PROLOG PROGRAMMER'S MANUAL

Database Predicates

ADDCL (<rule>,<clause number>)

ADDCL() is used to add new clauses to Prolog's internal relational database. Although a rule typed into Prolog in interactive mode will always cause that rule to be added to the list of rules for the predicate named in the head of the rule, rules added in interactive mode always appear as the last rule for their predicate. Usually this is correct, but when it is required to add a rule at a specific position in the list of rules, other than at the end, ADDCL() is used to achieve this. ADDCL() can also be invoked by a Prolog program to add new rules to the database. The <rule> parameter specifies a rule to be added to the database. Normally, rules are specified in the normal form, eg:-

```
father(?x,?y):-parent(?x,?y),male(?x);
```

In order to add this rule as the second rule for the predicate father(), the following form of ADDCL() could be used:

```
ADDCL(father(?x,?y):-parent(?x,?y),male(?x);,1);
```

The <clause number> parameter is set to 1 as clause numbers start at 0. If clause number 1 has already existed when this ADDCL() was invoked, the old clause number 1 would have become clause number 2, and any other clauses beyond would have been similarly renumbered. Thus ADDCL() should really be considered as performing an insert on the database, ie no old clauses are lost when ADDCL() is executed. If when ADDCL() is executed, there are no clauses for the predicate (ie. the predicate does not exist in the relational database), ADDCL() creates a new predicate with the given name and inserts the given rule as its first (and only) clause. If the clause number given exceeds the number of the last clause already in the database for the predicate, the new rule is simply appended as the new last clause for the predicate. Note that all variables appearing in the rule are substituted for before ADDCL() attempts to add the new rule to the database. By knowing the internal form in which rules are kept in the relational database, a program can construct new rules to add the the database more flexibly than using the simple template illustrated in the previous examples. Internally, a rule is simply a list of predicates. Each predicate in the list is either in the form:-

```
name.args
```

or:-

```
variable.args
```

In this form, the head of the predicate call gives the name of the predicate, or is a variable which at runtime has the value of the name of the predicate. The arguments consist simply of a list of arguments to the predicate, terminated by nil. For example:

```
W("hello, world") is represented as "W"."hello, world".nil
FAIL() is represented as "FAIL".nil
walk(a.b.(?x.?y),preorder) is represented as "walk".("a"."b".(?x.?
y))."preorder".nil
```

The list of predicates forming the rule consists of a head predicate which represents the goal predicate on the left-hand side of the ':-', and a tail which is a list of the predicates on the right-hand side of the ':-'

terminated by nil. For example:

```
p():-q(),r(); is represented as p().q().r().nil
p():-; is represented as p().nil
p():-q(),r(),s(); is represented as p().q().r().s().nil
```

Knowing this, the predicate call:

```
ADDCL(?x.?y,999)
```

would add the rule:

```
father(?a,?b):-parent(?a,?b),male(?a);
```

to the relational database provided that at the time of the call, the variable ?x had the value

```
"father".?a.?b.nil
```

and ?y had the value

```
("parent".?a.?b.nil).("male".?a.nil).nil
```

If, on the other hand, ?x had the value

```
"mother".?a.?b.nil
```

and ?y had the value

```
("parent".?a.?b.nil).("female".?a.nil).nil
```

then the execution of the same ADDCL() would have added the rule:

```
mother(?a,?b):-parent(?a,?b),female(?a);
```

to the relational database.

The user should note that ADDCL() always succeeds, that the addition of a new rule to the database is strictly non-logical, and that the new rule is not deleted on backtracking. It is however possible to write a non-deterministic version of ADDCL() that deletes the new rule on backtracking:

```
addcl(?rule,?num):-ADDCL(?rule,?num);
addcl((?name.?args).?body,?num):-DELCL(?name,?num);
```

Caution should be exercised in the use of such a rule - its behaviour on backtracking depends on other changes made to the predicate to which the rule was added before backtracking occurs.

The principle uses of ADDCL() are rule editing, using the relational database as scratch memory, and using Prolog programs to write other Prolog programs. Generally, programmers should avoid using the relational database as scratch memory. This is because accessing the database is slower than accessing variables. Most programs written using the relational database as scratch memory have more elegant

and faster solutions using variables.

Where the order of rules is not important, but a program needs to add many clauses to a database, a useful variation on ADDCL() can be written as:

```
addcl(?rule):-ADDCL(?rule,9999);
```

which will cause new rules to be written to the end of the list of clauses for their predicate. A limit of 9999 is used in the program, in fact it is fairly certain that all available memory will be used up before this limit is reached.

CL(<variable>,<predicate name>)

CL() is a system predicate which extracts the list of clauses for a particular predicate from the relational database (hence the mnemonic - Clause List). CL() must be given two arguments, a variable (or a term which evaluates to an unbound variable at the time of execution) and a constant (or a term which evaluates to a constant at the time of execution). CL() searches the relational database for the named predicate. If it is not found, an error is issued and program execution is aborted. If this is not the desired action, the DB() system predicate can be used to obtain a list of the predicates which are defined in the relational database, against which the <predicate name> parameter may be checked before CL() is invoked.

For notes on the form in which clauses are stored internally, see the documentation for ADDCL(). The clause list returned by CL() consists of a list of clauses in the form required by ADDCL(), terminated by nil. Note that the clauses are always returned in exactly the same order in which they are stored, which is the same as the order in which they are scanned to satisfy a goal during program execution.

For example, if a Prolog program contains the following clauses:

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

append(nil,?x,?x):-;
append(?t.?x,?y,?t.?z):-append(?x,?y,?z);

rev(nil,?x,?x):-;
rev(?x.?y,?z,?r):-rev(?y,?x.?z,?r);
```

then the following CL() calls would give the results shown during an interactive session:

```
P>CL(?x,not);
<?x=(("not".?p.?argp.nil).(?p.?argp).("/".nil).("FAIL".nil).nil).(("not".?p.?
argp.nil).nil).nil>

P>CL(?x,append);
<?x=(("append".nil.?x.?x.nil).nil).(("append".(?t.?x).?y.(?t.?z).nil).
("append".?x.?y.?z.nil).nil).nil>

P>CL(?x,rev);
<?x=(("rev".nil.?x.?x).nil).(("rev".(?x.?y).?z.?r.nil).("rev".?y.(?x.?z).?
r.nil).nil).nil>
```

Although the variable names in the invoking code (in the above cases, the variable ?x in the goal given by the user) may be the same as the variable names in the clauses retrieved from the relational database, Prolog actually creates new instances of these variables. So, despite the fact that ?x appears in the CL() query in the third example, and also in the clause list for rev(), the two variables are quite distinct, and there is no danger of Prolog confusing them (although there may be a danger of confusion by the user).

The CL() query is typically used for clause editing, for debugging, and for writing alternative query

evaluation strategies. For clause editing, if the user names a predicate to be edited, it is a simple matter for a Prolog program to retrieve the clause list from the relational database using the CL() predicate, and to perform context-sensitive editing directed by the user. The edited clauses can then be reinserted using the ADDCL() predicate. Normally the DELCL() or KILL() predicates would also be invoked to delete the old versions of the clauses before replacing them. The general scheme of such an editor would be:

```
edit(?name):-CL(?old,?name),change(?old,?new),KILL(?name),add(?new);
```

A rule for add() might be:

```
add(nil):-;
add(?clause.?rest):-ADDCL(?clause,9999),add(?rest);
```

Clearly the Prolog interpreter itself could be written in Prolog. The outline of such an interpreter might be:

```
goal(?p.?argp):-CL(?clauses,?p),try(?clauses,?p.?argp);

try(((?p.?argp).?body).?rest,?p.?argp):-goal-list(?body);
try(?head.?rest,?p.?argp):-try(?rest);

goal-list(nil):-;
goal-list(?first.?rest):-goal(?first),goal-list(?rest);
```

Clearly, further details need to be added, for instance regarding system predicates. The example serves to illustrate however that program tracing and query evaluation can in fact be written in Prolog. For tracing, information regarding which goals have been successfully unified, and at which points backtracking begins, as well as the opportunity for user intervention, need to be added. For alternative query evaluation strategies, the most obvious change is to alter the order in which rules are tried by try(). This could be accomplished by a simple sort - for example sorting the clause list by length of clause could be used to provide a simple heuristic.

DB(<variable>)

DB() is a system predicate which allows a complete list of all predicates in the relational database to be made. The list is built at the time of invocation by stepping through the relational database index. The resulting list is a list of constants which are the names of the predicates in the relational database sorted alphabetically and terminated by nil. For example, assume the relational database is empty (as after just starting the interpreter). The following interactive queries could be given:

```
P>a():-;

P>b():-;

P>c():-;

P>DB(?x);
<?x="a"."b"."c".nil>

P>apple():-;

P>banana():-;

P>DB(?x);
<?x="a"."apple"."b"."banana"."c".nil>
```

Notice that the DB() predicate does not return the names of the system predicates. This is because the system predicates are not stored as clause lists (they are in fact machine-code subroutines with their own index which enables the Prolog interpreter to take special actions when calling them), so they have no entry in the index to the relational database. Note also that if a predicate in created with a number of clauses, and then its clauses are deleted one by one using DELCL(), when the last remaining clause has been deleted, although the interpreter actually retains the index entry in the relational database, in all other respects the predicate is regarded as no longer existing, and its name will not be returned by DB(). The DB() predicate is most useful to check whether a particular predicate has any clauses. This is particularly important if the program is planning to execute the given predicate since the attempt to execute a non-existent predicate will cause an error and cause the Prolog interpreter to abort execution of the current program. A program to check the existence of a named predicate is given here:

```
exists(?name):-DB(?preds),isin(?name,?preds);

isin(?h,?h.?t):-;
isin(?h,?x.?t):-isin(?h,?t);
```

This program simply takes a predicate name, uses DB() to generate a list of all known predicate names, then invokes the Prolog program isin() to ascertain whether the given name occurs in the list returned by DB(). isin() is a useful general-purpose predicate which is included in the utility module - see utility module documentation.

DB() is also useful for maintaining modular Prolog programs. When working within a particular module, the DB() predicate can be used on entry and exit to establish what changes have been made to the predicates belonging to that module.

DELCL(<predicate name>,<clause number>)

DELCL() is a system predicate which allows individual clauses to be removed from the clause list for a predicate (unlike KILL() which deletes all clauses for a given predicate). Note that DELCL() is not a completely symmetrical partner to ADDCL(), for, where ADDCL() requires a clause containing a head for unification, and a tail, DELCL() only requires the name of the predicate from which a clause is to be deleted. The particular clause to be deleted is specified by the <clause number> parameter. As with ADDCL(), clauses within a predicate are numbered starting at zero. Thus, the execution of:

```
DELCL(a,0)
```

deletes the first clause for predicate 'a'. The following example shows more fully the effects of DELCL():

```
P>male(John):-;

P>male(Chris):-;

P>male(Dave):-;

P>male(Charles):-;

P>CL(?x,male);
<?x=(("male"."John".nil).nil).(("male"."Chris".nil).nil).
(("male"."Dave".nil).nil).(("male"."Charles".nil).nil).nil>

P>male(?x);
<?x="John">

<?x="Chris">

<?x="Dave">

<?x="Charles">

P>DELCL(male,3);
<>

P>CL(?x,male);
<?x=(("male"."John".nil).nil).(("male"."Chris".nil).nil).
(("male"."Dave".nil).nil).nil>

P>DELCL(male,0);

P>CL(?x,male);
<?x=(("male"."Chris".nil).nil).(("male"."Dave".nil).nil).nil>

P>male(?x);
<?x="Chris">

<?x="Dave">
```

The programmer should note that DELCL() always succeeds and is deterministic, ie it effects are not

undone on backtracking. Therefore DELCL() is strictly speaking non-logical. It is possible to define a non-deterministic version of DELCL() written in Prolog using ADDCL() and DELCL(). First, a useful predicate cl() is defined which ascertains the clause number for a clause matching a given pattern:

```
cl((?name.?args).?body,?n):-

      CL(?clist,?name),
      index(?clist,(?name.?args).?body,?n);

index(?x,?x.?y,0):-;
index(?x,?y.?z,?n):-

      index(?x,?z,?m),
      ADD(?m,1,?n);
```

Now delcl() is defined in terms of cl:

```
delcl((?name.?args).?body):-

      cl((?name.?args).?body,?n),
      or(DELCL(?name,?n),ADDCL((?name.?args).?body,?n));
```

or() is a simple meta-program:

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

KILL(<predicate name>)

KILL() is a system predicate which can be used to erase all the clauses for a particular predicate from the relational database. Note that although the space associated with the clauses is made available when KILL() is executed, and the interpreter will treat all subsequent references to the predicate as if it had never been defined, nevertheless the entry in the index of the relational database is retained. This means that although the relational database can be used as scratch memory by a Prolog program, the program cannot create an unlimited number of new predicates without filling up memory, even if these predicates are deleted in the course of the program. This point will not affect most Prolog programs, because the extent to which the relational database is generally used for scratch memory is quite limited, and the amount of memory required for the index entry in the relational database is fairly small. The retention of the index entry when the last clause of a predicate is deleted means that should clauses later be added to that predicate, the old index entry is re-used and no new index information is created. Furthermore, when a predicate has been removed using KILL(), despite retention of the index entry, the DB() predicate will not return the name of the deleted predicate. For example:

```
P>male(John):-;

P>male(Chris):-;

P>male(Fred):-;

P>female(Joan):-;

P>female(Sandra):-;

P>female(Christine):-;

P>DB(?x);
<?x=female.male.nil>

P>noun(book):-;

P>noun(cupboard):-;

P>DB(?x);
<?x=female.male.noun.nil>

P>KILL(male);
<>

P>DB(?x);
<?x=female.noun.nil>
```

KILL() is useful when editing a Prolog program, to abandon an incorrect predicate in order to start again. It is also useful when the state of the relational database is unknown and it is wished to force it to a particular state (for instance, when one program has finished and another is started, or when a program aborts whilst updating the relational database). For example, if the relational database is used to accumulate a list of answers to a problem, then a KILL() at the start of the program will ensure that no answers from an earlier execution creep into the list for a later run. Also, if a predicate is used to

store a global value that changes during the execution of the program, it is usual to delete the old value before using ADDCL() to record the new value. KILL() is a quick way of erasing the old value, eg:

```
update():-

      value(?x),
      KILL(value),
      ADD(?x,1,?y),
      ADDCL(value(?y),9999);
```

KILL() can also be used to erase the entire relational database. This is done by calling DB() to obtain a list of all the predicates in the database, then calling KILL() recursively to delete all the predicates whose names occur in the list:

```
killall():-
      DB(?x),
      kill-list(?x);

kill-list(nil):-;

kill-list(?x.?y):-
      KILL(?x),
      kill-list(?y);
```

Arithmetic Predicates

ADD(<number>,<number>,<variable>)

ADD() is a system predicate which allows the sum of two numbers to be computed. The first two arguments to ADD() must be numeric, and the third a variable which receives the sum of the first two. ADD() always succeeds (given arguments of the right type), and in particular no overflow checking is carried out. A few examples of ADD() are given below:

```
P>ADD(1,1,?x);
<?x=2>

P>ADD(10,-10,?x);
<?x=0>

P>ADD(-100,90,?x);
<?x=-10>

P>ADD(100,-90,?x);
<?x=10>
```

A more flexible version of ADD() can be defined using the subtraction predicate SUB() to solve simple equations when one of the first two arguments is a variable instead of the third. The higher-level version of ADD() is given below:

```
add(?x,?y,?z):-
      NUM(?x),
      NUM(?y),
      VAR(?z),
      ADD(?x,?y,?z);

add(?x,?y,?z):-
      NUM(?x),
      VAR(?y),
      NUM(?z),
      SUB(?z,?x,?y);

add(?x,?y,?z):-
      VAR(?x),
      NUM(?y),
      NUM(?z),
      SUB(?z,?y,?x);

add(?x,?y,?z):-
      NUM(?x),
      NUM(?y),
      NUM(?z),
      ADD(?x,?y,?u),
      EQ(?u,?z);
```

The non-deterministic add() is a predicate taking three arguments which may be interpreted as asserting that the sum of the first two arguments is equal to the third. In the case where all three arguments are

known, this amounts to a simple verification. In the case where two arguments are known but one argument is unknown, add() solves for the unknown, which results in the variable argument becoming bound. Some examples of goals using add() are given below:

```
P>add(1,2,?x);
<?x=3>

P>add(1,2,3);
<>

P>add(1,2,4);

P>add(1,?x,3);
<?x=2>

P>add(?x,2,3);
<?x=1>
```

Note that the non-deterministic version of ADD() includes the functionality of ADD() and that anywhere ADD() would work add() would also work (the converse is obviously not true).

DIV(<number>,<number>,<variable>)

DIV() is a system predicate which allows the integer quotient of two numbers to be computed. The first two arguments to DIV() must be numeric, and the third a variable which receives the quotient of the first two. DIV() always succeeds (given arguments of the right type), and in particular no overflow or division by zero checking is carried out. A few examples of DIV() are given below:

```
P>DIV(20,6,?x);
<?x=3>

P>DIV(10,-6,?x);
<?x=-1>

P>DIV(-100,90,?x);
<?x=-1>

P>DIV(-20,-6,?x);
<?x=3>
```

A more flexible version of DIV() can be defined using the multiplication predicate MUL() to solve simple equations when one of the first two arguments is a variable instead of the third. The higher-level version of DIV() is given below:

```
div(?x,?y,?z):-
      NUM(?x),
      NUM(?y),
      VAR(?z),
      DIV(?x,?y,?z);

div(?x,?y,?z):-
      NUM(?x),
      VAR(?y),
      NUM(?z),
      DIV(?x,?z,?y);

div(?x,?y,?z):-
      VAR(?x),
      NUM(?y),
      NUM(?z),
      MUL(?z,?y,?x);

div(?x,?y,?z):-
      NUM(?x),
      NUM(?y),
      NUM(?z),
      DIV(?x,?y,?u),
      EQ(?u,?z);
```

The non-deterministic div() is a predicate taking three arguments which may be interpreted as asserting that the quotient of the first two arguments is equal to the third. In the case where all three arguments are known, this amounts to a simple verification. In the case where two arguments are known but one argument is unknown, div() solves for the unknown, which results in the variable argument becoming

bound. Some examples of goals using div() are given below:

```
P>div(1,2,?x);
<?x=0>

P>div(20,6,3);
<>

P>div(20,6,9);

P>div(20,?x,3);
<?x=6>

P>div(?x,6,3);;
<?x=18>
```

Note that the non-deterministic version of DIV() includes the functionality of DIV() and that anywhere DIV() would work div() would also work (the converse is obviously not true).

MUL(<number>,<number>,<variable>)

MUL() is a system predicate which allows the product of two numbers to be computed. The first two arguments to MUL() must be numeric, and the third a variable which receives the product of the first two. MUL() always succeeds (given arguments of the right type), and in particular no overflow checking is carried out. A few examples of MUL() are given below:

```
P>MUL(2,3,?x);
<?x=6>

P>MUL(-2,3,?x);
<?x=-6>

P>MUL(2,-3,?x);
<?x=-6>

P>MUL(-2,-3,?x);
<?x=6>
```

A more flexible version of MUL() can be defined using the division predicate DIV() to solve simple equations when one of the first two arguments is a variable instead of the third. The higher-level version of MUL() is given below:

```
mul(?x,?y,?z):-
      NUM(?x),
      NUM(?y),
      VAR(?z),
      MUL(?x,?y,?z);

mul(?x,?y,?z):-
      NUM(?x),
      VAR(?y),
      NUM(?z),
      DIV(?z,?x,?y);

mul(?x,?y,?z):-
      VAR(?x),
      NUM(?y),
      NUM(?z),
      DIV(?z,?y,?x);

mul(?x,?y,?z):-
      NUM(?x),
      NUM(?y),
      NUM(?z),
      MUL(?x,?y,?u),
      EQ(?u,?z);
```

The non-deterministic mul() is a predicate taking three arguments which may be interpreted as asserting that the product of the first two arguments is equal to the third. In the case where all three arguments are known, this amounts to a simple verification. In the case where two arguments are known but one argument is unknown, mul() solves for the unknown, which results in the variable

argument becoming bound. Some examples of goals using mul() are given below:

```
P>mul(2,3,?x);
<?x=6>

P>mul(2,3,6);
<>

P>mul(3,4,13);

P>mul(3,?x,20);
<?x=6>

P>mul(?x,6,20);
<?x=3>
```

Note that the non-deterministic version of MUL() includes the functionality of MUL() and that anywhere MUL() would work mul() would also work (the converse is obviously not true).

SUB(<number>,<number>,<variable>)

SUB() is a system predicate which allows the difference between two numbers to be computed. The first two arguments to SUB() must be numeric, and the third a variable which receives the difference of the first two. SUB() always succeeds (given arguments of the right type), and in particular no overflow checking is carried out. A few examples of SUB() are given below:

```
P>SUB(1,1,?x);
<?x=0>

P>SUB(10,-10,?x);
<?x=20>

P>SUB(-100,90,?x);
<?x=-190>

P>SUB(100,-90,?x);
<?x=190>
```

A more flexible version of SUB() can be defined using the addition predicate ADD() to solve simple equations when one of the first two arguments is a variable instead of the third. The higher-level version of SUB() is given below:

```
sub(?x,?y,?z):-
      NUM(?x),
      NUM(?y),
      VAR(?z),
      SUB(?x,?y,?z);

sub(?x,?y,?z):-
      NUM(?x),
      VAR(?y),
      NUM(?z),
      SUB(?x,?z,?y);

sub(?x,?y,?z):-
      VAR(?x),
      NUM(?y),
      NUM(?z),
      ADD(?z,?y,?x);

sub(?x,?y,?z):-
      NUM(?x),
      NUM(?y),
      NUM(?z),
      SUB(?x,?y,?u),
      EQ(?u,?z);
```

The non-deterministic sub() is a predicate taking three arguments which may be interpreted as asserting that the difference between the first two arguments is equal to the third. In the case where all three arguments are known, this amounts to a simple verification. In the case where two arguments are known but one argument is unknown, sub() solves for the unknown, which results in the variable

argument becoming bound. Some examples of goals using sub() are given below:

```
P>sub(1,2,?x);
<?x=-1>

P>sub(1,2,-1);
<>

P>sub(1,2,3);

P>sub(1,?x,3);
<?x=-2>

P>sub(?x,2,3);
<?x=5>
```

Note that the non-deterministic version of SUB() includes the functionality of SUB() and that anywhere SUB() would work sub() would also work (the converse is obviously not true).

I/O Predicates

DEL(<filename>)

DEL() is a system predicate which allows a disk file to be deleted. If the file exists, it is deleted and DEL() succeeds. If the file does not exist, DEL() fails and normal backtracking begins. A particular use for the DEL() predicate other than for general file maintenance is in conjunction with the OUT() predicate. When OUT() is used to redirect output to a disk file, if that file already exists, data written to the file is appended at the end of the file. Sometimes it is necessary to make sure that if the file already exists it is deleted before new data are written so that old data are overwritten. This is easily accomplished using DEL(), remembering to trap the failure that occurs if the file does not exist:

```
del(?file):-
      DEL(?file);

del(?file):-;
```

Note that although the order of the clauses is apparently important for this predicate, in fact if failure occurs after the DEL() succeeds, the second clause succeeds without any side effects. A small saving in stack space may b achieved by placing a cut after the DEL():

```
del(?file):-
      DEL(?file),
      /;

del(?file):-;
```

The programmer may note that unnecessary use of the cut is generally frowned upon.

The above program could be modified to issue a message to the user in the event that a file already exists and the program wishes to delete it. For instance:

```
check(?file):-
      in(?file),
      in("/term"),
      issue-warning(),
      DEL(?file);

check(?file):-
      in("/term");
```

The issue-warning() predicate could simply warn the user that the file is about to be deleted, or it could ask for user intervention. If the predicate arranges to fail if the user does not wish to have the file deleted, then check() will not delete the file, as backtracking will ignore the rest of the first clause after issue-warning() and control will pass directly to the second clause which will simply restore input to the terminal. check() could then signal that the file has not been deleted by invoking the FAIL() predicate to force failure. There is no need for a cut to follow FAIL() since there are no alternative clauses for check at this point.

GETC(<variable>)

GETC() is a system predicate which allows one character to be read from the current input stream. If GETC() is executed when the current input stream is at the end-of-file position, it fails. Otherwise, the <variable> argument is bound to the character read from the input stream, as a single-character constant. Remember that the newline character is read as a single character from the input stream which may occasionally cause unexpected results, for example the following part of an interactive Prolog session:

```
P>GETC(?x);
<?x="@013">

P>
```

The point is that when the syntax analyser reads the semicolon on the goal line, the nature of the input line is entirely determined, ie a goal has been typed, and so the goal is submitted to the Prolog interpreter for evaluation before any further characters are taken from the input stream. Since the input stream is buffered when reading from the terminal, the newline that the user typed in after the semicolon is still waiting in the buffer. Of course the only action that is needed to prove the goal is to execute GETC(), which reads the newline from the input stream. The single result "@013" is then the decimal code for the unprintable newline character.

It is important to realise that input and output are strictly side effects of the execution of a logic program. Input and output functions are entirely deterministic, in particular GETC(), having read a character does not read the next character from the input stream when backtracking occurs. Control in fact returns to the most recent backtrack point (the most recent predicate with untried clauses).

GETC() can easily be used to write a Prolog program that reads in a list of characters from the input stream, terminated by a newline:

```
getlin(?x,?z):-
      GETC(?y),
      nextchar(?x,?y,?z);

nextchar(?z,"@013",?z):-;
nextchar(?y.?u,?y,?z):-
      getlin(?u,?z);
```

Note how the program uses two mutually recursive predicates to build up a line of input, one character at a time, until the newline character is read, terminating. Note also how an auxiliary list ?z is used to build up intermediate partial lines. This is required in order to build up the list in the correct order - a simpler program could be written without the use of an auxiliary list to return the characters in the line in reverse order. getlin() needs an initial auxiliary list to build from, this is the empty list nil, naturally. As an example of the use of getlin():

```
P>getlin(?x,nil),getlin(?y,nil);
Hello, world
<?y="H"."e"."l"."l"."o".","." "."w"."o"."r"."l"."d".nil,?x=nil>
```

Note that two calls to getlin() appear in the interactive goal, the first is simply to read in the newline waiting in the buffer. Since this line contains no characters, getlin() returns nil, the empty list. The second call to getlin() returns a list of the characters comprising the text "Hello, world" typed by the user.

IN(<filename>)

IN() is the system predicate used to redirect the input stream. All Prolog input predicates use the same input stream which is also the input stream through which interactive commands are received. The input stream can be redirected by use of the IN() predicate to take input from any valid input path. This means that input can be taken from any character device or file. When the Prolog interpreter starts, the input stream is the standard input passed from the calling program, usually the terminal passed by the shell. If the shell command invoking the Prolog interpreter contains an input redirection command, then Prolog will start by taking its input from whatever source the shell has redirected input from. This is useful for writing turnkey applications since shell command can be placed in a shell script which does not require the user to begin execution of a Prolog program. For example:

```
OS9:prolog <myprog
```

This shell command would start up the Prolog interpreter which would then take its commands from the file "myprog". If "myprog" contained the following:

```
countdown(0):-;
countdown(?n):-
      W(?n),
      W("@013"),
      SUB(?n,1,?m),
      countdown(?m);

countdown(5),QUIT();
```

the user would see the following:

```
5
4
3
2
1
```

The interpreter would then exit (because of the quit() predicate), without printing any results. Note that the Prolog interpreter notes that input is coming from a file rather than the terminal and therefore does not issue prompts.

The IN() predicate is also the way in which Prolog programs can be loaded from disk. Clearly the interactive command IN() can be used to cause predicates to be loaded from disk simply because if the disk file contains predicates in the form in which they would be typed from the terminal, redirecting input from the file will cause the predicates to be loaded into the relational database just as if they had been typed in from the terminal. However a useful feature of the Prolog interpreter is that if an end-of-file condition is detected whilst the syntax analyser is reading interactive commands (although strictly speaking they are not interactive if they come from a file), the input stream is automatically redirected to "/term".

The <filename> parameter to IN() must be a constant which names the path for input redirection. Generally, a pathname must be in quotes in order not to generate a syntax error. As an example of the

use of IN():

```
P>IN("myprog")
5
4
3
2
1
OS9:
```

This example assumes that "myprog" contains the same data as for the previous example. If however "myprog" contained:

```
countdown(0):-;
countdown(?n):-
      W(?n),
      cr(),
      SUB(?n,1,?m),
      countdown(?m);

cr():-
      W("@013");
```

then the dialogue would continue after the input redirection due to the automatic input redirection performed on reaching the end of "myprog", eg:

```
P>IN("myprog");
<>

P>countdown(3);
3
2
1
<>
```

Note that IN() can be used to load several Prolog programs into memory, effectively merging them, and even merging clauses for the same predicate from different files.

OUT(<filename>)

OUT() is the system predicate used to redirect output. All Prolog output predicates use the same output stream. This stream is initially directed to the standard output stream passed to the Prolog interpreter from the invoking program, usually the terminal passed the shell. If the shell command line contains an output redirection command, then Prolog starts output to the specified output destination. Note that any use of OUT() in a program overrides the redirection of standard output passed to the interpreter by the shell. Any valid path may be specified for output. The programmer should note that the syntax requirements of Prolog are such that most filename must be specified with quoted around them. Output may be redirected to a character device or file. For instance:

```
heading(?x):-
      OUT("/p"),
      W(?x),
      W("@013@03");
```

The heading() predicate redirects output to the parallel printer and writes its argument ?x to it followed by two newlines, so that the goal:

```
heading("Quarterly Financial Review");
```

causes the heading "Quarterly Financial Review" to be printed, followed by two newlines. Note that output redirection is permanent, ie that it persists after the termination or failure of the goal in which it occurred.

If OUT() specifies a disk file for output redirection, then different actions are taken depending on whether the disk file already exists or not. If the file does not exist, then it is created if possible. If the file already exists, then it is opened in append mode, which means that all its original contents are preserved and any data written to it by the Prolog program will be written to the end of the file. This means that if the programmer's intention is that the previous contents of the file should be overwritten, then the file must be explicitly deleted using the DEL() system predicate.

If the path specified for output redirection cannot be opened for any reason then OUT() fails, otherwise OUT() succeeds. Like all I/O predicates, OUT() is entirely deterministic and the actual I/O function may be regarded as a side effect from the logical point of view. In particular, the file remains open if backtracking occurs - the path is not closed if the goal containing the call to OUT() subsequently fails.

Note that the Prolog interpreter issues all its output through the same stream, including prompts and error messages.

Note that although IN() can be used to load Prolog programs from disk, OUT() cannot be used directly to save Prolog programs in the relational database to disk. This is because a moderate amount of work is required to convert clauses in the relational database to a form in which they can be reloaded using IN(), and this would constitute an unnecessary overhead to the Prolog interpreter which can be transferred to the utility module. This also means that customised versions of a save function can be written, for instance to provide program module handling. A save predicate is provided in the Prolog utility module. Any save function will naturally call upon OUT() to redirect output to the disk file in

which the clauses are to saves while it writes them.

PUTC(<character>)

PUTC() is the system predicate which is used to write a single character to the output stream. PUTC() always succeeds. PUTC() maybe passed either a constant or a number as its argument. If PUTC() is passed a constant, it should consist of just a single character and this character is written to the current output stream. For example:

```
PUTC("A")
```

writes the letter 'A' to the output stream.

```
PUTC("@013")
```

writes a newline to the output stream.

If PUTC() is passed a number, the character with that ASCII code is written to the current output stream. For example:

```
PUTC(65)
```

writes the letter 'A' to the output stream.

```
PUTC(13)
```

writes a newline to the output stream.

As a simple example of the use of PUTC(), below is listed a program which lists a disk file on the terminal:

```
listfile(?filename):-
      in(?filename),
      copy(),
      in("/term");

copy():-
      GETC(?x),
      PUTC(?x),
      /,
      copy();

copy():-;
```

The program is started by a goal such as:

```
listfile("/d1/SOURCE/myprog")
```

which will list the contents of disk file "/d1/SOURCE/myprog" at the terminal. The program works by redirecting input from the file passed to listfile(), invoking copy() to copy from input to output until end-of-file is reached. copy() itself fetches one character from the input stream using GETC() and sends it to the output stream using PUTC(). Because GETC() fails when end-of-file is reached, a

second clause is provided for copy() which is executed when end-of-file is reached. This clause succeeds unconditionally. Because this second clause is present, a cut is performed after a successful GETC() to eliminate the stack space occupied by the choice point for the second clause. copy() is then called recursively to copy the rest of the file. Of course copy() could be used to send a file to the printer:

```
printfile(?file):-
      in(?file),
      out("/p"),
      copy(),
      in("/term"),
      out("/term");
```

copy() could also be used to perform a disk-to-disk copy:

```
filecopy(?from,?to):-
      in(?from),
      out(?to),
      copy(),
      in("/term"),
      out("/term");
```

Note that the predicates invoking copy() make a point of tidying up after themselves by restoring input and output to the terminal.

R(<variable>)

R() is Prolog's high-level input predicate. It causes a Prolog token to be read from the input stream and bound (in its internal form) to the <variable> parameter. A complete formal definition of a Prolog term is given elsewhere in this manual. The R() predicate uses the same syntax analyser to read a token as the Prolog interpreter does when reading commands in interactive mode, therefore exactly the same rules apply. Note that the syntax analyser ignores whitespace (blanks, tabs and newlines except within quotes) but that they can serve to delimit tokens which would otherwise be combined to form one token. For example, the string 'abc123' consists of the single constant "abc123" whereas the string 'abc 123' consists of a constant "abc" and a number 123. Normally R() takes just one token from the input stream and any other tokens following it will still be available. If R() reaches the end of the file, the predicate fails, otherwise it succeeds. If an input stream contains several terms one after the other, delimited by whitespace where necessary, R() may be used repeatedly to read them one at a time. Therefore, in much the same way that GETC() may be used to read in a line of characters delimited by a newline as a list, R() may be used to read in a sentence given a delimiter token. This is demonstrated in the following program:

```
sentence(?x,?z,?t):-
      R(?y),
      nextword(?x,?y,?z,?t);

nextword(?x,?t,?x,?t):-;

nextword(?y.?u,?y,?z,?t):-
      sentence(?u,?z,?t);
```

The arguments to sentence are: ?x the list of terms read from the input stream, ?z the initial list - usually nil, ?t the terminator term. For instance:

```
P>sentence(?x,nil,stop);
send more money stop
<?x="send"."more"."money".nil>
```

Here the goal specifies that the terminator for the sentence is "stop". The user types in the characters "send more money stop" and sentence() analyses the input into a list of Prolog terms.

R() is also useful for writing Prolog editors in Prolog, since one of the valid forms for a term is the clause itself. If presented with a clause, R() will read in the whole clause and convert it to its internal form. For example, a program to read a clause from the input stream and add it to the relational database could be written as:

```
newclause():-
      R(?clause),
      addcl(?clause,9999);
```

Now a program can be written which reads all the clauses in the input stream (until end-of-file is reached) and adds them to the relational database:

```
addall():-
```

```
        newclause(),
        /,
        addall();

addall():-;
```

Note the use of the cut to prevent uncontrolled stack growth. Inspection of the newclause() rule shows that newclause() fails when end-of-file is reached, because R() fails and newclause() has no alternative clauses. At this point, the second clause of addall() is invoked, and looping ceases. At this stage we can now write a simple loader to load a file into the relational database:

```
load(?file):-
        IN(?file),
        addall(),
        IN("/term");
```

load() simply redirects input from the specified file, calls addall() to add all the clauses found in that file, then restores input to the terminal. Although IN() is useful for loading in files when using the Prolog interpreter interactively, it cannot be used directly for this purpose from within a Prolog program. load() on the other hand can be used interactively or from within a program.

W(<term>)

W() is Prolog's high-level output predicate. W() writes its argument to the current output stream. Numbers are written as signed decimal integers with leading zeros suppressed. Constants are written just as a string of the characters of which they are comprised. The special term nil is written as "nil", and lists are written using the '.' operator and use the right-associativity of '.' to eliminate as many brackets as possible. For example:

```
Goal                Sent to output stream
----                ---------------------
W(123)              123
W(-123)             -123
W(0)                0
W(abc)              abc
W("abc")            abc
W("abc@65@66@67")   abcABC
W(nil)              nil
W(a.b.c)            a.b.c
W(a.(b.c))          a.b.c
W((a.b).c)          (a.b).c
```

Note that W() cannot be used to write terms to a file if it is intended to read them back using R(). This is because no account is taken of syntactical ambiguities which may arise if constants contain non-alphanumeric characters, non-printable characters or other confusing data. For example W("a@13b") would write the character 'a' then a newline them the character 'b'. If R() were used to read this back, two constants would be read, "a" then "b". The intervening newline would be ignored as whitespace. The WQ() predicate can be used to overcome these difficulties. A more flexible version of W() can be written that will write out a nil-terminated list of terms separated by spaces. This is often useful when writing to the terminal in an interactive program.

```
p(nil):-;
p(?x.nil):-
      /,
      W(?x);

p(?x.?y):-
      W(?x),
      PUTC(" "),
      p(?y);
```

For example:

```
P>p(play.it.again.sam.nil);
play it again sam{}
```

Sometimes it is useful to send a carriage return after the last item printed. The pp() predicate is a modification of the p() predicate which does this:

```
pp(nil):-
      PUTC(13);
```

```
pp(?x.nil):-
      /,
      W(?x),
      PUTC(13);

pp(?x.?y):-
      W(?x),
      PUTC(32),
      pp(?y);
```

For example:

```
P>pp(play.it.again.nil),pp(sam.nil);
play it again
sam
{}
```

WQ(<term>)

WQ() is an alternative high-level output predicate. The only difference between W() and WQ() is that whereas W() writes the characters in a constant straight out to the output stream, WQ() encloses constants in quotes (hence the mnemonic, Write Quoted) and also expresses non-printable characters using the '@' notation. This means that terms written using WQ() can always be read back using R() which is not true of terms written by W(). This makes WQ() very useful for recording Prolog terms and clauses in disk files. A predicate can be written out using a program written using WQ():

```
wpred(?name.?args):-
      WQ(?name),
      PUTC("("),
      wargs(?args),
      PUTC(")");

wargs(nil):-;
wargs(?arg.nil):-
      /,
      WQ(?arg);

wargs(?arg.?rest):-
      WQ(?arg),
      PUTC(","),
      wargs(?arg);
```

For example:

```
P>wpred(p(a,b,c));
p(a,b,c){}
```

Now a predicate can be written in terms of wpred() to write a clause:

```
wcl(?head.?body):-
      wpred(?head),
      W(":-@13"),
      wbody(?body);

wbody(nil):-
      W("    ;@13");

wbody(?pred.nil):-
      /,
      wpred(?pred),
      W(";@13");

wbody(?pred.?rest):-
      wpred(?pred),
      W(",@13"),
      wbody(?rest);
```

For example:

```
P>wcl(p(a,b):-q(a),r(b););
```

```
"p"("a","b"):-
      "q"("a"),
      "r"("b");
```

Taking this still further, a predicate can be written which writes a list of clauses to the output stream:

```
wclist(nil):-;
wclist(?clause.?rest):-
      wcl(?clause),
      wclist(?rest);
```

Now a program can be written which lists the clauses for a particular predicate:

```
list(?name):-
      CL(?clauses,?name),
      wclist(?clauses);
```

For instance, to list the predicate "wcl" use the goal list(wcl). To list all the predicates in the database:

```
listall():-
      DB(?preds),
      plist(?preds);

plist(nil):-;

plist(?pred.?rest):-
      list(?pred),
      plist(?rest);
```

listall() is itself quite useful and with a small additional program can be turned into a predicate for saving the entire database to disk:

```
save(?file):-
      out(?file),
      listall(),
      out("/term");
```

The predicates just given provide only the minimum saving and listing functions, but the programs could easily be modified to be more flexible.

ASC(<constant>,<variable>)

ASC() is used for converting between single-character constants and their ASCII codes. The first argument to ASC() should be a single-character constant. The second argument should be a variable, which becomes bound to the ASCII code for the character. For example:

```
P>ASC(A,?x);
<?x=65>

P>ASC("*",?x);
<?x=42>

P>ASC(z,?x);
<?x=122>

P>ASC("@13",?x);
<?x=13>
```

ASC()is useful for performing character type tests, for example:

```
isupper(?c):-
      ASC(?c,?a),
      GE(?a,65),
      LE(?a,90);

islower(?c):-
      ASC(?c,?a),
      GE(?a,97),
      LE(?a,122);

isdigit(?c):-
      ASC(?c,?a),
      GE(?c,48),
      LE(?c,57);
```

These predicates test whether their argument is an upper case letter, a lower case letter or a digit respectively. For example:

```
isupper(K) succeeds,
isupper(p) fails,
islower(t) succeeds,
islower("$") fails,
isdigit("7") succeeds,
isdigit(H) fails.
```

Other useful tests can be defined in a similar way:

```
isalpha(?c):-
      islower(?c);

isalpha(?c):-
      isupper(?c);
```

```
isalnum(?c):-
      isalpha(?c);

isalnum(?c):-
      isdigit(?c);
```

isalpha() and isdigit() test whether their arguments are alphabetic characters (upper or lower case) and alphanumeric characters (alphabetic or digits) respectively.

As long as the arguments passed to ASC() are of the correct type, ASC() always succeeds. If the arguments are incorrect, an error is caused.

CHR(<number>,<variable>)

CHR() is used to convert from ASCII codes to constants. The first argument to CHR() should be a number, and the second argument should b a variable. The variable becomes bound to the single-character constant, the single character being that character whose ASCII code is the number given as the first argument. For example:

```
P>CHR(65,?c);
<?c="A">

P>CHR(61,?c);
<?c="=">

P>CHR(50,?c);
<?c="2">
```

CHR() is useful for performing character conversions, for example:

```
toupper(?in,?out):-
      islower(?in),
      /,
      ASC(?in,?x),
      SUB(?x,32,?y),
      CHR(?y,?out);

toupper(?in,?in);
tolower(?in,?out):-
      isupper(?in),
      /,
      ASC(?in,?x),
      ADD(?x,32,?y),
      CHR(?y,?out);

tolower(?in,?in);
```

toupper() takes a single-character constant as input and returns the upper case equivalent if it is lower case otherwise returns it unchanged. tolower() takes a single-character constant as input and returns the lower case equivalent if it is upper case otherwise returns it unchanged. The definitions of isupper() and islower() are given in the section for ASC(). Usually it is required to convert more than one character at a time. The following predicates apply tolower() and toupper() to a nil-terminated list of characters, returning a converted list of characters:

```
list2upper(nil,nil):-;

list2upper(?in.?inrest,?out.?outrest):-
      toupper(?in,?out),
      list2upper(?inrest,?outrest);

list2lower(nil,nil):-;

list2lower(?in.?inrest,?out.?outrest):-
      tolower(?in,?out),
```

```
        list2lower(?inrest,?outrest);
```

Finally a constant consisting of several characters can be converted by splitting it into its constituent characters, performing a list conversion, then joining the characters of the output list to form a new constant (see SPLIT() and JOIN() for more information):

```
con2upper(?in,?out):-
      SPLIT(?clist,?in),
      list2upper(?clist,?upperlist),
      JOIN(?out,?upperlist);

con2lower(?in,?out):-
      SPLIT(?clist,?in),
      list2lower(?clist,?lowerlist),
      JOIN(?out,?lowerlist);
```

JOIN(<variable>,<list>)

JOIN() takes a list of constants and concatenates them together to form a single constant. The first argument is a variable which becomes bound to the new constant. The second argument is a list of constants, terminated by a nil, from which the new constant is built. For example:

```
P>JOIN(?x,play.it.again.sam.nil);
<?x="playitagainsam">

P>JOIN(?x,h.e.l.l.o.","." ".w.o.r.l.d.nil);
<?x="hello, world">

P>JOIN(?x,"hello.", "."world".nil);
<?x="hello, world">
```

JOIN() is very useful for many types of string manipulation. As an example of the use of JOIN(), the following predicate can be used to check if a given constant is made from a combination of particular letters, or indeed to generate all possible constants formed from combinations of the given letters:

```
perm(?x):-
      letterlist(?y),
      JOIN(?z,?y),
      EQ(?x,?z);

letterlist(?a.nil):-
      letter(?a);

letterlist(?a.?b):-
      letter(?a),
      letterlist(?b);
```

Assume that the predicate letter() defines the letters "a", "b" and "c" as follows:

```
letter(a):-;
letter(b):-;
letter(c):-;
```

Then perm() can be used for checking:

```
P>perm(abc);
{}

P>perm(abd);

P>perm(cca);
{}

P>perm(cfa);
```

perm() can also be used to generate all possible permutations, but note that since no limit has been placed on the length of the constant the program loops and will never terminate:

```
P>perm(?x);
<?x="a">

<?x="b">

<?x="c">

<?x="aa">

<?x="ab">

<?x="ac">

<?x="ba">
```

and so on. The use of a dictionary allows many types of word puzzle to be solved. We need a word() predicate which can be used to generate all the words in the dictionary. A simple definition would be:

```
word(a):-;
word(aback):-;
word(abacus):-;
word(abaft):-;
word(abandon):-;
...
word(zoom):-;
word(zoophyte):-;
```

In practice only quite a small dictionary could be held this way in memory, and if a large dictionary was required some way would have to be found of compressing it, or it could be held on disk and word() rewritten so that it behaved in the same way as the version given above but took its words from a disk file. In any case, all possible words made from a particular list of letters can be generated by:

```
P>perm(?x),word(?x);
```

All words that can be made using letters satisfying letter() will be listed.

SPLIT(<variable>,<constant>)

SPLIT() is used to explode a constant into a list of its constituent letters. The first argument to SPLIT() is a variable which becomes bound to a nil-terminated list of single-character constants which are the characters comprising the second argument which must be a constant. For example:

```
P>SPLIT(?x,"hello, world");
<?x="h"."e"."l"."l"."o".","." "."w"."o"."r"."l"."d".nil>
```

SPLIT() is used with JOIN() to perform most string-manipulation functions. The following example splits a constant on a delimiter, returning the part before the delimiter and the part after the delimiter as two new constants:

```
d-split(?in,?delim,?before,?after):-
      SPLIT(?inchars,?in),
      append(?bchars,?delim.?achars,?inchars),
      JOIN(?before,?bchars),
      JOIN(?after,?achars);

append(nil,?x,?x):-;
append(?t.?x,?y,?t.?z):-
      append(?x,?y,?z);
```

This predicate works by splitting the input constant into a list of its characters then using the standard append() predicate to search for two lists which can be concatenated to form the input list, with the additional constraint that the first element in the second string must be the required delimiter. Note that this predicate can find more that one solution when the delimiter occurs more that once. For example:

```
P>d-split("hello, world",",",?a,?b);
<?b=" world",?a="hello">

P>d-split("13/5/60","/",?a,?b);
<?b="5/60",?a="13">

<?b="60",?a="13/5">

P>d-split("the cat sat on the mat"," ",?a,?b);
<?b="cat sat on the mat",?a="the">

<?b="sat on the mat",?a="the cat">

<?b="on the mat",?a="the cat sat">

<?b="the mat",?a="the cat sat on">

<?b="mat",?a="the cat sat on the">
```

The following predicate checks whether a particular character occurs in a string:

```
inchar(?con,?char):-
      SPLIT(?charlist,?con),
      isin(?char,?charlist);
```

```
isin(?x,?x.?y):-;
isin(?x,?y.?z):-
      isin(?x,?z);
```

inchar() takes a constant and a character, and succeeds or fails according as the character appears in the constant. For example:

```
P>inchar(",","hello, world");
{}

P>inchar(",","hello world");
```

JOIN() and SPLIT() are very useful because they allow any operation that can be programmed for lists to be applied to strings with very little extra effort. Note that in the examples above, the real work was done by append() and isin() which are straightforward list-processing predicates which can be used much more generally than just string processing.

Relational Predicates

EQ(<term>,<term>)

EQ() gives the programmer explicit access to unification. Although EQ() may be used to test whether or not two terms are equal to eachother, EQ() actually performs the test by attempting to unify its arguments. EQ() behaves in fact as if it were written in Prolog as:

```
EQ(?x,?x):-;
```

Simple examples of the use of EQ():

```
P>EQ(4,4);
{}

P>EQ(hello,hello);
{}

P>EQ(hello,goodbye);

P>EQ(5,0);
```

Because EQ() attempts unification of the two terms passed as arguments, trees are walked recursively, for example:

```
P>EQ(hello.world.nil,hello.world.nil);
{}

P>EQ(hello.world,hello.world.nil);
```

Of course unification actually solves equations in the more general case, so for example:

```
P>EQ(?x,hello.world);
<?x="hello"."world">

P>EQ(?x.world,hello.?y);
<?y="world",?x="hello">

P>EQ((play.?x).(again.?y),(?r.it).(?s.sam));
<?s="again",?r="play",?y="sam",?x="it">

P>EQ(plus(3,?x),plus(?y,7));
<?x=7,?y=3>

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

P>EQ(<s-node,left-branch,right-branch>,?x(?y,?z));
<?z="right-branch",?y="left-branch",?x="s-node">

P>EQ(?x,rev(nil,?x,?x):-;);
<?x=(("rev".nil.?x.?x.nil).nil).nil>
```

The last few examples may appear to be rather obscure at first, but serve to illustrate the point that Prolog deals with all its complex data in only one form, the list, which is built and represented using the list constructor (.). Prefixed terms and tuples are all reduced to this form on input so that they may be mixed freely and EQ() can solve equations between superficially different data types.

Some mischievous uses of EQ() although apparently useful (and occasionally actually useful) can be very dangerous because they lead to the construction of infinite terms. The innocent-looking goal:

```
P>EQ(?x,a.?x);
```

actually causes the interpreter to go into an infinite loop trying to print the value of ?x because the only solution is actually an infinite tree which Prolog does not have the ability to represent concisely. If the result of such an equation is not printed, there are some useful applications of infinite trees. For example:

```
EQ(?x,mon.tue.wed.thu.fri.sat.sun.?x>
```

creates a perpetual calendar bound to the variable ?x. Although extreme caution must be exercised in the use of infinite trees, and they cannot successfully be unified with other terms in general, some simple operations such as extracting the n-th element can be successfully programmed.

GE(<number>,<number>)

or

GE(<constant>,<constant>)

The first form of GE() is used to test whether the first argument is numerically greater than or equal to the second. If this is true, GE() succeeds, otherwise it fails. For example:

```
P>GE(10,3);
<>

P>GE(3,10);

P>GE(5,5);
<>

P>GE(10,-5);
<>

P>GE(-5,10);

P>GE(-5,-5);
<>
```

The second form of GE() is used to test whether the first argument is lexicographically greater than or equal to the second, ie whether it is the same or whether it appears later in the ASCII collation sequence. For example:

```
P>GE(beta,alpha);
<>

P>GE(alpha,beta);

P>GE(brother,brush);

P>GE(brush,brother);
<>

P>GE(triangle,triangle);
<>
```

GE() can be used to write a list sorting program. Many sorting techniques are known, but one of the most efficient and elegant is Hoare's "Quicksort". This is difficult to program in many languages, but in Prolog, Quicksort can be written down very concisely:

```
sort(nil,nil):-;
sort(?x.?y,?z):-
      partition(?y,?x,?l,?g),
      sort(?l,?ls),
      sort(?g,?gs),
      append(?ls,?x.?gs,?z);
```

```
partition(nil,?x,nil,nil):-;
partition(?z.?y,?x,?z.?l,?g):-
      GE(?x,?z),
      /,
      partition(?y,?x,?l,?g);

partition(?z.?y,?x,?l,?z.?g):-
      partition(?y,?x,?l,?g);
```

append() has the usual definition. The sort works by removing the first element of the list then partitioning the remainder of the list into two lists, one of elements less than or equal to the element removed, one of elements greater than the element removed. These lists are then sorted themselves using quicksort recursively. The results are joined back together using append(). For example:

```
P>sort(4.1.5.2.6.2.nil,?x);
<?x=1.2.2.4.5.6.nil>
```

GT(<number>,<number>)

or

GT(<constant>,<constant>)

The first form of GT() tests whether its first argument is numerically strictly greater than its second. If it is, GT() succeeds, otherwise it fails. For example:

```
P>GT(10,3);
<>

P>GT(3,10);

P>GT(5,-5);
<>

P>GT(-5,5);

P>GT(0,0);

P>GT(5,5);
```

The second form of GT() tests whether its first argument is lexicographically strictly greater than its second, ie whether it succeeds it in the ASCII collating sequence. For example:

```
P>GT(earwig,darling);
<>

P>GT(aroma,trilobite>;

P>GT(rollerball,rollerball);
```

As an example of the use of GT(), a predicate vgt() (vector greater than) cn be defined which checks whether the magnitude of one multi-dimensional vector is greater than that of another. Multi-dimensional vectors are represented as lists of numbers, terminated by nil.

```
vgt(?x,?y):-
      mag(?x,?magx),
      mag(?y,?magy),
      GT(?magx,?magy);

mag(?x,?magx):-
      sum-of-squares(?x,?ssx),
      sqrt(?ssx,?magx);

sum-of-squares(nil,0):-;
sum-of-squares(?x.?y,?z):-
      sum-of-squares(?y,?z'),
      MUL(?x,?x,?x'),
      ADD(?x',?z',?z);

sqrt(?x,?y):-
```

```
      trial-sqrt(?x,?y,0),
      /;

trial-sqrt(?x,?y,?y):-
      MUL(?y,?y,?z),
      GE(?z,?x);

trial-sqrt(?x,?y,?z):-
      ADD(?z,1,?z'),
      trial-sqrt(?x,?y,?z');
```

Note how the square root is approximated by trying squares of numbers starting at 0 until the square becomes greater than or equal to the number whose root is being found. This method is clearly simple to program but rather inefficient.

LE(<number>,<number>)

or

LE(<constant>,<constant>)

The first form of LE() checks whether its first argument is numerically less than or equal to its second. If it is, then LT() succeeds, otherwise it fails. For example:

```
P>LE(5,10);
<>

P>LE(5,5);
<>

P>LE(-5,5);
<>

P>LE(5,-5);

P>LE(-5,-5);
<>
```

The second form of LE() checks whether its first argument is lexicographically less than or equal to its second, ie whether it is equal or precedes it in the ASCII collating sequence. For example:

```
P>LE(marathon,marathon);
<>

P>LE(marathon,mars);
<>

P>LE(zodiac,mars);
```

As an example of the use of LE(), a predicate validate() is defined which validates a date against lower and upper limits passed to it. To do this, another predicate datenum() is written which converts a date to a number with the property that if one date precedes another, its date number is smaller than the other:

```
date-num(?y,?m,?d,?n):-
      SUB(?y,1900,?y'),
      MUL(?y',12,?y''),
      ADD(?y'',?m,?m'),
      MUL(?m',32,?m''),
      ADD(?m'',?d,?n);
```

date-num() assumes that years are given for example as 1987. Dates before 1900 are not catered for by date-num. validate() can now be written:

```
validate(?y,?m,?d,?ylo,?mlo,?dlo,?yhi,?mhi,?dhi):-
      date-num(?y,?m,?d,?n),
      date-num(?ylo,?mlo,?dlo,?nlo),
      date-num(?yhi,?mhi,?dhi,?nhi),
      range-check(?n,?nlo,?nhi);
```

```
range-check(?n,?nlo,?nhi):-
      LE(?nlo,?n),
      LE(?n,?nhi);
```

range-check() is a general-purpose predicate which checks that its first argument does not fall outside the range defined by its second and third arguments.

LT(<number>,<number>)

or

LT(<constant>,<constant>)

The first form of LT() checks whether its first argument is numerically strictly less than its second. If it is then LT() succeeds, otherwise it fails. For example:

```
P>LT(5,10);
<>

P>LT(5,10);

P>LT(-5,5);
<>

P>LT(5,-5);

P>LT(3,3);
```

The second form of LT() checks whether its first argument is lexicographically strictly less than its second, ie whether it precedes it in the ASCII collating sequence. For example:

```
P>LT(alpha,beta);
<>

P>LT(fiddle,cat);

P>LT(diddle,diddle);
```

As an example of the use of LT(), a predicate extract() is defined which extracts from a database all the records which are dated earlier than today. It is assumed that today's date is stored in a clause in the form today(<year>,<month>,<day>), for instance:

```
today(1986,12,25):-;
```

It is assumed that the records are stored in clauses for the predicate record(). We assume that the date is contained in the first three arguments of the predicate, eg:

```
record(1960,4,22,<blah,blah,blah>):-;
```

The tuple in the fourth argument represents the rest of the data in the record (for convenience). First, a general-purpose predicate is defined which performs the LT() comparison between two dates:

```
date-lt(?y1,?m1,?d1,?y2,?m2,?d2):-
      LT(?y1,?y2);

date-lt(?y,?m1,?d1,?y,?m2,?d2):-
      LT(?m1,?m2);

date-lt(?y,?m,?d1,?y,?m,?d2):-
```

```
        LT(?d1,?d2);
```

date-lt() can be used on its own, for example:

```
P>date-lt(1987,3,4,1986,1,7);

P>date-lt(1948,5,3,1987,4,3);
<>

P>date-lt(1955,6,6,1955,7,2);
<>
```

extract() is now quite easy to define:

```
extract():-
      today(?y,?m,?d),
      record(?yr,?mr,?dr,?recdata),
      date-lt(?yr,?mr,?dr,?y,?m,?d),
      display-record(?recdata);
```

We assume that the predicate display-record() prints the contents of one record. extract() is easily invoked by the goal extract();

NE(<term>,<term>)

NE() is a system predicate which may be used to establish that two terms are strictly different. NE() is the converse of EQ(), and is equivalent to the Prolog program:

```
NE(?x,?x):-
      /,
      FAIL();
```

NE() actually invokes unification on its arguments and tries to make them equal. Because unification is used rather than simple comparison, NE() can attempt to solve equations in an attempt to unify its arguments. If the unification succeeds, then NE() fails and failure automatically undoes all bindings made as a result of the unification. If unification fails, then any bindings made during the failed unification are undone before NE() succeeds. Therefore NE() will never result in the binding of any variables.

As an example of the use of the NE() predicate, the predicate not-in() checks that its first argument is not in a list given as its second argument:

```
not-in(?x,nil):-;
not-in(?x,?y.?l):-
      NE(?x,?y),
      not-in(?x,?l);
```

Using this predicate, another useful predicate different() can be written to check that all the elements of a list are different from each other:

```
different(nil):-;

different(?x.?l):-
      not-in(?x,?l),
      different(?l);
```

It is now an easy matter to use this predicate, for example, to calculate permutations of letters. The following program calculates permutations of the letters a, b, c, d:

```
perm(?x1,?x2,?x3,?x4):-
      letter(?x1),
      letter(?x2),
      letter(?x3),
      letter(?x4),
      different(?x1.?x2.?x3.?x4.nil);
```

For example:

```
P>not-in(a,g.f.b.c.a.t.w.nil);

P>not-in(g,a.d.h.w.nil);
<>

P>different(the.cat.sat.on.the.mat.nil);
```

```
P>different(a.cat.sat.on.the.mat.nil);
<>
```

The goal perm(?a,?b,?c,?d); results in the enumeration of all the possible permutations of the letters a, b, c, d, ie: abcd, abdc, acbd, acdb, adbc, adcd etc. A generalised version of such a program, particularly in conjunction with a dictionary program would very be useful for solving many types of word puzzle.

Control predicates

/

/ is the famous 'cut' predicate. Its basic effect is to cut out part of the search tree. As a Prolog program executes, it generally generates a stack of choice points. Each choice point on the stack represents a point where a particular clause has been tried for a predicate and that others remain should the last choice fail. Prolog makes several optimisations to a program as it runs to remove unnecessary stack entries, but there are occasions when the suppression of choices is essential for the correct operation of the program, and other cases where Prolog is not able to optimise, but a special knowledge of the program enables the programmer to explicitly suppress some choice points. Essentially, the appearance of a cut in a clause causes all choices to be deleted from the stack back to and including any clauses which are alternatives to the clause in which it appears. For example:

```
a(A):-
      p(),
      q(),
      /,
      r();

a(B):-
      p'(),
      q'();

a(C):-
      p''(),
      q''();
```

In this example, if the head of the first clause is matched, then the stack contains an entry indicating that a(B)... and a(C)... are alternative clauses. As p() and q() are evaluated, and assuming they succeed, further choices may be stacked up in exactly the same way. On encountering the cut (/), the alternatives for q(), p() and the alternative clauses a(B)... and a(C)... are deleted. Therefore, if r() should fail, bactracking would return to alternatives which were available at the time a() was invoked. The cut always succeeds and is only used for its effect on the stack. A side effect of this is that the space associated with the stack entry is freed, and the cut is often used simply to make programs more efficient in their use of the stack. If r() proceeded to generate alternatives on the stack, these alternatives would remain active on returning from a(), although they could be deleted by a subsequent cut found in the clause which invoked a() or indeed some higher-level parent clause. As an example o f a case where the use of the cut is essential, take the predicate not() which takes as its argument a single goal, and fails if this goal has a solution but succeeds otherwise. This is written:

```
not(?p.?args):-
      ?p.?args,
      /,
      FAIL();

not(?p.?args):-;
```

Careful study of this predicate is recommended. The first clause begins by evaluating the goal passed to

it as an argument (a simple case of metaprogramming). If this has no solution, control passes to the second clause which succeeds unconditionally. If the argument goal succeeds, a cut is then executed followed by a fail. The fail serves to fail the predicate in the event of a solution to the argument goal, but the cut prevents backtracking into alternative clauses for the argument goal (which might fail, thus invoking the second clause of not() which would then succeeds), or in their absence directly into the second clause of not().

ASSIGN(<variable>,<term>)

ASSIGN() is the nearest thing that Prolog has to an assignment statement (ie '=' of BASIC or 'C', or ':=' of PASCAL). The first argument is a variable, the second any Prolog term. The ASSIGN() always succeeds, and bind the variable to the term. Note that the variable is not dereferenced, ie if it is bound to another value before the ASSIGN(), the old value is replaced by the new one. ASSIGN() is a useful trick for holding values during backtracking. The rule is that the variable always behaves as if its current value was the one it was given on the first ASSIGN, even if its value changes after the first ASSIGN and before backtracking. Therefore a value stored by an ASSIGN() can be held when backtracking returns to a point prior to that at which the latest value was assigned (if the variable had taken its value in any other way, it would become unbound on backtracking). Use of ASSIGN() is not recommended and programmers coming to Prolog should not be tempted to try to use Prolog variables in the same way they would BASIC variables (say). ASSIGN() has one extremely useful application which justifies including it in a Prolog implementation - the isall() predicate. The isall() predicate is invoked by a goal of the form:

```
isall(<variable>,<term>,<goal)
```

This is interpreted as: "<variable> is a list, terminated by nil, of all terms <term> such that <goal> is satisfied". The program for isall() is:

```
isall(?x,?y,?p.?args):-
      ASSIGN(?z,nil),
      not(and(?p.?args,and(ASSIGN(?z,?y.?z),FAIL()))),
      EQ(?x,?z);

not(?p.?args):-
      ?p.?args,
      /,
      FAIL();

not(?p.?args):-;

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

There are many pitfalls in the use of ASSIGN() and the programmer is urged not to use it other than in the form suggested above. The isall() predicate works by setting a local variable to nil, then solving the goal passed as an argument. The value of ?y found for each solution is prefixed to the list, and a FAIL() is used to force backtracking until all solutions have been found. The not() ensures that control then proceeds to the EQ() to bind the output variable to the list built up during bactracking. Note that the operation of isall() is such that the terms appear in the list in the reverse order to that in which they are found. isall() is a very useful function because it allows many operations that would normally require explicit looping (normally through recursion) to be written very concisely without explicit loops of any kind. As a simple example of an application of isall(), sol-count() counts the number of solutions to a predicate:

```
sol-count(?n,?p.?argp):-
```

```
        isall(?x,nil,?p.?argp),
        length(?x,?n);

length(nil,0):-;
length(?x.?y,?n):-
        length(?y,?m),
        add(?m,1,?n);
```

As another example, suppose we have a customer() predicate which stores customer names:

```
customer("Bloggco"):-;
customer("Zoomco"):-;
customer("Brillco"):-;
customer("Goco"):-;
```

Then a list of all customers is obtained by the simple query:

```
P>isall(?x,?y,customer(?y));
<?y=?y,?x="Goco"."Brillco"."Zoomco"."Bloggco".nil>
```

Note that the 'dummy' variable ?y is left unbound after isall() has been evaluated - a careful look at the definition of isall() will show why this is the case.

FAIL()

FAIL() is a system predicate which has no effect other than to fail. FAIL() behaves just like an undefined predicate, but being a system predicate is slightly faster than deliberately invoking an undefined predicate when a programmed fail is required. Furthermore, being a system predicate, it cannot be overridden by an inadvertent user-defined FAIL() predicate. FAIL() is frequently used in conjunction with the cut, the simplest and perhaps most useful example being in the definition of not() (see /). In this program, FAIL() is used to deliberately fail the predicate after the successful execution of a goal passe to not() as an argument. Here, as in many other uses of FAIL(), the cut is necessary to prevent unwanted backtracking into alternative clauses. Us of FAIL() is generally frowned upon by Prolog programmers, as is the use of the cut. There are various theoretical reasons for considering FAIL() to be outside the scope of Prolog, but it is nevertheless essential to many Prolog programs. As a rule of thumb, use of FAIL() should be restricted to its implicit use through not() - most explicit FAILs can be handled more elegantly by use of not().

FAIL() can of course be used to deliberately force backtracking into alternative clauses. For example:

```
a(1):-
      w(hello),
      FAIL();

a(2):-
      w(world);
```

With this definition, the goal:

```
a(1)
```

causes the message:

```
helloworld
```

to be written to the output channel, but the goal:

```
a(2)
```

only writes:

```
world
```

Such a construction can be used to force a loop:

```
loop():-
      once-around();

loop():-
      loop();
```

In this case, the goal:

```
loop(),FAIL();
```

causes once-around() to be executed an indefinite number of times, since the FAIL() causes backtracking into the second clause of loop(), which starts loop() again at the first clause. This is not a good way of programming loops, but illustrates a phenomenon which often contributes to inefficiency in Prolog programs, namely thrashing, where control bounces backwards and forwards, in this case indefinitely between loop() and FAIL() in the goal.

QUIT()

QUIT() causes an immediate end to the Prolog interpreter. In interactive mode, QUIT() is used to terminate the interpreter and return to the invoking program, usually the OS9 shell, eg:

```
P>QUIT();


OS9:
```

If QUIT() is executed during the evaluation of a goal, processing ceases immediately. QUIT() never returns or succeeds or fails. If any file are open on the input or output channels at the time of the QUIT(), they are automatically closed and their buffers flushed. QUIT() causes the interpreter to return an error code of zero to its parent process (ie no error). When the interpreter terminates using QUIT(), all the memory associated with Prolog data structures and evaluation stacks (which are allocated dynamically during the execution of the program) is returned to OS9. It is not always necessary to use QUIT() to exit from Prolog, since Prolog always exits when it hits end-of-file reading from its standard input in interactive mode. Therefore a series of interactive goals stored in a disk file does not need to end with a QUIT(). QUIT() may be used to take user-directed exits, for instance in a menu, eg:

```
menu-action(1):-
      load();

menu-action(2)):-
      save();

menu-action(3):-
      edit();

menu-action(4):-
      configure();

menu-action(5):-
      QUIT();

main-loop():-
      menu-display(),
      get-user-option(?n),
      menu-action(?n),
      main-loop();
```

This skeleton program illustrates a typical menu-driven system implemented in Prolog. The general scheme is easily expanded, for instance, an error-trapping clause can be added to the menu-action() predicate:

```
menu-action(?n):-
      w("Invalid choice - retry@13");
```

If the option given by the user is not in the range 1 to 5, the error message is displayed and the menu loop restarted, giving the user an opportunity to rectify his error.

Note that in practice, a cut is usually required in each of the clauses of menu-action(). This prevents the choice stack growing out of control, and also unwanted backtracking (for instance to the error clause), which is a common problem with Prolog programs, especially when they become significantly complex, and one which is difficult to locate.

Type predicates

CON(<term>)

CON() is used to test the type of its argument. CON() succeeds if its argument is a constant, otherwise it fails. In the sense that CON() fails if its argument is an unbound variable, CON() is, strictly speaking, non-logical, because the variable may later be discovered, upon binding, to be a constant after all, thus invalidating the failure of CON(). Nevertheless, CON() is a useful and fairly harmless predicate. For example:

```
P>CON("hello, world");
<>

P>CON(Animal);
<>

P>CON(vegetable);
<>

P>CON("123");
<>

P>CON(123);
P>CON(?what);
P>CON(Animal.Vegetable);
```

Note that the last three examples all fail. CON() is most frequently used as the first predicate invoked for a clause in a predicate having several clauses, one for each data type. It is frequently followed by a cut to prevent backtracking into clauses which are known (following the success of CON()) not to be applicable. As an example, CON() is frequently used in formatted output predicates where each clause selects a particular data type and prints it in an appropriate format, for instance:

```
printf(?x):-
      CON(?x),
      /,
      w(">>"),
      w(?x),
      w("<<");
```

might be one clause in a formatted printing predicate for dealing with constants. Then a goal such as:

```
printf(hello)
```

results in the message:

```
>>hello<<
```

being sent to the current output channel. Of course no formatted printing predicate is this simple, and such a predicate would normally take additional argument to specify the precise format, for example:

```
printf(?x,?format):-
```

```
CON(?x),
/,
pad(?x,?format,?x-pad),
w(?x);
```

where pad() might be a predicate for padding ?x with spaces to meet the field width specified by ?format, giving the result as ?x-pad.

LST(<term>)

LST() is used to test the type of its argument. LST() succeeds if its argument is a list (including nil) otherwise it fails. In the sense that LST() fails if its argument is an unbound variable, LST() is, strictly speaking, non-logical, because the variable may later be discovered, upon binding, to be a list or nil after all, thus invalidating the failure of LST(). Nevertheless, LST() is a useful and fairly harmless predicate. For example:

```
P>LST(hello.world);
<>

P>LST(Animal.Vegetable.nil);
<>

P>LST(nil);
<>

P>LST(1.2.3);
<>

P>LST(123);
P>LST(?what);
P>LST("hello.world");
```

Note that the last three examples all fail. LST() is most frequently used as the first predicate invoked for a clause in a predicate having several clauses, one for each data type. It is frequently followed by a cut to prevent backtracking into clauses which are known (following the success of LST()) not to be applicable. As an example, LST() is frequently used in formatted output predicates where each clause selects a particular data type and prints it in an appropriate format, for instance:

```
printf(?x):-
      LST(?x),
      /,
      w("["),
      listprint(?x),
      w("]");

listprint(nil):-
      w("nil");

listprint(?x.?y):-
      printf(?x),
      w('.'),
      printf(?y);
```

might be one clause in a formatted printing predicate for dealing with lists. Then a goal such as:

```
printf(hello.world)
```

results in the message:

```
[hello.world]
```

being sent to the current output channel. Of course no formatted printing predicate is this simple, and such a predicate would normally take additional argument to specify the precise format, for example:

```
printf(?x,?format):-
      LST(?x),
      /,
      listprint(?x,?format);
```

where listprint() has the job of splitting up ?x and ?format into corresponding parts and printing each one with the appropriate format (using a recursive call to printf()).

NUM(<term>)

NUM() is used to test the type of its argument. NUM() succeeds if its argument is a number otherwise it fails. In the sense that NUM() fails if its argument is an unbound variable, NUM() is, strictly speaking, non-logical, because the variable may later be discovered, upon binding, to be a number after all, thus invalidating the failure of NUM(). Nevertheless, NUM() is a useful and fairly harmless predicate. For example:

```
P>NUM(123);
<>

P>NUM(0);
<>

P>NUM(-32);
<>

P>NUM(-1000);
<>

P>NUM("123");
P>NUM(?what);
P>NUM(Animal.Vegetable);
```

Note that the last three examples all fail. NUM() is most frequently used as the first predicate invoked for a clause in a predicate having several clauses, one for each data type. It is frequently followed by a cut to prevent backtracking into clauses which are known (following the success of NUM()) not to be applicable. As an example, NUM() is frequently used in formatted output predicates where each clause selects a particular data type and prints it in an appropriate format, for instance:

```
printf(?x):-
      NUM(?x),
      /,
      w("%"),
      w(?x);
```

might be one clause in a formatted printing predicate for dealing with numbers. Then a goal such as:

```
printf(123)
```

results in the message:

```
%123
```

being sent to the current output channel. Of course no formatted printing predicate is this simple, and such a predicate would normally take additional argument to specify the precise format, for example:

```
printf(?x,?format):-
      NUM(?x),
      /,
      pad(?x,?format,?x-pad),
```

```
w(?x);
```

where pad() might be a predicate for padding the string representation of the number ?x with spaces to meet the field width specified by ?format, giving the result as ?x-pad.

VAR(<term>)

VAR() is used to test the type of its argument. VAR() succeeds if its argument is an unbound variable, otherwise it fails. In the sense that VAR() succeeds if its argument is an unbound variable, VAR() is, strictly speaking, non-logical, because the variable may later be bound to a constant, number or list, thus invalidating the success of VAR(). Nevertheless, VAR() is a useful and fairly harmless predicate. For example:

```
P>VAR(?x);
<>

P>VAR(?Animal);
<>

P>VAR(?vegetable);
<>

P>VAR("123");
<>

P>VAR(123);
P>VAR(what);
P>VAR(Animal.Vegetable);
```

Note that the last three examples all fail. VAR() is most frequently used as the first predicate invoked for a clause in a predicate having several clauses, one for each data type. It is frequently followed by a cut to prevent backtracking into clauses which are known (following the success of VAR()) not to be applicable. As an example, VAR() is frequently used in formatted output predicates where each clause selects a particular data type and prints it in an appropriate format, for instance:

```
printf(?x):-
      VAR(?x),
      /,
      w(?x),
      w("(unknown)");
```

might be one clause in a formatted printing predicate for dealing with variables. Then a goal such as:

```
printf(?var)
```

results in the message:

```
?var(unknown)
```

being sent to the current output channel. Of course no formatted printing predicate is this simple, and such a predicate would normally take additional argument to specify the precise format, for example:

```
printf(?x,?format):-
      VAR(?x),
      /,
      pad(?x,?format,?x-pad),
      w(?x);
```

where pad() might be a predicate for padding the variable name ?x (note that this is not necessarily ?x, even if the argument is unbound - ?x could be bound to a different unbound variable, in which case it would be the name of this variable which would be printed) with spaces to meet the field width specified by ?format, giving the result as ?x-pad.

Prolog syntax definition

A formal description of the syntax for Prolog programs and queries is given using Bachus-Naur notation. In this notation, the syntax is specified as a set of rules or productions. A production has one symbol (strictly a metasymbol) on the left and a set of alternatives separated by '|' symbols on the right. Each alternative is a list of metasymbols or terminal symbols. Terminal symbols are those which are actually read from the input stream and are denoted by enclosing quotes. In some cases, one of the choices may be empty, indicating that the metasymbol on the left of the production if optional. In this case, no symbols appear in the entry for that choice, for example:

```
list:   | element list
```

which indicates that a 'list' consists of zero or more 'elements' (note the use of recursive definitions to indicate repeated symbols).

The syntax analyser ignores whitespace (tabs, newlines and space characters) except where they serve to delimit lexical tokens, and where they appear in quoted strings (in fact a newline will terminate a quoted string). For example, the sequence "abc123" consists of a single token, but "abc 123" consists of two tokens, "abc" followed by "123".

All grammars specified using Bachus-Naur notation have a special symbol, which is the point from which all possible sentences of the grammar can be derived by expanding productions and also the point from which parsing begins. In the grammar below, 'session' is the distinguished symbol.

The system predicate rterm() reads a term from the input stream using the definition of 'term' below. In effect, rterm() calls a parser for a subset of the full Prolog grammar, namely that subset for which 'term' is the distinguished symbol.

```
session:      | query session

query: rule | goal

rule: predicate ':-' term-list ';'

goal: predicate ',' term-list ';'

term-list:    | term | term ',' term-list

term: '/' | '(' term ')' | atom | atom '.' term

atom: '<' term-list '>' | number | name | predicate | rule

predicate: name '(' term-list ')'

name: constant | variable

constant: '"' print-list '"' | con-list
```

```
con-list: con-first con-rest

con-first: alpha-char

con-rest:    | con-rest-char con-rest

con-rest-char: alpha-num | '-' | '@' | '"'

print-list:    | print-char print-list

print-char: alpha-num | special-char

variable: '?' con-rest

number: sign integer

sign:   | '-'

integer: digit | digit integer

alpha-num: alpha-char | digit

alpha-char: upper-char | lower-char

digit: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'


  special-char: ' ' | '!' | '"' | '#' | '$' | '%' | '&' | ''' | '(' | ')' | '*' |
     '+' | ',' | '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' | '?' | '@' | '[' |
     '\' | ']' | '^' | '_' | '|' | '{' | '|' | '}' | '~'

  upper-char: 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' |
     'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' |
     'Y' | 'Z'

  lower-char: 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' |
     'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
     'y' | 'z'
```