Prolog in Smalltalk (VisualWorks)

Table of contents

1 Origins

Adapted/Tested on VisualWorks 7.10 by Martial Tarizzo.
Initial code for this version : GNU Smalltalk, modified par Paolo Bonzini from original code:

```
------------------------------------------------------------------
Goodies for VisualWorks 2.5 and VisualWave 1.0
Copyright (c) 1995-1998 AOKI Atsushi
1998/01/05
------------------------------------------------------------------
 AOKI Atsushi http://www.sra.co.jp/people/aoki
 Software Research Associates, Inc. mailto: aoki@sra.co.jp
 Marusho-Bldg.5F, 3-12 Yotsuya, Tel: 03-3357-9361
 Shinjuku-ku, Tokyo 160-0004, JAPAN Fax: 03-3351-0880
------------------------------------------------------------------
```

This document was first translated from japanese doc with the help of (fairly strange) Google translation... and adapted to the current version.

2 Overview

"Prolog In Smalltalk" (PIS) is a processing system (interpreter) of Prolog conforming to Dec-10 Prolog notation.
Everything in PIS is prefix notation, prefix / infix / postfix with op predicates cannot be defined.
Enjoy the atmosphere of Prolog programming: unification and backtrack.

2.1 Installation method

The Prolog bundle contains two packages:

- PrologCore
- PrologShell

2.1.1 PrologCore

PrologCore is the main package, containing 15 classes. The main class is the interpreter (`PrologInterpreter`) with two public methods (`refute:` and `refute:action:`)
Nine instance methods of Object are added to the category named prolog.
The set set is as follows:

Added classes:	Methods added to Object (prolog category)
• `PrologBody`	• `car`
• `PrologClause`	• `cdr`
• `PrologDefinition`	• `cons:`
• `PrologEntity`	• `consp`
• `PrologInterpreter`	• `isPrologEntity`
• `PrologList`	• `isPrologVariable`
• `PrologObject`	• `printPrologOn:`
• `PrologParser`	• `printPrologOn:level:`
• `PrologScanner`	• `printPrologString`
• `PrologScannerTable`	
• `PrologString`	
• `PrologStructure`	
• `PrologSymbol`	
• `PrologTerms`	
• `PrologVariable`	

2.1.2 PrologShell

A very simple GUI is in the package `PrologShell`. A subclass of `PrologInterpreter` (`PrologShellInterpreter`) is used as an example of interaction between Prolog and VisualWorks. To open the GUI, evaluate:

```
PrologShell open
```

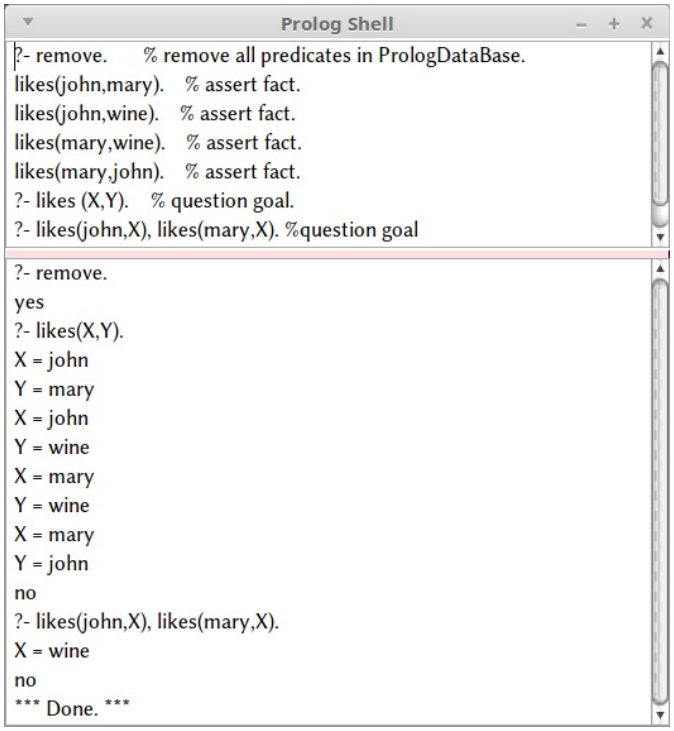
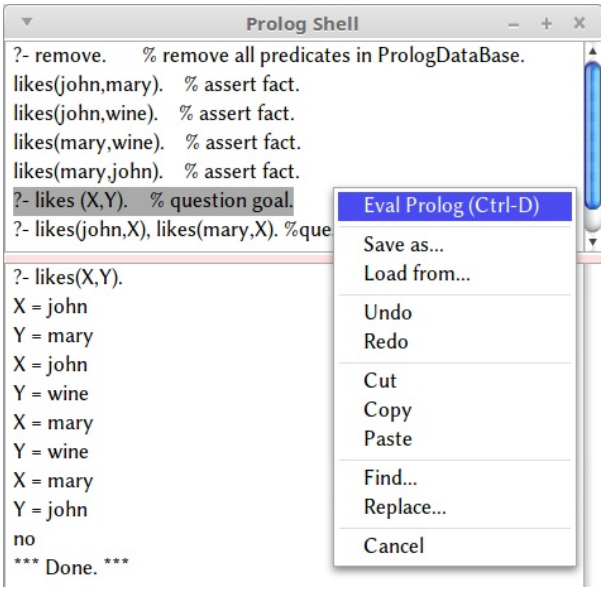


Figure 1. The PIS shell

The upper window is for input from the user, the lower window is Prolog output.
To evaluate the buffer (or only a selection in the buffer) with the interpreter, press **Ctrl-D**, or right-click in the editor to display the command menu and select **Eval Prolog**.
One can evaluate a section of lines between lines beginning by 2 or more characters '%' with 'Ctrl-Enter'.
All the solutions are then displayed in the output window.



One can save/load prolog text files with the corresponding menu commands.

Examples of interaction

1. Please enter the following in the upper window and evaluate it.
   ```
   ?- append(X, Y, [1, 2, 3, 4]).
   ```
   You should get in the output window:
   ```
   ?- append(X,Y,[1,2,3,4]).
   X = []
   Y = [1,2,3,4]
   X = [1]
   Y = [2,3,4]
   X = [1,2]
   Y = [3,4]
   X = [1,2,3]
   Y = [4]
   X = [1,2,3,4]
   Y = []
   no
   *** Done. ***
   ```

2. To query a small database:
   ```
   ?- remove.       % remove all User predicates in PrologDataBase.
   likes(john,mary).    % assert fact.
   likes(john,wine).    % assert fact.
   likes(mary,wine).    % assert fact.
   likes(mary,john).    % assert fact.
   ?- likes(john,X), likes(mary,X). %question goal
   ```
   Answer:
   ```
   ?- remove.
   yes
   ?- likes(john,X), likes(mary,X).
   X = wine
   no
   *** Done. ***
   ```

3. What are the predicates available ? With the previous database loaded in a fresh shell
   ```
   ?- predicates ([User | System]).
   ```
   gives:
   ```
   ?- predicates([User|System]).
   System = [-,!,*,/,//,\\,\=,\==,+,<,=,'=..',=<,==,>,>=,and,append,arg,
   assert,asserta,assertz,atom,atomic,call,clause,clear,clock,consult,
   dotp,double,fail,float,fraction,functor,gc,inspect,integer,is,length,
   lispAppend,lispAssoc,lispMember,lispNconc,lispReverse,list,listing,
   member,name,nl,nonvar,nospy,not,notrace,nth,number,or,predicates,
   printlist,read,reconsult,remove,repeat,retract,reverse,saving,send,spy,
   string,structure,symbol,systemListing,systemPredicates,tab,trace,true,
   userPredicates,var,verbose,write]
   User = [likes]
   no
   *** Done. ***
   ```

2.2 Using a Prolog interpreter without PrologShell

2.2.1 From Smalltalk to Prolog

From Smalltalk without passing through the user interface, you can also send `refute: aString` to an instance of `PrologInterpreter`.

Derivation will be attempted and the first unifier that satisfies the goal will be returned (or false if it fails). A unifier is a Smalltalk dictionary (variables are the keys, associated with the prolog values which satisfy the goal).

To get all the solutions, you should send `refute: aString action: aBlock`, with a one argument block `[:unifier | . . .]` as second argument.
If the block returns `false`, the interpreter will backtrack, providing the next solution (if it exists).

Examples

1. In a workspace, evaluate the following Smalltalk expressions:
```
prolog := PrologInterpreter new.
(prolog refute: '?- append (X, Y, [1, 2, 3, 4]).')
        keysAndValuesDo: [:varname :value |
            Transcript show: varname, '=', value printPrologString; cr].
```
   You should see in the Transcript
```
?- append(X,Y,[1,2,3,4]).
Y=[1,2,3,4]
X=[]
```
2. To get all the solutions:
```
prolog := PrologInterpreter new.
prolog refute: '?- append (X, Y, [1, 2, 3, 4]).'
       action:
           [:ans |
            ans keysAndValuesDo:
                [:varname :value |
                  Transcript
                    show: varname , '=' , value printPrologString;
                    cr].
            false]
```
   In the Transcript:
```
?- append(X,Y,[1,2,3,4]).
Y=[1,2,3,4]
X=[]
Y=[2,3,4]
X=[1]
Y=[3,4]
X=[1,2]
Y=[4]
X=[1,2,3]
Y=[]
X=[1,2,3,4]
no
```

2.2.2 From Prolog to Smalltalk

Within Prolog, one can evaluate a Smalltalk expression with
`send(smalltalk_object, message, [arguments],prolog_variable)`
and handle the result bounded to the `prolog_variable` within Prolog.

The method associated with the `message` in in charge of returning an object known to prolog (PrologString, PrologSymbol, PrologList, ...) ; otherwise a PrologObject will be returned.

If `smalltalk_object` is `self`, it represents the running instance of `PrologInterpeter`. By subclassing the core interpreter, it is easy to add new methods usable from Prolog.

The `smalltalk_object/argument` can be computed in Smalltalk by enclosing into braces `{}`.

Examples
In the shell (answers written after –>):

```
?- send(5, factorial, [], X). –> X = 120
?- send({5.35 floor factorial}, lcm:, [{5 + 45}], X).–> X = 600
?- send({5.35 floor factorial}, printStringRadix:, [16], X).–> X = {'78'}
```

The last result is aPrologObject (representing a Smalltalk string) not usable directly by prolog, but by Smalltalk:

```
?- send({'78'}, size, [],S). –> S = 2
```

2.3 Unification

There is no method to unify a term, but it is possible with code like below in a workspace.

```
| unification result prolog |
prolog := PrologInterpreter new.
unification :=
       [:term1 :term2 |
       "Returns unification of term1 and term2, false if it can not be unify."
       | parser pterm1 pterm2 valueEnv successFlag |
       parser := PrologParser new.
       pterm1 := (parser on: (ReadStream on: term1)) scanExpression.
       pterm2 := (parser on: (ReadStream on: term2)) scanExpression.
       valueEnv := prolog
                     resolveInitialize;
                     nullEnv.
       successFlag := prolog
                        unify: pterm1
                        env: valueEnv
                        and: pterm2
                        env: valueEnv.
       successFlag ifFalse: [false]
                   ifTrue: [
             "unification successful. To display unifier, restrict to the part
              related to the variables appearing in the original two terms. "
             | variableDict |
             variableDict := Dictionary new.
             prolog collectVariables: pterm1 to: variableDict.
             prolog collectVariables: pterm2 to: variableDict.
             variableDict keys do:
                  [:varName | | prologVar varValue |
                   prologVar := PrologVariable install: varName.
                   varValue := prolog represent: prologVar env: valueEnv.
                   variableDict at: varName put: varValue printPrologString].
             variableDict]].
```

Examples of use.

1. no unification:

```
result := unification value: 'f (X, g (Y))' value: 'f (g (Y), h (X))'.
Transcript show: result printString; cr.
```

   –> `false`

2. unification successful:

```
result := unification value: 'f (X, h (0))' value: 'f (g (Y), Y)'.
Transcript show: result printString; cr.
```

   –> `Dictionary ('Y' -> 'h(0)' 'X' -> 'g(h(0))' )`

As with ordinary Prolog processing systems, occurence check is not done. For example, if you try to unify `X` and `f(X)`, an infinite loop will occurs.

2.4 The “is” predicate

There is also a higher-order predicate like "is", but it is not perfect.
Example:

```
?- is (Z, F(12, 3)).
```

Result:

```
F = +
Z = 15
F = -
Z = 9
F = *
Z = 36
F = /
Z = 4
F = F1
Z = F1(12,3)
```

3 Prolog (PiS) manual

“Prolog In Smalltalk” (PiS) is a processing system (interpreter) of Prolog conforming to Dec-10 Prolog notation.
Important : Everything in PiS is prefix notation, prefix / infix / postfix with op predicates cannot be defined.

3.1 constants

The constants of PiS are as follows.

- symbol
- number
- String
- List
- object

3.1.1 Symbols

Symbols are alphanumeric characters that start with lowercase letters. If there are capital letters or special characters, enclose with single quotes.
If you want to have ’ in a symbol, write ”.
Examples:

```
aoki
atsushi
symbol 12345
'AOKI Atsushi'
'You are ''cool''.'
```

3.1.2 Numbers

Numbers are written in the same way as Smalltalk number objects.

```
123.456
-123.456
123.456e7
-123.456e-7
16r123
```

3.1.3 Strings

Enclose the character string within a pair of "...". Put "" in the string for a double quote.

```
"AOKI Atushi"
"Prolog is ""cool""."
```

3.1.4 Lists

Lists are expressed using brackets, with elements separated by commas.

```
[a,b,c]
```

The empty list is expressed as a pair of brackets:

```
[]
```

As in Lisp, a list has a head and a tail: `[a,b]` is equivalent to `[a | [b | []]]`.
The head of the previous list is `a`, the tail is `[b | []]`.
A Lisp "dotted pair" is `[a | b]`, where the tail is not a list.

Examples:

- `[AOKI | Atsushi]` is a dotted pair, `AOKI` is the head, `Atsushi` is the tail.
- Using a comma (`,`) instead of a pipe (`|`) in`[AOKI, Atsushi]`, `AOKI"` becomes the head, `[Atsushi]` is the tail part.
- `[AOKI, Atsushi]`, `[AOKI | [Atsushi]]` and `[AOKI | [Atsushi | []]]` are synonymous.
- `[]` is the empty list.

3.2 Smalltalk Objects

Any Smalltak expression is available in Prolog by surrouding it with braces `{ . . . }`. The expression is evaluated, ant the result is handled as a constant of Prolog.
In addition, `{}` is synonymous with `{nil}`.

Examples:

```
:- send({#(1 2 3)}, asList, [], L),
   send({[:a | Prolog.PrologList list: (a at: 2 put: 10; yourself)]},
        value:, [L], Z).
```

succeeds with:

```
L = {List (1 2 3)}
Z = [1,10,3]
```

3.3 Variables

Variables are alphanumeric characters that begin with an uppercase letter.
The anonymous variable is the character *tilde* ~ (not the character underline _ , as in the others implementations of Prolog)

Examples of variables:

```
AOKI
Atsushi
Variable1234
~
```

Code example:

```
likes (aoki, smalltalk).
likes (aoki, prolog).
likes (watanabe, smalltalk).
likes (watanabe, lisp).
likes (sakoh, lisp).
likes (sahara, prolog).
?- likes(~,smalltalk).
```

```
yes
?- likes(Who,lisp).
Who = watanabe
Who = sakoh
no
?- likes(watanabe,~).
yes
?- likes(sahara,What).
What = prolog
no
```

Why use ∼ for anonymous variables ?
"_" is used in some flavors of Smalltalk for substitution (also "←" or ":=") and "_" can appear in Smalltalk message names. Therefore, to ease the interface between Smalltalk and Prolog, "∼" is used for the anonymous variable.

3.4 System predicates

Predicates embedded in Prolog's system beforehand are described as system predicates.
One can find the corresponding prolog code in the Smalltalk methods

```
PrologInterpreter>>systemPredicatesNoX  (private protocol, X=0..9)
```

They are divided into the following ten groups:

- Embedded system predicate (systemPredicatesNo0)
- Basic system predicate (systemPredicatesNo1)
- System predicate on comparison operation (systemPredicatesNo2)
- System predicate on arithmetic operations (systemPredicatesNo3)
- System predicate on clauses and terms (systemPredicatesNo4)
- High-order system predicate (systemPredicatesNo5)
- System predicate on input / output (systemPredicatesNo6)
- System predicate for debugging (systemPredicatesNo7)
- Other system predicates (systemPredicatesNo8)
- User defined system predicate (systemPredicatesNo9)

3.4.1 Embedded system predicate (systemPredicatesNo 0)

!
The goal ! (pronounced 'cut') in the body of a rule always succeeds when first evaluated. On backtracking it always fails and prevents any further evaluation of the current goal, which therefore fails.
Note that backtracking over a cut not only causes the evaluation of the current clause to be abandoned but also prevents the evaluation of any other clauses for that predicate.

Since it removes branches while backtracking, it can increase efficiency of execution.
However, because it is a predicate that intentionally invalidates Prolog's backtracking, it is vital to use it properly.
There are three cases where cuts are properly used:

- to tell Prolog that the correct direction for the target has been found
- you want to instantly fail a goal
- you want to stop generating another solution

Example:

```
% incorrect program
classify(0,zero).
classify(N,negative):- <(N,0).
classify(N,positive).

?- classify(-5,X).
X = negative
X = positive
no
?- classify(0,X).
X = zero
X = positive
no
?- classify(5,X).
X = positive
no
```

```
% correct program using cut
classify(0,zero):- !.
classify(N,negative):- <(N,0), !.
classify(N,positive).

?- classify(-5,X).
X = negative
no
?- classify(0,X).
X = zero
no
?- classify(5,X).
X = positive
no
```

true

It is a predicate that always succeeds. It is a predicate which is not actually needed, but here for convenience.

fail

It is a predicate that always fails to intentionally cause backtracking.

Example:

```
output (X, Y):- write (X), write (","), write (Y), nl.
?- append (X, Y, [1, 2, 3]), output (X, Y), fail.
[],[1,2,3]
[1],[2,3]
[1,2],[3]
[1,2,3],[]
no
```

Also, combination with cut (`!`) is also effective: `"...,!, Fail."` is a way to abort the current goal. For example, the predicate `not(G)`, denying that a certain target `G` has been satisfied, can be defined as:

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

If target `G` fails while executing the first clause, Prolog reach the second section `"not (G)."` to satisfy the goal. Conversely, if `G` is successful, while executing the first clause, the goal will fail without backtracking.

var(X)

Succeeds when X is a variable that has not been assigned.

```
?- is(X,100), var(X).
no
?- is(X,Y), var(X).
X = X35
Y = X35
no
```

In order to prompt the user for input when a certain variable is not assigned, the idiom

```
"..., var (X), read (X), ..."
```

is useful.

send (X, Y, Z) **send (X, Y, Z, A)**

Predicate to send Smalltalk's message.

– `X` is a Smalltalk object, (Smalltalk code between braces for example)

- `Y` is the Smalltalk message name sent to the object
- `Z` is the argument list for the message
- `A` is unified to the result of the message

if `X` is `self`, the current PrologInterpreter is the Smalltalk object.
Example

```
% C is unified with a Smalltalk object (an OrderedCollection)
?- send({Number},allSubclasses,[],C).
C = {OrderedCollection (FixedPoint Fraction Integer SmallInteger LargeInteger
LargeNegativeInteger LargePositiveInteger LimitedPrecisionReal Float Double
SmallDouble)}
no

% C unified with a Prolog list of Prolog strings.
% Job is done in the Smalltalk block.
?- send({
        [Prolog.PrologList list:
            (Number allSubclasses collect:
                    [:c | Prolog.PrologString fromString: c name asString])
        ]},value,[],List).
List = ["FixedPoint","Fraction","Integer","SmallInteger","LargeInteger",
"LargeNegativeInteger","LargePositiveInteger","LimitedPrecisionReal","Float",
"Double","SmallDouble"]
no

% with self
?- send(self,printString,[],N).
N = {'a Prolog.PrologShellInterpreter'}
no
% with arguments
?- send(255,bitAt:put:,[8,0],N).
N = 127
no
```

3.4.2 Basic system predicate (systemPredicatesNo1)

repeat
It is a predicate to repeat, it always succeeds.

```
test :- repeat, read(S), write(S), nl , =(N,""), !.
:-test.
```

The previous code asks the user until the answer is the empty string.
nonvar(X)
Succeeds if X is an unassigned variable. It will fail if it is a substituted variable or constant.

```
?- is(X,100), nonvar(X).
X = 100
no
?- is(X,Y), nonvar(X).
no
```

integer(X)
Succeeds if X is an integer, otherwise it fails.
float(X)
Succeeds if X is a single-precision floating-point number, otherwise it fails.
double (X)
Succeeds if X is a double-precision floating-point number, otherwise it fails.
fraction(X)
Succeeds if X is a fraction, otherwise it fails.

number(X)
Succeeds if X is a number, otherwise it fails.
symbol (X)
If X is a symbol it succeeds, otherwise it fails.
string (X)
Succeeds if X is a string, otherwise it fails.
list(X)
If X is a list it succeeds, otherwise it fails.
dotp(X)
It succeeds if X is a cell (dot pair), otherwise it fails. The empty list fails.
atom(X)
Succeeds if X is an atom, otherwise it fails.

```
?- atom(a).
yes
?- atom("a").
yes
?- atom([]).
yes
?- atom([a,b,c]).
no
?- atom(123).
no
?- atom({Collection}).
no
?- atom(X).
no
```

atomic(X)
Succeeds if X is an atom or number, and fails otherwise.
structure(X)
Fails if X is an atom or number, otherwise it succeeds.

3.4.3 System predicate on comparison operation (systemPredicatesNo 2)

== (X, Y)
Succeeds when X and Y are equal in Smalltalk, and fails otherwise.

\== (X, Y)
Smalltalk fails when X and Y are equal, otherwise it succeeds.
= (X, X)
Succeeds when X and Y are equal, otherwise it fails.
\= (X, Y)
It fails when X and Y are equal, otherwise it succeeds.
> (X, Y)
Succeeds if X is greater than Y, otherwise it fails.
>= (X, Y)
Succeeds if X is greater than or equal to Y, otherwise it fails.
<(X, Y)
Succeeds if X is less than Y, otherwise it fails.
=<(X, Y)
Succeeds if X is less than or equal to Y, otherwise it fails.

3.4.4 System predicate on arithmetic operations (systemPredicatesNo 3)

+ (X, Y, Z)
Succeeds if the sum of X and Y equals Z, otherwise it fails.

```
?- +(12,5,X).
X = 17
no
```

- (X, Y, Z)

It succeeds if the subtraction of X and Y is equal to Z, otherwise it fails

```
?- -(12,5,X).
X = 7
no
```

*** (X, Y, Z)**

Succeeds if the product of X and Y is equal to Z, otherwise it fails

```
?- *(12,5,X).
X = 60
no
```

// (X, Y, Z)

If X and Y are rounded down and the remainder is rounded down, it is successful if it is equal to Z Otherwise it fails.

```
?- //(12,5,X).
X = 2
no
```

/ (X, Y, Z)

It succeeds if the division of X and Y equals Z, otherwise it fails.

```
?- /(12,5,X).
X = (12 / 5)
no
?- /(12.0,5,X).
X = 2.4
no
```

\\(X, Y, Z)

It succeeds if the modulo operation of X and Y is equal to Z, otherwise it fails.

```
?- \\(12,5,X).
X = 2
no
?- \\(12.3,5.1,X).
X = 2.1
no
```

is (X, Y)

Succeeds when X and Y are equal, otherwise it fails.

```
?- is(12,12).
yes
?- is(12,X).
X = 12
no
?- is(X,12).
X = 12
no
```

is (Z, F(X, Y))

F must be in the set $\{+, -, *, /\}$. Unification is done on any variable, and you must filter by `number(X)` to avoid the term.

```
?- is(12,*(6,2)).
yes
?- is(*(6,2),12).
no
?- is(X,*(6,2)).
X = 12
X = *(6,2)
```

```
no
?- is(X,*(6,2)), number(X).
X = 12
no
?- is(X,F(6,2)).
F = +
X = 8
F = -
X = 4
F = *
X = 12
F = /
X = 3
F = F1
X = F1(6,2)
no
?- is(X,F(6,Z)).
F = F1
X = F1(6,Z1)
Z = Z1
no
```

3.4.5 System predicate on clauses and terms (systemPredicatesNo 4)

listing
Output all clauses of user registered predicates and always succeed.
listing (X)
Outputs the clauses with the user predicate defined by the symbol specified by X and succeeds. Fail if there is no corresponding predicate.
systemListing
Output all registered system predicate clauses and always succeed.
systemListing (X)
Output the clauses with the system predicate defined by the symbol specified by X and succeed. Fails if there is no corresponding system predicate.

```
?- systemListing(member).
member(X,[X|Y]).
member(X,[Y|Z]) :-
        member(X,Z).
yes
```

consult (X)**********TODO
Read and load Prolog program from the file specified by X
We will add it at the end and succeed. Corresponding file does not exist
Farewell fails.

reconsult (X)**********TODO
Read and load Prolog program from the file specified by X
We will replace it with the same predicate clause and succeed. Corresponding file
If it does not exist it fails.

saving**********TODO
The clauses of all the registered predicates are specified in the input dialog
It outputs to the IL, and it always succeeds.

saving (X)**********TODO
A clause with the predicate of the symbol specified by X is specified in the input dialog
It outputs to the file and succeeds. It fails if there is no corresponding predicate

.

userPredicates (X)
It lists all the user registered predicates and unifies it with X.

systemPredicates (X)
List all registered system predicates X and Unify with X.

predicates ([X | Y])
Unifies all user registered predicates with X and all system predicates with Y.

functor (T, F, A)
This predicate implies that a structure T has function F and arity A (A arguments).

```
?- functor(likes(aoki,smalltalk),F,A).
A = 2
F = likes
no
```

**********TODO ? **********

'~addvar'(L,N,M)
En voici un test :

```
:- remove.
'~addvar'(L,0,M) :- !, =(L,M).
'~addvar'(L,NVars,M) :- -(NVars, 1, N), append(L,[A],LV), '~addvar'(LV,N,M).
:- '~addvar'([a],1,M).

?- remove.
yes
?- '~addvar'([a],1,M).
M = [a,A3]
no
```

****

arg (N, S, T)
This predicate means that the Nth argument of structure S is T. N and S must be assigned.

```
?- arg(2,likes(aoki,smalltalk),A).
A = smalltalk
no
```

=.. (X, L)
This predicate is called "Univ", L is unified to a list consisting of X's function and arguments.

```
?- '=..'(likes(aoki,smalltalk),L).
L = [likes,aoki,smalltalk]
no
?- '=..'(X,[likes,aoki,smalltalk]).
X = likes(aoki,smalltalk)
no
```

name (A, L)
Unifies a symbol A with a string or a list of character codes L.

```
?- name(apple,L).
L = "apple"
no
?- name(A,[97,112,112,108,101]).
A = apple
no
?- name(A,"apple").
A = apple
no
```

remove

Delete all clauses of user registered predicates. This predicate always succeeds.

remove (X)
Delete the clauses whose predicate is the symbol specified by X and succeed. Fail if there is no corresponding predicate.

clause (X)
Unify the clause specified by X to the registered clause.

```
% definition of concat
concat ([], X, X).
concat ([A | X], Y, [A | Z]) :- concat (X, Y, Z).

?- clause([[concat|Arguments]|Body]).
Arguments = ([],X6,X6)
Body = []
Arguments = ([A11|X11],Y11,[A11|Z11])
Body = concat(X11,Y11,Z11)
no
?- clause([concat(|Arguments)|Body]).
Arguments = ([],X6,X6)
Body = []
Arguments = ([A11|X11],Y11,[A11|Z11])
Body = concat(X11,Y11,Z11)
no
```

In Smalltalk, `Arguments` is an instance of `PrologTerms`, `Body` is an instance of `PrologBody`. These classes are used by the interpreter, and are not easily accessible in Prolog.

asserta (X)
Add the clause specified by X to the beginning of the clauses indicated by that predicate.

assert (X) , assertz (X)
Add the clause specified by X to the end of the clauses indicated by that predicate.
With this code:

```
?- remove.
:- asserta (likes (aoki, smalltalk)).
:- asserta (likes (wanatabe, smalltalk)).
:- asserta (likes (wanatabe, lisp)).
:- assertz ([likes (X, prologInSmalltalk),
             likes (X, smalltalk), likes (X, lisp)]).
:- assertz ([on (cat, dog)]).
:- asserta ([on (hen, cat)]).
:- assertz ([on (dog, donkey)]).
:- assertz ([above (X, Y):- on (X, Y)]).
:- assertz ([above (X, Y), on (X, Z), above (Z, Y)]).
```

you can query the listing:

```
?- listing.
above(X,Y) :-
        on(X,Y).
above(X,Y) :-
        on(X,Z),
        above(Z,Y).
likes(wanatabe,lisp).
likes(wanatabe,smalltalk).
likes(aoki,smalltalk).
likes(X,prologInSmalltalk) :-
        likes(X,smalltalk),
        likes(X,lisp).
on(hen,cat).
```

```
on(cat,dog).
on(dog,donkey).
yes
```

and query who likes what:

```
?- likes(X,Y).
X = wanatabe
Y = lisp
X = wanatabe
Y = smalltalk
X = aoki
Y = smalltalk
X = wanatabe
Y = prologInSmalltalk
no
```

retract (X)

Unify the clause specified by X to the registered clause and if successful, that clause will be deleted.

```
:- remove.
concat ([], X, X).
concat ([A | X], Y, [A | Z]) :- concat (X, Y, Z).

?- retract([[concat|Args]|Body]).
Args = ([],X6,X6)
Body = []
Args = ([A11|X11],Y11,[A11|Z11])
Body = concat(X11,Y11,Z11)
no
?- listing(concat).
no
```

3.4.6 High-order system predicate (systemPredicatesNo 5)

call (G)

G is a target goal. `call(G)` tries to satisfy G, and is successful if G succeeds.

```
:- =(F,is), member(Op, [+, *]),
    call(and( F(X,Op(5, 2)),
              number(X)   )).
F = is
Op = +
X = 7
F = is
Op = *
X = 10
no
```

not (G)

This goal will fail if G can be satisfied as a target, and will be successful if G cannot be satisfied.

or (X, Y)

Represents a disjunction of X and Y. If X succeeds or Y succeeds, the goal will be successful. When X fails, a satisfaction of Y is attempted. At this time, If Y fails too, the disjunction will fail.

and (X, Y)

Represents the conjunction of X and Y. If X succeeds and Y also succeeds, the goal will be successful.

3.4.7 System predicate on input / output (systemPredicatesNo 6)

The next two predicates are not defined in the core interpreter, but in PrologShellInterpreter.

read (X)
Unify input with X as a string from the input dialog.
read (X, M)
Unify input with X as a string from the input dialog, with M as the message of the dialog.
write (X)
X is written on the output. It is a predicate that always succeeds.
nl
Output a newline on the output. It is a predicate that always succeeds.
tab (X)
Will only output X tabs. It fails if X is not assigned.
clear
Clean the output. It is a predicate that always succeeds.

3.4.8 System predicate for debugging (systemPredicatesNo 7)

clock (X)
Unify the current time to X in milliseconds.
verbose (X)
When X is true, the satisfaction time and the number of goals will be written on output.
If X is not true, nothing is written.
gc
Calls Smalltalk garbage collection. This predicate always succeeds.
inspect (X)
Open a Smalltalk inspector on X. This predicate always succeeds.
spy (X)
Start tracing the clause whose predicate is the symbol specified by X.
nospy (X)
Finish the trace of the clause whose predicate is the symbol specified by X.
trace
Start tracing all registered clauses.
notrace
End the tracing of all registered sections.

3.4.9 Other system predicates (systemPredicatesNo 8)

Classical and nearly mandatory predicates. Any Prolog book describe them ...
append (X, Y, Z)
The list connecting X and Y is Z.
member (X, Y)
X is an element of the Y list.
reverse (X, Y)
The reverse list of X is Y's list.
length (X, Y)
Length of list X is Y.
nth (X, Y, Z)
The Yth element of the list X is Z.
printlist (L)
It sequentially outputs the elements of the list L.
lispAppend (X, Y, Z)
lispReverse (X, Y)
lispMember (X, Y)
lispMember (X, Y, Z)
lispAssoc (X, Y)
lispAssoc (X, Y, Z)
lispNconc (X, Y, Z)

These are predicates that call quasi-Lisp functions (implemented in Smalltalk).
They are fast, but backtracking is not done.

3.4.10 User defined system predicate (systemPredicatesNo 9)

There is currently no user-defined system predicate.
The Smalltalk method `PrologInterpreter>>systemPredicatesNo9` is a dummy method where you can define your owns.

3.4.11 System predicate list

!	(systemPredicatesNo 0)	lispNconc	(systemPredicatesNo 8)
*	(systemPredicatesNo 3)	lispReverse	(systemPredicatesNo 8)
+	(systemPredicatesNo 3)	list	(systemPredicatesNo1)
-	(systemPredicatesNo 3)	listing	(systemPredicatesNo 4)
/	(systemPredicatesNo 3)	member	(systemPredicatesNo 8)
//	(systemPredicatesNo 3)	name	(systemPredicatesNo 4)
<	(systemPredicatesNo 2)	nl	(systemPredicatesNo 6)
=	(systemPredicatesNo 2)	nonvar	(systemPredicatesNo 1)
= ..	(systemPredicatesNo 4)	nospy	(systemPredicatesNo 7)
= <	(systemPredicatesNo 2)	not	(systemPredicatesNo 5)
==	(systemPredicatesNo 2)	notrace	(systemPredicatesNo 7)
>	(systemPredicatesNo 2)	nth	(systemPredicatesNo 8)
> =	(systemPredicatesNo 2)	number	(systemPredicatesNo1)
and	(systemPredicatesNo 5)	or	(systemPredicatesNo 5)
append	(systemPredicatesNo 8)	predicates	(systemPredicatesNo 4)
arg	(systemPredicatesNo 4)	printlist	(systemPredicatesNo 8)
assert	(systemPredicatesNo 4)	read	(systemPredicatesNo 6)
asserta	(systemPredicatesNo 4)	reconsult	(systemPredicatesNo 4)
assertz	(systemPredicatesNo 4)	remove	(systemPredicatesNo 4)
atom	(systemPredicatesNo1)	repeat	(systemPredicatesNo1)
atomic	(systemPredicatesNo1)	retract	(systemPredicatesNo 4)
call	(systemPredicatesNo 5)	reverse	(systemPredicatesNo 8)
clause	(systemPredicatesNo 4)	saving	(systemPredicatesNo 4)
clear	(systemPredicatesNo 6)	send	(systemPredicatesNo 0)
clock	(systemPredicatesNo 7)	spy	(systemPredicatesNo 7)
consult	(systemPredicatesNo 4)	string	(systemPredicatesNo1)
dotp	(systemPredicatesNo1)	structure	(systemPredicatesNo1)
double	(systemPredicatesNo1)	symbol	(systemPredicatesNo1)
editing	(systemPredicatesNo 4)	systemListing	(systemPredicatesNo 4)
fail	(systemPredicatesNo 0)	systemPredicates	(systemPredicatesNo 4)
float	(systemPredicatesNo1)	tab	(systemPredicatesNo 6)
fraction	(systemPredicatesNo1)	trace	(systemPredicatesNo 7)
functor	(systemPredicatesNo 4)	true	(systemPredicatesNo 0)
gc	(systemPredicatesNo 7)	userPredicates	(systemPredicatesNo 4)
inspect	(systemPredicatesNo 7)	var	(systemPredicatesNo1)
integer	(systemPredicatesNo1)	verbose	(systemPredicatesNo 7)
is	(systemPredicatesNo 3)	write	(systemPredicatesNo 6)
length	(systemPredicatesNo 8)	\ =	(systemPredicatesNo 2)
lispAppend	(systemPredicatesNo 8)	\ ==	(systemPredicatesNo 2)
lispAssoc	(systemPredicatesNo 8)	\\	(systemPredicatesNo 3)
lispMember	(systemPredicatesNo 8)		