```
?- density(china,X).
X=200
yes
?- density(turkey,X).
no
```

In the first question, the `X=200` is Prolog's answer, meaning 200 people per square mile. The second question failed, because the population of Turkey could not be found in our example database.

Depending on what computer you use, various arithmetic operators can be used on the right-hand side of the "`is`" operator. All Prolog systems, however, will have:

`X + Y`	the sum of X and Y
`X - Y`	the difference of X and Y
`X * Y`	the product of X and Y
`X / Y`	the quotient of X divided by Y
`X mod Y`	the remainder of X divided by Y.

This list together with the above list of comparison operators should tell you nearly all you need for doing simple arithmetic problems. Of course, Prolog is mainly intended for non-numerical purposes, so arithmetic facilities are not as important as in other computer languages.

2.6 Summary of Satisfying Goals

Prolog performs a task in response to a *question* from the programmer (you). A question provides a *conjunction* of goals to be *satisfied*. Prolog uses the known *clauses* to satisfy the goals. A fact can cause a goal to be satisfied immediately, whereas a rule can only reduce the task to that of satisfying a conjunction of *subgoals*. However, a clause can only be used if it *matches* the goal under consideration. If a goal cannot be satisfied, *backtracking* will be initiated. Backtracking consists of reviewing what has been done, attempting to *re-satisfy* the goals by finding an alternative way to satisfying them. Furthermore, if you are not content with an answer to your question, you can initiate backtracking yourself by typing a semicolon when Prolog informs you of a solution. In this section, we present a diagrammatic notation for showing how and when Prolog attempts to satisfy and re-satisfy goals.

2.6.1 Successful satisfaction of a conjunction of goals

Prolog attempts to satisfy the goals in a conjunction, whether they appear in a rule body or in a question, in the order they are written (left to right). This means that Prolog will not attempt to satisfy a goal until its neighbour on the left has been satisfied. And, when it has been satisfied, Prolog will attempt to satisfy its neighbour on the right. Consider the following simple program about family relations:

```
female(mary).

parent(C,M,F) :- mother(C,M), father(C,F).

mother(john,ann).
mother(mary,ann).

father(mary,fred).
father(john,fred).
```

Let us look at the sequence of events that leads to the question:

```
?- female(mary), parent(mary,M,F), parent(john,M,F).
```

being answered. This question is to ascertain whether `mary` is a sister of `john`. To do this Prolog needs to satisfy the following sequence of subgoals shown in Figure 2.1. We represent goals as boxes distributed down the page. An arrow starting from the top of the page indicates which goals have already been satisfied. Boxes that the arrow has passed through indicate goals that have been satisfied. Boxes that lie below the arrowhead represent goals that Prolog has not yet considered. As a program runs, the arrow moves up and down the page as Prolog turns its attention to the various goals. We call this the *flow of satisfaction*. In the example, the arrow starts at the top of the page, as shown above. It will extend downwards, moving

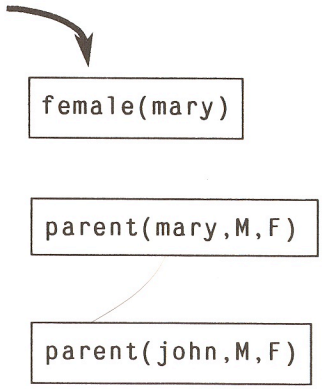

Figure 2.1 A sequence of subgoals not yet satisfied.

ugh the three boxes as the three goals are satisfied. So the final ıtion will be as shown in Figure 2.2. Notice that values have now been d for the variables M and F. This diagram shows the coarse structure of t has happened, but it fails to show *how* these three goals were satisfied. an show this by putting more detail inside the boxes. Let us concentrate ow the second goal is satisfied. Satisfying a goal involves searching the base for a *matching* clause, then marking the place in the database, and satisfying any subgoals.

We can show this for the second goal by indicating in the `parent` box ch clause was chosen and which subgoals had to be satisfied. The clause en is shown by a number in brackets, here `(1)`. This number indicates ch clause *out of the set of clauses for the appropriate predicate* has been sen. So the number 1 indicates that the first clause for the predicate has ı chosen. This is enough information to mark the place in the database. subgoals are shown in small boxes inside the box for the goal. At the ıt when the `parent` clause has been chosen, the situation looks like ure 2.3.

The arrow has entered the `parent` box and passed through the ckets indicating that a clause has been chosen. The clause has introduced subgoals, involving `mother` and `father`. At this point, the arrow must s through these two smaller boxes, emerge from the current `parent` box l then pass through the second `parent` box in order for the question to ceed. When the arrow passes through the smaller boxes, the same steps of osing a clause and satisfying the clause's subgoals must be performed. In s example, both of these goals succeed by finding facts in the database. So

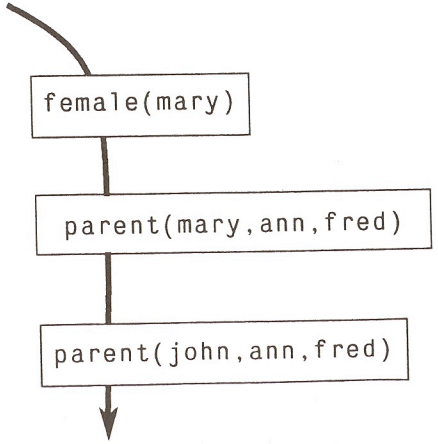

Figure 2.2 The sequence of subgoals has been satisfied. Note that variables have been instantiated.

Figure 2.4 shows a more detailed picture of the situation when the question succeeds

Note that to be precise we should have shown the details of how the goals `female(mary)` and `parent(john,ann,fred)` were satisfied. However, this would have been too much detail to fit onto one page. This example shows the general pattern of how Prolog attempts to satisfy goals in a case where the conjunction of goals succeeds. The arrow moves down the page, passing through the boxes in turn. When it enters a box, a clause is chosen and its position marked. If the clause matches the goal and the clause is a fact, then the arrow can leave the box (this happened for the `mother` and `father` goals. On the other hand, if the clause matches the goal and the clause is a rule then new boxes are created for the subgoals and the arrow must then pass through all of these before it can leave the original box.

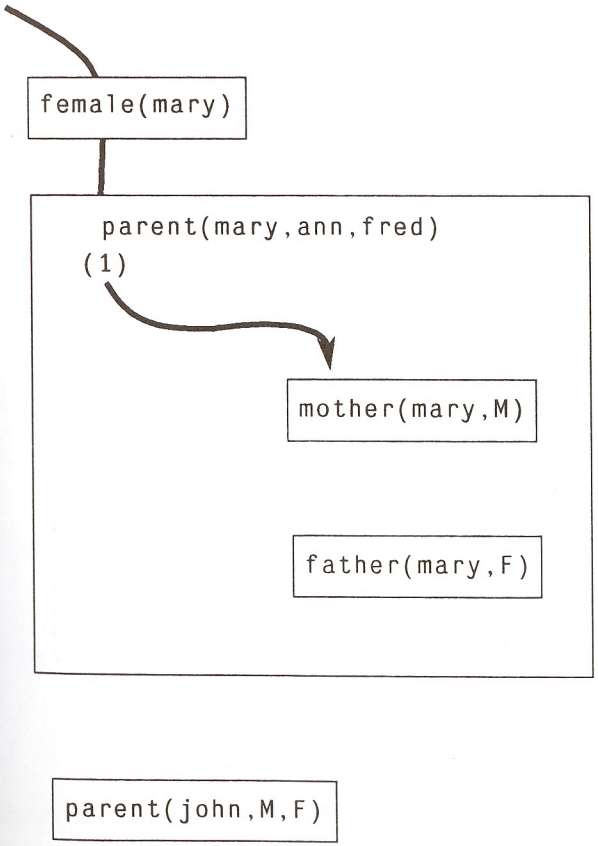

Figure 2.3 The number (1) indicates that the first clause for the predicate has been chosen. The subgoals are shown in small boxes inside the box for the goal.

2.6.2 *Consideration of goals in backtracking*

When a failure is generated (because all the alternative clauses for a goal have been tried, or because you type a semicolon), the "flow of satisfaction" passes *back* along the way it has come. This involves retreating back into boxes that have previously been left in order to *re-satisfy* the goals. When the arrow gets back to a place where a clause was chosen (represented by a number in brackets), Prolog attempts to find an alternative clause for the appropriate goal. First, it makes uninstantiated all variables that had been instantiated in the course of satisfying the goal. Then, it searches on in the database from where the place-marker was put. If it finds another matching possibility, it marks the place, and things continue as in Section 2.6.1 above.

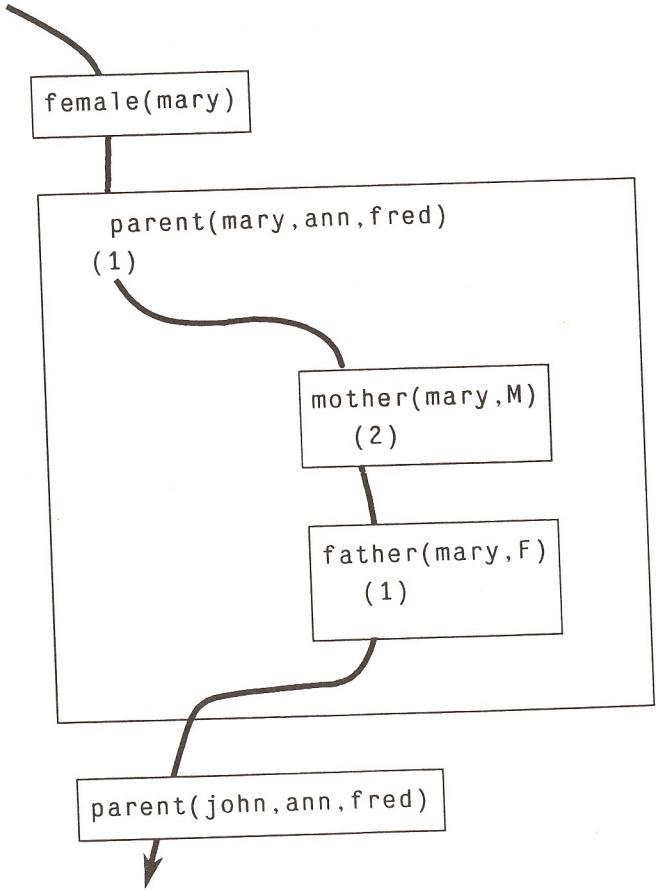

Figure 2.4 The question has succeeded.

Note that work on any goals "below" this (even if such goals w
under the previous alternative) will always start from scratch.
try to satisfy, and not to re-satisfy them. If no other matching pos
be found, the goal fails, and the arrow retreats further until
another place-marker.

In our example, if the goal `parent(john,ann,fred)`
arrow would retreat upwards from the `parent(john,ann,fre`
re-enter the `parent(mary,ann,fred)` box from below, attem
satisfy this goal. It would then retreat further, re-en
`father(mary,fred)` box and trying to re-satisfy this goal, a
Figure 2.5.

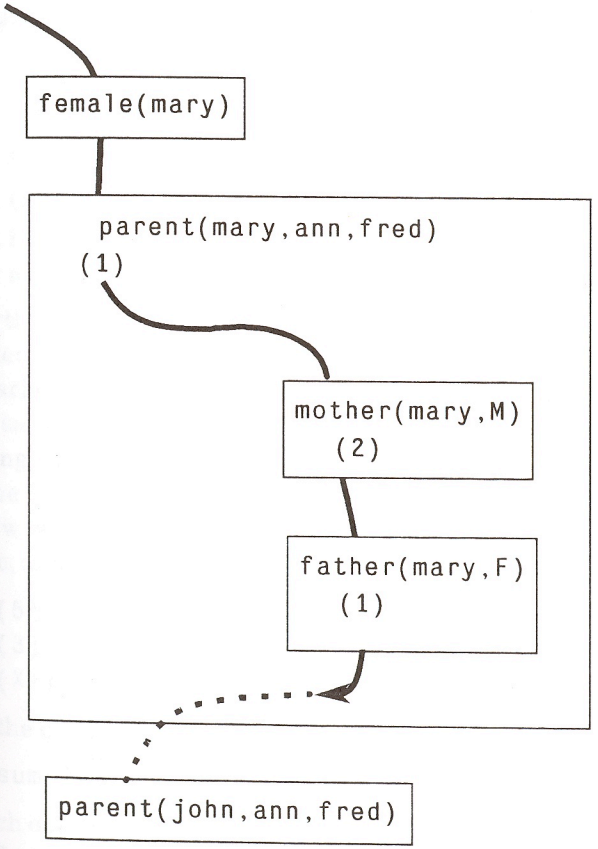

Figure 2.5 What happens if a goal fails.

father goal was chosen. First of all, all variables that became instantiated as the result of using this clause are set back to uninstantiated. This means that F becomes uninstantiated again. Then Prolog looks through the database, starting after the first father clause (the one marked), trying to find an alternative clause for this goal. Assuming that mary has only one father, this will not succeed. So the arrow will have to retreat further. It retreats upwards, out of the father(mary,F) box (this goal has failed) and back into the mother(mary,ann) box (to attempt to re-satisfy this goal). We get the situation shown in Figure 2.6.

We can see from these examples the general pattern of how goals are reconsidered in backtracking. When a goal fails, the arrow retreats upwards

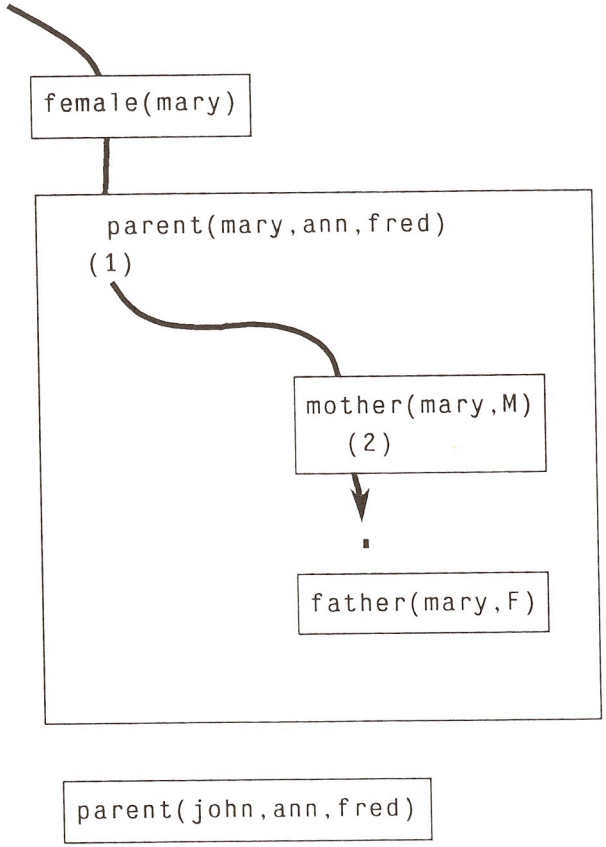

Figure 2.6 Attempting to re-satisfy a goal.

out of the box for the failing goal and back into the box for the goal above The arrow continues retreating until it reaches a place marker. Al variables that were instantiated as a result of the previous choice of claus are reset to uninstantiated. Then Prolog searches the database for a claus after the place marker. If it finds a clause that matches the goal, then a ne place mark is recorded, boxes for the subgoals are created and the arro starts moving downwards again. Otherwise, the arrow continues to retrea upwards, in search of another place marker.

2.6.3 Matching

The rules for deciding whether a goal matches the head of a use of a claus are as follows. Note that in the use of a clause, all variables are initiall uninstantiated.

- An uninstantiated variable will match any object. As a resul that object will be what the variable stands for.
- Otherwise, an integer or atom will match only itself.
- Otherwise, a structure will match another structure with th same functor and number of arguments, and all the correspondin arguments must match.

A noteworthy case in matching is one in which two uninstantiated variable are matched together. In this case, we say that these variables *share*. Tw sharing variables are such that as soon as one is instantiated, so is the oth (with the same value). If you have noticed a similarity between matchin and making arguments equal (Section 2.4), then you are correct. This because the "=" predicate attempts to make its arguments equal by matchin them. Now we can bring together what we have discussed about operator arithmetic, and matching. Suppose the following facts are in the database:

```
sum(5).
sum(3).
sum(X+Y).
```

Consider the question

```
?- sum(2+3).
```

Now, which one of the facts above will match with the question? If you thin it is the first one, then you should go back and read about structures an operators. In the question the argument of the sum structure is a structu having the plus sign as its functor, and having the 2 and 3 as its component In fact, the goal shown will match with the third fact, instantiating X to and Y to 3. On the other hand, if we actually wanted to compute a sum, w would use the "is" predicate. We would write