PROLOG II

MANUEL DE REFERENCE
ET
MODELE THEORIQUE

Mars 1982

Alain COLMERAUER

PROLOG II est un système qui a été développé par:

M. van Caneghem, A. Colmerauer, H. Kanoui

Groupe 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

Dix ans se sont écoulés depuis les premiers balbutiements de Prolog (A.Colmerauer, H.Kanoui, R.Pasero et P.Roussel 1973). Force est de constater que ce langage de programmation s'est bien développé et qu'il s'est répandu dans presque tous les pays où l'informatique est connue: Europe (France, Royaume Uni, Belgique, Portugal, Suede, Pologne, Hongrie, Espagne), Canada, USA, Amérique du sud (Venezuela, Argentine), Japon, Australie, Nouvelle Zelande... Il sera amené à jouer un rôle important dans le développement de l'informatique, ainsi que le montrent les rapports consacrés à l'effort gigantesque que la communauté scientifique japonaise se propose de fournir, dans la prochaine décennie, pour développer ses ordinateurs de "5ème genération".

Face à ce développement et compte tenu de l'expérience accumulée, le centre de recherche qui était à l'origine de Prolog se devait de le repenser et de proposer un nouveau système débarassé de toute maladie infantile. Trois personnes se sont mises au travail: Michel van Caneghem, Henry Kanoui et moi-même. Ce fut une tâche longue et difficile et je dois féliciter mes deux partenaires pour une ténacité à toute épreuve, ténacité qui a permis de créer le système Prolog II et de l'implanter sur micro-ordinateur, malgré son gigantisme.

Le présent rapport décrit, explique et justifie le modèle théorique de Prolog II, tout en le spécifiant en tant que langage de programmation. Il fait partie d'une documentation qui comporte en plus le manuel d'utilisation de M.Van Caneghem (1982) et le manuel d'exemples de H.Kanoui (1982).

Quelques informations supplémentaires sur le système Prolog proprement dit. La première implantation conséquente de Prolog consistait en un interpréteur conçu par P.Roussel et programmé en Fortran (G.Battani et H.Meloni 1972). C'est ce qui permit de le transplanter sur les matériels les plus divers et d'assurer une première diffusion importante. Le manuel décrivant le langage Prolog correspondant fut écrit bien plus tard (P.Roussel 1975). Pour Prolog II il s'agit de quelque chose de plus qu'un simple interpréteur: c'est tout un système, portable, extrêmement interactif, et

comprenant l'environnement adéquat pour gérer et mettre au point de gros programmes. Ce système tourne actuellement sur Apple II mais n'a pas été spécifiquement conçu pour cet ordinateur; pour le transplanter il suffit de simuler sa machine virtuelle MicroMegas sur tout autre ordinateur. Il est à noter que nous avons systématiquement priviligié la possibilité de disposer de grands espaces de mémoire (par le biais d'une mémoire virtuelle) au détriment de la vitesse d'exécution. Ceci est fondamental: le temps est extensible, mais pas l'espace!

Venons en au modèle théorique de Prolog II. Au départ Prolog était fondamentalement un démonstrateur de théorèmes reposant sur le principe de résolution de A.Robinson (1965) avec des restrictions draconiennes pour diminuer l'espace de recherche: démonstration linéaire, accès uniquement au premier littéral de chaque clause... Le mérite revient à R.Kowalski et M.van Emden (1976) d'avoir diagnostiqué nos restrictions comme équivalentes à l'utilisation de clauses ayant au plus un littéral positif, dites "de Horn", et d'avoir fourni un premier modèle théorique de ce que calcule exactement Prolog. Nous avons repris ce modèle dans A.Colmerauer (1976) pour systématiser l'utilisation de grammaires formées de schémas de règles et aboutir ainsi à un langage aussi puissant en ce qui concerne le traitement des langues naturelles, que les Systèmes-Q décrits dans A.Colmerauer (1970). Ces Systèmes-Q peuvent d'ailleurs être considérés comme l'ancêtre de Prolog: des règles de réecriture générales plus un premier type d'unification.

Paradoxalement le succès de Prolog dans la communauté informatique est dû à un certain nombre de rajouts, horribles du point de vue théorique, mais indispensables pour arriver à programmer des applications conséquentes. Le fameux, et combien contesté, operateur "/" en est le plus bel exemple... et pourtant il fallait bien disposer d'un moyen d'empêcher l'interpréteur d'explorer de trop nombreuses voies, dans certains cas. Un autre accroc à la théorie fut l'algorithme d'unification utilisé dans la plupart des interpréteurs: on permet d'unifier une variable "x" avec une formule contenant déjà "x", car l'empêcher amènerait à mettre en place des tests qui transformeraient la plupart des programmes, se déroulant en des temps proportionnels à la taille des données, en des programmes se déroulant en des temps quadratiques.

Nous nous proposons de réconcilier la théorie et la pratique. Tout d'abord la nécessité d'utiliser constamment l'opérateur "/" a une raison profonde: l'impossibilité de pouvoir exprimer que deux objets sont et devront toujours rester différents et ainsi de pouvoir mettre en place des conditions plus restrictives qui élimineront l'exploration de nombreuses voies dans l'espace de recherche. Nous avons donc introduit la notion d'inégalité dans Prolog avec toutes les conséquences que cela implique, tant sur le modèle théorique que sur une implantation efficace. Mentionnons que P.Roussel (1973) s'était déjà intéressé aux inégalités dans le cadre de la démonstration automatique et les avaient introduites dans son tout premier interpréteur, interpréteur qui, par contre, ne contenait pas encore l'operateur "/" sous sa forme actuelle.

La solution au problème de l'unification d'une variable contre une formule la contenant déjà, a consisté à étendre le domaine des données manipulées en Prolog: de l'univers d'Herbrand, (les arbres finis) nous sommes passés au domaine des arbres finis et infinis. Tout ceci a conduit à remplacer la notion d'unfication par celle de résolution d'un système d'équations et d'inéquations dans le domaine des arbres finis et infinis. Une grande partie de ce rapport est consacrée à familiariser le lecteur avec ces systèmes.

Les arbres infinis ajoutent à la richesse de Prolog: on dispose enfin d'une structure de données permettant de représenter toutes sortes de graphes avec des circuits. Cependant dans certaines applications il est nécessaire de ne travailler que sur les arbres finis. Le petit programme qui vérifie qu'un arbre partiellement ou complètement inconnu ne deviendra jamais infini s'écrit en quelques lignes (voir paragraphe 6.2). Comme par hasard, il fait intervenir des inégalités sur des arbres infinis. Il fait aussi intervenir une notion de retard d'évaluation qui est rendue possible par l'introduction du concept de "geler(x,p)" qui signifie: retarder "p" tant que la variable "x" est libre. Ce concept n'est pas sans rappeler les notions de coroutines, de processus attaché à un objet et de ces chers "génies" (en anglais "demons") si connus dans certains milieux d'intelligence artificielle.

Enfin le modèle théorique global de Prolog a été complétement refondu. Le but était non seulement d'introduire les arbres infinis mais aussi de dégager l'ensemble minimal de concepts permettant de donner à Prolog une existence

autonome et indépendante de longues considérations sur la logique du 1er ordre et les règles d'inférences. Il a perdu en magie mais y a gagné en clarté. Le lecteur devra donc s'attendre à rencontrer une terminologie nouvelle. Tout repose sur deux définitions équivalentes de ce qu'on entend par ensemble "d'assertions". Cette ambivalence permet de considérer tout programme à la fois comme un système de réecriture et comme un ensemble d'implications. La difficulté à bien programmer en Prolog consiste à acquérir un mécanisme de "double pensée" qui tire le meilleur parti de cette ambivalence. Ce mécanisme étant acquis, l'idéologie passe!

TABLE DES MATIERES

1. ELEMENTS DE BASE

1.1. JEU DE CARACTERES

Dans ce rapport nous utiliserons des règles "hors contexte" pour décrire la syntaxe de toutes les formules intervenant dans Prolog. Les conventions sont:

- Le signe de reécriture 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 representés par des caratères isolés, sauf le caractère d'espacement representé 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.

Voici le jeu de caractères nécessaires à Prolog:

```
<caractere>
    ::= <caractere special>
    ::= <lettre>
    ::= <chiffre>

<caractere special>
    ::= +
    ::= -
    ::= '
    ::= .
    ::= ,
    ::= ;
    ::= /
    ::= =
    ::= #
    ::= "
    ::= (
    ::= )
    ::= <
    ::= >
    ::= {
    ::= }
    ::= espace
    ::= <tout autre caractere diponible>

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

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

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

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

Les signes "+" et "-" interviennent dans l'écriture des nombres réels. Le caractère "-" intervient aussi comme trait d'union. Le caractère "apostrophe" est utilisé en tant que prime. Les caractères "point", "virgule" et "point-virgule" servent de séparateurs. Les caractères "=" et "#" corespondent à l'égalité et a l'inégalité. Les caractères "(", ")", "<", ">", "{", "}" servent a parenthéser. Le caractère "double-guillemet" encadre les chaines. Le caractère "/" marque une operation de coupure. Le couple de caractères "->" est utilisé comme flèche. Afin de conserver le maximum de liberté typographique, aucun rôle special n'est assigné aux lettres majuscules par rapport aux lettres minuscules.

Dés maintenant nous définissons plusieurs suites utiles de caractères:

```
<suite de caracteres>
   ::= <vide>
   ::= <caractere> <suite de caracteres>

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

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

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

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

<vide>
   ::=
```

1.2. CONSTANTES

Les données les plus simples sont des constantes. Elles peuvent être de quatre types:

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

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

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

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

<reel>
   ::= <signe> <entier> . <suite de chiffres> <exposant>

<signe>
   ::= +
   ::= -

<exposant>
   ::= <vide>
   ::= e <entier>
   ::= e <signe> <entier>
```

Tous les caractères peuvent apparaître a l'intérieur d'une chaîne mais toute occurence du caractère "double-guillemet" doit y être doublée.

Voici des exemples d'identificateurs corrects:

```
pomme pomme' pomme12 en-bas calcul22-des-v2''
```

des exemples d'identificateurs incorrects:

```
x 1er-lapin y-en-a l'herbe
```

des exemples de chaînes correctes:

```
"attention" "hum hum" "(<->)" "" "a""b" """"
```

des exemples de chaînes incorrectes:

```
""" "a"b"c"
```

des exemples d'entiers corrects:

```
122 0 0023 00
```

des exemples d'entiers incorrects:

+122 -122 30.0 40.

des exemples de nombres réels corrects:

+3.14 +0.314e1 +314.e-2 -50. -5.e+1

des exemples de nombres réels incorrects:

3.14 +314e-2

A chaque constante est attachée une valeur et l'on considerera que deux constantes sont égales si et seulement si leurs valeurs sont égales.

- La valeur d'un identificateur est le couple "(c,i)" ou "i" représente la suite de caractères qui le constitue et "c" une une information propre au contexte dans laquelle l'occurence de l'identificateur a été introduite. Cette information peut varier d'une occurence à l'autre et permet donc d'utiliser, sans risque d'interference, le même mnemonique dans des parties différentes d'un même programme. Cette possibilité est fondamentale dans l'écriture de gros programmes. Pour plus de précisions nous renvoyons le lecteur à la partie environnement de la machine Prolog (paragraphe 6.3).

- La valeur d'une chaine est la suite des caractères qui la constituent.

- La valeur d'un entier est l'entier non négatif qu'il représente. De ce fait les écritures "123" et "0123" sont équivalentes.

- La valeur d'un réel est le réel qu'il représente d'une façon approchée; avec tous les problèmes que cela pose. Bien entendu le point sert a marquer la virgule décimale et la lettre "e" (pour exposant) indique par quelle puissance décimale il faut multiplier le nombre. Nous considererons que les entiers et les réels forment des ensembles disjoints.

Il faut remarquer qu'entre autres, deux constantes de types différents seront toujous considérées comme différentes.

1.3. VARIABLES

Les derniers élements de base dont nous aurons besoin sont les variables. Elles servent aussi bien a désigner des constantes que des entités plus complexes. En voici la syntaxe:

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

Cette syntaxe ressemble à celle des identificateurs mais il faut remarquer que le debut d'une variable est toujours constitué

d'exactement une lettre alors que le début d'un identificateur est toujours constitué d'au moins deux lettres. Voici quelques exemples de bonnes variables:

```
x x' x'' x12 p-rix p-phrase y33' y1' y-en-a' x-toto
```

et quelques exemples de mauvaises:

```
ph xx prix toto 1er-x
```

2. ARBRES FINIS ET INFINIS

2.1. NOTION INTUITIVE

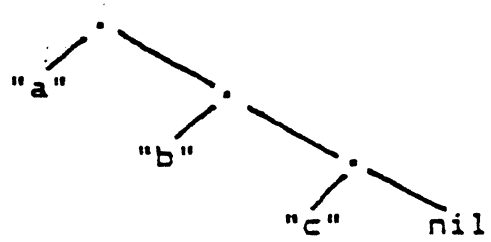
Toutes les données manipulées en Prolog sont des arbres éventuellement infinis dont nous allons tout d'abord donner une description informelle. Ces arbres sont formés de noeuds étiquetés:

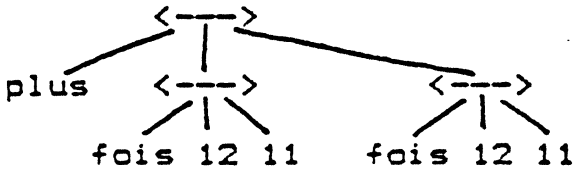
- soit par une constante et, dans ce cas, ils n'ont aucun fils,

- soit par le caractère "point" et, dans ce cas il ont deux fils,

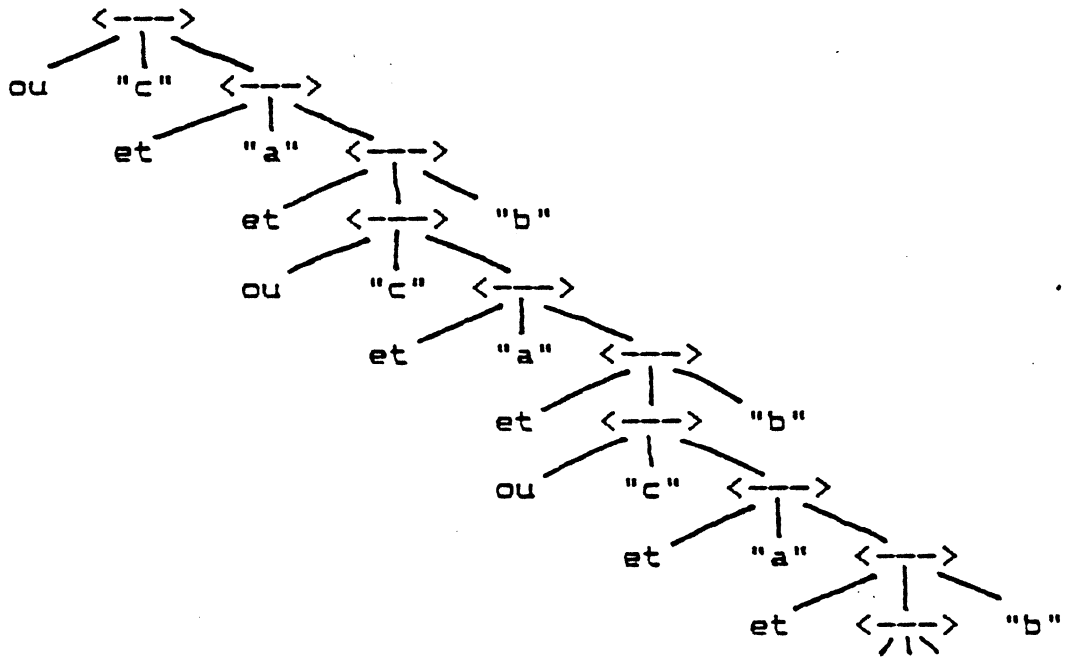
- soit par "<>" ou "<->" ou "<-->" ou "<--->" ou ... et, dans ce cas, le nombre de traits d'union correspond au nombre de leurs fils.

Voici deux exemples d'arbres finis:

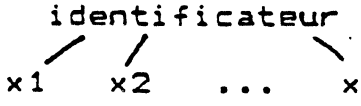




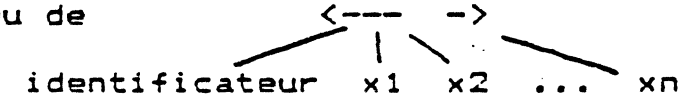
et voici un exemple d'arbre infini:



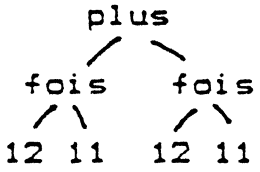
Afin d'alléger le dessin des arbres on dessinera souvent

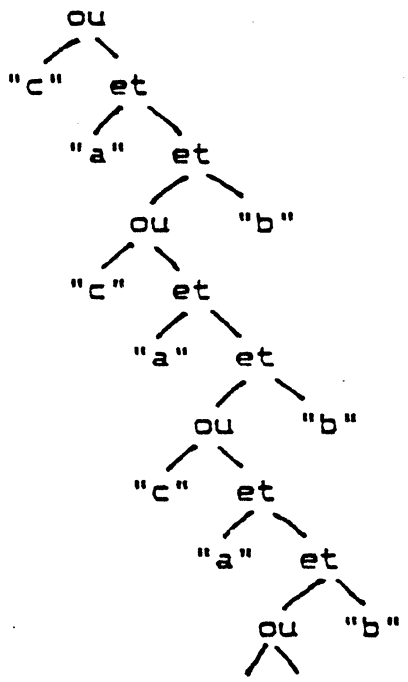


au lieu de



Bien entendu cette simplification présuppose que "n" ne soit pas nul. Les deux derniers arbres peuvent donc être representés sous la forme classique de:

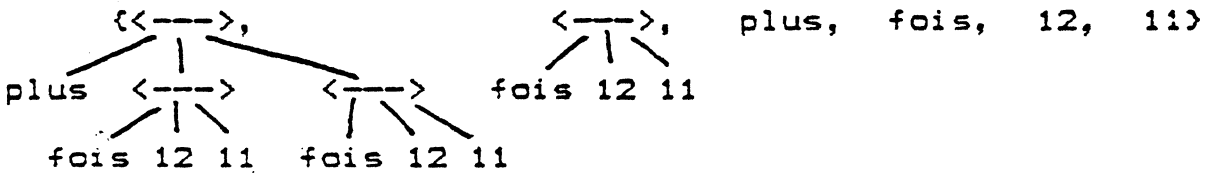


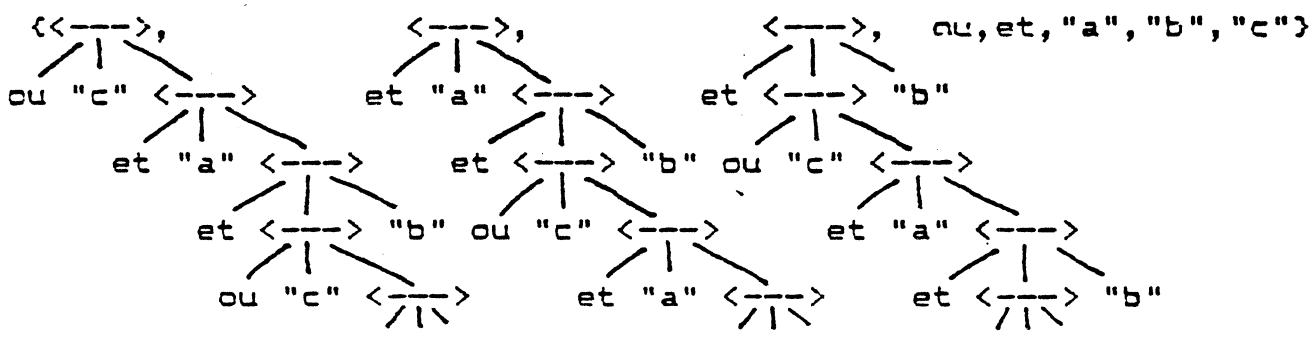


La notion d'arbre infini est suffisamment inhabituelle pour que nous nous étendions un peu dessus. Intuitivement un arbre est infini s'il possède une branche infinie. Nous nous intéresserons plus spécialement à la fraction des arbres infinis qui, ensemble avec les arbres finis, forme les arbres dits "rationnels".

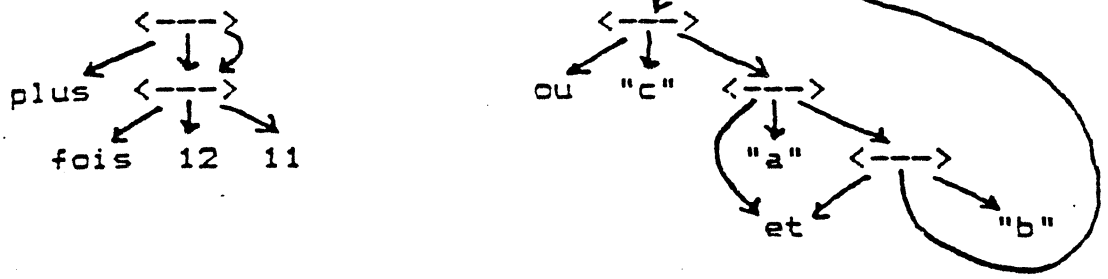
DEFINITION: Un arbre est "rationnel" si l'ensemble de ses sous-arbres est fini.

De même que nous nous sommes contentés jusqu'à maintenant d'une notion intuitive d'arbre, nous nous contenterons de la notion intuitive de sous-arbre. Si nous reprenons les deux derniers exemples d'arbres l'ensemble de leurs sous-arbres est respectivement:

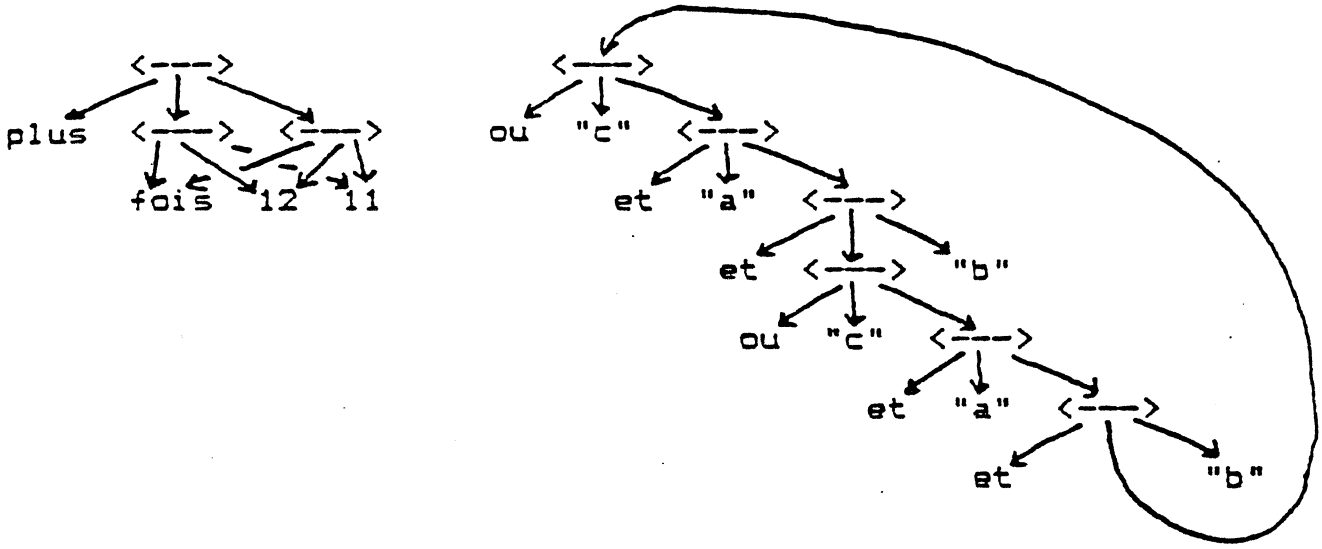




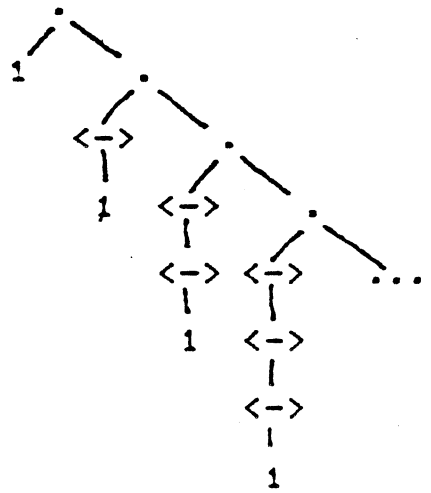
Ces ensembles étant finis, il s'agit donc d'arbres rationels. Le fait qu'un arbre rationnel contienne un ensemble fini de sous-arbres donne un moyen immédiat de le représenter par un diagramme fini: il suffit de fusionner tous les noeuds d'où partent les mêmes sous-arbres. Sur nos deux exemples on obtient:



Si on ne fait pas toutes les fusions on peut aussi obtenir:



Il faut donc se méfier du fait que des diagrammes différents puissent représenter le même arbre. Enfin, il est intéressant d'exhiber un arbre non rationnel, c'est à dire, un arbre qui a une infinité de sous-arbres. Voici le plus simple à notre connaissance:



2.2. LES DEUX PREMIERES PROPRIETES CARACTERISTIQUES

Il est maintenant temps de donner une définition plus formelle des arbres. Il est facile d'assimiler les arbres finis à un certain type de formules. Les difficultés surgissent lorsqu'un arbre est infini: la suite de caractères qui constitue la formule qui le representerait risque de devenir infinie en son milieu et, en mathématiques, on ne connaît que les suites infinies à une extrémité! Pour éviter ce problème on est amené a définir les arbres d'une facon relativement complexe, en faisant intervenir des suites d'entiers pour designer les noeuds. Une telle definition peut être trouvée dans le chapitre 5 de la these de G. Huet (1976). Ce type de définition, qui a au moins l'avantage d'être une vraie définition formelle, nous semble, à la limite, plus obscure que la notion intuitive du concept d'arbre. Pour notre part, tout ce que nous demandons à l'ensemble "R" des arbres, est d'avoir trois propriétés que nous allons expliciter, les deux premières immédiatement et la dernière un peu plus tard. Nous qualifierons ces propriétés de "caractéristiques" car elles semblent suffisamment contraignantes pour que tous les ensembles "R" auxquels elles s'appliquent soient isomorphes.

PROPRIETE CARACTERISTIQUE 1 (composition): on a les trois sous-propriétés:

(1) toute valeur "k" de constante (précédemment définie) est élément de "R";

(2) si "r1" et "r2" sont éléments de "R", alors l'être noté "(r1.r2)" l'est aussi;

(3) si "r1,r2,...,rn" est une suite éventuellement vide d'éléments de "R", alors l'etre note "<r1,r2,...,rn>" est élément de "R".

PROPRIETE CARACTERISTIQUE 2 (décomposition unique): si "r" est un élément de "R", alors une et une seule des trois propositions qui suivent est vraie:

(1) il existe une et un seule valeur de constante "k" telle que "r = k" et on dit que la suite des "fils" de "r" est vide;

(2) il existe un et un seul couple "r1,r2" d'éléments de "R", appelé suite de "fils" de "r", tel que "r = (r1.r2)";

(3) il existe une et une seule suite (eventuellement vide) "r1,...,rn" d'elements de "R", appelée suite de "fils" de "r", telle que "r = <r1,r2,...,rn>".

La première propriété donne le moyen de construire un arbre à partir d'autres arbres. Elle nous conduira à introduire la notion de "terme" qui est une formule représentant une telle construction. La deuxième propriété a deux conséquences importantes:

- toute égalité de la forme "(r1.r2) = (s1.s2)" entraîne les égalités "r1 = s1" et "r2 = s2"; de même toute égalité de la forme "<r1,...,rn> = <s1,...,sn>" entraîne les égalités "r1 = s1" ... et "rn = sn";

- tout arbre réduit à une valeur "k" de constante est différent d'arbres de la forme "(r1.r2)" ou "<r1,..,rm>"; tout arbre de la forme "(r1.r2)" est différent d'un arbre de la forme "<r1,...,rn>"; de plus si "n" et "m" sont différents, tous les arbres de la forme "<r1,...,rm>" sont différents des arbres de la forme "<s1,...,sn>".

Il faut aussi noter que la propriété 2 introduit la notion de "fils" d'une facon formelle. Il est donc possible maintenant de définir exactement ce qu'est un arbre infini et ce qu'est l'ensemble des sous-arbres d'un arbre donné.

DEFINITION: un arbre "r0" est infini si et seulement si il existe une suite infinie d'arbres "r0,r1,r2,..." telle que chaque "ri+1" soit un fils de "ri".

DEFINITION: l'ensemble des sous-arbres d'un arbre "r" est le plus petit ensemble d'arbres qui contient "r" et qui contient les fils des arbres qu'il contient.

La définition d'un arbre rationnel reste toujours la même: "qui a un ensemble fini de sous-arbres".

2.3. TERMES ET SYSTEMES D'EQUATIONS ET INEQUATIONS.

Pour représenter les arbres nous utiliserons des formules appelées "termes". Nous introduirons tout d'abord la notion de "terme strict":

```
<terme strict>
   ::= <variable>
   ::= <constante>
   ::= ( <terme strict> . <terme strict> )
   ::= < >
   ::= < <terme strict> >
   ::= < <terme strict> , <terme strict> >
   ::= < <terma strice> , <terme strict> , <terme strict> >
   ..................................................
```

Les termes stricts sont les vrais termes. Cependant pour des raisons de commodité on étend la syntaxe des termes stricts (sans en altérer le sens) en permettant:

- d'ajouter et d'enlever des parenthèses mais en convenant que "t1.t2.---.tn" représente "(t1.(t2.(---.tn)-)));

- d'écrire "id(t1,t2,...,tn)" au lieu de "<id,t1,t2,...,tn>" a condition que "id" soit un identificateur et que "n" soit différent de 0.

Ceci conduit à une notion plus générale de "terme" définie par:

```
<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> )
   ................................................
```

Voici un exemple de terme:

```
aiment(Pierre.Paul.Jean.nil,pere-de(x))
```

et le terme strict correspondant est:

```
<aiment,(Pierre.(Paul.(Jean.nil))),<pere-de,x>>.
```

Pour transformer un terme en un arbre il faut affecter ses variables par des arbres, d'où la notion "d'affectation sylvestre":

DEFINITION: Nous appellerons "affectation sylvestre" tout ensemble "X" de la forme:
X = {x1:=r1, x2:=r2, ...}
ou les "xi" sont des variables distinctes et les "ri" des arbres.

L'arbre associé à un terme "t" est noté "t(X)" et est defini comme suit:

DEFINITION: Si "t" est un terme strict faisant intervenir un sous-ensemble de l'ensemble des variables de l'affectation sylvestre "X = {x1:=r1,x2:=r2,...}" alors l'expression "t(X)" désignera l'arbre obtenu en remplaçant les variables "xi" par les arbres correspondants "ri". Plus précisement:
- "t(X)" = "ri" si "t" = "xi";
- "t(X)" = valeur de "k" si "t" est la constante "k";
- "t(X)" = "(t1(X).t2(X))" si "t" = "(t1.t2)";
- "t(X)" = "<t1(X),...,tn(X)>" si "t" = "<t1,...,tn>".

Si "t" ne contient pas de variable, "t(X)" se notera aussi "'t'".

Si "t1" et "t2" sont des termes alors les formules "t1=t2" et "t1#t2" sont respectivement une "équation" et une "inéquation". Un ensemble de telles formules est un "système" (d'equations et d'inequations). A moins d'une précision explicite contraire, nous utiliserons le mot "système" pour désigner un système fini. Nous pouvons donc définir syntaxiquement un système par:

```
<système>
   ::= { }
   ::= { <equations et inequations> }
```

```
<equations et inequations>
   ::= <equation>
   ::= <inequation>
   ::= <equation> , <equations et inequations>
   ::= <inequation> , <equations et inequations>

<equation>
   ::= <terme> = <terme>

<inequation>
   ::= <terme> # <terme>
```

Mentionnons dès maintenant un type très particulier de système qui nous a causé beaucoup d'ennuis: les "circuits de variables":

DEFINITION: un "circuit de variable" est un système non vide de la forme:
{x1=x2, x2=x3, ..., xn=x1},
les "xi" étant des variables distinctes et "n" pouvant être egal a 1.

Qui pense système (éventuellement infini), pense solutions de système. Cette notion de solution doit etre définie avec précision et de telle facon que, paradoxalement, l'ensemble des variables figurant dans une solution ne dépende pas de celui figurant dans le système. Ceci est fondamental si l'on veut pouvoir raisonner sur les solutions de l'union de plusieurs systèmes.

DEFINITION: l'affectation sylvestre "X" est dite "solution sylvestre" du système eventuellement infini:
{p1=q1, p2=q2, ...} u {s1#t1, s2#t2, ...}
si "X" est sous-ensemble d'une affectation sylvestre "Y" qui est telle que les "pi(Y)" soient respectivement égaux aux "qi(Y)" et que les "si(Y)" soient respectivement différents des "ti(Y)".

On remarquera que tout système eventuellement infini, qui admet au moins une solution sylvestre, admet aussi la solution sylvestre vide "{}".

2.4. LA TROISIEME PROPRIETE CARACTERISTIQUE

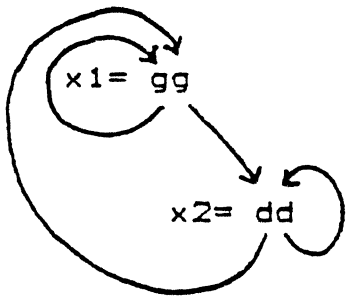
Nous disposons maintenant de tous les éléments pour pouvoir énoncer la troisième propriété caractéristique de l'ensemble "R" des arbres.

PROPRIETE CARACTERISTIQUE 3 (solution unique): Soit "S" un système d'équations éventuellement infini de la forme:
S = {x1=t1, x2=t2, ...},
les "xi" étant des variables distinctes et les "ti" étant des termes non réduits à des variables et ne contenant pas d'autres variables que les "xi". Il existe alors une et une seule suite d'arbres "r1,r2,..." telle que l'affectaion sylvestre:
{x1:=r1, x2:=r2, ...}
soit solution de "S".

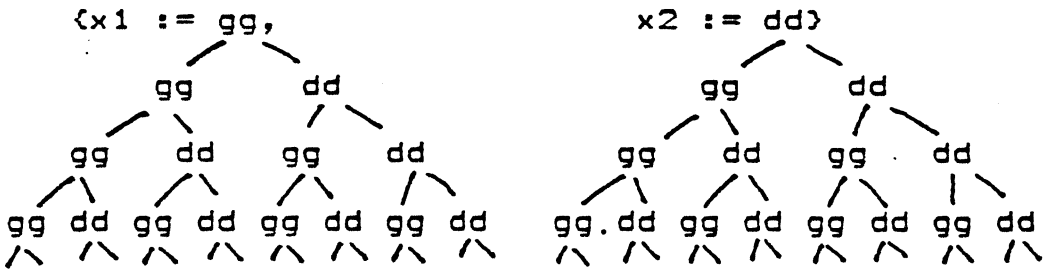
Soit par exemple le système:

{x1 = gg(x1,x2), x2 = dd(x1,x2)}

On passe immediatement au diagramme:



d'où la solution unique:



Soit "{r1,r2,...}" l'ensemble des sous-arbres d'un ensemble donné d'arbres. Si l'on associe une variable "xi" à chaque arbre "ri" et que l'on remarque que les fils de chaque "ri" sont aussi des "ri", on peut alors associer une équation à chaque "ri" et obtenir ainsi un systeme d'equations dont "{x1:=r1,x2:=r2,...}" est une solution. On en conclut donc:

PROPRIETE DE SYSTEME ASSOCIE: Quel que soit l'ensemble de sous-arbres "{r1,r2,...}" d'un ensemble donne d'arbres, il existe un système eventuellement infini "S":

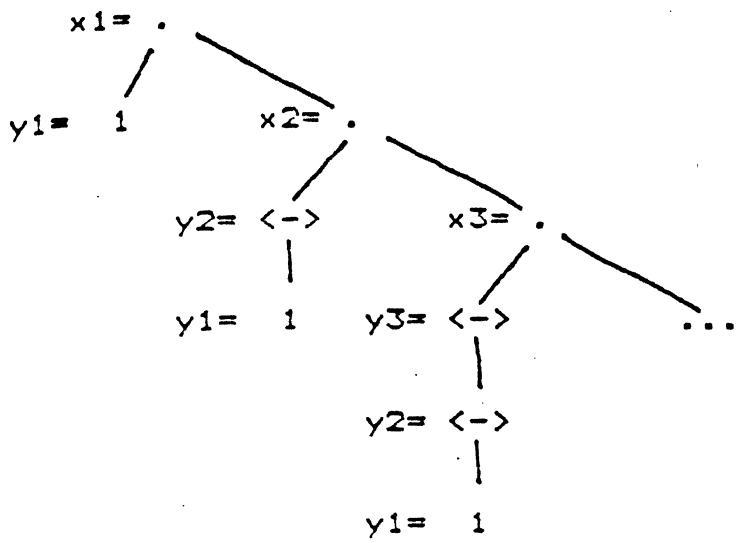
S = {x1=t1, x2:=t2, ...},

admettant

{x1:=r1, x2:=r2, ...}

comme solution sylvestre. Les "xi" sont des variables distinctes et les "ti" ne sont pas des variables et ne contenant pas d'autres variables que les "xi".

Reprenons l'exemple d'arbre non rationnel de la fin du paragraphe 2.1, et associons une variable à chacun de ses sous-arbres:



Cet arbre est donc solution en "x1" de:

{x1=(y1.x2),y1=1, x2=(y2.x3),y2=<y1>, x3=(y3.x4),y3=<y2>, ...}

3. RESOLUTION DE SYSTEMES D'EQUATIONS ET D'INEQUATIONS

3.1. SYSTEMES SOUS FORME REDUITE

Le problème qui nous intéresse ici est de déterminer si un système admet ou n'admet pas de solution sylvestre. Dans le premier cas on dira qu'il est "soluble" et dans le deuxième qu'il est "insoluble". Nous introduirons tout d'abord un type particulier de systèmes, les systèmes sous forme "réduite":

DEFINITION: un système "S" dont "E" représente l'ensemble des équations et "I" l'ensemble des inéquations, est sous forme "réduite" s'il est de l'une des deux formes:

(1) "I" est vide et "E", qui est éventuellement vide, ne contient pas de circuit de variables et est de la forme:
{x1=t1, ..., xn=tn},
les "xi" étant des variables distinctes, les "ti" étant des termes quelconques;

(2) "I" n'est pas vide et chacune de ses inéquations est de la forme:
<y1,...,ym>#<t1,...,tm>, distincts des "xi"
l'entier "m" n'étant pas nul, les "yj" étant des variables, les "tj" étant des termes quelconques et, pour chaque inéquation le système associé:
E u {y1=t1,...,ym=tm}
est sous forme réduite.

Voici la première propriété fondamentale des systèmes sous forme réduite:

PROPRIETE DE RESOLUBILITE: tout système sous forme réduite est soluble.

La démonstration est donnée en annexe. Donnons quelques exemples de systèmes sous forme réduite:

{x=1, y=2},
{x=<x,u>, y=<x,v>, <u,v>#<x,y>, <v,u>#<u,1>},

et quelques exemples de systèmes qui ne sont pas sous forme réduite:

{u=1, v=x, <x,y,z>#<y,z,x>},
{x=1, <u,v>#<x,u>, <x>#<1>}.

3.2. SYSTEMES EQUIVALENTS

Les systèmes sous forme réduite ont une deuxième propriété fondamentale liée à la notion de systèmes équivalents, notion que nous développerons tout d'abord.

DEFINITION: deux systèmes sont "équivalents" s'il ont le même ensemble de solutions sylvestres.

Si l'on se reporte à la définition des solutions sylvestres, on remarque que pour montrer que deux systèmes sont équivalents il suffit de ne considérer que les solutions qui font intervenir l'ensemble des variables figurant dans l'union des deux systemes. Citons 4 cas d'équivalence utiles et intéressants:

EQUIVALENCE 1: quels que soient les termes "ti" et "si", les systèmes:
"{(s1.s2)=(t1.t2)}" et "{s1=t1, s2=t2},
d'une part, et les systèmes:
"{<s1,...,sn>=<t1,...,tn}" et "{s1=t1, ..., sn=tn}",
d'autre part, sont équivalents.

EQUIVALENCE 2: Soient "S" et "T" deux systèmes sous forme réduite et de la forme:
"S = {x1=t1,...,xn=tn}" et "T = {x1=t1',...,xn=tn'}".
Si toute solution sylvestre de "S" est solution sylvestre de "T" alors "S" et "T" sont équivalents.

EQUIVALENCE 3: Soit "S" un système et soient "s" et "t" deux termes. Si le système "S u {s=t}" est insoluble, alors les systèmes:
"S u {s#t}" et "S"
sont équivalents.

EQUIVALENCE 4: Soit "S" un système, soient "s" et "t" des termes et soient "s1,...,sn" et "t1,...,tn" des suites éventuellement vides de termes. Si les systèmes:
"E u {s=t}" et "E u {s1=t1,...,sn=tn}
sont équivalents alors les systèmes:
"E u {s#t}" et "E u {<s1,...,sn>#<t1,...,tn>}"
le sont aussi.

L'équivalence 1 est une conséquence directe de la 2ème propriété caractéristique (décomposition unique). La démonstration de l'équivalence 2 est donnée en annexe. L'équivalence 3 est triviale et l'équivalence 4 l'est aussi si l'on remarque qu'aucune solution du système "{s1=t1,...,sn=tn}" qui fait intervenir toutes ses variables, n'est solution de "{<s1,...,sn>#<t1,...,tn>}" et vice versa.

Apres cette diversion nous pouvons maintenant donner la 2eme propriété fondamentale des systèmes sous forme réduite:

PROPRIETE DE FORME NORMALE: tout système soluble est équivalent à un système sous forme réduite.

Pour démontrer ceci il suffit d'exhiber des algorithmes formels, dit "de réduction", qui transforment tout système fini en un systeme équivalent, qui est, soit réduit, soit trivialement insoluble. Par la même occasion on résoud le problème initial: savoir si un système donné "S" est ou n'est pas soluble. En effet il suffit d'appliquer l'algorithme de réduction sur "S" et, suivant que le système resultant est, ou n'est pas, sous forme réduite, "S" est, ou n'est pas, soluble.

3.3. REDUCTION D'EQUATIONS

Le premier algorithme que nous présentons est un algorithme de base permettant de transformer un système de facon à reduire le sous-système constitué par l'ensemble de ses équations. Le but étant d'etablir une propriété d'existence nous avons choisi un algorithme simple et pédagogique. En aucun cas cet algorithme ne peut être considéré comme l'algorithme de base efficace que l'on aimerait programmer tel quel! Voici en quoi il consiste:

ALGORITHME DE REDUCTION DE BASE:

Soit un système "S". L'algorithme consiste à appliquer sur S, tant que cela est possible, des transformations prises parmi T1,T2,T3,T4 et T5:

T1 (absorption): supprimer toute équation de la forme "x=x" ou "k1=k2", "x" étant une variable et "k1" et "k2" des constantes de même valeur.

T2 (élimination de variable): si l'équation "x=y" appartient au système, "x" et "y" étant des variables distinctes, et si "x" a d'autres occurences, remplacer ces autres occurences par "y".

T3 (antéposition de variable): remplacer l'équation "t=x" par l'équation "x=t", quand "x" est une variable et que "t" ne l'est pas.

T4 (mise en conflit): remplacer le sous-systeme "{x=t1,x=t2}" par le sous-système "{x=t1,t1=t2}", à condition que, contrairement à "t1" et "t2", "x" soit une variable et que la taille du terme "t1" soit inférieure ou égale à celle du terme "t2". Par taille d'un terme on entend le nombre cumulé d'occurence de variables, de constantes, de signes "." et de signes "<" dans ce terme.

T5 (explosion): remplacer le sous-système "{(s1.s2)=(t1.t2)}" par "{s1=t1,s2=t2}". De même remplacer "{<s1,..,sn>=<t1,...,tn>}" par "{s1=t1,...,sn=tn}".

D'après la propriété qui suit (demonstration en annexe), cet algorithme se termine toujours.

PROPRIETE D'ARRET: si "S1" est un système alors il n'existe pas de suite infinie "S1,S2,S3,..." de systèmes, tel que chaque "Si+1" soit obtenu en appliquant sur "Si" la transformation T1, T2, T3, T4 ou T5.

D'autre part il s'agit bien d'un algorithme de réduction d'équations et ceci pour deux raisons:

(1) Chaque transformation produit un système equivalent à celui sur lequel elle s'applique. Ceci est dû, d'une part, aux propriétés de l'égalité dans le cas des transformations T1,T2,T3,T4 et, d'autre part, à l'équivalence 1 dans le cas de T5.

(2) Quand plus aucune transformation ne peut s'appliquer, le sous-système final des équations, s'il n'est pas sous forme

réduite, contient forcément une équation de l'une des 5 formes:
(1) "k1=k2", "k1" et "k2" constantes de valeurs differentes,
(2) "k=(t1.t2)" ou "(t1.t2)=k", "k" constante,
(3) "k=<t1,...,tn>" ou "<t1,...,tn>=k", "k" constante,
(4) "(s1.s2)=<t1,...,tn> ou <t1,....,tn>=(s1.s2),
(5) "<s1,...,sm>=<t1,...,tn>", "m" different de "n".
Il est donc trivialement insoluble d'apres la 2ème propriété caractéristique des arbres (décomposition unique).

Voici trois exemples d'utilisation de cet algorithme de réduction.

EXEMPLE 1

{<<x,jean>,y>=<y,<paul,jean>>} est un système soluble car:

{<<x,jean>,y>=<y,<paul,jean>>},	par explosion,
{<x,jean>=y, y=<paul,jean>},	par antep. de variable,
{y=<x,jean>, y=<paul,jean>},	par mise en conflit,
{y=<x,jean>, <x,jean>=<paul,jean>},	par explosion,
{y=<x,jean>, x=paul, jean=jean},	par absorption,
{y=<x,jean>, x=paul},	qui est reduit.

EXEMPLE 2

{x=<<x>>, <<bob>>=x} est un système insoluble car:

{x=<<x>>, <<bob>>=x},	par antéposition de variable,
{x=<<x>>, x=<<bob>>},	par mise en conflit,
{x=<<x>>, <<x>>=<<bob>>},	par explosion,
{x=<<x>>, <x>=<bob>},	par explosion,
{x=<<x>>, x=bob},	par mise en conflit,
{bob=<<x>>, x=bob},	qui est trivialement insoluble.

EXEMPLE 3

{x=y, x=<<x>>, y=<<<y>>>} est soluble car:

{x=y, x=<<x>>, y=<<<y>>>},	par élimination de variable,
{x=y, y=<<y>>, y=<<<y>>>},	par mise en conflit,
{x=y, y=<<y>>, <<y>>=<<<y>>>},	par explosion,
{x=y, y=<<y>>, <y>=<<y>>},	par explosion,
{x=y, y=<<y>>, y=<y>},	par mise en conflit,
{x=y, <y>=<<y>>, y=<y>},	par explosion,
{x=y, y=<y>},	qui est sous forme réduite.

Dans ce dernier exemple, il est à remarquer que, si l'on ne respecte pas le test concernant la taille des termes dans la deuxième transformation de mise en conflit, on entre dans une boucle:

{x=y, y=<<y>>, y=<y>},	par mise en conflit sans test,
{x=y, y=<<y>>, <<y>>=<y>},	par explosion,
{x=y, y=<<y>>, <y>=y},	par anteposition de variable,
{x=y, y=<<y>>, y=<y>},	meme chose que 3 lignes avant.

Si l'on dispose d'un système d'equations "S", contenant un sous-système "E" déjà réduit, il peut être intéressant, dans la mesure ou "S" est soluble, de calculer une forme réduite de "S" qui contienne toujours le même sous-système réduit "E". Ceci est possible en utilisant le fait suivant:

PROPRIETE DE CONSERVATION: Soit un système "S1", soluble, sans inequations et de la forme "S1 = E1uF1" ou "E1" est sous forme réduite:

E1 = {x1=t1,...,xn=tn},

avec la restriction que chaque "xi", dont le "ti" correspondant est une variable, n'a qu'une seule occurence dans tout le systeme "E1uF1". Si l'on applique l'algorithme de réduction de base sur "S1" on obtient alors un système reduit "S2" de la forme "S2 = E2uF2", avec "E2" et "F2" disjoints et "E2" de la forme:

E2 = {x1=t1',...,xn=tn'}.

De plus le système "E1uF2" est sous forme réduite et est équivalent au système initial "E1uF1".

La démonstration (en annexe) de cette propriété fait intervenir l'équivalence 2 entre systèmes. Considérons par exemple le système déjà partiellement réduit:

{x=<x,u>, y=<y,v>, v=u} u {<x,z>=<y,5>}.

On remarque que la variable "v" n'a bien qu'une seule occurence dans tout le système. En appliquant l'algorithme de réduction de base on obtient successivement:

{x=<x,u>, y=<y,v>, v=u, <x,z>=<y,5>},
{x=<x,u>, y=<y,v>, v=u, x=y, z=5},
{x=<x,u>, y=<y,u>, v=u, x=y, z=5},
{y=<y,u>, v=u, x=y, z=5},

et donc le système initial admet comme forme réduite, à la fois le système:

{x=y, y=<y,u>, v=u} u {z=5},

et le système qui conserve le sous-systeme réduit:

{x=<x,u>, y=<y,v>, v=u} u {z=5}.

3.4. REDUCTION D'EQUATIONS ET D'INEQUATIONS

Pour réduire un systeme contenant des inéquations il faut disposer de transformations permettant de les modifier et d'obtenir toujours un système equivalent. Les équivalences 3 et 4 nous en fournissent deux. En faisant intervenir en plus la propriété de conservation de l'algorithme de base, on construit l'algorithme general:

ALGORITHME GENERAL DE REDUCTION:

Soit "S" le système à reduire. On applique tout d'abord l'algorithme de reduction de base sur "S" et l'on obtient un

système forme d'un ensemble "E" d'équations et d'un ensemble "I" d'inéquations. Si "E" n'est pas sous forme réduite "S" est insoluble, sinon on pose:

E = {x1=t1, ..., xn=tn}.

On considère maintenant chaque inéquation "s#t" de "I" et on applique l'algorithme de réduction restreint sur le système "Eu{s=t}". A chaque fois deux cas se présentent:

(1) le système "Eu{s=t}" n'est pas soluble; on élimine alors l'inéquation "s#t";

(2) le système "Eu{s=t}" est soluble et sa forme réduite est:

{x1=t1', ..., xn=tn',y1=s1,...,ym=sm},

avec éventuellement "m" nul; on conserve alors l'inéquation:

<y1,...,ym>#<s1,...,sm>.

On désigne par "J" l'ensemble des inéquations conservées. Si "J" contient l'inéquation "<>#<>" alors le système initial "S" n'est pas soluble, sinon sa forme réduite est "EuJ" et, bien entendu, "S" est soluble.

Voici quelques exemples pour illustrer tout cela:

EXEMPLE 1

Soit à réduire le systeme:

{<u,v>=<<x>,w>, (v.w)=(w.v), u#1, <x,y,z>#<y,z,x>}.

On applique l'algorithme de base:

{<u,v>=<<x>,w>, (v.w)=(w.v), u#1, <x,y,z>#<y,z,x>},
{u=<x>, v=w, (v.w)=(w.v), u#1, <x,y,z>#<y,z,x>},
{u=<x>, v=w, w=v, u#1, <x,y,z>#<y,z,x>},
{u=<x>, v=w, w=w, u#1, <x,y,z>#<y,z,x>},
{u=<x>, v=w, u#1, <x,y,z>#<y,z,x>}.

Le système initial est donc équivalent a:

{u=<x>, v=w} u {u#1, <x,y,z>#<y,z,x>}.

Traitons la première inéquation. En la remplacant par une équation on obtient:

{u=<x>, v=w, u=1},
{1=<x>, v=w, u=1}.

Le dernier système étant insoluble, on peut supprimer l'inéquation "u#1" et donc il reste à réduire:

{u=<x>, v=w} u {<x,y,z>#<y,z,x>}.

Le traitement de l'inéquation donne:

{u=<x>, v=w, <x,y,z>=<y,z,x>},
{u=<x>, v=w, x=y, y=z, z=x},
{u=<y>, v=w, x=y, y=z, z=y},

{u=<z>, v=w, x=z, y=z, z=z},
{u=<z>, v=w, x=z, y=z}.

Le système initial est donc équivalent au système sous forme réduite:

{u=<x>, v=w, <x,y>#<z,z>}.

Il est donc soluble.

EXEMPLE 2

Soit le système:

{x=<x>, y=<<y>>, x#y}.

L'algorithme de base ne modifie pas ce système. On transforme donc l'inéquation en équation et l'on obtient:

{x=<x>, y=<<y>>, x=y},
{x=y, y=<y>, y=<<y>>},
{x=y, y=<y>, <y>=<<y>>},
{x=y, y=<y>}.

Le système initial avec inéquation est donc équivalent au système:

{x=<x>, y=<<y>>, <>#<>}

et donc n'est pas soluble puisqu'il est impossible de satisfaire l'inéquation "<>#<>".

EXEMPLE 3

Soit le système:

{x=<z,x>, y=<y,z>, x#y}.

Le traitement de l'inéquation donne:

{x=<z,x>, y=<y,z>, x=y},
{x=y, y=<z,y>, y=<y,z>},
{x=y, y=<z,y>, <z,y>=<y,z>},
{x=y, y=<z,y>, z=y, y=z},
{x=z, z=<z,z>, z=z, y=z},
{x=z, y=z, z=<z,z>}.

Le système initial avec l'inéquation est donc équivalent au système sous forme réduite:

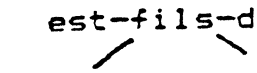
{x=<z,x>, y=<y,z>, z#<z,z>}.

Il est donc soluble.

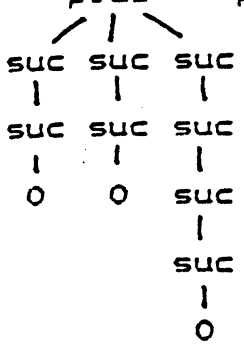
4. ASSERTIONS

4.1 LA DOUBLE DEFINITION

D'un point de vue théorique, un programme Prolog sert à definir un sous-ensemble "A" dans l'ensemble "R" de nos arbres. Les éléments de "A" sont appelés "assertions" et l'on peut généralement associer une phrase déclarative à chacun d'eux. Voici quelques exemples de telles associations:

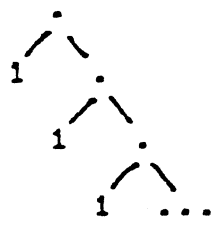


pour "Jacques est le fils de Marie",



pour "2 plus 2 donne 4",

suite-infinie pour "1 1 1 ... est une suite infinie".



L'ensemble "A" des assertions est généralement infini et constitue en quelque sorte une immense banque de données. Nous verrons plus loin que l'éxécution d'un programme peut être vue comme la consultation d'une fraction de cette banque. Bien entendu cette banque ne peut être enregistrée sous une forme explicite. Elle doit être représentée à partir d'une information finie mais suffisante pour pouvoir déduire la totalité de l'information contenue dans la banque.

Dans ce but, la définition de l'ensemble "A" des assertions est faite au moyen d'un ensemble fini de règles, chacune étant de la forme:

t0 -> t1...tn, S

où "n" peut être nul, où les "ti" sont des termes et où "S" est un système d'équations et d'inéquations éventuellement absent. Dans ce dernier cas le système est assimilé au système vide "()".

Voici la syntaxe précise de ces règles:

```
<règle>
   ::= <terme> -> <suite de termes> ;
   ::= <terme> -> <suite de termes>  , <systeme> ;

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

Pour le moment nous ferons abstraction de la notion de "<parasite>" qui, comme nous le verrons plus tard, est un moyen ad hoc d'appeler des sous-programmes non écrits en Prolog.

Ces règles qui sont donc de la forme:

```
t0 -> t1...tn, S
```

induisent un ensemble, généralement infini, de règles particulières portant sur des arbres:

```
t0(X) => t1(X)...tn(X)
```

obtenues, en considérant, pour chaque règle, toutes les affectations sylvestres possibles:

```
X = {x1:=s1,...,xm:=sm}
```

qui sont solutions de "S" et qui font intervenir les variables de la règle en question.

Chacune de ces règles particulières:

```
r0 => r1...rn
```

peut s'interpréter de deux facons:

(1) Comme une "règle de réecriture":
"r0" se réecrit dans la suite "r1...rn",
et donc, lorsque "n=0", comme:
"r0" s' efface.

(2) Comme une "implication logique" portant sur le sous-ensemble d'arbres "A":
"r1", "r2", ... et "rn" éléments de "A", entraine
"r0" élément de "A".
Dans ce cas, lorsque "n=0", l'implication se résume a:
"r0" element de "A".

Suivant que l'on prend l'une ou l'autre des interprétations, les "assertions" se définissent par:

DEFINITION 1: les assertions sont les arbres que l'on peut "effacer", en une, ou en plusieurs étapes au moyen des règles de réecriture.

DEFINITION 2: les assertions forment le plus petit ensemble "A" d'arbres qui satisfait les implications logiques.

Ces deux définitions sont équivalentes. Pour le justifier et aussi montrer l'existence du plus petit ensemble de la 2eme définition, il est nécessaire d'introduire quelques notations:

Le mot "vide" désignera toute suite vide, et "u =>i v" signifiera: la suite d'arbres "u" se réecrit en "i" étapes dans la suite "v" au moyen des règles de réecriture particulières. Plus précisement:

DEFINITION:

Si "u" et "v" sont deux suites, éventuellement vides, d'arbres alors "=>i" est défini par:

"u =>i+1 v" ssi:
il existe une règle particulière "ro => r1 ... rm" et une suite eventuellement vide d'arbres "s1 ... sn" tels que:
"u = r0 s1 ... sn" et "r1 ... rm s1 ... sn =>i v",

"u =>0 v" ssi "u = v".

On remarque immédiatement que l'énoncé "u => v" correspond à l'existence d'une suite de "ui" telle que:
u = u0 =>1 u1 =>1 u2 =>1 ... =>1 un = v.

La double définition de l'ensemble des assertion peut maintenant se justifier du fait de la propriété:

PROPRIETE DE DOUBLE DEFINITION: si l'on pose:
A = l'ensemble des arbres "r" tels
qu'il existe "i" avec "r =>i vide",
alors "A" est le plus petit ensemble d'arbres qui satisfait les implications logiques associées aux règles particulières.

La démonstration est donnée en annexe. Elle repose principalement sur le principe "d'indépendance des effacements" qui se démontre facilement par récurrence sur "k":

PRINCIPE D'INDEPENDANCE DES EFFACEMENTS: quels que soient les arbres "r1", "r2", ... et "rn",

r1...rn =>k vide ssi:
"k" est une somme de "n" entiers "k = k1+...+kn" avec:
"r1 =>k1 vide", "r2 =>k2 vide", ... et "rn =>kn vide".

Il s'ensuit que pour effacer une suite d'arbres, on peut permuter leurs ordres à tout instant, et donc ignorer la restriction de réecrire toujours le premier arbre, ainsi que le veut la définition de "=>i".

4.2. EXEMPLES

EXEMPLE 1

Soit les règles:

```
dans(nil,nil) -> ;
dans(x,e.y) -> élément(e) dans(x,y);
dans(e.x,e.y) -> élément(e) dans(x,y);

élément("a") -> ;
élément("b") -> ;
élément("c") -> ;
élément("d") -> ;
```

Ces règles induisent notamment les regles particulieres:

```
'dans(nil,nil)' => vide
.........
'dans("d".nil,"c"."d".nil)' =>
   'élément("c")' 'dans("d".nil,"d".nil)'
'dans("b"."d".nil,"a"."b"."c"."d".nil)' =>
   'élément("a")' 'dans("b"."d".nil,"b"."c"."d".nil)'
.........
'dans("d".nil,"d".nil) =>
   'élément("d")' 'dans(nil,nil)'
'dans("b"."d".nil,"b"."c"."d".nil)' =>
   'élément("b")' 'dans("d".nil,"c"."d".nil)'
.........
'élément("a")' => vide
'élément("b")' => vide
'élément("c")' => vide
'élément("d")' => vide
```

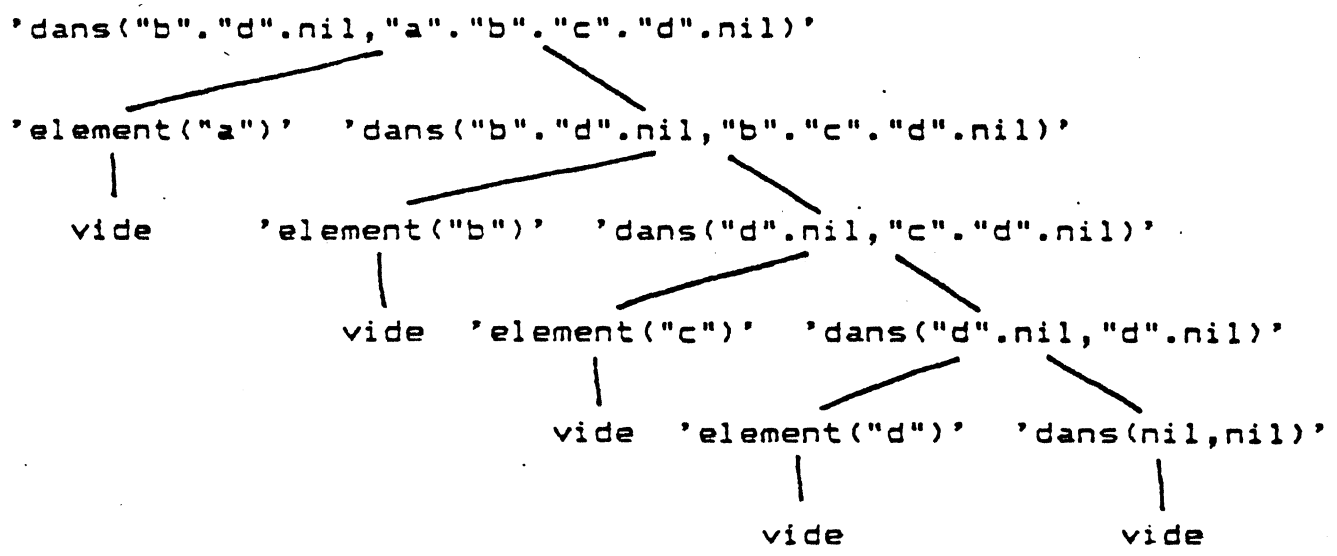
On a:

```
'dans("b"."d".nil,"a"."b"."c"."d".nil)' =>1
'élément("a")' dans("b"."d".nil,"b"."c"."d".nil)' =>1
'dans("b"."d".nil,"b"."c"."d".nil)' =>1
'élément("b")' 'dans("d".nil,"c"."d".nil)' =>1
'dans("d".nil,"c"."d".nil)' =>1
'élément("c")' 'dans("d".nil,"d".nil)' =>1
'dans("d".nil,"d".nil)' =>1
'élément("d")' 'dans(nil,nil)' =>1
'dans(nil,nil)' =>1 vide
```

et donc:

```
'dans("b"."d".nil,"a"."b"."c"."d".nil)' =>9 vide
```

ce qui montre qu'on a affaire à une assertion. Le schéma qui suit visualise bien le principe d'indépendance des effacements dans cette dérivation en 9 étapes:



Ce schema arborescent, vu de bas en haut, retrace aussi l'ensemble des implications logiques qui d'après la definition 2 font une assertion de l'arbre:
'dans("b"."c".nil,"a"."b"."c"."d".nil)'.
D'une façon générale on peut voir qu'en dehors des assertions "'element("a")'" .... "'element("d")'" toutes les assertions sont de la forme "dans(11,12)" ou "11" est une liste obtenue en supprimant de toutes les façons possibles certains éléments de la liste "12".

EXEMPLE 2

Soit les règles:

```
plus(0,x,x) -> ;
plus(suc(x),y,suc(z)) -> plus(x,y,z);
```

Ces règles induisent notamment les règles particulières:

```
'plus(0,suc(suc(0)),suc(suc(0)))' => vide
......
'plus(suc(0),suc(suc(0)),suc(suc(suc(0))))' =>
                    'plus(0,suc(suc(0)),suc(suc(0)))'
'plus(suc(suc(0)),suc(suc(0)),suc(suc(suc(suc(0)))))' =>
                    'plus(suc(0),suc(suc(0)),suc(suc(suc(0))))'
......
```

On a:

```
'plus(suc(suc(0)),suc(suc(0)),suc(suc(suc(suc(0)))))' =>1
'plus(suc(0),suc(suc(0)),suc(suc(suc(0))))' =>1
'plus(0,suc(suc(0)),suc(suc(0))) =>1  vide
```

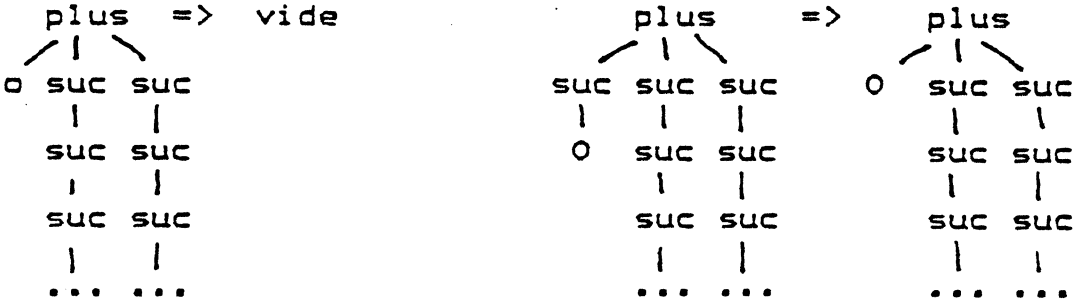
et donc:

```
'plus(suc(suc(0)),suc(suc(0)),suc(suc(suc(suc(0)))))' =>3 vide
```

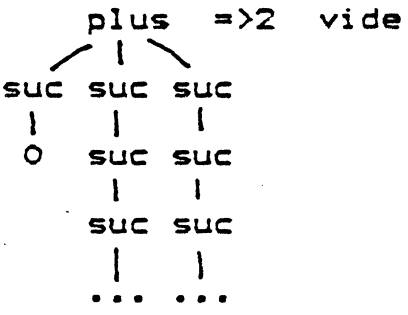
d'où l'assertion:

```
'plus(suc(suc(0)),suc(suc(0)),suc(suc(suc(suc(0)))))'
```

Il semblerait que les assertions de cet exemple soient de la forme "plus(x,y,z)" avec "x+y = z", les entiers naturels étant representés par des "suc" de "suc" de ..."0". Ceci n'est pas tout à fait le cas car les deux règles induisent aussi des règles particulières faisant intervenir des arbres infinis comme:



De ces règles on déduit l'assertion:



qui pourrait etre interprétée comme "1" plus "infini" donne toujours "infini". Si l'on veut vraiment que "plus(x,y,z)" corresponde a l'addition sur les entiers naturels il faut modifier les deux règles d'origine et écrire:

```
plus(0,x,x) -> entier(x);
plus(suc(x),y,suc(z) -> plus(x,y,z);

entier(0) -> ;
entier(suc(x)) -> entier(x);
```

EXEMPLE 3

Voici un dernier exemple faisant intervenir une inéquation:

```
hors-de(x,nil) -> ;
hors-de(x,y.l) -> hors-de(x,l), {x#y};
```

Ces règles engendrent entre autres les règles particuliéres:

```
'hors-de("d",nil)' => vide
..........
'hors-de("d","c".nil)' => 'hors-de("d",nil)'
'hors-de("d","b"."c".nil)' => 'hors-de("d","c".nil)'
'hors-de("d","a"."b"."c".nil)' =>
                              'hors-de("d","b"."c"."d".nil)'
..........
```

qui permettent la dérivation:

```
'hors-de("d","a"."b"."c".nil)' =>1
'hors-de("d","b"."c".nil)' =>1
'hors-de("d","c".nil)' =>1
'hors-de("d",nil)' =>1 vide
```

D'une façon générale toutes les assertions de cet exemple sont de la forme "hors-de(x,l)" où "l" est une liste d'éléments tous différents de "x".

5. CALCUL DE SOUS-ENSEMBLES D'ASSERTIONS

5.1. LE PROBLEME DE LA LUCARNE

Nous venons de montrer quelle est l'information implicite contenue dans un programme Prolog, mais nous n'avons pas montré ce qu'est l'éxécution d'un programme Prolog. Cette éxécution vise à résoudre le problème suivant, appelé "problème de la lucarne":

> Etant donné un programme qui est une définition récursive d'un ensemble "A" d'assertions,
>
> étant donné une "lucarne", c'est à dire un terme "t" et l'ensemble de ses variables "{x1,...,xn}",
>
> trouver toutes les assertions que l'on peut "voir" à travers cette "lucarne", c'est à dire, calculer toutes les affectations sylvestres "X = {x1:=r1,...,xn:=rn}" qui sont telles que "t(X)" soit une assertion.

La solution de ce problème passe par l'introduction de la relation binaire "->i" entre des couples de la forme "(u,S)" où "u" est une suite éventuellement vide de termes et S un système d'équations et d'inéquations.

DEFINITION:

Si "u" et "v" sont deux suites éventuellement vides de termes et si "S" et "T" sont deux systèmes, alors la relation "->i" est définie par:

"(u,S) ->i+1 (v,T)" ssi:
> il existe une règle "s0 -> s1...sm, U", dont les variables ont été renommées de facon à n'en avoir aucune de commune avec "(u,S)", et si l'on pose "u = t0 t1...tn", les "ti" étant des termes, on a:
> (s1...sm t1...tn, SuUu{t0=s0}) ->i (u,T);

"(u,S) ->0 (v,T)" ssi "u = v", "S = T" et "S" est soluble.

La relation "->i" a plusieurs propriétés analogues à celle de la relation "=>i":

Tout d'abord, il découle immediatement de la définition, que l'énoncé "(u,S) ->i (v,T)" correspond à l'éxistence d'une suite de "(ui,Si)" telle que:

(u,S) = (u0,S0) ->1 (u1,S1) ->1 ... ->1 (un,Sn) = (v,T).

De plus on retrouve, sous une forme légèrement différente, le principe d'indépendance des effacements, principe qui se démontre par récurrence sur l'entier "k":

PRINCIPE GENERALISE D'INDEPENDANCE DES EFFACEMENTS: Quels que soient les termes "t1,...,tn" et les systèmes "S" ET "T":

"(t1...tn,S) ->k (vide,T)" ssi:
"k" est une somme de "n" entiers "k1+...+kn = k" et "T" est une union de "n" systèmes "T1u...uTn = T", n'ayant deux à deux aucune variable commune autre que celles contenues dans "(t1...tn,S)", avec:
"(t1,S) =>k1 (vide,T1)" ... et "(tn,S) =>kn (vide,Tn)".

Il s'ensuit que pour "effacer" (au moyen de "=>i") une suite "u" de terme figurant dans un couple "(u,S)", on pourra permuter ses termes à tout instant. Voici la derniere analogie. Elle permet de conclure que "->i" est une "généralisation" de "=>i":

PRINCIPE DE GENERALISATION: Soit "i" un entier (non negatif), soit "t1...tn" une suite (eventuellement vide) de termes, soit "S" un système et soit "X" une affectation sylvestre des variables contenues dans "(t1...tn,S)", alors:

"X" est solution sylvestre de "S", et "t1(X)...tn(X) =>i vide" ssi:
Il existe un système "T" avec:
"(t1...tn,S) ->i (vide,T)" et "X" solution sylvestre de "T".

Nous donnons la démonstration de cette propriété en annexe. En particularisant ce principe a "n = 1", "S = {}" et en faisant intervenir la définition 1 de l'ensemble des assertions, on obtient:

PRINCIPE DE LA LUCARNE: Pour tout terme "t" et pour toute affectation sylvestre X de ses variables:

"t(X)" est une assertion ssi:
il existe un entier "i" et un systeme "S" avec:
"(t,{}) =>i (vide,S)" et "X" solution sylvestre de "S".

Pour résoudre notre problème initial il suffira donc d'énumérer toutes les suites:
(t,{}) = (u0,S0) ->1 (u1,S1) ->1 (u2,S2) ->1 ...
et d'essayer d'atteindre tous les "Si" dont le "ui" correspondant est vide. Chaque affectation sylvestre "X" des variables de la lucarne "t qui fait de "t(X)" une assertion sera alors solution d'un tel "Si". Bien entendu, du point de vue algorithmique, il y aura un problème du fait que certaines de ces suites peuvent être infinies.

Pour illustrer tout ceci nous allons reconsidérer les trois exemples du chapitre précédent:

5.2. EXEMPLES

EXEMPLE 1

Les règles sont:

```
dans(nil,nil) -> ;
dans(x,e.y) -> element(e) dans(x,y);
dans(e.x,e.y) -> element(e) dans(x,y);

element("a") -> ;
element("b") -> ;
element("c") -> ;
element("d") -> ;
```

Soit à calculer les assertions que l'on peut voir à travers la lucarne:

```
t = dans(z,"a"."b"."c"."d".nil)
```

c'est à dire soit à calculer toutes les affectations:

```
X = {z:=r}
```

telles que t(X) soit une assertion.

On a successivement (Si' représenté la forme reduite de Si):

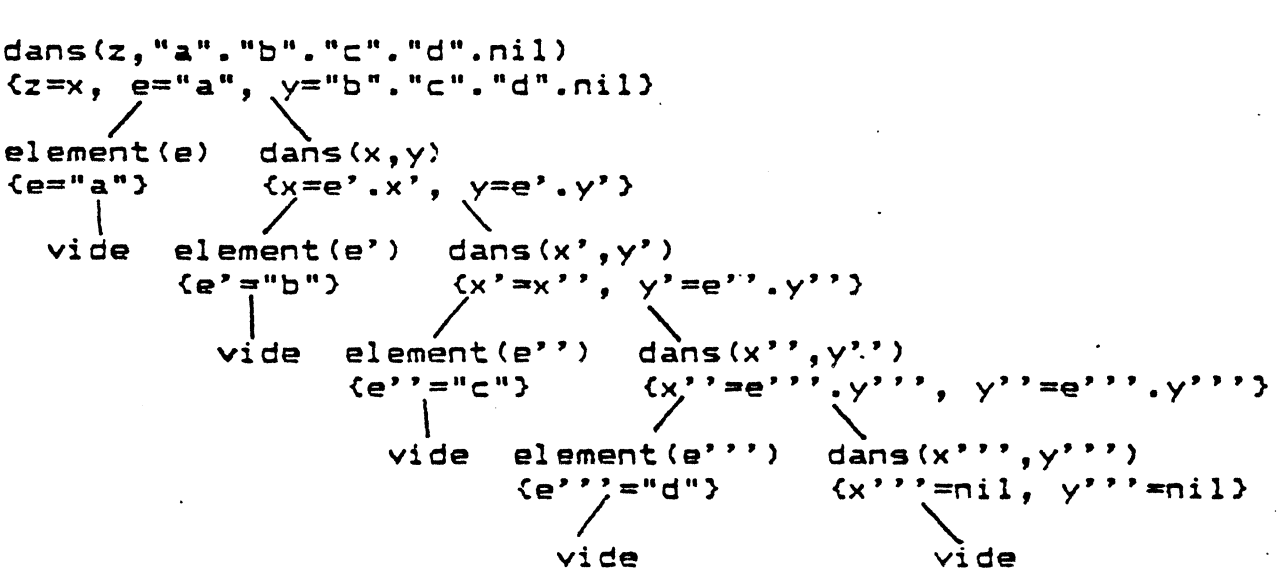
```
u0  = dans(z,"a"."b"."c"."d".nil)
S0  = {}
S0' = {}

u1  = element(e) dans(x,y)
S1  = S0 u {dans(z,"a"."b"."c"."d".nil)=dans(x,e.y)}
S1' = {z=x, y="b"."c"."d".nil, e="a"}

u2  = dans(x,y)
S2  = S1 u {element(e)=element("a")}
S2' = {z=x, y="b"."c"."d".nil, e="a"}

u3  = element(e') dans(x',y')
S3  = S2 u {dans(x,y)=dans(e'.x',e'.y')}
S3' = {z=x, x=e'.x', y=e'.y', y'="c"."d".nil, e="a", e'="b"}

u4  = dans(x',y')
S4  = S3 u {element(e')=element("b")}
S4' = {z=x, x=e'.x', y=e'.y', y'="c"."d".nil, e="a", e'="b"}

u5  = element(e'') dans(x'',y'')
S5  = S4 u {dans(x',y')=dans(x'',e''.y'')}
S5' = {z=x, x=e'.x', x'=x'', y=e'.y', y'=e''.y'',
       y''="d".nil, e="a", e'="b", e''="c"}

u6  = dans(x'',y'')
S6  = S5 u {element(e'')=element("c")}
S6' = {z=x, x=e'.x', x'=x'', y=e'.y', y'=e''.y'',
       y''="d".nil, e="a", e'="b", e''="c"}
```

```
u7  = element(e''') dans(x''',y''')
S7  = S6 u {dans(x'',y'')=dans(e'''.x''',e'''.y''')}
S7' = {z=x, x=e'.x', x'=x'', x''=e'''.x''', y=e'.y',
       y'=e''.y'', y''=e'''.y''', y'''=nil, e="a", e'="b",
       e''="c", e'''="d"}

u8  = dans(x''',y''')
S8  = S7 u {element(e''')=element("c")}
S8' = {z=x, x=e'.x', x'=x'', x''=e'''.x''', y=e'.y',
       y'=e''.y'', y''=e'''.y''', y'''=nil, e="a", e'="b",
       e''="c", e'''="d"}

u9  = vide
S9  = U8 u {dans(x''',y''')=dans(nil,nil)}
S9' = {z=x, x=e'.x', x'=x'', x''=e'''.x''', x'''=nil,
       y=e'.y', y'=e''.y'', y''=e'''.y''', y'''=nil, e="a",
       e'="b", e''="c", e'''="d"}
```

Donc entre autres:

```
X = {z := "b"."d".nil}
```

De la même manière on peut obtenir pour résultat les 15 autres sous-listes de "a"."b"."c"."d".nil correspondant à toutes les façons possibles de supprimer certains de ses éléments. Le schéma arborescent qui suit illustre bien le principe d'indépendance de l'effacement précédent, par "->9":



EXEMPLE 2

Les règles sont:

```
plus(0,x,x) -> ;
plus(suc(x),y,suc(z)) -> plus(x,y,z);
```

Afin de faire intervenir les arbres infinis, soit à calculer les assertions que l'on peut voir à travers la lucarne:

t = plus(suc(0),x,x)

c'est à dire à trouver les affectations sylvestres:

X = {x:=r}

telles que "t(X)" soit une assertion.

On a successivement:

u0 = plus(suc(0),x,x)
S0 = {}
S0' = {}

u1 = plus(x',y',z')
S1 = S0 u {plus(suc(0),x,x)=plus(suc(x'),y',suc(z'))}
S1' = {x=y', x'=0, y'=suc(z')}

u2 = vide
S2 = S1 u {plus(x',y',z')=plus(0,x'',x'')}
S2' = {x=x'', x'=0, x''=suc(x''), y'=x'', z'=x''}

La seule solution est donc:

X = {x := suc}
suc
suc
...

9037

EXEMPLE 3

Les règles sont:

hors-de(x,nil) -> ;
hors-de(x,y.l) -> hors-de(x,l), {x#y};

Prenons pour lucarne:

t = hors-de("c","a"."b".nil).

Il faut donc vérifier que "t({})" est une assertion.

En effet, on a successivement:

u0 = hors-de("c","a"."b".nil)
S0 = {}
S0' = {}

u1 = hors-de(x,l)
S1 = S0 u {hors-de("c","a"."b".nil)=hors-de(x,y.l), x#y}
S1' = {x="c", l="b".nil, y="a"}

u2 = hors-de(x',l')
S2 = S1 u {hors-de(x,l)=hors-de(x',y'.l'), x'#y'}
S2' = {x=x', x'="c", l=y'.l', l'=nil, y="a", y'="b"}

```
u3  = vide
S3  = S2 u {hors-de(x',l')=hors-de(x'',nil)}
S3' = {x=x', x'=x'', x''="c", l=y'.l', l'=nil, y="a", y'="b"}
```

6. LA MACHINE PROLOG

6.1. L'HORLOGE PROLOG

Comme nous l'avons vu, le langage de programmation Prolog permet, d'une part, de définir sous forme implicite des ensembles infinis d'assertions et, d'autre part, de calculer, c'est à dire d'énumérer, certaines assertions intéressantes. Ce calcul se fait par énumeration de suites de couples:

(u0,S0) ->1 (u1,S1) ->1 (u2,S2) ->1 ....

Jusqu'à maintenant nous avons toujours considéré que "u0" était réduit à un seul terme "t" et que le système S0 était l'ensemble vide "{}". Si l'on reconsidère le principe de généralisation et que l'on fait intervenir à la fois la 1ère définition des assertions et le principe d'indépendance des effacements par "->i" on peut reformuler le principe de la lucarne sous une forme plus générale où "u0" sera une suite "t1...tn" quelconque de termes et "S0" un système quelconque "S" d'équations et d'inéquations:

PRINCIPE DE LA LUCARNE ELARGIE: soit "{t1,...,tn}" un ensemble de termes, soit "S" un système, soit "{x1,...,xm}" l'ensemble des variables qu'ils font intervenir, et soit "X" une affectation sylvetre de la forme "X = {x1:=r1,...,xm:=rm}", alors:

"X" est solution de "S" et "{t1(X),...,tn(X)}" est contenu dans l'ensemble des assertions ssi:
il existe un entier "i" et un système "T" avec:
"(t1...tn,S) =>i (vide,T)" et "X" solution de "T"

L'énumération des couples "(ui,Si)" peut se faire de différentes manières et particulièrement dans différents ordres. Pour préciser tout cela et aussi pour pouvoir introduire des instructions extérieures au modèle décrit jusqu'à maintenant, nous décrirons la sémantique opérationelle de Prolog au moyen d'une machine abstraite appelée "horloge Prolog" car le temps y joue un role important.

La machine est composée:

(1) D'une cellule "suite-de-règles" contenant la suite des règles qui constitue le programme Prolog proprement dit.

(2) D'une cellule "t" contenant le temps. Ce temps n'est rien d'autre que l'indice "i" du couple "(ui,Si)" en cours de traitement.

(3) De trois infinités de cellules, chacune étant indicée par un entier non negatif "i":

(a) "buts(i)" contient la suite de termes "ui";

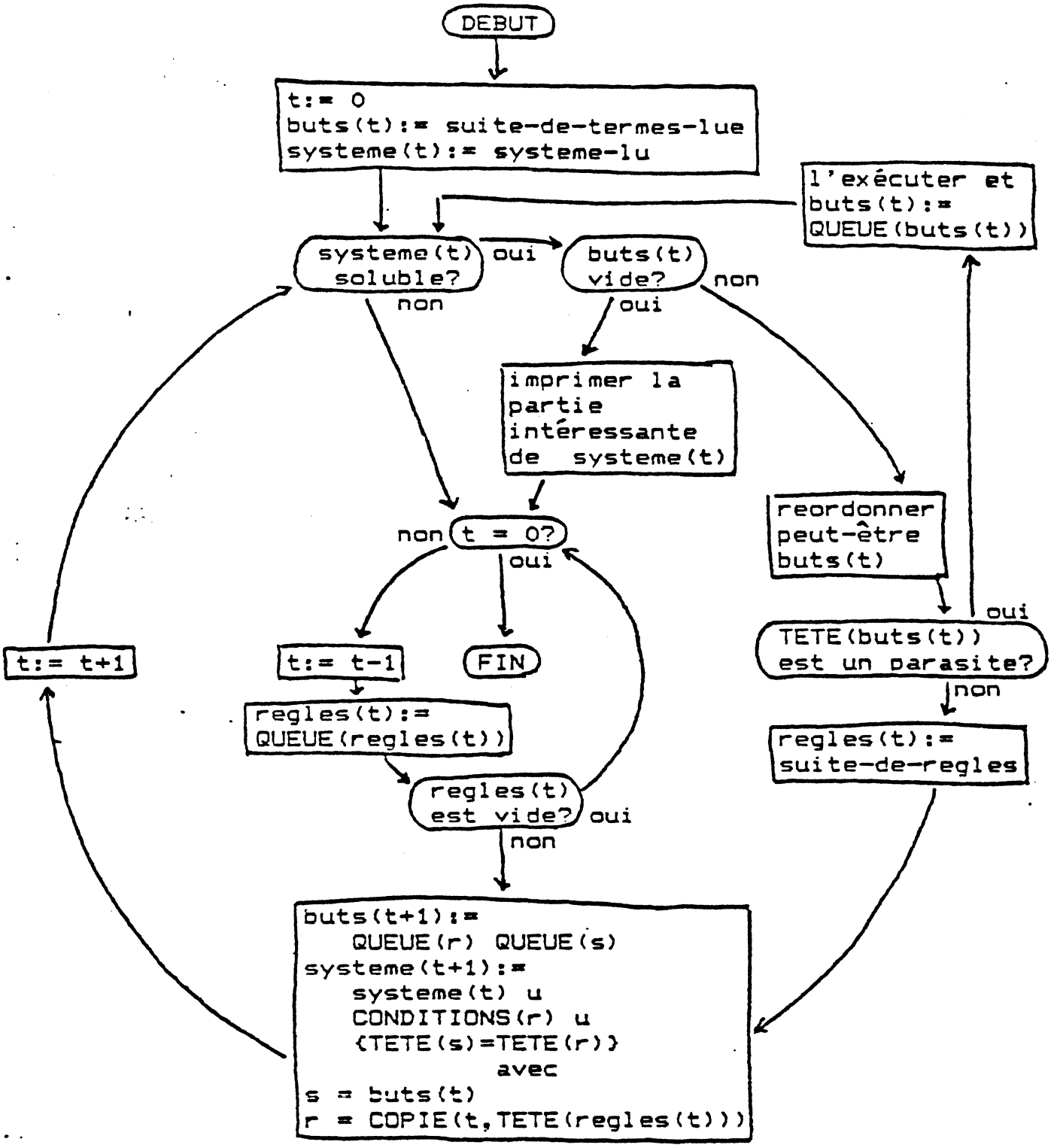
(b) "systeme(i)" contient le système "Si".

(c) "regles(i)" contient la suite des regles "actives" a l'instant "i".

Seule la cellule "suite-de-regles" possède une valeur initiale qui donne au programmeur un environnemet de départ. La machine lit des commandes sur son unité d'entree, les éxécute les unes après les autres et imprime chaque fois les résultats calculés. Voici la syntaxe d'une commande:

<commande>
::= <suite de terme> ;
::= <suite de termes> , <systeme> ;

Bien entendu l'absence de <systeme> dans une commande correspond au systeme vide "{}". Le fonctionnement de la machine se schématise comme suit:



TETE(x): représente le premier élément de la suite "x", ou bien le membre gauche de la règle "x".

QUEUE(x): représente la suite "x" amputée de son premier élément ou le membre droit (moins le système) de la règle "x".

CONDITIONS(r): représente le système d'équations et d'inéquations de la règle "r".

COPIE(t,r): représente une t-ième copie de la règle "r". Cette copie n'a de variable commune ni avec "buts(t)" ni avec "systeme(t)".

Les parasites sont des noms de sous-programmes permettant d'effectuer différentes tâches.

<parasite>
::= /
::= <syntaxe inconnue, mais differente de celle d'un terme>

Seul le parasite "/", dont nous décrirons l'effet un peu plus loin, est directement accessible au programmeur. Les autres parasites figurent à l'interieur du grand ensemble de règles prédéfinies qui est affecté au départ à la cellule "suite-de-regles". Cet ensemble de règles prédéfinies constitue un environnement de programmation très complet permettant notamment:

(1) de contrôler et de modifier le déroulement d'un programme;

(2) de modifier l'ensemble courant de règles contenu dans "suite-de-regles" et donc d'introduire et de modifier des programmes;

.3) d'avoir accès aux fonctions classiques d'arithmétique et de traitement de chaines;

(4) de gérer les entrées-sorties.

Cet environnement est longuement décrit dans le manuel d'utilisation Prolog II de Michel van Caneghem. Les points (1) et (2) étant fondamentaux, nous donnons dès maintenant une description précise du contrôle en Prolog et nous aborderons aussi, dans ses grandes lignes, la gestion des programmes c'est à dire la gestion des règles qui les constituent.

6.2. LE CONTROLE

Le contrôle s'effectue à l'aide du parasite "/" et de règles prédéfinies que nous allons énumerer et commenter. Pour pouvoir nommer les parasites figurant dans ces règles, et dont la syntaxe est en principe inconnue de l'utilisateur, nous utiliserons des identificateurs entre simples guillemets.

LE PARASITE "/"

L'horloge Prolog simule une machine non déterministe. Comme on peut le constater sur le schéma, cette simulation est faite par une technique de retour arrières (backtracking) sur des points de choix accumulés. Cette accumulation d'information a forcément des limites, et il est nécessaire de pouvoir supprimer tous les points de choix accumulés entre une date passée "t-k" et la date présente "t".

L'exécution du parasite "/" a pour effet d'affecter "vide" à toutes les cellues "regle(i)" dont l'indice "i" est élément de "{t-k,...,t-1,t}". L'entier "t-k" est la dernière date telle que "buts(t-k)" ne contenait pas l'occurence considérée de "/".

REGLE PREDEFINIE: pris(x) -> 'pris';

L'exécution du parasite 'pris' n'a aucun effet si "x" est "pris" et rend "systeme(t)" insoluble dans le cas contraire.

Par "x" est "pris" on entend que "x" est une variable telle que:
- soit, il existe une valeur "k" de constante telle que, dans toutes les solutions {x:=r} de "systeme(t)", "r = k";
- soit, dans toutes les solutions "{x:=r}" de "systeme(t)", "r" est de la forme "r = r1.r2";
- soit, il existe un entier "n" tel que, dans toutes les solutions "{x:=r}" de "systeme(t)", "r" est de la forme "r = <r1,...,rn>".

REGLE PREDEFINIE: libre(x) -> 'libre';

L'exécution du parasite 'libre' n'a aucun effet si "x" est "libre" et rend "systeme(t)" insoluble dans le cas contraire.

Par "x" est "libre" on entend que "x" n'est pas "pris".

REGLE PREDEFINIE: geler(x,p) -> 'en-attente' x p;

D'apres le principe généralisé d'indépendance des effacements, il est possible à tout instant de réordonner les éléments de la suite "buts(t)" et donc, notamment, de retarder la réecriture d'un élément particulier "p" de cette suite jusqu'à ce qu'une certaine variable "x" soit "prise". La règle prédéfinie ci-dessus est prévue à cet effet.

En principe le parasite 'en-attente' n'est jamais exécuté, il est pris en compte par l'opération qui consiste à réordonner la suite "buts(t)" dans l'horloge Prolog. Ce réordonnancement se fait en deux temps:

tout d'abord, tous les triplets de la forme "'en-attente' x p" qui apparaissent dans "buts(t)" sont regroupés à la fin de "buts(t)";

-ensuite, tous les "p" faisant partie de triplets de la forme "'en-attente' x p" dont la variable "x" est devenue "prise" a l'instant "t" sont regroupés au début de "buts(t)" et les occurences correspondantes de 'en-attente' et de "x" sont supprimées.

Tous ces regroupements sont faits en respectant les ordres des "p" dans "buts(t)". Il va de soi que si, malgré l'élimination progressive du parasite 'en-attente', on était amené à exécuter un tel parasite, l'effet serait d'enlever de "buts(t)" le terme "x" qui suit immédiatement.

Voici un exemple intéressant d'utilisation de "geler(x,p)". Il s'agit d'un programme permettant de vérifier qu'un terme "x" représente et représentera toujours un arbre fini.

```
arbre-fini(x) -> branche-finie(x,nil);
```

```
branche-finie(x,1) -> geler(x,branche-finie'(x,1));

branche-finie'(x,1) ->
   hors-de(x,1) domine(x,1') branches-finies(1',x.1);


branches-finies(nil,1) -> ;
branches-finies(x.1',1) ->
   branche-finie(x,1) branches-finies(1',1);

hors-de(x,nil) -> ;
hors-de(x,y.1) -> hors-de(x,1), {x#y};
```

"domine(x,1)" est supposé prédéfini avec des parasites qui simulent le pseudo-programme:

```
domine(x,1) -> libre(x) / erreur("dans domine");
domine(x,nil) -> constante(x);
domine(x1.x2,x1.x2.nil) -> ;
domine(<>,nil) -> ;
domine(<x1>,x1.nil) -> ;
domine(<x1,x2>,x1.x2.nil) -> ;
domine(<x1,x2,x3>,x1.x2.x3.nil) -> ;
.....

constante(x) -> ....

erreur(m) -> ...
```

REGLE PREDEFINIE: bloc(n,p) -> p 'etiquette' n /;

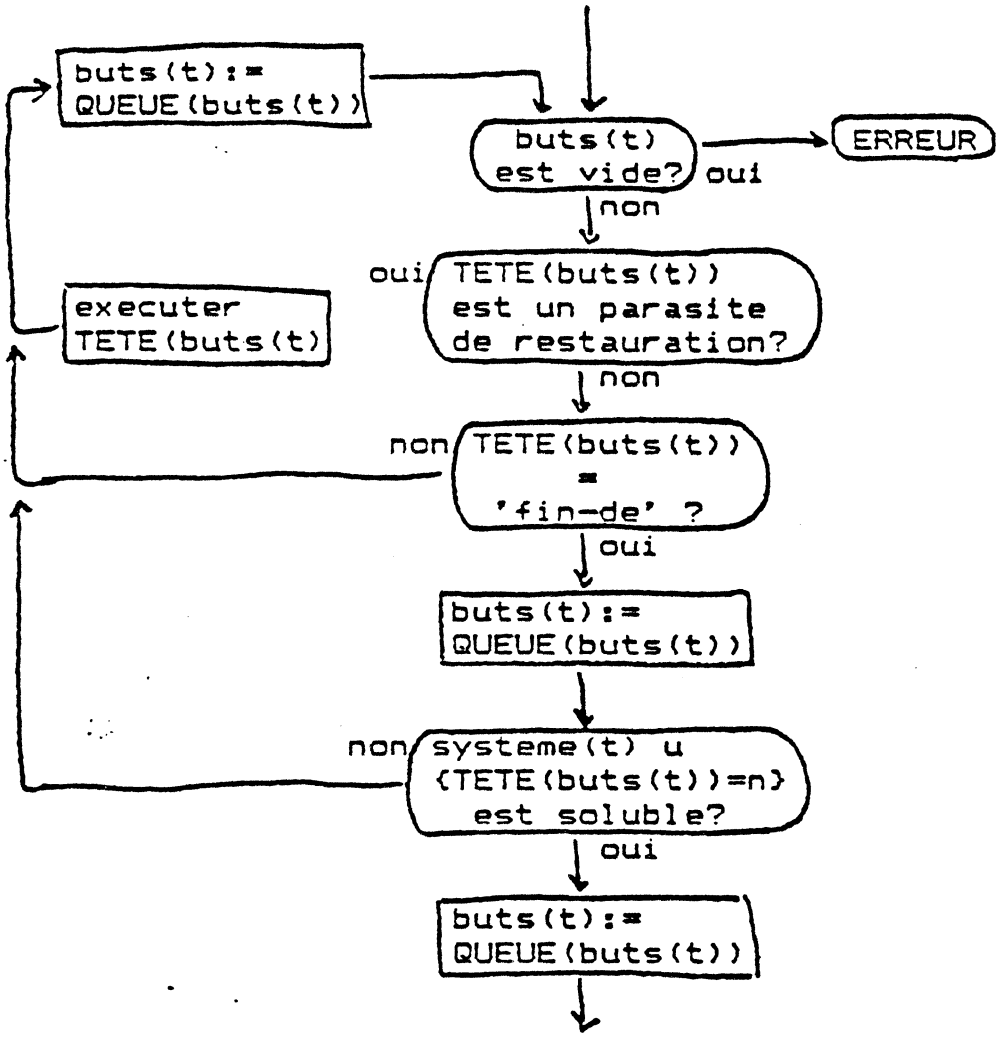
Cette règle permet d'insérer une etiquette "n" à l'interieur de la suite "buts(t)" et de pouvoir ainsi interrompre la réecriture d'un certain nombre de termes, par un saut vers cette étiquette. La facon recursive dont ces étiquettes sont introduites, structure la suite "buts(t)" en des "blocs" imbriqués portant comme noms les étiquettes correspondantes. Sauter vers une étiquette "n" revient alors à terminer brutalement l'effacement du bloc portant le nom "n".

L'éxécution du parasite 'etiquette' a pour effet d'éliminer de "buts(t)" le terme qui suit l'occurence de ce parasite.

REGLE PREDEFINIE: fin-bloc(n) -> 'fin-bloc';

Ceci est précisément la règle permettant de terminer brutalement l'effacement du premier bloc englobant et de nom "n". Certains parasites ne peuvent cependant pas être ignorés dans ce cas. On les appellent "parasites de restauration" et il seront quand même exécutés.

Plus précisément l'exécution du parasite 'fin-bloc' déroule les instructions suivantes:



REGLE PREDEFINIE: p.q -> p q;

Cette règle permet d'assimiler une suite de buts à un seul but.

REGLE PREDEFINIE: eg(x,x) -> ;

Cette règle permet d'éviter de faire figurer des équations dans les règles ou les commandes.

REGLE PREDEFINIE: dif(x,y) ->, {x#y};

Cette règle permet d'éviter de faire figurer des inéquations dans les règles ou les commandes.

6.3. UN APERCU DE LA GESTION DES PROGRAMMES

Il est possible, à tout instant, de modifier le contenu de la cellule "suite-de-regles" en ajoutant ou en supprimant certaines règles. Ces modification se font toujours par rapport à un pointeur courant dont on peut modifier la position. La modification la plus courante consiste à inserer une suite de règles. Cette suite constitue en fait un programme Prolog dans lequel peuvent apparaitre des commentaires. En voici la syntaxe:

```
<programme>
    ::= ;
    ::= <enonce> <programme>

<enonce>
    ::= <commentaire>
    ::= <regle>

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

L'insertion se fait au moyen de la regle predefinie:

```
inserer -> 'inserer';
```

Le parasite 'inserer' lit un programme sur l'unité d'entrée et l'insère juste au dessus du pointeur courant. Quoique les commentaires ne jouent aucun rôle dans l'exécution d'un programme, ils sont conservés pour pouvoir être restitués.

Il existe une restriction importante dans la syntaxe des règles d'un programme. Cette restriction vise à optimiser les recherches dans l'ensemble des règles, en permettant des accès directs par le biais des identificateurs. Elle s'énonce ainsi:

- Tout terme qui constitue un membre gauche de règle doit contenir au moins une occurence d'identificateur non précédée (à gauche), d'une variable, d'une constante, ou de "<>".

- Aucun terme qui constitue un membre gauche de règle ne peut être de la forme "t1.t2". Exception est faite pour la règle prédéfinie: p.q -> p q;

Afin de gérer commodement de grands ensembles de règles Prolog, l'ensemble des règles est partitionné en sous-ensembles appelés "mondes". Chacun de ces monde porte un nom "M" constitué d'une suite de chaines: "M = c1,c2,...,cn".

On dit que le monde de nom "M2" est "sous-monde" du monde de nom "M1" si "M1" et "M2" sont respectivement de la forme:
"M1 = c1,...,cm" et "M2 =c1,...,cm,cn+1,...,cn",
ou "n" est est strictement superieur a "m". Si "n=m+1" on parle de sous-monde immédiat. Cette notion de sous-monde crée une hiérarchie de mondes au sommet de laquelle se trouve le monde dont le nom est la suite vide.

A toute occurence d'identificateur est associé un monde. Cette association est faite au moment où cette occurence est lue et dépend du monde courant "M" dans lequel on se trouve à ce moment. Deux cas se présentent:

(1) Le monde "M" est sous-monde d'un monde "Mbis" associé à une autre occurence du meme identificateur. On associe alors "Mbis" à l'occurence de l'identificateur lu.

(2) Dans le cas contraire on associe le monde courant "M" à l'occurence de l'identificateur lu.

Deux occurences d'identificateur constitués de la même suite de caractères ne sont considérées comme égales que si le même monde leur est associé. Plus précisément la valeur d'une occurence d'identificateur "id" dont le monde associé est "M" est le couple "(M,id)".

Nous avons rassemblé ici toute les demonstrations importantes qui ne figurent pas dans le texte.

(3.1. systèmes sous forme réduite)

PROPRIETE DE RESOLUBILITE: tout système sous forme réduite est soluble.

Démonstration. Soit "S" un système sous forme réduite. Si "S" contient une équation de la forme "v=w" ou "v" et "w" sont des variables, on supprime cette équation et l'on remplace toute autre occurence de "v" par "w". En répétant cette opération autant de fois que nécéssaire, on aboutit à un systeme "T" qui est toujours sous forme réduite, qui ne contient plus d'équation de la forme "v=w" et dont la résolubilité entraine celle de "S". Il suffit donc de montrer que "T" est soluble.

Posons "T = EuI" ou "E" est l'ensemble des équations et "I" l'ensemble des inéquations. Soit "{x1,...,xm}" l'ensemble des variables constituant les membre gauches d'équations de "E" et soit "{y1,...,yn}" l'ensemble des autres variables apparaissant dans "T". On considère le système:

E u {y1=k1,...,yn=kn}

où les "ki" sont des constantes ayant des valeurs toutes différentes et différentes des valeurs de celles qui figurent dans "T". D'après la propriété caractéristique 3, ce système admet une solution de la forme:

X = {x1:=r1,...,xm:=rm} u {y1:='k1',...yn:='kn'}.

L'affectation sylvestre "X" est solution de "E", il ne reste plus qu'à démontrer qu'elle est aussi solution de "I".

Raisonnons par l'absurde: supposons que "X" ne soit pas solution de "I". Il existe donc une inéquation de "I":

<u1,...,up>#<t1,...,tp>

telle que les arbres "<u1,..up>(X)" et "<t1,...,tp>(X)" soient égaux et donc, d'après la propriété caractéristique de décomposition unique:

u1(X) = t1(X).

Du fait que "T" est sous forme réduite et ne contient pas d'équation de la forme "v=w", il ne peut se présenter que trois cas:

(1) "u1" et "t1'" sont des variables différentes prises dans "{y1,...,yn}". Du fait que les "ki" ont tous des valeurs différentes, "u1(X)" est différent de "t1(X)", ce qui contredit l'égalité "u1(X)=t1(X)"; c.q.f.d.

(2) "u1" est une variable prise dans "{y1,...,yn}", "t1" est une variable prise dans "{x1,...,xm}" et il existe dans "E" une équation de la forme "t1=t1'" où, bien entendu "t1'" n'est pas une variable. Du fait, entre autres, que les "ki" ont des valeurs différentes de celles des constantes qui figurent dans "T", "u1(X)" est différent de "t1'(X)", et donc aussi de "t1(X)", ce qui contredit l'égalité "u1(X)=t1(X)"; c.q.f.d.

(3) "u1" est une variable prise dans "{y1,...,yn}" et "t1" n'est pas une variable. Du fait, entre autres, que les "ki" ont des valeurs différentes de celles des constantes qui figurent dans "T", "y1(X)" est different de "t1(X)", ce qui contredit l'égalité "u1(X)=t1(X)"; c.q.f.d.

(3.2. systèmes équivalents)

EQUIVALENCE 2: Soient "S" et "T" deux systèmes sous forme réduite et de la forme:

"S = {x1=t1,...,xn=tn}" et "T = {x1=t1',...,xn=tn'}".

Si toute solution sylvestre de "S" est solution sylvestre de "T" alors "S" et "T" sont équivalents.

Démonstration. Soit "{y1,...,ym}" les variables autres que les "xi" qui interviennent dans "SuT". Pour montrer que "S" et "T" sont équivalents, il suffit de considérer une solution sylvestre de "T" de la forme:

X ={x1:=a1,...,xn:=an,y1:=b1,...,ym:=bm}

et de montrer qu'elle est aussi solution de "S". Soit

{b1,...,bm,c1,c2,...}

l'ensemble des sous-arbres de "{b1,...,bm}". D'après la propriété de système associé du paragraphe 2.4, il existe un système "E" de la forme:

E = {y1=p1, ..., ym=pm, z1=q1, z2=q2, ...}

dont l'affectation:

{y1:=b1, ..., ym:=bm, z1:=c1, z2:=c2, ...}

est solution sylvestre, et tel que les "pi" et "qi" ne soient pas des variables et qu'aucune autre variable que les "yi" et "zi" (tous differents) n'interviennent dans "E". Il est facile de transformer "SuE" en un système équivalent sur lequel s'applique la 3eme propriété caractéristique. De même pour le système "TuE". On en conclut que chacun des systèmes "SuE" et "TuE" n'a qu'une et une seule solution sylvestre de la forme:

{x1:=a1',...,xn:=an',y1:=b1',...,ym:=bm',z1:=c1',z2:=c2',...}.

Mais comme toute solution de "SuE" est solution de "TuE" ce ne peut être que la même solution. La solution de "TuE":

{x1:=a1,...,xn:=an,y1:=b1,...,yb:=pm,z1:=c1,z2:=c2,...}

est donc solution de "SuE". On en conclut que l'affectation:

X ={x1:=a1,...,xn:=an,y1:=b1,...,ym:=bm}

est bien solution de "S"; c.q.f.d.

(3.3. reduction d'equations)

PROPRIETE D'ARRET: si "S1" est un système fini d'équations alors il n'existe pas de suite infinie "S1,S2,S3,..." ,de systèmes, tel que chaque "Si+1" soit obtenu en appliquant sur "Si" la transformation T1, T2, T3, T4 ou T5.

Démonstration. Nous montrons tout d'abord qu'une telle suite est forcément finie si l'on met à l'ecart la transformation d'explosion. Considérons le système initial "S1". Soit "n" le nombre cumulé d'équations de la forme "x=x" et "x=y" et soit "m" le nombre cumulé d'équations de la forme "x=t" et "t=x". Si l'on exclut les explosions, les nombres d'absorptions et

d'éliminations ne peuvent être superieurs à "n" et les nombres d'antépositions et de mises en conflit ne peuvent être superieurs à "m".

Considérons maintenant l'application "f" qui associe respectivement les entiers non negatifs "f(S)" et "f(t)" à chaque système "S" et à chaque terme "t" et definie par:
- f({s1=t1,...,sn=tn}) = f({s1=t1}) + ... + f({sn=tn}),
- f({si=ti}) = k "exposant" max{f(si),f(ti)},
- f(t) = "taille du terme" t (defini dans T5),

ou "k" est un entier plus grand que 2 et plus grand que la longueur "l" de la construction "<e1,...,el>" la plus longue figurant dans "S1". On vérifie alors deux points.

(1) "f(Si)" est strictement supérieur a "f(Si+1)" dans le cas d'une transformation de type explosion car:
"f({<s1,...,sn>=<t1,...,tn>})" est strictement supérieur à "k exposant (q+1)", ou "q" désigne la plus petite parmi les tailles des "si" et "ti";
"k exposant (q+1)" est lui même strictement supérieur à "n fois (k exposant q)", qui est supérieur ou égal à "f({s1=t1}) +....+ f({sn=tn})", qui est supérieur ou égal à "f({<s1=t1>,...,<sn=tn>})";
et de même:
"f({s1.s2=t1.t2})" est strictement superieur a "f({s1=t1,s2=t2})".

(2) "f(Si)" n'est évidemment pas inférieur à "f(Si+1)", dans le cas de toute autre transformation.

D'après la première partie de notre démonstration, l'existence d'une suite infinie de "Si" présuppose l'application d'une infinité d'explosions. Les inégalités de la deuxième partie de la démonstration rendent cette suite infinie impossible car, contrairement à sa definition, "f(Si)" deviendrait négatif; c.q.f.d.

PROPRIETE DE CONSERVATION: Soit un système "S1", soluble, sans inéquations et de la forme "S1 = E1uF1" où "E1" est sous forme réduite:
E1 = {x1=t1,...,xn=tn},
avec la restriction que chaque "xi", dont le "ti" correspondant est une variable, n'a qu'une seule occurence dans tout le système "E1uF1". Si l'on applique l'algorithme de réduction de base sur "S1" on obtient alors un système réduit "S2" de la forme "S2 = E2uF2", avec "E2" et "F2" disjoints et "E2" de la forme:
E2 = {x1=t1',...,xn=tn'}.
De plus le système "E1uF2" est sous forme réduite et est équivalent au système initial "E1uF1".

Démonstration. Considérons une équation quelconque "xi=ti" du système "E1". Si "ti" est une variable, alors aucune des transformations T1, T2, T3, T4 et T5 ne pourra faire disparaitre l'occurence unique de "xi" au cours de la réduction de "S1", et donc il figurera une équation de la forme "xi=ti'" dans le système "S2". Si "ti" n'est pas une variable alors il figurera toujours une équation de la forme "xi=ti'" dans "S2", car sinon

il existerait au moins une affectation "{xi:=r}" qui serait solution de "S2" sans être solution du système equivalent "S1". Le système "S2" est donc bien de la forme prévue "S2 = E2uF2" avec "E2 = {x1=t1',...,xn=tn'}".

Le système "E1uF2" ne peut contenir de circuit de variable car, d'une part, "F2", qui est réduit, n'en contient pas et, d'autre part, tout membre gauche "xi" d'equation de "E1", dont le membre droit "ti" est une variable, n'a qu'une seule occurence dans "E1uF2". Il s'agit donc d'un système sous forme réduite.

Les membres gauches d'équations des deux systèmes "E2uF1" et "E1uF1" sont les mêmes. Toute solution de "E1uF1" est solution de "E2uF2" et donc de "E1uF2". D'après l'équivalence 2 du paragraphe 3.2, les systèmes "E1uF2" et "E1uF1" sont équivalents; c.q.f.d.

(4.1. la double definition)

PROPRIETE DE DOUBLE DEFINITION: si l'on pose:
A = l'ensemble des arbres "r" tels
qu'il existe "i" avec "r =>i vide",
alors "A" est le plus petit ensemble d'arbres qui satisfait les implications logiques associées aux règles particulières.

Démonstration: 1ère partie. Montrons que "A" satisfait l'implication logique associée à chaque règle particulière:
r0 => r1...rn.
Si "n=0" alors
"r0 =>1 vide" et donc "r0" élément de "A"; c.q.f.d.
Si "n" n'est pas nul alors
"r1" element de "A", ..., "rn" element de "A" entraine
"r1 =>k1 vide", ..., "rn =>kn vide" entraine,
d'après le principe d'indépendance des effacements,
"r1...rn =>k vide" avec "k = k1+...+kn" entraine
"r0" élément de "A"; c.q.f.d.

Démonstration: 2ème partie. Montrons que "A" est inclus dans tout ensemble "E" d'arbres qui satisfait les implications logiques, c'est à dire, que pour tout entier "i" et tout arbre "r"
"r =>i vide" entraine "r" élément de "E".
La proposition est vraie pour "i=1" car
"r =>1 vide" entraine "r => vide" entraine
"r" élément de "E"; c.q.f.d.
Supposons la proposition vraie pour "j<i" et montrons qu'elle est vraie pour "i":
"r =>i vide" et "i>1" entraine
"r => r1...rn =>i-1 vide" entraine,
d'après le principe d'indépendance,
"r => r1...rn", "r1 =>k1 vide", ..., "rn =>kn vide" avec
"k1<i",...,"kn<i", entraine,
d'après l'hypothèse de récurrence,
"r => r1...rn", "r1" élément de "E", ..., "rn" élément de "E"
entraine
"r" élément de "E"; c.q.f.d.

PRINCIPE DE GENERALISATION: Soit "i" un entier (non negatif), soit "t1...tn" une suite (eventuellement vide) de termes, soit "S" un système et soit "X" une affectation sylvestre des variables contenues dans "(t1...tn,S)", alors:

"X" est solution sylvestre de "S", et "t1(X)...tn(X) =>i vide" ssi:
   Il existe un système "T" avec:
   "(t1...tn,S) ->i (vide,T)" et "X" solution sylvestre de T.

Démonstration: 1ère partie. Montrons que le principe de généralisation est vrai pour "i=0":

"t1(X)...tn(X) =>0 vide" et "X" solution sylvestre de "S";

ssi

"t1...tn = vide" et "X" solution sylvestre de "S";

ssi

il existe un système "T" avec:
   "(t1...tn,S) ->0 (vide,T)" et "X" solution sylvestre de "S";
c.q.f.d.

Démonstration: 2ème partie. Supposons le principe de généralisation vrai pour "i" et montrons qu'il est vrai pour "i+1":

"t1(X)...tn(X) =>i+1 vide" et "X" solution sylvestre de "S";

ssi, d'apres la definition de "=>j",

il existe une règle particulière "r0 => r1...rm" avec:
   "t1(X) = r0",
   "r1...rm t2(X)...tn(X) =>i vide" et
   "X" solution de "S";

ssi, d'après la definition d'une règle particulière,

il existe une règle "s0 -> s1...sm, U" aux variables renommées de façon à n'en avoir aucune de commune avec "(t1...tn,S)" et il existe une affectation sylvestre "Y" avec:
   "Y" ne fait intervenir que les variables de "(s0...sm,U)",
   "Y" solution sylvestre de "U",
   "t1(X) = s0(Y)",
   "s1(Y)...sm(Y) t2(X)...tn(X) =>i vide" et
   "X" solution de "S";

ssi, du fait des variables intervenant dans "X" et "Y",

il existe une règle "s0 -> s1...sm, U" aux variables renommées de façon à n'en avoir aucune de commune avec "(t1...tn,S)" et il existe une affectation sylvestre "Y" avec:
   "Y" ne fait intervenir que les variables de "(s0...sm,U)",

"XuY" solution sylvestre de "U",
"t1(XuY) = s0(XuY)",
"s1(Y)...sm(Y) t2(X)...tn(X) =>i vide" et
"XuY" solution de "S";

ssi

il existe une règle "s0 -> s1...sm, U" aux variables renommées de façon à n'en avoir aucune de commune avec "(t1...tn,S)" et il existe une affectation sylvestre "Y" avec:
"Y" ne fait intervenir que les variables de "(s0...sm,U)",
"s1(XuY)...sm(XuY) t2(XuY)...tn(XuY) =>i vide" et
"XuY" solution de "SuUu{t1=s0}";

ssi, du fait que le principe est supposé vrai pour "i",

il existe une règle "s0 -> s1...sm, U" aux variables renommées de façon à n'en avoir aucune de commune avec "(t1...tn,S)" et il existe une affectation sylvestre "Y" avec:
"Y" ne fait intervenir que les variables de "(s0...sm,U)",
il existe un système "T" avec:
"(s1...sm t2...tn, SuUu{t1=s0}) ->i (vide, T)" et
"XuY" solution sylvestre de "T",

ssi, d'après la definition d'une solution sylvestre,

il existe une règle "s0 -> s1...sm, U" aux variables renommées de façon à n'en avoir aucune de commune avec "(t1...tn,S)" et il existe une affectation sylvestre "Y" avec:
"Y" ne fait intervenir que les variables de "(s0...sm,U)",
il existe un système "T" avec:
"(s1...sm t2...tn, SuUu{t1=s0}) ->i (vide, T)" et
"X" solution sylvestre de "T";

ssi, d'après la définition de "->j",

il existe un système "T" avec:
"(t1...tn,S) ->0 (vide,T)" et "X" solution sylvestre de "S";
c.q.f.d.

ANNEXE II: RESUME DE LA SYNTAXE

Voici, dans l'ordre alphabétique de leurs membres gauches, la liste de toutes les règles "hors contexte" utilisées pour définir des formules intervenant en Prolog. Nous rappelons que les conventions sont:

- 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 caractères 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.

```
<caractere>
   ::= <caractere special>
   ::= <lettre>
   ::= <chiffre>

<caractere special>
   ::= +
   ::= -
   ::= '
   ::= .
   ::= ,
   ::= ;
   ::= =
   ::= #
   ::= "
   ::= /
   ::= (
   ::= )
   ::= <
   ::= >
   ::= {
   ::= }
   ::= espace
   ::= <tout autre caractere disponible>

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

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

<commande>
   ::= <suite de termes> ;
   ::= <suite de termes> , <systeme> ;
```

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

<constante>
   ::= <identificateur>
   ::= <chaine>
   ::= <entier>
   ::= <reel>

<enonce>
   ::= <commentaire>
   ::= <regle>

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

<equation>
   ::= <terme> = <terme>

<equations et inequations>
   ::= <equation>
   ::= <inequation>
   ::= <equation> , <equations et inequations>
   ::= <inequation> , <equations et inequations>

<exposant>
   ::= <vide>
   ::= e <entier>
   ::= e <signe> <entier>

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

<inequation>
   ::= <terme> # <terme>

<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, mais differente de celle d'un terme>


<programme>
   ::= ;
   ::= <enonce> <programme>

<reel>
   ::= <signe> <entier> . <suite de chiffres> <exposant>

<regle>
   ::= <terme> - > <suite de termes> ;
   ::= <terme> - > <suite de termes> , <systeme> ;

<signe>
   ::= +
   ::= -

<suite de caracteres>
   ::= <vide>
   ::= <caractere> <suite de caracteres>

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

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

<systeme>
   ::= { }
   ::= { <equations et inequations> }

<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> )
```

```
   ..................................................

<terme strict>
   ::= <variable>
   ::= <constante>
   ::= ( <terme strict> . <terme strict> )
   ::= < >
   ::= < <terme strict> >
   ::= < <terme strict> , <terme strict> >
   ::= < <terme strict> , <terme strict> , <terme strict> >
   ..................................................

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

<vide>
   ::=
```

NOTE: Le caractère "double-guillemet" doit être double 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 être enlevés de tout endroit, sauf à l'intérieur des chaines et sauf si cela provoque la création de nouvelles constantes ou variables par agglutination d'anciennes.

REFERENCES

BATTANI G. et H. MELONI, (1973), Interpreteur du langage de programmation Prolog, rapport interne, Groupe Intelligence Artificielle, Université Aix-Marseille II, septembre.

COLMERAUER A., H. KANOUI, R. PASERO and P. ROUSSEL, (1973), Un système de communication homme-machine en francais, rapport de recherche CRI 72-18, Groupe Intelligence Artificielle, Université Aix-Marseille II, juin.

HUET G., (1976), Résolution d'équations dans des langages d'ordre 1,2,...,omega, thèse de doctorat d'état, Université Paris VII, septembre.

KANOUI H., (1982), PROLOG II, version 1, Manuel d'exemples, rapport interne, Groupe Intelligence Artificielle, Université Aix-Marseille II.

KOWALSKI R., and M. VAN EMDEN, (1976), The semantics of predicate logic as programming language, JACM, 23, no 4, pp. 733-743, octobre.

ROBINSON J.A., (1965), A machine-oriented logic based on the resolution principle, JACM 12, no 1, pp. 227-234, decembre.

ROUSSEL P., (1972), Définition et traitement de l'égalité formelle en démonstration automatique, thèse de 3eme cycle, Groupe Intelligence Artificielle, Université Aix-Marseille II.

ROUSSEL P., (1975), Prolog: manuel de réference et d'utilisation, rapport de recherche CNRS, Groupe Intelligence Artificielle, Université Aix-Marseille II.

VAN CANEGHEM M., (1982), PROLOG II, version 1, Manuel d'utilisation, rapport interne, Groupe Intelligence Artificielle, Université Aix-Marseille II.