PROLOG UTILITY MODULE

The Prolog utility module is a set of useful Prolog predicates contained in the file "util" on the root directory of the Prolog diskette. To use the utility module, it must be loaded into memory. If the Prolog diskette is in drive /d0, the goal:

```
P>IN("/d0/util");
```

will load the utility module into memory. As well as providing several useful predicates, the utility module maybe helpful as an example to Prolog programmers. The source code in the utility module may be modified and combined with your own programs without restriction.

Brief summary

NOT(<goal>) succeeds only if there exists no solution to <goal>.

AND(<goal1>,<goal2>) succeeds only if <goal1> and <goal2> have solutions.

OR(<goal1>,<goal2>) succeeds if <goal1> or <goal2> have a solution.

FORALL(<goal1>,<goal2>) succeeds only if for every solution of <goal1> there exists a solution to <goal2>.

ISALL(<var>,<term>,<goal>) succeeds by binding <var> to a list of every instance of <term> for which there is a solution to <goal>.

APPEND(<list1>,<list2>,<list3>) succeeds if the concatenation of <list1> and <list2> is <list3>.

LIST(<predicate name>) displays the clauses for <predicate name>.

REVERSE(<list1>,<list2>) succeeds if <list2> is a list of the elements of <list1> in reverse order.

LISTALL() lists all the clauses for all the predicates in the database.

SAVE(<filename>) saves the current database to <filename>.

ISIN(<term>,<list>) succeeds if <term> is an element of <list>.

KILLALL() deletes the entire database.

NOT(<goal>)

If <goal> has a solution, NOT() fails, otherwise it succeeds. To understand how NOT() works, it is necessary to study its implementation:

```
NOT(?p.?argp):-
      ?p.?argp,
      /,
      FAIL();

NOT(?p.?argp):-;
```

First the goal argument is evaluated. Should it fail, control passes to the second clause, which immediately succeeds. Should it succeed, a cut is immediately used to suppress unwanted backtracking. The predefined rule FAIL() is then used to fail the whole predicate without alternatives. As a simple example:

```
P>red(my-car):-;

P>NOT(red(my-car));

P>NOT(red(your-car));
<>

P>NOT(red(?x));
```

Note that NOT() assumes that if it cannot prove something then it must be false. This is called the closed world assumption, and leads to a useful and workable definition of negation, however there are some problems, eg NOT(NOT(<goal>)) is definitely not the same as <goal>.

AND(<goal1>,<goal2>)

If <goal1> has a solution, and <goal2> has a solution, then AND() succeeds. The implementation of AND() is as follows:

```
AND(?p.?argp,?q.?argq):-
      ?p.?argp,
      ?q.?argq;
```

The implementation is simple enough, but AND() is useful in metaprogramming for combining multiple goals into one metaquery, for example:

```
NOT(AND(make(my-car,Ford),colour(my-car,blue)))
```

OR(<goal1>,<goal2>)

The operation of OR() is most easily seen from its definition:

```
OR(?p.?argp,?q.?argq):-
      ?p.?argp;
```

```
OR(?p.?argp,?q.?argq):-
      ?q.?argq;
```

First, <goal1> is evaluated (first clause). If <goal1> succeeds, OR() succeeds (but the second clause remains available for backtracking). If <goal1> fails, control passes to the second clause and <goal2> is evaluated. If <goal2> succeeds, OR() succeeds, otherwise OR() fails. Like AND(), OR() is useful for defining combination metaqueries.

FORALL(<goal1>,<goal2>)

FORALL() is a high-level metaprogram which checks that for every solution to <goal1> there is a solution to <goal2>. FORALL() can often be used to avoid explicit looping. For example:

```
P>small(rat):-;
P>small(cat):-;
P>small(mouse):-;
P>furry(rat):-;
P>furry(cat):-;
P>furry(mouse):-;
P>FORALL(small(?x),furry(?x));
<>
```

ISALL(<variable>,<term>,<goal>)

ISALL() is another high-level metaprogram. ISALL() constructs a list of all the terms <term> for which <goal> has a solution, then binds <variable> to this list. The list is terminated by nil, and the elements of the list occur in the reverse order to the order in which they were found. For example:

```
P>p(a,b,c):-;
P>p(a,c,d):-;
P>p(c,e,a):-;
P>p(e,d,d):-;
P>ISALL(?x,?y.?z,p(?y,?t,?z));
<?t=?t,?z=?z,?y=?y,?x=("e"."d").("c"."a").("a"."d").("a"."c").nil>
```

Note that ISALL() leaves ?y, ?z, ?t all unbound after the query.

APPEND(<list1>,<list2>,<list3>)

APPEND() succeeds if, when <list2> is appended to <list1>, the result is <list3>. Because of its implementation, APPEND() can be used in many ways. For example:

```
P>APPEND(the.cat.sat.nil,on.the.mat.nil,?z);
<?z="the"."cat"."sat"."on"."the"."mat".nil>

P>APPEND(?x,the.?y,the.cat.sat.on.the.mat.nil);
<?y="cat"."sat"."on"."the"."mat".nil,?x=nil>
```

```
<?y="mat".nil,?x="the"."cat"."sat"."on"."the".nil>
```

LIST(<predicate name>>)

LIST() retrieves all the clauses for <predicate name> and formats them to the output stream. The format is such that if the output stream has been directed to a disk file, Prolog can read the predicate back in again.

REVERSE(<list1>,<list2>)

REVERSE() takes the elements of <list2> and reverses their order to form <list2>. For example:

```
P>REVERSE(jack.and.jill.nil,?x);
<?x="jill"."and"."jack".nil>
```

LISTALL()

LISTALL() lists all the predicates in the database in alphabetical order. The output format is the same as that used by LIST(). LISTALL() excludes from the list all those predicates defined by the utility module itself.

SAVE(<file>)

SAVE() writes all the predicates in the database to <file>. <file> would normally be a disk file, but a device could equally well be specified, for example, the query:

```
P>SAVE("/p");
```

could be used to list the database to the printer /p. Files saved using SAVE() can be loaded back again using IN(). As with, LISTALL(), predicates defined by the utility module itself are not saved in the file. If the file already exists, SAVE() deletes it before writing the new file.

KILLALL()

KILLALL() deletes the entire relational database (except the predicates defined by the utility module itself - otherwise KILLALL() would delete itself!). Naturally some care should be exercised when using this predicate.