

Simple sentences and their representation in Prolog

Sentences expressing facts can be written as Prolog **clauses**. These should be written in lower case letters. It is also practical to simplify some phrases.

Dick and Ed are friends.

dick_and_ed_are_friends.

Meg is Ed's wife.

meg_is_eds_wife.

Dick has a book.

dick_has_a_book.

Dick has a Mercedes.

dick_has_a_mercedes.

Dick likes his Mercedes.

dick_likes_the_mercedes.

Dick also likes wine.

dick_likes_wine.

Ed likes wine, too.

ed_likes_wine.

And Ed's wife likes wine.

eds_wife_likes_wine.

Prolog can memorize these facts, and when asked about them will confirm, and will deny when asked about any other facts.

?- dick_and_ed_are_friends.
yes

?- dick_and_ted_are_friends.
no
?- are_dick_and_ed_friends.
no

This way of representing knowledge has many drawbacks. For example, if we asked again the first question in a different form: `ed_and_dick_are_friends` then Prolog would not notice the connection between this and the original fact:

```
?- dick_and_ed_are_friends.  
yes  
?- ed_and_dick_are_friends.  
no
```

Therefore it is better to express facts using **structures**:

Database content:

```
friends(dick, ed).  
sameperson(meg, wife(ed)).
```

```
has(dick, book).  
has(dick, mercedes).  
  
likes(dick, mercedes).  
likes(dick, wine).  
likes(ed, wine).  
likes(wife(ed), wine).
```

Question answering:

```
?- friends(dick, ed).  
yes
```

```
?- likes(X, wine).  
X = dick ;  
X = ed ;  
X = wife(ed)  
yes
```

Structures and predicates

Prolog allows to write **structures** in the functional notation. This still leaves much freedom and it is good to use some discipline in writing such facts.

So instead of writing:

```
friend(dick).  
friend(ed).
```

```
have(dick, book).  
have(dick, mercedes).  
have(dick, mariola).
```

```
likes(dick, mercedes).  
likes(ed, wine).  
likes(wife(ed), wine).
```

we should not write this:

```
friend(dick).  
ed(friend).
```

```
dick(have, book).  
mercedes(have, dick).  
have(mariola, dick).
```

```
wife(ed, likes, wine).  
wife(ed, likes(wine)).  
likes(wife(ed(wine))).  
likes(wife, ed, wine).
```

The top level (external) structure is considered by Prolog to be the **relation** symbol **representing some connection between its arguments**.

The internal structures (nested arbitrarily deep) are taken to be **term functions** designating objects which have a relation to other objects.

The relation symbol denotes some logical fact, and is called a **predicate**. When converting facts written in natural language into logical predicates, the predicate (verb) of the sentence is typically used for the logical predicate symbol. The subject of the sentence, along with the an object, or objects, of the sentence.

A logical predicate should have some fixed number of arguments, each with a specific role, although there might me several related predicates with a different number of arguments, as in:

```
has2( who, what )  
has3( who, what, when )  
has4( who, what, when, where )
```

In Prolog it is possible for such predicates to have a common name as Prolog can distinguish them by the number of arguments: **has/2, has/3, has/4**

The variables in Prolog

A term symbol starting with a capital letter (or an underscore `_`) is always a variable in Prolog. **A predicate symbol cannot be variable and cannot start with a capital letter.** A variable occurring in an axiom is taken to be universally quantified, and a variable occurring in a query is considered to be existentially quantified. The scope of all variables is the whole clause where they occur.

Database content:

```
/* some like wine */
likes(ed, wine).
likes(wife(ed), wine).

/* everyone likes beer */
likes(X, beer).

/* dick likes his merc */
likes(dick, mercedes).
```

Question answering:

```
?- likes(X, wine).
X = ed ? ;
X = wife(ed)
yes
?- likes(ed, X).
X = wine ? ;
X = beer
yes
?- likes(X, beer), likes(X, mercedes).
X = dick ? ;
no
```

In the first two queries the variable `X` is a different variable each time. But in the query about the beer and the Mercedes, both `X` variables are the same variable, and must be assigned the same value.

The unification operator

The **unification** operator `=` compares the operands literally. If both are constant, then it returns the logical result of the comparison (either equal or not). If one or both operands are variables then the result is always true, with the side effect of assigning the variable to the constant operand value. If both operands are variables then they remain so, but are unified, which means they must have the same value in the future.

```
?- wife(ed) = meg.           /* no arithmetic */
no                            ?- 2 + 3 = 5.
                               no

?- wife(ed) = X.
X = wife(ed)                  ?- 2 + 3 = X.
yes                            X = 2+3
                               yes

?- wife(X) = wife(Y).
Y = X                          /* dont try */
yes                            ?- father(son(X)) = X.
```

As can be seen in the left-hand side examples, unification is flexible and compares expressions structurally, making variable assignments to match both operands. But no arithmetical evaluation is performed.

Introducing facts and asking questions

The facts written as Prolog clauses can be introduced to the Prolog system, which takes them as **axioms**, places (in order) in its database, and accepts that they are true.

Facts can also be introduced, in the same form terminated with a period, as questions which Prolog should answer. Normally, Prolog is in the question-answering mode. To introduce axioms, the special predicate `consult/1` should be used, which reads facts from the specified file, or standard input: `consult(user)`. Individual facts can also be introduced with the predicates: `asserta` and `assertz`.

Prolog answers questions by searching its database, in the order the facts have been introduced, matching predicates and arguments of the question to the axioms in the database.

Using gprolog

`gprolog` is an easily available Prolog interpreter. It can be instructed on start-up to load into its database all facts from a file.

```
> gprolog --init-goal "consult('dick2.pro')"  
compiling dick2.pro for byte code...  
dick2.pro compiled, 10 lines read - 1004 bytes written, 15 ms  
GNU Prolog 1.2.18  
By Daniel Diaz  
Copyright (C) 1999-2004 Daniel Diaz  
| ?- friends(dick, ed).  
  
(1 ms) yes
```

`gprolog` has a number of extensions to standard Prolog, as well as several configuration variables. For example, in case a question has been asked about a predicate which is not present in the database, `gprolog` generates an error.

A configuration flag can be set so that it would answer “no” as is the standard behavior:

```
| ?- is_cool.  
uncaught exception: error(existence_error(procedure,jest_fajnie/0),top_level/0)  
| ?- set_prolog_flag(unknown, fail).  
  
yes  
| ?- is_cool.  
no
```

Since `set_prolog_flag` is a predicate (like everything in Prolog), it is executed in the question-answering mode. An attempt to introduce it from a file in the `consult` mode would be understood as an attempt to redefine a built-in predicate, which is not allowed.

SWI Prolog

We can also try another useful Prolog interpreter — SWI Prolog:

```
> pl -f dick2.pro
% /home/witold/cla/ai/Prolog/dick2.pro compiled 0.00 sec, 2,800 bytes
Welcome to SWI-Prolog (Version 5.6.6)
Copyright (c) 1990-2005 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.
```

```
For help, use ?- help(Topic). or ?- apropos(Word).
```

```
1 ?- likes(X, wine).
X = dick ;
X = ed ;
X = wife(ed) ;
```

```
No
```

Exercise

Run some Prolog system, introduce some simple facts, with and without arguments, using `consult(user)`. Ask questions to verify the correctness of the defined facts. By pressing the `;` (semicolon) key, force a listing of all the bindings for some variable.

Google "[prolog tutorial](#)" to find some tutorial introductions to Prolog. Copy some simple programs found therein, eg. [hanoi](#), and try to run them using the instructions provided.

For example:

<http://www.learnprolognow.org/>

http://www.cs.bham.ac.uk/~pjh/prolog_course/se207.html

Rules

Besides simple unconditional facts, in Prolog we can use facts including an implication (reversed):

```
likes(dick, X) :- likes(X, wine).
```

These are called **rules**.

Prolog's question-answering process works by matching the question predicate to all the facts in the database, in order. If the fact is a rule, then the question predicate is matched to the head of the rule. If there is a match, Prolog analyzes the right-hand side of the rule, trying to find a positive answer. Any formulas found there Prolog tries to prove by calling itself recursively.

```
likes(dick, X) :- likes(X, wine).  
likes(dick, mercedes).  
likes(dick, wine).  
likes(ed, wine).  
likes(wife(ed), wine).
```

```
?- likes(dick,X).  
X = dick ;  
X = ed ;  
X = wife(ed) ;  
X = mercedes ;  
X = wine.
```

Connectives and relation to logic

The operator `:-` found in rules can be treated as a connective, as it allows building complex clauses from the simple ones. Logically, it corresponds to the implication going backward \Leftarrow . **It can only be used in database rules — not in questions — and then only once.**

Prolog also has other logical connectives: the conjunction \wedge , written with a comma, and logical alternative \vee , written with a semicolon:

```
?- has(dick, mercedes), has(dick, alfa_romeo_8c).
```

```
No
```

```
?- has(dick, mercedes); has(dick, alfa_romeo_8c).
```

```
true
```

The conjunction and alternative can only be used on the right-hand side of the rules. The left-hand side of the rules must be an atomic term. Logical formulas with no alternatives, or alternatives of only negative literals, with at most one positive literal, are called **Horn clauses**.

We can see, that **Prolog works only with Horn clauses.**

Negation, or rather the lack of it

Prolog does not have a negation. It does have a built-in predicate `not`, whose meaning can be described as: “it cannot be proved, that ...”. In some cases it can be used for negation, but in other cases it gives unexpected results.

Database content:

```
man(dick).  
dog(spot).
```

```
?- man(X).  
X = dick ?  
yes  
?- not(man(spot)).  
yes  
?- not(man(X)).  
no
```

One could expect that Prolog would find the individual who is not a man.

Or one could expect that, since it could be proved that `not(man(spot))` then it would also be proved that `not(man(X))`.

Both expectations fail. We could only conclude, that `not` is a strange, non-intuitive form of a negation.

The negation predicate `not` can only be used in questions, and not in stored facts.

An explanation why this is so will come later, and for now we must accept the positive thinking mode and the ban on negating anything.

Let us note, however, that Prolog itself has negative thinking built-in. It denies anything that is not found to be true. This feature — of denying anything that is not clearly stated, or is a consequence thereof — is called the **Closed-World Assumption, CWA**.

Many Prolog interpreters do not have the `not` predicate as such. Instead they offer the `\+` operator, with the same meaning.

Computing with structures

The Prolog structure, which is the notation of a predicate symbol with arguments, can be treated as a data structure, and run computations on it.

Consider the following arithmetic. We introduce zero as a symbol `zero`, and the number `X`'s successor as `s(X)`. This is called the **Peano arithmetic**. For example, the number 5 is written as: `s(s(s(s(s(zero)))))`.

We want to define addition with the `sum(S1, S2, S3)` predicate, which would be true if, and only if, the `S3` argument was the sum of the first two arguments:

```
sum(zero, S1, S1).  
sum(s(X), S1, s(S2)) :- sum(X, S1, S2).
```

Now we can conduct computations in this arithmetic, eg. to compute `3+4`:

```
?- sum(s(s(s(zero))), s(s(s(s(zero))))), X).  
X = s(s(s(s(s(s(s(zero)))))))
```

It is also possible to define multiplication, try it!

“Backward” computations

Note that the first two arguments of `sum` are meant to be (input) data, while the third argument is the result of the computation. It is analogous to some other programming languages, which permit functions to have the “out” arguments to return results in them. Like in those other languages such “function” is not a pure mathematical function, such a predicate in Prolog is not a pure logical predicate.

Prolog requires no declarations for which arguments are “out”. What would the happen if, instead of asking questions such as $3+4=?$, we tried to give equations to solve $3+?=4$:

```
?- sum(s(s(s(zero))), X, s(s(s(s(zero))))).  
X = s(zero) ;  
No
```

It works, the only solution of the above equation is `s(zero)`, and nothing else. This ability to run “backward” computations is a side effect of Prolog’s database searching and pattern matching algorithm.

Why not press further, and ask questions with no definite answer, such as:
 $?+?=4$? We would get all distributions of the number 4 into the components:

```
?- sum(X, Y, s(s(s(s(zero))))).
```

```
X = zero
```

```
Y = s(s(s(s(zero)))) ;
```

```
X = s(zero)
```

```
Y = s(s(s(zero))) ;
```

```
X = s(s(zero))
```

```
Y = s(s(zero)) ;
```

```
X = s(s(s(zero)))
```

```
Y = s(zero) ;
```

```
X = s(s(s(s(zero))))
```

```
Y = zero ;
```

```
No
```

If we had defined multiplication we would be able to obtain factorizations, or even compute square roots! Give it a try!

Real numbers

Prolog can use real numbers, compare them, and evaluate numerical expressions, although the latter it does reluctantly. We can try this using the unification operator `=` first.

```
?- 0 = 0.
```

```
Yes
```

```
?- 0 = 1.
```

```
No
```

```
?- 2+2 = 4.
```

```
No
```

```
?- 2+2 = X.
```

```
X = 2+2
```

```
Yes
```

Prolog thinks that its main duty is searching the database, matching predicates and arguments, and proving theorems, while numerical computation is outside its scope. The keyword to force it compute is `is`. It computes the right argument, and compares (or unifies) with the value of the left argument.

```
is_equal2(X, Y) :- X1 is X, X1 = Y.
```

Now the results are correct: but some are still unacceptable, or give errors:

```
?- is_equal2(2+2, 4).  
Yes
```

```
?- is_equal2(2+2,2+2).  
No
```

```
?- is_equal2(2+2, X).  
X = 4
```

```
?- is_equal2(X, 2+2).  
ERROR: (user://4:136):  
is/2: Arguments are not sufficiently instantiated
```

Equality operators in Prolog

Equality, or equivalence, has many facets in Prolog. In addition to the unification operator `=` which performs a structural comparison with variable unification, there are the numerical comparisons, where the terms are evaluated numerically. For this they must be fully instantiated (contain no variables), and evaluate to just a number:

`X is Y` — the right-hand side expression `Y` is evaluated arithmetically, and its value matches the left-hand side expression `X`, which can be a variable

`X := Y` — the arithmetic values of expressions `X` and `Y` are equal

`X \= Y` — the arithmetic values of expressions `X` and `Y` are not equal

Additionally there are the structural comparisons, which do not evaluate anything numerically (or otherwise), but check the complete, literal equality:

`X == Y` — the terms `X` and `Y` have literal equality, identical structure and identical arguments, including the variable names, eg. `X==Y` is always false

`X \== Y` — the terms `X` and `Y` are not literally identical

```

?- 3+4 = 4+3.
no % structures differ
?- 3+4 = 3+4.
yes
?- X = 3+4.
X = 3+4
yes
?- 3+X = 3+4.
X = 4
yes

?- 3+4 == 4+3.
no
?- 3+X == 3+4.
no
?- +(3,X) == 3+X.
yes

?- 3+4 \== 4+3.
yes

```

```

?- X is 3+4.
X = 7
yes
?- X = 7, X is 3+4.
X = 7
yes
?- X is 3+4, X = 7.
X = 7
yes
?- 3+4 is 4+3.
no % left arg.must be unassigned var.
    % or evaluate to a number

?- 3+4 ::= 4+3.
yes % calculates both values
?- X ::= 3+4.
error % both args must have values
?- a ::= 3+4.
error % and they must be arithm.values

?- 3+4 =\= 4+3.
no

```


Lists

Prolog has one real data structure which is the **list**. A list is a sequence of elements, which can be atoms or lists, and is written in square brackets, separated with commas, for example:

```
[a]
[X,Y]
[1,2,3,4]
[a,[1,X],[],[],a,[a]]    /* this list has 6 elements */
```

There is an alternative notation for lists as “head” (first element) and “tail” (the list of the remaining elements). It is mostly useful when the tail is written with a variable, for example:

```
[a|R]                /* this list has at least 1 element, R could be [] */
[1|[2|[3|[4|[]]]]]   /* exactly equal to the list [1,2,3,4] */
[1|[2|[3|[4]]]]      /* another way of writing the list [1,2,3,4] */
[1,2|[3,4]]          /* this is also allowed and is the same [1,2,3,4] */
```

The list in notation `[Head|Tail]` can also be written as the structure `.(Head,Tail)` (the name of the term is the period).

Although most contemporary Prolog implementations provide many operations on lists (the example predicates given below are called `member` and `append`, respectively) it is useful to study implementations of some basic operations.

The `element` predicate checks whether something is an element of some list:

```
element(X, [X|_]).  
element(X, [_|Y]) :- element(X,Y).
```

This predicate merges two lists together and unifies with the third argument.

```
merge([],X,X).  
merge([X|Y],Z,[X|Q]) :-  
    merge(Y,Z,Q).
```

Try these queries:

```
?- merge([a,b,c],[w,x,y,z],L).  
?- merge([a,b],Y,[a,b,c,d]).  
?- merge(Y,[c,d],[a,b,c,d]).  
?- merge([b,c],Y,[a,b,c,d]).  
?- merge(X,Y,[a,b,c,d]).  
?- merge(X,Y,Z).
```

Try this yourself: implement a predicate computing the last element of a list.

Program debugging

Prolog has a number of mechanisms useful for analyzing and debugging programs:

`spy/1` — starts tracing all calls to the predicate given as the argument (which can be specified in form: `pred/n` which designates the version with the given number of arguments),

`trace/0` — starts tracing all calls,

`nospy/1`, `notrace/0` — cancels all tracing,

`nodebug/0` — cancels all `spy`,

`debugging/0` — displays all `spy`,

`listing/1` — displays the all facts in the database (both simple facts and rules) concerning the predicate given as the argument,

`listing/0` — displays the full listing of the database.

Exercise — permutations

Effective use of lists is essential in many programs. A good exercise is to write a program for computing permutations of a list by means of a `permute(X,Y)` which checks if one of its arguments is a permutation of the other. Two lists are permutations of each other if they contain the same elements, in the same quantities (if repeated), but possibly in a different order.

```
?- permute([a,b,c],[b,c,a]).
```

Yes

```
?- permute([a,a,c],[c,c,a]).
```

No

Try to write such a predicate. After verifying its operations on fully instantiated lists, try running it with a variable as one of the arguments, to see if it will generate all permutations of the list given.

```
?- permute([a,b,c],X).
```


A useful scheme: Generate and Test

Many programs can be written in Prolog according to a certain scheme. Consider first a simple generator producing subsequent natural numbers, as long as some other predicate will continually requests these numbers:

```
nat_num(0).  
nat_num(N) :- nat_num(M), N is M + 1.
```

In order to demonstrate the use of the generator, which is to repeatedly resuming the computation of this predicate, we can use the zero-argument `fail` predicate, which always returns the value `false`:

```
nat_num(N), write(N), nl, fail.
```

Many programs can be built by generating objects potentially useful as a solution to the problem, and a testing predicate, which only checks if an object is an acceptable solution:

```
generate(X), test(X), ready(X).
```

For example, to generate all prime numbers it is sufficient to write a predicate checking divisibility, or rather the lack thereof:

```
% The sieve of Eratosthenes, from Clocksin & Mellish (pri2)
%      finding the prime numbers up to 98.

main :- primes(98, X), write(X), nl.

primes(Limit, Ps) :- integers(2, Limit, Is), sift(Is, Ps).

integers(Low, High, [Low | Rest]) :-
    Low =< High, !,
    M is Low+1,
    integers(M, High, Rest).
integers(_,_,[]).

sift([],[]).
sift([I | Is], [I | Ps]) :- remove(I,Is,New), sift(New, Ps).

remove(P,[],[]).
remove(P,[I | Is], Nis) :- 0 is I mod P, !, remove(P,Is,Nis).
remove(P,[I | Is], [I | Nis]) :- not(0 is I mod P), !, remove(P,Is,Nis).
```


Suspending and resuming computation

`http://www.inf.ed.ac.uk/teaching/courses/aipp/`

`http://www.inf.ed.ac.uk/teaching/courses/aipp/lecture_slides/07_Cut.pdf`

The Cut

The **cut** operator, written with the bang symbol **!**, is a zero-argument predicate with the logical meaning of truth, which blocks the backtracking mechanism of Prolog, making it impossible to return to the choice points preceding the execution of the cut. Consider the following examples:

We have the axioms:

```
fact(a).  
fact(b) :- !.  
fact(c).
```

```
?- fact(a).
```

```
Yes
```

```
?- fact(b).
```

```
Yes
```

```
?- fact(c).
```

```
Yes
```

Each of the facts **a,b,c**, is individually satisfied. When Prolog must restart the proof of **fact** many times, then after encountering the cut, it cannot continue restarting the search, and gives the negative answer.

```
?- fact(X).
```

```
X = a ;
```

```
X = b ;
```

```
No
```

As we can see below, the presence of the cut in the definition of `fact` makes it difficult calling it from other predicates.

```
fact2(X,Y) :- fact(X), X = Y.
```

```
?- fact2(X,a).
```

```
X = a
```

```
Yes
```

```
?- fact2(X,b).
```

```
X = b
```

```
Yes
```

```
?- fact2(X,c).
```

```
No
```

Switching around the tests in the conjunction postpones the execution of the cut, which helps some, but some existing choices are still cut off.

```
fact3(X,Y) :- X = Y, fact(X).
```

```
?- fact3(X,c).
```

```
X = c
```

```
Yes
```

```
?- fact3(X,Y).
```

```
X = a
```

```
Y = a ;
```

```
X = b
```

```
Y = b ;
```

```
No
```

So why do we really need to cut operator?

The Cut — case 1: confirms the correct rule choice

Imagine the `sum_to/2` predicate for computing the sum of the numbers from 1 to some value. The second argument will hold the result.

```
sum_to( 1, 1 ).  
sum_to( N, R ) :-  
    N1 is N - 1,  
    sum_to( N1, R1 ),  
    R is R1 + N.
```

This solution works fine, except for some special cases, like when the user has invoked it with the wrong parameters, or when she has typed „;” forcing the program to resume the computation:

```
?- sum_to(5,X).  
X = 15 ;  
ERROR: (user://1:22):  
    Out of local stack
```

```
?- sum_to(5,14).  
ERROR: (user://1:27):  
    Out of local stack
```

This program is an example of a case which should never resume the computation after having obtained one answer. After all, there is only one sum of numbers from 1 to N. This can be implemented with the cut (left-hand side):

```
sum_to( 1, 1 ) :- !.  
sum_to( N, R ) :-  
    N1 is N - 1,  
    sum_to( N1, R1 ),  
    R is R1 + N.  
  
sum_to( N, 1 ) :- N < 1, !, fail.  
sum_to( 1, 1 ).  
sum_to( N, R ) :-  
    N1 is N - 1,  
    sum_to( N1, R1 ),  
    R is R1 + N.
```

One erroneous case that still is not handled correctly, is when the first argument is negative. This can be fixed with the version on the right-hand side above.

Aside from the cut, there exist a simple solution to all of the problems mentioned, without the cut:

```
sum_to( 1, 1 ).  
sum_to( N, R ) :-  
    N > 1,  
    N1 is N - 1,  
    sum_to( N1, R1 ),  
    R is R1 + N.
```

The Cut — case 2: confirms the falsehood of the target

The Cut — case 3: cuts off unneeded options

The Cut — problems

Prolog — database operations

Normally Prolog is in question answering mode. New facts can be added to the database by reading a file with the `consult` predicate. (Reading facts from the terminal can be invoked by `consult('user').`) Additionally, new facts can be build dynamically and added or removed from the database.

`asserta(term)`, `assertz(term)` — adds the fact `term` to the database, at the beginning or at the end, respectively

`retract(term)` — removes the fact `term` from the database, if present

Note: during Prolog backtracking the effects of these operations are not retracted, ie. the previous database state is not recovered!

Prolog — operations on terms

```
var(term)      /* true if term is an unbound variable */

nonvar(term)   /* inverse of var */

atom(term)     /* true if term is bound to a literal atom (not string) */

integer(term)

atomic(term)   /* atom or integer */

clause        /* C&M(4)p.115 */

functor        /* C&M(2)p.120(4)p.117 */

arg           /* C&M(2)p.122(4)p.119 */

=..          /* C&M(2)p.173,123(4)p.120 */
```

Prolog — input/output operations

Reading and writing terms:

```
?- read(X).      /* reads from terminal a single term terminated
                  by a period '.' and binds X to it;
                  at the end of file returns end_of_file */
?- write(X).     /* writes to terminal the term currently
                  bound to X                               */
?- nl.           /* writes a newline to terminal           */
```

Reading and writing characters:

```
?- get(X)        /* reads a character as a numerical code */
?- put(X)        /* writes a char., eg. put(104) writes 'h' */
```

Operations on files:

```
?- tell('nowy')./* opens a new file named 'nowy' and switches
                  standard output to that file;
                  subsequent output go to that file      */
?- told.        /* closes file currently open and switches
                  standard output to terminal              */
```

```
?- see('stary').      /* opens existing file for reading */
?- seen.              /* ends reading from and closes file */
```

Reading whole files as axiom definitions:

```
?- consult(file1).    /* or shorthand: [file1]. */
?- reconsult(file2). /* or shorthand: [-file2]. */
```

Example: writing all axioms defining the predicates 'has' i likes' to a file:

```
?- tell('program'), listing(has), listing(likes), told.
    /* predicate 'listing' writes out all stored clauses
       eg.: listing, listing(has), listing(has/2)      */
```