est constitué d'un hors d'oeuvre, d'un plat principal (viande ou poisson), et d'un dessert.

Un repas est donc un triplet : "e,p,d" où "e" est un hors d'oeuvre, "p" un plat et "d" un dessert. Ceci est exprimé très naturellement par la règle:

```
repas(e,p,d) -> hors-d-oeuvre(e) plat(p) dessert(d) ;
```

qui se lit :

"e,p,d" satisfont la relation "repas" si "e" satisfait la

relation "hors-d-oeuvre", et si "p" satisfait la relation "plat" et si "d" satisfait la relation "dessert".

D'une facon générale, nous avons défini une nouvelle relation comme la conjonction de trois autre relations.

A la question "quels sont les repas ?", c'est à dire :

```
repas(e,p,d) ;
```

Prolog répond par :

```
e=Artichauts-Melanie  p=Grillade-de-boeuf  d=Sorbet-aux-poires
e=Artichauts-Melanie  p=Grillade-de-boeuf  d=Fraises-chantilly
...
e=Artichauts-Melanie  p=Chapon-farci  d=Melon-en-surprise
e=Truffes-sous-le-sel  p=Grillade-de-boeuf  d=Sorbet-aux-poires
...
e=Truffes-sous-le-sel  p=Chapon-farci  d=Melon-en-surprise
e=Cresson-oeuf-poche  p=Grillade-de-boeuf  d=Sorbet-aux-poires
...
e=Cresson-oeuf-poche  p=Chapon-farci  d=Melon-en-surprise
```

qui est la liste des 36 combinaisons possibles.

Gardons le meme ensemble de relations et posons une question un peu plus précise : on veut connaitre les repas comportant du poisson en plat principal. Ceci se traduit par :

```
repas(e,p,d) poisson(p) ;
```

qui exprime bien la conjontion des deux conditions que nous voulons voir remplies. Comme tout à l'heure, Prolog calculera des valeurs pour les variables e,p,d telles que la premiere condition repas(e,p,d) soit vérifiée. Remarquons qu'avant l'évaluation de la relation repas( ), les variables e,p,d n'ont encore recu aucune valeur.
Ce n'est plus le cas après l'évaluation de repas( ) : e,p,d ont été affectées par certaines valeurs, par exemple:

```
e=Artichauts-Melanie, p=Grillade-de-boeuf, d=Sorbet-aux-poires).
```

Les varíables ont pris une valeur et "p" en particulier, lorsqu'on passe à la deuxième relation "poisson(p)" qui, avec la valeur indiquée pour "p", devient alors:

```
poisson(Grillade-de-boeuf).
```

La banque ne contenant pas cette assertion, le jeu de valeurs proposé pour e,p,d ne satisfait pas notre question : c'est un echec et l'on essaie la solution suivante.

Finalement le programme imprime les 18 solutions possibles :

```
e=Artichauts-Melanie  p=Bar-aux-algues  d=Sorbet-aux-poires
e=Artichauts-Melanie  p=Bar-aux-algues  d=Fraises-chantilly
e=Artichauts-Melanie  p=Bar-aux-algues  d=Melon-en-surprise
e=Artichauts-Melanie  p=Chapon-farci  d=Sorbet-aux-poires
e=Artichauts-Melanie  p=Chapon-farci  d=Fraises-chantilly
e=Artichauts-Melanie  p=Chapon-farci  d=Melon-en-surprise
e=Truffes-sous-le-sel  p=Bar-aux-algues  d=Sorbet-aux-poires
```

```
...
e=Truffes-sous-le-sel  p=Chapon-farci  d=Melon-en-surprise
e=Cresson-oeuf-poche  p=Bar-aux-algues  d=Sorbet-aux-poires
...
e=Cresson-oeuf-poche  p=Chapon-farci  d=Melon-en-surprise
```

Quelques remarques :

- Pour satisfaire une conjonction de relations, on les examine de la gauche vers la droite.
- Au cours de l'execution, certaines variables peuvent recevoir une valeur. Si une variable recoit une valeur, toutes ses ocurences prennent la meme valeur.
- Dans une relation il n'y a pas de distinction entre arguments d'entrée et arguments de sortie (un argument d'entrée est un argument dont on connait la valeur avant d'evaluer la relation; un argument de sortie est un argument auquel est affectée une valeur durant l'évaluation de la relation.
Au contraire, une meme relation peut jouer plusieurs roles : si tous ses arguments sont connus, on ne fait que vérifier si la relation est satisfaite ou non; si certains de ses arguments sont inconnus, on calcule l'ensemble des valeurs qu'on peut leur donner pour satisfaire la relation.
- L'execution est non-deterministe : on calcule tous les jeux de valeurs des arguments qui satisfont la relation.

Complétons un peu notre savoir sur la consommation des repas et introduisons la valeur calorique de chacun des mets proposés.

```
" valeur calorique pour une portion "

calories(Artichauts-Melanie,150) ->;
calories(Cresson-oeuf-poche,202) ->;
calories(Truffes-sous-le-sel,212) ->;
calories(Grillade-de-boeuf,532) ->;
calories(Poulet-au-tilleul,400) ->;
calories(Bar-aux-algues,292) ->;
calories(Chapon-farci,254) ->;
calories(Sorbet-aux-poires,223) ->;
calories(Fraises-chantilly,289) ->;
calories(Melon-en-surprise,122) ->;
```

L'assertion:

```
calories(Chapon-farci,254) -> ;
```

s'interprète comme : "la portion de Chapon-farci servie apporte 254 calories".

Pour connaitre la valeur calorique des hors d'oeuvres, on posera :

```
hors-d-oeuvre(e) calories(e,c) ;
```

Pour chaque valeur de "e" satisfaisant la relation hors-d-oeuvre( ), le programme affectera à la variable c le nombre qui satisfait la relation calories(e,).

Les reponses sont :

```
e=Artichauts-Melanie  c=150
```

```
e=Truffes-sous-le-sel   c=212
e=Cresson-oeuf-poche   c=202
```

Un consommateur plus curieux voudra connaitre la valeur calorique totale de son repas. Pour cela, on définira la relation :

```
valeur(e,p,d,v) ->
    calories(e,x)
    calories(p,y)
    calories(d,z)
    ajouter(x,y,z,v) ;
```

où v est la somme des valeurs caloriques des constituants du repas. Pour connaitre ces valeurs, on pose la question :

```
repas(e,p,d) valeur(e,p,d,v) ;
```

qui donne :

```
e=Artichauts-Melanie   p=Grillade-de-boeuf d=Sorbet-aux-poires
   v=905
e=Artichauts-Melanie   p=Grillade-de-boeuf d=Fraises-chantilly
   v=971
e=Artichauts-Melanie   p=Grillade-de-boeuf d=Melon-en-surprise
   v=804
...
e=Artichauts-Melanie   p=Chapon-farci   d=Melon-en-surprise
   v=526
e=Truffes-sous-le-sel   p=Grillade-de-boeuf d=Sorbet-aux-poires
   v=967
e=Truffes-sous-le-sel   p=Grillade-de-boeuf d=Fraises-chantilly
   v=1033
...
e=Truffes-sous-le-sel   p=Chapon-farci   d=Melon-en-surprise
   v=588
e=Cresson-oeuf-poche   p=Grillade-de-boeuf d=Sorbet-aux-poires
   v=957
e=Cresson-oeuf-poche   p=Grillade-de-boeuf d=Fraises-chantilly
   v=1023
...
e=Cresson-oeuf-poche   p=Chapon-farci   d=Melon-en-surprise
   v=578
```

On est alors tout naturellement conduit à definir un repas équilibré par:

```
repas-equilibre(e,p,d) ->
    repas(e,p,d)
    valeur(e,p,d,v)
    inferieur(v,800) ;
```

qui exprime qu'un repas equilibre est un repas dont la valeur calorique est inferieure a 800 calories.

La question:

```
repas-equilibre(e,p,d) ;
```

donne la liste :

```
e=Artichauts-Melanie  p=Poulet-au-tilleul  d=Sorbet-aux-poires
e=Artichauts-Melanie  p=Poulet-au-tilleul  d=Melon-en-surprise
...
e=Truffes-sous-le-sel  p=Bar-aux-algues  d=Sorbet-aux-poires
...
e=Cresson-oeuf-poche  p=Chapon-farci  d=Melon-en-surprise
```

Terminons avec cet exemple par la question :

```
repas-equilibre(e,p,d) viande(p) ;
```

qui permet de choisir un repas équilibré à base de viande et qui peut etre:

```
e=Artichauts-Melanie  p=Poulet-au-tilleul d=Sorbet-aux-poires
e=Artichauts-Melanie  p=Poulet-au-tilleul d=melon-en-surprise
e=Truffes-sous-le-sel p=Poulet-au-tilleul d=melon-en-surprise
e=Cresson-oeuf-poche  p=Poulet-au-tilleul d=melon-en-surprise
```

ce qui bannit tout plat contenant du boeuf !

Voici le programme complet:

```
" la carte "

hors-d-oeuvre(Artichauts-Melanie) ->;
hors-d-oeuvre(Truffes-sous-le-sel) ->;
hors-d-oeuvre(Cresson-oeuf-poche) ->;

viande(Grillade-de-boeuf) ->;
viande(Poulet-au-tilleul) ->;

poisson(Bar-aux-algues) ->;
poisson(Chapon-farci) ->;

dessert(Sorbet-aux-poires) ->;
dessert(Fraises-chantilly) ->;
dessert(Melon-en-surprise) ->;

" plat de resistance "

plat(p) -> viande(p);
plat(p) -> poisson(p);

" composition d'un repas "

repas(e,p,d) -> hors-d-oeuvre(e) plat(p) dessert(d);

" valeur calorique pour une portion "

calories(Artichauts-Melanie,150) ->;
calories(Cresson-oeuf-poche,202) ->;
calories(Truffes-sous-le-sel,212) ->;
calories(Grillade-de-boeuf,532) ->;
calories(Poulet-au-tilleul,400) ->;
calories(Bar-aux-algues,292) ->;
```

```
calories(Chapon-farci,254) ->;
calories(Sorbet-aux-poires,223) ->;
calories(Fraises-chantilly,289) ->;
calories(Melon-en-surprise,122) ->;

" valeur calorique d'un repas "

valeur(e,p,d,v) ->
   calories(e,x)
   calories(p,y)
   calories(d,z)
   ajouter(x,y,z,v);

" repas equilibre "

repas-equilibre(e,p,d) ->
   repas(e,p,d)
   valeur(e,p,d,v)
   inferieur(v,800);

" divers "

ajouter(a,b,c,d) -> val(add(a,add(b,c)),d);

inferieur(x,y) -> val(inf(x,y),1);
```

2.2 OU L'ON REVOIT LES CHOSES DE PLUS PRES.

L'exemple précédent nous a permis d'aborder quelques points caractéristiques de Prolog :

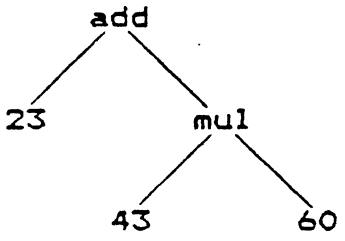
- Définitions de relations entre des objets.
- Questions concernant ces relations.
- Notion de variable.
- Représentation de conjonctions et de disjontions de relations.
- Non-déterminisme de l'execution.
- Pas de distinctions entre arguments données et arguments résultats.

Nous allons maintenant revenir avec plus de détails sur ces points.

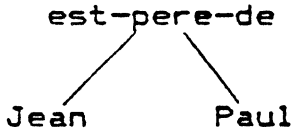
2.2.1 LES ARBRES.

Dans notre exemple les seuls objets manipulés étaient des constantes, représentes par leur nom (Artichauts-Melanie) ou par leur valeur (254). Parfois, des variables nous ont servi à désigner des objets encore inconnus.

En fait, la structure la plus générale des objets manipulés par Prolog est la structure d'arbre. Par exemple :

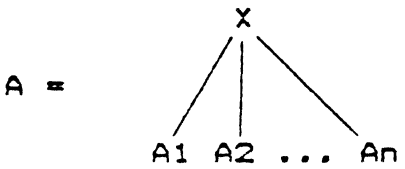


est un arbre qui représente l'expression arithmétique 23+45*60.



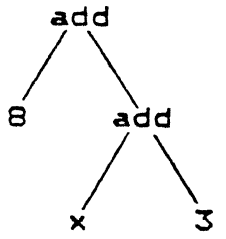
est un arbre qui represente la relation est-pere-de(Jean,Paul).

Plus généralement, un arbre A est constitué d'un noeud X appelé racine et d'un ensemble (eventuellement vide) ordonné d'éléments A1,...,An, qui sont eux-meme des arbres :



A1,...,An sont des sous-arbres de A, et tout sous-arbre de Ai est aussi un sous-arbre de A. Les racines de A1,...,An sont les descendants de X. Tout noeud sans descendants est dit terminal ou feuille.

En Prolog, un arbre peut etre partiellement inconnu. Dans ce cas, une de ses feuilles se reduit à une variable. Par exemple:



est un tel arbre. Il représente l'infinité des arbres obtenus en remplacant x par un arbre quelconque.

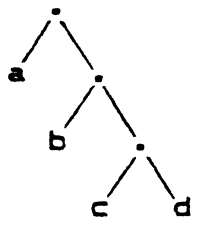
Un arbre sans variable peut se reduire à sa racine : l'objet associe est une constante : indentificateur chaine de caractères ou entier.

La notation linéaire de l'arbre ci-dessus est :

add(8,add(x,3) ou en notation "n-uplet" <add,8,<add,x,3>>.

Parmi les autres structures de donnees, la liste autorise un opérateur infixé "." parenthésé de droite à gauche :

la liste a.b.c.d représente l'arbre



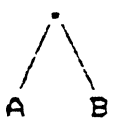
2.2.2 L'UNIFICATION.

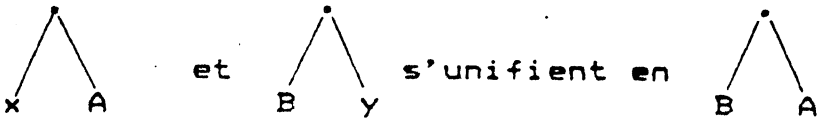
L'opération principale effectuée sur les arbres est l'unification. Etant donnés deux arbres comportant eventuellement des parties variables, l'unification consiste à trouver, s'il elles existent, les valeurs que doivent prendre les variables présentes dans l'un ou l'autre des arbres pour que ceux-ci coincident. L'ensemble des egalités (variable=valeur) qui rendent les deux arbres égaux est appelée substitution.

Par exemple, les arbres :

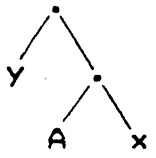
et s'unifient par la substitution (x=B)

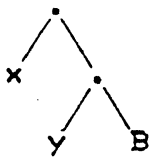
La forme commune des deux arbres est :



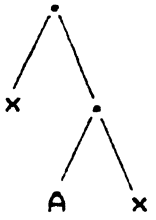


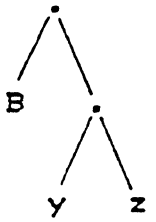
par la substitution ((x=B),(y=A)).

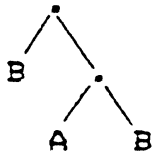
Les arbres 

et 

ne s'unifient pas car il faudrait : ((x=y),(x=B),(y=A)).

Les arbres 

et 

s'unifient en 

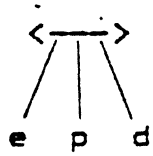
avec : ((x=B),(y=A),(z=B)).

2.3 STRUCTURE D'UN PROGRAMME PROLOG.

En général, un prédicat peut etre défini par un mélange de règles et d'assertions et un programme Prolog consiste donc en une suite de règles.
Chaque règle comprend un membre gauche (ou tete de règle) et un membre droit (ou queue de règle) éventuellement vide, reliés par le connecteur "->".
La tete de règle se réduit à un seul littéral (prédicat suivi de ses arguments), la queue de règle est une suite de littéraux.

Un programme Prolog est construits à l'aide de "termes" : les règles,les litteraux, leurs arguments sont des termes. Un terme peut etre une constante (identificateur, nombre, chaine de caractères), une variable ou un terme structuré (n-uplet).
La notion de n-uplet est utile lorsqu'on veut regrouper ensemble plusieurs termes pour en construire un nouveau que l'on peut alors manipuler plus commodément. Par exemple, au chapitre précédent, nous avions défini un repas comme étant constitué de trois composants: un hors d'oeuvre "e", un plat "p", un dessert "d". Il est agréable de pouvoir regrouper ces trois composants pour former une nouvelle entité, le repas, qui est le triplet <e,p,d>.

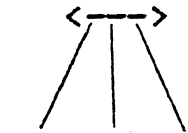
Ce nouveau terme a une structure d'arbre qui est :



e p d

où <——> représente le symbole fonctionnel à 3 places.

Quand le premier composant d'un n-uplet (avec n supérieur ou égal à 2) est un identificateur, on permet de le représenter sous forme fonctionnelle. Ainsi:



add x 34

est la meme chose que

x 34

Bien sur, l'unification tient compte de cette identité.

2.3.1 SYNTAXE D'UN PROGRAMME

Dans les règles de syntaxe hors contexte qui suivent :

Le signe de réecriture est ::= et le membre gauche d'une règle n'est pas répété lorsqu'il est identique à celui de la règle précédente.

Les terminaux sont des caractères et sont effectivement représentés par des caratères isolés, sauf le caractère d'espacement représenté par le mot espace.

Les non-terminaux sont des suites de mots entourées des signes < et >. Nous prévenons le lecteur que les caractères < et > interviennent aussi en tant que symboles terminaux. Dans ce cas ils apparaissent isolément.

```
<caractère>
   ::= <caractère spécial>
   ::= <lettre>
   ::= <chiffre>

<caractère spécial>
   ::= +
   ::= -
   ::= '
   ::= .
   ::= ,
   ::= ;
   ::= =
   ::= "
   ::= /
   ::= (
   ::= )
   ::= <
   ::= >
   ::= espace

<chaine>
   ::= " <suite de caractères> "

<chiffre>
   ::= 0
   ::= 1
   .....
   ::= 9

<commande>
   ::= <suite de termes> ;

<commentaire>
   ::= <chaine>
```

```
.constante>
   ::= <identificateur>
   ::= <chaine>
   ::= <entier>

<énonce>
   ::= <commentaire>
   ::= <règle>

<entier>
   ::= <chiffre> <suite de chiffres>

<identificateur>
   ::= <mot long>
   ::= <identificateur> - <mot>

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

<minuscule>
   ::= a
   ::= b
   .....
   ::= z

<majuscule>
   ::= A
   ::= B
   .....
   ::= Z

<mot>
   ::= <mot court>
   ::= <mot long>

<mot court>
   ::= <lettre> <suite de chiffres>
   ::= <mot court> '

<mot long>
   ::= <lettre> <mot court>
   ::= <lettre> <mot long>

<parasite>
   ::= /
   ::= <syntaxe inconnue de l'utilisateur>

<programme>
   ::= ;
   ::= <énonce> <programme>

<regle>
   ::= <terme> - > <suite de termes> ;
```

```
<suite de caractères>
   ::= <vide>
   ::= <caractère> <suite de caractères>

<suite de chiffres>
   ::= <vide>
   ::= <chiffre> <suite de chiffres>

<suite de termes>
   ::= <vide>
   ::= <terme> <espace> <suite de termes>
   ::= <parasite> <espace> <suite de termes>

<terme>
   ::= <terme simple>
   ::= <terme simple> . <terme>

<terme simple>
   ::= ( <terme> )
   ::= <variable>
   ::= <constante>
   ::= < >
   ::= < <terme> >
   ::= < <terme> , <terme> >
   ::= < <terme> , <terme> , <terme> >
   ..............................
   ::= <identificateur> ( <terme> )
   ::= <identificateur> ( <terme> , <terme> )
   ::= <identificateur> ( <terme> , <terme> , <terme> )
   ..........................................

<variable>
   ::= <mot court>
   ::= <variable> - <mot>

<vide>
   ::=
```

Le caractère " doit etre doublé à l'intérieur d'une chaine.

Sans altérer en quoi que ce soit le sens des choses écrites:

-des espaces peuvent etre insérés à tout endroit, sauf à l'intérieur des constantes et des variables.

-des espaces peuvent etre enlevés de tout endroit, sauf à l'intérieur d'une chaine et sauf si cela provoque la création de nouvelles constantes ou variables par agglutination d'anciennes.

2.4 COMMENT MARCHE PROLOG : L'EFFACEMENT.

Après avoir décrit le monde des objets sur lesquels on travaille, et les relations qui les lient, on va poser des questions au programme en lui demandant d'établir si une relation (ou plus généralement une conjonction de relations) est satisfaite pour certaines valeurs des arguments.

Instinctivement, Prolog va tenter "d'effacer" la suite des termes représentant ces relations qui sont traités de la gauche vers la droite, en transportant et en complétant au fur et à mesure qu'il avance dans ce travail les substitutions des variables qui permettent cet effacement.
S'il y arrive , c'est cet ensemble de substitutions qui constituera la réponse à la question posée.
Dans le cas où plusieurs substitutions permettent l'effacement, c'est à dire s'il y a plusieurs réponses possibles, toutes les solutions seront calculées et exhibées.

Le principe d'effacement peut s'expliquer comme suit :

Une règle du programme telle que :

```
P(...) -> Q(...) R(...) ;
```

peut s'interpréter par :

"pour effacer P(...) , effacez Q(...) puis R(...)".

Une règle comme :

```
S(...) -> ;
```

s'interprète par : "S(...) s'efface".

Une question comme :

```
S(...) T(...) ;
```

s'interprète par : "effacez S(...) puis T(...)".

Pour satisfaire le premier but, effacez S(...), on cherche parmi les règles du programme et de haut en bas, la première dont la tete s'unifie avec S(...) moyennant quelques substitutions sur les variables figurant soit dans le terme à effacer, soit dans la tete de la règle choisie.

Si Q1(...)...Qp(...) est la queue de la règle choisie pour effacer S(...), notre nouveau but est d'effacer :

```
Q1(...)...Qp(...) T(...)
```

modulo les substitutions nécessaires.

On continue ainsi de la gauche vers la droite jusqu'a ce que:
- tout soit effacé : c'est un succès et la solution est matérialisée par la substitution qui a permi l'effacement.
- sinon on bute sur un terme qu'on ne peut effacer : c'est un echec.

Dans les deux cas, on ne s'arrete pas là : rappelons nous que pour effacer un terme on commence par choisir la première règle qui permettait de la faire. D'autres possibilités restent peut-etre ouvertes et au cours de notre progression nous avons laissé des choix en attente. Il y a alors un retour en arrière, on reconsidère le dernier choix effectué et on va tenter d'effacer le terme concerné d'une autre facon, de manière à essayer de trouver une autre réponse à notre question.

On procède ainsi tant qu'il reste des choix en attente. Il est important de remarquer que lorsqu'on effectue un nouveau choix pour tenter d'effacer un terme L(...) on "défait" toutes les substitutions des variables qui ont eu lieu entre le précédent choix fait pour L(...) et l'instant présent.

Pour illustrer ces mécanismes, reprenons le début de notre exemple :

(1) viande(Grillade-de-boeuf) ->;
(2) viande(Poulet-au-tilleul) ->;

(3) poisson(Bar-aux-algues) ->;
(4) poisson(Chapon-farci) ->;

" plat de resistance "

(5) plat(p) -> viande(p);
(6) plat(p) -> poisson(p);

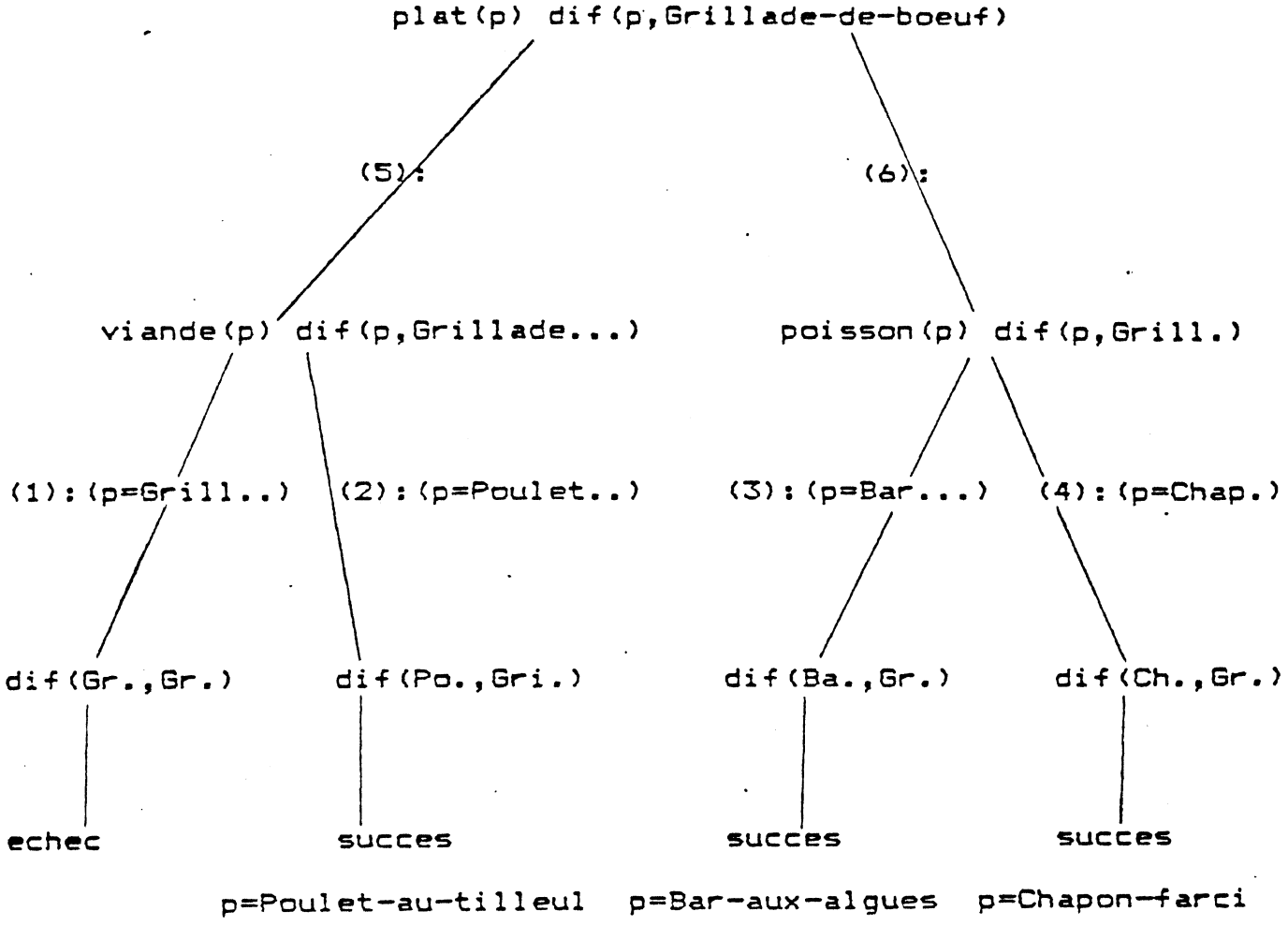
avec la question : plat(p) dif(p,Grillade-de-boeuf);

qui constitue notre ensemble iniatial de buts et où "dif" est un terme qui s'efface si et seulement si ses arguments sont différents.

9037

Nous représenterons l'effacement par un arbre où :

- à un noeud est associée l'ensemble courant des termes à effacer (les buts).
- à une branche sont associées la règle choisie pour effacer le premier terme et la substitution courante.
- les successeurs d'un noeud sont les nouveaux ensembles de buts pouvant etre engendrés par l'effacement du premier des buts associés au noeud considéré.


plat(p) dif(p,Grillade-de-boeuf)
(5):
(6):
viande(p) dif(p,Grillade...)
poisson(p) dif(p,Grill.)
(1):(p=Grill..)
(2):(p=Poulet..)
(3):(p=Bar...)
(4):(p=Chap.)
dif(Gr.,Gr.)
dif(Po.,Gri.)
dif(Ba.,Gr.)
dif(Ch.,Gr.)
echec
succes
succes
succes
p=Poulet-au-tilleul
p=Bar-aux-algues
p=Chapon-farci

2.4.1 LES REGLES PREDEFINIES.

Certaines regles sont connues d'office par le système au lieu d'etre définies par l'utilisateur :elles sont appelées "règles prédéfinies" et offrent des facilités qui ne pourraient pas en géneral etre obtenues en Prolog pur. En fait, c'est l'existence de ces règles prédéfinies qui fait que Prolog est un langage de programmation utilisable.

Ces règles ont souvent des "effets de bord", c'est à dire que lorsqu'elles servent à effacer un terme, non seuleument les arguments du terme effacé peuvent etre modifiés, mais aussi le contexte du systeme peut etre modifié.

Elles concernent principalement:

-Le controle de l'execution : les lois qui régissent l'effacement, le non-determinisme,... sont modifiées.
-Les entrées/sorties qui permettent au programme de communiquer avec l'exterieur.
-Des test sur le type des objets.
-L'arithmétique.
-Les sous-mondes.
-Les coroutines.

LE FAMEUX "/"

Nous avons vu que l'effacement d'un terme se fait de facon non-déterministe, le système gardant en réserve les divers points de choix pour y revenir ulterieurement.

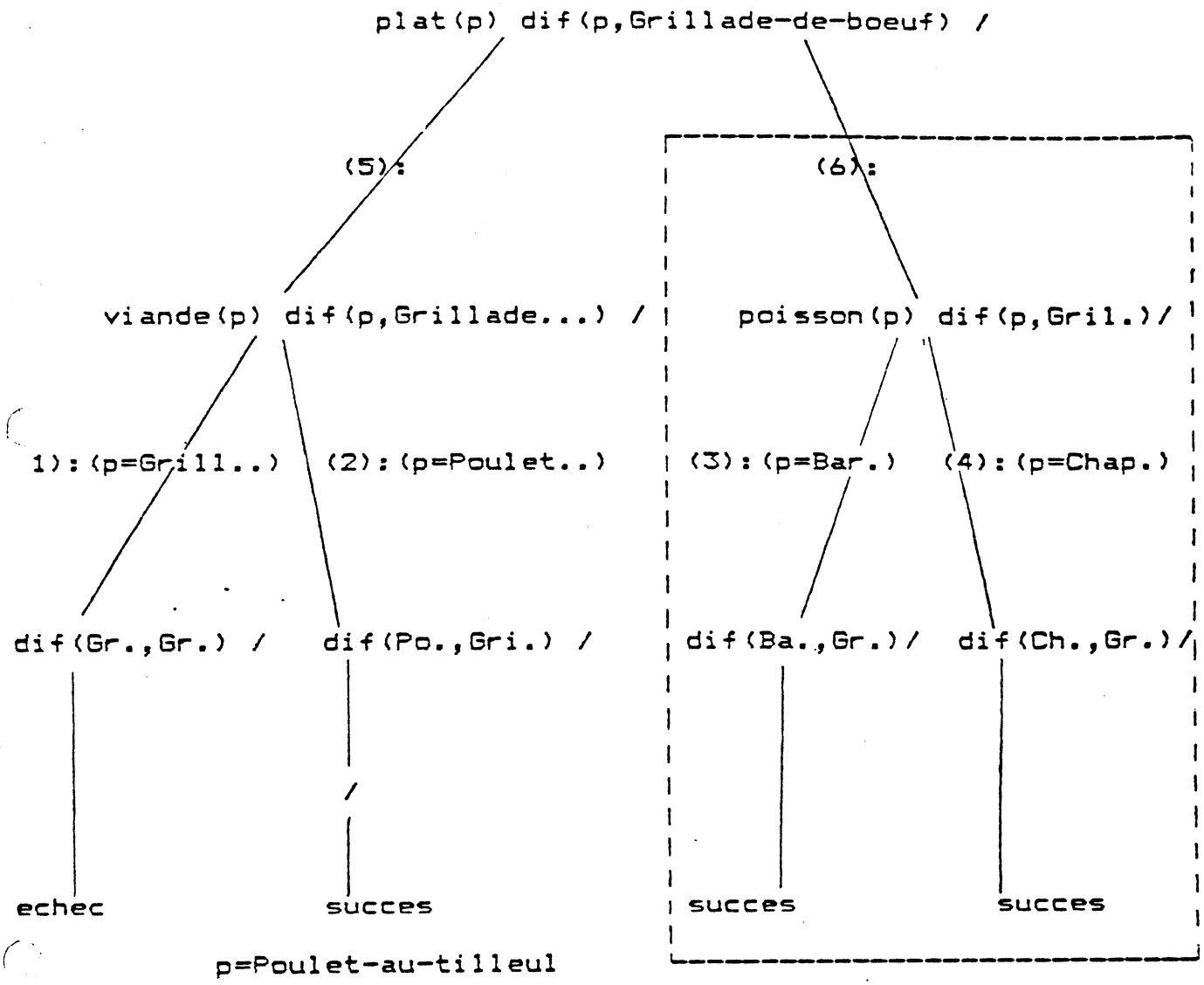
L'introduction du "/" dans une clause permet de supprimer certains de ces points de choix et, à la limite, de rendre un programme entièrement déterministe. Il s'agit là d'un mécanisme primordial de Prolog. La règle d'utilisation du "/" est très simple et s'énonce comme suit :

"l'effacement du prédicat "/" a pour effet de supprimer tous les choix en attente pour tous les littéraux de la résolvante à partir de celui qui a activé la clause où figure le "/" et celui qui précède le "/" dans cette clause".

Par exemple si un "/" figure dans une question comme :

plat(p) dif(p,Grillade-de-boeuf) / ;

l'arbre précédent devient :

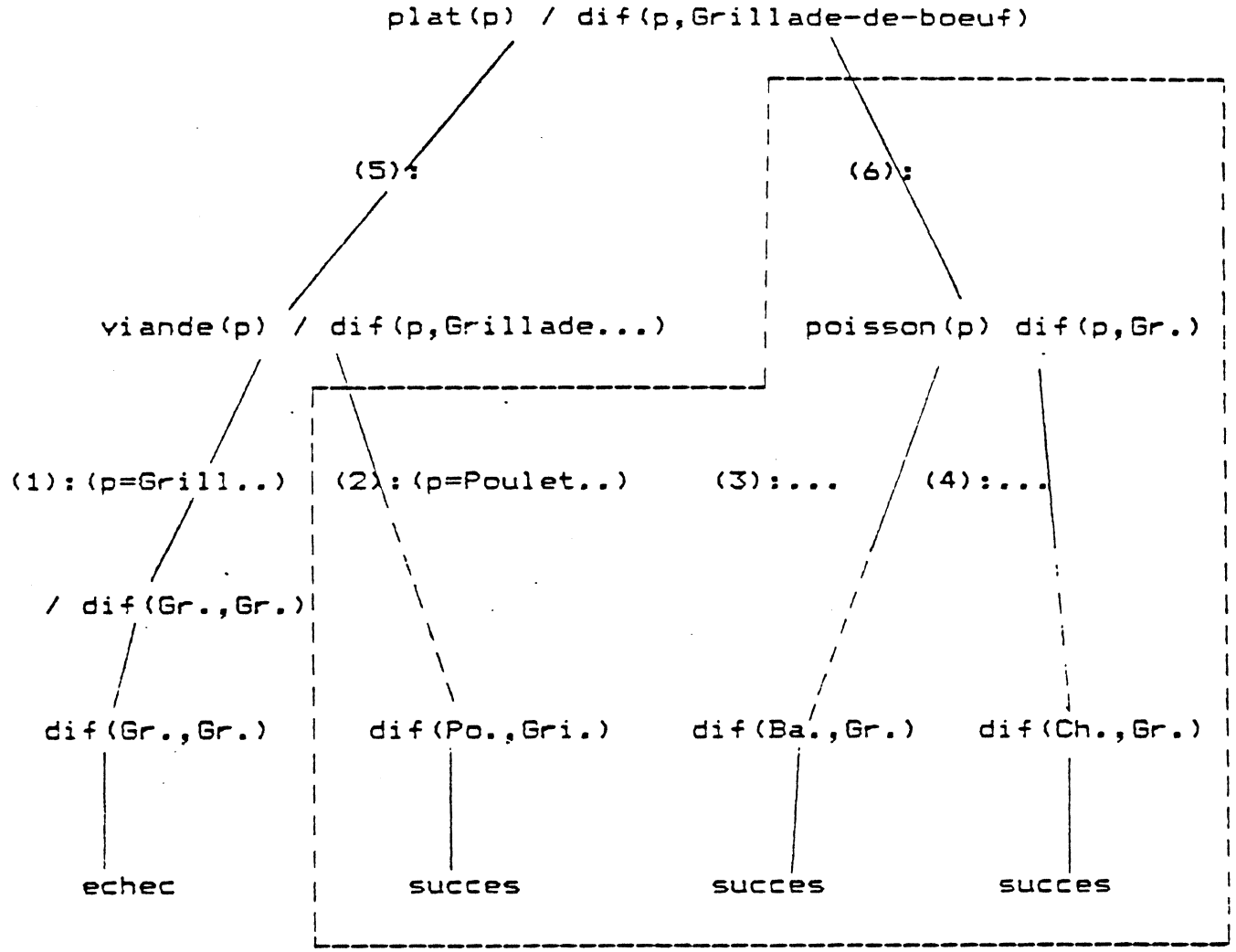


L'effacement du "/" a pour effet de supprimmer tous les choix en attente pour plat(...) : toute la partie encadrée de l'arbre est occultée et il n'y a plus qu'une seule réponse à notre question, c'est la première obtenue.

Si on avait posé la question :

plat(p) / dif(p,Grillade-de-boeuf) ;

notre arbre aurait eu l'allure suivante :



L'effacement du "/" éliminant la partie encadrée de l'arbre, il n'y a plus aucune réponse qui satisfait notre question.

-A-			
add	*	(<entier1>,<entier2>)	val(add(3,4),7);
affecter		(<ident>,<constante>)	affecter(aa,3);
ajout		(<<tete>,<queue>>)	ajout(<vrai(x),nil>);
arg		(<entier>,<terme1>,<terme2>)	arg(2,<aa,bb>,bb);
aujourd-hui;			
-B-			
bas;			
bloc		(<terme1>,<terme2>)	bloc(x,editer);
bonsoir;			
boucle;			
boum		(<ident>,<chaine>)	boum(toto,"toto");
-C-			
car-apres		(<car>)	car-apres(",");
car-apres'		(<car>)	car-apres'(",");
chaine		(<terme>)	chaine("toto");
-D-			
date		(<entier>)	date(21);
		(<chaine>)	date(lurdi);
		(<entier>,<chaine>)	date(1,novembre);
		(<entier>,<chaine>,<entier>)	date(1,janvier,1982);
def-tab		(<ident>,<entier>)	def-tab(pile,100);
descendre		(<entier>)	descendre(27);
		(<chaine>)	descendre("ordinaire");
dico;			
dif		(<terme1>,<terme2>)	dif(toto,lulu);
div	*	(<entier1>,<entier2>)	val(div(8,2),4);
-E-			
echo;			
editer;			
eg		(<terme1>,<terme2>)	eg(x,toto);
entier		(<terme>)	entier(2056489);
entree		(<chaine>)	entree("console");
en-xy		(<entier1>,<entier2>)	en-xy(5,5);
eq	*	(<entier1>,<entier2>)	val(eq(3,3),1);
etat;			
ex		(<terme>)	ex(a.b);
exm		(<chaine>)	exm("bonjour");
-FGH-			
fermer-sortie;			
fin-bloc		(<terme>)	fin-bloc(erreur);
fin-ligne		(<car>)	fin-ligne(";");
geler		(<variable>,<terme>)	geler(x,liste(x));
haut;			
-I-			
ident		(<terme>)	ident(toto);
in		(<terme>)	in(x);
in-car		(<car>)	in-car(c);
in-car'		(<car>)	in-car'(c);
in-chaine		(<chaine>)	in-chaine(c);
in-entier		(<entier>)	in-entier(e);
in-ident		(<ident>)	in-ident(i);
in-ph		(<liste>)	in-ph(l);

```
inf              * (<constante1>,<constante2>)    val(inf("to","zou"),1);
inserer;

-L-
lg-ligne           (<entier>)                     lg-ligne(40);
libre              (<terme>)                      libre(x);
ligne;
liste-des          (<variable>,<liste>,<terme>,<liste>);
lister             (<entier>)                     lister(10000);

-MN-
mod              * (<entier1>,<entier2>)          val(mod(7,3),1);
monter             (<entier>)                     monter(10);
                   (<chaine>)                     monter("origine");
monde              (<chaine>);
ms-err             (<entier>,<chaine>);
mul              * (<entier1>,<entier2>)          val(mul(2,3),6);
neuf;
no-car             (<car>,<entier>)               no-car("1",49);

-P-
page;
papier;
pointeurs;
pos                (<entier>)                     pos(27);
pris               (<terme>)                      pris(x);
purger;

-RS-
renommer           (<chaine1>,<chaine2>)          renommer("to","lulu");
sans-boucle;
sans-papier;
sans-trace;
si               * (<terme1>,<terme2>,<terme3>)   val(si(eq(3,4),1,2),2);
sortie             (<chaine>)                     sortie("resultat");
sourd;
sous-mondes        (<liste>)                      sous-mondes(1);
sub              * (<entier1>,<entier2>)          val(sub(8,6),2);
supprimer          (<entier>)                     supprimer(1);

-TV-
tampon-neuf        (<terme>)                      tampon-neuf(pp(x));
tasser;
tete               (<ident>)                      tete(toto);
trace;
tuer-monde         (<chaine>)                     tuer-monde("toto");
val                (<terme>,<terme>)              val(3,3);
```

3 EXEMPLES COMMENTES.

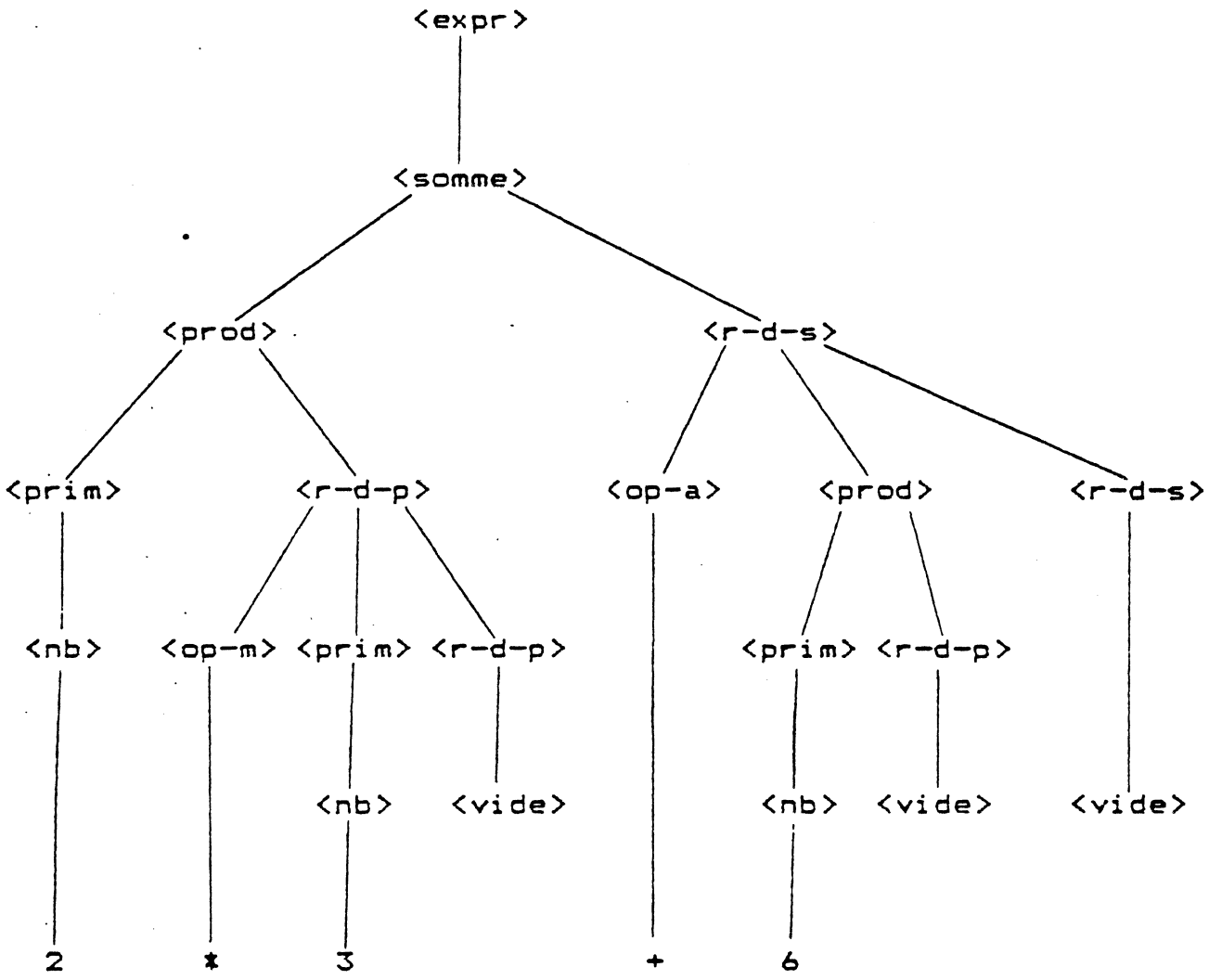
3.1 LES GRAMMAIRES.

La grammaire d'un langage est un ensemble de règles qui permet de déterminer si oui ou non une suite de mots construits sur un certain alphabet appartient au langage.
Si oui, on peut alors mettre en évidence la structure sous jacente de la phrase analysée.

Les langages "hors-contexte" forment une classe importante: voici par exemple la grammaire définissant l'ensemble de toutes les expressions arithmétiques que l'on peut construire sur les nombres entiers, les opérateurs +,-,* avec leurs priorités habituelles :

```
(0)  <expression> ::= <somme>

(1)  <somme> ::= <produit> <reste de somme>

(2)  <produit> ::= <primaire> <reste de produit>

(3)  <primaire> ::= <nombre>
(4)               ::= ( <expression> )

(5)  <reste de somme> ::= <op add> <produit> <reste de somme>
(6)                     ::= <vide>

(7)  <reste de produit> ::= <op mul> <primaire> <reste de produit>
(8)                       ::= <vide>

(9)  <op mul> ::= *

(10) <op add> ::= +
(11)            ::= -

(12) <nombre> ::= 0
                ::= 1
                ::= 2
.....

(13) <vide> ::=
```

Par exemple, la décomposition de l'expression: 2 * 3 + 6 est la suivante (les noms des symboles non-terminaux ont été abrégés) :

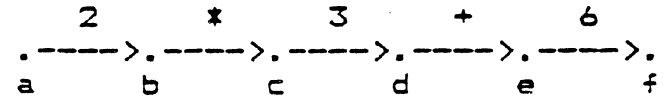


3.1.1 REPRESENTATION DE LA GRAMMAIRE EN PROLOG.

Examinons comment écrire cet analyseur en Prolog.

Pour cela, considérons la chaine d'entrée comme un graphe: le premier sommet du graphe est placé au début de la phrase, et on ajoute un sommet après chaque mot de la phrase. Chacun des mots va alors étiqueter l'arc qui joint les deux sommets qui l'encadrent.

Avec notre exemple nous obtenons:



Les règles de la grammaire peuvent alors etre vues comme des instructions permettant de compléter ce graphe.

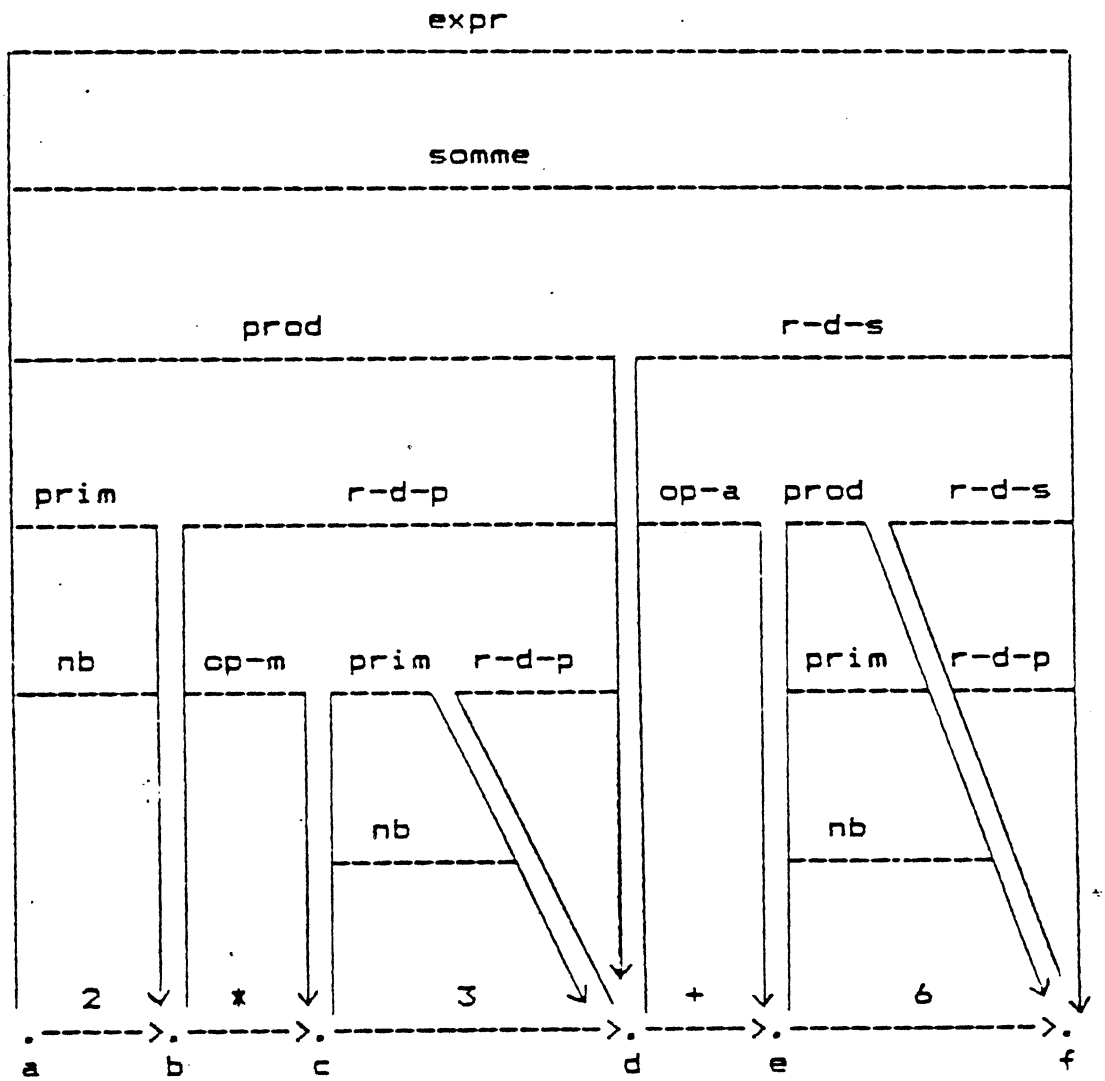
C'est ainsi qu'une règle comme (12) sera interprétée par:

-si il y a un arc etiqueté par "2" entre le noeud x et le noeud y du graphe, rajouter entre x et y un arc etiqueté "nombre".

Une règle comme (1) donnera:

-si il y a un arc étiqueté "produit" entre les noeuds x et y et un arc étiqueté "reste de somme" entre les noeuds y et z, rajouter un arc étiqueté "expression" entre x et z.

Avec ces conventions, analyser une phrase revient à chercher un arc étiqueté "expression" entre le premier et le dernier sommet du graphe associé. Si on y parvient, la phrase sera acceptée, sinon elle sera rejetée. Voici le graphe complété pour notre exemple :



Ecrivons maintenant le programme en Prolog: à chaque symbole non-terminal N de la grammaire, nous associerons un prédicat du meme nom:

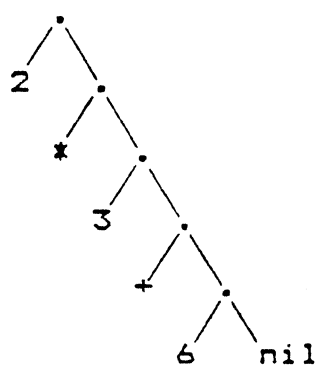
N(<x,y>) sera interprétée par:

"il existe un arc étiqueté N entre les noeuds x et y du graphe".

Pour la règle (1) ceci donne la clause :

somme(<x,y>) -> produit(<x,z>) reste-de-somme(<z,y>) ;

Revenons un peu sur le graphe; celui-ci peut etre simplement représenté par la liste des mots constituant la phrase d'entrée:



sommet a
sommet b
sommet c
sommet d
sommet e
sommet f

chacun des sommets du graphe correspondant à un sous arbre de l'arbre précédent.

Avec cette facon de faire, dans N(<x,y>), x représente la chaine d'entrée "avant" d'effacer N et y représente la chaine restant à analyser "après" avoir effacé N.

De plus, cette représentation permet de traduire les règles terminales comme (12) par:

nombre(<x,y>) -> mot(n,<x,y>) entier(n) ;

qui utilise la règle standard :

mot(a,<a.x,x>) -> ;

et le prédicat évaluable entier.

Enfin, reconnaitre si une phrase appartient au langage défini par notre grammaire revient à effacer :

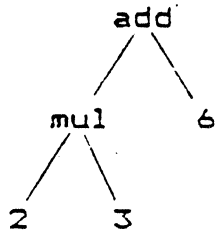
expression(<p,nil>)

où p représente la phrase d'entrée sous forme de liste.

3.1.2 MISE EN EVIDENCE DE LA STRUCTURE PROFONDE.

Nous sommes donc rendus au point où nous savons traduire une grammaire en un ensemble de clauses qui diront si oui ou non une phrase appartient au langage.

Nous pouvons faire mieux en complétant nos relations pour faire ressortir l'arbre construit par l'analyse de la phrase :



pour l'exemple précédent.

Pour cela, nous n'aurons besoin que d'ajouter un argument aux prédicats associés aux non-terminaux de la grammaire. Celui-ci ne fera qu'exprimer comment une phrase est construite à partir des sous-phrases qui la composent.

Nous changeront ainsi :

```
     nombre(<x,y>) -> mot(n,<x,y>) ;
en:  nombre(n,<x,y>) -> mot(n,<x,y>) ;

et   primaire(<x,y>) -> nombre(<x,y>) ;
en:  primaire(n,<x,y>) -> nombre(n,<x,y>) ;
```

et ainsi de suite.

Finalement, notre règle de départ deviendra :

```
expression(e,<x,y>) -> produit(p,<x,z>) reste-de-somme(p,e,<z,y>) ;
```

où e représentera l'arbre associé à l'expression analysée.

Voici le programme définitif:

```
" grammaire des expressions "

expression(e,<x,y>) -> somme(e,<x,y>) ;

somme(e,<x,y>) -> produit(p,<x,z>) reste-de-somme(p,e,<z,y>) ;

produit(p,<x,y>) -> primaire(f,<x,z>) reste-de-produit(f,p,<z,y>) ;

primaire(n,<x,y>) -> mot(n,<x,y>) entier(n) ;

primaire(e,<x,y>) -> mot("(",<x,z>) expression(e,<z,t>)
                     mot(")",<t,y>) ;

reste-de-somme(p,e,<x,y>) -> mot(o,<x,z>) op-add(o,o-a)
     produit(p',<z,t>) reste-de-somme(<o-a,p,p'>,e,<t,y>) ;
reste-de-somme(e,e,<x,x>) -> ;

reste-de-produit(f,p,<x,y>) -> mot(o,<x,z>) op-mul(o,o-m)
     primaire(f',<z,t>) reste-de-produit(<o-m,f,f'>,p,<t,y>) ;
reste-de-produit(f,f,<x,x>) -> ;

mot(a,<a.x,x>) -> ;

op-add("+",add) -> ;
op-add("-",sub) -> ;

op-mul("*",mul) -> ;

" lecture "

lire(nil) -> car-apres'(".") / in-car'(".");
lire(a.b) -> in-ident(a) / lire(b);
lire(a.b) -> in-entier(a) / lire(b);
```

```
lire(a.b) -> in-ident(a) / lire(b);
lire(a.b) -> in-entier(a) / lire(b);
lire(a.b) -> in-car'(a) lire(b) ;

" lancement "

en-route ->
   exm("l'expression ")
   lire(p)
   analyse(p,e)
   val(e,f)
   exm("       vaut   ") ex(f);

analyse(p,e) -> expression(e,<p,nil>) / ;
analyse(p,e) -> exm("...    est incorrecte") impasse ;
```

On utilise le prédicat "lire" qui lit une phrase terminée par "." et la transforme en une liste de mots .

Par exemple :

```
>lire(p) ;
```

si on tape :

```
12+(23-4)*5+210-34-43.
```

Prolog répond :

```
p=12."+"."(".23."-".4.")"."*".5."+".210."-".34."-".43.nil
```

Le prédicat "expression" construit l'arbre syntaxique associé à la phrase lue (si elle appartient au langage) :

```
>lire(p) expression(e,<p,nil>) ;
```

après avoir tapé:

```
12+(23-4)*5+210-34-43.
```

on obtient :

```
p=12."+"."(".23."-".4.")"."*".5."+".210."-".34."-".43.nil

e=sub(sub(add(add(12,mul(sub(23,4),5)),210),34),43)
```

On peut ensuite communiquer cet arbre au prédicat évaluable "val" qui calcule et imprime sa valeur. Tout ceci est fait par :

```
>en-route;

l'expression 12+(23-4)*5+210-34-43.
       vaut  240
```

3.2 DERIVATION FORMELLE.

En analyse mathématique, la dérivation est une opération qui à une expression algébrique associe une autre expression algébrique appelée dérivée.

Les expressions sont construites sur les nombres, les opérateurs +,-,*,^, des symboles fonctionnels comme SIN, COS, ... et un certain nombre d'inconnues. La dérivation se fait par rapport à l'une de ces inconnues.

Dans notre exemple, les nombres seront toujours entiers nous n'auront qu'une inconnue : x, et les seuls symboles fonctionnels unaires seront "sin" et "cos".

La première partie du programme est la grammaire d'analyse de ces expressions: c'est celle de l'exemple précédent qui a été complétée.

La voici :

```
" grammaire des expressions "

expression(e,<x,y>) -> somme(e,<x,y>) ;

somme(e,<x,y>) -> produit(p,<x,z>) reste-de-somme(p,e,<z,y>);

produit(p,<x,y>) -> facteur(f,<x,z>) reste-de-produit(f,p,<z,y>);

facteur(f,<x,y>) -> primaire(p,<x,z>) reste-de-facteur(p,f,<z,y>);

primaire(n,<x,y>) -> mot(n,<x,y>) entier(n);
primaire("x",<x,y>) -> mot("x",<x,y>);
primaire(<o-u,p>,<x,y>) -> mot(o,<x,z>) op-un(o,o-u)
                           primaire(p,<z,y>);
primaire(e,<x,y>) -> mot("(",<x,z>) expression(e,<z,t>)
                     mot(")",<t,y>);

reste-de-somme(p,e,<x,y>) ->
    mot(o,<x,z>)
    op-add(o,o-a)
    produit(p',<z,t>)
    reste-de-somme(<o-a,p,p'>,e,<t,y>);
reste-de-somme(e,e,<x,x>) ->;

reste-de-produit(f,p,<x,y>) ->
    mot(o,<x,z>)
    op-mul(o,o-m)
    facteur(f',<z,t>)
    reste-de-produit(<o-m,f,f'>,p,<t,y>);
reste-de-produit(f,f,<x,x>) ->;

reste-de-facteur(p,f,<x,y>) ->
    mot(o,<x,z>)
    op-exp(o,o-e)
    primaire(p',<z,t>)
    reste-de-facteur(<o-e,p,p'>,f,<t,y>);
reste-de-facteur(f,f,<x,x>) ->;

mot(a,<a.x,x>) ->;
```

```
op-add("+",add) ->;
op-add("-",sub) ->;

op-mul("*",mul) ->;

op-exp("^",exp) ->;

op-un("-",sub) ->;
op-un(sin,sin) ->;
op-un(cos,cos) ->;

" regles de derivation "

derivee(x,x,1) -> ;
derivee(n,x,0) -> entier(n) ;
derivee(add(u,v),x,add(u',v')) -> derivee(u,x,u') derivee(v,x,v');
derivee(sub(u,v),x,sub(u',v')) ->  derivee(u,x,u') derivee(v,x,v');
derivee(mul(u,v),x,add(mul(u,v'),mul(v,u'))) ->
   derivee(u,x,u')
   derivee(v,x,v');
derivee(exp(u,n),x,mul(n,mul(u',exp(u,sub(n,1))))) ->
   derivee(u,x,u');
derivee(sub(u),x,sub(u')) -> derivee(u,x,u');
derivee(sin(u),x,mul(u',cos(u))) -> derivee(u,x,u');
derivee(cos(u),x,sub(mul(u',sin(u)))) -> derivee(u,x,u');

" regles de simplification "

simplifier(<o-b,x,y>,u) ->
   simplifier(x,x')
   simplifier(y,y')
   simp(o-b,x',y',u);
simplifier(<o-u,x>,u) -> simplifier(x,x') simp(o-u,x',u);
simplifier(x,x) ->;

simp(add,0,x,x) -> ;
simp(add,x,0,x) -> ;
simp(sub,x,0,x) -> ;
simp(sub,0,x,y) -> simp(sub,x,y);
simp(mul,0,x,0) -> ;
simp(mul,x,0,0) -> ;
simp(mul,1,x,x) -> ;
simp(mul,x,1,x) -> ;
simp(exp,x,0,1) -> ;
simp(mul,sub(x),y,u) ->
   simp(mul,x,y,v)
   simp(sub,v,u);
simp(mul,x,sub(y),u) ->
   simp(mul,x,y,v)
   simp(sub,v,u);
simp(exp,x,1,x) -> ;
simp(exp,0,x,0) -> ;
simp(exp,1,x,1) -> ;
simp(o-b,x,y,u) -> dif(o-b,exp) entier(x) entier(y)
                   val(<o-b,x,y>,u);
simp(o-b,x,<o-b,u,v>,t) -> simp(o-b,x,u,z) simp(o-b,z,v,t);
simp(o-b,x,y,<o-b,x,y>) ->;
simp(sub,0,0) -> ;
simp(sub,sub(x),x) -> ;
simp(sin,0,0) -> ;
```

```
simp(cos,0,1) -> ;
simp(o-u,x,<o-u,x>) ->;

" lecture "

lire(nil) -> car-apres'(".") / in-car'(".");
lire(a.b) -> in-ident(a) / lire(b);
lire(a.b) -> in-entier(a) / lire(b);
lire(a.b) -> in-car'(a) lire(b);

" ecriture "

ecrire(<o-b,x,y>) ->
   /
   op-bin(o,o-b)
   exm("(")
   ecrire(x)
   exm(o)
   ecrire(y)
   exm(")");
ecrire(sub(x)) -> / exm("-") ecrire(x);
ecrire(<o-u,x>) -> / op-un(o,o-u) ex(o) ecrire(x);
ecrire(x) -> chaine(x) / exm(x) ;
ecrire(x) -> ex(x);

op-bin(o,o-b) -> op-add(o,o-b);
op-bin(o,o-b) -> op-mul(o,o-b);
op-bin(o,o-b) -> op-exp(o,o-b);

" lancement "

en-route ->
   exm("l'expression   ")
   lire(p)
   analyse(p,e)
   derivee(e,"x",e')
   simplifier(e',f)
   exm("a pour derivee ")
   ecrire(f)
   ligne
   /;

analyse(p,e) -> expression(e,<p,nil>) / ;
analyse(p,e) -> exm(" ...    est incorrecte ") impasse ;
```

Voici par exemple le résultat de l'analyse d'une expression :

Après la commande :

```
>lire(p) expression(e,<p,nil>) ;
```

on tape :

```
3*x^2+6*x+5.
```

et on obtient :

```
p=3."*"."x"."^".2."+".6."*"."x"."+".5.nil
```

```
e=add(add(mul(3,exp("x",2)),mul(6,"x")),5)
```

Nous avons ensuite défini la relation de dérivation :

derivee(f,x,f') signifie: f' est la dérivée de f par rapport à x.

Cette relation comporte une clause par opérateur ou symbole fonctionnel et s'exprime de manière très naturelle.
Par exemple, la dérivée de l'expression précédente est donnée par:

```
>lire(p) expression(e,<p,nil>) derivee(e,"x",e') ;
```

en tapant :

```
3*x^2+6*x+5.
```

on obtient:

```
.......

e'=add(add(add(mul(3,mul(2,mul(1,exp("x",sub(2,1))))),mul(exp(
   "x",2),0)),add(mul(6,1),mul("x",0))),0)
```

Pour rendre les choses plus lisibles, nous avons introduit le prédicat "ecrire" dont l'effet est d'écrire linéairement l'arbre recu en argument, avec peut-etre quelques parenthèses superflues.
Voici ce que cela produit pour la valeur de e' ci-dessus:

```
((((3*(2*(1*(x^(2-1)))))+((x^2)*0))+((6*1)+(x*0)))+0)
```

Comme on le voit, le résultat est loin d'etre simplifié!
Nous avons donc adjoint un programme de simplification (très incomplet!) qui permet d'obtenir une écriture plus condensée.
Tout cela est résumé par le prédicat "en-route" dont voici un exemple d'utilisation :

```
>en-route;

l'expression   3*x^2+6*x+5.

a pour derivee ((6*x)+6)


>en-route;

l'expression   cos(-3*x^2+2).

a pour derivee ((6*x)*sin(-(3*(x^2))+2))
```

3.3 UN PEU D'ARITHMETIQUE.

Le problème suivant a été proposé par L. Baxter:

Trouver une suite de chifrres décimaux d1,...dn tels que le nombre<d1d2...dn5> multiplié par 5 s'écrive <5d1d2...dn>.

Nous allons voir que pour résoudre ce problème, nous n'écrirons pas un algorithme spécifique comme on le ferait avec un langage de programmation classique. Au contraire, nous nous contenterons d'énoncer que très peu de relations très générales qui ne feront qu'axiomatiser la division par 2. Et comme ces règles sont générales, elles permettront de résoudre d'autres problèmes que celui qui est posé.

Revenons donc à notre problème et remarquons tout d'abord que multiplier un nombre par 5 revient à le multiplier par 10 puis à diviser le résultat par 2, et que multiplier un nombre par 10 revient à lui ajouter le chiffre 0 à la fin.
Notre problème s'exprime donc ainsi :

trouver d1,...,dn tels que d1d2...dn50 : 2 = 5d1d2...dn.

Voici le programme complet suivi de quelques explications:

```
un peu d'arithmetique

"table de multiplication par 2"

deux-fois(0,0,0.0) ->;
deux-fois(0,1,0.1) ->;
deux-fois(1,0,0.2) ->;
deux-fois(1,1,0.3) ->;
deux-fois(2,0,0.4) ->;
deux-fois(2,1,0.5) ->;
deux-fois(3,0,0.6) ->;
deux-fois(3,1,0.7) ->;
deux-fois(4,0,0.8) ->;
deux-fois(4,1,0.9) ->;
deux-fois(5,0,1.0) ->;
deux-fois(5,1,1.1) ->;
deux-fois(6,0,1.2) ->;
deux-fois(6,1,1.3) ->;
deux-fois(7,0,1.4) ->;
deux-fois(7,1,1.5) ->;
deux-fois(8,0,1.6) ->;
deux-fois(8,1,1.7) ->;
deux-fois(9,0,1.8) ->;
deux-fois(9,1,1.9) ->;

"moitie(x,y) : y est la moitie de x"

moitie(<0.x,x>,<y,y>) ->;
moitie(<c.a.x,u>,<b.y,v>) ->
   deux-fois(b,c',c.a)
   moitie(<c'.x,u>,<y,v>);

" queue(l,l') : l' est la queue de la liste l "
```

```
queue(1,1) ->;
queue(x.l,l') -> queue(l,l');
```

Nous aurons besoin d'axiomatiser la division par 2. Pour cela, il nous faut la table de multiplication par 2 avec report, qui s'énonce par les premières règles :
deux-fois(c,r,d.u), c étant un chiffre décimal, r valant 0 ou 1 signifie: 2 * c + r = du.

Par exemple : 2 * 9 + 1 = 19 .

L'énoncé du problème suggère que l'on doit pouvoir accéder aux chiffres qui composent un nombre à la fois par le début et par la fin. Convenons donc de représenter un nombre par un doublet de deux listes, le nombre effectivement représenté par un tel doublet étant composé des chifrres obtenus en faisant la différence de ces deux listes.

Par exemple, le doublet <1.2.3.4.nil,nil> représente 1234, de meme que le doublet <1.2.3.4.5.6.nil,5.6.nil>.

Introduisons la relation "moitie" où moitie(u,v) signifie:

le nombre représenté par le doublet u a pour moitié le nombre représenté par le doublet v.

Pour des raisons d'homogeneité, on peut faire en sorte que u commence par le chiffre 0 ou 1. Comme on le voit, "moitie" n'est rien d'autre que l'axiomatisation de la division par 2 comme on la fait à la main.

Enfin, la relation queue(l,l') dit que la liste l' est la fin de la liste l.

Faisons tourner tout ceci. La question :

```
>moitie(<1.2.8.nil,nil>,<u,nil>);
```

donne la réponse:

```
u=6.4.nil
>
```

En fait on peut faire un peu plus en demandant:

```
>moitie(<c.2.8.nil,nil>,<u,nil>);
```

ce qui donne:

```
c=0  u=1.4.nil
c=1  u=6.4.nil
>
```

(rappelons que le dividende commence par 0 ou 1).

Allons encore plus loin en remarquant que :

"v est la moitie de u" équivaut à:
"u est le double de v".

et posons la question :

```
>moitie(<u,nil>,<1.2.3.nil>);
```

qui donne :

```
u=2.4.6.nil
```

Nous voyons donc que notre relation tourne dans les deux sens : si le premier argument est connu, on calcule sa moitié, si le deuxième argument est connu, on calcule son double.
En fait, cette relation "moitie" peut faire bien plus et résoudre par exemple le problème suivant:

Trouver deux chiffres a et b tels que 35a2b:2=1ab64.

Ceci correspond à la question :

```
>moitie(<0.3.5.a.2.b.nil,nil>,<1.a.b.6.4.nil,nil>);
```

et donne pour réponse:

```
a=7  b=8
>
```

Revenons pour finir au premier problème qui consiste à trouver d1,...,dn tels que d1d2...dn50 :2 = 5d1d2...dn.

Si on appelle p le nombre d1...dn50 et q le nombre 5d1...dn, le problème s'énonce comme suit :

- p est représenté par <d1...dn.5.0.nil,nil>
- q est représenté par <5.d1...dn.nil,nil>
  ou par <5.d1...dn.5.0.nil,5.0.nil>.
- q est la moitié de p.

C'est à dire :

- p est représenté par <x,nil> et la queue de x est 5.0.nil
- q est représenté par <5.x,5.0.nil>
- q est la moitié de p.

Tout ceci se résume en :

```
>moitie(<x,nil>,<5.x,5.0.nil>) queue(x,5.0.nil) / ;
```

qui donne pour réponse :

```
x=1.0.2.0.4.0.8.1.6.3.2.6.5.3.0.6.1.2.2.4.4.8.9.7.9.5.9.1.8.3.
   6.7.3.4.6.9.3.8.7.7.5.5.0.nil
>
```

3.4 L'ART DE CONJUGUER

Voici un programme capable de conjuguer tous les verbes du francais à tous les temps simples de l'indicatif.

Inversement, ce meme programme sait analyser un verbe, c'est à dire determiner l'infinitif, le temps et la personne à partir de la forme conjuguée d'un verbe.

Pratiquement, ce programme contient la meme somme d'informations que le Bescherelle (l'Art de Conjuguer, dictionnaire des 12000 verbes) qui nous a servi de référence pour ce qui concerne les quatre temps simples de l'indicatif des verbes francais.

La règle de base qui est utilisée est :

```
conjugaison(v,t,p,w) ->
   formes-possibles(v,k)
   temps(t,k,u)
   personne(p,u,w);
```

dans laquelle :

-v désigne un verbe à l'infinitif.
-t un temps (indicatif présent, imparfait, passé simple, futur simple).
-p une personne (singulier ou pluriel; 1ère, 2ème ou 3ème personne).
-w la forme conjuguée de v à la personne "p" du temps "t".

Pour cela, la consultation des tableaux de conjugaison type des verbes permet de determiner la liste "k" des formes conjuguées de "v". Ces tableaux sont matérialisés par une série de règles du genre :

```
formes-possibles(...,
                 ind-pres(...).
                 ind-imp (...).
                 ind-ps  (...).
                 ind-fs  (...).nil) -> ... ;
```

dans lesquelles :

- Le premier argument met en évidence la terminaison caractéristique de la famille de verbes concernée, le reste étant le radical.
- A celle-ci est associée la liste des formes conjuguées qui fait appel à des conjugaisons type que nous détaillerons plus tard.
- L'application de la règle est éventuellement restreinte par des conditions sur la forme du radical apparaissant en second membre.

Revenons à notre problème. On choisit la règle qui convient le mieux au verbe "v" à conjuguer : c'est celle qui correspond à la plus longue terminaison commune avec celle de "v" et respectant les restrictions d'application.

On récupère ainsi en "k" les formes possibles de "v" aux quatre temps simples de l'indicatif. On en extrait le temps "t"

qui nous interesse en effacant :

temps(t,k,u)

par application de la règle :

temps(t,k,u) -> dans(<t,r,x,u>,k) <t,r,x,u> ;

<t,r,x,u> est tel que :

- t figure parmi : ind-pres, ind-imp, ind-ps, ind-fs.
- r est le radical du verbe "v" qui nous interesse.
- x indique le type de conjugaison du temps "t" auquel obeit le verbe "v".
- u est un 6-uplet dont les composantes sont les formes conjuguées de "v" aux 6 personnes du temps "t".

Notre recherche se termine en extrayant du 6-uplet "u" la personne qui nous interesse par effacement de "personne(p,u,w)", ce qui fournit le résultat "w".

Précisons enfin que les règles servant à effacer les "<t,x,r,u>" se trouvent sous la rubrique "terminaisons type de l'indicatif" dans le programme qui suit et se répartissent ainsi :
- 4 terminaisons types pour le présent singulier.
- 11 pour le présent pluriel.
- 2 pour l'imparfait.
- 4 pour le passé simple.
- 1 pour le futur simple.

En pratique, nous n'utiliserons pas la directement la relation "conjugaison", mais nous passeront plutot par "jolie-conjugaison(v,t,p)" qui se chargera de transformer v en v', liste retournée des caractères composant v et, faisant appel à "conjugaison(v',t,p,w)", calculera puis imprimera "w" précédé du pronom adéquat.

Il est très interessant de souligner que bien que ce programme ait été pensé et écrit dans le sens de la synthèse, il marche aussi en analyse, c'est à dire que connaissant la valeur de "w", il calcule "v", "t" et "p" tels que la relation :

conjugaison(v,t,p,w) soit verifiée.

Ici encore, pour une presentation agréable, on passera par "jolie-analyse(w)" qui etant donné w, forme conjuguée d'un verbe, determinera w', liste retournée des caractères de w, puis fera appel à "conjugaison(v,t,p,w')" qui calculera et imprimera les solutions trouvées pour "v", "t", et "p".

Voici quelques exemples de ceci. Si on ne donne que "v" et "t" à "jolie-conjugaison", on obtient toute la conjugaison de "v" au temps "t"; si on ne donne que "v", on obtient tous les temps et toutes les personnes.

```
>jolie-conjugaison("obtenir",ind-pres,plur-1-pers);

nous obtenons

>jolie-conjugaison("conquerir",ind-imp);

   je conquerais
   tu conquerais
   il conquerait
nous conquerions
vous conqueriez
 ils conqueraient

>jolie-conjugaison("consentir");

   je consens
   tu consens
   il consent
nous consentons
vous consentez
 ils consentent

   je consentais
   tu consentais
   il consentait
nous consentions
vous consentiez
 ils consentaient

   je consentis
   tu consentis
   il consentit
nous consentimes
vous consentites
 ils consentirent

   je consentirai
   tu consentiras
   il consentira
nous consentirons
vous consentirez
 ils consentiront

>jolie-conjugaison("vetir",ind-pres,sing-1-pers);

   je vets

>jolie-conjugaison("ouvrir",ind-imp,plur-2-pers);

vous ouvriez

>jolie-conjugaison("recueillir",ind-ps,plur-1-pers);

nous recueillimes

>jolie-conjugaison("assaillir",ind-fs,sing-1-pers);

   j' assaillirai
```

```
>jolie-conjugaison("faillir",ind-pres,sing-1-pers);
  je faux
>jolie-conjugaison("redormir",ind-imp,sing-2-pers);
  tu redormais
>jolie-conjugaison("discourir",ind-fs,plur-3-pers);
 ils discourront
>jolie-conjugaison("servir",ind-pres,sing-1-pers);
  je sers
>jolie-conjugaison("asservir",ind-pres,sing-1-pers);
  j' asservis
>jolie-conjugaison("fuir",ind-imp,sing-3-pers);
  il fuyait
>jolie-conjugaison("mourir",ind-ps,sing-3-pers);
  il mourut
>jolie-conjugaison("percevoir",ind-fs,sing-2-pers);
  tu percevras
>jolie-conjugaison("revoir",ind-fs,sing-2-pers);
  tu reverras
>jolie-conjugaison("devoir",ind-fs,sing-2-pers);
  tu devras
>jolie-conjugaison("defendre",ind-ps,sing-1-pers);
  je defendis
>jolie-conjugaison("epandre",ind-imp,plur-1-pers);
nous epandions
>jolie-conjugaison("confondre",ind-fs,sing-2-pers);
  tu confondras
>jolie-conjugaison("perdre",ind-pres,sing-1-pers);
  je perds
>jolie-conjugaison("mordre",ind-ps,plur-2-pers);
vous mordites
```

```
>jolie-conjugaison("apprendre",ind-fs,sing-3-pers);
  il apprendra
>jolie-conjugaison("abattre",ind-imp,plur-3-pers);
 ils abattaient
>jolie-conjugaison("compromettre",ind-ps,sing-1-pers);
  je compromis
>jolie-conjugaison("restreindre",ind-imp,plur-1-pers);
nous restreignions
>jolie-conjugaison("disjoindre",ind-pres,plur-1-pers);
nous disjoignons
>jolie-conjugaison("plaindre",ind-imp,sing-2-pers);
  tu plaignais
>jolie-conjugaison("traire",ind-fs,plur-1-pers);
nous trairons
>jolie-conjugaison("faire",ind-pres,plur-2-pers);
vous faites
>jolie-conjugaison("plaire",ind-pres,plur-2-pers);
vous plaisez
>jolie-conjugaison("naitre",ind-ps,sing-1-pers);
  je naquis
>jolie-conjugaison("conclure",ind-pres,sing-3-pers);
  il conclut
>jolie-conjugaison("absoudre",ind-imp,plur-3-pers);
 ils absolvaient
>jolie-conjugaison("poursuivre",ind-ps,plur-3-pers);
 ils poursuivirent
>jolie-conjugaison("survivre",ind-fs,sing-3-pers);
  il survivra
>jolie-conjugaison("elire",ind-pres,plur-3-pers);
 ils elisent
```

```
>jolie-conjugaison("medire",ind-ps,plur-1-pers);

nous medimes

>jolie-conjugaison("moudre",ind-fs,plur-1-pers);

nous moudrons

>jolie-conjugaison("moudre",ind-ps,plur-1-pers);

nous moulumes

>jolie-conjugaison("sourire",ind-imp,plur-1-pers);

nous souriions

>jolie-conjugaison("inscrire",ind-pres,sing-1-pers);

  j' inscris

>jolie-conjugaison("confire",ind-ps,plur-2-pers);

>jolie-conjugaison("conduire",ind-ps,plur-1-pers);

nous conduisimes

>jolie-conjugaison("finir");

   je finis
   tu finis
   il finit
nous finissons
vous finissez
 ils finissent

   je finissais
   tu finissais
   il finissait
nous finissions
vous finissiez
 ils finissaient

   je finis
   tu finis
   il finit
nous finimes
vous finites
 ils finirent

   je finirai
   tu finiras
   il finira
nous finirons
vous finirez
 ils finiront

>jolie-conjugaison("aimer",ind-pres,plur-1-pers);

nous aimons
```

```
>jolie-conjugaison("manger",ind-imp);

  je mangeais
  tu mangeais
  il mangeait
nous mangions
vous mangiez
 ils mangeaient

>jolie-conjugaison("jeter",ind-pres);

  je jette
  tu jettes
  il jette
nous jetons
vous jetez
 ils jettent

>jolie-conjugaison("acheter",ind-pres);

  j' achete
  tu achetes
  il achete
nous achetons
vous achetez
 ils achetent

>jolie-conjugaison("apprecier",ind-ps);

  j' appreciai
  tu apprecias
  il apprecia
nous appreciames
vous appreciates
 ils apprecierent

>jolie-conjugaison("broyer",ind-pres);

  je broie
  tu broies
  il broie
nous broyons
vous broyez
 ils broient

>jolie-conjugaison("broyer",ind-fs,sing-1-pers);

  je broierai

>jolie-conjugaison("envoyer",ind-fs,sing-1-pers);

  j' enverrai
```

Voici maintenant quelques exemples où la meme relation "conjugaison est utilisée à l'envers, c'est à dire dans le sens de l'analyse.

```
>jolie-analyse("tenais");
tenir ind-imp sing-1-pers
tenir ind-imp sing-2-pers
>jolie-analyse("finis");
finir ind-pres sing-1-pers
finir ind-pres sing-2-pers
finir ind-ps sing-1-pers
finir ind-ps sing-2-pers
>jolie-analyse("crumes");
croitre ind-ps plur-1-pers
croire ind-ps plur-1-pers
>jolie-analyse("mirent");
mettre ind-ps plur-3-pers
mirer ind-pres plur-3-pers
>
```

```
conjugaison(v,t,p,w) -> formes-possibles(v,e,k) temps(t,k,u)
                        personne(p,u,w) ;

temps(t,k,u) -> dans(<t,r,x,u>,k) <t,r,x,u> ;

personne(sing-1-pers,<j,t,i,n,v,i'>,j) -> ;
personne(sing-2-pers,<j,t,i,n,v,i'>,t) -> ;
personne(sing-3-pers,<j,t,i,n,v,i'>,i) -> ;
personne(plur-1-pers,<j,t,i,n,v,i'>,n) -> ;
personne(plur-2-pers,<j,t,i,n,v,i'>,v) -> ;
personne(plur-3-pers,<j,t,i,n,v,i'>,i') -> ;

"pour faire joli"

jolie-conjugaison(v,t,p) -> envers(v,v') ligne
                            conjugaison(v',t,p,w')
                            jolie-sortie(p,w') ligne ;

jolie-conjugaison(v,t) -> jolie-conjugaison(v,t,p) ;

jolie-conjugaison(v) -> jolie-conjugaison(v,t,p) ;

jolie-analyse(v) -> envers(v,v') ligne
                    conjugaison(w,t,p,v')
                    jolie-sortie'(w)
                    exm(" ") ex(t) exm(" ") ex(p) ligne;

jolie-sortie(p,u.nil) -> accord(p,u,p') exm(p') exm(" ") exm(u) ;
jolie-sortie(p,v.w) -> dif(w,nil) jolie-sortie(p,w) exm(v) ;

jolie-sortie'(nil) -> ;
jolie-sortie'(a.b) -> jolie-sortie'(b) exm(a) ;

accord(sing-1-pers,u,"  j'") -> dans(u,"a"."e"."i"."o"."u".nil) / ;
accord(sing-1-pers,u,"  je") -> ;
accord(sing-2-pers,u,"  tu") -> ;
accord(sing-3-pers,u,"  il") -> ;
accord(plur-1-pers,u,"nous") -> ;
accord(plur-2-pers,u,"vous") -> ;
accord(plur-3-pers,u," ils") -> ;

"divers"

dans(a,a.x) ->;
dans(a,b.x) -> dans(a,x);

envers(v,w) -> arg(0,v,1) envers'(1,v,w) ;

envers'(0,v,nil) -> ;
envers'(1,v,d.w) -> dif(1,0) arg(1,v,d) val(sub(1,1),1')
                    envers'(1',v,w) ;

ne-commence-pas(a.1,nil) -> ;
ne-commence-pas(a.1,b.1') -> dif(a,b) ;
ne-commence-pas(a.1,a.1') -> ne-commence-pas(1,1') ;

ne-commencent-pas(nil,q) -> ;
ne-commencent-pas(a.b,q) -> ne-commence-pas(a,q)
                            ne-commencent-pas(b,q);
```

```
"les formes possibles des verbes"

"on trouvera d'abord les auxilliaires etre et avoir"
"puis les verbes des troisieme, deuxieme et premier groupe"

formes-possibles("e"."r"."t"."e".nil,etre,
                        ind-pres(nil,type-etre,i-pres).
                        ind-imp ("t"."e"."nil",type-1,i-imp).
                        ind-ps  ("f".nil,type-3,i-ps).
                        ind-fs  ("e"."s".nil,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."v"."a".nil,avoir,
                        ind-pres(nil,type-avoir,i-pres).
                        ind-imp ("v"."a"."nil",type-1,i-imp).
                        ind-ps  ("e".nil,type-3,i-ps).
                        ind-fs  ("u"."a".nil,type-1,i-fs).nil) -> ;

formes-possibles("r"."e"."l"."l"."a".nil,aller,
                    ind-pres(nil,type-aller,i-pres).
                    ind-imp ("l"."l"."a".nil,type-1,i-imp).
                    ind-ps  ("l"."l"."a".nil,type-4,i-ps).
                    ind-fs  ("i".nil,type-1,i-fs).nil)      -> ;

formes-possibles("r"."i"."n"."e".x.q,venir.tenir,
                     ind-pres(<"n"."e"."i".x.q,x.q>,
                                <type-s-1,type-p-8>,i-pres).
                     ind-imp ("n"."e".x.q,type-1,i-imp).
                     ind-ps  ("n"."i".x.q,type-1,i-ps).
                     ind-fs  ("d"."n"."e"."i".x.q,type-1,i-fs).nil)

       -> dans(x,"t"."v".nil);

formes-possibles("r"."i"."r"."e"."u"."q".q,acquerir,
                     ind-pres(<"r"."e"."i"."u"."q".q,"u"."q".q>,
                                <type-s-1,type-p-9>,i-pres).
                     ind-imp ("r"."e"."u"."q".q,type-1,i-imp).
                     ind-ps  ("u"."q".q,type-2,i-ps).
                     ind-fs  ("r"."e"."u"."q".q,type-1,i-fs).nil) ->;

formes-possibles("r"."i"."t".x.q,sentir.sortir,
                    ind-pres(<x.q,"t".x.q>,<type-s-1,type-p-1>,i-pres).
                    ind-imp ("t".x.q,type-1,i-imp).
                    ind-ps  ("t".x.q,type-2,i-ps).
                    ind-fs  ("i"."t".x.q,type-1,i-fs).nil)

       -> dans(x,"n"."r".nil);

formes-possibles("r"."i"."r".x.q,ouvrir.souffrir,
                       ind-pres(<"e"."r".x.q,"r".x.q>,
                                  <type-s-4,type-p-1>,i-pres).
                       ind-imp ("r".x.q,type-1,i-imp).
                       ind-ps  ("r".x.q,type-2,i-ps).
                       ind-fs  ("i"."r".x.q,type-1,i-fs).nil)

       -> dans(x,"f"."v".nil);
```

```
formes-possibles("r"."i"."l"."l"."i"."e"."u"."c".q,cueillir,
            ind-pres(<"e"."l"."l"."i"."e"."u"."c".q,
                      "l"."l"."i"."e"."u"."c".q>,
                     <type-s-4,type-p-1>,i-pres).
            ind-imp ("l"."l"."i"."e"."u"."c".q,type-1,i-imp).
            ind-ps  ("l"."l"."i"."e"."u"."c".q,type-2,i-ps).
            ind-fs  ("e"."l"."l"."i"."e"."u"."c".q,type-1,i-fs).nil) ->

formes-possibles("r"."i"."l"."l"."i"."a"."s".q,assaillir,
            ind-pres(<"e"."l"."l"."i"."a"."s".q,"l"."l"."i"."a"."s".q>,
                     <type-s-4,type-p-1>,i-pres).
            ind-imp ("l"."l"."i"."a"."s".q,type-1,i-imp).
            ind-ps  ("l"."l"."i"."a"."s".q,type-2,i-ps).
            ind-fs  ("i"."l"."l"."i"."a"."s".q,type-1,i-fs).nil) ->;

formes-possibles("r"."i"."l"."l"."i"."u"."o"."b".q,bouillir,
                ind-pres(<"u"."o"."b".q,"l"."l"."i"."u"."o"."b".q>,
                         <type-s-1,type-p-1>,i-pres).
                ind-imp ("l"."l"."i"."u"."o"."b".q,type-1,i-imp).
                ind-ps  ("l"."l"."i"."u"."o"."b".q,type-2,i-ps).
                ind-fs  ("i"."l"."l"."i"."u"."o"."b".q,type-1,i-fs).nil

             ->;

formes-possibles("r"."i"."m"."r"."o"."d".q,dormir,
                  ind-pres(<"r"."o"."d".q,"m"."r"."o"."d".q>,
                           <type-s-1,type-p-1>,i-pres).
                  ind-imp ("m"."r"."o"."d".q,type-1,i-imp).
                  ind-ps  ("m"."r"."o"."d".q,type-2,i-ps).
                  ind-fs  ("m"."r"."o"."d".q,type-1,i-fs).nil) ->;

formes-possibles("r"."i"."r"."u"."o"."c".q,courir,
                  ind-pres(<"r"."u"."o"."c".q,"r"."u"."o"."c".q>,
                           <type-s-1,type-p-1>,i-pres).
                  ind-imp ("r"."u"."o"."c".q,type-1,i-imp).
                  ind-ps  ("r"."u"."o"."c".q,type-3,i-ps).
                  ind-fs  ("r"."u"."o"."c".q,type-1,i-fs).nil) ->;

formes-possibles("r"."i"."v"."r"."e"."s".q,servir,
                  ind-pres(<"r"."e"."s".q,"v"."r"."e"."s".q>,
                           <type-s-1,type-p-1>,i-pres).
                  ind-imp ("v"."r"."e"."s".q,type-1,i-imp).
                  ind-ps  ("v"."r"."e"."s".q,type-2,i-ps).
                  ind-fs  ("i"."v"."r"."e"."s".q,type-1,i-fs).nil)

      -> dif(q,"s"."a".nil);

formes-possibles("r"."i"."u"."f".q,fuir,
                  ind-pres(<"i"."u"."f".q,"u"."f".q>,
                           <type-s-1,type-p-2>,i-pres).
                  ind-imp ("y"."u"."f".q,type-1,i-imp).
                  ind-ps  ("u"."f".q,type-2,i-ps).
                  ind-fs  ("i"."u"."f",type-1,i-fs).nil) ->;

formes-possibles("r"."i"."o"."v"."e"."c".q,recevoir,
                ind-pres(<"i"."o"."c".q,"e"."c".q>,
                         <type-s-1,type-p-6>,i-pres).
                ind-imp ("e"."c".q,type-1,i-imp).
                ind-ps  ("c".q,type-3,i-ps).
                ind-fs  ("v"."e"."c".q,type-1,i-fs).nil) ->;
```

```
formes-possibles("r"."i"."o"."v".q,voir,
                 ind-pres(<"i"."o"."v".q,"o"."v".q>,
                          <type-s-1,type-p-2>,i-pres).
                 ind-imp ("y"."o"."v".q,type-1,i-imp).
                 ind-ps  ("v".q,type-2,i-ps).
                 ind-fs  ("r"."e"."v".q,type-1,i-fs).nil)

      -> dans(q,("e"."r"."t"."n"."e".nil).("e"."r"."p".nil).
                ("e"."r".nil).nil.nil) ;

formes-possibles("e"."r"."d"."n"."e"."r"."p".q,prendre,
                 ind-pres(<"d"."n"."e"."r"."p".q,
                           "n"."n"."e"."r"."p".q>,
                          <type-s-2,type-p-1>,i-pres).
                 ind-imp ("n"."e"."r"."p".q,type-1,i-imp).
                 ind-ps  ("r"."p".q,type-2,i-ps).
                 ind-fs  ("d"."n"."e"."r"."p".q,type-1,i-fs).nil) ->;

formes-possibles("e"."r"."d"."n"."i"."e".q,peindre,
                 ind-pres(<"n"."i"."e".q,"n"."g"."i"."e".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("n"."g"."i"."e".q,type-1,i-imp).
                 ind-ps  ("n"."g"."i"."e".q,type-2,i-ps).
                 ind-fs  ("d"."n"."i"."e".q,type-1,i-fs).nil) ->;

formes-possibles("e"."r"."d".q,rendre,
                 ind-pres(<"d".q,"d".q>,<type-s-2,type-p-1>,i-pres).
                 ind-imp ("d".q,type-1,i-imp).
                 ind-ps  ("d".q,type-2,i-ps).
                 ind-fs  ("d".q,type-1,i-fs).nil)

      -> ne-commencent-pas(("n"."i"."a".nil).("n"."i"."o".nil).
                           ("n"."i"."e".nil).("n"."e"."r"."p".nil).
                           ("u"."o".nil).nil,q) ;

formes-possibles("e"."r"."t"."t"."a"."b".q,battre,
                 ind-pres(<"t"."a"."b".q,"t"."t"."a"."b".q>,
                          <type-s-2,type-p-1>,i-pres).
                 ind-imp ("t"."t"."a"."b".q,type-1,i-imp).
                 ind-ps  ("t"."t"."a"."b".q,type-2,i-ps).
                 ind-fs  ("t"."t"."a"."b".q,type-1,i-fs).nil) ->;

formes-possibles("e"."r"."t"."t"."e"."m".q,mettre,
                 ind-pres(<"t"."e"."m".q,"t"."t"."e"."m".q>,
                          <type-s-2,type-p-1>,i-pres).
                 ind-imp ("t"."t"."e"."m".q,type-1,i-imp).
                 ind-ps  ("m".q,type-2,i-ps).
                 ind-fs  ("t"."t"."e"."m".q,type-1,i-fs).nil) ->;
```

```
formes-possibles("e"."r"."d"."n"."i".x.q,joindre.craindre,
                 ind-pres(<"n"."i".x.q,"n"."g"."i".x.q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("n"."g"."i".x.q,type-1,i-imp).
                 ind-ps  ("n"."g"."i".x.q,type-2,i-ps).
                 ind-fs  ("d"."n"."i".x.q,type-1,i-fs).nil) ->

       dans(x,"o"."a".nil) ;

formes-possibles("e"."r"."i"."a"."r"."t".q,traire,
                 ind-pres(<"i"."a"."r"."t".q,"a"."r"."t".q>,
                          <type-s-1,type-p-2>,i-pres).
                 ind-imp ("y"."a"."r"."t".q,type-1,i-imp).
                 ind-fs  ("i"."a"."r"."t".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i"."a"."f".q,faire,
                 ind-pres(<"i"."a"."f".q,"f".q>,
                          <type-s-1,type-p-5>,i-pres).
                 ind-imp ("s"."i"."a"."f".q,type-1,i-imp).
                 ind-ps  ("f".q,type-2,i-ps).
                 ind-fs  ("e"."f".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i"."a".q,plaire,
                 ind-pres(<"i"."a".q,"s"."i"."a".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("s"."i"."a".q,type-1,i-imp).
                 ind-ps  (q,type-3,i-ps).
                 ind-fs  ("i"."a".q,type-1,i-fs).nil) ->

       dans(q,("l"."p".q').("t".nil).nil) ;

formes-possibles("e"."r"."t"."i"."a"."p".q,paitre,
                 ind-pres(<"i"."a"."p".q,"s"."s"."i"."a"."p".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("s"."s"."i"."n"."a".q,type-1,i-imp).
                 ind-fs  ("t"."i"."a"."n".q,type-1,i-fs).nil) ->

    dans(q,nil.("e"."r".nil).nil) ;

formes-possibles("e"."r"."t"."i"."a"."n".q,naitre,
                 ind-pres(<"i"."a"."n".q,"s"."s"."i"."a"."n".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("s"."s"."i"."n"."a".q,type-1,i-imp).
                 ind-ps  ("u"."q"."a"."n".q,type-2,i-ps).
                 ind-fs  ("t"."i"."a"."n".q,type-1,i-fs).nil) ->

    dans(q,nil.("e"."r".nil).nil) ;

formes-possibles("e"."r"."t"."i"."a".q,connaitre,
                 ind-pres(<"i"."a".q,"s"."s"."i"."a".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("s"."s"."i"."a".q,type-1,i-imp).
                 ind-ps  (q,type-3,i-ps).
                 ind-fs  ("t"."i"."a".q,type-1,i-fs).nil)

       -> dif(q,"n".nil) dif(q,"n"."e"."r".nil)
          dif(q,"p".nil) dif(q,"p"."e"."r".nil) ;
```

```
formes-possibles("e"."r"."t"."i"."o".q,croitre,
                 ind-pres(<"i"."o".q,"s"."s"."i"."o".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("s"."s"."i"."o".q,type-1,i-imp).
                 ind-ps  (q,type-3,i-ps).
                 ind-fs  ("t"."i"."o".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i"."o"."r"."c".q,croire,
                 ind-pres(<"i"."o"."r"."c".q,"o"."r"."c".q>,
                          <type-s-1,type-p-2>,i-pres).
                 ind-imp ("s"."s"."i"."o"."r"."c".q,type-1,i-imp).
                 ind-ps  ("r"."c".q,type-3,i-ps).
                 ind-fs  ("i"."o"."r"."c".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i"."o"."b".q,boire,
                 ind-pres(<"i"."o"."b".q,"u"."b".q>,
                          <type-s-1,type-p-6>,i-pres).
                 ind-imp ("v"."u"."b".q,type-1,i-imp).
                 ind-ps  ("b".q,type-3,i-ps).
                 ind-fs  ("i"."o"."b".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."o"."l"."c".q,clore,
        ind-pres(<"o"."l"."c".q,"s"."o"."l"."c".q>,
                  <type-s-1,type-p-1>,i-pres).
        ind-fs  ("o"."l"."c".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."u"."l"."c".q,conclure,
                 ind-pres(<"u"."l"."c".q,"u"."l"."c".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("u"."l"."c".q,type-1,i-imp).
                 ind-ps  ("l"."c".q,type-3,i-ps).
                 ind-fs  ("u"."l"."c".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."d"."u"."o"."s".q,absoudre,
                 ind-pres(<"u"."o"."s".q,"l"."o"."s".q>,
                          <type-s-6,type-p-1>,i-pres).
                 ind-imp ("v"."l"."o"."s".q,type-1,i-imp).
                 ind-fs  ("d"."u"."o"."s".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."d"."u"."o"."c".q,coudre,
                 ind-pres(<"d"."u"."o"."c".q,"s"."u"."o"."c".q>,
                          <type-s-2,type-p-1>,i-pres).
                 ind-imp ("s"."u"."o"."c".q,type-1,i-imp).
                 ind-ps  ("s"."u"."o"."c".q,type-2,i-ps).
                 ind-fs  ("d"."u"."o"."c".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."d"."u"."o"."m".q,moudre,
                 ind-pres(<"d"."u"."o"."m".q,"l"."u"."o"."m".q>,
                          <type-s-2,type-p-1>,i-pres).
                 ind-imp ("l"."u"."o"."m".q,type-1,i-imp).
                 ind-ps  ("l"."u"."o"."m".q,type-3,i-ps).
                 ind-fs  ("d"."u"."o"."m".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."v"."i"."u"."s".q,suivre,
                 ind-pres(<"i"."u"."s".q,"v"."i"."u"."s".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("v"."i"."u"."s".q,type-1,i-imp).
                 ind-ps  ("v"."i"."u"."s".q,type-2,i-ps).
                 ind-fs  ("v"."i"."u"."s".q,type-1,i-fs).nil) -> ;
```

```
formes-possibles("e"."r"."v"."i"."v".q,vivre,
                 ind-pres(<"i"."v".q,"v"."i"."v".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("v"."i"."v".q,type-1,i-imp).
                 ind-ps   ("c"."e"."v".q,type-3,i-ps).
                 ind-fs   ("v"."i"."v".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i"."l".q,lire,
                 ind-pres(<"i"."l".q,"s"."i"."l".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("s"."i"."l".q,type-1,i-imp).
                 ind-ps   ("l".q,type-3,i-ps).
                 ind-fs   ("i"."l".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i"."d".q,dire,
                 ind-pres(<"i"."d".q,"i"."d".q>,
                          <type-s-1,type-p-7>,i-pres).
                 ind-imp  ("s"."i"."d".q,type-1,i-imp).
                 ind-ps   ("d".q,type-2,i-ps).
                 ind-fs   ("i"."d".q,type-1,i-fs).nil)

        -> dif(q,"u"."a"."m".nil) ;

formes-possibles("e"."r"."i"."r".q,rire,
                 ind-pres(<"i"."r".q,"i"."r".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("i"."r".q,type-1,i-imp).
                 ind-ps   ("r".q,type-2,i-ps).
                 ind-fs   ("i"."r".q,type-1,i-fs).nil)

          -> ne-commence-pas("c".nil,q) ;

formes-possibles("e"."r"."i"."r"."c".q,ecrire,
                 ind-pres(<"i"."r"."c".q,"v"."i"."r"."c".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("v"."i"."r"."c".q,type-1,i-imp).
                 ind-ps   ("v"."i"."r"."c".q,type-2,i-ps).
                 ind-fs   ("i"."r"."c".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i"."u".q,cuire,
                 ind-pres(<"i"."u".q,"s"."i"."u".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("s"."i"."u".q,type-1,i-imp).
                 ind-ps   ("s"."i"."u".q,type-2,i-ps).
                 ind-fs   ("i"."u".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i".q,confire.circoncire.frire.suffire,
                 ind-pres(<"i".q,"s"."i".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("s"."i".q,type-1,i-imp).
                 ind-ps   (q,type-2,i-ps).
                 ind-fs   ("i".q,type-1,i-fs).nil)

        -> ne-commencent-pas(("a".nil).("o".nil).("l".nil).
                             ("d".nil).("r".nil),q) ;
```

```
formes-possibles("r"."i"."t"."e"."v".q,vetir,
                 ind-pres(<"t"."e"."v".q,"t"."e"."v".q>,
                          <type-s-2,type-p-1>,i-pres).
                 ind-imp ("t"."e"."v".q,type-1,i-imp).
                 ind-ps  ("t"."e"."v".q,type-2,i-ps).
                 ind-fs    ("i"."t"."e"."v".q,type-1,i-fs).nil)   ->

       ne-commencent-pas(("u".nil).("c"."r".nil).("r".nil).
                          ("d".nil).("l".nil).nil,q) ;

formes-possibles("r"."i"."l"."l"."i"."a"."f".q,faillir,
                 ind-pres(<"u"."a"."f".q,"l"."l"."i"."a"."f".q>,
                          <type-s-3,type-p-1>,i-pres).
                 ind-imp ("l"."l"."i"."a"."f".q,type-1,i-imp).
                 ind-ps  ("l"."l"."i"."a"."f".q,type-2,i-ps).
                 ind-fs                    ("i"."l"."l"."i"."a"."f".q,
                          type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."r"."u"."o"."m".nil,mourir,
                 ind-pres(<"r"."u"."e"."m".nil,"r"."u"."o"."m".nil>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("r"."u"."o"."m".nil,type-1,i-imp).
                 ind-ps  ("r"."u"."o"."m".nil,type-3,i-ps).
                 ind-fs  ("r"."u"."o"."m".nil,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."v"."r"."u"."o"."p".q,pourvoir,
                 ind-pres(<"i"."o"."v"."r"."u"."o"."p".q,
                           "o"."v"."r"."u"."o"."p".q>,
                          <type-s-1,type-p-2>,i-pres).
                 ind-imp ("y"."o"."v"."r"."u"."o"."p".q,type-1,i-imp).
                 ind-ps  ("v"."r"."u"."o"."p".q,type-3,i-ps).
                 ind-fs          ("r"."i"."o"."v"."r"."u"."o"."p".q,
                          type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."v"."a"."s".q,savoir,
                 ind-pres(<"a"."s".q,"v"."a"."s".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("v"."a"."s".q,type-1,i-imp).
                 ind-ps  ("s".q,type-3,i-ps).
                 ind-fs  ("u"."a"."s".q,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."v"."e"."d".q,devoir,
                 ind-pres(<"i"."o"."d".q,"e"."d".q>,
                          <type-s-1,type-p-6>,i-pres).
                 ind-imp ("v"."e"."d".q,type-1,i-imp).
                 ind-ps  ("d".q,type-3,i-ps).
                 ind-fs  ("v"."e"."d".q,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."v"."u"."o"."m".q,mouvoir,
                 ind-pres(<"u"."e"."m".q,"v"."u"."e"."m".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("v"."u"."o"."m".q,type-1,i-imp).
                 ind-ps  ("m".q,type-3,i-ps).
                 ind-fs  ("v"."u"."o"."m".q,type-1,i-fs).nil) -> ;
```

```
formes-possibles("r"."i"."o"."l"."a"."v".q,valoir,
                 ind-pres(<"u"."a"."v".q,"l"."a"."v".q>,
                          <type-s-3,type-p-1>,i-pres).
                 ind-imp ("l"."a"."v".q,type-1,i-imp).
                 ind-ps  ("l"."a"."v".q,type-3,i-ps).
                 ind-fs  ("d"."u"."a"."v".q,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."v"."u"."o"."p".nil,pouvoir,
                 ind-pres(<"u"."e"."p".nil,"v"."u"."o"."p".nil>,
                          <type-s-3,type-p-1>,i-pres).
                 ind-imp ("v"."u"."o"."p".nil,type-1,i-imp).
                 ind-ps  ("p".nil,type-3,i-ps).
                 ind-fs  ("r"."u"."o"."p".nil,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."l"."u"."o"."v".nil,vouloir,
                 ind-pres(<"u"."e"."v".nil,"l"."u"."o"."v".nil>,
                          <type-s-3,type-p-1>,i-pres).
                 ind-imp ("l"."u"."o"."v".nil,type-1,i-imp).
                 ind-ps  ("l"."u"."o"."v".nil,type-3,i-ps).
                 ind-fs  ("d"."u"."o"."v".nil,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."e"."s"."s"."a".q,asseoir,
                 ind-pres(<"i"."o"."s"."s"."a".q,"y"."o"."s"."s"."a".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("y"."o"."s"."s"."a".q,type-1,i-imp).
                 ind-ps  ("s"."s"."a".q,type-2,i-ps).
                 ind-fs  ("i"."o"."s"."s"."a".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."p".q,rompre,
                 ind-pres(<"p".q,"p".q>,<type-s-1,type-p-1>,i-pres).
                 ind-imp ("p.q,type-1,i-imp).
                 ind-ps  ("p".q,type-2,i-ps).
                 ind-fs  ("p".q,type-1,i-fs).nil) -> ;


formes-possibles("e"."r"."c"."n"."i"."a"."v".q,vaincre,,
          ind-pres(<"c"."n"."i"."a"."v".q,"u"."q"."n"."i"."a"."v".q>,
                   <type-s-2,type-p-1>,i-pres).
          ind-imp ("u"."q"."n"."i"."a"."v".q,type-1,i-imp).
          ind-ps  ("u"."q"."n"."i"."a"."v".q,type-2,i-ps).
          ind-fs  ("c"."n"."i"."a"."v".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."d"."n"."i".x.q,joindre.craindre,
                 ind-pres(<"n"."i".x.q,"n"."g"."i".x.q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("n"."g"."i".x.q,type-1,i-imp).
                 ind-ps  ("n"."g"."i".x.q,type-2,i-ps).
                 ind-fs  ("d"."n"."i".x.q,type-1,i-fs).nil) ->

       dans(x,"o"."a".nil) ;

formes-possibles("e"."r"."i"."a"."r"."t".q,traire,
                 ind-pres(<"i"."a"."r"."t".q,"a"."r"."t".q>,
                          <type-s-1,type-p-2>,i-pres).
                 ind-imp ("y"."a"."r"."t".q,type-1,i-imp).
                 ind-fs  ("i"."a"."r"."t".q,type-1,i-fs).nil) -> ;
```

```
formes-possibles("e"."r"."i"."a"."f".q,faire,
                 ind-pres(<"i"."a"."f".q,"f".q>,
                          <type-s-1,type-p-5>,i-pres).
                 ind-imp ("s"."i"."a"."f".q,type-1,i-imp).
                 ind-ps  ("f".q,type-2,i-ps).
                 ind-fs  ("e"."f".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i"."a".q,plaire,
                 ind-pres(<"i"."a".q,"s"."i"."a".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("s"."i"."a".q,type-1,i-imp).
                 ind-ps  (q,type-3,i-ps).
                 ind-fs  ("i"."a".q,type-1,i-fs).nil) ->

       dans(q,("l"."p".q').("t".nil).nil) ;

formes-possibles("e"."r"."t"."i"."a"."p".q,paitre,
                 ind-pres(<"i"."a"."p".q,"s"."s"."i"."a"."p".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("s"."s"."i"."n"."a".q,type-1,i-imp).
                 ind-fs  ("t"."i"."a"."n".q,type-1,i-fs).nil) ->

    dans(q,nil.("e"."r".nil).nil) ;

formes-possibles("e"."r"."t"."i"."a"."n".q,naitre,
                 ind-pres(<"i"."a"."n".q,"s"."s"."i"."a"."n".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("s"."s"."i"."n"."a".q,type-1,i-imp).
                 ind-ps  ("u"."q"."a"."n".q,type-2,i-ps).
                 ind-fs  ("t"."i"."a"."n".q,type-1,i-fs).nil) ->

    dans(q,nil.("e"."r".nil).nil) ;

formes-possibles("e"."r"."t"."i"."a".q,connaitre,
                 ind-pres(<"i"."a".q,"s"."s"."i"."a".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("s"."s"."i"."a".q,type-1,i-imp).
                 ind-ps  (q,type-3,i-ps).
                 ind-fs  ("t"."i"."a".q,type-1,i-fs).nil)

       -> dif(q,"n".nil) dif(q,"n"."e"."r".nil)
          dif(q,"p".nil) dif(q,"p"."e"."r".nil) ;

formes-possibles("e"."r"."t"."i"."o".q,croitre,
                 ind-pres(<"i"."o".q,"s"."s"."i"."o".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("s"."s"."i"."o".q,type-1,i-imp).
                 ind-ps  (q,type-3,i-ps).
                 ind-fs  ("t"."i"."o".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i"."o"."r"."c".q,croire,
                 ind-pres(<"i"."o"."r"."c".q,"o"."r"."c".q>,
                          <type-s-1,type-p-2>,i-pres).
                 ind-imp ("s"."s"."i"."o"."r"."c".q,type-1,i-imp).
                 ind-ps  ("r"."c".q,type-3,i-ps).
                 ind-fs  ("i"."o"."r"."c".q,type-1,i-fs).nil) -> ;
```

```
formes-possibles("e"."r"."i"."o"."b".q,boire,
                 ind-pres(<"i"."o"."b".q,"u"."b".q>,
                          <type-s-1,type-p-6>,i-pres).
                 ind-imp ("v"."u"."b".q,type-1,i-imp).
                 ind-ps  ("b".q,type-3,i-ps).
                 ind-fs  ("i"."o"."b".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."o"."l"."c".q,clore,
                 ind-pres(<"o"."l"."c".q,"s"."o"."l"."c".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-fs  ("o"."l"."c".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."u"."l"."c".q,conclure,
                 ind-pres(<"u"."l"."c".q,"u"."l"."c".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("u"."l"."c".q,type-1,i-imp).
                 ind-ps  ("l"."c".q,type-3,i-ps).
                 ind-fs  ("u"."l"."c".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."d"."u"."o"."s".q,absoudre,
                 ind-pres(<"u"."o"."s".q,"l"."o"."s".q>,
                          <type-s-6,type-p-1>,i-pres).
                 ind-imp ("v"."l"."o"."s".q,type-1,i-imp).
                 ind-fs  ("d"."u"."o"."s".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."d"."u"."o"."c".q,coudre,
                 ind-pres(<"d"."u"."o"."c".q,"s"."u"."o"."c".q>,
                          <type-s-2,type-p-1>,i-pres).
                 ind-imp ("s"."u"."o"."c".q,type-1,i-imp).
                 ind-ps  ("s"."u"."o"."c".q,type-2,i-ps).
                 ind-fs  ("d"."u"."o"."c".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."d"."u"."o"."m".q,moudre,
                 ind-pres(<"d"."u"."o"."m".q,"l"."u"."o"."m".q>,
                          <type-s-2,type-p-1>,i-pres).
                 ind-imp ("l"."u"."o"."m".q,type-1,i-imp).
                 ind-ps  ("l"."u"."o"."m".q,type-3,i-ps).
                 ind-fs  ("d"."u"."o"."m".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."v"."i"."u"."s".q,suivre,
                 ind-pres(<"i"."u"."s".q,"v"."i"."u"."s".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("v"."i"."u"."s".q,type-1,i-imp).
                 ind-ps  ("v"."i"."u"."s".q,type-2,i-ps).
                 ind-fs  ("v"."i"."u"."s".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."v"."i"."v".q,vivre,
                 ind-pres(<"i"."v".q,"v"."i"."v".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("v"."i"."v".q,type-1,i-imp).
                 ind-ps  ("c"."e"."v".q,type-3,i-ps).
                 ind-fs  ("v"."i"."v".q,type-1,i-fs).nil) -> ;
```

```
formes-possibles("e"."r"."i"."l".q,lire,
                 ind-pres(<"i"."l".q,"s"."i"."l".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("s"."i"."l".q,type-1,i-imp).
                 ind-ps   ("l".q,type-3,i-ps).
                 ind-fs   ("i"."l".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i"."d".q,dire,
                 ind-pres(<"i"."d".q,"i"."d".q>,
                          <type-s-1,type-p-7>,i-pres).
                 ind-imp  ("s"."i"."d".q,type-1,i-imp).
                 ind-ps   ("d".q,type-2,i-ps).
                 ind-fs   ("i"."d".q,type-1,i-fs).nil)

        -> dif(q,"u"."a"."m".nil) ;

formes-possibles("e"."r"."i"."r".q,rire,
                 ind-pres(<"i"."r".q,"i"."r".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("i"."r".q,type-1,i-imp).
                 ind-ps   ("r".q,type-2,i-ps).
                 ind-fs   ("i"."r".q,type-1,i-fs).nil)

          -> ne-commence-pas("c".nil,q) ;

formes-possibles("e"."r"."i"."r"."c".q,ecrire,
                 ind-pres(<"i"."r"."c".q,"v"."i"."r"."c".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("v"."i"."r"."c".q,type-1,i-imp).
                 ind-ps   ("v"."i"."r"."c".q,type-2,i-ps).
                 ind-fs   ("i"."r"."c".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i"."u".q,cuire,
                 ind-pres(<"i"."u".q,"s"."i"."u".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("s"."i"."u".q,type-1,i-imp).
                 ind-ps   ("s"."i"."u".q,type-2,i-ps).
                 ind-fs   ("i"."u".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."i".q,confire.circoncire.frire.suffire,
                 ind-pres(<"i".q,"s"."i".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("s"."i".q,type-1,i-imp).
                 ind-ps   (q,type-2,i-ps).
                 ind-fs   ("i".q,type-1,i-fs).nil)

        -> ne-commencent-pas(("a".nil).("o".nil).("l".nil).
                             ("d".nil).("r".nil),q) ;
```

```
formes-possibles("r"."i"."t"."e"."v".q,vetir,
                 ind-pres(<"t"."e"."v".q,"t"."e"."v".q>,
                          <type-s-2,type-p-1>,i-pres).
                 ind-imp  ("t"."e"."v".q,type-1,i-imp).
                 ind-ps   ("t"."e"."v".q,type-2,i-ps).
                 ind-fs   ("i"."t"."e"."v".q,type-1,i-fs).nil) ->

      ne-commencent-pas(("u".nil).("c"."r".nil).("r".nil).("d".nil).
                        ("l".nil).nil,q) ;

formes-possibles("r"."i"."l"."l"."i"."a"."f".q,faillir,
                 ind-pres(<"u"."a"."f".q,"l"."l"."i"."a"."f".q>,
                          <type-s-3,type-p-1>,i-pres).
                 ind-imp  ("l"."l"."i"."a"."f".q,type-1,i-imp).
                 ind-ps   ("l"."l"."i"."a"."f".q,type-2,i-ps).
                 ind-fs   ("i"."l"."l"."i"."a"."f".q,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."r"."u"."o"."m".nil,mourir,
                 ind-pres(<"r"."u"."e"."m".nil,"r"."u"."o"."m".nil>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("r"."u"."o"."m".nil,type-1,i-imp).
                 ind-ps   ("r"."u"."o"."m".nil,type-3,i-ps).
                 ind-fs   ("r"."u"."o"."m".nil,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."v"."r"."u"."o"."p".q,pourvoir,
                 ind-pres(<"i"."o"."v"."r"."u"."o"."p".q,
                           "o"."v"."r"."u"."o"."p".q>,
                          <type-s-1,type-p-2>,i-pres).
                 ind-imp  ("y"."o"."v"."r"."u"."o"."p".q,type-1,i-imp).
                 ind-ps   ("v"."r"."u"."o"."p".q,type-3,i-ps).
                 ind-fs   ("r"."i"."o"."v"."r"."u"."o"."p".q,
                          type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."v"."a"."s".q,savoir,
                 ind-pres(<"a"."s".q,"v"."a"."s".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("v"."a"."s".q,type-1,i-imp).
                 ind-ps   ("s".q,type-3,i-ps).
                 ind-fs   ("u"."a"."s".q,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."v"."e"."d".q,devoir,
                 ind-pres(<"i"."o"."d".q,"e"."d".q>,
                          <type-s-1,type-p-6>,i-pres).
                 ind-imp  ("v"."e"."d".q,type-1,i-imp).
                 ind-ps   ("d".q,type-3,i-ps).
                 ind-fs   ("v"."e"."d".q,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."v"."u"."o"."m".q,mouvoir,
                 ind-pres(<"u"."e"."m".q,"v"."u"."e"."m".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("v"."u"."o"."m".q,type-1,i-imp).
                 ind-ps   ("m".q,type-3,i-ps).
                 ind-fs   ("v"."u"."o"."m".q,type-1,i-fs).nil) -> ;
```

```
formes-possibles("r"."i"."o"."l"."a"."v".q,valoir,
                 ind-pres(<"u"."a"."v".q,"l"."a"."v".q>,
                          <type-s-3,type-p-1>,i-pres).
                 ind-imp  ("l"."a"."v".q,type-1,i-imp).
                 ind-ps   ("l"."a"."v".q,type-3,i-ps).
                 ind-fs   ("d"."u"."a"."v".q,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."v"."u"."o"."p".nil,pouvoir,
                 ind-pres(<"u"."e"."p".nil,"v"."u"."o"."p".nil>,
                          <type-s-3,type-p-1>,i-pres).
                 ind-imp  ("v"."u"."o"."p".nil,type-1,i-imp).
                 ind-ps   ("p".nil,type-3,i-ps).
                 ind-fs   ("r"."u"."o"."p".nil,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."l"."u"."o"."v".nil,vouloir,
                 ind-pres(<"u"."e"."v".nil,"l"."u"."o"."v".nil>,
                          <type-s-3,type-p-1>,i-pres).
                 ind-imp  ("l"."u"."o"."v".nil,type-1,i-imp).
                 ind-ps   ("l"."u"."o"."v".nil,type-3,i-ps).
                 ind-fs   ("d"."u"."o"."v".nil,type-1,i-fs).nil) -> ;

formes-possibles("r"."i"."o"."e"."s"."s"."a".q,asseoir,
                 ind-pres(<"i"."o"."s"."s"."a".q,"y"."o"."s"."s"."a".q>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp  ("y"."o"."s"."s"."a".q,type-1,i-imp).
                 ind-ps   ("s"."s"."a".q,type-2,i-ps).
                 ind-fs   ("i"."o"."s"."s"."a".q,type-1,i-fs).nil) -> ;

formes-possibles("e"."r"."p".q,rompre,
                 ind-pres(<"p".q,"p".q>,<type-s-1,type-p-1>,i-pres).
                 ind-imp  ("p.q,type-1,i-imp).
                 ind-ps   ("p".q,type-2,i-ps).
                 ind-fs   ("p".q,type-1,i-fs).nil) -> ;


formes-possibles("e"."r"."c"."n"."i"."a"."v".q,vaincre,,
                 ind-pres(<"c"."n"."i"."a"."v".q,"u"."q"."n"."i"."a"."v".q>,
                          <type-s-2,type-p-1>,i-pres).
                 ind-imp  ("u"."q"."n"."i"."a"."v".q,type-1,i-imp).
                 ind-ps   ("u"."q"."n"."i"."a"."v".q,type-2,i-ps).
                 ind-fs   ("c"."n"."i"."a"."v".q,type-1,i-fs).nil) -> ;


formes-possibles("r"."i".q,finir,
                          ind-pres(<"i".q,"s"."s"."i".q>,
                                   <type-s-1,type-p-1>,i-pres).
                          ind-imp  ("s"."s"."i".q,type-1,i-imp).
                          ind-ps   ("i".q,type-1,i-ps).
                          ind-fs   ("i".q,type-1,i-fs).nil)

                 -> ne-commencent-pas(("o".nil).("n"."e".nil).
                                      ("r"."e".nil).("t".nil).
                                      ("r"."v".nil).("r"."f".nil).
                                      ("l"."l".nil).("m".nil).
                                      ("r".nil).("u"."f".nil).nil,q)

                    dif(q,"v"."r"."e"."s".nil)
                    dif(q,"v"."r"."e"."s"."s"."e"."d".nil)
                    dif(q,"v"."r"."e"."s"."s"."e"."r".nil) ;
```

```
formes-possibles("e"."r"."i"."d"."u"."a"."m".nil,maudire,
                 ind-pres(<"i"."d"."u"."a"."m".nil,
                          "s"."s"."i"."d"."u"."a"."m".nil>,
                          <type-s-1,type-p-1>,i-pres).
                 ind-imp ("i"."d"."u"."a"."m".nil,type-1,i-imp).
                 ind-ps  ("d"."u"."a"."m".nil,type-2,i-ps).
                 ind-fs  ("i"."d"."u"."a"."m".nil,type-1,i-fs).nil)

       -> ;

formes-possibles("r"."e".q,aimer,
                 ind-pres(<"e".q,q>,
                          <type-s-4,type-p-1>,i-pres).
                 ind-imp (q,type-1,i-imp).
                 ind-ps  (q,type-4,i-ps).
                 ind-fs  ("e".q,type-1,i-fs).nil)

       -> ne-commencent-pas(("g".nil).("l"."e".nil).
                            ("t"."e".nil).("y"."o".nil).
                            ("y"."u".nil).nil,q) ;

formes-possibles("r"."e"."g".q,manger,
                 ind-pres(<"e"."g".q,"e"."g".q>,
                          <type-s-4,type-p-10>,i-pres).
                 ind-imp ("g".q,type-2,i-imp).
                 ind-ps  ("e"."g".q,type-4,i-ps).
                 ind-fs  ("e"."g".q,type-1,i-fs).nil) -> ;

formes-possibles("r"."e"."l"."e".q,appeler,
                 ind-pres(<"e"."l"."l"."e".q,"l"."e".q>,
                          <type-s-4,type-p-11>,i-pres).
                 ind-imp ("l"."e".q,type-1,i-imp).
                 ind-ps  ("l"."e".q,type-4,i-ps).
                 ind-fs  ("e"."l"."l"."e".q,type-1,i-fs).nil)

       -> ne-commencent-pas(("c".nil).("t"."n"."a"."m"."e"."d".nil).
                 ("s".nil).("t"."r"."a"."c"."e".nil).("g".nil).
                 ("t"."r"."a"."m".nil).("d"."o"."m".nil).
                 ("p".nil).nil,q);

formes-possibles("r"."e"."t"."e".q,jeter,
                 ind-pres(<"e"."t"."t"."e".q,"t"."e".q>,
                          <type-s-4,type-p-11>,i-pres).
                 ind-imp ("t"."e".q,type-1,i-imp).
                 ind-ps  ("t"."e".q,type-4,i-ps).
                 ind-fs  ("e"."t"."t"."e".q,type-1,i-fs).nil)

       -> ne-commencent-pas(("h"."c"."a".nil).("u"."g"."e"."b".nil).
                 ("s"."r"."o"."c".nil).("h"."c"."o"."r"."c".nil).
                 ("l"."i"."f".nil).("r"."u"."f".nil).
                 ("l"."a"."h".nil).nil,q);
```

```
formes-possibles("r"."e"."l"."e".q,modeler,
                 ind-pres(<"e"."l"."e".q,"l"."e".q>,
                          <type-s-4,type-p-1>,i-pres).
                 ind-imp ("l"."e".q,type-1,i-imp).
                 ind-ps  ("l"."e".q,type-4,i-ps).
                 ind-fs  ("e"."l"."e".q,type-1,i-fs).nil)

      -> dans(q,("c".nil).("c"."e"."d".nil).("c"."e"."r".nil).
                ("t"."n"."a"."m"."e"."d".nil).("s".nil).
                ("t"."r"."a"."c"."e".nil).("g".nil).
                ("g"."e"."d".nil).("g"."n"."o"."c".nil).
                ("g"."r"."u"."s".nil).("t"."r"."a"."m".nil).
                ("d"."o"."m".nil).("p".nil).nil) ;

formes-possibles("r"."e"."t"."e".q,acheter,
                 ind-pres(<"e"."t"."e".q,"t"."e".q>,
                          <type-s-4,type-p-1>,i-pres).
                 ind-imp ("t"."e".q,type-1,i-imp).
                 ind-ps  ("t"."e".q,type-4,i-ps).
                 ind-fs  ("e"."t"."e".q,type-1,i-fs).nil)

      -> dans(q,("h"."c"."a".nil).("h"."c"."a"."r".nil).
                ("u"."g"."e"."b".nil).("s"."r"."o"."c".nil).
                ("h"."c"."o"."r"."c".nil).("l"."i"."f".nil).
                ("r"."u"."f".nil).("l"."a"."h".nil).nil) ;

formes-possibles("r"."e"."y".x.q,broyer,
                 ind-pres(<"e"."i".x.q,x.q>,
                          <type-s-4,type-p-2>,i-pres).
                 ind-imp ("y".x.q,type-1,i-imp).
                 ind-ps  ("y".x.q,type-4,i-ps).
                 ind-fs  ("e"."i".x.q,type-1,i-fs).nil)

      -> dans(x,"o"."u".nil)
         ne-commence-pas("o"."v"."n"."e".nil,x.q) ;

formes-possibles("r"."e"."y"."o"."v"."n"."e".q,envoyer,
                 ind-pres(<"e"."i"."o"."v"."n"."e".q,
                          "o"."v"."n"."e".q>,
                          <type-s-4,type-p-2>,i-pres).
                 ind-imp ("y"."o"."v"."n"."e".q,type-1,i-imp).
                 ind-ps  ("y"."o"."v"."n"."e".q,type-4,i-ps).
                 ind-fs  ("r"."e"."v"."n"."e".q,type-1,i-fs).nil) ->
```

```
"terminaisons types de l'indicatif"

"present"

ind-pres(nil,type-etre,<"s"."i"."u"."s".nil,
                        "s"."e".nil,
                        "t"."s"."e".nil,
                        "s"."e"."m"."m"."o"."s".nil,
                        "s"."e"."t"."e".nil,
                        "t"."n"."o"."s".nil>) -> ;

ind-pres(nil,type-avoir,<"i"."a".nil,
                         "s"."a".nil,
                         "a".nil,
                         "s"."n"."o"."v"."a".nil,
                         "z"."e"."v"."a".nil,
                         "t"."n"."o".nil>) -> ;

ind-pres(nil,type-aller,<"s"."i"."a"."v".nil,
                         "s"."a"."v".nil,
                         "a"."v".nil,
                         "s"."n"."o"."l"."l"."a".nil,
                         "z"."e"."l"."l"."a".nil,
                         "t"."n"."o"."v".nil>) -> ;

ind-pres(<s,p>,<t-s,t-p>,<j,t,i,n,v,i'>) ->
       ind-pres-s(s,t-s,<j,t,i>) ind-pres-p(p,t-p,<n,v,i'>) ;

"present singulier"

ind-pres-s(p,type-s-1,<"s".p,"s".p,"t".p>) -> ;

ind-pres-s(p,type-s-2,<"s".p,"s".p,p>) -> ;

ind-pres-s(p,type-s-3,<"x".p,"x".p,"t".p>) -> ;

ind-pres-s(p,type-s-4,<p,"s".p,p>) -> ;

"present pluriel"

ind-pres-p(p,type-p-1,<"s"."n"."o".p,"z"."e".p,"t"."n"."e".p>) -> ;

ind-pres-p(p,type-p-2,<"s"."n"."o"."y".p,
                       "z"."e"."y".p,
                       "t"."n"."e"."i".p>) -> ;

ind-pres-p(p,type-p-3,<"s"."n"."o".x."u"."o".p,"z".
                       "e".x."u"."o".p,
                       "t"."n"."e".x."u"."e".p>) ->

            dans(x,"l"."v".nil) ;
```

```
ind-pres-p(p,type-p-4,<"s"."n"."o".p,
                       "z"."e".p,
                       "t"."n"."e"."n".p>) -> ;

ind-pres-p(p,type-p-5,<"s"."n"."o"."s"."i"."a".p,
                       "s"."e"."t"."i"."a".p,
                       "t"."n"."o".p>) -> ;

ind-pres-p(p,type-p-6,<"s"."n"."o"."v".x.p,
                       "z"."e"."v".x.p,
                       "t"."n"."e"."v"."i"."o".x.p>)

          -> dans(x,"e"."u".nil) ;

ind-pres-p(p,type-p-7,<"s"."n"."o"."s".p,
                       "s"."e"."t".p,
                       "t"."n"."e"."s".p>) -> ;

ind-pres-p(p,type-p-8,<"s"."n"."o"."n"."e".p,
                       "z"."e"."n"."e".p,
                       "t"."n"."e"."n"."n"."e"."i".p>) -> ;

ind-pres-p(p,type-p-9,<"s"."n"."o"."r"."e".p,
                       "z"."e"."r"."e".p,
                       "t"."n"."e"."r"."e"."i".p>) -> ;

ind-pres-p(p,type-p-10,<"s"."n"."o".p,"z".p,"t"."n".p>) -> ;

ind-pres-p(x.p,type-p-11,<"s"."n"."o".x.p,
                          "z"."e".x.p,
                          "t"."n"."e".x.x.p>) -> dans(x,"t"."l".nil)

"imparfait"

ind-imp(p,type-1,<"s"."i"."a".p,
                  "s"."i"."a".p,
                  "t"."i"."a".p,
                  "s"."n"."o"."i".p,
                  "z"."e"."i".p,
                  "t"."n"."e"."i"."a".p>) ->;

ind-imp(p,type-2,<"s"."i"."a"."e".p,
                  "s"."i"."a"."e".p,
                  "t"."i"."a"."e".p,
                  "s"."n"."o"."i".p,
                  "z"."e"."i".p,
                  "t"."n"."e"."i"."a"."e".p>) ->;
```

```
"passe-simple"

ind-ps(p,type-1,<"s".p,
                 "s".p,
                 "t".p,
                 "s"."e"."m".p,
                 "s"."e"."t".p,
                 "t"."n"."e"."r".p>) ->;

ind-ps(p,type-2,<"s"."i".p,
                 "s"."i".p,
                 "t"."i".p,
                 "s"."e"."m"."i".p,
                 "s"."e"."t"."i".p,
                 "t"."n"."e"."r"."i".p>) ->;

ind-ps(p,type-3,<"s"."u".p,
                 "s"."u".p,
                 "t"."u".p,
                 "s"."e"."m"."u".p,
                 "s"."e"."t"."u".p,
                 "t"."n"."e"."r"."u".p>) ->;

ind-ps(p,type-4,<"i"."a".p,
                 "s"."a".p,
                 "a".p,
                 "s"."e"."m"."a".p,
                 "s"."e"."t"."a".p,
                 "t"."n"."e"."r"."e".p>) ->;

"futur-simple"

ind-fs(p,type-1,<"i"."a"."r".p,
                 "s"."a"."r".p,
                 "a"."r".p,
                 "s"."n"."o"."r".p,
                 "z"."e"."r".p,
                 "t"."n"."o"."r".p>) ->;
```

3.5 UN DIALOGUE EN LANGUE NATURELLE.
(par R. Pasero)

Ce programme permet de dialoguer en francais avec la machine, le but du dialogue étant de lui décrire un monde.

Les informations sont fournies à la machine, de facon conversationnelle, à l'aide de phrases déclaratives. Pour chacune d'elles le système verifie:

- que les présuppositions qui lui sont associées ne contredisent pas les informations qu'elle possède deja. Sinon la phrase est declarée absurde.

- que les assertions ne contredisent pas les informations deja acquises. Sinon la phrase est declarée fausse.

Si la phrase est absurde, la machine n'enregistre aucune nouvelle information: elle ne modifie pas son monde. Si elle est fausse, le monde est augmenté des informations associées aux présuppostions. Sinon, la phrase est declarée vraie, et toutes les informations qui lui sont associées, présuppositions et assertions sont enregistrées. Le système fournit alors une réponse à la phrase précisant ainsi à l'interlocuteur comment évolue son monde.

Ce programme représente un noyau de base autour duquel peuvent se constituer des systèmes plus élaborés en ce qui concerne le sous-ensemble et la sémantique du francais accepté. Il possède les modules fondamentaux nécessaires à la réalisation de tels systèmes, permettant de représenter la sémantique à 3 valeurs d'une phrase par deux formules à 2 valeurs.

Il est intéressant, par ailleurs, par la technique de programmation consistant à utiliser des "lemmes" pour ne démontrer un résultat qu'une fois.

DESCRIPTION DU PROGRAMME.

La fonction essentielle du programme est, étant donnée une suite de phrases:

P1,P2,...Pn,...

de calculer, pour chacune d'elles, une réponse.

Pour calculer la réponse "Ri" à la phrase "Pi", définie par "phrase(i,p)", le système procède comme suit:

1- la phrase "Pi" est analysée: le résultat de l'analyse est une structure profonde "SPi" définie par "structprof(i,s-p)".

Cette structure profonde est définie par la forme normale de Backus:

```
<phrase> ::= pr(<verbe1>(<argument>))
         ::= pr(<verbe2(<argument>,<argument>))
         ::= non(<phrase>)
         ::= et(<phrase>,<phrase>)
         ::= un(<variable>,<phrase>,<phrase>)
         ::= le(<variable>,<phrase>,<phrase>)
         ::= chaque(<variable>,<phrase>,<phrase>)
         ::= aucun(<variable>,<phrase>,<phrase>)

<argument> ::= <variable>
           ::= <nom propre>

<variable> ::= "toute variable prolog"

<nom propre> ::= "n'importe quel nom propre"

<verbe1> ::= "tout verbe ou nom commun à 1 argument"

<verbe2> ::= "tout verbe ou nom commun à 2 arguments"
```

Si la phrase est incorrecte l'analyse produit comme résultat le terme "agrammatical".

2- cette structure profonde "SPi" est traduite alors en une formule logique à 3 valeurs "LPi" définie par "formlogic(i,l-p)" si "SPi" n'est pas le terme "agrammatical".

Etant donnés les ensembles N des entiers naturels, R1 et R2 des symboles relationnels à 1 et 2 arguments, l'ensemble E des expressions logiques est défini comme l'union de l'ensemble T des termes logiques et F des formules logiques; T et F sont définis comme les plus petits ensembles de mots sur

N U R1 U R2 U (non,et,cond,exist,choix,vrai,pr)

vérifiant les propriétes suivantes:

(1a) x est dans T si x est dans N

(2a) choix(x,f) est dans t si x est dans N, f dans F

(1b) vrai est dans F

(2b) pr(r(t)) est dans F si r est dans R1 et t dans T

(3b) pr(r(t1,t2)) est dans F si r est dans R2, t1 et t2 dans T

(4b) non(f) est dans F si f est dans F

(5b) et(f1,f2) est dans F si f1 et f2 sont dans F

(6b) cond(f1,f2) est dans F si f1 et f2 sont dans F

(7b) exist(x,f) est dans F si x est dans N, f dans F

La sémantique des formules logiques est définie par un triplet <U,B,v> où U est l'ensemble des termes fermés, B=(1,0,z), v une application (valuation) qui associe à toute expression fermée e une valeur v(e) telle que:

-v(t)=t si t est dans U

-v(vrai)=1

-v(pr(r(t))) est dans (0,1)

-v(pr(r(t1,t2))) est dans (0,1)

-v(non(f))=1 ssi v(f)=0
v(non(f))=0 ssi v(f)=1
v(non(f))=z sinon

-v(et(f1,f2))=1 ssi v(f1)=v(f2)=1
v(et(f1,f2))=z ssi v(f1)=z ou v(f2)=z
v(et(f1,f2))=0 sinon

-v(cond(f1,f2))=1 ssi v(f1)=v(f2)=1
v(cond(f1,f2))=0 ssi v(f1)=1 et v(f2)=0
v(cond(f1,f2))=z sinon

-v(exist(x,f))=1 ssi v(ss(f,<x,choix(x,f)>))=1
v(exist(x,f))=0 ssi v(ss(f,<x,choix(x,f)>))=0
et pour tout t dans U on a
v(ss(f,<x,t>))#1
v(exist(x,f))=z ssi pour tout t dans U on a
v(ss(f,<x,t>))=z

L'expression ss(f,<x,t>) représente l'expression obtenue en remplacant dans f toute occurence libre de x par t.

3- à partir de cette formule logique à 3 valeurs "LPi" le programme calcule deux formules logiques à 2 valeurs:

- une formule "PPi" représentant les présuppositions de "LPi", définie par "presup(i,p-p)".

- une formule "APi" représentant les assertions associées à "LPi", définie par "assert(i,a-p)".

4- on dit alors que:

- la phrase "Pi" est absurde ("semantique(i,absurde)") si le système :

(F1,...Fi-1,PPi)

est inconsistant ("inconsistant(pr(i))").

- la phrase "Pi" est fausse ("semantique(i,faux)") si le système :

  (F1,...Fi-1,et(PPi,APi))

  est inconsistant ("inconsistant(as(i))") alors que le précédent ne l'est pas.

- la phrase "Pi" est vraie ("semantique(i,vrai)") si aucun des 2 systèmes n'est inconsistant.

avec chaque Fj défini par:

- Fj=vrai si la phrase "Pj" est absurde ou agrammaticale.
- Fj=PPj si la phrase "Pj" est fausse
- Fj=et(PPj,APj) si la phrase "Pj" est vraie

5- pour réaliser les tests d'inconsistance d'un système de formules nous utilisons un démonstrateur fonctionnant sur le principe de SL-resolution.

La propriété :

exist(x,f) ssi ss(f,<x,choix(x,f)>)

est utilisée dans l'algorithme de productions des clauses associées à la phrase "Pi" ("clauses(i,c)") pour supprimer les quantificateurs. L'axiome de généralisation prend alors ici la forme suivante:

non(ss(f,<x,t>)) si non(ss(f,<x,choix(x,f)>))

quel que soit le terme t.

6- les réponses "Ri" à la phrase "Pi" sont alors définies comme suit par "reponse(i,r)" si "Pi" n'est pas agrammaticale:

- "Ri"="ok" si la phrase "Pi" est vraie.
- "Ri"="cette phrase est fausse" si "Pi" est fausse.
- "Ri"="cette phrase est absurde" si "Pi" est absurde.

A ces réponses nous en avons ajouté 2:

- "Ri"="cette phrase est agrammaticale" dans le cas où la phrase "Pi" est incorrecte. Aucune valeur sémantique ne lui est associée.
- "Ri"="bonsoir" dans le cas où la phrase "Pi" est elle-meme la phrase "bonsoir". Dans ce cas la production de

cette réponse a pour effet de bord d'arreter le programme.

Le système dispose en outre:

- d'une "boucle d'échange" permettant de calculer dans l'ordre les réponses "Ri" aux phrases "Pi".
- d'une procédure d'entrée ("lire(x)") pour lire une phrase.
- d'une procédure de sortie ("dire(x)") pour imprimer les réponses.
- d'une procédure "lemme(x)" permettant de ne démontrer certains résultats qu'une seule fois (la 1ere. fois). Le système mémorise (par ajout) qu'un résultat a été démontré.

Le programme utilisé ne disposait pas d'axiomes généraux de déductions, en particulier l'égalité n'était pas traitée.

Nous disposions en outre d'un mini-analyseur du francais que nous n'avons pas fait figurer.

PROGRAMME ET EXEMPLE.

Nous présentons ici le programme, sans l'analyseur.

```
"(0) boucle d'echange"

bonjour ->
   initialiser
   nb-entier(i)
   reponse(i,r)
   dire(r)
   /;

"(1) definition des reponses"

reponse(i,bonsoir.nil) ->
   lemme(phrase(i,mt-bonsoir.".".nil))
   /;
reponse(i,cette.phrase.est.agrammaticale.nil) ->
   lemme(structprof(i,agrammatical))
   /;
reponse(i,ok.nil) ->
   lemme(semantique(i,vrai))
   /;
reponse(i,cette.phrase.est.fausse.nil) ->
   lemme(semantique(i,faux))
   /;
reponse(i,cette.phrase.est.absurde.nil) ->
   lemme(semantique(i,absurde));

"(2) definition des enonces"

phrase(i,p) ->
   lire(p);

structprof(i,s-p) ->
   lemme(phrase(i,p))
   analyse(s-p,<p,nil>);

formlogic(i,l-p) ->
   lemme(structprof(i,s-p))
   traduc(s-p,l-p);

presup(i,p-p) ->
   lemme(formlogic(i,l-p))
   presupposition(l-p,p-p);

assert(i,a-p) ->
   lemme(formlogic(i,l-p))
   assertion(l-p,a-p);

clauses(pr(i),c) ->
   lemme(presup(i,p-p))
   axiome(p-p,c);

clauses(as(i),c) ->
   lemme(assert(i,a-p))
   axiome(a-p,c);
```

```
"(3) analyseur"

analyse(p,<u,v>) -> ...


"(4) definition semantique"

semantique(i,absurde) ->
   inconsistant(pr(i))
   / ;
semantique(i,faux) ->
   inconsistant(as(i))
   / ;
semantique(i,vrai) ->;

"(5) traduction de la structure profonde en enonce logique"

traduc(non(p),non(p1)) ->
   traduc(p,p1);
traduc(et(p,q),et(p1,q1)) ->
   traduc(p,p1)
   traduc(q,q1);
traduc(un(x,p,q),exist(x,r)) ->
   nvl-var(x)
   traduc(et(p,q),r);
traduc(le(x,p,q),cond(et(exist(x,p1),u),q2)) ->
   nvl-var(x)
   traduc(p,p1)
   eg(u,non(exist(x,et(p1,non(pr(egal(x,choix(x,p1))))))))
   traduc(q,q1)
   subs(q1,<x,choix(x,p1)>,q2);
traduc(chaque(x,p,q),non(exist(x,r))) ->
   nvl-var(x)
   traduc(et(p,non(q)),r);
traduc(aucun(x,p,q),non(exist(x,r))) ->
   nvl-var(x)
   traduc(et(p,q),r);
traduc(pr(x),pr(x)) ->;

"(6) definition des presuppositions"

presupposition(vrai,vrai) -> ;
presupposition(non(p),p1) ->
   presupposition(p,p1);
presupposition(et(p,q),r) ->
   presupposition(p,p1)
   presupposition(q,q1)
   reduction(et(p1,q1),r);
presupposition(cond(p,q),r) ->
   presupposition(p,p1)
   assertion(p,p2)
   presupposition(q,q1)
   reduction(et(p2,q1),r1)
   reduction(et(p1,r1),r);
presupposition(exist(x,p),r) ->
   presupposition(p,r1)
   subs(r1,<x,choix(x,p)>,r);
presupposition(pr(x),vrai) ->;

reduction(et(vrai,p),p) -> ;
```

```
reduction(et(p,vrai),p) ->
   dif(p,vrai);
reduction(et(p,q),et(p,q)) ->
   dif(p,vrai)
   dif(q,vrai);

"(7) definition des assertions"

assertion(vrai,vrai) -> ;
assertion(non(p),non(p2)) ->
   assertion(p,p2);
assertion(et(p,q),et(p2,q2)) ->
   assertion(p,p2)
   assertion(q,q2);
assertion(cond(p,q),q2) ->
   assertion(q,q2);
assertion(exist(x,p),ss(r,<x,choix(x,p)>)) ->
   assertion(p,r);
assertion(pr(x),pr(x)) ->;

"(8) production des clauses"

axiome(non(vrai),nil) -> ;
axiome(et(p,q),c) ->
   axiome(p,c);
axiome(et(p,q),c) ->
   axiome(q,c);
axiome(non(et(p,q)),c) ->
   axiome(non(p),c1)
   axiome(non(q),c2)
   conc(c1,c2,c);
axiome(ss(r,<x,y>),c) ->
   subs(r,<x,y>,r1)
   axiome(r1,c);
axiome(non(ss(r,<x,y>)),c) ->
   individu(v)
   subs(r,<x,ind(v)>,r1)
   axiome(non(r1),c);
axiome(non(non(p)),c) ->
   axiome(p,c);
axiome(non(pr(x)),moins(x').nil) ->
   simple(pr(x),x');
axiome(pr(x),plus(x').nil) ->
   simple(pr(x),x');

simple(pr(<p,u,v>),<p,u',v'>) ->
   simple(u,u')
   simple(v,v');
simple(pr(<p,u>),<p,u'>) ->
   simple(u,u');
simple(<p,u,v>,<p,u',v'>) ->
   simple(u,u')
   simple(v,v');
simple(non(u),non(u')) ->
   simple(u,u');
simple(u,u) ->
   nompropre(u,l);
simple(u,u) ->
   meta-variable(u);
simple(ind(u),u) ->;
```

```
"(9) substitution"

subs(x,<x,y>,y) -> ;
subs(vrai,<x,y>,vrai) -> ;
subs(non(p),<x,y>,non(p1)) ->
   subs(p,<x,y>,p1);
subs(et(p,q),<x,y>,et(p1,q1)) ->
   subs(p,<x,y>,p1)
   subs(q,<x,y>,q1);
subs(cond(p,q),<x,y>,cond(p1,q1)) ->
   subs(p,<x,y>,p1)
   subs(q,<x,y>,q1);
subs(exist(x,p),<x,y>,exist(x,p)) -> ;
subs(exist(z,p),<x,y>,exist(z,p1)) ->
   dif(z,x)
   subs(p,<x,y>,p1);
subs(choix(x,p),<x,y>,choix(x,p)) -> ;
subs(choix(z,p),<x,y>,choix(z,p1)) ->
   dif(z,x)
   subs(p,<x,y>,p1);
subs(ss(p,<x,t>),<x,y>,ss(p,<x,t1>)) ->
   subs(t,<x,y>,t1);
subs(ss(p,<z,t>),<x,y>,ss(p1,<z,t1>)) ->
   dif(z,x)
   subs(p,<x,y>,p1)
   subs(t,<x,y>,t1);
subs(pr(<p,u>),<x,y>,pr(<p,u1>)) ->
   subs(u,<x,y>,u1);
subs(pr(<p,u,v>),<x,y>,pr(<p,u1,v1>)) ->
   subs(v,<x,y>,v1)
   subs(u,<x,y>,u1);
subs(z,<x,y>,z) ->
   meta-variable(z)
   dif(z,x);
subs(z,s,z) ->
   nompropre(z,1);
subs(ind(i),s,ind(i)) ->;

"(10) demonstrateur"

inconsistant(i) ->
   lemme(clauses(i,c))
   effacer(i,c,nil);

effacer(i,nil,1) ->;
effacer(i,c1.c2,1) ->
   suite(i,c1,1,1)
   effacer(i,c2,1);

suite(i,c,nil,1) ->
   lemme(clauses(i,c1))
   oppose(c,c2)
   minus(c1,c2,c3)
   effacer(i,c3,c.1);
suite(as(i),c,nil,1) ->
   lemme(clauses(pr(i),c1))
   oppose(c,c2)
   minus(c1,c2,c3)
   effacer(as(i),c3,c.1);
suite(<o,i>,c,nil,1) ->
```

```
      anterieur(j,i)
      axiome-bis(j,c1)
      oppose(c,c2)
      minus(c1,c2,c3)
      effacer(<o,i>,c3,c.1);
   suite(i,c,c1.11,1) ->
      oppose(c,c1);
   suite(i,c,c1.11,1) ->
      dif(c,c1)
      suite(i,c,11,1);

   axiome-bis(j,c) ->
      lemme(semantique(j,v))
      dif(v,absurde)
      lemme(clauses(pr(j),c));
   axiome-bis(j,c) ->
      lemme(semantique(j,vrai))
      lemme(clauses(as(j),c));

   "(11) definitions des individus"

   individu(x) ->
      geler(x,individu-bis(x));

   individu-bis(x) ->
      nompropre(x,1);
   individu-bis(choix(x,f)) ->
      meta-variable(x)
      formule(f);

   meta-variable(x) ->
      entier(x);

   formule(pr(<p,u>)) ->
      argument(u);

   formule(pr(<p,u,v>)) ->
      argument(u)
      argument(v);

   formule(non(f)) ->
      formule(f);

   formule(et(f1,f2)) ->
      formule(f1)
      formule(f2);

   formule(cond(f1,f2)) ->
      formule(f1)
      formule(f2);

   formule(exist(x,f)) ->
      meta-variable(x)
      formule(f);

   argument(x) ->
      geler(x,argument-bis(x));

   argument-bis(x) ->
      meta-variable(x);
```

```
argument-bis(x) ->
   individu-bis(x);

"(12) definitions utiles"

initialiser ->
   affecter(variable,0);

nvl-var(x) ->
   val(add(variable,1),x)
   affecter(variable,x);

oppose(plus(p),moins(p)) ->;
oppose(moins(p),plus(p)) ->;

conc(nil,x,x) ->;
conc(e.x,y,e.z) ->
   conc(x,y,z);

anterieur(i,j) ->
   val(inf(1,j),1)
   val(sub(j,1),i)
   grammaticale(i);
anterieur(i,j) ->
   val(inf(2,j),1)
   val(sub(j,1),k)
   anterieur(i,k);

grammaticale(i) ->
   lemme(structprof(i,s-p))
   dif(s-p,agrammatical);

nb-entier(1) ->;
nb-entier(i) ->
   nb-entier(j)
   val(add(j,1),i);

minus(x.y,x,y) ->;
minus(x.y,z,x.y1) -> minus(y,z,y1);

lemme(<p,i,x>) ->
   homologue(p,p1,p2)
   <p1,i>
   /
   <p2,i,x>;
lemme(<p,i,x>) ->
   homologue(p,p1,p2)
   <p,i,y>
   ajout(<<p2,i,y>,nil>)
   ligne ex(<p,i,y>) ligne
   impasse;
lemme(<p,i,x>) ->
   homologue(p,p1,p2)
   ajout(<<p1,i>,nil>)
   <p2,i,x>;

homologue(phrase,deja-connu-phrase,valeur-phrase) ->;
homologue(structprof,deja-connu-structprof,valeur-structprof) ->;
homologue(formlogic,deja-connu-formlogic,valeur-formlogic) ->;
```

```
homologue(presup,deja-connu-presup,valeur-presup) ->;
homologue(assert,deja-connu-assert,valeur-assert) ->;
homologue(clauses,deja-connu-clauses,valeur-clauses) ->;
homologue(semantique,deja-connu-semantique,valeur-semantique) ->;

dire(bonsoir.nil) ->
   /
   ligne
   exm("bonsoir");
dire(r) ->
   ligne
   imprimer(r)
   ligne
   impasse;

imprimer(nil) -> ;
imprimer(x.l) -> ex(x)
   exm(" ")
   imprimer(l);

lire(p) ->
   ligne
   exm("a vous")
   ligne
   ligne
   in-ph(p);
```

Voici un exemple simple comportant certains résultats intermediaires. Les sorties de la machine sont décalées à droite.

```
>

bonjour;

   a vous

pierre habite a mazargues.

   phrase(1,mt-pierre.mt-habite.mt-a.mt-mazargues.".".nil)

   structprof(1,pr(habite-a(mazargues,pierre)))

   formlogic(1,pr(habite-a(mazargues,pierre)))

   presup(1,vrai)

   assert(1,pr(habite-a(mazargues,pierre)))

   clauses(as(1),plus(habite-a(mazargues,pierre)).nil)

   semantique(1,vrai)

   ok

   a vous

pierre n'habite pas a mazargues.

   cette phrase est fausse

   a vous

aucune personne n'habite a mazargues.

   ok

   a vous

pierre est une personne.

   cette phrase est fausse

   a vous

la personne qui habite a mazargues pese 50 kilos.

   phrase(5,mt-la.mt-personne.mt-qui.mt-habite.mt-a.
      mt-mazargues.mt-pese.50.mt-kilos.".".nil)

   structprof(5,le(x55,et(pr(personne(x55)),
      pr(habite-a(mazargues,x55))),
      pr(pese(x55,kilo(50)))))

   formlogic(5,cond(et(exist(2,et(pr(personne(2)),
      pr(habite-a(mazargues,2)))),
```

```
        non(exist(2,et(et(pr(personne(2)),
        pr(habite-a(mazargues,2))),
        non(pr(egal(2,choix(2,et(pr(personne(2)),
        pr(habite-a(mazargues,2))))))))),
        pr(pese(choix(2,et(pr(personne(2)),
        pr(habite-a(mazargues,2)))),kilo(50)))))

    presup(5,et(ss(et(pr(personne(2)),pr(habite-a(mazargues,2))),
        <2,choix(2,et(pr(personne(2)),pr(habite-a(mazargues,2))))>),
        non(ss(et(et(pr(personne(2)),pr(habite-a(mazargues,2))),
        non(pr(egal(2,choix(2,et(pr(personne(2)),
        pr(habite-a(mazargues,2))))))),
        <2,choix(2,et(et(pr(personne(2)),
        pr(habite-a(mazargues,2))),non(pr(egal(2,
        choix(2,et(pr(personne(2)),pr(habite-a(mazargues,2))))))))>))))

    clauses(pr(5),plus(personne(choix(2,et(personne(2),
        habite-a(mazargues,2))))).nil)

    clauses(pr(5),plus(habite-a(mazargues,choix(2,et(personne(2),
        habite-a(mazargues,2))))).nil)

    clauses(pr(5),moins(personne(x80)).moins(habite-a(mazargues,x80)).
        plus(egal(x80,choix(2,et(personne(2),
        habite-a(mazargues,2))))).nil)

    semantique(5,absurde)

    cette phrase est absurde

    a vous

bonsoir.

    bonsoir
>
```

4 QUELQUES PROGRAMMES

4.1 LES MUTANTS

Il s'agit de produire des "mutants" issus d'animaux différents.
Pour cela, les animaux sont connus par leur nom sous forme de chaine de caractères. Deux animaux donnent naissance à un mutant si la fin du nom du premier animal est identique au début du nom du second.

L'aspect interessant de ce programme est qu'on y utilise une meme relation, "conc", de deux facons différentes: d'une part pour réunir deux listes, d'autre part pour décomposer une liste en deux sous-listes.

Voici les résultats produits en partant de l'ensemble d'animaux : alligator, tortue, caribou, ours, cheval, vache, lapin, pintade, hibou, bouquetin et chèvre :

```
>joli-mutant;
alligatortue
caribours
caribouquetin
chevalligator
chevalapin
vacheval
vachevre
lapintade
hibours
hibouquetin
>
```

et voici le programme :

"MUTANTS"

```
mutant(z) ->
    animal(x)
    animal(y)
    conc(a,b,x)
    dif(b,nil)
    conc(b,c,y)
    dif(c,nil)
    conc(x,c,z);

conc(nil,y,y) ->;
conc(e.x,y,e.z) -> conc(x,y,z);

joli-mutant -> mutant(z) exp(z) ;

exp(nil) -> ;
exp(a.l) -> exm(a) exp(l);

animal("a"."l"."l"."i"."g"."a"."t"."o"."r".nil) ->;
animal("t"."o"."r"."t"."u"."e".nil) ->;
animal("c"."a"."r"."i"."b"."o"."u".nil) ->;
animal("o"."u"."r"."s".nil) ->;
animal("c"."h"."e"."v"."a"."l".nil) ->;
animal("v"."a"."c"."h"."e".nil) ->;
animal("l"."a"."p"."i"."n".nil) ->;
animal("p"."i"."n"."t"."a"."d"."e".nil) ->;
animal("h"."i"."b"."o"."u".nil) ->;
animal("b"."o"."u"."q"."u"."e"."t"."i"."n".nil) ->;
animal("c"."h"."e"."v"."r"."e".nil) ->;
```

4.2 LOGIQUE ET BANQUES DE DONNEES.

4.2.1 UNE BANQUE DE DONNEES ADMINISTRATIVES.
(par M. van Caneghem)

Il s'agit d'une banque de données sur les villes des Bouches-du-Rhone qui est décrite par les relations:

- ville(x) qui dit que "x" est une ville des Bouches-du-Rhone.

- xxx(c,n) qui exprime que "c" est le code postal de la ville "xxx" et "n" le nombre de ses habitants (en 1975).

L'accès à ces informations est fait à l'aide de :

- code-postal(v,c) : "c" est le code postal de "v".

- nb-habitants(v,n) : "n" est le nombre d'habitants de "v".

- entre(x,a,b) : verifie que a<x<b .

- combien(p,n) : "n" est le nombre de fois où la relation "p" est vraie (à condition que "p" soit définie sans répétitions).

- somme(x,p,n) : "n" est la somme des valeurs de "x" telles que la relation p(x) soit vraie (aux memes conditions que ci-dessus).

Voici quelques exemples d'utilisation (la question est donnée en clair entre parenthèses) :

```
>code-postal(cassis,c);

   (quel est le code postal de cassis?)

c=13260

>code-postal(v,13200);

   (quel ville a le code postal 13200?)

v=arles

>nb-habitants(cassis,n);

   (combien est-ce qu'il y a d'habitants a cassis?)

n=5831

>nb-habitants(v,n) entre(n,1000,2000);

   (quels sont les villes qui ont entre 1000 et 2000 habitants?)

v=alleins  n=1041
v=cadolive  n=1115
v=condoux  n=1042
v=cuges-les-pins  n=1288
v=destrousse  n=1205
```

```
v=ensues-la-redonne  n=1699
v=eygalieres  n=1284
v=maillane  n=1430
v=maussane-les-alpilles  n=1352
 =molleges  n=1048
v=mouries  n=1876
v=plan-d-orgon  n=1745
v=rognes  n=1426
v=rousset  n=1626
v=st-cannat  n=1862
v=st-savournin  n=1140

>eg(p,ville(x)) combien(p,n);

   (combien est-ce qu'il y a de villes)

p=ville(x)  n=100

>eg(p,nb-habitants(v,x).entre(x,1000,2000)) combien(p,n);

   (combien est-ce qu'il y a de villes
    qui ont entre 1000 et 2000 habitants?)

=nb-habitants(v,x).entre(x,1000,2000)  n=16

>eg(<p,x>,<nb-habitants(v,u).entre(u,1000,2000),u>) somme(x,p,n);

   (combien est-ce qu'il y a de personnes qui vivent dans
    des villes qui ont entre 1000 et 2000 habitants?)

p=nb-habitants(v,x).entre(x,1000,2000)  N=22179
```

Et voici le programe :

```
de-postal(v,c) ->
   ville(v)
   <v,c,h>;

nb-habitants(v,n) ->
   ville(v)
   <v,c,n>;

entre(x,a,b) ->
   val(add(inf(a,x),inf(x,b)),2);

combien(p,n) -> affecter(compteur,0) impasse;
combien(p,n) ->
   p
   val(add(compteur,1),x)
   affecter(compteur,x)
   impasse;
combien(p,n) -> val(compteur,n);

somme(x,p,n) -> affecter(totaliseur,0) impasse;
somme(x,p,n) ->
   p
   val(add(totaliseur,x),y)
   affecter(totaliseur,y)
   impasse;
somme(x,p,n) -> val(totaliseur,n);


"Les villes"

ville(aix-en-provence) ->;
ville(albaron) ->;
ville(allauch) ->;
ville(alleins) ->;
ville(arles) ->;
...
lle(velaux) ->;
ille(venelles) ->;
ville(ventaben) ->;
ville(vernegues) ->;
ville(vitroiles) ->;

"code postal et nombre d'habitants"

aix-en-provence          (13100,114014) -> ;
albaron                  (13123,0) -> ;
allauch                  (13190,11149) -> ;
alleins                  (13980,1041) -> ;
arles                    (13200,50345) -> ;
...
velaux                   (13880,2638) -> ;
venelles                 (13770,2672) -> ;
ventabren                (13122,1537) -> ;
vernegues                (13116,285) -> ;
itrolles                 (13127,13441) -> ;
```

4.2.2 INTERROGATION PAR EVALUATION D'UNE FORMULE LOGIQUE.
(par R. Pasero d'après A. Colmerauer)

Dans cet exemple on a constitué une banque de données portant sur des individus dont on connait le nom, l'age, la ville d'origine et le fait qu'ils portent ou non des lunettes.

Tous ces renseignements sont résumés par une assertion telle que :

```
individu(candide,20,constantinople,non) -> ;
```

qui indique que l'individu nommé "candide" est agé de 20 ans, qu'il est né à "constantinople" et ne porte pas de lunettes.

On dispose également de relations élémentaires (les formules atomiques) portant sur ces données.

Le programme consiste à évaluer une formule logique construite à partir des formules atomiques, des connecteurs "et" et "ou" et des quantificateurs existentiel et universel portant sur des variables typées, c'est à dire dont le domaine des valeurs est précisé.

Ainsi, la question type que l'on peut poser est du genre :

"quelles sont les valeurs de x appartenant au domaine D pour lesquelles la propriété P est vraie ?"

ce qui est traduit par la formule :

```
element(x,ens(x,D,P)).
```

Par exemple, la question :

(1) dans quelle ville habite mimosa ?

se traduit par la formule :

```
element(x,ens(x,ville,habite-a(mimosa,x))) ,
```

(2) olive porte-t-elle des lunettes ?

se traduit par :

```
element(x,ens(x,booleen,lunette(olive,x)))
```

et enfin :

(3) quelles sont les villes ayant au moins un habitant age de moins de 20 ans et portant des lunettes ?

correspond à :

```
element(x,ens(x,ville,existe(y,nom,et(habite-a(y,x),
                                      et(est-age-de(y,a),
                                      et(inferieur(a,20),
                                         lunette(y,oui)))))))).
```

Ces trois questions ont été pré-enregistrées et sont

activées par "reponse-a-tout" qui écrit la question en clair suivie des réponses.

Voici ce que cela donne (les réponses de la machine sont précédées de "--->") :

```
>reponse-a-tout;

dans quelle ville habite mimosa ?

---> aspres-sur-buech

olive porte-t-elle des lunettes ?

---> non

quelles sont les villes ayant au moins un habitant
age de moins de 20 ans et portant des lunettes ?

 --> severac-le-chateau

---> aspres-sur-buech
```

Voici le programme complet :

```
"(1) banque de donnees"

individu(candide,20,constantinople,non) ->;
individu(cunegonde,20,constantinople,oui) ->;
individu(gontran,94,aspres-sur-buech,non) ->;
individu(casimir,2,severac-le-chateau,oui) ->;
individu(clementine,1,cucugnan,non) ->;
individu(popeye,99,aspres-sur-buech,oui) ->;
individu(olive,99,aspres-sur-buech,non) ->;
individu(mimosa,1,aspres-sur-buech,oui) ->;
individu(bip,15,pampelune,non) ->;
individu(ignace,114,loyola,oui) ->;
individu(balthazar,87,jerusalem,non) ->;
individu(gaspard,96,smyrne,oui) ->;
individu(melchior,34,kartoum,non) ->;

"(2) definition des types"

type(x,nom) -> nom(x);
type(x,age) -> age(x);
type(x,ville) -> ville(x);
type(x,booleen) -> booleen(x);

nom(candide) ->;
nom(cunegonde) ->;
nom(gontran) ->;
nom(casimir) ->;
nom(clementine) ->;
nom(popeye) ->;
nom(olive) ->;
nom(mimosa) ->;
nom(bip) ->;
nom(ignace) ->;
nom(balthazar) ->;
nom(gaspard) ->;
nom(melchior) ->;

age(20) ->;
age(94) ->;
age(2) ->;
age(1) ->;
age(99) ->;
age(15) ->;
age(114) ->;
age(87) ->;
age(96) ->;
age(34) ->;

ville(constantinople) ->;
ville(aspres-sur-buech) ->;
ville(severac-le-chateau) ->;
ville(cucugnan) ->;
ville(pampelune) ->;
ville(loyola) ->;
ville(jerusalem) ->;
ville(smyrne) ->;
```

```
ville(kartoum) ->;

booleen(oui) ->;
booleen(non) ->;

"(3) listes des formules atomiques"

atomique(habite-a(x,y)) ->;
atomique(est-age-de(x,y)) ->;
atomique(lunette(x,y)) ->;
atomique(plus-age(x,y)) ->;
atomique(inferieur(x,y)) ->;
atomique(different(x,y)) ->;

"(4) evaluation des formules atomiques"

habite-a(x,y) -> individu(x,a,y,b);

est-age-de(x,y) -> individu(x,y,v,b);

plus-age(x,y) ->
   individu(x,a,v,b)
   individu(y,a',v',b')
   val(inf(a',a),1);

lunette(x,y) -> individu(x,a,v,y);

inferieur(x,y) -> val(inf(x,y),1);

different(x,y) -> dif(x,y);

"(5) evaluation des formules"

vrai(p) ->
   atomique(p)
   p;

vrai(non(p)) ->
   non(vrai(p));

vrai(et(p,q)) ->
   vrai(p)
   vrai(q);

vrai(ou(p,q)) ->
   vrai(p) ;
vrai(ou(p,q)) ->
   vrai(q) ;

vrai(existe(x,t,p)) ->
   type(x,t)
   vrai(p);

vrai(tout(x,t,p)) ->
   non(vrai(existe(x,t,non(p))));
```

```
"(6) definitions utiles"

non(p) ->
    p
    /
    impasse;
non(p) ->;

"(7) calcul des reponses"

reponse-a-tout ->
    question(i,q)
    element(y,q)
    ligne
    exm("--->")
    ex(y)
    ligne;

 ement(x,ens(x,t,p)) ->
    type(x,t)
    vrai(p);

"(8) listes des questions"

question(1,ens(x,ville,habite-a(mimosa,x))) ->
    exm("(1) dans quelle ville habite mimosa ?");

question(2,ens(x,booleen,lunette(olive,x))) ->
    exm("(2) olive porte-t-elle des lunettes ?");

question(3,ens(x,ville,existe(y,nom,et(habite-a(y,x),et(est-age-de(y,a),
    et(inferieur(a,20),lunette(y,oui)))))) ->
    exm("(3) quelles sont les villes ayant au moins un habitant")
    exm("age de moins de 20 ans et portant des lunettes ?");
```

5.1 DIF

Cet exemple est du à Michel van Caneghem.
Il s'agit de résoudre le célèbre problème de crypt-arithmétique dans lequel, en remplacant chaque ocurence des lettres s,e,n,d,m,o,r,y par un meme chiffre, on ait :

SEND + MORE = MONEY .

Une programmation conventionnelle oblige à prendre en compte deux problèmes simultanément : celui de l'addition proprement dite et le fait que deux lettres différentes sont remplacées par deux chiffres différents.

Au contraire, avec le "dif" retardé, ces deux problèmes sont bien séparés : le prédicat "differents" met en place tous les "dif" à l'avance. On n'a plus ensuite qu'à exprimer l'addition et les "dif" se debloquent progressivement au fur et à mesure que l'on avance dans la résolution du problème. Le programme est rendu plus clair, mais aussi plus efficace.

Voici la solution:

```
>jolie-solution;

  9567
 +1085
 -----
 10652
```

```
"RESOLUTION DE SEND+MORE=MONEY"

solution(s.e.n.d.m.o.r.y) ->
   differents(s.e.n.d.m.o.r.y.nil)
   somme(r1,0,0,m,0)
   somme(r2,s,m,o,r1)
   somme(r3,e,o,n,r2)
   somme(r4,n,r,e,r3)
   somme( 0,d,e,y,r4);

somme(x,0,0,x,0) -> / retenue(x);
somme(r,x,y,z,r') ->
   retenue(r)
   chiffre(x)
   chiffre(y)
   val(add(r,add(x,y)),t)
   val(div(t,10),r')
   val(mod(t,10),z);

chiffre(0) ->;
chiffre(1) ->;
chiffre(2) ->;
chiffre(3) ->;
chiffre(4) ->;
chiffre(5) ->;
chiffre(6) ->;
chiffre(7) ->;
chiffre(8) ->;
chiffre(9) ->;

retenue(1) ->;
retenue(0) ->;

differents(nil) -> ;
differents(x.l) -> hors-de(x,l) differents(l);

hors-de(x,nil) -> ;
hors-de(x,a.l) -> dif(x,a) hors-de(x,l);

jolie-solution -> solution(s) jolie-sortie(s) ;

jolie-sortie(s.e.n.d.m.o.r.y) ->
   exm(" ") ex(s) ex(e) ex(n) ex(d) ligne
   exm'"+") ex(m) ex(o) ex(r) ex(e) ligne
   exm("-----") ligne
   ex(m)   ex(o) ex(n) ex(e) ex(y) ligne;
```

5.2 GELER

Le programme qui suit est du à A. Colmerauer.
Il énumère des chemins sans boucle par l'utilisation du prédicat "geler".

Rappelons que "geler" retarde l'effacement du littéral sur lequel il porte tant qu'une certaine variable n'est pas affectée.

Il est utilisé ici pour construire un "bon chemin" qui est un chemin qui ne passe pas deux fois par la meme etape.

Pour cela, on calcule un chemin possible par le prédicat "chemin" que l'on valide au fur et à mesure par le prédicat retardé "bonne liste", ce qui premet de rejeter automatiquement le chemin en cours de construction dès qu'on tente de lui adjoindre une étape qui y figure déjà.

Voici la liste des chemins sans boucle passant par Marseille, Londres et Los Angeles suivie du programme:

```
>bon-chemin(l);

l=Marseille.nil
l=Londres.nil
l=LosAngeles.nil
l=Marseille.Londres.nil
l=Marseille.Londres.LosAngeles.nil
l=Marseille.LosAngeles.nil
l=Marseille.LosAngeles.Londres.nil
l=Londres.Marseille.nil
l=Londres.Marseille.LosAngeles.nil
l=Londres.LosAngeles.nil
l=Londres.LosAngeles.Marseille.nil
l=LosAngeles.Marseille.nil
l=LosAngeles.Marseille.Londres.nil
l=LosAngeles.Londres.nil
l=LosAngeles.Londres.Marseille.nil
>
```

Et voici le programme:

```
"chemins sans boucles"

bon-chemin(l) -> bonne-liste(l) chemin(l);

chemin(x.nil) -> etape(x);
chemin(x.x'.l) -> route(x,x') chemin(x'.l);

route(x,x') -> etape(x) etape(x');

etape(Marseille) ->;
etape(Londres) ->;
etape(LosAngeles) ->;

"listes sans repetitions"

bonne-liste(l) -> geler(l,bonne-liste'(l));

bonne-liste'(nil) ->;
bonne-liste'(x.l) -> hors-de(x,l) bonne-liste(l);

hors-de(x,l) -> geler(l,hors-de'(x,l));

hors-de'(x,nil) ->;
hors-de'(x,x'.l) -> dif(x,x') hors-de(x,l);
```

5.3 LES ARBRES INFINIS

Il s'agit d'un programme extrait de l'article de A.Colmerauer "Prolog and the infinite trees".

Ce programme concerne des automates représentés par des arbres infinis. Il peut etre utilisé de trois facons :
- pour minimiser cet automate,
- pour reconnaitre le langage accepté par cet automate,
- pour construire un automate à partir d'un ensemble de phrases acceptées ou rejetées.

Pour plus de détails on pourra se reporter à l'article précédent.

Ne pas oublier de faire "boucle" pour permettre à l'interprèteur de travailler sur arbres bouclés.

Minimisation du nombre d'états d'un automate

```
automate1(s1) ->
   etat-non-final(s1)
   etat-final(s2)
   etat-final(s3)
   fleche(s1,lettre-a,s2)
   fleche(s1,lettre-b,s3)
   fleche(s2,lettre-a,s3)
   fleche(s2,lettre-b,s3)
   fleche(s3,lettre-a,s2)
   fleche(s3,lettre-b,s2);

>automate1(s) equations(s,x) /;

s=non-final(final(final(final(*1,final(final(*3,*2),final(final
   (*4,*3),*1))),final(final(final(*1,*3),*2),final(final(final(
   *1,*4),*3),*1))),final(final(final(*1,final(*3,*4)),final(*2,
   *3)),*1)),final(final(*1,final(final(*3,*2),final(final(*4,*3
   ),*1))),final(final(final(*1,*3),*2),final(final(final(*1,*4)
   ,*3),*1))))
x=id(2,final(2,2)).id(1,non-final(2,2)).nil
```

Construction d'un automate à partir de phrases qu'il accepte ou refuse.

```
automate3(s)->
   accept(s,lettre-a.lettre-b.nil)
   accept(s,lettre-b.lettre-a.nil)
   refuse(s,lettre-a.nil)
   refuse(s,lettre-b.nil)
   refuse(s,lettre-a.lettre-a.nil)
   refuse(s,lettre-b.lettre-b.nil)
   refuse(s,lettre-a.lettre-a.lettre-b.nil)
   refuse(s,lettre-b.lettre-b.lettre-a.nil);

>solution(x);
x=id(3,non-final(1,2)).id(2,non-final(3,1)).id(1,final(2,3)).nil
```

```
"AUTOMATES D'ETATS FINIS"

"ANALYSE"

accepte(s,nil,vrai) -> etat-final(s);
accepte(s,nil,faux) -> etat-non-final(s);
accepte(s,a.l,v) -> fleche(s,a,s') accepte(s',l,v);

etat-final(final(x,y)) ->;
etat-non-final(non-final(x,y)) ->;

fleche(<e,x,y>,lettre-a,x) ->;
fleche(<e,x,y>,lettre-b,y) ->;

"SORTIE"

equations(p,s) -> sous-arbre(p,x) dedans(x,s,x);

dedans(nil,nil,x) ->;
dedans(<n,p>.x,id(n,q).s,y) ->
   similaire(p,q) domine(p,u) domine(q,v)
   dedans'(u,v,y) dedans(x,s,y);

dedans'(nil,nil,x) ->;
dedans'(p.u,n.v,x) -> dans(<n,p>,x) dedans'(u,v,x);

dans(a,a.x) ->;
dans(a,b.x) -> dans(a,x);

"SOUS-ARBRE"

sous-arbre(p,x) -> union-sous-arbre(nil,p,x);

union-sous-arbre(x,p,x) -> dans(<n,p>,x);
union-sous-arbre(x,p,y) ->
   domine(p,u) nouveau-entier(x,n) unions(<n,p>.x,u,y);

unions(x,nil,x) ->;
unions(x,p.u,z) -> union-sous-arbre(x,p,y) unions(y,u,z);

nouveau-entier(nil,1) ->;
nouveau-entier(<n,p>.x,m) -> suivant(n,m);

suivant(1,2) ->;
suivant(2,3) ->;

domine(<e,x,y>,x.y.nil) ->;

similaire(<e,p,q>,<e,x,y>) ->;

solution(x) ->automate3(s) equations(s,x);

accept(s,x) -> accepte(s,x,vrai);
refuse(s,x) -> accepte(s,x,faux);
```

6 CONTENU DU DISQUE PRO-EX

7 REFERENCES BIBLIOGRAPHIQUES

Parmi la littérature sur Prolog qui devient très abondante, on peut citer les trois livres suivant:

R.KOWALSKI, Logic for problem solving, North Holland, 1979.

W.F. CLOCKSIN, C.S. MELLISH,
Programming in PROLOG, Springer Verlag 1981.

H. COELHO, How to solve it with PROLOG,
Laboratorio nacional de engenharia civil, Lisbonne 1979.

Il faut également citer: 'Logic Programming newsletter' qui est un bulletin édité par 'Universidade nova de Lisboa'.

et bien sur tous les document rédigés par notre equipe, en particulier le manuel d'utilisation et le manuel de réference PROLOG II.