11.1 Database Manipulation

Prolog has four database manipulation commands: assert, retract, asserta, and assertz. Let's see how these are used. Suppose we start with an empty database. So if we give the command:

```
?- listing.
```

then Prolog will simply respond yes; the listing (of course) is empty.

Suppose we now give this command:

```
?- assert(happy(mia)).
```

This succeeds (assert/1 commands always succeed). But what is important is not that it succeeds, but the side-effect it has on the database. For if we now give the command

```
?- listing.
```

we get:

```
happy(mia).
```

That is, the database is no longer empty: it now contains the fact we asserted.

Suppose we then made four more assert commands:

```
?- assert(happy(vincent)).
yes

?- assert(happy(marcellus)).
yes

?- assert(happy(butch)).
yes

?- assert(happy(vincent)).
yes
```

and then ask for a listing:

```
?- listing.

happy(mia).
happy(vincent).
happy(marcellus).
happy(butch).
happy(vincent).
yes
```

All the facts we asserted are now in the knowledge base. Note that happy(vincent) is in the knowledge base twice. As we asserted it twice, this seems sensible.

The database manipulations we have been making have changed the meaning of the predicate happy/1 . More generally, database manipulation commands give us the ability to change the meaning of predicates while we are running programs. Predicates whose definitions change during run-time are called dynamic predicates, as opposed to the static predicates that we have previously dealt with. Most Prolog interpreters insist that we explicitly declare the predicates that we wish to be dynamic. We will soon examine an example involving dynamic predicates, but let's first complete our discussion of the database manipulation commands.

So far we have only asserted facts into the database, but we can also assert new rules. Suppose we want to assert the rule that everyone who is happy is naive. That is, suppose we want to assert that:

```
naive(X):-  happy(X).
```

We can do this as follows:

```
assert(  (naive(X):-  happy(X))  ).
```

Note the syntax of this command: the rule we are asserting is enclosed in a pair of brackets . If we now ask for a listing we get:

```
happy(mia).
happy(vincent).
happy(marcellus).
happy(butch).
happy(vincent).

naive(A):-
     happy(A).
```

Now that we know how to assert new information into the database, we should also learn how to remove information when we no longer need it. There is an inverse predicate to assert/1 , namely retract/1 . For example, if we carry straight on from the previous example by giving the command:

```
?-  retract(happy(marcellus)).
```

and then list the database, we get:

```
happy(mia).
happy(vincent).
happy(butch).
happy(vincent).

naive(A)  :-
      happy(A).
```

That is, the fact happy(marcellus) has been removed.

Suppose we go on further, and say

```
?-  retract(happy(vincent)).
```

and then ask for a listing. We get:

```
happy(mia).
happy(butch).
happy(vincent).

naive(A)  :-
      happy(A).
```

Note that the first occurrence of happy(vincent) , and only the first occurrence, was removed.

To remove all of our assertions contributing to the definition of the predicate happy/1 we can use a variable:

```
?-  retract(happy(X)).

X  =  mia  ;
```

```
X = butch ;

X = vincent ;
no
```

A listing reveals that the database is now empty, except for the rule naive(A) :- happy(A) .

```
?- listing.
naive(A) :-
        happy(A).
```

If we want more control over where the asserted material is placed, there are two variants of assert/1 , namely:

1. assertz. Places asserted material at the end of the database.
2. asserta. Places asserted material at the beginning of the database.

For example, suppose we start with an empty database, and then we give the following command:

```
assert( p(b) ), assertz( p(c) ), asserta( p(a) ).
```

Then a listing reveals that we now have the following database:

```
?- listing.

p(a).
p(b).
p(c).
yes
```

Database manipulation is a useful technique. It is especially useful for storing the results to computations, so that if we need to ask the same question in the future, we don't need to redo the work: we just look up the asserted fact. This technique is called memoisation, or caching, and in some applications it can greatly increase efficiency. Here's a simple example of this technique at work:

```
:- dynamic lookup/3.

add_and_square(X,Y,Res):-
     lookup(X,Y,Res), !.

add_and_square(X,Y,Res):-
     Res is (X+Y)*(X+Y),
     assert(lookup(X,Y,Res)).
```

What does this program do? Basically, it takes two numbers X and Y, adds X to Y, and then squares the result. For example we have:

```
?- add_and_square(3,7,X).

X = 100
yes
```

But the important point is: how does it do this? First, note that we have declared lookup/3 as a dynamic predicate. We need to do this as we plan to change the definition of lookup/3 during run-time. Second, note that there are two clauses defining add_and_square/3 . The second clause performs the required arithmetic calculation and asserts the result to the Prolog database using the predicate lookup/3 (that is, it caches the result). The first clause checks the Prolog database to see if the calculation has already been made in the past. If it has been, the program simply returns the result, and the cut prevents it from entering the second clause.

Here's an example of the program at work. Suppose we give Prolog another query

```
?-   add_and_square(3,4,Y).

Y = 49
yes
```

If we now ask for a listing we see that the database now contains

```
lookup(3, 7, 100).
lookup(3, 4, 49).
```

Should we later ask Prolog to add and square 3 and 4, it wouldn't perform the calculations again. Rather, it would just return the previously calculated result.

Question: how do we remove all these new facts when we no longer want them? After all, if we give the command

```
?- retract(lookup(X,Y,Z)).
```

Prolog will go through all the facts one by one and ask us whether we want to remove them! But there's a much simpler way. Simply use the command

```
?- retractall(lookup(_,_,_)).
```

This will remove all facts about lookup/3 from the database.

To conclude our discussion of database manipulation, a word of warning. Although it is a useful technique, database manipulation can lead to dirty, hard to understand, code. If you use it heavily in a program with lots of backtracking, understanding what is going on can be a nightmare. It is a non-declarative, non logical, feature of Prolog that should be used cautiously.