6 Built-in functors and predicates

We now discuss a selection of useful predefined functors and predicates available in most Prolog implementations.

6.1 Equality

\[ r = s \]  succeeds iff the terms \( r \) and \( s \) are unifiable
in which case an mgu of \( s \) and \( t \) is computed;
the predicate “=” could be defined by the unit clause
\[ X = X. \]

\[ r \neq s \]  tests if the terms \( r \) and \( s \) are non-unifiable

\[ r == s \]  tests if the terms \( r \) and \( s \) are syntactically identical;

\[ r \neq= s \]  tests if the terms \( r \) and \( s \) are not syntactically identical.

Examples.

?- [a,b,b] = [b,b,b].

?- [a,b,b] \= [b,b,b].

?- [a\L] = [X,b,b].

?- [a\L] \= [X,b,b].

?- [a\L] == [X,b,b].

?- [a\L] \== [X,b,b].
Exercise. Using the predicate “==”, complete the following
to a program for computing the concatenation of lists:

app(K, L, M) :-
app(K, L, M) :-

Remarks. 1. Although the predicate = does not increase the power of Prolog,
it can be very useful to make programs shorter and more readable.

2. The predicate \( \neq \) should normally be used only when it will be called with
closed terms. In this case its meaning is just inequality, that is, the negation of =. Compare, for example, Prolog’s answers to the following queries:

?- L = [a,b,c], member(X, L), X \( \neq \) b.

?- L = [a,b,c], X \( \neq \) b, member(X,L).

In the first query the goal X \( \neq \) b will only be called when X has been substituted
by one of the values a, b, c. Consequently we get the expected answers, namely
all members of L different from b. In the second query, however, the goal X \( \neq \) b
(un-intendedly) fails because X and b are unifiable.

3. The predicates == and \( \neq \) should normally not be used. Their semantics
obscures the intended meaning of variables.
6.2 Arithmetic

1. Arithmetic operations are special functors:

   + addition
   - subtraction
   * multiplication
   / division, (the result is a floating point number),
   // integer division
   mod remainder of integer division
   ^ exponentiation

   All these operations are binary (take two arguments) and are written in infix notation.

2. Arithmetic terms are built from variables, integers (..., −2, −1, 0, 1, 2, ...) and real numbers (to be entered in decimal notation) using arithmetic operations. Every closed arithmetic term has a canonical value.

   Examples. \( x + (5 \times y) \), 7 and \( 3 \times (7 - 2) \) are arithmetic terms. The second and third term are closed, their values are 7 and 15, respectively.

   \([3, 5]\) is not an arithmetic term.

3. Assignment.

   A goal

   \( s \text{ is } t \)

   succeeds iff \( t \) is a closed arithmetic term and \( s \) is unifiable with the value of \( t \). If \( t \) is not a closed arithmetic term an error is returned.

   If the goal \( s \text{ is } t \) succeeds there are two possible cases for the term \( s \):

   (i) \( s \) is a variable, say \( X \), in which case \( X \) is substituted by the value of \( t \);

   (ii) \( s \) is identical with the value of \( t \).
Examples.

?- X is 4+3.

X = 7 ;
No

?- 8 is 4+3.
No

?- 8 is 4+X.
ERROR: Arguments are not sufficiently instantiated

?- X is 4+3, Y is X*X, L = [X,Y].

X = 7
Y = 49
L = [7, 49] ;
No

?- Y is X*X, X is 4+3, L = [X,Y].
ERROR: Arguments are not sufficiently instantiated

What are Prolog’s answers to the following queries?

?- X = 4+3, Y = X*X, L = [X,Y].

?- X == 4+3, Y == X*X, L == [X,Y].

The length of a list (as an ‘ordinary’ integer):

\[\text{len([], 0).} \]
\[\text{len([_|L], N) :- len(L, M), N is M+1.} \]

?- len([a,b], N).
\[ N = 2; \]
\[ \text{No} \]

What will be Prolog’s answers to
\[ ?- \text{len}(L, N). \]
\[ ?- \text{len}(L, 3). \]

**Exercise.** Write a program for two predicates `numInt` and `intNum` that translate numerals \( 0, s(0), \ldots \) into nonnegative integers and vice versa.

4. **Predicates comparing the values of two closed arithmetic terms.**

\[
< \quad \text{less} \\
\leq < \quad \text{less or equal} \\
> \quad \text{greater} \\
\geq \quad \text{greater or equal,} \\
=:= \quad \text{equal} \\
=:\neq \quad \text{different}
\]

It is an error to call these predicates with terms other than closed arithmetic terms.

**Examples.**

\[ ?- 4 + 5 =:= 3 * 3. \]

Yes

\[ ?- 4 + 5 = 3 * 3. \]

No

\[ ?- X =:= 7. \]

**ERROR:** Arguments are not sufficiently instantiated
?- 3 =\= 3*1.

No

**Examples.** Testing whether a list of integers is sorted, that is, of the form \([n_1, \ldots, n_k]\), where \(n_1 \leq \ldots \leq n_k\).

\[
\text{sorted([],).} \\
\text{sorted([_],).} \\
\text{sorted([X,Y|L]) :- X =\= Y, sorted([Y|L]).}
\]

**Sorting:** A list \(L_1\) is the sorted version of a list \(L\) if \(L_1\) is a permutation of \(L\) and \(L\) is sorted. This specification gives rise to a purely declarative sorting program called **permutation sort**.

\[
\text{permutation([],[]).} \\
\text{permutation(L,[X|K1]) :- remove(X,L,K), permutation(K,K1).}
\]

\[
\text{permutationSort(L,L1) :- permutation(L,L1), sorted(L1).}
\]

This program is extremely slow (why?). It can be useful though to test, for example, the efficiency of a Prolog compiler (use this program to sort a list with 10 elements). A more efficient sorting algorithm is

**Insertion sort:** First we need a program \(\text{insert}(X,L,L_1)\) for inserting a number \(X\) into a sorted list \(L\) at the ‘right’ place, that is, such that the result \(L_1\) is again sorted.

\[
\text{insert(X,[],[X]).} \\
\text{insert(X,[Y|L],[X,Y|L]) :- X =\< Y.} \\
\text{insert(X,[Y|L],[Y|L1]) :- X > Y, insert(X,L,L1).}
\]

Now **insertion_sort/2** can be defined as follows:

\[
\text{insertionSort([],[]).} \\
\text{insertionSort([X|L],L2) :- insertionSort(L,L1), insert(X,L1,L2).}
\]
6.3 Input and output

1. Reading and writing.

\texttt{read(t)} reads the next term (which must be terminated by a ".") from the current input stream and unifies it with \texttt{t}

\texttt{write(t)} writes the term \texttt{t} to the current output stream.

Example.

cube :-
    write('Next item please: '),
    read(X),
    process(X).

process(N) :-
    integer(N),
    C is N*N*N,
    write('Cube of '),
    write(N),
    write(' is '),
    write(C),
    nl,
    cube.

process(stop) :- write('Bye').

Here we used the built-in predicate \texttt{integer(N)} which tests whether \texttt{N} is an integer.

With \texttt{nl} we prompt Prolog to begin a new line in the output stream.

A conversation with the program \texttt{cube} would be, for example,

?- cube.
Next item please: 5.
Cube of 5 is 125
Next item please: 12.
Cube of 12 is 1728
Next item please: stop.
Bye
**Example.** Writing a list without brackets and commas.

```prolog
write_list([]).
write_list([X|L]) :-
    write('['),
    write(X),
    write_list(L).
```

**Exercise.** Write a program `write_list_of_lists` such that, for example, the query

?- write_list_of_lists([[a,b],[c],[d,e,f]]).

makes Prolog write

```
 a b
c
d e f
```

**Example.** Writing all permutations of a given list.

```prolog
write_permutations(L) :- permutation(L, P), write_list(P), nl, fail.
write_permutations()._.
```

This program uses a failure driven loop to create all permutations of a list. Because the last goal in the first clause always fails Prolog is forced to backtrack and tries all possible solutions P of the goal `permutation(L, P)`. The second clause has the effect that the program terminates with success (this can be useful as we will see later).

2. Communication with files.

- `see(filename)` switches current input stream to `filename`
- `seen` closes current input stream and switches back to the default input stream
- `tell(filename)` switches current output stream to `filename`
- `told` closes current output stream and switches back to the default output stream

**Example.** Create all permutations of a list and store them in a specified file.
write_perms_to_file(L,Filename) :-
   tell(Filename),
   write('The list '),
   write_list(L),
   write(' and its permutations:'), nl, nl,
   write_permutations(L),
   told.

After executing the query

?- write_perms_to_file([a,b,c], 'test.txt').

the content of the file test.txt will be

The list abc and its permutations:

abc
acb
bac
bca
cab
cba

6.4 Database manipulation

assert(Clause) adds Clause to the database (program);
retract(Clause) deletes Clause from the database.

The Clause is to be written without full-stop, and if Clause is a rule, it has to be
enclosed in parentheses.

Example. Suppose we have the program

fast(ann).
slow(tom).
slow(pat).

We can add a rule to this program as follows:
?- assert(
    (faster(X,Y) :- fast(X), slow(Y))
).

Yes

?- faster(A,B).
A = ann
B = tom;
A = ann
B = pat;

no

Now we would like to delete all facts of the form slow(t). In order to do this we first have to make the static predicate slow/1 dynamic:

?- dynamic(slow/1).

Yes

?- retract(slow(X)).
X = tom ;
X = pat ;

No

?- faster(ann,_).

No

When using assert and retract it is useful to control which clauses of a predicate are currently in force:

?- listing(fast/1).
fast(ann).

Yes
?- listing(slow/1).
:- dynamic slow/1.

Yes
?- listing(faster/2).
:- dynamic faster/2.
faster(A, B) :-
    fast(A),
    slow(B).

Yes

There are variants of assert, called asserta respectively assertz that assert clauses at the beginning respectively at the end of a program.

6.5 Declaring operators

The readability of programs can often be improved by using the operator notation. Operators can be infix, prefix or postfix.

We can make, for example, the following operator declaration:

?- op(300, xfx, was).
?- op(250, xfx, of).
?- op(200, fx, the).

This declares
was as an infix operator with priority 300,
of as an infix operator with priority 250,
the as a prefix operator with priority 200.

In general priorities have to be numbers between 0 and 1200. Only the relative sizes of priorities are important.

Now we can write, for example, the clause

diana was the secretary of the department.

We can ask the following queries:
?- Who was the secretary of the department.

Who = diana ;

No
?- diana was the secretary of What.

What = the department ;

No

The program and the queries are identical to the following, written in familiar, but less readable notation:

was(diana, of(the(secretary), the(department))).

?- was(Wo, of(the(secretary), the(department))).

Who = diana ;

No
?- was(diana, of(the(secretary), What)).

What = the(department) ;

No

Exercise. What would be Prolog’s answer to the following query:

?- diana was What.

Write this query and its answer in the traditional notation.

6.6 Collection of data

Prolog provides several built-in predicates for collecting all objects satisfying a given goal. Consider, for example the following data base:

age(peter,7).
age(ann,5).
age(pat,8).
age(tom,5).
If we want to know all children of age 5 we can ask the query

?- age(Child, 5).

Child = ann;
Child = tom;
No

This yields the children of age 5 one by one. If, however, we wish to collect all
children of age 5 into a list we can use the query

?- findall(Child, age(Child, 5), L).

L = [ann, tom];
No

In general, in a call of findall(X, P, L) all variables in P different from X are
being treated as existentially quantified. For example, if we replace the age 5 by a
variable we obtain the list of all children:

?- findall(Child, age(Child, Age), L).

L = [tom, pat, ann, peter];
No

We may also ask for the list of all children having an age < 8:

?- findall(Child, (age(Child, Age), Age < 8), L).

L = [tom, ann, peter];
No

The order of the solutions in the list L is the order in which they are found by
Prolog. It may happen that solutions are repeated.

Remark. The second argument of findall is not a term, but a query, that is,
a list of atomic formulas. Therefore, findall is not an ordinary predicate, but,
what is called in logic programming jargon, a meta-predicate. We will learn
more about meta-predicates later in the course.
**Exercise.** Using `findall` write a program `powerset/2` that computes the powerset of a finite set of natural numbers, where a finite set of natural numbers shall be represented by a list without repetitions. The powerset of a set \( S \) shall be represented as a list of all the subsets of \( S \).

There is a variant of `findall`, called `bagof` which, in a call `bagof(X,P,L)`, does *not* consider the variables in \( P \) different from \( X \) as existentially quantified. Instead, it finds instances for these variables such that solutions \( X \) can be found. For example:

?- bagof(Child,age(Child,Age),L).

Age = 7
L = [peter] ;

Age = 8
L = [pat] ;

Age = 5
L = [ann, tom];
No

If one uses bagof, one can force existential quantification of a variable Y by writing
bagof(X, Y ^ P, L). For example:

?- bagof(Child, Age ^ age(Child, Age), L).
L = [peter, ann, pat, tom];
No

Another variant is setof, which works like bagof, but lists solutions without
repetitions and ordered according to the built-in predicate @<. For ordinary words,
@< coincides with the alphabetic order. For example:

?- setof(Child, Age ^ age(Child, Age), L).
L = [ann, pat, peter, tom]
No

6.7 Disjunction

Sometimes it is convenient to join two clauses with the same head by using the
built in logical operator “;” for disjunction (“or”). For example, the clauses

connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).

can be replaced by

connection(X, Y) :- direct(X, Y); (direct(X, Z), connection(Z, Y)).

The logical principle behind this is the fact that

\((A_1 \rightarrow B) \land (A_2 \rightarrow B)\) is equivalent to \((A_1 \lor A_2) \rightarrow B\)
6.8 Exercises

1. Define a predicate sumTwoSquares/3 such that if \( n, m \) are numbers, then 
\[ \text{sumTwoSquares}(n, m, X) \] 
will compute the number \( X = n^2 + m^2 \).

2. Define a predicate sumList/2 that computes for any list of numbers the sum of its members.

3. Define a predicate sumSquares/2 that computes for any list of numbers the sum of the squares of its members.

4. Define a predicate nat_init/2 that creates for any number \( n \) the list \([0, \ldots, n]\)

5. Write a program that computes the Fibonacci numbers

\[ 1, 1, 2, 3, 5, 8, 13, \ldots \]

That is,

\[ f_1 = f_2 = 1, \]
\[ f_n = f_{n-1} + f_{n-2} \quad (n > 2) \]

6. The program of question 5 will probably quite slow (try it with \( N = 20 \)). Write a faster program using the following idea: Write an auxiliary program fib_pair/3 that computes for given number \( n \) the Fibonacci numbers \( f_n \) and \( f_{n+1} \). Think how one can get the \( n \)-th pair from the \( n - 1 \)-st pair, for \( n > 1 \).

7. Define a predicate write_sums_of_sublists/1 that writes for a given list \( L \) of numbers all its sublists and their sums on the screen.

8. Implement Quicksort. The idea is as follows: Given a nonempty list \([X|L]\) of numbers, split the tail, \( L \), into two parts, Low and High, such that Low contains the members of \( L \) that are less than \( X \) and High those that are greater than or equal to \( X \). Recursively sort Low into Low_sorted and High into High_sorted. The sorted version of \([X|L]\) is then given by the concatenation of Low_sorted and [X|High_sorted].

9. Write a program allConnections/2 that computes for every city the list of all reachable cities.

10. Write a variant of program allConnections/2 that writes the solution into a specified file.