# Initiation à la Programmation en Logique
# avec
# SISCtus Prolog



**Identificateurs**

Ils sont représentés par une suite de caractères alphanumériques commençant par une lettre minuscule (les lettres accentuées sont à éviter).

*Exemples* : f, aLPHA, jean-paul, h1.

**Variables**

Elles sont représentées par une suite de caractères alphanumériques commençant par une lettre majuscule (les lettres accentuées sont à éviter).

*Exemples* : X, X1, MACHIN, Truc.

**Fonctions et prédicats**

Ils sont représentés à l'aide d'un identificateur. Les arguments sont écrits après, entre parenthèses, séparés par des virgules (notation préfixée et parenthésée).

**Constantes**

Elles sont représentées avec un nombre ou un identificateur.

**Clauses**

Une clause Prolog est définie par une tête et éventuellement par un corps (une clause Prolog ne possède pas forcément un corps).

La tête de clause est formée par un seul prédicat, tandis que le corps est constitué de un ou plusieurs prédicats qui sont séparés par des virgules.

La tête est séparée du corps par les signes ' :-'. Une clause Prolog se termine toujours par un point.

Soit une formule F définie sous forme normale disjonctive (DNF) par :

$$F = (A_1^1 \wedge A_2^1 \wedge ... \wedge A_{n_1}^1) \vee (A_1^2 \wedge A_2^2 \wedge ... \wedge A_{n_2}^2) \vee ... \vee (A_1^m \wedge A_2^m \wedge ... \wedge A_{n_m}^m)$$

Cette formule F sera représentée en Prolog par le prédicat `f` tel que :

f :- $\mathtt{A}_1^1, A_2^1, ..., A_{n_1}^1$.
f :- $\mathtt{A}_1^2, A_2^2, ..., A_{n_2}^2$.
f :- $\mathtt{A}_1^m, A_2^m, ..., A_{n_m}^m$.

Ainsi, les conjonction sont remplacées dans les clauses Prolog par des virgules et chaque ligne débutant par une même tête de clause est séparée par une disjonction. Ainsi, dans le cadre d'une unification, les lignes sont testées l'une après l'autre.

### Premier exemple de programme

```
% Exemple 1 : cette ligne est un commentaire
habite(christophe,vesoul).
habite(fabrice,marseille).
habite(fabien,belfort).
habite(laurent,vesoul).
meme-endroit(X,Y) :- habite(X,V),habite(Y,V).
```

Pour faire fonctionner ce petit exemple, il faut dans un premier temps le saisir dans un éditeur de texte (WordPad par exemple) et l'enregistrer au format texte (les fichiers Prolog ont habituellement l'extension .pl).

Dans un deuxième temps, il faut lancer SICStus Prolog. Les symboles '| ?-' indique que le programme est en attente d'une question.

Avant de poser des questions, il est nécessaire de mettre dans la mémoire de SICStus les clauses précédentes. Cette opération est effectuée au moyen de la commande `reconsult`[1] de la manière suivante :

```
| ?- reconsult('Z :\\mon_repertoire\\...\\exemple.pl').
```

Les éventuelles erreurs détectées dans le programme source exemple.pl sont alors présentées. Elles doivent être corrigées avant de pouvoir poser des questions.

Si la consultation n'a décelé aucune erreur, on peut dès lors questionner (c'est à dire proposer une clause négative).

[1] Voir les prédicats `listing`, `reconsult`, `assert`, `retract` et `abolish` page 7 et 8

```
   % Exemple 2 : Exemples de question
habite(fabrice,marseille).
habite(fabien,X).
meme-endroit(X,laurent).
```

Ne jamais oublier le point qui doit terminer toute clause, mais aussi toute question.

Si la question ne contient aucune variable, Prolog répond `yes` ou `no` suivant que la clause vide a été aperçue ou non, lors du parcours de l'arbre de résolution. Si une clause vide est rencontrée, Prolog n'en cherchera pas d'autre.

Si la question contient une variable ou plus, Prolog fournit le premier résultat. En frappant un point virgule, on obtiendra le second résultat, et ainsi de suite. En frappant, un retour charriot, on arrête le parcours de l'arbre. La réponse `no` précise qu'il n'y a pas ou plus de solution, tandis que `yes` signifie que l'arbre n'a pas été parcouru entièrement (dans le cas d'un retour charriot).

### Listes

Les listes sont décrites à l'intérieur de crochets. Pour former des listes, on utilise en plus des crochets l'un ou l'autre symbole : `,` ou `|`. Mais, attention, leur signification n'est pas du tout la même.

La virgule est un séparateur permettant d'énumérer les éléments d'une liste. Ainsi, `[a,b,c]` est la liste formée par a, b et c.

`[a|L]` est la liste obtenue en ajoutant l'élément `a` au début de la liste `L`. Ainsi, le symbole `|` doit être considéré comme une fonction de degré 2 (en mathématiques souvent notée &). Ainsi, ce symbole permet de récupérer le premier élément d'une liste ou, d'ajouter un élément en début de liste.

Les notations suivantes désignent le même objet : `[a,b]`, `[a|[b]]`, `[a|[b|[]]]`.

Les deux exemples suivants, illustrant la construction et la manipulation des listes, doivent être absolument compris avant de s'engager dans une programmation plus avancée.

```
   % Exemple 3 : appartient(X,L) ⇔ 'X appartient à L'
appartient(X,[X|Y]).
appartient(X,[Z|Y]) :- appartient(X,Y).
```

```
  % Exemple 4 : concatene(L1,L2,R) ⇔ 'R est la concaténée de L1 et L2'
concatene([],L2,L2).
concatene([X|L1],L2,[X|R]) :- concatene(L1,L2,R).
```

Simuler pas à pas le traitement des clauses : `'appartient(X,[a,b,c]).'` et `'concatene([a,b],[c,d],L).'` et comprendre à quoi correspond leur résultat.

### Nombres

Les nombres sont des constantes. Ils peuvent faire l'objet d'opérations notées de façon infixée avec +, -, *, /,//(division entière), mod. Ces opérations sont utilisées avec le prédicat `is`. Les conditions dans lesquelles on peut utiliser ce prédicat sont assez strictes.

Ainsi, on doit écrire à gauche de `is` un nombre ou une variable. A sa droite, on doit trouver une expression numérique. Cette expression peut contenir des variables à condition que ces dernières aient fait l'objet d'une unification avec un nombre avant le traitement de la clause contenant `is` (une sorte d'initialisation des variables du membre droit). Ainsi, la question `X is Y+3.` n'est pas acceptée mais la question `Y is 0, X is Y+3.` est valable.

Le traitement de `is` consiste en une unification. SICStus va effectivement tenter d'unifier le membre de gauche avec le résultat du membre droit. Si l'unification échoue, la valeur `no` est renvoyée. Si elle aboutit, l'unification est effectuée (le membre gauche prend la valeur du résultat du membre droit) et la valeur `yes` est renvoyée.

Il est possible de comparer des nombres au moyen de `<,=<,...` La notation est infixée et, les expression à droite et à gauche du prédicat de comparaison peuvent également contenir des variables si celles-ci sont instanciées par unification à un nombre avant le traitement de la clause de comparaison.

### Fonctions min et max

Il est possible, dans une formule mathématique, d'extraire directement la plus grande valeur et la plus pletite valeur de deux nombres.

Ainsi la valeur de `min(X,Y)` est la plus petite valeur de X et de Y, tandis que `max(X,Y)` correspond à la plus grande.

```
  % Exemple 5 : utilisation de min, max et is
add_min_max :- X is min(4,5)+max(5,1).
```

### Fonction dif

La fonction `dif` permet de comparer deux constantes. Elle prend comme valeur `true` si les deux constantes sont différentes, `false` dans le cas contraire.

Notons que les arguments de cette fonction (ou un des deux) peuvent être des variables. Celles-ci doivent alors être instanciées par unification avant le traitement du `dif`.

```
   % Exemple 6 : utilisation de dif
| ?- A=1, dif(A,2).
```

### Fonction ==

La fonction == permet également une comparaison mais cette dernière n'est pas restreinte aux constantes. Ainsi, dans le cas d'une comparaison entre variables, la syntaxe de ces dernières constitue l'élément de comparaison (et non plus la valeur comme dans le cas d'une constante).

Ainsi, cette fonction prend comme valeur `true` si les deux termes sont identiques, `false` dans le cas contraire.

```
   % Exemple 7 : utilisation de ==
| ?- A==B.
```

### Lecture et écriture à l'écran

Il s'agit de prédicats de poids 1 : `read` pour la lecture et `write` pour l'écriture. Ces prédicats s'exécutent lorsque la résolution s'effectue.

```
   % Exemple 8 : utilisation de read/1 et write/1
addition :- write('Entrez un nombre :'), read(A), write('Entrez un autre
nombre :'), read(B), Z is A+B, nl, write(A), write('+'), write(B), write('='),
write(Z).
```

Pour faire fonctionner cet exemple, questionner avec `addition.`, sans oublier le point à la fin de chaque entrée. Le prédicat de poids 0, `nl`, permet de passer à une nouvelle ligne.

### Lecture et écriture dans un fichier

Il s'agit de prédicats de poids 2 : `read` pour la lecture et `write` pour l'écriture. Ces prédicats s'exécutent également lorsque la résolution s'effectue.

La différence avec les deux prédicats de lecture et écriture à l'écran déjà cités consiste en l'ouverture d'un fichier cible. L'ouverture du fichier est commandée par la fonction `open`, qui prend en argument le nom du fichier, le mode d'ouverture et un descripteur de fichier (variable) qui vient s'ajouter à l'argument des deux prédicats de lecture et écriture à l'écran.

Le mode d'ouverture peut être `read` pour ouvrir le fichier en lecture, `write` pour ouvrir le fichier en écriture en l'écrasant s'il existe déjà, ou `append` pour ouvrir le fichier en écriture sans écraser les données qu'il comporte déjà éventuellement (les nouvelles données étant écrites à la suite des anciennes).

Au terme d'une écriture ou d'une lecture, le fichier cible doit être fermé au moyen de la clause `close` qui prend en argument le descripteur de fichier.

```
   % Exemple 9 : utilisation de read/2, write/2, close et open
addition :- open('Fichier.txt',write,Stream), write(Stream,'3'), write(Stream,'+'),
write(Stream,'4'), write(Stream,'='), write(Stream,'7'),close(Stream).
```

### Cut ( !)

Ce mécanisme de contrôle du parcours de l'arbre est très important. Il est noté `' !'`. Il peut être placé dans toute clause, comme un prédicat prédéfini de poids 0. Lors du parcours de l'arbre, la clause le contenant ne rendra plus qu'un seul résultat après avoir franchi le cut. Pour cette raison, on nomme ce prédicat un coupe-choix.

Revenons à l'exemple 3 qui définissait `appartient`, et modifions-le en faisant intervenir un cut :

```
   % Exemple 10 : appartient(X,L) ⇔ 'X appartient à L'
appartient1(X,[X|Y]) :- !.
appartient1(X,[Z|Y]) :- appartient1(X,Y).
```

Comparer le résultat de `appartient(X,[a,b,c]).` et de `appartient1(X,[a,b,c]).`, et suivre le déroulement pas à pas de `appartient1` afin de comprendre le mécanisme du cut.

### Not (\+)

Le 'not' permet de considérer la négation d'une fonction ou d'un prédicat. Ainsi, soit la clause `complement(X,L,M)` dont le sens sera 'X appartient à la liste L et n'appartient pas à la liste M' :

```
   % Exemple 11 : utilisation de \+
complement(X,L,M) :- appartient(X,L), \+appartient(X,M).
```

### Prédicats Prolog

Prolog caractérise un prédicat par son nom et son poids. Ainsi, si on introduit la clause `habite(alain,paris,16).`, Prolog considérera qu'il a à faire à un autre prédicat que celui introduit précédemment (cf. `read` et `write`).

Il en est de même avec les fonctions. On pourra considérer le terme `a(a,a(b))` dans lequel la lettre `a` identifie à la fois une fonction de degré 2, une fonction de degré 1, et une constante. Ces trois objets seront considérés comme différents, bien qu'ils portent le même nom.

### Listing

SICStus dispose d'une mémoire dans laquelle il stocke les différentes clauses sous forme de base de données. Ce prédicat permet de voir les clauses de cette base de données.

Ainsi, la question `listing.` permet de visualiser l'ensemble des clauses de la base de données, tandis que `listing(habite)` permet de visualiser les clauses de la base de données commençant par le prédicat `habite`.

### Reconsult

Le prédicat `reconsult` ajoute à cette base de données les clauses situées dans un fichier dont le chemin et le nom constituent l'argument de ce prédicat.

Une clause de la base de donnée n'est modifiée par cette opération que si le fichier consulté contient une clause ayant même tête. Ceci permet de travailler avec une base de données dont les clauses proviennent de plusieurs fichiers sources.

Mais il s'en suit qu'il est plus facile d'ajouter des clauses que de les enlever. Pour exemple, reconsultons le fichier dans lequel ont été saisies les clauses de l'exemple 1. La réponse à la question `habite(X,marseille).` est bien évidemment `fabrice`.

Effaçons maintenant du fichier source la clause `habite(fabrice,marseille).` et reconsultons. La réponse à la question `habite(X,marseille).` est toujours la même...

### Assert et retract

Le prédicat `assert` permet d'ajouter une clause à la base de données depuis le programme Prolog, tandis le prédicat `retract` permet d'en faire disparaître.

Ainsi la question `assert(habite(emilie,lyon)).` a pour effet d'ajouter la clause `habite(emilie,lyon).` aux clauses déjà présentes dans la base de données. On peut le vérifier en demandant `habite(X,lyon).` par exemple.

La question `retract(habite(X,lyon)).` fera disparaître toutes les clauses de la forme `habite(X,lyon)` en unifiant éventuellement la variable `X`.

### Abolish

Ce prédicat permet d'effacer des clauses de la base de données, depuis le programme Prolog, en ne citant que le prédicat de leur tête.

Ainsi la question `abolish(habite/2).` a pour effet de supprimer de la base toutes les clauses dont le prédicat de tête se nomme `habite` et a pour poids 2.

### Exemples

On veut construire un prédicat `pair/2` qui permet de renvoyer la liste des nombres pairs d'une liste L de nombres donnée en argument.

Rappelons qu'un nombre est pair si le reste de sa division par 2 est égal à 0. On va ainsi appliquer modulo 2 sur chacun des termes de la liste `L` pour tester s'il est pair. Si c'est le cas, il doit être présent dans la liste résultat, sinon il ne doit pas y figurer.

Deux procédés peuvent être envisagés pour implanter ce prédicat. Le premier `pair1` utilise un accumulateur, c'est à dire que le résultat est construit lors de l'empilement des appels de prédicats. Le second prédicat `pair2` n'utilise pas l'accumulateur et le résultat est construit lors du dépilement des appels de prédicats.

```
   % Exemple 12 : Exemple avec accumulateur
pair1(L,L_Nombres_Pairs) :- pair1(L,[],L_Nombres_Pairs).
pair1([],Acc,Acc) :-!.
pair1([E|L],Acc,L_Nombres_Pairs) :- 0 is E mod 2,!,pair1(L,[E|Acc],L_Nombres_Pairs).
pair1([_|L],Acc,L_Nombres_Pairs) :- pair1(L,Acc,L_Nombres_Pairs).


   % Exemple 13 : Exemple sans accumulateur
pair2([],[]) :-!.
pair2([E|L],[E|L_Nombres_Pairs]) :- 0 is E mod 2,!,pair2(L,L_Nombres_Pairs).
pair2([_|L],L_Nombres_Pairs) :- pair2(L,L_Nombres_Pairs).
```