Chapitre 1

Introduction à Prolog

1.1 Introduction

Prolog est un langage de programmation basé sur la logique du premier ordre, il a été inventé au début des années 70 par Alain Colmerauer à Marseille justement dans le but de pouvoir faire du traitement de la langue naturelle, mais il s'est vite aperçu que ce langage pouvait avoir un champ d'application beaucoup plus large.

Cette section introduit très brièvement Prolog en utilisant des concepts et des exemples extraits du domaine du traitement de la langue naturelle. Nous employons la syntaxe dite *d'Edimbourg* (Clocksin Mellish 84), (Saint-Dizier 87), (Sterling et Shapiro 86),

Un programme Prolog se présente comme une suite de *règles* ou *clauses* de la forme suivante :

```
P :- Q1, Q2, ... , Qn.
```

qu'on interprète comme :

$$P \text{ est vrai si } Q_1, Q_2, \ldots, Q_n \text{ sont vrais.}$$

Une règle de la forme

```
P.
```

est appelé un fait car il n'y a aucune condition pour sa véracité. P est appelé la *tête* et $Q_1, Q_2, \ldots, Q_n$ le *corps* de la règle. Chacun de ces éléments est appelé un *littéral* composé d'un symbole de *prédicat* et de *paramètres* ou *arguments* entre parenthèses séparés par des virgules. Par exemple,

```
syntagmeNominal    adverbe(X)    pronom(je,1,Nombre)
```

sont des littéraux indiquant des relations entre les arguments. Les paramètres sont des *termes* composés de *variables* dénotant des objets à définir plus tard, des *symboles atomiques* ou des *termes composés*. Revenons sur ces notions :

variables notées par des identificateurs débutant par une majuscule ou un souligné ; par exemple :

```
X   Genre     _nombre    _
```

Une variable notée par un seul souligné indique que la valeur de la variable ne nous importe pas.

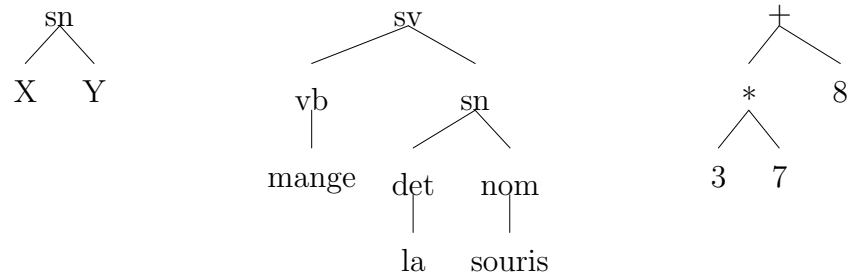
symboles atomiques notés soit par des nombres ou des identificateurs débutant par une minuscule.

```
325    3.1416    pluriel    masculin
```

termes composés notés par un *foncteur* et des paramètres qui sont eux-mêmes des termes, par exemple :

```
sn(X,Y)    sv(vb(mange),sn(det(la),nom(souris)))    +(*(3,7),8)
```

Ils représentent des arbres où le foncteur étiquette la racine et les paramètres les branches. Les arbres qui correspondent aux exemples précédents sont donc :



1.2 Notion de programme Prolog

Un programme Prolog est composé d'une suite de faits et de relations entre ces faits exprimées par des règles selon la syntaxe introduite dans la section précédente. Soit la liste suivante indiquant quelques *faits* du vocabulaire français :

```
determinant(la).
determinant(le).

nom(souris).
nom(chat).

adjectif(blanc).
adjectif(rouge).
adjectif(blanche).

genre(la,feminin).
genre(le,masculin).
genre(souris,feminin).
genre(chat,masculin).
genre(blanc,masculin).
genre(blanche,feminin).
genre(rouge,_).
```

L'utilisation d'une variable dans le dernier fait signifie que sa valeur ne nous importe pas. Les règles indiquent des relations entre des faits ; par exemple :

```
accord(X,Y) :- genre(X,Z), genre(Y,Z).
```

dit que `X` est du même genre que `Y` si le genre de `X` est d'un certain `Z` et ce `Z` est également le genre de `Y`. L'affectation à une variable Prolog est unique et ne se fait que par le passage de paramètre. Une fois que sa valeur a été fixée par un fait ou une règle, elle est répercutée à toutes les autres apparitions dans la règle. Dans l'exemple précédent, aussitôt que la première occurrence de `Z` est fixée par la recherche du genre de `X` alors la valeur de `Z` pour `Y` est fixée et on vérifiera alors que `Y` est bien du même genre que `X`. On exprime une alternative en donnant plusieurs possibilités de règles :

```
sn(X,Y):-determinant(X), nom(Y), accord(X,Y).
sn(X,Y):-nom(X), adjectif(Y), accord(X,Y).
```

indique une relation entre deux individus quelconques (`X` et `Y`) disant que le couple `X` et `Y` forment un syntagme correct soit si `X` est un déterminant et `Y` un nom, soit `X` est un nom et `Y` est un adjectif. Dans les deux cas, on vérifie l'accord des genres avec le prédicat défini plus haut. Les faits et les règles illustrés plus haut sont vraiment très simplistes et ne servent qu'à illustrer nos premiers exemples ; nous verrons plus loin d'autres méthodes plus efficaces de travailler.

L'exécution d'un tel programme est lancée par une question (ou *but*) posée à l'interprète Prolog qui va vérifier si ce qu'on demande est vrai ; cette question est donnée après l'incitation qui est :

```
| ?-
```

En Sicstus Prolog qui nous a servi comme outil de test des programmes donnés ici. Par exemple[1],

```
| ?- nom(chat).
   yes
| ?- nom(chien).
   no
| ?- accord(le,chat).
   yes
| ?- accord(chat,souris).
   no
| ?-
```

Le système répond par `yes` s'il trouve une façon de prouver le nouveau fait à partir des faits et des relations donnés dans le programme. Par exemple, le premier fait à prouver est vrai car il est présent dans notre liste de faits alors que le second n'y est pas. Dans le troisième cas, on utilise la règle `accord` pour trouver que les deux arguments sont du même genre alors que le quatrième échoue car les mots *chat* et *souris* ne sont pas du même genre (et non pas pour des raisons sémantiques). Dans le cas où on a mis des variables dans la question, on reçoit les valeurs des variables permettant de prouver le fait qu'on demande. Par exemple :

```
| ?- genre(souris,X).
X = feminin
```

[1]Dans les exemples d'interactions, ce qui est précédé de ?- correspond aux messages de l'usager et les lignes ne débutant pas par ce symbole sont les réponses du système

```
    yes
```

l'interprète demande alors si on désire les autres possibilités, si oui, on répond par ; et il tente de trouver une autre affectation des variables rendant ce fait vrai. Si on répond autre chose, alors le système répond `yes` pour indiquer qu'il y avait peut-être d'autres solutions. Pour obtenir tous les mots masculins dans notre banque, il suffit de faire :

```
| ?- genre(Mot,masculin).
Mot = le ;
Mot = chat ;
Mot = blanc ;
Mot = rouge ;
    no
```

le système répond par `no` lorsque toutes les possibilités ont été épuisées. Il faut remarquer que `rouge` est donné en réponse car il peut correspondre à n'importe quelle valeur.

```
| ?- sn(X,Y).
X = la
Y = souris ;

X = le
Y = chat ;

X = souris
Y = rouge ;

X = souris
Y = blanche ;

X = chat
Y = blanc ;

X = chat
Y = rouge ;

    no
```

L'ordre de sortie des résultats dépend de l'ordre d'apparition des faits et des règles. Il correspond à un parcours en profondeur des différentes possibilités : ainsi dans le dernier exemple, on débute par la première possibilité pour un `sn` soit un `déterminant` et un `nom` ; pour un déterminant on commence par le premier `la`, ensuite on tente de trouver un nom mais du bon accord. Ensuite, on reprend avec l'autre article jusqu'à épuisement de ces possibilités ; on passe à la deuxième possibilité de `sn` : on cherche tous les noms et pour chacun d'eux les adjectifs du bon genre.

On peut également donner plusieurs buts à résoudre et alors il faut que tous soient satisfaits ; par exemple, pour obtenir tous les noms masculins de notre dictionnaire :

```
| ?- genre(Mot,masculin),nom(Mot).
```

```
Mot = chat ;
   no
```

Le processus qui vérifie la conformité entre les paramètres est beaucoup plus général que celui qu'on rencontre dans la plupart des langages de programmation car les paramètres et les valeurs peuvent contenir des variables qui sont liées aux valeurs correspondant dans l'autre. Cette opération est appelée unification et est très pratique pour la construction de structures durant des processus d'analyse syntaxique. Par exemple :

```
p(snm(det(X),nom(Y),G),X,Y):-sn(X,Y),genre(X,G).
```

qu'on lance comme suit :

```
| ?- p(S,le,chat).
S = snm(det(le),nom(chat),masculin) ;
    no
| ?- p(S,X,rouge).
S = snm(det(souris),nom(rouge),feminin)
X = souris ;

S = snm(det(chat),nom(rouge),masculin)
X = chat ;

    no
```

Dans le premier cas, le genre s'est propagé dans la structure finale même s'il n'est déterminé qu'à la fin. Dans le deuxième cas, on cherche à trouver une structure de `sn` avec `rouge` en deuxième position. On voit qu'il suffit de définir la forme du foncteur qu'on veut créer pour que le processus d'unification fasse le nécessaire pour créer la structure.

1.3 Création de programmes

Pour créer une base de faits et de règles, on écrit normalement un fichier dont on demande le chargement par l'interprète ou le compilateur. Cette opération est appelée *consultation* d'un fichier et se commande en faisant exécuter le but

```
consult(fichier)
```

qui va interpréter le contenu de *fichier* comme une suite de clauses à ajouter à la banque de faits et règles du système. Il est souvent pratique de pouvoir remplacer un jeu de clauses par d'autres du même nom : c'est ce qui arrive lorsqu'on découvre une erreur dans un fichier, qu'on la corrige avec l'éditeur et fait interpréter le programme en s'attendant à ce que la nouvelle version des clauses remplace les anciennes au lieu de s'ajouter aux anciennes qui sont erronées. Cette opération de remplacement est commandée par

```
reconsult(fichier)
```

On peut ajouter des commentaires dans un fichier en les faisant précéder par le caractère `%` ; le commentaire s'étend jusqu'à la fin de la ligne.

1.4 Arithmétique

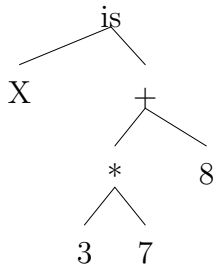
Les foncteurs permettent d'exprimer toutes les valeurs, mais dans ce cas l'arithmétique devient encombrante et inefficace car pour représenter par exemple `3`, il faudrait écrire `succ(succ(succ(0)))`. Heureusement, on peut effectuer des opérations en utilisant le foncteur `is` qui évalue son deuxième argument et l'unifie avec le premier. Ainsi

```
is(X, +(*(3,7),8))
```

unifiera `X` avec 29, car le deuxième argument est considéré comme un arbre dont les noeuds sont soit des valeurs soit des opérations à effectuer sur les branches. C'est déjà beaucoup plus pratique mais l'arithmétique sous forme préfixe n'est pas très habituelle (à moins d'être un lispien convaincu...). C'est pourquoi la plupart de ces foncteurs peuvent aussi s'écrire sous forme infixe (pour les foncteurs binaires) et sous forme préfixe ou suffixe pour les foncteurs unaires. L'utilisateur peut également déclarer certain de ses propres foncteurs comme étant infixe, préfixe ou suffixe. A chaque opérateur sont associées une priorité et une associativité, ce qui permet de rendre les programmes plus lisibles et reproduire ainsi les priorités rencontrées habituellement dans les langages de programmation. L'exemple précédent devient donc :

```
X is 3*7+8
```

qui n'est pas sans rappeler l'affectation classique, mais c'est une illusion car ceci n'est qu'une autre façon de dénoter l'arbre suivant :



Ce n'est que lorsqu'on tente de résoudre `is(X,+(*(3,7),8))` que l'unification de `X` avec l'évaluation du deuxième opérande est effectuée. Cette évaluation est également commandée par les opérations de comparaison : `>=`,`=<`,`<`,`>`,`==`,`\==` les deux derniers dénotant l'égalité ou l'inégalité mais forçant l'évaluation des arguments.

1.5 Les listes

Dans nos exemples précédents, notre syntagme était représenté par un foncteur à n arguments pour n mots, mais la longueur des suites de mots est inconnue a priori ; il est donc important de trouver une représentation pour exprimer des listes d'éléments. Ceci est fait à l'aide d'un foncteur spécial `.` à deux arguments dont le premier indique un élément et le deuxième la suite de la liste. Donc la liste de deux éléments `a` et `b` est représentée comme suit :

```
.(a,b)
```

cette notation est un peu encombrante et on utilise normalement l'abréviation infixe suivante où la liste est indiquée entre crochets et le début est séparé de la suite par une barre verticale.

```
[a | b]
```

Ceci est particulièrement pratique lorsqu'il y a plusieurs éléments dans la liste. Comparez :

```
.(a, .(.(c,d),e))     et   [a |[[c|d]|b]]
.( .( .(c,d), e),a)   et   [[[c|d]|b]|a]
```

Pour éviter des niveaux de crochets, on peut remplacer `.[` par une virgule (et enlever le crochet correspondant). Par exemple le premier exemple devient :

```
[a,[c|d]|b]
```

La fin de liste est habituellement indiquée par `[]` et on permet de ne pas indiquer le dernier élément vide (i.e.`|[]`) ; ceci permet de simplifier encore plus, comparez :

```
.(a, .(b, .(c,[])))   [a|[b|[c|[]]]]  [a,b,c]
```

qui représentent tous le même objet ; évidemment la troisième possibilité est la plus pratique à utiliser mais elle n'est qu'une abréviation pour les deux autres ; la première étant la représentation "logique". La notation avec une barre verticale est habituellement utilisée lorsqu'on veut indiquer que le "reste" d'une liste est une variable. Ainsi `[a,b | X]` équivaut à `.(a,.(b,X))` et représente une liste dont les deux premiers éléments sont `a` et `b` mais dont on ne connaît pas la suite (qui peut être `[]`). `[X|Y]` équivaut à `.(X,Y)` et donc `X` est le premier élément et `Y` le reste ; ceci correspond au `car` et `cdr` de Lisp. Voyons maintenant quelques prédicats utiles sur les listes. Nous n'en donnons que la définition et quelques exemples d'utilisation. On pourra consulter (Clocksin et Mellish 84) (Sterling et Shapiro 86) pour apprendre à dériver ces prédicats. Nous allons les utiliser ici régulièrement et les supposer prédéfinis.

```
%%%
%%%    outils de manipulations de listes
%%%      fichier "listes.pro"
%%%

%    concatenation
concat([],L,L).
concat([X|Xs],Y,[X|Z]):-concat(Xs,Y,Z).

%    verification d'appartenance
dans(X,[X|Xs]).
dans(X,[_|Xs]):-dans(X,Xs).

%    donner le nieme element
nieme([H|T],1,H):-!.
nieme([H|T],N,H1):-N1 is N-1, nieme(T,N1,H1).

%    trouver le nombre d'elements
longueur([],0).
```

```
longueur([H|T],N):-longueur(T,N1),N is N1+1.

%    renverser l'ordre des elements
renverse(L,R):-renverse(L,[],R).
renverse([H|T],L1,L):-renverse(T,[H|L1],L).
renverse([],L,L).
```

Voyons maintenant quelques exemples d'utilisation de ces prédicats :

```
| ?- concat([l,e],[c,h,a,t],X).
X = [l,e,c,h,a,t]

| ?- concat(Radical,[e,r],[a,i,m,e,r]).
Radical = [a,i,m]

| ?- concat(Debut,Fin,[j,e,u,n,e]).
Debut = []
Fin = [j,e,u,n,e] ;

Debut = [j]
Fin = [e,u,n,e] ;

Debut = [j,e]
Fin = [u,n,e] ;

Debut = [j,e,u]
Fin = [n,e] ;

Debut = [j,e,u,n]
Fin = [e] ;

Debut = [j,e,u,n,e]
Fin = [] ;

    no

| ?- dans(h,[c,h,a,t]).
    yes
| ?- dans(X,[c,h,a,t]).
X = c ;
X = h ;
X = a ;
X = t ;
    no

| ?- nieme([c,h,a,t],3,X).
```

```
X = a

| ?- longueur([c,h,a,t],L).
L = 4

| ?- renverse([e,t,a,l,e,r],X).
X = [r,e,l,a,t,e]
```

On remarque qu'on peut utiliser le même prédicat de plusieurs façons différentes selon que l'on instancie ou non certains paramètres : ainsi `concat` sert aussi bien à coller deux listes bout à bout qu'à en décomposer une en deux morceaux. `dans` permet d'obtenir tous les éléments d'une liste.

1.6 Prédicats extra-logiques

Si Prolog est basé sur la logique, on a quelquefois besoin d'un contrôle ou d'opérations qui sortent du cadre de la logique. Nous allons ici décrire très brièvement les outils extra-logiques dont nous aurons besoin pour le traitement de la langue naturelle.

1.6.1 Le coupe-choix

Si le non-déterminisme est souvent pratique, il y a des cas où il est intéressant de le limiter. L'outil le plus utile en Prolog est le "coupe-choix" *cut* notée par un point d'exclamation qui, lorsqu'il est "prouvé", a pour effet d'enlever tous les choix en attente dans le groupe de clauses où il apparaît. Il est utilisé pour éviter des tests dont on est certain qu'ils vont échouer ou pour s'assurer que certaines opérations ne vont être effectuées qu'une seule fois. Par exemple,

```
max(X,Y,X):- X>=Y.
max(X,Y,Y):- X<Y.
```

pourrait être réécrit comme

```
max(X,Y,X):- X>=Y,!.
max(X,Y,Y).
```

Ainsi, si le test de la première clause réussit, il est inutile de tenter la deuxième et s'il échoue, alors il est sûr que le deuxième réussit et nous n'avons pas besoin d'un autre test.

Le coupe-choix permet également de forcer l'échec d'un prédicat même s'il reste d'autres possibilités à explorer. Un cas particulier mais très utile de ce cas de figure est la définition de `not(P)` qui réussit lorsque `P` échoue et échoue lorsque `P` réussit. Il est défini comme suit :

```
not(P) :- call(P), !, fail.
not(P).
```

où `call` est tout simplement une demande de résolution du prédicat `P`. Le coupe-choix assure que la seconde clause ne sera pas tentée (pour réussir) après le succès de `P`. Si `P` échoue, alors le coupe-choix n'est pas essayé et on tente la deuxième possibilité qui renvoie un succès car il n'y a rien à prouver. En Sicstus, ce prédicat est prédéfini sous la forme de l'opérateur préfixe `\+`.

1.6.2 Décomposition de littéraux

On a quelquefois besoin de manipuler les littéraux eux-mêmes, en particulier lorsqu'on calcule un but à résoudre. Comme en C-Prolog il n'est pas possible d'écrire le but `X(Y)` où `X` aurait été instancié dans la clause où il apparaît, il faut d'abord composer le foncteur avant de l'appeler. Ceci peut être fait en utilisant l'opérateur `=..` (nommé `univ` depuis la première version de Prolog). Ainsi

```
f(a,b,c) =.. X
```

unifiera `X` à `[f,a,b,c]`. On obtient donc une liste dont le premier élément est le foncteur et les autres ses arguments. Donc pour construire `X(Y,Z)` et l'évaluer, on fera plutôt

```
P =.. [X,Y,Z], call(P).
```

1.6.3 Décomposition d'un identificateur

On peut "exploser" ou "imploser" le nom d'un identificateur avec `name` ; par exemple :

```
name(chat,X)
```

unifiera `X` à la liste `[99,104,97,116]` où chaque élément correspond au code ASCII du caractère correspondant de l'identificateur donné comme premier argument. Cette opération est réversible en ce sens que si on donne une liste de codes ASCII comme deuxième argument et une variable comme premier, alors on recompose un identificateur. Comme la manipulation de codes ASCII est assez malcommode et rend les programmes difficiles à lire, nous définirons à la section suivante un autre prédicat `nom` qui ne travaille qu'en terme d'identificateur (mais il utilise `name` de façon interne).

1.6.4 Modification de la base de faits

On peut ajouter et retrancher dynamiquement des clauses à un programme : ainsi

```
assert(c(p1,p2)).
```

ajoute la clause `c(p1,p2)` au programme et

```
retract(c(X1,X2))
```

enlève la première clause qui s'unifie avec `c(X1,X2)`. Ce mécanisme est utilisé pour transformer des programmes ou pour conserver des informations entre plusieurs exécutions de clauses car normalement toutes les liaisons de variables sont brisées lors de la fin de la preuve.

Nous donnons maintenant une illustration de l'arithmétique et de la modification de la base de faits en définissant un prédicat permettant de tirer des nombres au hasard ; ceci nous sera utile dans certains exemples des prochains chapitres.

```
%%     ce qu'il faut pour generer au hasard
%%      c'est le fichier "hasard.pro"

%   tirage au sort via un generateur congruentiel lineaire
%       donne X un nombre reel dans [0,1]
hasard(X):-
```

```
  retract(germe_hasard(X1)), X2 is (X1*824) mod 10657,
  assert(germe_hasard(X2)), X is X2/10657.0 .
%       donne X un nombre entier dans [1,N]
hasard(N,X) :-
  hasard(Y),
  X is floor(N*Y)+1.

%   initialisation du germe (a la consultation du fichier)
:- abolish(germe_hasard,1), X is floor(cputime*1000),
 assert(germe_hasard(X)).
```

1.7 Entrées-Sorties

Comme nous voulons traiter la langue naturelle et que ces applications impliquent souvent l'écriture d'interfaces, il est important d'avoir des outils commodes d'entrée et de sortie de mots et de phrases. Nous allons passer rapidement en revue les prédicats de lecture et d'écriture de C-Prolog et ensuite nous montrerons la programmation de prédicats de plus haut niveau.

1.7.1 Entrée de caractères

Voici les prédicats permettant l'entrée de caractères :

`get0(X)` unifie `X` avec le code ASCII du prochain caractère sur le terminal.

`get(X)` comme `get0` mais ignore tous les caractères de contrôle.

`skip(X)` où `X` doit être instancié à un code ASCII, saute tous les caractères en entrée en s'arrêtant au premier dont la valeur est égale à `X`.

1.7.2 Ecriture

Voici les prédicats permettant l'écriture de caractères :

`put(X)` écrit sur le terminal le caractère dont le code ASCII est `X`.

`write(X)` écrit le terme `X` selon la syntaxe standard y compris les opérateurs infixe

`nl` écrit une fin de ligne

`tab(I)` écrit `I` blancs sur la ligne courante. `I` peut être une expression à évaluer.

1.7.3 Entrée de mots et de listes de mots

Nous définissons maintenant des prédicats qui permettent de lire une suite de mots séparés par des blancs et terminée par un point, un point d'interrogation ou une fin de ligne et qui retourne une liste d'identificateurs. Ainsi

```
| ?- lire_les_mots(X).
|: bonjour les degats
X = [bonjour,les,degats]

| ?- lire_les_mots(X).
|: Salut,    les grands copains .
X = [Salut,les,grands,copains]
```

Ce programme n'est qu'un exemple de traitement de listes de mots et il est facile de le modifier pour traiter autrement certains caractères. On y remarque également la transformation des codes ASCII en identificateurs.

```
%%%
%%%   Les utilitaires de lecture
%%%     fichier "entree.pro"
%%%

%%  transformation de la ligne lue en une suite d'atomes
lire_les_mots(Ws):-lire_car(C),lire_les_mots(C,Ws).

lire_les_mots(C,[W|Ws]):-               % ajoute un mot
    car_mot(C),!,
    lire_mot(C,W,C1),lire_les_mots(C1,Ws).
lire_les_mots(C,[]):-                   % fin de phrase
    car_fin_des_mots(C),!.
lire_les_mots(C,Ws):-                   % on oublie ce
    lire_car(C1),lire_les_mots(C1,Ws).  % caractere

lire_mot(C,W,C1):-                      % construit un mot
    cars_des_mots(C,Cs,C1),nom(W,Cs).

cars_des_mots(C,[C|Cs],C0):-
    car_mot(C),!,lire_car(C1),cars_des_mots(C1,Cs,C0).
cars_des_mots(C,[],C):- \+(car_mot(C)).

%%     classes des caracteres
car_mot(C):- a @=< C, C @=< z.
car_mot(C):-'A' @=< C,C @=< 'Z'.
car_mot('''').

%%     une liste de mots est terminee par un point un
%%     un point d'interrogation ou une fin de ligne
car_fin_des_mots('.'):-skip(10).        % sauter jusqu'a la
car_fin_des_mots('?'):-skip(10).        % fin de ligne
car_fin_des_mots(X)  :-name(X,[10]).    % le "newline"
```

```
%%%   predicats necessaires en C-Prolog pour eviter
%%%   la manipulation de codes ascii

%%        lecture d'un caractere et
%%        transformation en l'atome correspondant
lire_car(X) :- get0(X1), name(X,[X1]).

%%      nom explose un identificateur en
%%      une liste d'identificateurs d'une lettre
%%      appel:   nom(aimer,[a,i,m,e,r])
%%

nom(X,Xl) :-                              % explose
    var(Xl),!, name(X,Codes), nom1(Codes,Xl).
nom(X,Xl) :-                              % compose
    var(X),!, nom1(Codes,Xl), name(X,Codes).

nom1([],[]).
nom1([X|Xs],[X1|X1s]):- name(X1,[X]), nom1(Xs,X1s).
```

1.7.4 Ecriture de listes de mots et d'arbres

Prolog nous fournit les outils de base pour l'écriture de termes, mais dans une interface nous aimerions souvent pouvoir écrire une liste d'identificateurs comme une liste de mots séparés par un blanc et terminée par une fin de ligne. Le prédicat `ecrire-les-mots` permet de faire cette opération. De plus, nous montrons un prédicat un peu plus complexe qui permet de faire ressortir la structure d'un terme composé en utilisant l'indentation des différentes lignes. Cette opération est communément appelé "pretty-print" d'où le nom de `pp` pour ce prédicat. Sa principale particularité étant qu'il est possible de contrôler les termes qui seront écrits sur plusieurs lignes. Voici des exemples d'utilisation :

```
| ?- ecrire_les_mots([bonjour,les,degats]).
bonjour les degats

| ?- pp(ph(snm(det(le),nom(chat)),svb(mange,snm(det(la),nom(souris))))).
ph(snm(det(le),
       nom(chat)),
   svb(mange,
       snm(det(la),
           nom(souris))))
```

Ces prédicats peuvent être définis comme suit :

```
%%%
%%%   Les utilitaires d'ecriture
%%%     fichier "sortie.pro"
%%%
```

```
%%  ecriture d'une liste de mots separes par un blanc
%%  et terminee par une fin de ligne
ecrire_les_mots([H|T]):-write(H),write(' '),ecrire_les_mots(T).
ecrire_les_mots([]) :- nl.

%%%
%%%    impression d'expressions Prolog avec controle
%%%    des foncteurs qui sont eclates sur plusieurs lignes
%%%         appel: pp(expression)
%%%           ou   pp(expression,decalage)
%%%
pp(X):-pp(X,0).                          % appel simplifie

pp(X,I):-var(X),!,write('_').            % une variable ?
pp(X,I):-
        X=.. [Y,Y1|Ys], boum(Y,Long),!, % predicat a eclater?
        write(Y),write('('),             % ecrire le foncteur
        pp(Y1,I+Long),                   % son premier argument
        pp1(Ys,I+Long),write(')').       % et les autres
pp(X,I):-write(X).                       % autre cas

%%%
%%%    impression d'une liste de parametres
%%%    sur des lignes differentes
%%%    en tenant compte du decalage
%%%
pp1([],I):-!.
pp1([X],I):-!,write(','),nl,tab(I),pp(X,I).
pp1([X|Xs],I):-!,write(','),nl,tab(I),pp(X,I),pp1(Xs,I).

%%%
%%%     les predicats a eclater avec le decalage
%%%     des lignes suivantes normalement c'est 1 de plus
%%%     que la longueur de l'identificateur
%%%
boum(ph,3).
boum(snm,4).
boum(svb,4).
boum(rel,4).
boum(int,4).
```

Nous terminons ici notre courte présentation de Prolog. Elle est loin d'être complète et on pourra consulter les ouvrages cités en bibliographie. Les outils définis dans ce chapitre serviront dans les prochains chapitres.

Chapitre 2

Analyse automatique en utilisant les grammaires logiques

2.1 Introduction

Les grammaires présentées au chapitre précédent avaient pour but de décrire la structure des phrases et certains phénomènes linguistiques. Il n'était pas nécessairement question de développer un moyen automatique de mettre en adéquation une phrase et une structure. Si nous voulons disposer d'outils informatiques de traitement de la langue naturelle, alors nous avons besoin d'algorithmes permettant de déterminer automatiquement si une phrase répond à une structure déterminée et, si oui, comment sont reliés les différents constituants entre eux.

Les premiers travaux dans ce domaine ont été effectués dans le cadre des grammaires formelles d'abord utilisées pour la définition de langages de programmation. Comme ce sont des langages artificiels, il est possible de les façonner sur mesure de façon à obtenir des analyseurs rapides et efficaces. Dans le cas de langues naturelles, il faut tenir compte de phénomènes complexes et donc les structures des grammaires ou des analyseurs deviennent plus compliquées.

Un formalisme qui a maintenant fait ses preuves est celui des grammaires logiques basées essentiellement sur le langage Prolog. Son grand mérite est sa déclarativité : comme nous l'avons vu au chapitre précédent, un programme Prolog est essentiellement une suite d'énoncés logiques indiquant des relations entre ceux-ci et l'exécution d'un programme se ramène à une preuve de la véracité d'un nouvel énoncé à partir de ceux qu'on a déjà.

Par exemple, soit la structure suivante : une phrase (p) est composée d'un syntagme nominal (sn) et d'un syntagme verbal (sv) ; un syntagme nominal (sn) est composé d'un déterminant (det) et d'un nom (n) ; un syntagme verbal *(sv)* est composé d'un verbe (v) ou d'un verbe et d'un syntagme nominal (sn). Ceci peut très bien être exprimé comme une suite d'énoncés logiques décrits à la figure 2.1.

Maintenant, il s'agit de prouver si la phrase :

le chat mange la souris

correspond bien à notre définition d'une phrase correcte. Pour continuer notre analogie avec la preuve de théorème, il s'agit de montrer que la première relation est vraie pour cette

$$\begin{aligned} p &= sn \wedge sv \\ sn &= det \wedge n \\ sv &= v \vee (v \wedge sn) \end{aligned}$$

FIG. 2.1: Grammaire en forme logique

phrase : elle le sera si nous trouvons un *sn* et un *sv* ; un *sv* est composé d'un déterminant *det* et d'un nom *nom* (ce qui correspond bien à *le* et à *chat*) ; il reste à prouver que la suite de la phrase *mange la souris* est un *sv* composé d'un verbe *v mange* et d'un *sn* ; la deuxième relation nous indique encore que le déterminant *la* et le nom *souris* forment un tel syntagme. Nous avons donc ainsi réussi à "prouver" que notre phrase a une structure répondant bien à notre définition. Evidemment, il faut remarquer que le *et* ($\wedge$) est assez spécial car en plus d'indiquer la conjonction, il indique la juxtaposition ; il n'est donc pas commutatif. On peut représenter ce traitement avec le schéma suivant :

```
| le chat  | mange     la souris  |
| <- sn -> | <------ sv --------> |
           | <- v -> | <-- sn --> |
```

Ce petit exemple informel illustre l'intuition derrière l'utilisation de la programmation logique (ou d'un démonstrateur de théorème) pour l'analyse de langages. Evidemment, il reste encore plusieurs problèmes à régler et nous allons regarder les outils fournis par Prolog pour exprimer une grammaire.

2.2 Codage de la grammaire en Prolog

Maintenant voyons comment coder la grammaire de la figure2.1 en Prolog, de façon à vérifier si une phrase donnée peut être analysée par cette grammaire. La première décision concerne évidemment la représentation de la phrase à analyser. Nous prenons la représentation la plus simple et la plus intuitive : une phrase est une suite de mots considérés comme atomiques. Ainsi la phrase

```
Le chat mange la souris.
```

sera représentée comme une liste d'atomes Prolog.

```
[le,chat,mange,la,souris]
```

Normalement c'est une interface qui construit cette liste par l'emploi d'outils semblables au prédicat `lire_les_mots` que nous avons défini au chapitre 2. Il reste maintenant à coder les règles : nous choisissons la représentation suivante : un symbole non-terminal sera représenté par un prédicat à deux arguments : le premier indique la chaîne d'entrée et le second, ce qu'il reste de la chaîne après son traitement par le prédicat. Cette représentation est communément appelée liste de différences. (Pereira Shieber 87) (Giannesini 85) montrent comment il possible de dériver "naturellement" cette représentation. Ici nous la prenons

tout simplement comme acquise et elle date d'ailleurs des premiers travaux de (Colmerauer 78) sur les grammaires de métamorphose. La définition d'un syntagme verbal peut donc être exprimée ainsi :

```
sv(L0,L) :- v(L0,L).
sv(L0,L) :- v(L0,L1), sn(L1,L).
```

Les paramètres Prolog nous permettent de connecter les différentes listes : dans le premier cas, où il n'y a qu'un verbe pour le syntagme, les listes du syntagme sont évidemment celles du verbe. Dans le deuxième cas, la liste d'entrée du verbe est celle du syntagme mais sa sortie est donnée en entrée au syntagme nominal pour qu'il fasse l'analyse de la suite et sa propre sortie sera celle du syntagme.

Dans le cas d'un symbole terminal, il suffit d'avoir une clause qui vérifie la présence du terminal au début de la chaîne et qui donne en sortie la suite de la chaîne. Cette vérification s'exprime simplement avec la clause unitaire suivante :

```
terminal(Mot, [Mot | L], L).
```

Son utilisation est également naturelle : par exemple, pour vérifier qu'un `nom` peut être `chat`, on écrit la clause :

```
n(L0,L) :- terminal(chat,L0,L).
```

un lecteur attentif ou un programmeur Prolog averti aura tout de suite remarqué qu'il est possible d'optimiser cette clause en intégrant immédiatement la clause `terminal`. On obtient alors par substitution de `Mot` et `L` :

```
n([chat | L], L)
```

mais, par souci d'uniformité et aussi parce que nous verrons que ce sera le système qui va gérer ces listes d'entrée et de sortie, nous garderons la première version.

La grammaire complète devient donc :

```
%%%
%%%    Grammaire en Prolog
%%%    (grammaire1.pro)
%%%
p(L0,L) :- sn(L0,L1), sv(L1,L).

sn(L0,L) :- det(L0,L1), n(L1,L).

sv(L0,L) :- v(L0,L).
sv(L0,L) :- v(L0,L1), sn(L1,L).

det(L0,L) :- terminal(le,L0,L).
det(L0,L) :- terminal(la,L0,L).

n(L0,L) :- terminal(souris,L0,L).
n(L0,L) :- terminal(chat,L0,L).

v(L0,L) :- terminal(mange,L0,L).
```

```
v(L0,L) :- terminal(trottine,L0,L).

terminal(Mot,[Mot|L],L).
```

que nous pouvons tester comme suit :

```
| ?- p([le,chat,mange,la,souris],[]).
        yes
| ?- p([le,chat,mange],[]).
        yes
| ?- p([la,chat,mange,le,souris],[]).
        yes
| ?- p([le,chat,trottine,la,souris],[]).
        yes
| ?- p([le|X],[]),write([le|X]),nl,fail.
[le,souris,mange]
[le,souris,trottine]
[le,souris,mange,le,souris]
[le,souris,mange,le,chat]
[le,souris,mange,la,souris]
[le,souris,mange,la,chat]
[le,souris,trottine,le,souris]
[le,souris,trottine,le,chat]
[le,souris,trottine,la,souris]
[le,souris,trottine,la,chat]
[le,chat,mange]
[le,chat,trottine]
[le,chat,mange,le,souris]
[le,chat,mange,le,chat]
[le,chat,mange,la,souris]
[le,chat,mange,la,chat]
[le,chat,trottine,le,souris]
[le,chat,trottine,le,chat]
[le,chat,trottine,la,souris]
[le,chat,trottine,la,chat]
        no
```

Comme le montrent les troisième et quatrième exemples, notre grammaire naïve accepte des phrases ayant une structure correcte, mais la langue naturelle impose d'autres contraintes qu'il faut vérifier : par exemple, l'accord en genre entre l'article et le nom et le fait que seul un verbe transitif peut accepter un complément d'objet direct. Ce "laxisme" est particulièrement évident dans le dernier exemple qui nous engendre toutes les phrases possibles commençant par `le` à partir de la grammaire ; il faut tout de suite remarquer que si cette exploitation "à l'envers" de la grammaire est un avantage indéniable de l'utilisation du non-déterminisme de Prolog, cette méthode n'est pas vraiment utilisable en pratique comme outil de génération de texte ; en effet, une grammaire réelle générera trop de possibilités et aura souvent une structure qui bouclerait à la génération, sans compter que la génération de texte est beaucoup

plus qu'une simple sortie de listes de mots (voir chapitre 6).

Voyons maintenant comment implanter ces vérifications : il faut tout d'abord coder l'information sur le genre des articles et des noms et sur la transitivité des verbes. L'approche la plus simple est de la coder dans la grammaire et de la répercuter via des paramètres qui devront s'unifier pour assurer la conformité des informations, notre grammaire devient donc :

```
%%%
%%%      Grammaire avec vérifications
%%%          (grammaire2.pro)
%%%
p(L0,L) :- sn(L0,L1), sv(L1,L).

sn(L0,L) :- det(Genre,L0,L1), n(Genre,L1,L).

sv(L0,L) :- v(_,L0,L).
sv(L0,L) :- v(transitif,L0,L1), sn(L1,L).

det(masculin,L0,L) :- terminal(le,L0,L).
det(feminin,L0,L) :- terminal(la,L0,L).

n(feminin,L0,L) :- terminal(souris,L0,L).
n(masculin,L0,L) :- terminal(chat,L0,L).

v(transitif,L0,L) :- terminal(mange,L0,L).
v(intransitif,L0,L) :- terminal(trottine,L0,L).

terminal(Mot,[Mot|L],L).
```

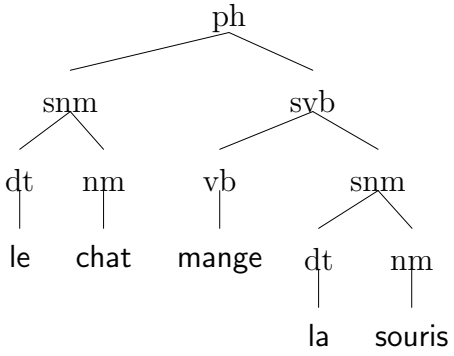
L'information de base est donnée comme premier paramètre aux prédicats `det`, `n` et `v`. La vérification de conformité de genre est faite dans la clause `sn` qui force le déterminant et le nom à avoir la même valeur pour `Genre`. La transitivité est vérifiée dans `sv` : dans la première clause, nous décidons ici d'accepter un verbe transitif ou intransitif comme verbe sans complément et nous donnons tout simplement une variable muette notée par un souligné (`_`) ; dans la deuxième clause de `sv`, nous exigeons toutefois la caractéristique `transitif`.

Testons maintenant cette grammaire sur les cas problèmes pour constater qu'ils sont maintenant rejetés :

```
| ?- p([la,chat,mange,le,souris],[]).
        no
| ?- p([le,chat,trottine,la,souris],[]).
        no
```

Jusqu'ici, nous nous sommes limités à vérifier si une phrase répondait à une structure, mais nous aimerions que notre processus d'analyse nous permet-te d'obtenir la structure de dépendance entre les constituants de notre phrase d'entrée. Voyons maintenant comment Prolog nous permet de construire ces structures, ici encore la procédure d'unification de Prolog nous donne un outil puissant, simple et naturel de construire de façon incrémentielle

la structure d'une phrase. Nous rappelons qu'une structure arborescente peut être représentée facilement en Prolog avec un foncteur indiquant la valeur du noeud et dont les paramètres corespondent aux branches issues de ce noeud. Ainsi l'arbre :



sera représenté par

```
ph(snm(dt(le),
       nm(chat)),
   svb(vb(mange),
       snm(dt(la),
           nm(souris))))
```

Cet arbre est construit via un paramètre véhiculant l'information structurelle. Un foncteur Prolog est très facile à construire car il suffit d'écrire sa forme et le processus d'unification le complète ou vérifie son adéquation. Ainsi la structure d'une phrase est donnée par le foncteur `ph` avec comme paramètre la structure du syntagme nominal et celle du syntagme verbal :

```
p(ph(SN_Struct,SV_Struct),L0,L) :-
    sn(SN_Struct,L0,L1), sv(SV_Struct,L1,L),
```

Nous faisons de même pour le `sn` et le `sv` ainsi que pour les terminaux. Notre grammaire n'est maintenant constituée que des prédicats d'analyse mais rien ne nous empêche d'utiliser d'autres prédicats Prolog pour vérifier des conditions supplémentaires (par exemple sémantiques) qui demandent d'autres opérations que la vérification de conformité disponible par l'unification. Par exemple, pour éviter l'acceptation de la phrase :

`La souris mange la souris`

il faudra vérifier qu'on ne retrouve pas le même syntagme nominal avant et après le verbe. Nous pouvons ajouter ce test dans `p` pour s'assurer qu'on ne retrouve pas la même structure de syntagme nominal comme sujet et comme complément du verbe. Pour refuser une phrase comme

`La souris mange le chat.`

Il faut vérifier que celui qui est mangé par un autre est bien une proie potentielle... On ajoutera donc de l'information "sémantique" à notre programme et on la vérifiera dans la clause `sv`. Notre nouvelle grammaire devient donc :

```
%%%
%%%      grammaire avec construction de structures
%%%          (grammaire3.pro)
%%%
```

```
p(ph(SN_Struct,SV_Struct),L0,L) :-
    sn(SN_Struct,L0,L1), sv(SV_Struct,L1,L),
    % s'assurer que les SN avant et apres le verbe sont differents
    \+(SV_Struct = svb(_,SN_Struct)).

sn(snm(Det_Struct,N_Struct),L0,L) :-
    det(Det_Struct,Genre,L0,L1), n(N_Struct,Genre,L1,L).

sv(svb(vb(Mot)),L0,L) :- v(vb(Mot,_),_,L0,L).
sv(svb(vb(Mot),SN_Struct),L0,L) :-
    v(vb(Mot,Comp),transitif,L0,L1), sn(SN_Struct,L1,L),
    % verifier que le complement est semantiquement acceptable
    SN_Struct = snm(_,nm(Nom)),         % extrait le nom
    T =.. [Comp,Nom],                   % construit le predicat
                                        % de verification semantique
    call(T).                            % on l'appelle

det(dt(le),masculin,L0,L) :- terminal(le,L0,L).
det(dt(la),feminin,L0,L) :- terminal(la,L0,L).

n(nm(souris),feminin,L0,L) :- terminal(souris,L0,L).
n(nm(chat),masculin,L0,L) :- terminal(chat,L0,L).

v(vb(mange,proie),transitif,L0,L) :- terminal(mange,L0,L).
v(vb(trottine,_),intransitif,L0,L) :- terminal(trottine,L0,L).

terminal(Mot,[Mot|L],L).

% la semantique
proie(souris).
```

Evidemment nous pourrions continuer ainsi à ajouter des informations de toutes sortes et, si nous regardons bien, nous constatons que l'expression de la grammaire est assez naturelle : nous ajoutons des paramètres pour véhiculer l'information entre les noeuds de l'arbre. Il reste toutefois à gérer le passage des listes d'entrée et de sortie entre les prédicats d'analyse d'une même clause. Ici notre grammaire est très simple et il est facile de le faire à la main ; or cet ajout est systématique et pourrait être facilement automatisé : c'est d'ailleurs ce qui a été fait pour les grammaires de métamorphose (Colmerauer 78) et plus récemment pour les grammaires à clauses définies (Definite Clause Grammars ou DCG) (Pereira et Warren 80) qui en sont un cas particulier. Voyons maintenant notre dernière grammaire écrite avec le formalisme DCG où les parties gauches sont séparées des parties droites par (`-->`). Dans ces clauses, c'est le système qui gère le passage de la chaîne entre les prédicats ; en fait, dans beaucoup de systèmes Prolog, ces clauses sont transformées à la lecture en des clauses équivalentes ayant une structure semblable à celle que nous avons développée au cours des exemples précédents, c'est-à-dire avec deux paramètres supplémentaires à tous les prédicats

d'analyse ; mais ils ne doivent pas être ajoutés aux autres prédicats que nous devons marquer spécialement. En DCG, ces prédicats sont simplement entourés d'accolades. Un terminal est indiqué entre crochets et la clause `terminal` n'est plus nécessaire dans notre programme car elle est maintenant prédéfinie par le système (elle est traditionnellement appelée `'C'` pour *Connects*).

```
%%%
%%%       Version DCG de la grammaire avec construction
%%%       de structures
%%%          (grammaire3dcg.pro)
%%%
p(ph(SN_Struct,SV_Struct)) -->
    sn(SN_Struct), sv(SV_Struct),
    % s'assurer que les SN avant et après le verbe sont différents
    {not(SV_Struct = svb(_,SN_Struct))}.

sn(snm(Det_Struct,N_Struct)) -->
    det(Det_Struct,Genre), n(N_Struct,Genre).

sv(svb(vb(Mot))) --> v(vb(Mot,_),_).
sv(svb(vb(Mot),SN_Struct)) -->
    v(vb(Mot,Comp),transitif), sn(SN_Struct),
    % verifier que le complément est sémantiquement acceptable
    {SN_Struct = snm(_,nm(Nom)), (T =.. [Comp,Nom]), call(T)}.

det(dt(le),masculin) --> [le].
det(dt(la),feminin) --> [la].

n(nm(souris),feminin) --> [souris].
n(nm(chat),masculin) --> [chat].

v(vb(mange,proie),transitif) --> [mange].
v(vb(trottine,_),intransitif) --> [trottine].

% la semantique
proie(souris).
```

Une grammaire DCG est donc présentée sous la forme de règles de grammaire exprimée par des prédicats avec des paramètres permettant de véhiculer de l'information entre eux. De plus, il est possible d'ajouter des prédicats Prolog quelconques à condition de les entourer d'accolades.

Le formalisme DCG est a priori indépendant de Prolog et nous pourrions en écrire des interprètes ou des compilateurs dans tout langage ou système qui permettrait d'unifier des paramètres. Evidemment comme ce sont des opérations fondamentales de tout interprète Prolog, les DCG sont assez simples à implanter dans ce langage car il suffit d'un simple préprocesseur (lui-même écrit en Prolog).

En comparant les deux derniers programmes, on constate que le fait de cacher les manipulations de listes permet de faire ressortir les traitements et les paramètres importants ; les accolades regroupent les traitements autres que l'analyse syntaxique. C'est pourquoi à partir de maintenant nous n'utiliserons que le formalisme DCG ; mais si votre Prolog favori ne dispose pas d'un formalisme équivalent, il vous suffira d'ajouter à la main les listes d'entrées et de sorties[1] pour appliquer ce que nous montrons ici.

2.3 Une autre application des DCG

Nous allons nous servir des DCG pour écrire une version de Eliza, un programme écrit originalement par J. Weizenbaum au milieu des années soixante pour illustrer un système de traitement de liste. Comme ce programme avait pour cadre la conversation entre un psychanalyste et son patient, il a excité l'intérêt du public non averti. En y regardant de plus près, on constate que ce programme n'est constitué que d'une suite de modèles cherchés dans une chaîne d'entrée qui sont répétés avec quelques modifications mineures. Ceci peut très bien être modélisé en terme de grammaires et c'est pourquoi nous allons utiliser les DCG pour les exprimer. Ici au lieu de construire l'arbre de dérivation lors de l'analyse d'une chaîne, nous allons tout simplement donner la "réponse" du psychanalyste...

Cette grammaire est d'ailleurs très simpliste ici car elle se borne à une suite de terminaux à trouver dans l'entrée. La seule complication consiste à permettre la recherche de ces terminaux à partir de n'importe quelle position dans la chaîne d'entrée. Pour cela, il suffit de pouvoir exprimer une chaîne de longueur "variable" dans l'entrée. Ceci peut être facilement exprimé en utilisant le non-déterminisme implicite de Prolog avec le prédicat `concat` défini au chapitre précédent. Pour bien montrer que ce qui nous intéresse est plutôt le fait de sauter des parties de chaîne nous nommons ce prédicat "..." ; c'est un prédicat à trois arguments, mais comme les deux derniers sont fournis par les DCG, c'est donc un prédicat à un seul argument ; nous pouvons donc nous passer des parenthèses autour de son paramètre en le déclarant comme opérateur préfixe. Ainsi `..._` désigne une liste éventuellement vide de mots qui sont ignorés et `...X` unifie `X` avec la liste de mots qui sont sautés. Afin d'introduire un peu de variation, nous utilisons le prédicat, `ok(P)` qui ne réussit qu'avec une probabilité de `P` en utilisant le prédicat `hasard` défini dans un fichier que nous décrivons pas ici ; `hasard(X)` retourne un nombre au hasard différent dans l'intervalle $[0, 1)$ à chaque appel.

```
%%%         programme Eliza de Weizenbaum qui simule une
%%%           conversation avec un thérapeute.
%%%     Le programme original a été fourni par Michel Boyer.
%%%         Les formules françaises sont inspirées de
%%%             Wertz, Lisp, Masson, 1985, p180
%%%                 (eliza.pro)

:- reconsult('entree.pro'), reconsult('sortie.pro'),
   reconsult('hasard.pro'), reconsult('listes.pro').
```

[1] ou consulter (Sterling et Shapiro 86), (Pereira et Sheiber 87) ou (Saint-Dizier 87) (en français), pour écrire un interprète ou un traducteur qui le fera automatiquement

```
%  autre nom pour "concat" pour indiquer que tout ce qui
%  nous interesse ici c'est de sauter un bout de phrase
%  attention le resultat est dans le deuxieme parametre !!!
:- op(900,fy,'...').
 ...(X,Y,Z) :- concat(X,Z,Y).

eliza :-
   write(' Oui, tu peux tout me dire¡), nl,
   write('|: '),
   repeat,
   lire_les_mots(Entree),
   eliza(Entree).
eliza([bonsoir]) :-
   write(merci), nl, !.
eliza(Entree) :-
   reponse(Reponse,Entree,[]), !,
   ecrire_les_mots(Reponse), fail.

%% Generation de la reponse a une entree du patient.
%%  reponse(LaReponse) --> le modele a verifier dans l'entree
reponse([pourquoi,n,etes,vous,pas | X]) -->
   [je,ne,suis,pas], ...X.
reponse([alors,vous,savez,programmer]) -->
    ..._, [prolog], ..._.
reponse([alors,vous,ne,savez,pas,programmer]) -->
    ..._, [lisp], ..._.
reponse([dites,m,en,plus,sur,votre,X]) -->
    ..._, [X], ..._, {important(X)}.
reponse([hmmm]) -->
    [_,_,_].
reponse([vous,etes,bien,negatif]) -->
    [non].
reponse([c,est,un,peu,court]) -->
    [_], {ok(0.33)}.
reponse([vous,n,etes,pas,bavard]) -->
    [_], {ok(0.5)}.
reponse([vous,m,en,direz,tant]) -->
    [_].
%  on n'a pas trouve les modeles qu'on cherchait ...
reponse([je,ne,vous,suis,pas,tres,bien]) -->
    ..._,{ok(0.33)}.
reponse([ca,alors]) -->
    ..._,{ok(0.5)}.
```

```
reponse([n,importe,quoi]) -->
    ...._.

%  ne reussit qu'avec probabilite P
ok(P):- hasard(X),!,X<P.

%%  la semantique ...
important(mere).
important(pere).
```

et un exemple d'interaction avec ce programme

```
| ?- eliza.
 Oui, tu peux tout me dire!
|: j'aime bien faire du prolog
alors vous savez programmer
|: oui
vous n etes pas bavard
|: non
vous etes bien negatif
|: pas autant que mon pere
dites m en plus sur votre pere
|: je ne suis pas tout a fait votre conversation!
pourquoi n etes vous pas tout a fait votre conversation
|: parce que ce serait trop long
n importe quoi
|: vous m insultez
hmmm
|: c'est tout ce que ca vous fait
n importe quoi
|: eh oui
ca alors
|: bonsoir.
merci
```

L'adaptation française de ce programme est moins spectaculaire que la version originale anglaise car il faut trouver des formules invariantes sous le genre et le nombre tout en tenant compte des variations morphologiques. Ce problème étant beaucoup moins aigu en anglais, il était plus facile de tromper l'interlocuteur avec un programme plus simple ne faisant somme toute que des remplacements ou des déplacements de mots rencontrés précédemment dans les questions. On pourrait arriver au même résultat en français en utilisant une programmation plus complexe pour tenir compte de la morphologie mais ceci n'ajouterait rien à la compréhension.

Nous vous invitons à comparer cette version avec ce qu'il aurait fallu écrire dans d'autres langages notamment Lisp (utilisé habituellement pour programmer cet exemple) ; il faut alors écrire un algorithme de vérification de modèle assez complexe alors qu'ici on utilise tout simplement l'unificateur de Prolog.