**Prolog** is a [logic programming](https://en.wikipedia.org/wiki/Logic_programming)language associated with [artificial intelligence](https://en.wikipedia.org/wiki/Artificial_intelligence) and [computational linguistics](https://en.wikipedia.org/wiki/Computational_linguistics).[[1]](https://en.wikipedia.org/wiki/Prolog#cite_note-Clocksin2003-1)[[2]](https://en.wikipedia.org/wiki/Prolog#cite_note-Bratko2012-2)[[3]](https://en.wikipedia.org/wiki/Prolog#cite_note-Covington1994-3)

Prolog has its roots in [first-order logic](https://en.wikipedia.org/wiki/First-order_logic), a [formal logic](https://en.wikipedia.org/wiki/Formal_logic), and unlike many other [programming languages](https://en.wikipedia.org/wiki/Programming_language), Prolog is intended primarily as a [declarative](https://en.wikipedia.org/wiki/Declarative_programming)programming language: the program logic is expressed in terms of relations, represented as facts and [rules](https://en.wikipedia.org/wiki/Rule_of_inference). A computation is initiated by running a *query* over these relations.[[4]](https://en.wikipedia.org/wiki/Prolog#cite_note-lloyd84-4)

The language was first conceived by Alain Colmerauer and his group in Marseille, France, in the early 1970s and the first Prolog system was developed in 1972 by Colmerauer with Philippe Roussel.[[5]](https://en.wikipedia.org/wiki/Prolog#cite_note-Kowalski-5)[[6]](https://en.wikipedia.org/wiki/Prolog#cite_note-6)

Prolog programs specify relationships among objects and properties of object

When we say “Amit has a Bike”, we are declaring the ownership relationship between two objects; Amit and the bike

When we ask, “Does Amit own the bike?” Then we are trying to find out about a relationship.

Relationships can be also rules such as

Two people are brother if

They are both male and

They both have the same parents(F and M)(A and B are not same)

A rule allows us to find out about a relationship even if the relationship isn’t explicitly stated as a fact.

**Facts Rules and Queries**

In Prolog program we declare facts describing explicit relationships between objects and properties objects might have( e.g. Tulika likes ice-cream,Hair is black, Maruti is a company, Mini is a cat, Zahir teaches Saleem)

We Define rules defining implicit relationships between objects (e.g. brother relationships) and/or rules defining implicit object properties(A is a child of B if B is parent of A)

Ask questions to generate queries.

**Facts:**

Facts are properties of objects or relationships between objects:

Ram has phone number 123123123” is written in prolog as

phoneno(ram,123123123).

It should be noted that:

* Names of properties /relationships begin with lowercase letter
* The Relationship name appears as the first term
* Objects appear as comma separated arguments within parenthesis
* A period “.” Must end a fact
* Objects also begin with lower case letters. The also can begin with digits like 1234 and can be strings of characters enclosed in quotes e.g color(penink,’red’).
* phoneno(ram,123123123). Is also called a predicate or a clause

**Facts about a Hypothetical Department of Physics-**

%teaches(X,Y): person X teaches course Y

teaches(ram, CS301).

teaches(shyam, CS302).

%student(X,Y): person X student course Y

student(hari, CS301).

student(shiva, CS302).

Together, these facts will form Prolog’s database/ Knowledge base

**Rules**

Consider the following case which produc3es a general rule-

One teacher will guide a student if the student studies that very course id on which the teacher teaches

In prolog this will be written as:

Guide(Teacher,Student):-

Teaches(Teacher,Coursed).

Studies(Student, Courseid).

Facts are unit clauses and rules are non-unit clauses.

Variable name will start with a capital letter.

**Rules and facts**[[edit](https://en.wikipedia.org/w/index.php?title=Prolog&action=edit&section=3)]

Prolog programs describe relations, defined by means of clauses. Pure Prolog is restricted to [Horn clauses](https://en.wikipedia.org/wiki/Horn_clauses). There are two types of clauses: facts and rules. A rule is of the form

Head :- Body.

and is read as "Head is true if Body is true". A rule's body consists of calls to predicates, which are called the rule's **goals**. The built-in [predicate](https://en.wikipedia.org/wiki/Predicate_(mathematics)) ,/2(meaning a 2-arity [operator](https://en.wikipedia.org/wiki/Operator_(programming)) with name ,) denotes [conjunction](https://en.wikipedia.org/wiki/Logical_conjunction) of goals, and ;/2 denotes [disjunction](https://en.wikipedia.org/wiki/Logical_disjunction). Conjunctions and disjunctions can only appear in the body, not in the head of a rule.

Clauses with empty bodies are called **facts**. An example of a fact is:

cat(tom).

which is equivalent to the rule:

cat(tom) :- true.

Clauses with bodies are called **rules**. An example of a rule is:

animal(X) :- cat(X).

**Queries**

Queries will be based on facts and rules. We can ask questions based on the stored information.

Suppose we want to know if ram lectures in CSC301 or not, then we can ask:

?-teaches(ram,CSC301).

Yes

In GNU prolog, queriesd are terminated by a full stop.

To answer this query, prolog consults its database to see if this is a known fact or not.

We can also ask-

?-teaches(ram,X).

X=CSC301

If answer is true or yes then the query is succeeded

If answer is false or no then the query failed.

Syntax of a clause

:- means “if” or “is implied by”. Also called the neck symbol.

The left hand side of the neck is called the head.

The right hand side of the neck is called the body.

The comma, “,” stands for and/ conjunction

The semi-colon “;” stands for or/ disjunction

A Program consists of clauses. These are three types: fact, rules and queries. A procedure is a set of clauses about the same relation.

Example on clause writing:

P:- Q;R.

Can also be written as

P:-Q.

P:-R.

P:-Q,R;S,T,U.

Is understood as

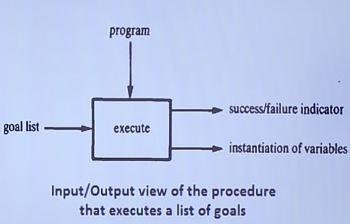
P;-(Q,R);(S,T,U).

Can also be written as

P:-Q,R.

P:-S,T,U.

How does a prolog program execute

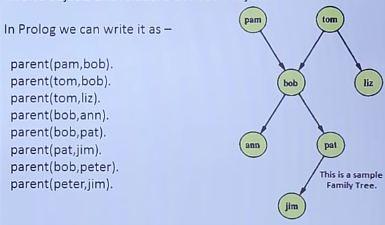


**Declarative and procedural semantics**

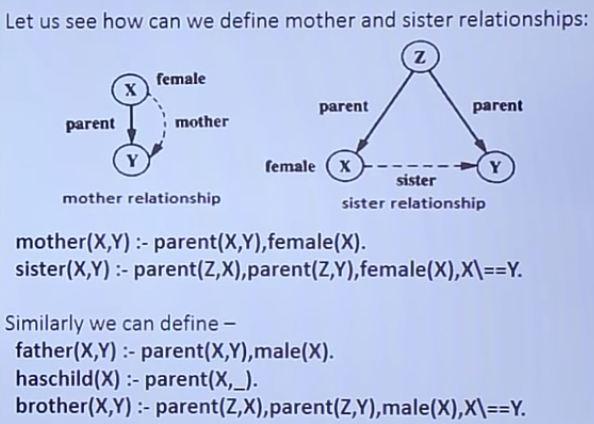
**The Declarative Semantics**  of prolog defines whether a goal is true with respect to a given program, if it is true, for what instantiation of variables it is true.

**The Procedural Semantics**  of prolog is a procedure for satisfying a list of goals in the context of a given program. The procedure outputs the truth or falsity of the goal list and the corresponding instantiations of variables. The procedure automatically backtracks to examine alternatives.

**Family Relationship in prolog**

****

The arguments of relations can {among other things} be: **concrete objects, or constants (**such as pat and jim**),** general objects such as X and Y. Objects of the first kind in our programs are called **atoms.** Objects of the second kind are called variables.



X\==Y means X is not equal to Y

\_ means some variable X in prolog

Example of execution using trace and no trace

{trace}

| ?- mother(pam,X).

1 1 Call: mother(pam,\_42) ?

2 2 Call: parent(pam,\_42) ?

2 2 Exit: parent(pam,bob) ?

3 2 Call: female(pam) ?

3 2 Exit: female(pam) ?

1 1 Exit: mother(pam,bob) ?

X = bob

yes

{trace}

| ?- mother(tom,X).

1 1 Call: mother(tom,\_42) ?

2 2 Call: parent(tom,\_42) ?

2 2 Exit: parent(tom,bob) ?

3 2 Call: female(tom) ?

3 2 Fail: female(tom) ?

2 2 Redo: parent(tom,bob) ?

2 2 Exit: parent(tom,liz) ?

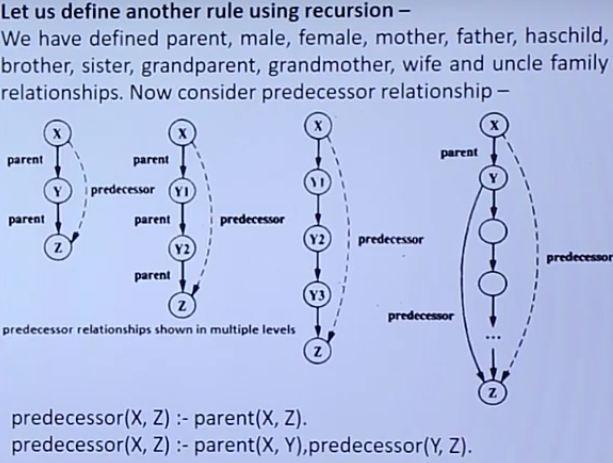
3 2 Call: female(tom) ?

3 2 Fail: female(tom) ?

1 1 Fail: mother(tom,\_42) ?

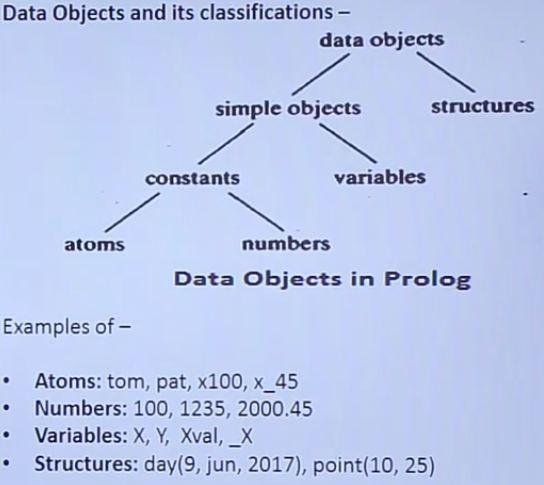
(31 ms) no

**Recursion in Family relationship:Prolog**



Check with trace predecessor(peter,X).

**Data Objects in prolog**

****

An **Atom** is a general-purpose name with no inherent meaning.

1. It can be strings of letters,digits and the underscore character,’\_’, starting with a lower-case letter. As example: azahar,b59,b\_59,b\_59AB,b\_x25,antara\_sarkar etc
2. Strings of special characters:

< ------ >

=======>

……..

.:.

::=

When using atoms of this form, some care is necessary because some strings of special characters already have a predefined meaning; an example is ‘:-‘.

1. Strings of characters enclosed in single quotes. This is useful if we want,

For example to have an atom that starts with a capital letter. By inclosing it in quotes we make it distinguishable from variables:

‘Rubai’

‘Ram\_Bahadur’

‘Hari Parsad’

**Numbers**

Numbers can be **floats** or **integers** (-16383 to 16383)

e.g. 100,4,-89,1020

e.g. 3.14159,-0.00062,450.18

**Variables**

Variables are strings of letters, digits and underscore characters. They start with an upper-case letter or an underscore character. They resemble placeholders for arbitrary terms E.g.

X

Sum

Memer\_name

Student\_list

Shoppinglist

\_a50

\_15

In a clause when a variable is used once only then the variable name can be replaced by so-called ‘anonymous’ variable, which is written as a single underscore’\_’ character. For example-

hasachild(X):-parent(X,Y). can also be written as,

hasachild(X):-parent(X,\_).

**Structures(compound terms)-functor followed by comma separated arguments**

Structures are objects that have multiple components. The components themselves can, in turn, be structures

For example the date can be viewed as a structure with three components: day,month,year.

Then the date 9th june,2017 can be written as :

date(9,june,2017)

**Tree representation**

**root of tree: functor**

**subtrees: arguments**

**Example**

P**rolog representation Functor:date,** **Arguments 9,june,2017**

The number of arguments is called the terms **arity**.

 An atom(rule of naming) can be regarded as a compound term with **arity** zero

Example

To represent a point, a line segment and a triangle using structure in prolog, consider the following statements.

P1:point(1,1)

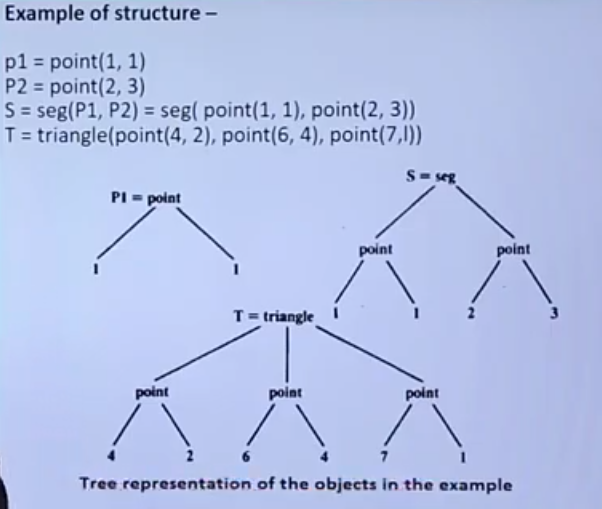
P2:point(2,3)

S:seg(P1,P2): seg(point(1,1),point(2,3))

T: triangle(point(4,Z),point(6,4),point(7,1))

Special cases of compound terms:

* A *List* is an ordered collection of terms. It is denoted by square brackets with the terms separated by commas or in the case of the empty list, []. For example, [1,2,3] or [red,green,blue].
* *Strings*: A sequence of characters surrounded by quotes is equivalent to either a list of (numeric) character codes, a list of characters (atoms of length 1), or an atom depending on the value of the Prolog flag double\_quotes. For example, "to be, or not to be".

****

**Representation of Lists**

The list is a simple data Structure widely used in non-numeric programming. List can be either empty or non-empty in the first case, the list is simply written as a prolog atom, []. IN the second case, the list can be viewed as consisting of two things:

1. The first item, called the **head** of the list;
2. The remaining part of the list called the **tail**.

For example

[red,green,blue,whte,dark]

The head is red and the tail is the list

[green,blue,white,dark]

Let us consider L=[a,b,c]

If we write tail=[b,c] then we can also write L=[a | Tail]

Here the vertical bar separates the head and tail.

So, following list representations are also valid

[a,b,c]= [ x | [ b , c ] ] = [a, b | [c] ] = [a, b, c | [ ] ]

**Operation on list :**

1. **Membership**

Problem statement:

To check whether an object X is member of List L or not.

list\_member(X, L).

Where X is an object and L is a list

The goal list\_members(X,L) is true if X occurs in L.

For example,

list\_member(b,[a, b, c]) is true

list\_member(b, [a, [b,c] ] is not true but

list\_member([b,c],[a,[b,c]]) is true.

The program for membership relation can be based on the following observation:

X is a member of L if either

* 1. X is the head of L, or
  2. X is a member of the tail of L.

1. **Length Calculation**

Problem Statement:

Calculate the number of items of a given list.

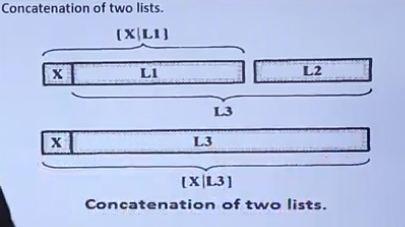
list\_length( List, N)

Which will count the elements in a list ‘List’ and instantiate ‘N’ to their number.

If the list is empty it has a length of 0

If the list is not empty then List =[Head | Tail]; then its length is equal to 1 plus the length of the tail Tail.

1. **Concatination of two lists**

****

1. **Operations in words**

**Problem statement**

We sdhall have to define a relation

inwords(List1,List2)

To translate a list of numbers between 0 to 9 to a list of the corresponding words. For Example:

inwords([3,5,1,3]), [three, five, one, three]).

Use the following as an auxillary relation:

interm(0,zero). interm(1,one). interm(2,two). … .

1. **Delete an Item**

We can have two cases

If X is the head of the list then the result after the deletion is the tail of the list

If X is in the tail then it is deleted from there

1. **\*Permutation**

Find all possible permutation of all the items of a given list

list\_permutation([a,b,c],P);

P: [a,b,c];

P: [a,c,b];

P: [b,a,c];

1. **\*Append**

list\_append(L1,L2,L3).

### The Cut

Automatic backtracking is one of the most characteristic features of Prolog. But backtracking can lead to inefficiency. Sometimes Prolog can waste time exploring possibilities that lead nowhere. It would be pleasant to have some control over this aspect of its behaviour, but so far we have only seen two (rather crude) ways of doing this: changing rule order, and changing goal order. But there is another way. There is a built-in Prolog predicate ! (the exclamation mark), called cut, which offers a more direct way of exercising control over the way Prolog looks for solutions.

What exactly is cut, and what does it do? It’s simply a special atom that we can use when writing clauses. For example,

p(X):- b(X), c(X), !, d(X), e(X).

is a perfectly good Prolog rule. As for what cut does, first of all, it is a goal that always succeeds. Second, and more importantly, it has a side effect. Suppose that some goal makes use of this clause (we call this goal the parent goal). Then the cut commits Prolog to any choices that were made since the parent goal was unified with the left hand side of the rule (including, importantly, the choice of using that particular clause).

http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse43

1. **Last element**

**lastelement([X],X).**

**lastelement([\_|T],Y) :- lastelement(T,Y).**

1. **nodoubles(L1,L2).**

L1 may contain repetition L2 is the desired output without repetition

1. **dsfs**
2. **ds**
3. **ds**

modus ponens

1. the rule of logic which states that if a conditional statement (‘if *p* then *q* ’) is accepted, and the antecedent ( *p* ) holds, then the consequent ( *q* ) may be inferred

**Operators**

**Comparison operators**

X>Y -> X is greater than Y

X<Y -> X is less than Y

X >= Y -> X is greater than or equal to Y

X =< Y -> X is less than or equal to Y

X =:= Y -> the X and Y values are equal

X =\= Y -> the X and Y values are not equal

For example

1+2=:=2+1. -> yes

1+2=2+1. -> no

1+A= B+2.

A=2

B=1

**Arithmetic Operators**

* = addition
* = subtraction

\* = multiplication

/ =division

\*\* =power

// = integer division

mod= modulo

Example

?- X =1+2

X = 1+2

?- X is 1+2.

X = 3

?- X is 5/2, Y is 5 // 2, Z is 5 mod 2, W is mod(10,3).

X =2.5

Y =2

Z=1

W=1

**Preventing Backtracking**

Prolog will automatically backtrack if this is necessary for satisfying a goal. Uncontrolled backtracking may cause inefficiency in a program. ‘Cut’ can be used to control or prevent backtracking

Experiment 1-

**Double step function :**

* Rule 1 : if X<3 then Y=0
* Rule 2 : if 3 <= X and X < 6 then Y = 2
* Rule 3 : if 6 <= X then Y= 4

**In prolog,**

f(X,0) :- X < 3. % Rule 1

f(X,2) :- 3 =< X, X<6. % Rule 2

f(X,4) :- 6 =< X. % Rule 1

**Question:**

?- f(1,Y), 2< Y.

The first goal f(1,Y) instantiated Y to 0.

The second goal becomes 2<0 which fails

Prolog tries through Backtracking two useless alternatives

(Rule 2 and Rule 3)

**Inference**

The three rules are mutually exclusive and one of them at most will succeed

As soon as one of them succeeds there is no point in trying to use the others as they are bound to fail

‘cut’ is used in this case

f(X,0) :- X < 3,!. % Rule 1

f(X,2) :- 3 =< X, X<6,!. % Rule 2

f(X,4) :- 6 =< X. % Rule 1

**Question:**

?- f(1,Y), 2< Y.

Prolog choose rule 1 since 1<3 and fails the goal 2<Y. Prolog will try to backtrack, but not beyond the point marked ! in the program. Rule 2 and Rule 3 will not be generated.

**Negation as Failure**

“Mary likes all animals but snakes”

How can we write this in Prolog?

Mary likes any X if X is an Animal

In prolog,

likes(mary,X) :- animal(X).

if X is n snake then ‘Mary likes X’ is not true,

Otherwise, if X is an animal then Mary likes X.

In prolog,

likes(mary, X) :- snake(X), !, fail.

likes(mary, X) :- animal(X).

**Input/output in prolog :-**

write(term) and read(term):-

* Predicate write(term) causes a term to be written to the current output stream (the monitor screen by default).
* If term is uninstantiated, an underscore followed by a number unique to the variable will be output.
* Predicate read(term) is used to read a term from the current input stream (the keyboard by default).

Input can be redirected using predicates see(filename) to open a file for input, and seen to close file and read from keyboard again. Output can be redirected using predicates tell(filename) to open a file for output, and told to close file and write to the keyboard again.

I/O in Prolog - Example:

position("Kowalczyk",'Teacher').

position('Joy','director').

position("Raj",manager).

position('Hasan','supervisor').

find\_position:-

write('Whose position do you wish to know?'),nl,

read(Input), position(Input, Output),

write('The position of '), write(Input),

write(' is '), write(Output), write(' . ').

**Built in predicates**

**Types of Terms**

First, we will look at predicates that test whether their arguments are terms of a certain type, whether they are, for instance, an atom or a number.

**atom,atomic,var,novar**

**Structure of terms**

Then, we will see predicates that tell us something about the structure of complex terms.

**functor/3**

**arg/3**

?- functor(f(a,b),F,A).

A=2

F=f

yes

?- functor(a,F,A).

A=0

F=a

yes

?- functor([a,b,c],X,Y).

X = ’.’

Y=2

yes

’=..’/2.

It takes a complex term and returns a list that contains the functor as first element and then all

the arguments. So, when asked the query ’=..’(loves(vincent,mia),X) Prolog

will answer X = [loves,vincent,mia]. This predicate is also called univ and can

be used as an infix operator. Here are a couple of examples.

?- cause(vincent,dead(zed)) =.. X.

X = [cause, vincent, dead(zed)]

Yes

?- X =.. [a,b(c),d].

X = a(b(c), d)

Yes

?- footmassage(Y,mia) =.. X.

Y = \_G303

X = [footmassage, \_G303, mia]

Yes

Univ (’=..’) is always useful when something has to be done to all arguments

of a complex term. Since it returns the arguments as a list, normal list processing

strategies can be used to traverse the arguments. As an example, let’s define a predicate called copy\_term which makes a copy of a term replacing variables that occur

in the original term by new variables in the copy. The copy of dead(zed) should

be dead(zed), for instance. And the copy of jeallou(marcellus,X) should be

jeallous(marcellus,\_G235); i.e. the variable X in the original term has been replaces by some new variable.

So, the predicate copy\_term has two arguments. It takes any Prolog term in the first

argument and returns a copy of this Prolog term in the second argument. In case the

input argument is an atom or a number, the copying is simple: the same term should

be returned.

copy\_term(X,X) :- atomic(X).

In case the input term is a variable, the copy should be a new variable.

copy\_term(X,\_) :- var(X).

With these two clauses we have defined how to copy simple terms. What about complex terms? Well, copy\_term should return a complex term with the same functor and

arity and all arguments of this new complex term should be copies of the corresponding arguments in the input term. That means, we have to look at all arguments of the

input term and copy them with recursive calls to copy\_term. Here is the Prolog code

for this third clause:

copy\_term(X,Y) :-

nonvar(X),

functor(X,F,A),

A > 0,

functor(Y,F,A),

X =.. [F|ArgsX],

Y =.. [F|ArgsY],

copy\_terms\_in\_list(ArgsX,ArgsY).

copy\_terms\_in\_list([],[]).

copy\_terms\_in\_list([HIn|TIn],[HOut|TOut]) :-

copy\_term(HIn,Hout),

copy\_terms\_in\_list(TIn,TOut).

So, we first check whether the input term is a complex term: it is not a variable and

its arity is greater than 0. We then request that the copy should have the same functor

and arity. Finally, we have to copy all arguments of the input term. To do so, we use

univ to collect the arguments into a list and then use a simple list processing predicate

copy\_terms\_in\_list to one by one copy the elements of this list.

Here is the whole code for copy\_term:

copy\_term(X,\_) :- var(X).

copy\_term(X,X) :- atomic(X).

copy\_term(X,Y) :-

nonvar(X),

functor(X,F,A),

functor(Y,F,A),

A > 0,

X =.. [F|ArgsX],

Y =.. [F|ArgsY],

copy\_terms\_in\_list(ArgsX,ArgsY).

copy\_terms\_in\_list([],[]).

copy\_terms\_in\_list([HIn|TIn],[HOut|TOut]) :-

copy\_term(HIn,Hout),

copy\_terms\_in\_list(TIn,TOut).