

# prédicats prédéfinis

## connaissances algorithmiques

Toutes les connaissances ne sont pas déclaratives, ou du moins pour certaines on préfère fournir la méthode de calcul par une recette standard c'est le cas par exemple de la multiplication que l'on gagne à faire faire par la boîte noire constituée par le micro, c'est aussi le cas pour la fonction "sinus" , dont un algorithme de calcul est codé dans l'interpréteur Prolog, d'où un premier groupe de fonctions prédéfinies. Ce peut être aussi le cas pour d'autres fonctions spécifiques à une application, d'où un deuxième groupe de primitives prédéfinies servant à définir des algorithmes. Autrement dit, on peut aussi utiliser prolog comme un langage classique, procéduralement.

on se propose maintenant de décrire tous ces prédicats prédéfinis.

on en donne des exemples d'utilisation et lorsqu'ils ne sont pas indispensables c'est à dire lorsqu'ils peuvent être programmés simplement, on en donne une programmation.

## manipulation de la session de travail

### quit

fin de l'exécution du programme Prolog, retour au dos

### exit

fin de l'exécution du programme, retour à l'interpréteur prolog

### module

module (nom\_de\_module).

Permet de subdiviser l'application en modules ou les sous programmes et les variables globales peuvent avoir le même nom sans risque de conflit. les créations dynamiques de règles ou de symboles se font dans le module spécifié par la dernière commande "module" portant sur la clause en cours lors de la compilation.

### end\_module

end\_module.  
ou main.  
pour se remettre dans le segment principal.

## **define**

`define (nom_de_sous_programme).`

Premet de déclarer dans un module un sous-programme qui sera utilisé aussi dans un autre module.

## **external**

`external (nom_de_sous_programme).`

Permet dans un module d'utiliser un sous-programme défini dans un autre module avec l'instruction "define".

## **manipulation des clauses**

### **clear**

`clear (nom_sous_programme).`

effacement de tous les faits et règles de ce sous programme du module.  
il est aussi possible de mettre une liste de noms de sous programmes.

`clear_module.`

effacement de tous les objets définis, de tous les faits et de toutes les règles du module.  
`clear_all.`

effacement de tous les objets définis, de tous les faits et de toutes les règles du programme. retour à la case départ avec uniquement les prédicats prédéfinis.

### **assert**

`assert ( conclusion, liste de conditions).`

permet de créer dynamiquement des faits (liste = [] ) ou des règles.  
l'index de la règle ajoutée peut être connu grâce à `_x := index`

### **assert\_first**

`assert_first ( conclusion, liste de conditions).`

comme assert, mais l'ajout se fait en tête de la liste des clauses.

### **assert\_before , assert\_after**

`assert_before ( conclusion, liste de conditions).`

`assert_after ( conclusion, liste de conditions).`

`assert_before ( conclusion, liste de conditions , index).`

`assert_after ( conclusion, liste de conditions , index).`

insère la nouvelle règle devant (ou derrière) celle dont l'index est spécifié ou devant(ou derrière) la dernière accédée. Cela permet de classer les règles au moment de leur entrée dans la base. before va moins vite que after, mais c'est le plus utile ! of course.

soit à ajouter des faits en les classant selon leurs premier paramètre :

```
ajoute(relation(_p , _q)) ->
    relation(_x , _y) ,
    _p < _x ,
    assert_before( relation(_p , _q) , [] ) , / .
ajoute(relation(_p , _q)) -> assert( relation(_p , _q) , [] ) .
```

Si la comparaison est plus compliquée, il faut tenir compte du fait que l'index courant va être modifié par l'appel des prédicats de comparaison

```
ajoute(relation(_p , _q)) ->
    relation(_x , _y) ,
    _save_index := index ,
    plus_petit( relation(_p , _q) , relation(_x , _y)) ,
    assert_before( relation(_p , _q) , [] , _save_index ) , / .
ajoute(relation(_p , _q)) -> assert( relation(_p , _q) , [] ) .
```

et, en tenant compte que "after" est beaucoup plus rapide que "before" :

```
ajoute(relation(_p , _q)) ->
    prec_index := 0 ,
    relation(_x , _y) ,
    _save_index := prec_index ,
    prec_index := index ,
    plus_petit( relation(_p , _q) , relation(_x , _y)) ,
    assert_after( relation(_p , _q) , [] , _save_index ) , / .
ajoute(relation(_p , _q)) -> assert( relation(_p , _q) , [] ) .
```

## **retract**

retract (conclusion , liste de conditions).

permet de retirer de la base les faits ou règles s'unifiant avec les deux paramètres.

retract\_one (conclusion , liste de conditions).

ne retire que le premier trouvé

on aurait pu programmer :

```
retract_one(_x,_y) -> retract(_x,_y) , / .
```

retract (conclusion , conditions , index).

retire le fait spécifié

retract\_from (conclusion , conditions , index).

retire les faits à partir du fait spécifié

Dans les deux cas, si la règle était hsh\_coded, elle ne l'est plus. oh !

## **anafact**

anafact (conclusion , liste , index).

Permet de retrouver, par accès direct et avec une seule unification, un fait ou une règle dont on connaît l'index sans le retirer de la base. La règle peut être hsh\_coded ou non. l'index est un pointeur dans les tables de la base de données de Prolog, il peut être connu grâce à la fonction "`_x := index`" après un assert ou après un appel classique du fait . Cela permet donc de gérer des tables de hshcode et des arbres binaires où l'on range l'index des faits à classer.

## **replace**

replace (conclusion , liste , conclusion nouvelle , nouvelle liste).

permet de remplacer le premier fait ou règle s'unifiant. si la règle était hsh\_coded, elle le reste . ah! tant mieux. mais la clé du hshcode (le premier symbole) doit être la même.

replace(conclusion , conditions).  
replace(conclusion , conditions , index).

permet de remplacer la règle spécifiée par "index" ou la dernière accédée

```
a(b , _x) .  
replace(a(b,0) , []).  
  
a(b , _x) ,  
_save_index := index,  
....  
replace(a(b,0) , _save_index) .
```

dans tous les cas, l'index de la règle remplacée est inchangé.

## **erase**

erase (conclusion , liste de conditions).

permet de supprimer tous les faits et règles s'unifiant

## **edit**

edit (nom\_de\_sous\_programme).

edit\_module.

edit\_all.

Permet de créer ou modifier le sous-programme spécifié du module.

## **edt**

edt ('nom\_de\_fichier').

permet de créer ou modifier un fichier texte.

## **flot courant des commandes**

### **load**

load ('nom\_de\_fichier').

les commandes suivantes seront lues depuis le fichier spécifié.

cela sert en particulier à faire lire un fichier de connaissances (faits et règles) en début de session. l'imbrication est possible (jusqu'à 5 niveaux), le fichier appelé pouvant contenir une commande load(...) le fichier peut contenir des questions, c'est à dire l'appel d'un sous-programme. le retour au fichier imbriquant se fait à la fin du fichier imbriqué ou lors de la commande :

load.

### **reload.**

reload ('nom\_de\_fichier').

les déclarations ne s'ajoutent pas, elles remplacent les éventuelles précédentes.

### **verify**

verify ('nom\_de\_fichier').

les commandes suivantes sont lues depuis le fichier spécifié.

mais elles ne s'exécutent pas : les faits et les règles ne sont pas stockées. si ca se passe bien, on fait évidemment suivre cette commande d'un load(nom\_de\_fichier). si le fichier à compiler contient des commandes load(...), les fichiers correspondant sont vérifiés également.

## **sauvegarde**

### **save**

save ('nom\_de\_fichier' , nom\_sous\_programme).

save\_module ('nom\_de\_fichier').

save\_all ('nom\_de\_fichier').

les faits et les règles spécifiées sont écrit dans le fichier texte. ce fichier est éditable, on pourra le relire par : load('nom\_de\_fichier').

cette méthode est plus souple mais moins rapide que save\_bin et load\_bin.

## **édition d'informations**

### **crossref**

crossref (nom\_de\_sous\_programme)

crossref\_module.

crossref\_all.

donne la liste des sous programmes appelant

### **list**

list (sous\_programme).

list ([sous\_programme , ... ]).

list\_module.

list\_all.

liste le ou les sous-programmes spécifiés

### **print**

print ('nom\_de\_fichier')

print (nom\_de\_sous\_programme)

print ([sous\_programme,...])

print\_module

print\_all

### **stat**

stat.

édite l'état de l'occupation mémoire

### **clear\_stat**

clear\_stat.

remet à zéro les compteurs de statistiques

### **flag**

flag.

imprime la valeur de tous les indicateurs

## **identificateurs**

ident\_predef.

imprime la liste des prédicats prédéfinis.

ident\_module.

imprime tous les identificateurs définis dans le module.

ident\_main.

imprime tous les identificateurs définis dans le module main et tous les identificateurs définis "external".

ident\_all.

imprime tous les identificateurs définis dans le programme.

ident\_undef.

Imprime tous les identificateurs d'au moins un paramètre et qui ne sont pas définis par un fait ou une règle. Ils sont peut-être non définis par oubli ou faute d'orthographe, mais ce n'est pas forcément une erreur, si ils se définissent dynamiquement ou si ce sont simplement des structures. utiliser alors :

crossref(identificateur)

pour voir où l'identificateur non défini est utilisé.

## **retour arrière**

La plupart des prédicats prédéfinis ne rendent qu'un résultat et échouent toujours lors d'un retour arrière ultérieur. les exceptions sont :

pop  
element  
retract  
dir

si on ne désire que le premier résultat, il suffit d'utiliser le "/" :

popbis(\_x , \_y) -> pop(\_x , \_y) , /.

## **contrôle du moteur d'inférence**

### **exit**

exit.

fin du programme, retour à la boucle de fond de l'interpréteur prolog.

### **quit**

quit.

fin du programme et retour au dos.

## définition de bloc

block (nom\_de\_bloc).  
va de paire avec un fail(nom\_de\_block) qui backtrackera jusque là.

## coupe des choix /

./.

Empêche le retour arrière au dernier choix, super astuce, mais n'influe pas sur la valeur vrai ou faux de la règle en cours. cela permet de forcer la sortie du sous-programme. mais la règle courante est évaluée jusqu'a sa dernière condition. on ne sort de la règle courante que lorsque suite à un retour arrière on tombe sur le "/" cela sert par exemple à éviter de parcourir une "table" d'un bout à l'autre :

on s'arrête à la première réussite (comme le ESCAPE d'une boucle FOR en pascal).

/(nom\_de\_bloc).

on ne sort pas de la règle mais on revient en arrière au niveau du bloc spécifié

soit un ensemble de faits du genre :

message(1,"blabla 1").  
message(2,"blabla 2").

message(\_x , \_message)                      rend (ou au moins essaye) successivement tous les messages

on aura tout interet à faire :

ecrire(\_x)-->  
    message(\_x,\_y) ,  
    put(\_y) ,  
    / .

ou à déclarer les faits :

message(1,"blabla 1") -> / .  
message(2,"blabla 2") -> / .

alors    message(\_x , \_message) ? rend uniquement un message

Voici range\_liste qui rend la liste des nombres :

**range\_liste**(\_min , \_max , [\_min]) ->  
    \_min = \_max .  
range\_liste(\_min , \_max , [\_min , !\_suite]) ->  
    \_min < \_max ,  
    \_n := \_min + 1 ,  
    range\_liste(\_n , \_max , \_suite) .

Et voici le même range\_liste avec un "/" par soucis d'efficacité.

supprimer le "/" est ici indiffèrent sur le résultat puisque les 2 clauses sont exclusives :



```

range_liste(_min , _max , [_min]) ->
    _min = _max , / .
range_liste(_min , _max , [_min , !_suite]) ->
    _min < _max ,
    _n := _min + 1 ,
    range_liste(_n , _max , _suite) .

```

Le même encore, supprimer le "/" ici n'est plus indifférent :

```

range_liste(_min , _max , [_min]) ->
    _min = _max , / .
range_liste(_min , _max , [_min , !_suite]) ->
    _n := _min + 1 ,
    range_liste(_n , _max , _suite) .

```

Et voici le petit dernier de la famille , qui est le premier dans mon cœur :

```

range_liste(_min , _min , [_min]) -> / .
range_liste(_min , _max , [_min , !_suite]) ->
    _n := _min + 1 ,
    range_liste(_n , _max , _suite) .

```

les deux derniers n'aiment pas que max soit inférieur à min.

## fail

```

fail.
fail(nom_de_block).

```

rend toujours faux et donc force l'échec de la solution en cours, mais pas forcément la sortie de la règle, s'il y a des choix à refaire. fail ne peut être logiquement que la dernière condition d'une règle. on remonte soit jusqu'à la règle suivante du sous-programme soit jusqu'au block ou au repeat spécifié, lequel peut être dans un autre sous-programme.

soit à compter le nombre de fois qu'une condition est vrai :

```

compte(_x) ->  compteur := 0 ,
                condition ,
                compteur := compteur + 1 ,
                fail.

compte(_x) ->  _x := compteur ,
                writeln('ca fait : ' , _x) .

```

bien comprendre le programme suivant, qui définit le "not" en combinant "/" et "fail" .

```

not(propriété) -> propriété , / , fail .
not(propriété) .

```

**for**

```
for (variable,min,max).
for (identificateur,min,max).
ffor.
```

il faut mieux éviter de se servir de ce prédicat.

on peut imbriquer les for.

s'il n'y a pas de ffor, la boucle s'exécute par retour arrière.

soit à afficher les carrés des 10 premiers nombres :

```
for(i,1,10) ,
    _x := i^2 , writeln(_x),          /* traitement */
ffor ,
```

sans utiliser le "for", on peut définir le prédicat "range" qui rend tous les entiers entre deux bornes :

```
range(_min , _max , _min) -> _min <= _max .
range(_min , _max , _n) ->
    _min < _max ,
    _n := _min + 1 ,
    range(_n , _max , _n) .
```

```
range(1 , 10 , _i) ,
    _x := _i^2 , writeln(_x),          /* traitement */
fail
```

ne pas confondre "range" décrit ici avec "range\_liste" décrit plus haut dans le paragraphe du "/" et qui lui rend la liste des nombres, alors que "range" rend successivement tous les nombres, un à chaque fois, par retour arrière.

on peut transformer "range" pour générer tous les entiers :

```
genere_integer(0) .
genere_integer(_n) ->
    genere_integer(_x) ,
    _n := _x + 1 .
rend    _n = 0
        _n = 1
        _n = 2 ...
```

**repeat**

```
repeat.
```

C'est une boucle, on n'en sort que par / , quit ou exit.  
la boucle s'effectue par retour arrière.

```
repeat ,
    actions ,
until(condition).
```

La boucle se réexecute complètement depuis le repeat à chaque arrivée sur until. il est fortement conseillé de ne placer entre repeat et until que des prédicats ne rendant qu'une solution "oui". ils peuvent être prédéfinis ou non.

```
repeat(nom_de_repeat).
fail(nom_de_repeat).
    entraine le retour arrière à partir de repeat(nom_de_repeat).
/(nom_de_repeat).
    entrainera le retour arrière à partir de repeat(nom_de_repeat).
succeed(nom_de_repeat).
    entraine le branchement derrière until(nom_repeat,condition).
until(nom_de_repeat,condition).
    entraine la reexecution à son début du repeat(nom_de_repeat).
    sans retour arrière.
```

on peut imbriquer les repeat sans limitations.

Exemples d'utilisation :

```
prog ->
    repeat ,
        readkbd(_car) ,
        traite(_car) ,
        condition_finale ,
    / ,

prog ->
    repeat ,
        readkbd(_car) ,
        traite(_car) ,
        until(condition_finale) ,

prog(_traitement,_nombre) ->
    compteur := 0 ,
    repeat ,
        _traitement ,
        compteur := compteur + 1 ,
        until(compteur == _nombre) ,
    write('c'est fini').
```

```
prog ->
    (* repeat ,                ici le repeat ne servirait à rien *)
    a(_x) ,
    traite(_x) ,
    _x = f ,                    (* et le until(_x=f) amènerait à
    / ,                          boucler indéfiniment sur la première
                                solution de a(_x) *)
```

soit à simuler hlp(\_x) :

```
hlp(_x) ->
    reset(help, _x),
    repeat(lire) ,
```

```

        readcars(help, _ligne , 1) ,
        édite(_ligne) ,
        fail .
    édite([*]) -> wait , / .
    édite([Ctrlz]) -> close(help) , fail(lire) ,
    édite(_ligne) -> putln(_ligne) .

```

## not

```
not (condition).
```

not n'est pas indispensable, on pourrait programmer :

```

not(_p) -> _p , / , fail.
not(_p) .

```

## traitement conditionnel

```

    if (condition) ,
        actions_si ,
    else ,
        actions_sinon ,
    end ,

```

```

ou    if , conditions ,
        then ,
            actions_si ,
        else ,
            actions_sinon ,
        end ,

```

Le then n'est là que pour ne pas changer les habitudes, mais il est facultatif. On peut évidemment imbriquer les if, et la partie else n'est pas obligatoire. L'utilisation de ce prédicat est très fortement déconseillée, faites plutôt du pascal.

```

    if , _valeur > 10 ,
        write('il y en a beaucoup') ,
        if _valeur = 100 ,
            writeln(output) ,
            end ,
        traite_beaucoup(_valeur) ,
    else ,
        write('c est pas beaucoup') ,
        end ,

```

Il est conseillé de ne placer entre if et end que des prédicats ne rendant qu'une valeur "oui". ils peuvent être prédéfinis ou non. dans la forme if(condition), la condition peut rendre plusieurs valeurs, seule la première est prise en compte.

If n'est pas indispensable (peu de prologs le fournissent), on pourrait programmer :

```

    si_alors_sinon (_condition, _si, _sinon) ->
        _condition ,
        /                               /* / pour sortir sans faire le sinon */
        _si ,
    si_alors_sinon (_condition, _si, _sinon) ->
        _sinon .
soit : si_alors_sinon (soif, boire, manger).
    si_alors (_condition, _si) ->
        _condition ,
        _si .

```

## accès direct

### hsh code

```
hsh_coded ( nom_relation( _, ..., ... ))
```

Cette primitive doit être utilisée avant de déclarer des faits ou règles pour cette relation. Les faits et règles de la relation spécifiée sont stockés avec en plus un hsh code calculé sur le premier argument.

si le premier argument est une liste, le hsh code est calculé sur le premier élément de la liste. si l'argument est un terme structuré, le hsh code est calculé avec son terme fonctionnel. si l'argument est un nombre, la valeur du nombre sert à calculer le hsh code. si l'argument est une variable, la relation est chaînée sur toutes les sous chaînes des hsh codes possibles.

Cela accélère considérablement l'exécution, mais prend un peu de place. Attention, cela ne sert que si lors de l'appel le premier paramètre n'est pas une variable ou si c'est une variable liée, sinon, l'accès habituel est utilisé avec unifications successives sur tous les faits de la relation. la primitive retract fait perdre cette table de hsh\_code, pas replace.

### multi hsh code

```
multi_hsh_coded ( nom_relation( _, ..., ... ))
```

Le fonctionnement est semblable à "hsh\_coded", un hsh code est établi sur tous les arguments de la relation spécifiée. Ça prend plus de place. Lors de l'appel, on utilise le hsh code du premier argument non variable.

## accès direct

```
index.
```

cette fonction arithmétique rend l'index de la dernière règle utilisée, valeur que l'on utilisera dans le prédicat `index(valeur, prédicat)` qui permettra de réaccéder rapidement à cette règle et aux suivantes.

```

index (nombre , backtrack).
index(nombre)

```

si le nombre est positif , pour le sous-programme suivant (y compris pour "replace") on se contentera d'appeler la n-ième règle. il faut alors que toutes les règles du sous programme aient été entrées séquentiellement

si le nombre est négatif (il a été calculé par la fonction arithmétique index, pour le sous programme suivant (y compris "replace") , on se contentera d'appeler la règle ayant cet index.

si backtrack = 0 (défaut), on ne backtrackera pas avec les suivants, si non, devine.

## **traitement des listes et chaines de caractères**

Tous les prédicats prédéfinis de ce paragraphe peuvent se programmer.

### **element**

element (objet,liste).

indique si l'objet est un élément de la liste.

element(variable,liste).

rend tous les éléments de la liste les uns après les autres

element\_one(variable,liste).

rend le 1er élément de la liste

on aurait pu programmer "element":

**element**(\_objet , [\_objet , !\_reste]) .

element(\_objet , [\_debut , !\_reste]) -> element(\_objet , \_reste) .

arg (liste,index,objet).

rend le nème élément de la liste ou de la structure

arg ( fon(a,b,\_c) , 3 , resu)

donne : \_c = resu

arg ( fon(a,b,c) , 3 , \_resu)

donne : \_resu = c

### **first**

first (liste,objet).

rend le premier objet de la liste

### **second**

second (liste,objet).

rend le deuxième objet de la liste

### **last**

last (liste,objet).

rend le dernier objet de la liste

```
last([_x] , _x) -> / .
last([_debut , !_fin] , _x) ->
    last(_fin , _x) .
```

## reste

reste (liste,reste).  
rend la liste constituée par la queue de la 1ère liste

on aurait pu programmer :  
reste([\_debut , !\_fin] , \_fin) .

Et on devrait plutôt utiliser la notation **[\_début , !\_fin]** qui est équivalente au "car" et au "cdr" du langage Lisp.

## conc

conc (liste1,liste2,liste3).  
concatène les deux premières liste dans la troisième

on aurait pu la programmer :  
**append**([] , \_liste , \_liste) -> / .  
append([\_debut,!\_fin] , \_liste, [\_debut , !\_reste]) ->  
append(\_fin , \_liste , \_reste).

Cet append est très intéressant, il donne en plus toutes les solutions pour séparer une liste en deux sous listes :

append(\_debut , \_fin , [a , b ,c]) ?  
rend : [a] et [b,c]  
[a , b] et [c]

## substr

substr (liste1,index1,taille,liste\_resultat).  
extrait de la liste 1 la sous liste spécifiée par index et taille

## left

left (liste,taille,liste\_resultat).  
c'est une variante de substr

## right

right (liste,taille,liste\_resultat).  
c'est une variante de substr

## delete

delete(liste , position, taille, liste\_resultat).  
supprime taille éléments à partir du pième compris

## **insert**

```
insert(liste1, liste2, position, liste_resultat).
insère les éléments de la liste 1 devant le pième élément de la liste 2.
si p = 1 : en tête
si p > length(liste2) : en queue
soit donc pour insérer un seul objet, qui peut être une liste :
    insert([objet] , liste2, position , liste_resultat)
avec des chaînes de caractères :
    insert("bucko","nadosor",3,_nabuckodonor)
```

## **uper**

## **lower**

## **compact**

```
compact (liste1,liste2).
supprime les doubles espaces et tabulations de la liste 1
```

## **revers**

```
revers (liste1,liste2).
en vérité, je vous le dis : les premiers seront les derniers

on aurait pu programmer :
revers(_liste , _inv) -> invers(_liste , [] , _inv) .
invers([], _1 , _l) -> / .
invers([_debut , !_fin] , _1 , _liste) ->
    invers(_fin , [_debut , !_l] , _liste) .
```

## **miroir**

```
miroir (liste1,liste2).
encore mieux, les sous-listes sont aussi inversées.
```

On aurait put programmer :

```
miroir([] , []) -> / .
miroir(_x , [_x]) -> notliste(_x) , / .
miroir([_début , !_fin] , [_mirfin , !_mirdébut]) ->
    miroir(_début , _mirdébut) ,
    miroir(_fin , _mirfin) .
```

```
list([_x , !_fin]) .
```

```
notlist(_x) -> list(_x) , / , fail.
notlist(_x) .
```



## flat

flat(liste1,liste2).

les sous\_listes sont aplaties : tout le monde au même niveau

on aurait pu programmer :

```
flat([], []) -> / .
flat(_x, [_x]) -> notliste(_x) , / .
flat([_x, !_fin], _resu) ->
    flat(_x, _platx) ,
    flat(_fin, _platfin) ,
    conc(_platx, _platfin, _resu) .
```

ou encore :

```
flat(_x, _y) -> flat(_x, [], _y) .
flat([], [], []) -> / .
flat([_x, !_suite], _s, _y) ->
    list(_x) ,
    flat(_x, [_suite, !_s], _y) .
flat([_x, !_suite], _s, [_x, !_y]) ->
    notliste(_x) ,
    flat(_suite, _s, _y) .
flat([], [_x, !_suite], _y) ->
    flat(_x, _suite, _y) .
```

quelques programmes :

## select

le sous programme **select** permet d'extraire un élément \_x d'une liste, et rend ce qui reste.

```
select(_x, [_x, !_reste], _reste) .
select(_x, [_y, !_suite], [_y, !_fin]) ->
    select(_x, _suite, _fin) .
```

## append

append donne toutes les solutions pour couper une liste en deux et permet donc de concaténer deux listes.

```
append([], _liste, _liste) -> / .
append([_debut, !_fin], _liste, [_debut, !_reste]) ->
    append(_fin, _liste, _reste).
```

## supprime

supprime toutes les occurrences d'un élément dans la liste :

```
supprime_all([], _x, []) -> / .
supprime_all([_objet, !_liste], _objet, _sansobjet) ->
    supprime_all(_liste, _objet, _sansobjet) , / .
supprime_all([_x, !_liste], _objet, [_x, !_sansobjet]) ->
    supprime_all(_liste, _objet, _sansobjet).
```

supprime une seule occurrence d'un élément dans la liste :

```
supprime_one([_], [_x], [_]) -> / .  
supprime_one([_objet, !_liste], _objet, _liste) -> / .  
supprime_one([_x, !_liste], _objet, [_x, !_sansobjet]) ->  
    supprime_one(_liste, _objet, _sansobjet').
```

### consecutif

test si deux éléments sont consécutifs dans une liste :

```
consecutif(_x, _y, [_x, _y, !_suite]) -> / .  
consecutif(_x, _y, [_deb, !_fin]) ->  
    consecutif(_x, _y, _fin) .
```

### permutation

```
permutation([_], [_]) -> / .  
permutation(_x, [_y, !_z]) ->  
    select(_y, _x, _t),  
    permutation(_t, _z) .
```

une autre façon de faire :

```
permutation([_x], [_x]) -> / .  
permutation([_x, !_y], _t) ->  
    permutation(_y, _z),  
    insère(_x, _z, _t).
```

```
insère(_x, [_], [_x]) -> / .  
insère(_x, [_y, !_z], [_x, _y, !_z]) .  
insère(_x, [_y, !_z], [_y, !_t]) -> insère(_x, _z, _t) .
```

une autre méthode :

```
permutation([_], [_]) -> / .  
permutation(_list, [_debut, !_fin]) ->  
    append(_v, [_debut, !_u], _list),  
    append(_v, _u, _w),  
    permutation(_w, _fin) .
```

### traitement des ensembles

il n'y a pas d'ensemble dans Prolog, mais avec les primitives suivantes, les listes sont traitées comme des ensembles :

#### listset

```
listset (liste1, liste2).
```

supprime les éléments en double dans la liste 1. c'est à dire que l'on peut considérer la liste comme un ensemble. les éléments sont réordonnés.

on aurait pu programmer :

```
listset([] , []) -> /.  
listset([_x , !_fin] , _resu) ->  
    element(_x , _fin) , / ,  
    listset(_fin , _resu) .  
listset([_x , !_fin] , [_x , !_queue]) ->  
    listset(_fin , _queue) .
```

## **inclus**

inclus (liste1,liste2).

on aurait pu programmer :

```
inclus([] , _ensemble)-> /.  
inclus([_debut , !_fin] , _ensemble) ->  
    element(_debut , _ensemble) ,  
    inclus(_fin , _ensemble).
```

indique si les éléments de la liste 1 sont tous éléments de la liste 2

## **inter**

inter (liste1,liste2,liste3).

rend dans la liste 3 les éléments communs à la liste 1 et à la liste 2

on aurait pu programmer :

```
inter([] , _ensemble , []) -> /.  
inter([_debut , !_fin] , _ensemble , [_debut , !_reste]) ->  
    element(_debut , _ensemble) ,  
    inter(_fin , _ensemble , _reste) , / .  
inter([_debut , !_fin] , _ensemble , _inter) ->  
    inter(_fin , _ensemble , _inter).
```

## **union**

union (liste1,liste2,liste3).

rend dans la liste 3 tous les éléments de la liste 1 et de la liste 2  
(c'est équivalent à conc puis listens)

on aurait pu programmer :

```
union([] , _ensemble , _ensemble) -> /.  
union([_debut , !_fin] , _ensemble , _union) ->  
    element(_debut , _ensemble) , / ,  
    union(_fin , _ensemble , _union).  
union([_debut , !_fin] , _ensemble , [_debut , !_union]) ->  
    union(_fin , _ensemble , _union).
```

## **complement**

complement (liste1,liste2,liste3).

rend dans la liste 3 tous les éléments de la liste 2 qui ne sont pas dans la liste 1.

```
complement([], _liste, _liste) / .  
complement([_debut, !_fin], _liste, _comp) ->  
    element(_debut, _liste) / ,  
    supprime(_debut, _liste, _inter) ,  
    complement(_fin, _inter, _comp) .  
complement([_debut, !_fin], _liste, _comp) ->  
    complement(_fin, _liste, _comp).
```

### **compset**

```
compset (liste1, liste2).  
indique si les deux ensembles sont égaux
```

### **pushfirst**

```
pushfirst (identificateur, objet).  
ajoute l'objet en tête de la liste assignée à l'identificateur
```

### **pushlast**

```
pushlast (identificateur, objet).  
ajoute l'objet en queue de la liste assignée à l'identificateur
```

### **pushdiff**

```
pushdiff (identificateur, objet).  
  
ajoute l'objet en tête de la liste assignée à l'identificateur, s'il n'est pas déjà dans cette liste.
```

### **pop**

```
pop (identificateur, objet).  
retire les objets de la liste assignée à l'identificateur
```

```
pop_one (identificateur, objet).  
retire le 1er objet de la liste assignée à l'identificateur
```

l'identificateur doit avoir comme valeur une liste, qui lui a été assignée par l'une des instruction :=, ==, pushfirst pushlast, ou pushdiff. avec pushfirst, on gère une pile dernier entré, premier sorti et avec pushlast, premier entré, premier sorti.

### **sort**

```
sort(liste_a_trier, _liste_triée).
```

si les éléments de la liste sont des noms, c'est l'ordre alphabétique qui sert. si les éléments de la liste sont des listes, le tri se fait sur le premier argument des listes.

on aurait pu programmer :

### tri naïf

on calcule les permutations de la liste, et on vérifie si elles sont triées !!!

```
tri_naïf(_liste , _liste_triée) ->
    permutation(_liste , _liste_triée) ,
    ordonnée(_liste_triée) .
```

```
ordonnée( [_x] ) -> / .
ordonnée( [_x , _y , !_suite] ) ->
    _x <= _y ,
    ordonnée( [_y , !_suite] ) .
```

### tri par insertion

on insère les éléments les uns après les autres, en respectant leur classement.

```
tri_insertion( [] , [] ) -> / .
tri_insertion([_x , !_suite] , _resu) ->
    tri_insertion(_suite , _inter) ,
    inserer(_x , _inter , _resu) .

inserer(_x , [] , [_x]) -> / .
inserer(_x , [_y , !_suite] , [_x , _y , !_suite] ) ->
    _x <= _y .
inserer(_x , [_y , !_suite] , [_y , !_inter] ) ->
    _x > _y ,
    inserer(_x , _suite , _inter) .
```

### tri par échange, bubble sort

◊on échange les éléments voisins non classés.

```
◊tri_echange(_liste , _triée) ->
    append(_a , [_x , _y , !_fin] , _liste) ,
    _x > _y , / ,
    append(_a , [_y , _x , !_fin] , _inter) ,
    tri_echange(_inter , _triée) .
tri_echange(_liste , _liste) ->
    ordonnée(_liste) .
```

### tri rapide

on sépare la liste en deux, d'une part les éléments inférieurs au 1er élément et d'autre part les supérieurs.

```
tri_rapide([], []) -> / .
tri_rapide([_x , !_suite] , _resu) ->
    séparer(suite , _x , _petits , _grands) ,
    tri_rapide(_petits , _tpetits) ,
    tri_rapide(_grands , _tgrands) ,
    conc(_tpetits , [_x , !_tgrands] , _resu) .
```

```

séparer([], _x, [], []) -> /.
séparer([_x, !_suite], _y, [_x, !_petits], _grands) ->
    _x <= _y,
    séparer(_suite, _y, _petits, _grands) .
séparer([_x, !_suite], _y, _petits, [_x, !_grands]) ->
    _x > _y,
    séparer(suite, _y, _petits, _grands) .

```

une autre version , sans aucun append (conc):

```

tri_rapide([], _x, _x) -> /.
tri_rapide([_h, !_t], _s, _x) ->
    séparer(_t, _h, _u1, _u2),
    tri_rapide(_u1, _s, [_h, !_y]),
    tri_rapide(_u2, _y, _x) .

```

la même version , faisant apparaître des "différences de liste" :

```

tri_rapide([], _x - _x) -> /.
tri_rapide([_h, !_t], _s - _x) ->
    séparer(_t, _h, _u1, _u2),
    tri_rapide(_u1, _s - [_h, !_y]),
    tri_rapide(_u2, _y - _x) .

```

## fusion

fusionne deux listes ordonnées

```

merge([], _x, _x) -> /.
merge(_x, [], _x) -> /.
merge([_a, !_b], [_m, !_n], [_a, _m, !_inter]) ->
    _a = _m, /,
    merge(_b, _n, _inter) .
merge([_a, !_b], [_m, !_n], [_a, !_inter]) ->
    _a < _m, /,
    merge(_b, [_m, !_n], _inter) .
merge([_a, !_b], [_m, !_n], [_m, !_inter]) ->
    _a > _m,
    merge([_a, !_b], _n, _inter) .

```

## Prédicats du second ordre

ils s'occupent des propriétés d'ensemble plutôt que de propriétés d'éléments.

### findall

```
findall(_variable , prop(_variable) , _liste) .  
findall_diff(_variable , prop(_variable) , _liste) .
```

rend la liste, avec ou sans double, des éléments ayant la propriété prop.

la programmation est :

```
findall(_variable , _prop(_variable) , _liste) ->  
    find(_variable , _prop(_variable) , [] , _liste).  
find(_variable , _prop(_variable) , _inter , _liste). ->  
    _prop(_variable) ,  
    notelement(_variable , _inter) , / ,  
    find(_var , _prop(_var) , [_variable , !_inter] , _liste) .
```

cette programmation est très inefficace, on parcourt la relation propriété N fois.

une autre programmation en se servant d'une relation intermédiaire "bag" :

```
findall(_x , _prop(_x) , _l) ->  
    _prop(_x) ,  
    assert( bag(_x) , [] ) ,  
    fail.  
findall(_x , _prop(_x) , _l) ->  
    find([], _liste).  
find(_in , _out) ->  
    retract(bag(_x) , [] ) , / ,  
    find([_x , !_in] , _out) .  
find(_in , _in) .
```

on aurait pu la programmer mieux, mais avec push :

```
findall(_x, prop(_x) , _l) ->  
    find := [] ,  
    prop(_x) ,  
    push_first(find , _x) , /* ou push_diff */  
    fail .  
findall(_x, prop(_x) , _l) ->  
    _l := find ,  
    find := [] .
```

### set\_of

On distingue **set\_of** , l'ensemble des éléments ayant la propriété Prop (sans double) et **bag\_of** , la liste avec doubles éventuels.

```
set_of(_x , _prop(_x) , _ensemble) ->  
    bag_of(_x , _prop(_x) , _liste) ,  
    listset(_liste , _ensemble) .
```

Application : soit à modifier une relation rel pour qu'elle soit triée selon un de ses paramètres. trions par ordre alphabétique la relation rel :

rel(nom , age , profession).

```
trie -> findall(_param , rel(_param,_age,_profession) , _liste) ,
        sort(_liste , _ordonnée) ,
        element(_elem , _ordonnée) ,
        retract_one(rel(_elem,_a,_p) , _y) ,
        assert(rel(_elem,_a,_p) , _y) ,
        fail .

trie .
```

Soit maintenant à construire la liste des noms et professions des gens de plus d'un certain âge , à la trier par profession et la traiter.

```
rel(titi,53,tolerie).
rel(moi,38,informatique).
rel(toto,37,politique).
rel(tata,52,informatique).
rel(tyty,54,peinture).
rel(tutu,55,peinture).
```

```
trie ?          /* ordonne la relation rel par ordre alphabétique */
```

```
constr ->findall(_couple , prop(_couple) , _liste) ,
        sort(_liste , _liste_resul) ,
        writeln('il faut',' licencié :') ,
        profession:=",
        element([_profession,_nom] , _liste_resul) ,
        _nom \= moi ,          /* deux précautions */
        _profession \= informatique , /* valent mieux qu'une */
        if(_profession \== profession) ,
            writeln(' ',_profession) ,
            end ,
        profession := _profession ,
        writeln(' ',_nom) ,
        fail .

constr .
```

```
prop([_profession,_nom]) ->
    rel(_nom,_age,_profession) ,
    _age >= 50 .
```

on obtient :

```
constr ?
    peinture
        tutu
        tyty
    tolerie
```



titi

Le résultat est trié alphabétiquement par profession et de même dans chaque profession.  
une autre solution consiste à faire :

```
constr -> findall_diff(_profession, rel(_n, _a, _profession), _liste) ,
        sort(_liste , _liste_resul) ,
        writeln('il faut','licencier :') ,
        element(_profess , _liste_resul) ,
        _profess \= informatique ,
        writeln(' ',_profess) ,
        findall(_nom , prop([_profess,_nom]) , _l) ,
        element(_nn , _l) ,
        _nn \= moi ,
        writeln(' ',_nn) ,
        fail .

constr .
```

### **size\_of**

```
size_of(_x , _prop(_x) , _nombre) ->
    set_of(_x , _prop(_x) , _ensemble) ,
    _nombre := length(_ensemble) .
```

### **for\_all**

```
for_all(_prop(_x) , _condition(_x)) ->
    set_of(_x , _prop(_x) , _set) ,
    check(_set) .
check([]) -> / .
check([_x , !_reste] , _condition(_x)) ->
    condition(_x) ,
    check(_reste) .
```

### **has\_property**

```
has_property([_x , !_reste] , _prop(_x)) ->
    prop(_x) ,
    has_property(_reste , _prop(_x)) .
```

### **map\_list**

```
map_list([], [], _match) -> / .
map_list [_x , _restex] , [_y , !_restey] , _match(_x , _y)) ->
    match(_x , _y) ,
    map_list(_restex , _restey , _match(_x , _y)) .
```

### **numbervars**

```
ground(_x) -> numbervars(_x , 0 , 0) .
```

```
numbervars( var(_n) , _n , _n1) ->
```

```

    _n1 := _n + 1 .
numbervars(_term , _n1 , _n2) ->
    novar(_term) ,
    functor(_term , _nom , _nombre) ,
    numbervars(0 , _n , _term , _n1 , _n2) .

```

```

numbervars(_n , _n , _term , _n1 , _n1) -> / .
numbervars(_i , _n , _term , _n1 , _n3) ->
    _i < _n ,
    _i1 := _i + 1 ,
    arg(_i1 , _term , _arg) ,
    numbervars(_arg , _n1 , _n2) ,
    numbervars(_i1 , _n , _term , _n2 , _n3) .

```

### variant

```

variant(_term1 , _term2) ->
    notnoty.

```

```

y ->
    numbervars(_term1,0,_nb) ,
    numbervars(_term2,0,_nb) ,
    _term1 = _term2 .

```

```

noty -> y , / , fail.
noty.

```

```

notnoty -> noty , / , fail .
notnoty.

```

### freeze

```

freeze(_x , _term) ->
    copy(_x , _term) ,
    numbervars(_term , 0 , _n) .

```

### genère nom

```

genère_nom(_prefix , _nom) ->
    var(_v) ,
    ident(_prefix) ,
    get_value(gensym , _n) ,
    _n1 := _n + 1 ,
    set_flag(gensym,_n1) ,
    name(_prefix , _prefix_s) ,
    name(_n1 , _n1_s) ,
    append(_prefix_s , _n1_s , _nom_s) ,
    name(_nom , _nom_s) .

```

**/\* pour gérer en prolog pur  
des variables globales au programme \*/**

```
set_flag(_nom , _valeur) ->  
    retract( flag(_nom , _v) , [] ) , / ,  
    assert( flag(_nom , _valeur) , [] ) .  
set_flag(_nom , _valeur) ->  
    assert( flag(_nom , _valeur) , [] ) .  
  
get_value(_nom , _valeur) ->  
    flag(_nom , _valeur) , / .  
get_value(_nom , 0) .
```

### **type d'un objet**

ces directives permettent de tester la nature d'un objet (pas de son contenu)

#### **liste**

liste (objet).  
retourne vrai si l'objet est une liste

#### **nil**

nil (objet).  
retourne vrai si l'objet est la liste vide.

#### **number**

number (objet).  
retourne vrai si l'objet est un nombre.

#### **real**

real (objet).  
rend vrai si l'objet est un réel

#### **integer**

integer (objet).  
rend vrai si l'objet est un entier

#### **digit**

digit (objet).  
rend vrai si l'objet est un chiffre

#### **letter**

letter (objet).  
rend vrai si l'objet est une lettre

#### **special**

special (objet).  
rend vrai si l'objet est un caractère spécial (ni lettre, ni chiffre).

#### **var**

var (objet).  
retourne vrai si l'objet est une variable

**nonvar**

nonvar (objet).  
 retourne vrai si l'objet n'est pas une variable

**ident**

ident (objet).  
 retourne vrai si l'objet est un identificateur

**structure**

structure (objet).  
 retourne vrai si l'objet est un terme structuré.

**numeric**

numeric (objet).  
 retourne vrai si l'objet est une expression calculable

**unification et affectation**

Prolog est un langage symbolique, et il importe de distinguer le nom d'une variable, son contenu et sa valeur, ce que les langages traditionnels ne font pas, il ne manipulent que des contenus. lorsque l'on doit évaluer une expression, on prend en compte la valeur des identificateurs qu'elle contient. une expression non arithmétique a pour valeur elle même.

:=

variable := expression  
 identificateur := objet.

variable ::= expression

**la variable est unifiée à l'expression, non évaluée**

identificateur := objet.

associe l'objet à l' identificateur. cet objet peut être quelconque. attention : si c'est une expression arithmétique, elle n'est pas évaluée et reste sous forme symbolique. c'est le "assign" du prolog classique.

a := 2.  
 b := [2,3,[7,8],4].  
 c := 2+3.  
 d := h  
     d contient h  
 e ::= b  
     recopie le contenu de b dans e  
     e contient [2,3,[7,8],4]

::=

variable ::= expression  
 identificateur ::= objet

### **l'expression est évaluée et s'unifie à la variable**

c'est le "is" du prolog classique, c'est l'affectation des langages traditionnels à une variable locale.

```
_y ::= 3 + 1..  
    _y vaut 4  
_x ::= _y + 2..  
    _x vaut 6
```

identificateur ::= objet

la valeur de l'expression est associée à l'identificateur

c'est l'affectation des langages traditionnels à une variable globale.

```
a ::= 3+2  
    a contient 5  
b ::= a+1  
    b contient 6  
c ::= b  
    c contient 6  
d ::= e  
    e n'ayant jamais été initialisé, d contient e
```

pushlast(liste,p)

```
pushlast(liste,q)  
    liste contient [p,q]  
    on aurait pu écrire :  
        liste := [p,q]  
_x ::= liste  
    _x vaut [p,q]  
a ::= liste  
    a contient [p,q]
```

les fonctions "!=" et "===" permettent donc de gérer des variables globales en distinguant leur nom de leur contenu. "!=" n'évalue pas les expressions, "===" les évalue.

```
c := b  
    c contient b et vaut b  
b ::= a  
    c contient b et vaut a  
a ::= 1  
    c contient b, et vaut 1  
_x ::= c  
    _x vaut 1  
_x := c  
    _x vaut c  
value(c, _x)      rend le contenu non évalué de c  
    _x vaut b
```

c'est 'ach'ment puissant, mais pas tellement dans l'esprit Prolog.

**:=.**

identificateur :=. identificateur.  
variable :=. identificateur  
value(identificateur , variable).

**rend le contenu, non évalué, de l'identificateur.**

### comparaisons

objet = objet  
objet \= objet    ou objet objet  
compare symboliquement les deux objets, si ce sont des expressions arithmétiques, elles ne sont pas évaluées.  
1=3-2    est faux

objet == objet  
objet \== objet  
les expressions arithmétiques sont évaluées.  
1 == a-1            est vrai si a vaut 2 ( a :== 2 )  
\_x :== a , 2 == \_x            est vrai

objet < objet  
ou > ou <= ou >=  
les expressions arithmétiques sont évaluées (semblable à ==)

objet << objet  
objet <== objet

On ne compare pas les contenus ni les valeurs mais les noms. (semblable à =) si le premier est un nombre (donc sans nom) les objets sont évalués sinon cela n'aurait aucun sens. les objets sont des symboles et c'est l'ordre alphabétique qui définit la relation.

\_x = pierre ,  
\_x << richard ,            oui  
                 même si on a fait pierre := sylvain

\_x = 3 ,  
\_x << 5 ,            oui

\_x = pierre ,  
\_x < richard ,    ne veut rien dire à moins que l'on ait fait :  
                 pierre :== 5,  
                 richard :== 3,  
                 et ca rend d'ailleurs non  
         et si on l'a fait,  
         \_x << richard donne quand même oui.

résumé :

=        \=        <<        <<=  
           le contenu des objets n'est pas pris en compte  
           les expressions arithmétiques ne sont pas évaluées

==        \==        <        <=        >        >=  
           le contenu des objets est pris en compte  
           les expressions arithmétiques sont évaluées

**:=:**

          structure :=: liste  
 transforme la structure en liste, ou l'inverse  
           auteur(goscinny,rené,belgique) :=: \_x  
 donne \_x = [auteur,goscinny,rené,belgique]

et        \_x :=: [auteur,goscinny,rené,belgique]

ou        \_attributs = [goscinny,rené,belgique],  
           \_x :=: [auteur, !\_attributs]

donne \_x = auteur(goscinny,rené,belgique)

## **name**

          name (identificateur , "caractères").  
 transforme une liste de caractères en un identificateur ou l'inverse.  
           name(toto , \_x).  
 donne : \_x = "toto"

et        name(\_x , "toto")  
 donne : \_x = toto

## **function**

          function (\_fon(a,b,c) , identificateur).

Unifie \_fon à l'identificateur  
 Cela permet de créer dynamiquement une nouvelle fonction \_fon .  
           function(\_x( ) , cosinus).  
 donne : \_x = cosinus

## **functor**

          functor (\_struct , nom , \_nb\_arg).

Cela permet de créer une structure \_struct ayant \_nb\_arg variables  
 différentes non instanciées en argument.

          functor(\_fon , auteur , 3) ,  
 donne à \_fon la valeur : auteur(\_\_1 , \_\_2 , \_\_3)

et donc :            functor(\_fon , auteur , 3) ,  
                          \_fon ,  
                          arg(\_fon , 1 , \_nom) ,  
 donne le même résultat que :  
                          auteur(\_nom , \_\_2 , \_\_3)

La deuxième forme vous semble plus simple ? , je ne vous contredirai pas, mais il arrive que l'on ne puisse pas l'employer ou plutôt que l'on ne veuille pas l'employer pour avoir un programme général. Voir des exemples d'utilisation dans le compilateur SQL.

## **arg**

arg(rang\_arg , \_structure , \_objet)

unifie l'objet à l'argument spécifié dans la structure (si la structure est une liste, se rappeler qu'une liste est une structure à deux éléments : le premier éléments et la fin de la liste).

                         arg(2 , auteur(gosciny , rené , belgique) , \_prénom)  
 donne : \_prénom = rené

                         arg( 2 , auteur(\_1 , \_2 , \_3) , rené)  
 donne : \_2 = rené

Exemple d'utilisation, extrait de SQL :

```

functor(_struct , auteur , 3) ,
arg(3 , _struct , belgique) ,
_struct ,
    arg(1 , _struct , _nom) ,
    /* ou _struct = auteur(_nom , __pr , __pp) */
    writeln(_nom) ,
    fail .
écrit la liste des auteurs belges

```

## **replace\_arg**

replace\_arg(rang\_arg , \_structure , \_struct\_resultat , objet)

Crée une structure résultat égale à la structure de départ à l'exception d'un argument qui y est remplacé par l'objet spécifié.

donc si \_struct vaut : auteur(gosciny , rené , france)

replace\_arg( 3 , \_struct , \_maj , belgique) ,  
 replace(\_struct , [] , \_maj , []) .

a pour effet de corriger la base de donnée.



## **calcul arithmétique**

une expression arithmétique peut contenir une des fonctions suivantes:  
(fonctions au sens traditionnel, rendant une valeur arithmétique).

### **value**

value (identificateur).  
rend le contenu au 1er niveau.

### **addition +**

### **soustraction -**

### **multiplication \***

### **division / div mod**

pour mod, on aurait pu programmer :

```
modulo(_x , _y , _x) ->  
  _x < _y , / .  
modulo(_x , _y , _z) ->  
  _x1 := _x - _y ,  
  modulo(_x1 , _y , _z) .
```

**puissance ^**

**sin**

**cos**

**tg**

**cotg**

**round**

**trunc**

**exp**

**ln**

**log**

**fact**

**abs**

**sign**

**int**

**frac**

**arctg**

**arccos**

**arcsin**

**sh**

**ch**

**th**

**sqrt**

**random**

random (borne\_max).  
rend un entier compris entre 1 et borne\_max

**max\_fait**

max\_fait.  
rend un nombre entier égal au nombre de faits maximum enregistrables  
400 ou 2000 selon le pc  
2000 sur vax  
sur sdx

**index**

index  
rend l'index de la dernière règle utilisée, valeur négative qui sera  
utilisée pour réaccéder rapidement à cette règle par index(valeur).

## **option**

option(nom\_option)  
rend un entier égal à la valeur de l'option déclarée par  
%option nom\_option=valeur

## **ioresult**

ioresult  
rend la valeur du dernier status d'entrée sortie, ou la valeur des  
paramètres numériques de readlhm, menu, read\_modify.

## **param\_line**

param\_line(rang)  
rend le numéro de ligne où est affiché (par readlhm ou writelhm) le  
paramètre de rang spécifié.

## **ord**

ord(char)  
rend le code ascii du premier caractère du paramètre qui peut être une liste ou un nom.

## **length**

length (liste)  
rend la taille de la liste

length (objet)  
rend le nombre d'arguments + 1

pour une liste, on aurait pu programmer :  
length([], 0) -> /.  
length([\_debut ,!\_fin] , \_length) ->  
length(\_fin , \_long) ,  
\_length := \_long+1 .

## **locate**

locate (liste1,liste2,index).  
locate (liste1,liste2).  
rend la position de la liste 1 dans liste2 à partir de index (ou de 1)  
si elle n'y est pas, rend 0  
pour retrouver la position d'un objet , qui peut être une liste :  
locate([objet] , liste , index)  
soit donc avec des caractères :  
locate("bucko","nabuckodonosor") == 3

## calcul logique

### not

not (predicat)  
cela sert à nier un prédicat

la question :

homme(\_qui) , not (affamé(\_qui)) ?  
sigifie visiblement que l'on recherche les hommes rassasiés.

not n'est pas indispensable, on aurait put programmer :

**not**(\_p) -> \_p , / , fail.  
not(\_p).

### and

and(\_p , \_q) -> \_p , \_q.

### or

or(\_p , \_q) -> \_p , / .  
or(\_p , \_q) -> \_q.

### xor

xor(\_p , \_q) -> \_p , not(\_q) , / .  
xor(\_p , \_q) -> \_q , not(\_p).

## gestion des fichiers texte

### <>nom logique

<><>

<>les noms logiques sont associés à un fichier physique lors des commandes reset ou rewrite, et ils sont donnés dans tous les ordres d'entrée-sortie ultérieurs. si on les omet, un défaut est pris par prolog. le défaut peut être modifié par les commandes input et output.

<>trois noms logiques sont prédéfinis et non modifiables :

keyboard, associé généralement au clavier par le système hôte

con, associé généralement à l'écran, ou à un fichier (batch)

print, sur ibmpc, associé à l'imprimante.

par défaut, input et output sont associés à keyboard et à console

ioresult vaut le dernier status d'entrée sortie

### <>reset

<><> reset (nom\_logique,'nom\_de\_fichier').

le fichier spécifié doit exister, on s'apprête à le lire.

le nom logique ne peut pas être un des noms logiques prédéfinis.

### <>rewrite

<><> rewrite (nom\_logique,'nom\_de\_fichier').

le fichier spécifié est créé.

le nom logique ne peut pas être un des noms logiques prédéfinis.

### <>eoln

<><> eoln (nom\_logique).

eoln.

rend vrai en fin de ligne

### <>eof

<><> eof (nom\_logique).

eof.

rend vrai en fin de fichier

affiche ->

reset(help,'toto.dat') ,

repeat ,

readcars(help,\_x,1) ,

putln(\_x) ,

eof(help) ,

/ ,

close(help) .

### <>status

<><> status (nom\_logique,\_status)

status (\_status).

<>

<>

## <>close

<><> close (nom\_logique).

le nom logique ne peut pas être un des noms prédéfinis.

## <>output

<><> output ('nom\_logique').

<>cette commande spécifie le fichier par défaut à utiliser dans les ordres d'écriture à venir. le nom logique doit avoir subi une commande rewrite ou être prédéfini en écriture.

<> output.

ou output (console).

dirige vers l'écran.

pour diriger la sortie vers l'imprimante, faire :

output (print).

## <>input

<><> input ('nom\_logique').

<>cette commande spécifie le fichier par défaut à utiliser dans les ordres de lecture à venir. le nom logique doit avoir subi une commande reset ou être prédéfini en lecture.

<> input.

ou input (keyboard).

dirige vers l'écran.

## <>prompt

<><> prompt(mot).

modifie la chaîne d'invitation à frapper. (par défaut c'est ">")

"mot" peut être égal à "" (pas de chaîne d'invitation).

pour le remettre :

prompt('>').

## <>read

<><> read (nom\_logique, objet, objet, ...).

read (objet, objet, ...).

<>lecture d'objets prolog sur le fichier associé au nom logique spécifié ou à son défaut. le nom logique doit avoir subi un reset ou être "keyboard". si la lecture se fait depuis le clavier, deux possibilités selon que l'on a utilisé la commande :

<>

nosilent.

le nom de l'objet s'affiche suivi de >

c'est l'option par défaut

ou silent.

seul le > s'affiche

les objets entrés sont unifiés à la commande:

read (\_a,toto(\_b))

prolog pose deux questions auxquelles il faut répondre :

```
_a > valeur_de_a  
toto(_b) > toto(valeur_de_b)
```

d'ou      \_a = valeur\_de\_a

et        \_b = valeur\_de\_b

<><> la ligne frappée est analysée comme étant un objet prolog et est unifiée  
aux objets spécifiés

<><>

<>

<>

### <>write

```
<><><> write (nom_logique, objet,...).  
write (objet,...).
```

```
writeln (idem)
```

les chaînes de caractères sont éditées sous forme de liste :

```
write("c'est la différence avec put").
```

imprime :

```
[c,'e,s,t, ,l,a, ,d,i,f,f,é,r,e,n,c,e, ,a,v,e,c, ,p,u,t]
```

put aurait imprimé :

```
c'est la différence avec put
```

### <>ordres d'édérations étendus

<>

### <>put

```
<><> put (nom_logique, objet,...).  
put (objet,...).
```

```
putln (idem).
```

<>écriture d'objets prolog sur le fichier associé au nom logique spécifié ou à son défaut. le nom logique doit avoir subi un rewrite ou être "output" ou "print". impression d'objets. à la différence de write, les chaînes de caractères, qui sont des listes, ne conservent pas le [ (voir la commande write).

```
<> put("abcd").
```

```
ou put([a,b,c,d]).
```

donne :

```
abcd
```

### <>writewords

```
<><> writewords (liste).  
writewords (nom_logique,liste).
```

```
writewordsln (idem).
```

complète le trio avec write et put. affiche une liste en séparant les éléments par un espace.

```
writewords([ça,c'est,fort],mon,petit,pote).
```

```

donne : ça c est fort mon petit pote
write("ça c est fort",mon,petit,pote).
donne : [ç,a ,c ,e,s,t ,f,o,r,t]monpetitpote
writewords("ça c est fort",mon,petit,pote)
donne : ç a c e s t f o r t mon petit pote

```

<>

<>

### <>line

```

<><>   line (nom_logique, nombre).
        line (nombre).
saut de N ligne

```

### <>space

```

<><>   space (nom_logique, nombre).
        space (nombre).
impression de N espaces

```

### <>put\_chr

```

<><>   put_chr(entier).
        put_chr(print , entier).
convertit l'entier en ascii, et envoie la purée.

```

### <>get\_ord

```

<><><>  get_ord(_entier).
convertit l'ascii frappé en son code ascii, et renvoie le sel.

```

ord(char) est une fonction mathématique rendant le code ascii

### <>write\_format

```

<><>   write_format(nom_logique , objet , description).

```

la description est une liste contenant les éléments :

videonor, videoinv, textcolor(), backcolor(), cursor(,), longueur

la longueur est celle désirée pour le nombre ou le mot

par défaut, pas de padding (chiffres significatifs uniquement)

cadrage à gauche si >0

cadrage à droite si négatif

<>

<>

### <>ordres de lecture étendus

<>

### <>readcar

```

<><>   readcar (nom_logique, variable).
        readcar (variable).

```



## <>readkbd

<><> readkbd (variable).  
lecture bloquante du clavier caractère par caractère

## <>inkey

<><> inkey (variable).  
lecture non bloquante du clavier caractère par caractère  
rend faux si on n'a rien frappé au clavier

## <>readcars

<><> readcars (variable).  
readcars (variable , spec).

lit une ligne complète et rend la liste constituée des caractères.  
spec = 0 les tabulations sont transformées en espace  
les doubles espaces ou tabulations sont supprimés  
c'est le défaut  
1 tout est rendu  
silent et nosilent ont le même effet que pour read

<>

<>

## <>readwords

<><> readwords (variable).  
readwords (fichier , variable).  
readwords (variable , format).  
readwords (fichier , variable , format).

<>lit une ligne complète et rend la liste constituée des mots. les nombres sont convertis. les séparateurs sont espace, tabulation, virgule, point, .. ils sont supprimés ou non selon l'indicateur format. silent et nosilent ont le même effet que pour read si un fichier est spécifié, il faut avoir fait : reset(fichier , 'nom.fic').

<>

format =

0 ou 4 séparateurs supprimés, nombres convertis. c'est le défaut  
ca ne conserve rien.

donne : [ ca , ne , conserve , rien ]

1 les séparateurs sont rendus, pas les espaces et tabulations.  
ca les conserve, bien sur.  
[ ca , les , conserve , ' , bien , sur , ' . ]

2 les séparateurs (ou tabulations) sont rendus  
un seul espace est rendu  
et supprimé si avant ou après un autre séparateur)  
ca les conserve , bien sur .  
[ca , ' , les , ' , conserve , ' , bien , ' , sur , ' . ]

3 tous les espaces ou tabulations, différenciés  
ca les conserve, bien sur.

```
[ca,' ','',les,' ',conserve,',',bien,' ',sur,',']
```

si format<0 on fait l'echo si fichier<>keyboard et format vaut -format

si format>10, on lit des lignes jusqu'à trouver un '.' en fin de ligne  
et format vaut format - 10. le '.' final est rendu  
sur fin de fichier, ioresult = 1 et on rend [.] au lieu de []

si format > 100, on neglige les lignes commençant par "\*\*\*  
et format vaut format - 100

si format > 1000, les caractères ' et " sont traités selon la tradition  
et format vaut format - 1000.

```
'toto.dat' est un seul élément : toto.dat
"toto.dat"      est 8 éléments constitués des 10 lettres
                 les 2 " sont conservés.
```

si format vaut > 10000 le " " est transformé en sous liste  
"toto.dat" est 1 élément constitué de la liste des 8 lettres  
le " n'est pas conservé, sauf à l'intérieur de la liste

<>

<>

Par exemple si format vaut -11111, on lit des lignes jusqu'à un point final en faisant éventuellement l'echo, en sautant les lignes commentaires, en traitant ' et " et en conservant les autres séparateurs. C'est le format idéal pour analyser un langage structuré. Toutefois si votre langage est si bizarre que cela ne vous suffit pas, il vous faut utiliser readcar qui rend la liste de tous les caractères, et en faire ce que bon vous semble .

<>

```
readwords(_sql , -11111) ?
_sql >executer le fichier
>*      execution du fichier client
>*
>toto.dat 'client.dat' avec  "r0/;$" .
```

rend :

```
[executer, le, fichier, toto, ., dat, client.dat , avec , [r,0/,;,$] ]
```

Pour des jeux , le format idéal est 0 ,  
readwords(\_phrase , 0) ,

en plus, on peut supprimer tous les articles :  
complement([le,la,les,un,une,des,l] , \_phrase , \_lu) ,

Pour parachever notre oeuvre, on désire remplacer tous les é, è, ê  
par des e, pour cela, il suffit d'avoir fait :  
noaccent ,

<>

<>

On pourrait se passer des gadgets de format et lire plusieurs lignes en retirant soit même les commentaires et en attendant le '.' final, comme cela est fait dans l'exemple suivant (extrait de SQL) que vous me copierez 100 fois et me ferez signer par vos parents :

<>

En supposant que echo vaut oui ou non et que fichin vaut keyboard ou fichier et évoluent subtilement par ailleurs :

```
lire(_phrase) ->
    lire_ligne(_début) ,
    empile([], _début , _phrase) .

empile([], ['*', !_fin], []) -> / .      /* commentaire */
empile(_début , ['*', !_fin] , _phrase) -> / ,
    lire(_début , _phrase) .
empile(_début , [], _début) -> / .
empile(_début , _lu , _phrase) ->
    conc(_début , _lu , _ligne) ,
    lire(_ligne , _phrase).

lire(_phrase , _phrase) -> last(_phrase , '.'), / .
lire(_début , _phrase) ->
    write(' '),
    lire_ligne(_suite) ,
    empile(_début , _suite , _phrase).

lire_ligne(_ligne) ->
    fichin == fichier , / ,
    readwords(fichier , _ligne , 1) ,
    repasser_kbd(_ligne) .
lire_ligne(_ligne) ->
    readwords(_ligne , 1) .

repasser_kbd(_ligne) ->
    echo == oui ,
    writewordsln(_ligne) ,           /* faire l'echo */
    fail .
repasser_kbd(_ligne) ->
    eof(fichier) , / ,
    fichin := keyboard .           /* repasser sur le clavier */
repasser_kbd(_ligne) .
```

<>

<>

### <>saisie de paramètres, menus

<>

<>La saisie de données est une opération fastidieuse pour le programmeur, surtout si l'on fait un minimum de vérifications. D'où ces primitives qui ont, entre autres, l'avantage de présenter à l'opérateur des réactions semblables d'un programme à l'autre : reformulation de la question en cas d'erreurs, "?" pour l'affichage des caractéristiques du paramètre attendu, "\$" pour l'annulation. Si il n'y a pas d'adressage curseur spécifié (par l'option cursor) ou implicite (par les options next et num) dans les descriptions, ces primitives fonctionnent en mode ligne.

<>

## <>store

```
<><> store(ligne , colonne , colonne_val ,colonne_saisie , ecart).
```

Définit le formatage de la grille de saisie.

une grille de saisie en mode écran à la tête suivante :

n1	libellé1	valeur1	>saisie1
n2	libellé2	valeur2	>saisie2
n3	libellé3	valeur3	>saisie3

rubrique à modifier >

Donc store enregistre :

- la ligne où commencer à afficher les readlhm ou les writelhm dont la description contient : "next" . par défaut, cette position est mise à 3 lors de la création de la fenêtre.
- colonne\_val est la colonne où s'affiche la valeur (défaut = 35)
- colonne\_saisie est la colonne où sera saisie la valeur (défaut = 55)  
si colonne\_saisie <> colonne\_val, la valeur est recopiée en colonne\_val après la saisie (cela permet d'afficher une valeur initiale ou une valeur par défaut en position valeur).
- l'écart est le nombre de ligne+1 entre chaque affichage (1 par défaut)

<>

## <>readlhm

```
<><> readlhm(texte , variable , description).  
readlhm(texte , variable , description , suite_description).
```

Cette directive permet de saisir des informations tout en vérifiant leur cohérence.

Les descriptions sont des listes ou des identificateurs ayant reçu une liste comme valeur par : ident := [ ].

une liste contient un ou plusieurs des éléments :

integer	le paramètre est un entier (défaut) compris entre min et max (défaut = 32767)
real	le paramètre est un réel compris entre min et max
word	le paramètre est un mot de taille min max arrêt au premier séparateur (' ' ou ',') max est limité à 15 sur ibmpc, à 40 sur vax
list([mot1,mot2,...])	donne la liste des mots possibles
list(nom_list)	la liste peut être placée dans une var globale
nom_list	avec nom_list:=[mot1,mot2,mot3]
<> [mot1,mot2,mot3]	la liste peut être spécifiée directement
words	le paramètre est une phrase de taille min max sans tenir compte des séparateurs
yesno	le paramètre est oui, yes, non, no, o, y ou n
date	le paramètre est une date
time	le paramètre est une heure

nombre min	limites du nombre ou taille du mot
nombre max	ou écart pour la commande menu
oblig	le parametre est obligatoire (défaut)
facul	le parametre est facultatif
nosilent	affichage du texte même si anticipation (défaut)
silent	pas d'affichage si anticipation
numérote	pour writelhm et readlhm : la ligne est numérotée en vue du read_modify pour menu : les lignes sont numérotées (défaut sur vax) par défaut, menu en mode caractère
next	la question se pose sur la ligne suivante et complète le menu affiché. (voir les commandes store, writelhm, readlhm et read_modify)
num(N)	la question se pose sur la ligne de ce numéro auquel cas, le texte n'est pas utile (") pour conserver le libellé qui y est affiché. pour un menu, indique la ligne à faire clignoter
cursor(ligne,colonne)	indique ou poser la question
clreol	effacement de la ligne avant affichage (défaut)
all	pas d'effacement
videonor	affichage en vidéo normale
videoinv	affichage en vidéo inverse
textcolor(couleur)	couleur du texte
backcolor(couleur)	couleur du fond

ce dernier groupe d'élément peut être doublé (séparation par un /)  
pour donner les mêmes informations pour la réponse.

<>

<>

si on ne désire qu'un élément, il est possible de le mettre à la place  
de la liste.

dans la première liste , on place la description du formatage de la ligne  
et dans la seconde (qui contient un "/" implicite ,la description du  
paramètre attendu

un readlhm est soit next soit num soit cursor  
un readlhm ne peut être numérote que si next

Si le buffer d'entrée n'est pas vide, il est analysé

Le texte de la question s'affiche comme avec un putwords

Si le paramètre est mal fourni par l'opérateur, un message s'affiche  
(en ligne 24 si cursor, next ou num a été spécifié), et la question est  
reposée.

Si l'opérateur frappe "?", um message d'aide

s'affiche (en ligne 24 si cursor ou next ou num a été spécifié).

ioresult prend les valeurs :

- 0 le paramètre a été fourni et est correct  
si le paramètre est un entier, il est rangé dans ioresult  
si c'est oui, ioresult vaut 1, et non : 0
- 1 le paramètre, facultatif, n'a pas été fourni
- 2 l'annulation ("\$\$") est demandée par l'opérateur

si ioresult est négatif, la variable est initialisée à 0 pour un nombre  
et à "" pour un mot.

```
readlhm('nombre de gateaux ',_nb , [integer,videoinv,cursor(10,5)]).  
readlhm([age,de,monsieur,_toto], _age , [integer,0,100,facul]).  
readlhm(numéro , _num , next).
```

<>

<>

### <>writelhm

<><> writelhm(libellé , valeur , liste\_description).

l'affichage est le même que pour un readlhm à ceci près que la valeur est  
ici donnée en paramètre.

<>avec plusieurs writelhm et éventuellement quelques readlhm "next", on présente une liste de libellé et  
de valeurs soit donc l'état d'une certaine entité. par read\_modify, on demande la valeur à modifier, et par  
readlhm "num(N)", on la modifie.

<>

pour afficher une valeur par défaut, poser la question correspondante  
puis effacer le défaut :

```
store(3,2,35,55,2) ,  
  
writelhm(libellé, défaut, [next, numérote]) ,  
readlhm(" , _valeur , [num(rang)]) ,
```

pour modifier la valeur sur une ligne déjà affichée :

```
writelhm(" , _valeur , [num(rang)]).
```

<>

### <>razlhm

<><> razlhm(numero).

Cette directive sert à effacer la fin de l'écran, à partir du rang de  
paramètre spécifiée.

### <>param\_line

<><> param\_line(rang)

c'est une fonction mathématique rendant le numéro de ligne où est affiché  
le paramètre de rang spécifié.

## **<>read\_modify**

```
<><>    read_modify(description).
        read_modify.
```

Le paramètre que frappe l'opérateur correspond à la numérotation des lignes writelm et readlhm. Il est rendu dans ioresult.

Si pas de spécification de curseur, la question se pose en bas de l'écran

<>

<>

## **<>menu**

```
<><>    menu( liste_libellés , liste_description) .
        menu( liste_libellés ) .
```

Le menu peut avoir 2 formes, selon que l'on a employé "numéroté" ou non

<>En mode numéroté , les libellés sont affichés à raison de un par ligne à partir de la première spécification cursor spécifiée ( ou 2,10) avec un écart de N entre chaque ligne (N est 1 ou le nombre spécifié dans la description) chaque ligne est numérotée en vidéo inverse à partir de 1. la question "votre choix" s'écrit selon la deuxième partie de la description et le nombre frappé est rendu dans ioresult.

<>

<>En mode non numéroté (utilisable uniquement sur ibm pc) les touches de déplacement du curseur font afficher une ligne selon la deuxième partie de la description ou en inverse video si aucune spécification n'a été précisée. Return valide le choix, \$ annule, esc rend facultatif non fourni. il est vivement conseillé de mettre le menu dans une fenêtre qui l'encadre car c'est joli joli.

<>

dans ces deux modes, la liste de libellés peut contenir des "/", ce qui fait passer à la colonne suivante.  
il y a 25 caractères par colonne.

ioresult vaut dans les deux cas :

```
>0 : choix, de 1 à N
-1 : facultatif non fourni
-2 : annulation
```

## **<>wait**

```
<><>    wait.
```

cette directive invite l'opérateur à frapper "return"

ioresult est initialisé à :

```
-2 si $
0 si return
```

## **<>write\_error**

```
<><>    write_error(libellé).
```

le message s'écrit en video inverse en bas de l'écran.

## **<>write\_title**

```
<><>    write_title(libellé,description).
        write_title(libellé).
```

le libellé s'écrit en vidéo inverse sur la première ligne

<>

<>

Exemple de programme créant des menus :

inilhm ->

```
iniwindow(5,2,3,22,60) ,
window(5) ,
textcolor(4) ,
backcolor(1) ,
clrscr ,
store(3,4,35,50,2) ,
/* essayer avec : store(3,4,35,35,2) , */
write_title([demonstration , de , lhm] , cursor(1,20)) ,
description := [next , numerote , textcolor(2) , backcolor(4) ,
/ , textcolor(4) , backcolor(7)] ,
description2 := [num(ioresult) , / , textcolor(5) , backcolor(2)] .
```

aff ->

```
readlhm(alphabeta , _x , description , [word,[m1,m2,m3]]),
readlhm(quezako , _y , description , [integer,3,4]),
writelhm(azertyuiop , 31 , description ) ,
writelhm(zorglub , 44 , description),
writelhm(musclor , 50 , description) ,
writelhm(fin , 66 , description) .
```

sous\_menu ->

```
razlhm(4) ,
writelhm(cidre , 52 , description) ,
writelhm(chouchen , ghj , description) ,
writelhm(bière , s(t,i) , description) ,
writelhm(pastis , 5 , description) ,
writelhm(porto , 50 , description) ,
wait .
```

menuons ->

```
inilhm ,
aff ,
repeat(boucle) ,
read_modify ,
_choix := ioreult ,
traite(_choix , _x) ,
fail .
```

menuons -> remenuons .

traite(1,\_x) -> readlhm(abcd,\_z,description2,[integer,textcolor(17)]),/.

traite(2,\_x) ->

```
list_mot := [mot1,mot2,mot3] ,
readlhm(" , _z , description2 , [word,list_mot]) , / .
```

traite(3,\_x) -> readlhm(" , \_z , description2 , word) , / .

traite(4,\_x) -> readlhm(" , \_z , description2 , [integer,8,99]) , / .

traite(5,\_x) -> readlhm(" , \_z , description2 , integer) , / .



```

traite(6,_x) -> sous_menu , fail(boucle) , / .
traite(_z,_x) -> _z == -1 , fail(boucle) .
traite(_z,_x) -> _z == -2 , write_error(annulation), beep, fail(boucle).
traite(_y,_x) -> write_error('non modifiable') .

```

<>

<>

```

remenuons->
    iniwindow(6,1,1,8,20) ,      window(6) ,
    textcolor(4) ,      bgcolor(7) .
remenuons ->
    clrscr , write_title([menu,mode,car] , cursor(1,3)) ,
    menu(['premiere action','deuxieme action','troisieme','et derniere'],
        [textcolor(2),bgcolor(4),/,textcolor(7),bgcolor(1)]) ,
    _x := ioresult ,
    write_format([ca,fait,_x],cursor(22,10)) , writeln ,
    wait .
remenuons ->
    window(5,1) ,
    write_title([demonstration , de , menu , numerote] , cursor(1,20)) ,
    menu(['premiere action','deuxieme action','troisieme','et derniere'],
        [numerote,textcolor(2),bgcolor(4),2]) ,
    _y := ioresult , write_format([ca,fait,_y] , cursor(22,10)) .

```

window(3,1).

```

/* pour faire executer les
primitives de gestion
de menu , faire :
    menuons?
ou      remenuons? */

```

Un autre exemple se situe dans sql, au bout des commandes "saisir" et "modifier" .

<>

<>

## <>gestion de l'écran

<><>ces commandes sont dirigées vers le nom logique output, l'écran.

### <>beep

```

<><>      beep (nombre).
sonnerie

```

### <>sound

```

<><><>      sound( fréquence_hertz) .
              sound( fréquence_hertz , durée_milli_seconde ) .

nosound.

```

par défaut, la durée est infinie, jusqu'à nosound.  
si la fréquence vaut 0, c'est le silence

139	156		185	208	233		
131	147	165	175	196	220	247	262

### **<>logo**

<><> logo(nom , programme , titre).  
affiche le logo sur l'écran

### **<>clrscr**

<><> clrscr.  
effacement de l'écran, la couleur du fond remplit tout.  
(ou de la fenêtre active uniquement, si il y en a une).

### **<>clreol**

<><> clreol.  
effacement de la ligne

### **<>clrtoend**

<><> clrtoend.  
effacement depuis le curseur jusqu'à la fin de l'écran (ou de la fenêtre)

### **<>videoinv**

<><> videoinv.  
passage en vidéo inverse

### **<>videonor**

<><> videonor.  
passage en vidéo normale

<>

<>

### **<>cursor**

<><> cursor (ligne,colonne).  
positionnement du curseur dans l'écran ou la fenêtre active

### **<>wherex**

<><> wherex  
fonction arithmétique qui rend la position du curseur

### **<>wherey**

<><> wherey  
fonction arithmétique qui rend la position du curseur



## **<>textmode**

<><>     textmode (mode).  
textmode.

## **<>textcolor**

<><>     textcolor (couleur).  
la valeur de la couleur est un entier compris entre 0 et 15  
au dela de 16, la couleur clignote.

## **<>backcolor**

<><>     backcolor (couleur).  
définit la couleur du fond en mode texte

## **<>color**

<><>     color.  
Cette directive permet de réinitialiser les couleurs du texte et du fond  
dans la fenêtre, lorsqu'elle les a perdu suite à videoinv ou videonor

## **<>prédicats graphiques**

<>voir la doc turbo pascal

## **<>graphcolor**

<><>     graphcolor.  
passage en mode graphique couleur basse résolution  
ligne de 0 à 199, colonne de 0 à 319  
0,0, c'est le coin gauche haut  
tortue : ligne de -99 à 100, colonne de -159 à 160  
0,0, c'est le centre de l'écran

## **<>graphmode**

<><>     graphmode.  
passage en mode graphique noir et blanc basse résolution  
mêmes coordonnées que graphcolor

<>

<>

## **<>hires**

<><>     hires  
passage en mode graphique haute résolution  
ligne de 0 à 199, colonne de 0 à 639  
0,0        c'est le coin gauche haut  
tortue : ligne de -99 à 100, colonne de -319 à 320  
0,0, c'est le centre de l'écran

### **<>hirescolor**

<><> hirescolor (couleur).  
définit la couleur du fond en haute résolution

### **<>palette**

<><> palette (numero\_palette).  
change toutes les couleurs en mode graphique couleur

### **<>graphback**

<><> graphback (couleur).  
définit la couleur du fond en mode graphique couleur  
<>

### **<>plot**

<><> plot (ligne,colonne,couleur).  
dessine un point

### **<>draw**

<><> draw (ligne,colonne,ligne,colonne,couleur).  
draw(ligne,colonne,couleur).  
dessine une droite

lorsque le 1er point est omis, on utilise le dernier point spécifié par la commande plot ou draw précédente.

### **<>forgetdraw**

<><> forgetdraw.

<>Fait en sorte que la commande draw(x,y,c) (ou drawpol) suivante ne tienne pas compte du point précédemment tracé et se transforme donc en plot(x,y,c).

<>

### **<>circle**

<><> circle (ligne,colonne,rayon,couleur).  
dessine un cercle

<>

<>

### **<>fillshape**

<><> fillshape (ligne,colonne,couleur,couleurbord).  
remplit une forme

### **<>fillscreen**

<><> fillscreen (couleur).  
remplit l'écran

## <>getdot

<><>     getdot (ligne,colonne)  
fonction arithmétique qui rend la couleur d'un point

## <>coordonnées polaires

<><>

## <>plotpol

<><>     plotpol(angle,rayon,couleur).

## <>drawpol

<><>     drawpol(angle,rayon,angle,rayon,couleur).  
           drawpol(angle,rayon,couleur).  
dessine une droite entre les deux points.  
même remarque que pour draw

## <>fillpol

<><>     fillpol( angle, rayon , couleurfond , couleurbord).

<>

<>

## <>tortue graphique

<><><><>la tortue est inspirée du langage logo créé par seymour papert. Elle sert à dessiner de jolis dessins.

les coordonnées de la tortue au départ sont 0,0 au centre de l'écran,  
elle est orientée vers le haut (angle=0).

il faut avoir fait :

           hires  
ou        graphmode  
ou        graphcolor

<><>logomode  
           le " ? " se pose en haut de l'écran, sans scroller  
prologmode  
           le " ? " se pose en bas de l'écran, en scrollant

## <>définition de la tortue

<><><><><><><><><><><>     setpencolor (couleur)  
           définit la couleur du dessin

           turtledelay (milliseconde)  
           définit la vitesse de la tortue

           initturtlewindow(ligne,colonne,ligne,colonne)  
           définit une fenêtre graphique en coordonnées tortue  
           turtlewindow (numero\_fenetre,spec)  
           se positionne dans la fenêtre

&lt; &gt;

```
forward(longueur)
turnright(angle)
```

polyinc (longueur , angle , nombre\_répétitions , incrément)  
idem en ajoutant l'incrément à la longueur  
et donc dessine des spirales

`fillshape(couleurfond , couleurbord)`  
remplit la forme ou se trouve la tortue

## <>fonctions mathématiques de la tortue

```
<><><><><>  xturtle
rend le numéro de ligne de la tortue
```

```
yturtle
```

rend le numéro de colonne de la tortue

heading  
rend l'angle

```
turtlethere
rend 1 si la tortue est visible dans la fenêtre
```

&lt;&gt;

<>

## <>mouvements de la tortue en 3 dimensions

[illegible]

```

tourner3d(angle)
    la tortue tourne sur sa gauche si l'angle est positif
tanguer3d(angle)
    la tortue bascule d'avant en arrière
roulis3d(angle)
    la tortue se penche sur le coté

```

tournerx(angle)  
 tournery(angle)  
 tournerz(angle)

`fillshape3d(couleurfond , couleurbord)`  
remplit la forme ou se trouve la tortue 3d

focal(distance\_focale)  
paramétrage de la perspective

fonctions mathématiques :

xturtle3d, yturtle3d, zturtle3d  
les coordonnées de la tortue 3d



pxturtle3d, pyturtle3d  
les coordonnées de la projection sur l'écran de la tortue 3d

<>

<>

## <>gestion des fichiers

<><>

### <>delete

<><> delete ('nom\_de\_fichier').  
ne marche que sur sdx et vax

### <>purge

<><> purge ('nom\_de\_fichier').  
ne fonctionne que sur sdx et vax

### <>copy

<><> copy ('nom\_fichier\_source','nom\_fichier\_destinataire').  
sur vax et pc, copie uniquement des fichiers textes  
sur sdx, copie tout fichier (il peut y avoir des \* )

### <>dir

<><> dir ('specification\_de\_fichier').  
dir .  
dir ('specification\_de\_fichier' , \_x).

dir('\*.\*pas') liste tous les fichiers de type pas  
dir('\*\')  
dir('b:\*.\') liste tous les fichiers de tous les sous directories  
dir('b:\*.\toto.dat') permet de rechercher le fichier toto dans tous les  
sous directories  
dir('/w')  
dir('/w b:') liste 5 fichiers par ligne  
dir('/t')  
dir('/t b:') donne l'arborescence des sous directories  
dir('/T b:') donne la liste des sous directories du courant

### <>type

<><> type ('nom\_de\_fichier').

<>cette commande permet de visualiser à l'écran le contenu du fichier texte spécifié, formaté à 22 lignes  
par écran.

<>

### <>print

<><><><> print (nom\_de\_sous\_programme).  
print ([nom\_de\_sous\_programme, ...]).  
print\_all.

```
print_module
```

```
print ('nom_de_fichier').
```

cette commande permet d'imprimer un sous programme ou un fichier (dont le nom contient un ".").

```
<>
```

```
<>
```

### **<>hardcopy**

```
<><>    hardcopy.
```

effectue une copie d'écran sur l'imprimante

si graphique, il faut avoir frappé la commande dos : graphics.

### **<>getpic**

```
<><>    getpic(nom_fichier).
```

crée un fichier et y place une copie d'écran graphique

### **<>putpic**

```
<><>    putpic(nom_fichier).
```

relit un fichier d'image d'écran graphique

### **<>imprimante hardcopy**

```
<><><><>    printon.
```

```
    printoff.
```

pilote le démarrage et l'arrêt de l'imprimante parallèle

ne fonctionne que sur vax et sdx

```
<>
```

```
<>
```

### **<>edt**

```
<><>    edt ('nom_de_fichier').
```

permet de créer ou modifier un fichier grace à un éditeur pleine page

voir la commande edit.

### **<>touches de fonction**

<>	home	debut de ligne
	end	fin de ligne
	pgup	initialiser la recherche de mot
	pgdn	chercher un mot
	ins	supprimer la ligne
	del	supprimer caractere vers la droite
	bs	supprimer caractere vers la gauche
	touches de déplacement du curseur	

## <>touches programmées

<>	f1	debut de ligne suivante
	f10	help

## <>fonctions esc

<>	esc ,	ligne courante en haut de l'écran
	esc :	ligne courante en bas de l'écran
	esc w	reaffichage de la page
	esc e	fin avec prise en compte
	esc q	fin sans prise en compte
	esc up	noter debut de block
	esc right	supprimer le block et le placer dans buffer.dat
	esc left	insérer le block buffer.dat
	esc home	debut de fichier
	esc end	fin de fichier
	esc pgup	page précédente
	esc pgdn	page suivante
	esc ins	remettre la ligne supprimée
	esc del	supprimer la fin de la ligne

## <>setdir

<><> setdir ('nom\_de\_directory').  
ne fonctionne que sur pc et sdx

<>

<>

## <>appel du système hôte

<>

### <>execute

<><> execute ('nom\_de\_fichier').  
sur pc et sdx, active le fichier spécifié. (.com ou .exe)  
sur vax, execute la commande spécifiée. ce peut être un truc ou @machin

### <>spawn

<><> spawn.  
ne fonctionne que sur vax  
lance un sub process lisant le clavier, ce qui a pour effet de rendre temporairement la main au vax et on revient par logout.

### <>user

<><> user1(paramètre1).  
user2(paramètre1,paramètre2).  
user3(paramètre1,paramètre2,paramètre3).  
user4(paramètre1,paramètre2,paramètre3,paramètre4).

<>Ces primitives servent à appeler des sous programmes écrits par l'utilisateur en pascal, assembleur, ... qui seront traités comme des prédicats prédéfinis (sans retour arrière possible)

<>les sous programmes se déclarent :

```
procedure user1(noeud      : ^pnoeud ;
                paramètre : integer);
```

par paramètre, on entend un numéro de noeud, et on l'utilise par

```
noeud^[paramètre].code
                    .fils
                    .frere
                    .value
                    .niveau
```

pour rendre des valeurs, il suffit de créer la valeur dans un ou plusieurs noeuds que l'on demande par

```
prendre_noeud(paramvaleur);
```

puis d'appeler la fonction d'unification :

```
if unifier(paramètre, paramvaleur) then;
```

<>en turbo il suffit d'insérer ces routines dans muser.pas sinon, il faut les compiler à part en y insérant les déclarations mdecl.pas et faire le lien par un link prolog,user,spawn sur vax ou par un bkb obj=prolog,obj=user sur sdx

<>

<>

<>

## <>aide au debug

<><>

<>les facilités de debug sont très grandes, puisque c'est l'interpréteur lui même : apres arrêt, on a accès à toutes les commandes.

## <>help

<><>

<> help (mot).

affichage d'un message d'aide extrait du fichier d'aide prolog.hlp expliquant le concept "mot".

help.

ou ?

affichage du fichier d'aide prolog.hlp

## <>touches de fonctions du help

<>	up,down,left,right,8,2,6,4	selection du choix du menu
	home,7	premier choix du menu
	end,1	dernier choix du menu
	return	validation du choix
	del_pc,pf3,.	recherche d'un mot
	ins,pf4	recherche d'un autre fichier d'aide
	esc,-	retour d'un niveau

## <>touches programmables

<>	f1	fenêtre avec l'écran complet
	f2, ,	fin
	f3	noir et blanc
	f4,tab	verifier la structure d'un fichier .hlp
	f5	afficher le 1er menu du fichier
	f6,espace	suite du texte en séquentiel
		cela permet de tout lire avec une seule touche
	f7	afficher le menu principal menu.hlp
	f8	retour au premier menu affiché
	f9,/	fin de help f10
	f10,del	help

## <>gestion des couleurs

<>si la première ligne du fichier .hlp est :

..

la fenetre est l'écran complet

le fichier help.cfg définit les couleurs à utiliser :

pas de fichier : couleurs par défaut

color_texte	: 1
color_menu	: 0
color_choix	: 10
color_fchoix	: 5
color_titre	: 11
color_ftitre	: 4
color_fond	: 7
color_ligne	: 1

fichier vide : noir et blanc

fichier contenant une ligne de 8 valeurs séparées par un espace

<>

<>

## <>activation du help

<>depuis l'éditeur par f10

depuis l'éditeur prolog par f10

depuis prolog, au clavier ou dans un programme, par

help.

help(mot).

hlp(nom\_fichier).

ou hlp(nom\_fichier , mot).

depuis un programme quelconque en pascal :

uses uhelp;

help(nom\_fichier , mot); (le mot peut valoir '')

## <>hlp

<><> hlp ('nom\_de\_fichier').

hlp ('nom\_de\_fichier' , mot).

cette commande permet d'utiliser à l'écran le fichier d'aide spécifié.

même effet que help, mais en utilisant le fichier utilisateur spécifié

## <>structure des fichiers d'aide

<>

<>help et hlp utilisent des fichiers texte, créés par votre éditeur préféré, vous savez: le dernier dsprintusny 4 , d'ashtland borton.

```
.mot
.mot_synonyme
ligne de texte 1
ligne de texte n
$      (ou $$)
choix1
choix3.hlp
/ghu/choix4.hlp
blabla-/kkk/ch.hlp,rubrique
blablabla-/kkk/ch.hlp,*
```

Les lignes de texte sont en nombre quelconque si la ligne de texte ne contient que \* cela force le passage à la page le fichier d'aide peut ne contenir que des lignes de textes (sans . ni \$ en colonne 1) il est alors simplement affiché et il faut frapper return toutes les 23 lignes les différents choix sont présentés en ligne sur 4 colonnes (\$) ou en colonne sur 2 colonnes (\$\$)

.\$\$ est conseillé si les mots sont longs

. \$ est conseillé si les mots sont nombreux.

Et on selectionne avec les touches de déplacement du curseur puis return le mot sur lequel on désire des informations. les choix peuvent être le nom d'un autre fichier d'aide on affiche alors sa première rubrique :

```
choix-=:toto.hlp
```

ou la rubrique spécifiée :

```
choix-=:toto.hlp,rubrique
choix-=:toto.hlp,*
```

( \* évite de répéter le mot)

<>

<>

il doit y avoir dans le fichier une rubrique et une seule pour chacun des choix qui ne sont pas des fichiers (vérification de la cohérence par F4)

## <>edit

<><><><>

```
<>      edit ('nom_sous_programme').
        edit_module.
        edit_all
```

l'éditeur de texte pleine page permet de modifier les faits et les règles (voir la commande edt).

## <>trace

<><> trace.

l'interpréteur signale ce qu'il fait, les règles qu'il utilise, dans

la mesure on on a utilisé la commande spy(nom\_sous\_programme).

### **<>notrace**

<><> notrace.  
c'est l'option par défaut

### **<>spy**

<><> spy (nom\_de\_sous\_programme).  
permet de tracer le sous-programme spécifié.

### **<>nospy**

<><> nospy (nom\_de\_sous\_programme).

### **<>stop**

<><> stop.  
Ca s'arrête, que dire de plus ? ah oui :

<>Lors de l'exécution d'un fichier par load(fichier), rend la main à l'opérateur. cela sert à faire un cours d'auto-apprentissage: on met des exemples dans le fichier et un "stop." les exemple s'affichent, puis on a la main à la console.on essaye et on termine par "continue." pour avoir la suite.

<>

### **<>continue**

<><> continue.  
Et ca repart, roulez jeunesse, roulez, roulez, roulez.

### **<>divers**

<>

### **<>delay**

<><> delay (milliseconde).  
mise en attente pendant N millisecondes.

<>

<>

### **<>date**

<><> date (variable).  
rend la date.  
write\_date.  
write\_date(nom\_logique).

### **<>time**

<><> time (variable)  
rend l'heure.  
write\_time.  
write\_time(nom\_logique).

## **<>seed**

<><> seed (nombre).  
initialise le générateur de nombres aléatoires, que l'on obtient par  
\_x := random(borne\_max).

<>

<>

## **<>indicateurs**

<>

## **<>echo**

<><> echo.  
pendant la lecture d'un fichier par load(fichier), il y a impression des  
lignes lues.

## **<>noecho**

<><> noecho.  
c'est l'option par défaut.  
suppression de l'impression des lignes lues sur les fichiers input

## **<>auto**

<><> auto.  
l'interpréteur donne toutes les réponses aux questions posées sans  
demander à chacune :  
on continue ?  
c'est l'option par défaut.

## **<>noauto**

<><> noauto.

## **<>fait**

<><> fait.

<>une clause sans queue est considérée comme un fait et s'ajoute aux précédents. c'est l'option par  
défaut.

<>

## **<>nofait**

<><> nofait.

<>les faits sont transformés en question et donc on ne peut plus ajouter de nouveaux faits. cela ne sert  
qu'en phase de test, et évite d'entrer par erreur des faits alors que l'on voulait poser des questions (en  
frappant "." au lieu de "?"). on commence par "loader" la base de données contenant les faits et les règles,  
puis on utilise "nofait" et on interroge la base.

<>

## **<>noaccent**

<><> noaccent.



<>cela sert a transformer toutes les lettres avec accent en les mêmes sans accent (é,è,ê,ë,à,ç,ï,ù devenant e,a,c,i,u)

<>

<>

<>

### <>nocolor

<><>      nocolor.

indique que l'écran n'est pas en couleur

### <>silent

<><>      silent.

les commandes read, readcars et readwords se font avec uniquement le ">"

### <>nosilent

<><>      nosilent.

les commandes read, readcars et readwords se font avec l'affichage du nom de l'objet.

### <>logomode

<><>      logomode.

l'invitation à frapper une commande s'affiche en haut de l'écran, pour éviter le scrolling de la page

### <>prologmode

<><>      prologmode.

l'invitation à frapper une commande s'écrit en bas de l'écran

<>

<>

## <>définition de grammaire

<><>

### <>parse\_until

<><>      parse\_until (phrase,debut,séparateurs,fin).

<>rend le début et la fin de la phrase (la fin contient l'un des séparateur)

<>

### <>parse\_space

<><>      parse\_space (phrase,reste).

supprime les espaces et les tabulations en tête de phrase

### <>parse\_string

<><>      parse\_string (phrase,chaine,reste).

### <>parse\_integer

<><>      parse\_integer (phrase,entier,reste).

<><>

<>

<><><><><><><><><><>

<>      %**option** symbol = expression

<>      %if symbol = expression

...

```
%else
```

...

**%end**

Ne pas confondre ce If avec le prédicat If dérivé du pascal.

<>

&lt; &gt;

**%case** symbol

?: valeur1

...

?: valeur2,valeur3,valeur4

...

%fcase

Quelques options sont prédéfinies :

l'option `ioresult` est aussi utilisable.

```
%goto nom_label
%backgoto nom_label
%label nom_label
```

branchements inconditionnels

280

équivalent à un %case sans %fcase et sans débranchement sur les %label suivants

<><><>par exemple, le fichier "exemple.pro" est conçu comme ceci :

```
%option exemple=1
%label debut
menu([exemple1,exemple2,fin] , num(option(exemple))).
%exemple=ioresult+1
%on ioresult
%label 3
cont.
%label 1
load('$:exemple1').
%backto debut
%label 2
load('$:exemple2').
%backto debut
```

<>Cette structure est due au fait que l'on désire charger les programmes au fur et à mesure, et en faisant des clear\_all. Une structure pure est aussi possible, voir le fichier "menulogo.pro" qui présente une succession de menus et se branche dans les sous-programmes correspondant, qui sont tous chargés en mémoire depuis le début.

<>

<> %page, %index, %cadre, %cadrepas, %\*, %paragraphe, %chapitre , ...

Ces commandes ne servent que pour la mise en page par le programme "doc".

<>

## <>rappel des commandes précédentes

<><>la commande .. sert à rappeler les 8 commandes précédentes.

on peut alors la modifier par des commandes analogues à celles de l'éditeur de texte.

end	: fin de ligne
home	: début de ligne
up	: commande précédente
down	: commande suivante
bs	: supprimer le caractère devant le curseur
del	: supprimer le caractère sous le curseur
x	: insérer le caractère
left	: déplacement vers la gauche
right	: déplacement vers la droite
pgup	: visualiser les 8 dernières commandes (enter sur vax)
pgdown	n : visualiser la commande n (0 sur vax)
return	: exécuter la nouvelle commande
esc	: ne rien faire

<>

## <>mise en oeuvre

<>  
<><>

### <>sur vax

<> define mosnay dua1:[mosnay.pro] prolog :== "\$mosnay:prolog" prolog prolog \$:exemple ou  
prolog toto

### <>sur ibmpc

<> myprol myprol exemple ou myprol toto si un fichier (.pro par défaut) est spécifié, il est lu, compilé  
et exécuté comme si on avait fait load('toto.pro'). on peut aussi faire , avec un \$ : myprol \$echo.  
load('toto.pro'). apres le \$, on peut placer toutes les commandes de prolog.

### <>sur sdx2

<>mettre 500 k de heap système @telprol pour télécharger les fichiers nécessaires modecr oui  
vt100 run prolog,stack=6000

### <>les exemples

<>Tous les exemples cités ci-après peuvent être exécutés en faisant :  
load('\$:exemple').

le "\$:" est remplacé par "mosnay:" sur vax, par [prol]" sur sdx2 et par rien sur ibmpc; cela sert à pointer  
sur les fichiers standards et d'exemple.

# Moteur d'inférence

## Fonctionnement

L'interpréteur va étudier les conditions de la question posée les unes après les autres pour déterminer si elles sont réalisées ou non. Dès qu'une condition n'est pas remplie, le sous-programme s'arrête et rend la valeur "faux". L'ordre d'analyse des conditions est, on l'a vu, théoriquement indifférent, mais Prolog effectue toujours son analyse de gauche à droite et en ce sens, le langage est procédural car dans la mesure où certaines conditions sont en fait des directives cela permet justement de préciser l'ordre dans lequel on veut les exécuter, autrement dit cela permet de préciser un algorithme.

La vérification d'une condition consiste à chercher des faits (ou des têtes de règles) qui s'unifient aux éléments de la condition. (si c'est une règle il faut alors vérifier toutes les conditions de la queue de la règle). Certaines conditions contenant des variables, Prolog s'efforce de donner à ces variables des valeurs telles que la condition soit réalisée. Les résultats obtenus pour démontrer les premières conditions servent pour vérifier les suivantes, c'est à dire que si la même variable apparaît dans plusieurs conditions, et que la variable a reçu une valeur dans les premières conditions, c'est cette valeur qui est utilisée pour la variable pour vérifier les conditions suivantes, et cette variable ne pourra plus changer de valeur. Et bien voilà autre chose ! on ne peut pas, dans le même sous-programme faire :

```
    _x := 4  
puis, un peu plus loin :  
    _x := 5  ?
```

Non, car les conditions précédemment vérifiées l'ont été avec la première valeur (4) et ne le seraient peut être pas avec la nouvelle (5), et cela conduirait à déduire tout et le contraire de tout.

Lorsque l'interpréteur a vérifié la dernière condition, c'est qu'il a trouvé une solution qui satisfait toutes les conditions. Il affiche alors la valeur qu'il a donné aux variables présentes dans la question ou simplement "oui" s'il n'y en avait pas. il lui reste alors à chercher d'autres valeurs pour les variables, pour avoir toutes les solutions. Si il ne peut trouver de solution, il affiche "non", qu'il faut parfois comprendre comme :

"je ne sais pas, ce fait n'est pas dans la base"

## Unification

C'est le mécanisme de base de la résolution. il consiste à chercher à mettre en correspondance deux objets (pattern matching). C'est à cette occasion que les variables sont substituées par ce qu'il faut pour que l'unification réussisse et donc pour que la condition soit

réalisée. On montre que si la substitution est possible, il existe une substitution minimal unique des variables qui rendent les deux termes égaux et toute autre substitution se décompose en un produit dont l'un des termes est la substitution minimale.

Soit à unifier :

homme(\_qqn) et homme(Nabuchodonosor) \_qqn s'unifie à Nabuchodonosor

$1 + \sin(\_x) - \_y(a,b)$  et  $1 + \sin(\_p) - \text{fonc}(\_p,b)$   
\_x s'unifie à \_p puis à 'a', soit donc à 'a' et \_y à 'fonc'

Le cas de la liste est intéressant, il faut bien comprendre le signe "!"

[1,2,\_reste] et [1,2,3] \_reste s'unifie à 3

[1,2,\_reste] et [1,2,[3,4]] \_reste s'unifie à [3,4]

[1,2,\_reste] et [1,2,3,4] échoue  
car si \_reste s'unifiait à [3,4] on serait dans le cas précédent

[1,2,!\_reste] et [1,2,3,4] \_reste s'unifie à [3,4]  
car "!" indique que \_reste est la liste égale à la fin de la liste

si \_phrase vaut [Pierre , donne , chocolat , à , Marc]  
alors, unifier \_phrase à [\_sujet , donne , \_qqchose , à , qqun]  
donne : \_sujet = Pierre, \_qqchose = chocolat et \_qqun = Marc

On voit donc comment l'on organiserait un petit programme chargé d'analyser les commandes d'un petit système informatique intégré :  
intégré -> repeat , readwords(\_phrase) , analyse(\_phrase) .

analyse([imprime , \_fichier , sur , l , imprimante] )->print(\_fichier).  
analyse([imprime , \_fichier , sur , l , écran] ) -> type(\_fichier) .  
analyse([calcule , !\_reste]) -> calcul(\_reste) .  
analyse(\_x) -> writeln("vous dites ?") .

calcul([ le , bénéfice , de , \_mois]) -> execute('benef.exe' , \_mois) .  
calcul([ la , paye]) -> paye.  
calcul(\_x) -> writeln("je ne sais pas calculer " , \_x) .

Le sous-programme "analyse" réalise, en beaucoup plus fort, ce que l'on fait par un "case" en pascal. L'élément du "case" n'est plus un simple entier mais un terme composé qui peut être très complexe, c'est ici la liste des mots de la phrase à analyser.

## Retour arrière

En cas d'échec à l'unification, et même en cas de succès, si l'on veut toutes les solutions, l'interpréteur est amené à revenir sur les choix qu'il a fait tant au niveau des variables unifiées que au niveau des règles des sous-programmes. il y a donc parcours de l'arbre de

résolution en arrière jusqu'au nœud correspondant au choix à refaire. Les variables qui avaient été substituées sont alors remises à l'état de variables.

### algorithme d'unification

On peut exprimer en Prolog ce que unifier deux choses veut dire :

```
unifier(_x , _y) ->
    var(_x) , _x = _y .
unifier(_x , _Y) ->
    var(_y) , _y = _x .

unifier(_x , _y) ->
    constant(_x) , constante(_y) , _x = _y .
unifier(_x , _y) ->
    structure(_x) , structure(_y) ,
    unifier_structure(_x , _y) .

unifier_structure(_x , _y) ->
    functor(_x , _fx , _nbarg) ,
    functor(_y , _fy , _nbarg) ,
    unifier_arguments(_x , _y , _nbarg).

unifier_arguments(_x , _y , 0).
unifier_arguments(_x , _y , _n) ->
    _n > 0 ,
    arg(_n , _x , _ax) ,
    arg(_n , _y , _ay) ,
    unifier(_ax , _ay) ,
    _n1 := _n - ! ,
    unifier_arguments(_x , _y , _n1) .
```

Pour empêcher une variable de s'unifier avec un terme qui la contient, certains préfèrent remplacer le début par :

```
unifier(_x , _y) ->
    var(_x) , var(_y) ,
    _x = _y.
unifier(_x , _y) ->
    var(_x) , nonvar(_y) ,
    not_occurs_in(_x , _y) ,
    _x = _y.
unifier(_x , _y) ->
    var(_y) , nonvar(_x) ,
    not_occurs_in(_y , _x) ,
    _x = _y.
```

et le reste comme dans la première version.

```

not_occurs_in(_x , _y) ->
    var(_y) , _x /= _y.
not_occurs_in(_x , _y) ->
    nonvar(_y) , constant(_y).
not_occurs_in(_var , _structure) ->
    nonvar(_structure) ,
    structure(_structure) ,
    functor(_y , _fy , _nbarg) ,
    not_occurs_in_args(_nbarg , _x , _y).

not_occurs_in_args(0 , _x , _y).
not_occurs_in_args(_n , _x , _y) ->
    _n > 0 ,
    arg(_n , _y , _arg) ,
    not_occurs_in(_x , _arg) ,
    _n1 := _n - ! ,
    not_occurs_in_args(_n1 , _x , _y) .

```



## Exemple de résolution

Soient les faits et les règles constituant les trois sous\_programmes suivants:

```
aime(Claude , nager).
aime(Catherine , nager).
```

```
parler_tout_le_temps(Claude).
```

```
boire(Philippe , petit_bordeaux_1948).
boire(c_est_pas_moi , la_tasse).
boire(_gus , la_tasse) ->
    aime(_gus , nager) ,
    parler_tout_le_temps(_gus).
```

Lors de la question :

```
boire(_qui , la_tasse)?
```

qui signifie visiblement :

qui c'est-t-y qui a bu la tasse ?

L'interpréteur choisit la première clause du sous-programme "boire" et la compare mot pour mot (l'unifie) à la question posée. on obtient donc d'abord \_qui = philippe. puis il y a échec, car faut pas confondre le pinard et le château la pompe. Il y a alors abandon de la clause et l'interpréteur choisit la deuxième clause du sous-programme boire. L'unification réussit et donne la première solution:

\_qui = c'est pas moi, c'est ma soeur qu'a cassé la machine à vapeur.

il y a tout de même abandon de la clause et choix de la troisième, qui est une règle. L'unification réussit en liant \_gus à \_qui et l'interpréteur cherche à démontrer les deux conditions de la règle. La première entraîne le déroulement du sous-programme "aime" et \_gus s'unifie avec "Claude". l'appel du sous programme "parler\_tout\_le\_temps" permet de satisfaire la deuxième condition et cela donne la deuxième solution :

\_qui = Claude

Il y a alors retour à la deuxième clause du sous-programme "aime" et \_gus s'unifie à "Catherine" mais la condition parler(Catherine) ne s'unifiant avec aucune clause du sous-programme "parler", il y a échec et retour arrière. et cette fois il n'y a plus de clause possible , ni pour aime, ni pour boire. Lors de la résolution, l'interpréteur affiche au fur et à mesure de chaque démonstration la valeur de substitution des variables non anonymes de la question posée.

soit dans notre cas :

\_qui = c'est pas moi  
\_qui = claude

Lors de la question :

boire(\_ , la\_tasse)?

qui signifie :

y a t-y quelqu'un qui a bu la tasse ?

la variable est anonyme, on ne s'intéresse pas à sa valeur et il n'y a pas d'autre affichage que "oui".

# rôle des caractères spéciaux

(	début des arguments d'un prédicat
)	fin des arguments d'un prédicat
,	séparateurs des arguments d'un prédicat ou des éléments d'une liste
, ou &	séparateur des prédicats d'une queue de clause (règle) ou des prédicats d'une question
[	début de liste
]	fin de liste
"	début et fin d'une chaîne de caractère
'	début et fin d'un identificateur contenant des caractères spéciaux
/	coupe des choix
!	séparateur entre tête et queue de liste
_	indicateur de variable
+	opérateurs arithmétiques
-	
*	
/	
^	
:=	affectation du nom (sans évaluation)
::=	affectation de la valeur (avec évaluation)
:=.	affectation du contenu (sans évaluation)
/*	début de commentaire
*/	fin de commentaire
->	séparateur entre tête et queue de règle
.	fin de fait, de règle ou de question
?	indicateur de question
<	comparateurs arithmétiques avec évaluation des deux membres
<= ou =<	
>	
>= ou =>	
=	égalité du nom sans évaluation
::=	égalité de la valeur avec évaluation
\=	différence sans évaluation
\:= ou <>	différence avec évaluation
<<	comparateurs symboliques sans évaluation
<<=	
::	transformation de liste en structure

les majuscules et les minuscules sont supposées distinctes. en particulier, les prédicats prédéfinis le sont en minuscule. l'espace ou la tabulation sont sans signification dans la mesure où ils ne coupent pas un identificateur ou une chaîne de caractères enclose de " ou '. une ligne peut ne pas être complète, la suite se trouvant sur la ligne suivante. une ligne peut contenir plusieurs commandes.