Groupe d'Intelligence Aritficielle
U.E.R. de Luminy
Université d'Aix-Marseille

1973

RAPPORT DE D.E.A.

D'INFORMATIQUE APPLIQUEE

INTERPRETEUR DU LANGAGE DE PROGRAMMATION PROLOG

G. BATTANI
H. MELONI

TABLE DES MATIERES

Que le passage est difficile
de la résolution à l'exécution et
qu'il est ordinaire d'y échouer.

BOURDALOU

INTRODUCTION

Ce rapport présente l'interpréteur d'une nouvelle version du langage de programmation PROLOG. Ce langage, proche de la logique du premier ordre, utilise le calcul des prédicats et la démonstration automatique.

Après quelques rappels de démonstration automatique et une présentation rapide du langage, nous décrirons l'interpréteur.

I RAPPELS DE DEMONSTRATION AUTOMATIQUE

Nous utiliserons fréquemment les concepts de base propres à la logique du premier ordre et à la méthode de résolution. Nous rappelons donc brievement les notions de littéral, clause, unification et résolution.

I-1 Littéral

Un littéral est une formule atomique ou une négation de formule atomique.

exemples

PERE(x,y)
¬ PERE(x,y)

CONVENTION D'ECRITURE

La formule atomique du littéral est précédée des signes "+" ou "-" suivant qu'il s'agit d'une affirmation ou d'une négation. L'exemple précédent s'écrira donc

+PERE(x,y) ou +PERE(*x,*y)
-PERE(x,y) ou -PERE(*x,*y)

les variables sont précédées du signe "*".

I-2 Clause

Une clause est une disjonction de littéraux dont les variables sont quantifiées universellement.

exemple

$\forall x \forall y \forall z \forall t$ +cousin(*x,*y) v -fils(*x,*z)v -fils(*y,*t)v -frere(*z,*t)

CONVENTION D'ECRITURE

Les quantificateurs universels sont supprimés dans l'écriture de la clause, ainsi que les signes de disjonction.
L'exemple précédent s'écrira:

+cousin(*x,*y) -fils(*x,*z) -fils(*y,*t) -frere(*z,*t)

I-3 Unification

I-3-1 Ensemble de clauses

Un ensemble de clauses est une conjonction de clauses. Chaque variable étant quantifiée universellement dans une clause déterminée, les variables de deux clauses différentes n'ont aucun lien entre elles.

exemple

```
-cousin(*x, Antoine)
∧+cousin(*x,*y) -fils(*x,*z) -fils(*y,*t) -frere(*z,*t)
∧+fils(*x,*y) -pere(*y,*x)
∧+frere(*x,*y) -pere(*z,*x) -pere(*z,*y)
∧+pere(Paul, Jacques)
∧+pere(Paul, Jean)
∧+pere(Jacques,Antoine)
∧+pere(Jean,Pierre)
```

CONVENTION D'ECRITURE

Les signes de conjonction sont supprimés dans l'écriture d'un ensemble de clauses.

I-3-2 Substitutions -Instances

Une substitution est une suite de "transformations" $(*x_1 \rightarrow t_1, *x_2 \rightarrow t_2, \ldots, *x_n \rightarrow t_n)$ où $(*x_1, *x_2, \ldots *x_n)$ sont des variables et $(t_1, \ldots t_n)$ des termes.

Chaque transformation consiste à remplacer dans une expression E toute occurence de la variable $*x_i$ par le terme correspondant t_i.

La nouvelle expression ainsi obtenue est appelée instance de E.

Exemple

L'expression	+pere(*x,*y)
substituée par	(*x → Paul,*y → *z)
devient	+pere(Paul,*z)

I-3-3 Unification

On dit que deux termes ou formules atomiques sont unifiables s'il existe une substitution qui les rend égaux.

exemple

pere(Paul,*x) et pere(*y,*z) peuvent être rendus égaux par la substitution (*y → Paul,*x → *t,*z → *t)

I-4 Résolution

La résolution permet de générer une clause à partir de deux autres. La clause ainsi engendrée est appelée résolvante. Pour pouvoir générer une résolvante à partir de deux clauses il suffit que leurs premiers littéraux soient formés de formules atomiques unifiables précédées de signes opposés.

Exemples

La Résolution des deux clauses:

-cousin(*u,Antoine)
+cousin(*x,*y) -fils(*x,*z) -fils(*y,*t) -frere(*z,*t)

est possible grace à la substitution (*x → *u,*y → Antoine)
la résolvante obtenue est:

-fils(*u,*z) -fils(Antoine,*t) -frere(*z,*t)

cette nouvelle clause peut se résoudre avec la clause

+fils(*x,*y) -pere(*y,*x) en utilisant la substitution (*u → *x,*z → *y); la résolvante obtenue est:

-pere(*y,*x) -fils(Antoine,*t) -frere(*y,*t)

II DESCRIPTION DE PROLOG

II-1 Présentation d'un programme

Un programme PROLOG est formé d'une suite finie de clauses. On met en évidence une partition $(P_i)_i$ de cet ensemble de clauses de la manière suivante: chaque P_i est la suite des clauses commençant par un même prédicat précédé du même signe, ordonnée suivant l'ordre d'apparition des clauses dans le programme. Cette partition est unique.

Deux programmes qui sont formés des mêmes clauses et dont les partitions sont égales sont équivalents. Ils produisent les mêmes résultats à partir des mêmes données.

L'importance particulière de l'ordre des clauses dans une partie P_i est due aux stratégies de démonstration employées pour l'éxécution d'un programme.

Nous donnons l'exemple d'un programme dont les formes A et B sont équivalentes.
(Nous utilisons la syntaxe rudimentaire en notation préfixée proche de la représentation interne en laquelle seront traduit les programmes PROLOG utilisant une syntaxe plus riche.)

Exemple A

```
+boulot -cousin(Paul,*x)
+cousin(*x,*y) -fils(*x,*z) -fils(*y,*t) -frere(*z,*t)
+fils(*x,*y) -pere(*y,*x)
+frere(*x,*y) -pere(*z,*x) -pere(*z,*y)
+pere(Paul,Jacques)
+pere(Paul,Jean)
+pere(Jacques,Antoine)
+pere(Jean,Pierre)
```

Exemple B

```
+boulot -cousin(Paul,*x)
+frere(*x,*y) -pere(*z,*x) -pere(*z,*y)
+pere(Paul,Jacques)
+boulot -cousin(*x,Antoine)
+cousin(*x,*y) -fils(*x,*z) -fils(*y,*t) -frere(*z,*t)
+pere(Paul,Jean)
+fils(*x,*y) -pere(*y,*x)
+pere(Jacques,Antoine)
+pere(Jean,Pierre)
```

On obtient pour ces deux programmes la partition:

P_1 ={+pere(Paul,Jacques),+pere(Paul,Jean),+pere(Jacques,Antoine),
+pere(Jean,Pierre)}

P_2 ={+cousin(*x,*y)-fils(*x,*z)-fils(*y,*t)-frere(*z,*t)}

P_3 ={+fils(*x,*y) -pere(*y,*x)}

P_4 ={+boulot -cousin(Paul,*x),+boulot -cousin(*x,Antoine)}

P_5 ={+frere(*x,*y) -pere(*z,*x) -pere(*z,*y)}

(l'ordre des parties entr'elles n'a aucune importance.)

II-2 Déroulement d'un programme

Le déroulement d'un programme consiste à engendrer la clause vide de toutes les manières possibles à partir d'une clause donnée (-boulot) qui conditionne le départ de toute démonstration (déroulement).

Une démonstration ne peut donc être exécutée que s'il existe une partie P_i de la partition définie en II-1, qui contienne au moins une clause dont le premier littéral est +boulot.
Une démonstration sera la construction successive de résolvantes à partir d'une clause donnée et de l'axisme qui lui correspond.

Au départ la clause donnée est -boulot, elle est ajoutée par l'interpréteur à tout programme et doit être résolue avec le premier axiome commençant par +boulot. Les deux premiers littéraux étant unifiables,on forme la résolvante en juxtaposant dans l'ordre les littéraux restant de l'axiome avec ceux restant de la clause donnée (ancienne résolvante). On recommence ensuite avec la nouvelle résolvante et le premier axiome qui lui correspond etc...

Lors d'une démonstration, le processus de construction de la résolvante peut être arrêté pour trois raisons: la résolvante engendrée est la clause vide, l'unification des deux premiers littéraux est impossible, il n'y a pas d'axiome pouvant se résoudre avec la clause "donnée" (résolvante précédente). Dans chacun de ces cas il convient de continuer différemment la démonstration.

a) La résolvante engendrée est la clause vide.

Cela signifie que l'on a engendré la clause vide à partir de la clause donnée -boulot. Cependant lors des résolutions successives

nous n'avons utilisé que le premier choix dans la suite des axiomes d'une partie P_j. Nous devons donc en "remontant" chercher le cas le plus récent où la partie n'a pas été épuisée. S'il en existe une on reprend la résolution de l'ancienne clause "donnée" avec l'axiome suivant de la partie P_j. Ce nouveau choix est éliminé et seuls les axiomes suivant pourront être utilisés lors d'un prochain retour(Backtracking).

b) L'unification des deux premiers littéraux est impossible

Il faut dans ce cas essayer de résoudre la clause "donnée" avec l'axiome suivant celui déjà utilisé (s'il en reste un) sinon nous devons "remonter" comme précédemment.

c)Aucun axiome ne peut se résoudre avec la clause "donnée"

Il faut dans ce cas "remonter" directement comme précédemment. Si lors d'une "remontée" tous les P_j utilisés ont été épuisés, la démonstration s'arrête car tous les chemins permettant d'engendrer la clause vide ont été parcourus.

Exemple, déroulement du programme donné en II-1

La première résolvante R_1 est -boulot
Le premier axiome correspondant à R_1 est +boulot-cousin(Paul,*x)
L'unification est possible (-boulot et +boulot) sans substitution.
La résolvante R_2 engendrée est -cousin(Paul,*x)
Le premier axiome correspondant à R_2 est +cousin(*u,*y)-fils(*u,*z)
-fils(*y,*t)-frere(*z,*t)
L'unification est possible avec la substitution (*u→Paul,*y→*x)
La résolvante R_3 obtenue est
-fils(Paul,*z) -fils(*x,*t) -frere(*z,*t)
L'axiome correspondant est
+fils(*x,*y) -pere(*y,*x)
on obtient avec la substitution (*x→Paul,*y→*z)
la résolvante R_4
-pere(*z,Paul) -fils(*x,*t) -frere(*z,*t)
Le premier axiome correspondant est
+pere(Paul,Jacques)
l'unification des premiers littéraux est impossible on est donc dans le cas (b).
L'axiome suivant est
+pere(Paul,Jean)
L'unification des premiers littéraux est impossible ainsi que pour les deux choix suivants. On "remonte" et on cherche un autre choix pour R_3. Il n'en existe pas de même que pour R_2.

Le choix suivant pour R_1 est l'axiome:
+boulot -cousin(*x,Antoine)
La résolvante R_5 obtenue est:
-cousin(*x,Antoine)
Puis successivement en utilisant seulement les premiers choix les résolvantes
R_6: -fils(*x,*z) -fils(Antoine,*t) -frere(*z,*t)
R_7: -pere(*z,*x) -fils(Antoine,*t) -frere(*z,*t)
R_8: -fils(Antoine,*t) -frere(Paul,*t)
R_9: -pere(*t,Antoine) -frere(Paul,*t)

Les deux premiers choix des axiomes correspondant à R_9 ne permettent pas l'unification. Avec le 3ème choix on obtient
R_{10}: -frere(Paul,Jacques)
R_{11}: -pere(*z,Paul) -pere(*z,Jacques)

Les quatre choix possibles des axiomes correspondant à R_{11} ne permettent pas l'unification. On "remonte" jusqu'à R_9 où il reste un choix possible, mais celui-ci ne permet pas l'unification. On remonte alors en R_7 où il reste 3 choix dont 2 mènent encore à des impasses. L'unification avec le 3ème produit la résolvante
R_{12}: -fils(Antoine,*t) -frere(Jean,*t)
R_{13}: -pere(*t,Antoine) -frere(Jean,*t)

Après deux choix menant à une impasse l'unification avec le 2ème produit la résolvante:
R_{14}: - frere(Jean,Jacques)
R_{15}: -pere(*z,Jean) -pere(*z,*Jacques)

Après un choix menant à une impasse l'unification avec le 2ème produit la résolvante:
R_{16}: -pere(Paul,Jacques)
et la résolution avec le 1er choix produit la résolvante vide (cas a).

On continue la démonstration jusqu'à ce que tous les choix soient épuisés. L'exécution du programme est alors terminée.

II-3 Prédicats évaluables

A l'issue de l'exécution de ce programme on se rend compte que l'on n'a pas accès extérieurement aux résultats. On peut se demander par exemple si l'on a réussi à engendrer la clause vide, à l'aide de quelles substitutions etc...

Pour cela nous utiliserons des "prédicats évaluables" "d'entrée sortie" ainsi que d'autre types de "prédicats évaluables" (édition, modification des remontées...)

Ces prédicats évaluables font du démonstrateur un langage de programmation.

Ce qui distingue les prédicats évaluables des autres prédicats est la façon particulière de les "unifier". Ils ne sont jamais résolus avec des axiomes; et au lieu d'être supprimés par résolution ils sont "évalués". Trois cas peuvent se présenter:

1)L'évaluation est VRAI
2)L'évaluation est FAUX
3)L'évaluation est impossible.

L'évaluation du prédicat conduit à traiter le littéral correspondant de trois manières différentes:

1) +VRAI et -FAUX conduisent à une impasse.

L'impasse est analogue au cas b du paragraphe II-2 où l'unification est impossible.

2) -VRAI et +FAUX conduisent à la suppression du littéral.

Ce cas est analogue à celui où l'unification est possible.

3) L'évaluation impossible conduit à une erreur.

Cela consiste à remplacer le littéral impossible à évaluer par le littéral -erreur(*x), où la variable x est substituée par le prédicat erroné.

L'évaluation des prédicats est particulière à chacun d'eux. Leurs variables pourront être éventuellement substituées au cours de l'évaluation en fonction des règles que nous donnons ci-dessous.

II-3-1 Prédicats d'entrée sortie

a) Prédicat LU

se présente sous la forme LU(<terme>)

Ce prédicat provoque la lecture d'un caractère sur une unité d'entrée et unifie <terme> avec le caractère lu.
Il est évalué à VRAI si l'unification est possible à FAUX sinon. Si la lecture sur le fichier d'entrée est impossible (fin de fichier) il est alors impossible à évaluer.

b) Prédicat ECRIT

<Prédicat écrit> ::= ECRIT(<terme>)

Ce prédicat n'est jamais évalué à FAUX. Il est évalué à VRAI si ‹terme› est un caractère et provoque l'écriture du caractère sur une unité de sortie. Il est impossible à évaluer si ‹terme› n'est pas un caractère.

c)Prédicat LIGNE

<Prédicat ligne> ::= LIGNE

Ce prédicat est toujours évalué à VRAI. Si le littéral est supprimé, il provoque un passage à la ligne sur l'imprimante.

II-3-2 Prédicats d'édition

a) Prédicat AJOUT

<Prédicat ajout> ::= AJOUT(<clause>)
<clause> ::= .(<littéral>,<suite de littéraux>)
<littéral> ::= +(<terme>)|-(<terme>)
<suite de littéraux> ::= NIL|.(<littéral>,<suite de littéraux>)

b) Prédicat ECRIT

<Prédicat écrit> ::= ECRIT(<terme>)

Ce prédicat n'est jamais évalué à FAUX. Il est évalué à VRAI si ‹terme› est un caractère et provoque l'écriture du caractère sur une unité de sortie. Il est impossible à évaluer si ‹terme› n'est pas un caractère.

c)Prédicat LIGNE

<Prédicat ligne> ::= LIGNE

Ce prédicat est toujours évalué à VRAI. Si le littéral est supprimé, il provoque un passage à la ligne sur l'imprimante.

II-3-2 Prédicats d'édition

a) Prédicat AJOUT

<Prédicat ajout> ::= AJOUT(<clause>)
<clause> ::= .(<littéral>,<suite de littéraux>)
<littéral> ::= +(<terme>)|-(<terme>)
<suite de littéraux> ::= NIL|.(<littéral>,<suite de littéraux>)

II-3-3 Prédicats de stratégie

a) Prédicat slash o

<prédicat slasho> ::= /

Ce prédicat est toujours VRAI.
Lorsque le littéral correspondant est supprimé, il provoque la suppression de tous les choix éventuels d'axiomes pour les littéraux le précédant dans la clause.
Lors d'une "remontée", aucun autre essai d'unification sur ces littéraux ne sera tenté.
Le slash permet en particulier de rendre certains programmes entièrement déterministes.

b) Prédicat slash 1

<prédicat slash1> ::= /(<littéral>)

Ce prédicat est impossible à évaluer si la syntaxe de <littéral> est erronée.
Il est évalué à VRAI s'il existe un de ses "ancêtres" A qui soit unifiable avec son argument (<littéral>). L'unification est alors effectuée si le littéral correspondant est supprimé.
Il est évalué à FAUX s'il n'existe pas"d'ancêtre" unifiable avec littéral .
(l'ancêtre A d'un littéral L est le littéral de la résolvante qui a été unifié avec le premier littéral de la clause contenant L:
Tous les ancêtre de A sont également des ancêtres pour L.
Seuls les littéraux de la résolvante ont donc des ancêtres; les premiers littéraux de chaque clauses n'ont jamais d'ancêtres.)

Si le littéral correspondant est supprimé, il provoque la suppression de tous les choix éventuels d'axiomes (pour tous les littéraux) qui seraient utilisés lors d'une "remontée" jusqu'à l'ancêtre A.

c)Prédicat ETAT

<prédicat état> ::= ETAT(<terme>)

Il est toujours possible d'évaluer ce prédicat.
Définissons d'abord l'état de la démonstration. C'est un terme dont la syntaxe est la suivante:

<terme état> ::= .(<suite de littéraux>,<suite état>)
<suite état> ::= NIL /<terme état>

La première suite de littéraux L_1 est constituée par l'ancêtre R_1 le plus récent suivit des littéraux formant le morceau de résolvante ajouté au moment de la suppréssion de R_1.

La suite L_n des littéraux est constituée de la même manière par R_n (ancêtre de R_{n-1}) suivit des littéraux formant le morceau de résolvante ajouté au moment de la suppression de R_n.

Le terme ainsi constitué (terme état) est unifié avec <terme>. Si l'unification est possible le prédicat est évalué à VRAI sinon on l'évalue à FAUX.

L'unification est effectuée si le littéral correspondant que l'on peut récupérerdans une variable, aulieu la modifier.

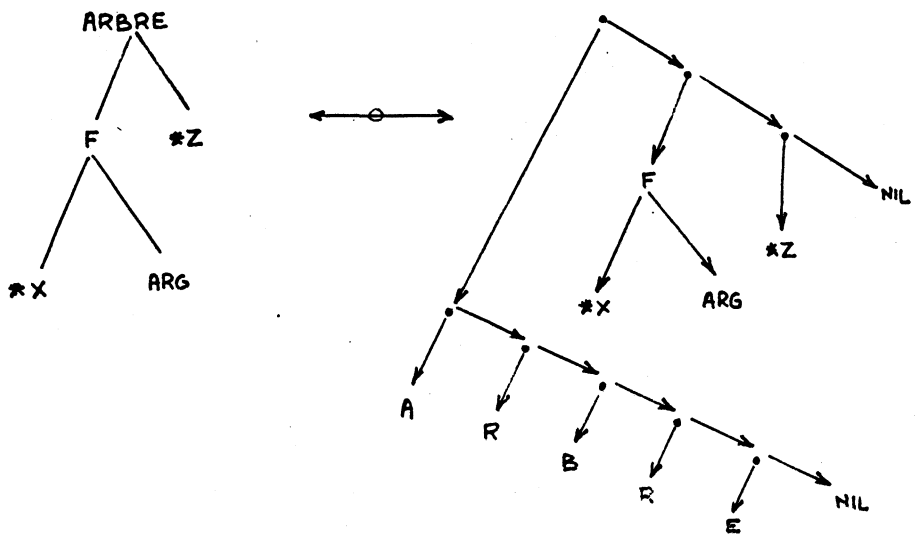
II-3-4 Prédicat UNIV

Ce prédicat permet de construire dynamiquement des arbres ou bien de les décomposer.

```
<Prédicat univ> ::= UNIV(<variable>,<arbre décomposé>)|UNIV(<arbre>,<terme>)
<arbre >::= <unité><arguments>
<arguments> ::=.|(<arbre><suite arbre>)
<suite arbre> ::=)|,<arbre><suite arbre>
<unité> ::=<caractère><reste unité>
<reste unité> ::= |<unité>
<arbre décomposé> ::= .(<unité décomposée>,<suite de termes>)
<unité décomposée> ::=.(<caractère>,<suite caractères>)
<suite de termes> ::= .(<terme>,<suite de termes>)| NIL
<suite caractères> ::= .(<caractère>,<suite caractères>)| NIL
```

A tout arbre correspond biunivoquement un arbre décomposé.

exemple



L'évaluation de ce prédicat se fait de deux manières différentes suivant que le premier argument est une variable ou non.

a) Le premier argument est une variable

Si la syntaxe de l'arbre décomposé est incorrecte l'évaluation est impossible.
Le prédicat est toujours évalué à VRAI.
La suppression du littéral correspondant entraine l'unification de la variable avec l'arbre associé à l'arbre décomposé.

b) Le premier argument est un arbre.

L'évaluation est toujours possible.
Il est évalué à VRAI si l'unification du <terme> avec l'arbre décomposé associé à <arbre> est possible. Il est évalué à FAUX dans le cas contraire.
La suppression du littéral correspondant entraine l'unification de <terme> avec l'arbre décomposé associé à <arbre>.

II-3-5 Prédicat SAUVE

Il peut être utile dans un programme très long de disposer d'un prédicat qui permette (lorsqu'on est amené à le supprimer) d'arrêter le processus de démonstration afin de reprendre le travail plus tard.
Pour cela il faut que le contenu de toute la mémoire soit "sauvé" sur des périphériques.
Lors de la remise en mémoire du programme il y a rechargement et le travail peut s'exécuter à partir de l'endroit où il avait été interrompu.

<prédicat sauve> ::= SAUVE

Ce prédicat est toujours évaluable et évalué à VRAI.
La suppression du littéral correspondant entraîne l'arrêt de l'exécution du programme ainsi que le chargement sur un périphérique (choisi par assignation par le programmeur) du contenu de la mémoire.
Lors de la reprise du travail, celui-ci débutera avec la résolvante obtenue après suppression du littéral -SAUVE.

II-3-6 Prédicats divers

Pour certaines utilisations il est souvent utile et parfois nécessaire de disposer d'autres prédicats évaluables.

Le traitement des entiers nécessiterait un grand nombre d'axiomes. Une simple addition serait pratiquement impossible à faire.

Il est souvent intéressant de savoir rapidement si une variable a été substituée ou si un terme ne comprend aucune variable.

Il est utile également de savoir si un caractère substitué à une variable est un chiffre, une lettre ou un caractère non alphanumérique. (on pourrait en fait se passer de ces prédicats comme de certains autres mais l'emploi du langage serait alors rendu pratiquement impossible).

Nous donnons ci-dessous une liste non exhaustive de ces prédicats évaluables.

a) Prédicats de traitement des entiers

<Prédicat plus> ::= PLUS (<terme>,<terme>,<terme>)
<Prédicat moins> ::= MOINS (<terme>,<terme>,<terme>)
etc...

Pour ces prédicats, l'évaluation est impossible les deux premiers arguments ne sont pas des nombres entiers ou bien si le 3ème argument n'est ni un entier ni une variable.

Ils réalisent l'opération (+,-,etc) sur les deux premiers arguments et unifient le résultat obtenu avec le 3ème argument.

Si l'unification est possible le prédicat est évalué à VRAI. (c'est toujours le cas si le 3ème argument est une variable.)

Si l'unification est impossible le prédicat est évalué à FAUX.

Lors de la suppression du littéral, l'unification est réalisée.

b) Prédicat de comparaison des caractères

<prédicat lettre> ::= LETTRE (<terme>)
<prédicat chiffre> ::= CHIFFRE (<terme>)
etc...

L'évaluation est impossible si terme n'est pas un caractère.

Ils sont évalués à VRAI si le caractère est une lettre ou un chiffre (respectivement pour LETTRE et CHIFFRE) à FAUX sinon.

III L'INTERPRETEUR

III-1 FONCTIONNEMENT - GENERALITES

L'interpréteur est écrit en FORTRAN. Ce langage étant encore le plus répandu, cela permet d'adapter l'interpréteur sur toutes les machines possèdant un compilateur de FORTRAN.

Avant d'être traité par le démonstrateur un programme S écrit en PROLOG est soumi à une analyse syntaxique puis codé en mémoire (sous forme exécutable) par un autre programme PROLOG A (analyseur).

L'analyseur A a d'abord été lui-même analysé puis codé par un programme FORTRAN d'initialisation I.

L'exécution du programme A provoque l'analyse syntaxique, puis l"ajout" des clauses du programme source S, et enfin son exécution.

Donc le démonstrateur exécute toujours le même programme A.

Avant d'exécuter le programme A, l'interpréteur initialise ses zones de travail (lecture sur disque). Cette initialisation comporte en particulier le chargement du programme A.

Indépendamment du démonstrateur, il existe donc un programme I d'initialisation écrit en FORTRAN qui code l'analyseur écrit en PROLOG (la syntaxe de cet analyseur étant rudimentaire). Cela permet en particulier à chacun des utilisateurs d'écrire s'il le désire un analyseur personnel. Celui-ci sera chargé sur disque par le programme I à l'emplacement qu'utilise le démonstrateur pour initialiser ses zones de travail.

Méthode employée pour la résolution des clauses

La résolution de deux clauses produit une résolvante.

Pour chaque résolution il faut donc créer une résolvante, d'autre part il faut pouvoir remonter et retrouver la résolvante générée à l'étape précédente. Cela implique de garder en mémoire toutes les résolvantes éventuellement nécessaires lors d'une remontée.

Le problème est de garder toutes les informations sous une forme occupant peu de place en mémoire et d'une interprétation aisée.

Pour résoudre ce problème il nous faût considérer la manière dont se forment les résolvantes. Deux étapes essentielles interviennent dans leur construction:

a) l'unification des premiers littéraux de la résolvante et de l'axiome associé,

b) la formation de la résolvante obtenue à partir de la résolvante précédente et de l'axiome.

Lors de l'étape (a) l'information utile à conserver est la substitution utilisée pour l'unification des littéraux.

Lors de l'étape (b) l'information à stocker est la résolvante formée des littéraux restant après l'unification. Les résolvantes successives sont donc formées de littéraux figurant déjà dans les axiomes, mais dont les variables ont pu être modifiées, parfois de façon très complexe.

Il suffit donc pour connaitre une résolvante de savoir de quels littéraux elle est formée et quelles sont les substitutions qu'ont subies ces littéraux. Il faut mémoriser pour cela un "squelette" de la résolvante et l'instance associée aux variables de ce squelette.

La résolvante évolue donc comme une pile dont seul le sommet est modifié. Il est donc naturel de conserver les informations le concernant (adresses des littéraux, adresses des substitutions) dans une pile (tableau PILE).

Chaque identificateur est utilisé de nombreuses fois dans les axiomes; il est commode de représenter chacune de ces utilisations d'un même prédicat ou d'une même fonction par un seul pointeur. Ce pointeur indique une zone de longueur déterminée, contenant toutes les informations relatives à un identificateur. L'ensemble de ces zones forme un dictionnaire construit dynamiquement dans le tableau (TAB).

Lors de la reconnaissance d'un identificateur, afin d'accèder rapidement à la zone du dictionnaire qui lui correspond, nous utilisons un système de clés.

On calcule pour chaque identificateur I, la clé qui lui est associée (plusieurs identificateurs peuvent avoir la même clé), puis par l'intermédiaire d'un tableau de clés (mémoire associative) on obtient l'adresse de la liste des zones relatives aux identificateurs de clé C.

Au lieu d'utiliser systématiquement de nombreuses piles, dont certaines peuvent être saturés alors que d'autres seront pratiquement vides, nous préférons employer seulement deux piles qui contiennent toutes les autres sous forme de n-uplets dont les premiers éléments renferment les informations, le dernier donne l'adresse dans la pile du n-uplet suivant (structure de liste).

Le programme d'initialisation et celui du démonstrateur sont découpés au maximum en sous-programmes.

Cela permet de comprendre aisément les mécanismes utilisés. Chacun de ces sous-programmes représente en fait une phase particulière de la démonstration.

Chaque prédicat évaluable aura également un sous-programme associé.

Nous décrivons successivement la manière dont est organisée la mémoire, le programme d'initialisation, le programme du démonstrateur et les sous-programmes correspondants aux prédicats évaluables.

III-2 DESCRIPTION DE L'ORGANISATION DE LA MEMOIRE

III-2-1 Partage des zones de la mémoire

La mémoire est partagée entre diverses zones dont la taille de certaines (d'entre elles) est fonction de la capacité mémoire de la machine utilisée.

Chacune de ces zones correspond à un tableau (DIMENSION du langage FORTRAN). Chacun de ces tableaux sera défini en commun (COMMON) pour tous les sous-programmes FORTRAN.

Il y a deux tableaux qui à eux seuls occuppent pratiquement la totalité de la mémoire. Ce sont les tableaux TAB et PILE.

Le tableau TAB est à deux entrées (une à chacune de ses extrémités). Les sommets des deux zones sont repérés par deux pointeurs qui localisent la première case non encore utilisée. Il s'agit des pointeurs ITAB et IVAR.

La première partie du tableau TAB contient tout ce qui est codé (dictionnaire des prédicats et fonctions, clauses).

La seconde partie renferme les "substitutions".

Le tableau PILE a également deux entrées et les pointeurs utilisés de la même manière que dans TAB sont IBAC et IREC.

La première partie de PILE sert à mémoriser les pointeurs donnant accès aux morceaux successifs de la résolvante ainsi que les choix d'axiomes.

La seconde partie est utilisée comme pile de récursivité.

Pour ces deux tableaux il y a constamment controle de non débordement. En effet il peut arriver si le programme PROLOG est trop important que les pointeurs ITAB et IVAR deviennent égaux, il faut alors arrêter le programme car sa continuation entrainerait des erreurs. Il en est de même pour les pointeurs IBAC et IREC.

Les autres tableaux ont des dimensions fixes (ils ne dépendent pas de la machine utilisée).

Il s'agit des tableaux CLES,CALCUL,ENTREE et SORTIE.

CLES est un tableau de dimension 800; il permet d'accéder directement à une liste de prédicats et fonctions codés dans le dictionnaire. Il est parcouru à l'aide du pointeur ICLES.

CALCUL est un tableau de dimension 256 qui contient le code d'accès des caractères dans le dictionnaire. Il permet d'éviter le calcul constant de la place du caractère lors de la lecture.

ENTREE est un tableau de dimension 80 utilisé comme buffer d'entrée.

SORTIE est un tableau de dimension 132 utilisé comme buffer de sortie.

Les dimensions de ces derniers tableaux peuvent être aisément modifiées.

III-2-2 Codage interne

Pour qu'un programme puisse se dérouler, il faut qu'il soit entré en mémoire. Cela suppose un codage de toutes les informations nécessaires à la reconnaissance des prédicats et des fonctions ainsi que le codage des axiomes sous une forme facilement utilisable.

Les clauses représentées en mémoire sont de deux types. Il y a d'une part les axiomes qui sont codés linéairement et d'autre part les résolvantes dont la structure est plus complexe.

Nous donnons donc ci-dessous la manière dont sont codées toutes les informations.

a) Dictionnaire des prédicats et fonctions

Les prédicats et les fonctions sont entrés dans le dictionnaire soit par le programme d'initialisation, soit dynamiquement par le prédicat évaluable UNIV.

Le dictionnaire comporte une séquence de 5 informations pour tous les identificateurs autres que les caractères (identificateurs à une seule lettre ou chiffre ou caractère non alphanumérique et de poids nul).

La séquence est atteinte soit directement à l'aide d'une clé, soit par l'intermédiaire de la dernière information d'une séquence de même clé. (Les identificateurs ayant la même clé d'accès au dictionnaire sont listés entre eux).

Si α est l'adresse de la séquence, nous atteignons les 5 informations qu'elle contient en prenant l'une des 5 adresses α+POIDS, α+PRED, α+NOM, α+AXTER, α+NEXT.

Les constantes POIDS,PRED,NOM,AXTER,NEXT étant initialisées respectivement à 0,1,2,3,4.

Pour les identificateurs de type caractère, l'accès ne se fait pas par clé mais directement par calcul sur le code machine du caractère. En effet ces identificateurs étant tous entrés en mémoire à l'initialisation, il n'y a donc pas de problème de recherche.

Ils sont codés un peu différemment dans le dictionnaire. Au lieu d'une séquence de 5 informations, il leur est associé une séquence de 3 informations. Les deux premieres étant les mêmes que pour les autres identificateurs (α+POIDS,α+PRED) et la 3ème contient le code machine du caractère (cela permet une écriture rapide en sortie).

a-1 Le poids

TAB(α+POIDS) est la valeur du nombre d'arguments de l'identificateur.

a-2 Adresse éventuelle des clauses

TAB(α+PRED) est égal à zéro s'il n'existe pas de clause dont le premier littéral a pour prédicat celui d'adresse α.

TAB(α+PRED) est dans le cas contraire l'adresse β d'une séquence de deux informations. Ces informations sont atteintes par les adresses β+ACPL, β+ACMO (ACPL=0, ACMO=1).

L'une au moins des deux informations contenues n'est pas nulle. TAB(β+ACPL) et TAB(β+ACMO) donnent si leur valeur n'est pas nulle l'adresse γ d'une séquence de deux informations atteintes par γ+CLO et γ+SUIV (CLO=0, SUIV=1).

TAB(γ+CLO) donne l'adresse de la première clause dont le premier littéral a pour prédicat celui d'adresse α, le signe du littéral étant déterminé par le fait que γ est TAB(β+ACPL) (+), ou γ est TAB(β+ACMO) pour (-).

TAB(γ+SUIV) donne si ce n'est pas zéro l'adresse de la clause suivante dont le premier littéral a pour prédicat celui d'adresse α et de même signe que la précédente.

Exemple (les clauses suivantes appartiennent à la même partie P_i) voir II-1.

ν : adresse le la clause, ρ : adresse éventuelle de la clause suivante de la partie P_i.

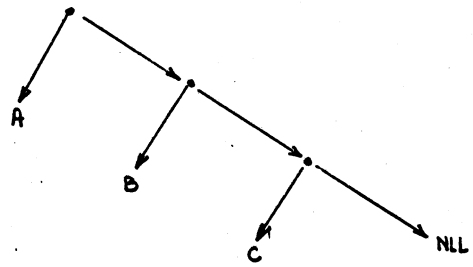
a-3 Adresse du nom

TAB(α+NOM) donne l'adresse δ d'une séquence de longueur variable dont les informations sont utilisées pour reconnaitre la chaine de caractère qui représente un indentificateur.

Le nom est représenté sous forme de "peigne" dont la syntaxe est la suivante:

<nom> ::= .(<caractère>,<suite nom>)
<suite nom> ::= NIL|<nom>

Par exemple l'identificateur ABC sera donc d'abord transformé en l'arbre:

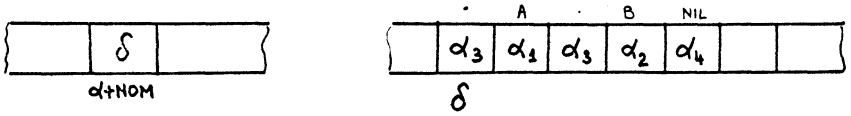


Cette structure d'arbre se parcours aisément.
Ce "peigne" est codé linéairement en mémoire dans la séquence variable d'adresse δ. Le (.) est un prédicat à deux arguments (différent du caractère), il a évidemment lui-même une adresse dans le dictionnaire. NIL est un prédicat sans argument.

Les informations contenues dans la séquence d'adresse δ sont les adresses successives dans le dictionnaire des différents identificateurs qui composent l'arbre du "peigne" (.,caractères,NIL).

Exemple

L'identification AB sera donc codé (si α_1 est l'adresse dans le dictionnaire de A, α_2 celle de B,α_3 celle du (.) à deux arguments et α_4 celle de NIL) comme suit:



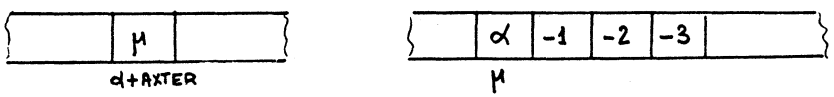
a-5 Adresse du terme représentant l'identificateur

TAB(α+AXTER) donne l'adresse μ du terme codé en mémoire qui représente l'identificateur. Ce terme comporte autant de variables que l'identificateur possède d'arguments.

Les variables sont codées en mémoire par des entiers négatifs -1,-2,-3,..etc...

Exemple

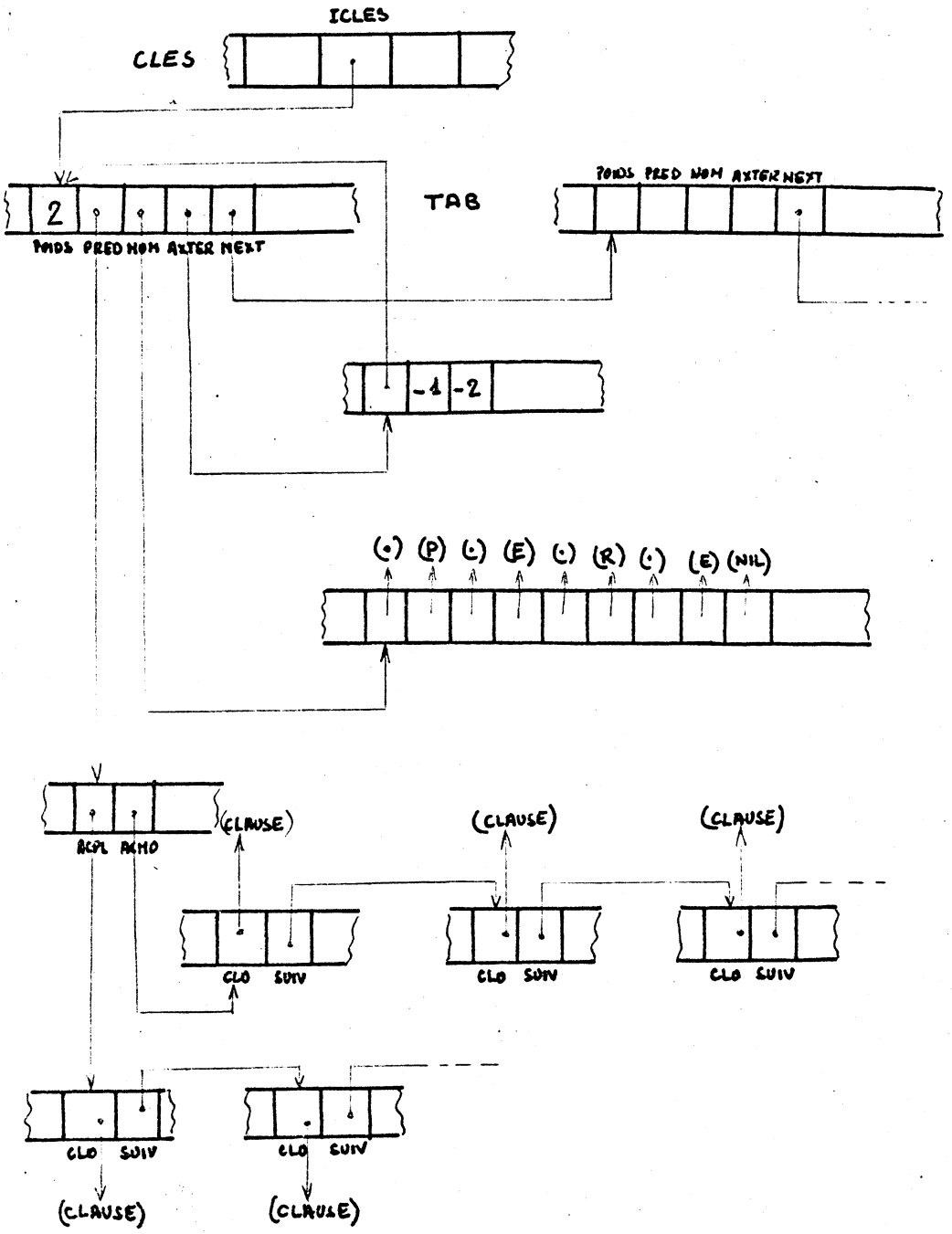
Si αest l'adresse en mémoire d'un identificateur à 3 arguments nous avons:



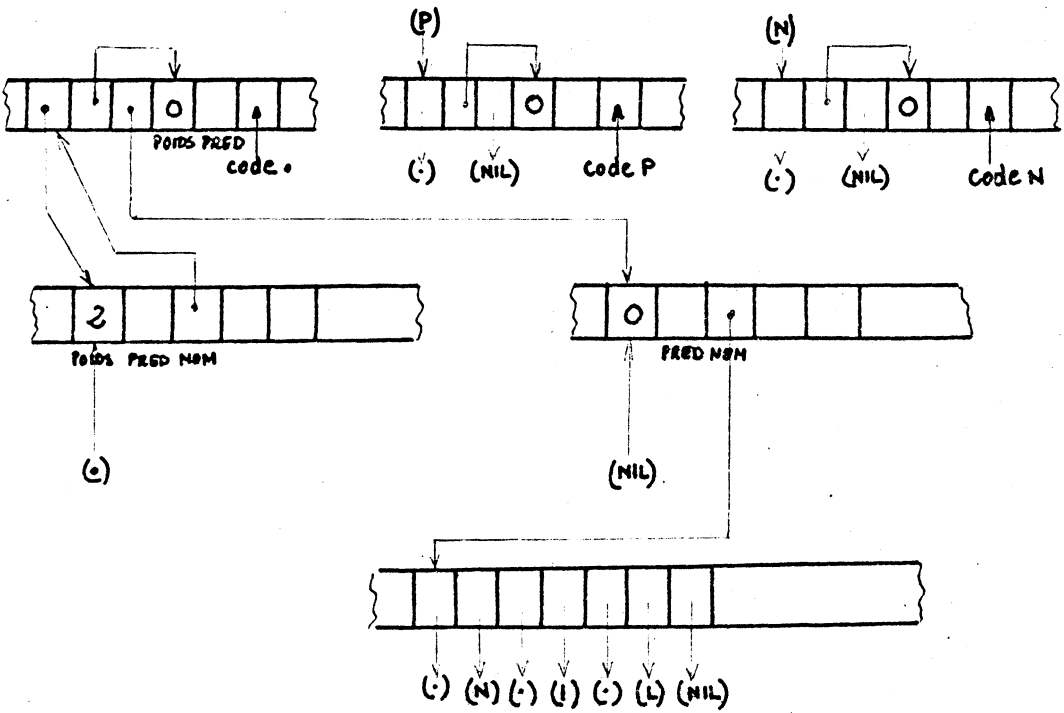
a-6 Adresse du suivant

TAB(α+NEXT) donne s'il est différent de zéro l'adresse α' d'entrée dans le dictionnaire (séquence de 5) pour l'identificateur suivant qui possède la même clé d'accès.

En résumé, nous obtenons à partir d'une clé les renseignements suivants: (identificateur PERE à deux arguments)



Les prédicats . et NIL sont également codés dans le dictionnaire d'une façon permanente de même que les caractères. Leur codage est le suivant:

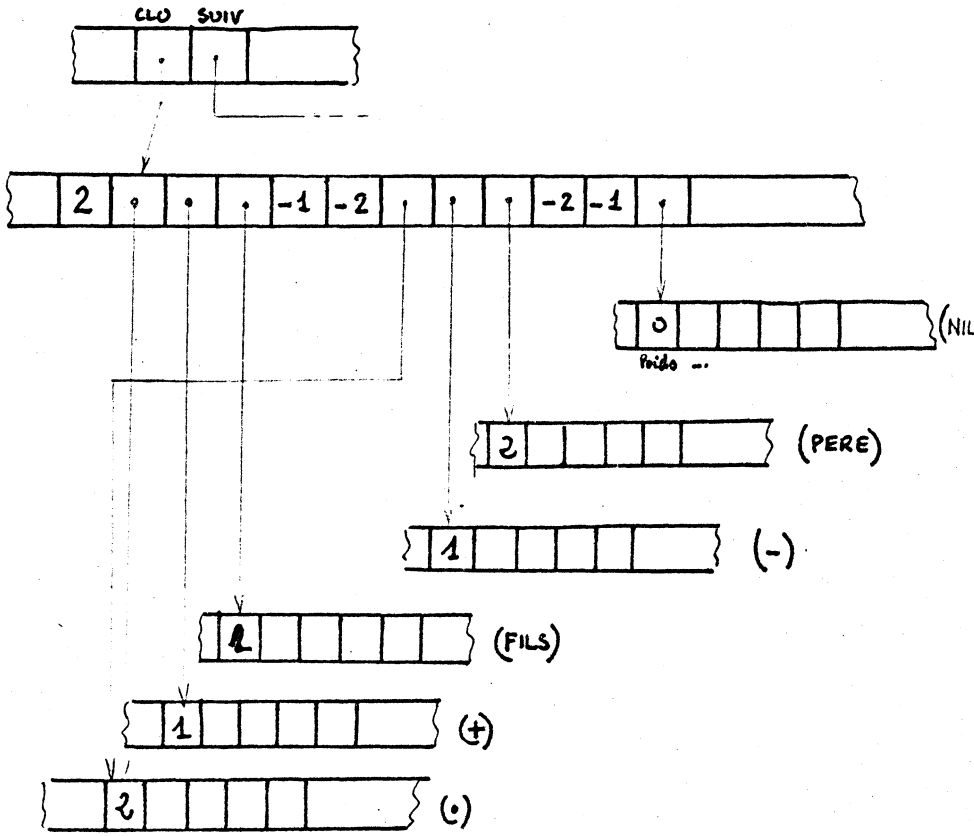


b) Codage des axiomes

Une clause axiome sera l'arbre défini en II-3-2 (a) codé linéairement et précédé d'un entier représentant le nombre de variables de la clause. Les variables sont codées suivant leur ordre d'apparition dans la clause par des entiers relatifs -1,-2,...etc.

Exemple codage de la clause:

+FILS(*x,*y) -PERE(*y,*x)



c)Codage de la résolvante

Les résolvantes sont codées dans le tableau PILE où se trouve également la pile de remontée (pile de backtraking) car les deux piles varient de la même manière.

Nous décrivons sa structure générale, puis comment à partir de cette pile on peut retrouver la résolvante.

Cette pile est formée de séquences de 4 adresses séparées éventuellement par des suites d'entiers négatifs.

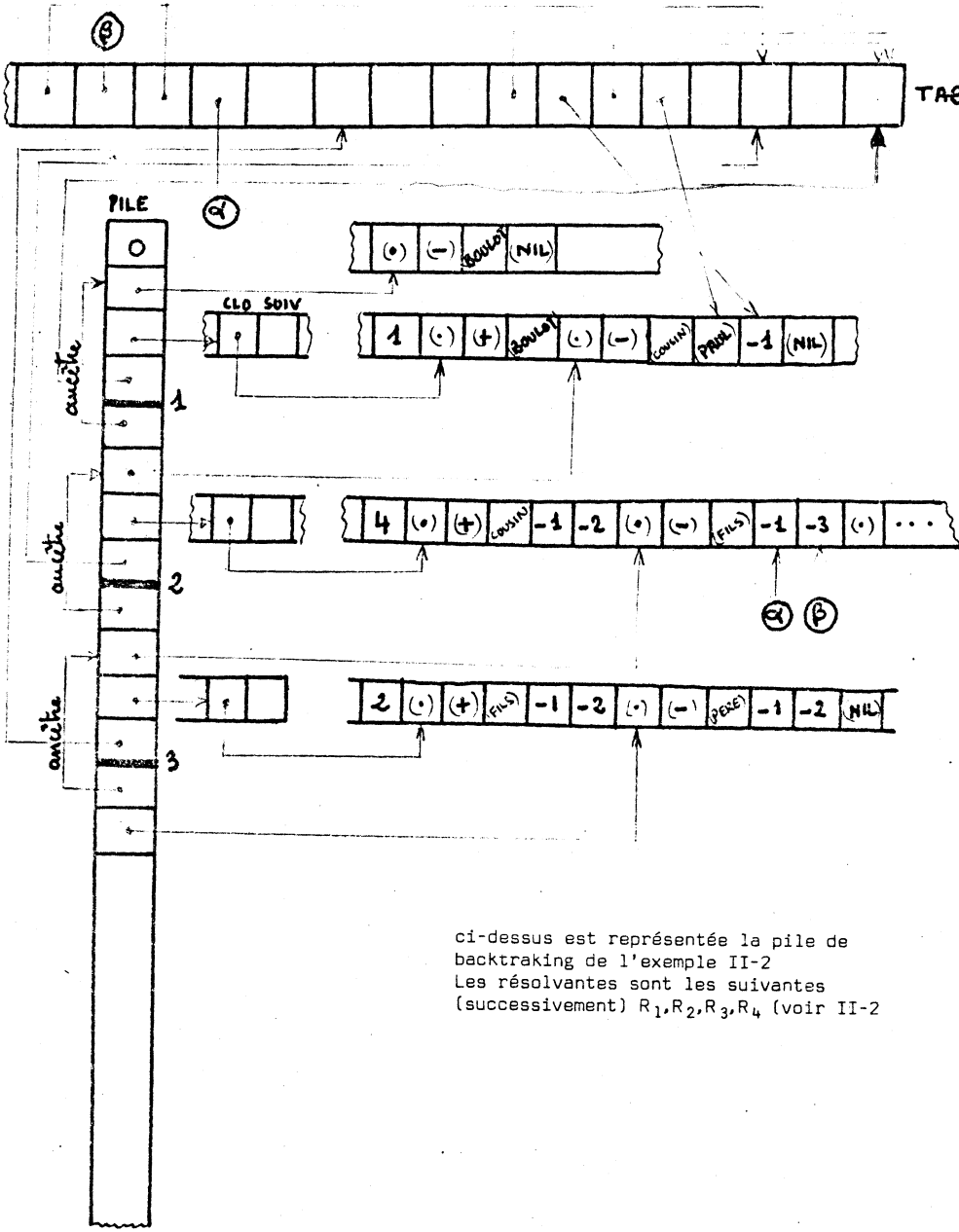
La première adresse de chaque séquence est l'adresse dans la pile du pointeur indiquant l'ancêtre du littéral qu'on unifie au moment du chargement de la pile.

La deuxième adresse est celle du premier littéral de la résolvante en cours au moment du chargement de la pile.

La troisième adresse est celle de l'axiome correspondant au littéral précédent.

La quatrième adresse est celle de la substitution associée à l'axiome précédent.

La liste des entiers négatifs représente des adresses de la pile des variables (voir démonstrateur).



ci-dessus est représentée la pile de backtraking de l'exemple II-2
Les résolvantes sont les suivantes (successivement) R_1, R_2, R_3, R_4 (voir II-2

La résolvante est retrouvée dans la pile à l'aide de la deuxième information de la séquence de longueur 4.

Le squelette est reconstitué comme suit:
les premiers littéraux de la résolvante sont atteints directement à l'aide du 2ème pointeur de la séquence de 4 qui se trouve au sommet de la pile. Les littéraux suivant seront successivement atteints en remontant à l'aide du premier pointeur de la séquence (ancêtre). L'information ainsi pointée donne l'adresse du littéral ancêtre qui a été unifié. Le morceau de la résolvante mémorisé à ce niveau est la suite des littéraux qui suivent cet ancêtre. On remonte successivement les ancêtres jusqu'à ce que le pointeur soit nul.

A ce squelette est associé une instance que l'on récupère par remontées successives comme suit:
les variables contenues dans les littéraux récupérés à un étage de la pile sont lues à travers la substitution mémorisée dans le quadruplet ancêtre de celui dans lequel sont récupérés les littéraux. (C'est en effet lors de l'unification de l'ancêtre avec l'axiome associé qu'a été crée cette partie de la résolvante et la substitution associée à ses variables est donc celle mémorisée à ce moment.)

Précisions sur la structure de la zone des variables

A chacun des axiomes utilisés, il est associé un n-uplet comportant des séquences successives de deux informations pour chacune des variables de l'axiome.
A partir de l'adresse de la substitution associée à l'axiome on atteint la séquence de chaque variable en ajoutant à cette adresse deux fois l'entier négatif associé à la variable (le tableau TAB varie en sens inverse pour les variables).

Lorsqu'une variable est substituée par un terme les deux informations mémorisées dans le doublet lui correspondant sont d'une part l'adresse du terme et d'autre part la substitution associée à l'axiome qui contient le terme substitué.

III-3 Le programme d'initialisation

Le programme d'initialisation a pour but de coder en mémoire toutes les informations nécessaires au fonctionnement du démonstrateur.

Son premier travail est de coder dans le dictionnaire tous les identificateurs du type caractère.

Ensuite il faut également entrer dans le dictionnaire les prédicats spéciaux. Il s'agit de (.) à deux arguments, de (+) à un argument de (-) à un argument et de NIL à zéro argument.

Il faut ensuite coder les prédicats évaluables.

Enfin, nous devons entrer l'analyseur. Les prédicats utilisés par celui-ci sont codés dans le dictionnaire puis la grammaire est entrée.

Certaines clauses spéciales sont également codées. En particulier les clauses -BOULOT, -ERREUR(*x) et l'axiome -.(*x,*y) (on ne se sert en fait que du terme).

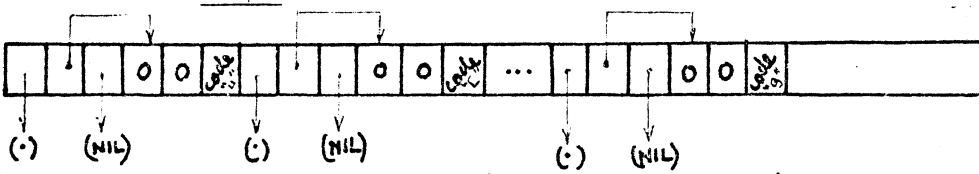
a)Codage des caractères

Les informations relatives à chacun des 63 caractères du code E.B.C.D.I.C. sont mémorisées dans des séquences successives de longueur 6.

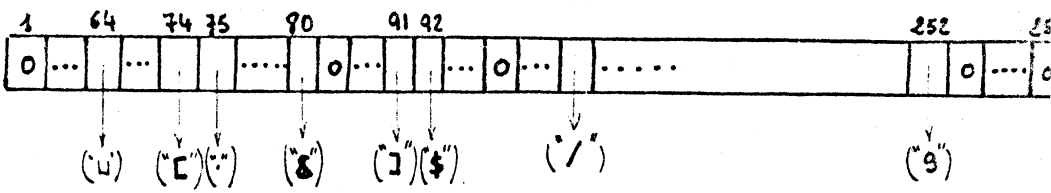
Il faut disposer au moment du codage de l'adresse des prédicats . à deux arguments et de NIL.

Les 3 premières informations sont relatives au nom, la quatrième est le poids (il est toujours nul), la cinquième est l'adresse du doublet qui doit donner l'adresse des clauses commençant par ce prédicat (au moment du chargement cette adresse est zéro) et enfin la sixième information est le code effectif du caractère.

Exemple



Dans ce codage, tous les caractères se suivent sans intervalles. Or, dans le code E.B.C.D.I.C. il y a des espaces vides entre chaque séries de caractères. Il faudrait donc pour chaque caractère déterminer la suite continue à laquelle il appartient afin, après avoir opéré le décalage approprié, de pouvoir l'atteindre. Cela suppose que pour chaque caractère il faudrait faire autant de tests qu'il y a de suite disjointes de caractères. Nous avons créé pour éviter cela un tableau (CALCUL) qui a pour dimension 256. A chaque position correspondant à un caractère on trouve dans le tableau l'adresse effective dans TAB de la séquence correspondant au caractère. Les cases ne correspondant pas à un caractère EBCDIC sont chargées à 0 et la lecture entraine un branchement sur la clause -ERREUR(*x) où *x est substitué par LU(*y).
Nous avons la structure suivante pour CALCUL:



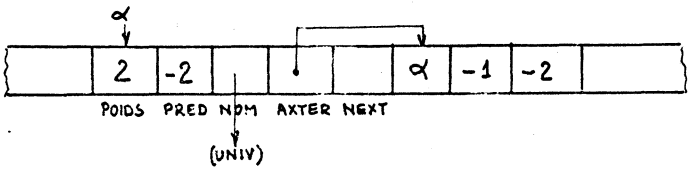
b)Codage des prédicats spéciaux

On charge le dictionnaire correspondant aux deux identificateurs . et NIL puis avec + et - et enfin les identificateurs BOULOT et ERREUR.
On charge les clauses -BOULOT et -ERREUR(*x) à la suite.Tout ce qui précède est chargé par programme, ce qui suit est chargé par lecture.

c)Chargement des prédicats évaluables et des prédicats utilisés dans l'analyseur

Les prédicats évaluables sont entrés sous forme d'une chaine continue de caractères. Chaque identificateur est séparé du suivant par un chiffre représentant le nombre d'arguments du prédicat. Le dernier caractère de la chaine est le caractère "#". Il indique que la liste suivante ne devra pas être traité de la même manière.
En effet les prédicats évaluables sont chargés un peu différemment. Afin de pouvoir les atteindre immédiatement nous notons dans la case repérée par PRED l'opposé du chiffre représentant le numéro d'apparition du prédicat.

Exemple Codage du 2ème prédicat le la liste (UNIV)



Les prédicats de la grammaire sont codés de la même manière, mais la case PRED en chargée à 0.

d) Chargement de la grammaire

La grammaire doit être écrite sous une forme un peu particulière afin de condenser l'écriture de l'algorithme qui doit la coder.
Tous les identificateurs sont suivis d'un entier représentant le nombre d'arguments qu'ils utilisent. La clause est écrite sous forme fonctionnelle préfixée. Chaque clause doit être précédée d'un entier représentant le nombre des variables utilisées dans cette clause.
Les variables étant notée par un entier correspondant à leur ordre d'apparition sans la clause.

Exemple

La clause .(+FRERE(*x,*y),.(-PERE(*z,*x),.(-PERE(*z,*y),NIL)

sera entrée sous la forme:

3.2+1 FRERE212.2-1PERE231.2-1PERE232 NILO

III-4 Le démonstrateur

Le programme principal du démonstrateur lit sur le disque, chargé par l'initialisateur, les données qu'il utilise pour son fonctionnement. Il appelle ensuite le sous-programme RESOUD qui est l'algorithme de base de la démonstration. Celui-ci utilise d'autres sous-programmes dont la fonction est très spécialisée, UNIFIE réalise les unifications et BACKT récupère les résolvantes et autres informations au cours des remontées. On peut dont décomposer en trois parties l'algorithme de démonstration :l'unification, la résolution, le backtraking; ces trois parties correspondent à trois sous programmes FORTRAN qui gèrent la démonstration. Les algorithmes relatifs à chacun des prédicats évaluables viennent s'ajouter au démonstrateur. Les programmes sont donnés dans l'annexe-programmes.

III-4-1 L'unification

L'algorithme de base est le sous-programme UNIFIE. Cet algorithme réalise l'unification de deux termes d'adresses respectives CLAUSA et CLAUSR auxquels sont associés les substitutions d'adresses INSTA et INSTR. Il rend par l'intermédiaire d'une variable T un résultat logique (suivant que T=1 ou T=0) qui permet de savoir que l'unification des deux termes a pu être réalisée.

III-4-2 Le Backtraking

L'algorithme de base est le sous-programme BACK.
Cet algorithme a pour effet de "remonter" dans la pile de backtraking lorsque la démonstration est arrêtée sur une impasse. Il cherche donc le choix d'axiome le plus récent stocké dans PILE. Cette opération permet de vider les sommets de la pile de backtraking (jusqu'au 1er choix trouvé)ainsi que le sommet de la pile des variables (jusqu'à l'instance correspondant à l'ancêtre de la séquence du nouveau sommet de PILE).

Les différentes variables utilisées lors des résolutions sont réinitialisées.

Après l'exécution de ce sous-programme la démonstration peut donc se poursuivre.

III-4-3 La résolution

L'algorithme de base est le sous-programme RESOUD. Son role est de gérer la démonstration.
Il choisit les axiomes ou sélectionne les sous-programmes associés aux prédicats évaluables,

-provoque l'unification des littéraux
-construit les résolvantes successives
-déclanche le backtraking
-arrête la démonstration.

III-4-4 Sous-programmes divers

Certains algorithmes sont fréquemment utilisés pour des taches bien déterminées, il a donc été nécessaire d'en faire des sous-programmes utilisables dans tout le démonstrateur.

a) Le sous-programme DESC

Ce sous-programme permet à partir de l'adresse d'un terme (TERM) et de la substitution qui lui est associée (INST) de rendre les adresses (TERM et INST) du terme et de la substitution situés à l'extrémité de la chaine de reprise des variables.

Si TERM est l'adresse d'une variable qui a été substituée, l'adresse rendue sera différente de l'adresse initiale. Si TERM est l'adresse d'un terme différent d'une variable ou celle d'une variable non substituée l'adresse rendue par le sous-programme sera l'adresse initiale.

b)Le sous-programme SAUT.

Ce sous-programme,auquel est donné en paramètre une adresse α , saute le terme d'adresse $\alpha+1$ codé linéairement et restitue (dans RESTER) l'adresse du terme suivant.

Il permet donc dans la zone des termes et des clauses codés de se déplacer de toute une suite linéaire correspondant à un arbre. En particulier il peut permettre de sauter des littéraux entiers.

c)Le sous-programme SUBST

Ce sous-programme utilise trois paramètres:
-l'adresse α d'un terme
-l'adresse β de la substitution associée au terme d'adresse α
-l'adresse δ de la substitution associée à la variable que l'on veut substituer.

Il charge donc le doublet réservé à la variable à substituer en mettant dans la première case l'adresse du terme qui est substitué et dans la seconde l'adresse de la substitution associée à ce terme.

d) Le sous-programme ASHCOD

Ce sous-programme utilise comme paramètre l'adresse d'un peigne codé linairement représentant une unité, il calcule la clé d'accès au dictionnaire de cet identificateur et restitue dans KEY le résultat de ce calcul.

Dans le cas où l'unité n'a qu'un seul caractère et pas d'argument KEY donne directement l'adresse dans le dictionnaire.

e)Le sous-programme SOREV

Ce sous programme est appelé après chaque exécution d'un sous-programme relatif à un prédicat évaluable.
Il fournit la résolvante après l'évaluation du prédicat et son action est directement liée à cette évaluation:

-Si la syntaxe des arguments du prédicat évaluable est erronée (T=2), le nouveau morceau de la résolvante engendrée sera -ERREUR ("Prédicat impossible à évaluer").

-Si le prédicat est évalué à VRAI (T=0) et le signe le précédant est - (EVAL=1) ou bien si le prédicat est évalué à FAUX (T=1) et le signe est +(EVAL=0) alors le sous-programme fournit la valeur logique VRAI(T=0) et la résolvante est construite normalement par la suite de la démonstration.

Si l'un de ces deux premiers cas ne peut s'appliquer, le sous-programme fournit la valeur logique FAUX(T=1) et la démonstration arrivée dans une impasse devra se poursuivre par un backtraking.

f)Le sous-programme TESTT

Ce sous-programme a pour fonction de tester si les deux sommets dynamiques opposés des piles du tableau TAB sont distant d'une certaine valeur (MARGE) qui puisse autoriser la poursuite de l'exécution du programme.

Dans le cas où la marge n'est plus suffisante le programme est actuellement arreté, il faudrait cependant que soit déclenché une procédure de récupération de la place mémoire dans TAB qui n'est plus indispensable à l'exécution du programme (les suppressions de clauses intervenues au cours de la démonstration).

g)Le sous-programme TESTP

La fonction de ce sous-programme est identique à celle du précédent, il contrôle le débordement des piles du tableau PILE.

III-4-5 Les prédicats évaluables

A chacun des prédicats évaluable est associé un sous-programme FORTRAN.

Par l'intermédiaire de la variable T ils peuvent rendre les informations suivantes:

T=2	évaluation impossible
T=1	évaluation FAUX
T=0	évaluation VRAI

Les fonctions particulières des prédicats évaluables ont été décrites précédemment nous nous attachons ci-dessous à montrer comment elles se traduisent au niveau des algorithmes.

Cette descriptions succinte des principaux sous-programmes a pour but de rendre plus facile la consultation des sous-programmes FORTRAN.

a)Le sous- programme LU

Ce sous-programme trouve l'information qu'il doit traiter (un caractère) dans un tableau (buffer d'entrée) ENTRE de dimension 80 à l'adresse IENTRE.

Si IENTRE =80 alors le tableau ENTRE est chargé par un ordre de lecture FORTRAN. Le contenu d'une carte est transmis dans le tableau. S'il n'y a pas de carte suivante l'évaluation du prédicat étant impossible on transmet l'information T=2. Le pointeur IENTRE est remis égal à 1.

Si IENTRE n'est pas égal à 80 alors le terme argument du prédicat est unifié contre le terme représentant le caractère contenu dans ENTRE(IENTRE) Suivant que l'unification est possible ou non on communiquera les informations T=0 ou T=1.

b) Le sous-programme LUB

Ce sous-programme est identique au précédent si ce n'est qu'il ne tient pas compte des caractères "blancs" et qu'il se positionne sur le premier caractère non blanc.

c)Le sous-programme ECRIT

Ce sous-programme donne l'information T=2 si le terme argument du prédicat n'est pas un caractère.

Si la syntaxe de l'argument est correcte il charge le tableau (buffer de sortie) SORTIE de dimension 132 en mettant à l'adresse ISORTI le code du caractère récupéré dans le dictionnaire.

Si ISORTI=132 le tableau est vidé par une instruction d'écriture FORTRAN et ISORTI est réinitialisé à 1.

d)Le sous-programme LIGNE

Ce sous programme a pour effet de vider le buffer de sortie (SORTIE) et de repositionner ISORTI à 1. Si le tableau SORTIE est vide (ie ISORTI=1) l'effet produit est l'écriture d'une ligne de"blancs" , autrement dit le saut d'une ligne.

e) Le sous-programme AJOUT

Ce sous-programme utilise lui-même un autre sous-programme important (AJOULI) qui a pour fonction d'ajouter un littéral.

Il a pour paramètre l'adresse d'un terme. Si la syntaxe du terme est bonne il va coder linéairement la clause suivant la syntaxe précisée précédemment, puis il modifie la liste des clauses commençant par le même prédicat que celle ajoutée en chargeant un doublet d'informations qui devient le premier de la liste. La première information étant l'adresse de la clause codée, la seconde celle du doublet qui était précédemment en tête de la liste.

f) Le sous-programme SUPP

Ce sous programme, à l'inverse du précédent doit supprimer la clause du début de la liste de celles dont le premier littéral correspond à l'argument du prédicat.

Pour cela il suffit donc de supprimer le premier doublet de la liste et de faire pointer la tête de liste sur le suivant.

g)Le sous-programme SLASH1

Ce sous-programme charge un zéro dans le 3ème case des séquences de quatre informations du tableau PILE, jusqu'à la séquence repérée par le pointeur "ancêtre" du sommet de pile.

Cela permet lors des remontées de ne pas chercher les axiomes suivants celui qui était précédemment mémorisé à cet endroit.

h)Le sous-programme SLASH2

Sa fonction est identique au précédent, mais au lieu de na remonter dans PILE que jusqu'à l'ancêtre immédiat on remonte jusqu'à celui que l'on a pu unifier contre l'argument du prédicat.

En fait il y a deux étapes dans le déroulement du sous-programme:

1- Recherche parmi les ancêtres du premier unifiable contre l'argument du prédicat.

2- Remise à zéro des informations concernant les choix d'axiomes jusqu'à l'ancêtre unifié.

Cette deuxième partie est en fait plus complexe car elle supprime également de la pile les séquences devenues inutiles (celles dont seule l'information sur les choix d'axiomes demeurait nécessaire). La pile est alors tassée et les adresses des ancêtres rectifiées.

i)Le sous-programme SAUVE

Ce sous-programme a pour seul effet d' affecter une variable (INPL), jusqu'alors nulle, à 1.

Cette affectation permettra dans la boucle la plus externe du sous-programme RESOUD de stocker sur disque l'état de tous les tableaux utilisés ainsi que les pointeurs qui les parcourent.

j)Le sous-programme ETAT

Ce sous-programme construit le terme formé des ancêtres successifs suivis des littéraux restant à résoudre.

Il remonte donc dans PILE et récupère les adresses correspondant aux ancêtres successifs et les substitue aux variables du terme. .(*x,*y) qui est utilisé pour construire le terme représentant l'état de la résolvante.

k)Le sous-programme UNIV

Ce sous-programme est divisé en deux parties, la première correspondant au cas où le premier argument du prédicat est un arbre, la seconde au cas où cet argument est une variable.

-cas d'un arbre

Dans ce cas l'arbre est décomposé en un "peigne"à l'aide du terme .(*x,*y).

Il est ensuite unifié avec le deuxième argument du prédicat.

-cas d'une variable

-La syntaxe du deuxième argument est **vérifié**e

-L'identificateur est "reconnu", s'il n'existe pas dans le dictionnaire, l'algorithme le code.

-Le terme correspondant à l'arbre décomposé est construit et unifié avec la variable du premier argument.

Dans chacun des cas précédents on procède différemment suivant qu'il s'agit de nombres ou non.

ANNEXE - 1

UTILISATION DES VARIABLES DECLAREES EN COMMUN

TAB..... Tableau,dimension variable
PILE..... Tableau, dimension variable
CLES..... Tableau,dimension 800
CALCUL..... Tableau, dimension 256
ENTRE..... Tableau, dimension 80
SORTIE..... Tableau, dimension 132
DPILE..... A pour valeur la dimension du tableau PILE
DTAB..... A pour valeur la dimension du tableau TAB
ITER..... Pointeur dans TAB indiquant la première case non utilisée de la zone des termes.
IVAR..... Pointeur dans TAB indiquant la première case non utilisée de la zone des variables.
IBAC..... Pointeur dans PILE indiquant la première case libre de la pile de backtraking.
IREC..... Pointeur dans PILE indiquant le sommet de la pile de récursivité.
CLAUS A..... Pointeur dans TAB (zone des termes) utilisé pour l'unification
CLAUS R..... idem
INSTA..... Pointeur dans TAB (zone des variables) utilisé pour l'unification
INSTR..... idem
TERM..... Pointeur dans TAB (zone des termes) utilisé par le sous-programme DESC
TERM 1..... Idem
INST..... Pointeur dans TAB (zone des variables) utilisé dans le sous-programme DESC
INST 1..... Idem
SUB..... Variable utilisée par le sous-programme DESC

POIDS
PRED
NOM .....
AXTER
CODE

Constantes utilisées pour l'adressage dans le dictionnaire, respectivement initialisées à:0,1,2,3,4,3.

INPL.....
INMO

variables ayant diverses utilisations (sauve)

ACMO
ACPL.....
CLO
SUIV

Constantes utilisées pour l'adressage des axiomes respectivement initialisées à: 0,1,0,1.

U..... Variable utilisée par les ous-programmes BACKT et RESOUD, arrête la démonstration lorsque U=1.

T..... Variable utilisée dans l'unification
EVAL..... Variable indiquant le signe précédant les prédicats évaluables.
ENTIER
UNLETT.... } Variables utilisées dans le sous-programme UNIV.
NIL..... Variable initialisée à zéro
MARGE..... Variable indiquant l'écart minimum entre les pointeurs ITER et IVAR, et entre les pointeurs IBAC et IREC.
NBRE..... Variable indiquant l'opposé du nombre de variables permis dans une clause (-1000)
LETTRE
CHIFFRE... } Constantes indiquant dans TAB respectivement la fin des zones
CARACT où sont stockés les lettres, les chiffres et les caractères.
CNIL..... Adresse dans le dictionnaire du terme NIL à zéro argument
CPOINT.... Adresse dans le dictionnaire du terme .(Point) à deux arguments
AXNIL..... Adresse de l'axiome NIL
AXPOIN.... Adresse de l'axiome .
ERREUR.... Adresse du . précédent l'axiome -ERREUR(*x)
BOULOT.... Initialisé à 2
PLUS..... Adresse dans le dictionnaire du terme + à deux arguments
MOINS..... Adresse dans le dictionnaire du terme - à deux arguments
CPINAX.... Variable
RETOUR.... Pointeur indiquant provisoirement le dernier ancêtre dans la pile de backtraking.
RESTER.... Pointeur de sortie utilisé dans le sous-programme SAUT
COMPTE.... Variableutilisée dans le sous-programme ASHCOD
LIMITE.... Pointeur utilisé dans SUBST
SUBSAX.... Pointeur utilisé dans UNIV indiquant l'adresse de la substitution du terme créé
ARG..... Compteur d'arguments utilisé dans UNIV
DEBUT..... Pointeur indiquant le début d'un "peigne" utilisé dans UNIV
EXPLOR
ISTOR .... } Variables diverses
SEPAR
KEY..... Pointeur dans le tableau CLES
IENTRE.... Pointeur dans le tableau ENTRE indiquant le dernier caractère lu
ISORTI.... Pointeur dans le tableau SORTIE indiquant le dernier caractère écrit
IMPMAX.... Nombre maximum de caractères contenus dans une ligne en écriture (132)
LECMAX.... Nombre maximum de caractères contenus dans une ligne en lecture (80)

BIBLIOGRAPHIE

PROLOG: "Un système de communication homme-machine en français"
Rapport interne
A. COLMERAUER, H. KANOUI, R. PASERO, P. ROUSSEL

ROBINSON J.A
"A machine-oriented logic based on the Resolution Principle (1965)

NILSONN
"Problem-solving methods in artificial intelligence"

HERMES H.
"Enumerability. Decidability. Computability"

ROUSSEL P.
"Définition et traitement de l'égalité formelle en démonstration automatique" Thèse 3è cycle (1972)

KOWALSKI R. and HAYES P.

"Semantic trees in automatic theorem-proving" (1969)

Janvier 1974

Mise à jour de la liste des prédicats évaluables de PROLOG

I Prédicats d'entrée-sortie

I.1 Prédicat LUB

‹prédicat LUB› ::= LUB(‹terme›)

Ce prédicat est évalué de la même manière que LU, mais sauté les caractères blancs et unifie terme avec le premier caractère non blanc lu.

I.2 Prédicat TTY

‹prédicat TTY› ::= TTY

Ce prédicat est toujours évalué à VRAI
Si le littéral correspondant est supprimé il commande le changement de fichier de lecture.
Si la lecture se faisait sur le terminal, TTY provoque le changement et fait continuer la lecture sur un fichier FORTRAN.

Remarque

La ligne courante en lecture est conservée dans un buffer;lors d'un changement de lecture d'un fichier T sur un fichier F, puis d'un retour à T, la lecture reprend sur le caractère suivant celui sur lequel s'était arrêté la lecture avant le changement.
Au départ de l'exécution, le fichier de lecture est le terminal.

I.3 Prédicat BOOLISTE

‹prédicat BOOLISTE› ::= BOOLISTE

ce prédicat est toujours évalué à VRAI
Si le littéral correspondant est supprimé il provoque soit le listage des instructions analysées, (si li listage ne se faisait pas) , soit l'arrêt du listage des instructions (si le listage se faisait)

I.4 Prédicat IMPRIME

‹prédicat IMPRIME› ::= IMPRIME

ce prédicat est toujours évalué à VRAI
Lorsque le littéral correspondant est supprimé il provoque l'impression en sortie de la ligne en cours de lacture à condition que celle-ci n'aie pas déjà été imprimée à l'aide du prédicat BOOLISTE.

II Prédicat d'édition

II.1 Prédicat AJOUTB

<prédicat AJOUTB> ::= AJOUTB(<clause>)

ce prédicat est évalué de la même manière que le prédicat AJOUT.
Lorsque le littéral correspondant est supprimé il provoque l'ajout de la clause à la fin du fichier des clauses de la partie P_i correspondante.

Remarque sur l'utilisation d'AJOUTB
Lorsque l'on ajoute (à l'aide d'AJOUTB) des clauses consécutives commençant par le même prédicat P, elles forment un fichier P_i correspondant au prédicat P.
Si les clauses ajoutées ne sont pas consécutives, chaque fichier P_i de clauses commençant par le même prédicat P efface le précédant fichier P_i, et en crée un nouveau.

III Prédicat ATOME

<prédicat ATOME> ::= ATOME(<variable>,<arbre décomposé>) ATOME(<arbre>,<terme>)

Ce prédicat est évalué de la même manière que UNIV à une différence près dans le cas ATOME (<variable>,<arbre décomposé>)
l'arbre décomposé n'est transformé en l'arbre correspondant que si le prédicat ainsi constitué existait déjà dans le dictionnaire.
Dans le cas contraire, l'évaluation de ATOME est FAUX.

IV Prédicat divers

IV.1 Prédicat EGALF

<prédicat EGALF> ::= EGALF(<terme>,<terme>)

Ce prédicat est évalué à VRAI si les deux termes sont formellement égaux
il est évalué à FAUX si les deux termes sont différents.

IV.2 Prédicat ANCETRE

<prédicat ANCETRE> ::= ANCETRE(<littéral>)

ce prédicat est évalué de la même manière que le prédicat slash1
Si le littéral correspond est supprimé il provoque l'unification de littéral avec le premier ancêtre avec lequel il est unifiable.

V Prédicat arithmétiques

V.1 Prédicats MULT, DIV, RESTE

<prédicat MULT> ::= MULT (<terme>,<terme>,<terme>)

<prédicat RESTE> ::= RESTE(<terme>,<terme>,<terme>)

<prédicat DIV >::= DIV (<terme>,<terme>,<terme>)

ces prédicats sont évalués de la même manière que PLUS et MOINS;les opérations effectuées sont la multiplication, la division entière, et le reste de la division entière.

V.2 Prédicat INF

<prédicat INF> ::= INF(<terme>,<terme>)

ce prédicat est impossible à évaluer si l'un des termes n'est ni un caractère ni un entier, ou bien est une variable.
Il est évalué à VRAI si le premier terme est strictement inférieur au second et évalué à FAUX dans le cas contraire. L'ordre des termes est celui défini, pour les caractères, par le code E.B.C.D.I.C, et par l'ordre naturel pour les entiers. Les entiers sont supérieurs aux caractères.

L'interpréteur analyse et exécute les commandes les unes après les autres. Une commande est une clause qui n'est pas ajoutée au fichier mais exécutée littéral par littéral de façon déterministe.

Exemple: La commande +A+B+C. est équivalente au point de vue démonstration à +A -/ +B -/+C -/.
Les littéraux de commandes peuvent être n'importe lesquels; en particulier les prédicats évaluables peuvent apparaitre à la première place dans une commande. Certains prédicats utilisés avec le signe - font appel à des procédures écrites en PROLOG et déjà codées:

-LIREFICHIER permet l'analyse et le codage des clauses suivantes jusqu'à la clause +FIN. et les ajoute au fichier.
-SORCHA(*x) permet l'écriture de la chaine unifiée contre *x.
-SORTER(*x) écrit l'expression selon les conventions d'opérateurs.
-STOP provoque l'arrêt du programme.
-BLANC(*x) le caractère "blanc" est unifié contre la variable
-ETOILE(*x) le caractère * est unifié contre la variable
-VIRG(*x) le caractère , est unifié contre la variable
-PARG(*x) le caractère (est unifié contre la variable
-PARD(*x) le caractère) est unifié contre la variable
-GUILLEMET(*x) le caractère " est unifié contre la variable.