Sec. 1.11. Negative Goals ("Not")  

?- non_parent(elmo,cathy).  
yes  
?- non_parent(sharon,cathy).  
yes  
?- non_parent(charles,cathy).  
yes  

and non_parent fails if its arguments are in fact a parent and his or her child:

?- non_parent(michael,cathy).  
no  
?- non_parent(melody,cathy).  
no  

So far, so good, but what happens if you ask about people who are not in the knowledge base at all?

?- non_parent(donald,achsa).  
yes  

Wrong! Actually, Donald (another of the authors of this book) is the father of Achsa, but FAMILY.PL doesn't know about it. Because the computer can't prove father(donald,achsa) nor mother(donald,achsa), the non_parent query succeeds, giving a result that is false in the real world.  

Here we see a divergence between Prolog and intuitively correct thinking. The Prolog system assumes that its knowledge base is complete (e.g., that there aren't any fathers or mothers in the world who aren't listed). This is called the CLOSED-WORLD ASSUMPTION. Under this assumption, \+ means about the same thing as "not," but without the closed-world assumption, \+ is merely a test of whether a query fails. That's why many Prolog users refuse to call \+ "not," pronouncing it "cannot-prove" or "fail-if" instead.

Note also that a query preceded by \+ never returns a value for its variables. You might think that the query

?- \+ father(X,Y).  

would instantiate X and Y to two people, the first of which is not the father of the second. Not so. To solve \+ father(X,Y), the computer attempts to solve father(X,Y) and then fails if the latter goal succeeds or succeeds if the latter goal fails. In turn, father(X,Y) succeeds by matching a clause in the knowledge base. Therefore, \+ father(X,Y) has to fail, and because it fails, it does not report variable instantiations.

As if this were not enough, the order of subgoals in a query containing \+ can affect the outcome. Let's add the fact

blue_eyed(cathy).  

to the knowledge base. Now look at the results of the following queries:
?- blue_eyed(X), non_parent(X, Y).
   X = cathy
   yes
?- non_parent(X, Y), blue_eyed(X).
   no

The first query succeeds because X gets instantiated to cathy before non_parent(X, Y)
is evaluated, and non_parent(cathy, Y) succeeds because there are no clauses that
list Cathy as a mother or father. But in the second query, X is uninstantiated when
non_parent(X, Y) is evaluated, and non_parent(X, Y) fails as soon as it finds a clause
that matches father(X, Y).

To make negation apply to a compound goal, put the compound goal in parentheses, and be sure to leave a space after the negation symbol. Here’s a whimsical
eample:\(^2\)

\[
\text{blue-eyed_non_grandparent}(X) : - \\
\text{blue_eyed}(X), \\
\text{\(+\)} \text{(parent}(X, Y), \text{parent}(Y, Z)).
\]

That is, you’re a blue-eyed non-grandparent if you are blue-eyed, and you are not
the parent of some person Y who is in turn the parent of some person Z.

Finally, note that \(+\) (with its usual Prolog meaning) can appear only in a query
or on the right-hand side of a rule. It cannot appear in a fact or in the head of a rule.
If you say
\[
\text{\(+ father\text(cathy, michael).}
\]
\% \text{wrong!}

you are not denying that Cathy is Michael’s father; you are merely redefining the
built-in predicate \(+\), with no useful effect. Some Prolog implementations will allow
this, with possibly unpleasant results, while others will display an error message
saying that \(+\) is a built-in predicate and you cannot add clauses to it.

Exercise 1.11.1

Define \text{non_grandparent}(X, Y), which should succeed if X is not a grandparent of Y.

Exercise 1.11.2

Define \text{young_parent}(X), which should succeed if X has a child but does not have any
grandchildren. Make sure it works correctly; consider the case of someone who has two
children, one of whom, in turn, has a child of her own while the other one does not.

1.12. TESTING FOR EQUALITY

Now consider the problem of defining “sibling” (brother or sister). Two people are
siblings if they have the same mother. (They also have the same father, but this
is irrelevant because everyone has both a father and a mother — at least in \text{this}
knowledge base.) So a first approximation is:

\(^2\)Some Prologs will print a warning message that the value of \text{Z} in this clause is never put to any
use. See “Anonymous Variables” (Sec. 1.13).
Sec. 1.12. Testing for Equality

sibling(X,Y) :- mother(M,X), mother(M,Y).

If we put this rule into FAMILY.PL and then ask for all the pairs of siblings known to the computer, we get a surprise:

?- sibling(X,Y).
   X = cathy   Y = cathy
   X = cathy   Y = sharon
   X = sharon  Y = cathy
   X = sharon  Y = sharon (etc.)

Cathy is not Cathy's sibling, yet Cathy definitely has the same mother as Cathy. We need to rephrase the rule: "X is a sibling of Y if M is the mother of X, and M is the mother of Y, and X is not the same as Y."

To express "not the same" we need an equality test: if X and Y are instantiated to the same value, then

\[ X == Y \]

succeeds and, of course,

\[ \neg X == Y \]

fails. The new rule is:

sibling(X,Y) :- mother(M,X), mother(M,Y), \neg X == Y.

With it, we get the desired result:

?- sibling(X,Y).
   X = cathy   Y = sharon
   X = sharon  Y = cathy (etc.)

Wait a minute, you say. That's the same answer twice! We reply: No, it isn't. Remember that, as far as Prolog is concerned, the two conclusions sibling(cathy,sharon) and sibling(sharon,cathy) are separate pieces of knowledge. Both of them are true, so it's entirely correct to get them both.

Here's another example of equality testing. X is an only child if X's mother doesn't have another child different from X. In Prolog:

only_child(X) :- mother(M,X), \neg (mother(M,Y), \neg X == Y).

Note how the negations are nested. Given X, the first step is to find X's mother, namely M. Then we test whether M has another child Y different from X.

There are actually two "equal" predicates in Prolog. The predicate '==' tests whether its arguments already have the same value. The other equality predicate, '=', attempts to unify its arguments with each other, and succeeds if it can do so. Thus, you can use it not only to test equality, but also to give a variable a value: X = a will unify X with a. With both arguments instantiated, '=' and '==' behave exactly alike.

It's a waste of time to use an equality test if you can do the same job by simply putting a value in an argument position. Suppose, for instance, you want to define a predicate parent_of_cathy(X) that succeeds if X is a parent of Cathy. Here is one way to express it:
parent_of_cathy(X) :- parent(X,Y), Y = cathy.  % poor style

That is: first find a person Y such that X is a parent of Y, then check whether Y is Cathy. This involves an unnecessary step, since we can get the same answer in a single step with the rule:

parent_of_cathy(X) :- parent(X,cathy). % better style

However, '=' and '==' are often necessary in programs that perform input from the keyboard or a file during the computation. We can have goals such as:

?- read(X), write(X), X = cathy.

This means: Instantiate X to a value read in from the keyboard, then write X on the screen, then test whether X equals cathy. It is necessary to use '=' or '==' here because we cannot predict what value X will have, and we don’t want the computation to fail before printing X out. We will deal with input and output in Chapter 2.

Exercise 1.12.1

Does FAMILY.PL list anyone who satisfies only_child as defined in this section? Explain why or why not.

Exercise 1.12.2

Can a query such as ‘?- only_child(X).’ retrieve a value for X? Explain why or why not. If necessary, add an instance of an only child to the knowledge base in order to test this.

Exercise 1.12.3

From the information in FAMILY.PL, can you tell for certain who is married to whom? Explain why or why not.

Exercise 1.12.4

Add to FAMILY.PL the definitions of brother, sister, uncle, and aunt. Verify that your predicate definitions work correctly. (Hint: Recall that you have two kinds of uncles: the brothers of your parents, and the husbands of your aunts. You will need to add facts to specify who is male, who is female, and who is married to whom.)

1.13. ANONYMOUS VARIABLES

Suppose we want to find out whether Hazel is a mother but we don’t care whose mother she is. We can express the query this way:

?- mother(hazel, _).

Here the underscore mark stands for an ANONYMOUS VARIABLE, a special variable that matches anything, but never takes on a value. The values of anonymous variables are not printed out in response to a query. More importantly, successive anonymous variables in the same clause do not take on the same value; they behave as if they were different variables.
Sec. 1.14. Avoiding Endless Computations

You should use an anonymous variable whenever a variable occurs only once in a clause and its value is never put to any use. For example, the rule

\[
is\_a\_grandmother(X) :- \text{mother}(X,Y), \text{parent}(Y,Z).
\]

is exactly equivalent to

\[
is\_a\_grandmother(X) :- \text{mother}(X,Y), \text{parent}(Y,\_).
\]

but is less work for the computer because no value need be assigned to the anonymous variable. Here \(X\) and \(Y\) cannot be replaced with anonymous variables because each of them has to occur in two places with the same value.

Exercise 1.13.1

Modify \texttt{blue\_eyed\_non\_grandparent} (p. 22) by putting an anonymous variable in the appropriate place.

Exercise 1.13.2

Why isn't the following a proper definition of \texttt{grandparent}?

\[
\text{grandparent}(G,C) :- \text{parent}(G,\_), \text{parent}(\_,C).
\]

\% wrong!

1.14. AVOIDING ENDLESS COMPUTATIONS

Some Prolog rules, although logically correct, cause the computation to go on endlessly. Suppose, for example, we have the following knowledge base:

\[
\text{married(michael, melody). \ \ \ [1]}
\]
\[
\text{married(greg, crystal).}
\]
\[
\text{married(jim, elleanor).}
\]

and we want to express the fact that if \(I\) is married to \(Y\), then \(Y\) is married to \(X\). We might try the rule:

\[
\text{married}(I,Y) :- \text{married}(Y,I). \ \ \ \ \ \ \ \ \ [2]
\]

Now suppose we type the query:

\texttt{?- married(don, jane).}

Don and Jane are not in the knowledge base. Accordingly, this query does not match any of the facts in [1], so rule [2] gets invoked and the new goal becomes:

\texttt{?- married(jane, don).}

Again, this does not match any of the facts in [1], so rule [2] is invoked and the new goal becomes:

\texttt{?- married(don, jane).}
Now we're back where we started. The loop continues until the computer runs out of stack space or the user interrupts the computation.

One way to prevent the loop is to have two "married" predicates, one for facts and one for rules. Given the facts in [1], we can define a predicate couple/2 which, unlike married, will take its arguments in either order. The definition is as follows:

\[
\text{couple}(X,Y) :\text{-} \text{married}(X,Y). \\
\text{couple}(Y,X) :\text{-} \text{married}(X,Y).
\]

No loop can arise because no rule can call itself directly or indirectly; so now the query `?- couple(don,jane).` fails, as it should. (Only because they are not in the knowledge base; we hasten to assure readers who know us personally that they are married!)

Sometimes a rule has to be able to call itself in order to express repetition. To keep the loop from being endless, we must ensure that, when the rule calls itself, it does not simply duplicate the previous call.

For an example, let's go back to FAMILIYLPL and develop a definition for "ancestor." One clause is easy, since parents are ancestors of their children:

\[
\text{ancestor}(X,Y) :\text{-} \text{parent}(X,Y). \quad [3]
\]

But the relation of ancestor to descendant can span an unlimited number of generations. We might try to express this with the clause:

\[
\text{ancestor}(X,Y) :\text{-} \text{ancestor}(X,Z), \text{ancestor}(Z,Y). \quad \% \text{ wrong!} \quad [4]
\]

But this causes a loop. Consider the query:

\[?- \text{ancestor(cathy,Who)}.\]

Cathy isn't an ancestor of anyone, and the query should fail. Instead, the computer goes into an infinite loop. To solve the query, the computer first tries clause [3], which fails because it can't satisfy \text{parent(cathy,Who)}. Then it tries clause [4], generating the new goal:

\[?- \text{ancestor(cathy,Z), ancestor(Z,Who)}.\]

In order to solve ancestor(cathy,Z) the computer will do exactly the same things as for ancestor(cathy,Who); in fact, since both Z and Who are uninstantiated, the new goal is in effect the same as the old one. The loop continues over and over until the computer runs out of stack space or the user interrupts the computation.

We can fix the problem by replacing [4] with the following:

\[
\text{ancestor}(X,Y) :\text{-} \text{parent}(X,Z), \text{ancestor}(Z,Y). \quad [5]
\]

This definition will still follow an ancestor-descendant chain down an unlimited number of generations, but now it insists on finding a parent-child relation in each step before calling itself again. As a result, it never gets into endless loops. Many, though not all, transitive relations can be expressed in this way in order to prevent looping.

Finally, and more obviously, Prolog can get into a loop whenever two rules call each other without imposing any additional conditions. For example:
Sec. 1.15. Using the Debugger to Trace Execution

human_being(X) :- person(X).
person(X) :- human_being(X).

The cure in this case is to recognize that the predicates human_being and person are equivalent, and use only one of them.

It is possible to have a computation that never halts but never repeats a query. For instance, with the rules:

positive_integer(1).
positive_integer(X) :- Y is X-1, positive_integer(Y).

the query ‘?- positive_integer(2.5).’ generates the endless sequence:

?- positive_integer(1.5).
?- positive_integer(0.5).
?- positive_integer(-0.5).
?- positive_integer(-1.5).

and so on.

Exercise 1.14.1

Add to FAMILY.PL the predicate related(X,Y) such that X is related to Y if X and Y have any ancestor in common but are not the same person. (Note that when you ask for all the solutions, it will be normal to get many of them more than once, because if two people have one ancestor in common, they also have earlier ancestors in common, several of whom may be in the knowledge base.)

Verify that Michael and Julie are related, Cathy and Danielle are related, but Michael and Melody are not related.

Exercise 1.14.2

Describe how to fix positive_integer so that queries with noninteger arguments would fail rather than looping. (You haven’t been given quite enough Prolog to actually implement your solution yet.)

1.15. USING THE DEBUGGER TO TRACE EXECUTION

Almost all Prolog systems have a DEBUGGER (perhaps it should be called a tracer) modeled on the one in Edinburgh Prolog. The debugger allows you to trace exactly what is happening as Prolog executes a query. Here’s an example (using GEO.PL):

?- spy(located_in/2). (specifies what predicate you are tracing)
yes
?- trace. (turns on the debugger)
yes
?- located_in(toronto,canada).
** (0) CALL: located_in(toronto,canada) ? > (press Return)
** (1) CALL: located_in(toronto,ontario) ? > (press Return)
** (1) EXIT: located_in(toronto,ontario) ? > (press Return)
** (0) EXIT: located_in(toronto,canada) ? > (press Return)
yes
That is: to prove located_in(toronto,canada), the computer first had to prove located_in(toronto,ontario). Here's an example in which the backtracking is more complicated:

?- located_in(What,texas).
** (0) CALL: located_in(.0085,texas) ? > (Return)
** (0) EXIT: located_in(houston,texas) ? > (Return)
What = houston ->;
** (0) REDO: located_in(houston,texas) ? > (Return)
** (0) EXIT: located_in(austin,texas) ? > (Return)
What = austin ->;
** (0) REDO: located_in(austin,texas) ? > (Return)
** (0) FAIL: located_in(.0085,texas) ? > (Return)
no

Here .0085 denotes an uninstantiated variable. Notice that each step is marked one of four ways:

CALL marks the beginning of execution of a query;

REDO means an alternative solution is being sought for a query that has already succeeded once;

EXIT means that a query has succeeded;

FAIL means that a query has failed.

If you keep hitting Return you will see all the steps of the computation. If you hit s (for "skip"), the debugger will skip to the end of the current query (useful if the current query has a lot of subgoals which you don't want to see). If you hit a ("abort"), the computation will stop.

To turn off the debugger, type

?- notrace.

To learn more about what the debugger can do, consult your manual.

Exercise 1.15.1

Use the debugger to trace each of the following queries:

?- located_in(austin,What). (using GEO.PL)
?- parent(michael,cathy). (using FAMILY.PL)
?- uncle(Who,cathy). (using your solution to Exercise 1.12.4)

Describe what happens in each case.

1.16. STYLES OF ENCODING KNOWLEDGE

In FAMILY.PL, we took the relations "mother" and "father" as basic and defined all other relations in terms of them. We could equally well have taken "parent" as basic and used it (along with "male" and "female") to define "mother" and "father":
Sec. 1.16. Styles of Encoding Knowledge

\[ \text{parent(michael, cathy).} \]
\[ \text{parent(melody, cathy).} \]
\[ \text{parent(charles_gordon, michael).} \]
\[ \text{parent(hazel, michael).} \]

\[ \text{male(michael).} \]
\[ \text{male(charles_gordon).} \]

\[ \text{female(cathy).} \]
\[ \text{female(melody).} \]
\[ \text{female(hazel).} \]

\[ \text{father}(X, Y) :- \text{parent}(X, Y), \text{male}(X). \]
\[ \text{mother}(X, Y) :- \text{parent}(X, Y), \text{female}(X). \]

Is this an improvement? In one sense, definitely so, because now the information is broken down into simpler concepts. If you say “mother” you’re asserting parent-hood and femaleness at once; if you say “parent” and “female” separately, you’re distinguishing these two concepts.

Not only that, but now you can tell without a doubt who is female and who is male. In FAMILY.PL, you could deduce that all the mothers are female and all the fathers are male, but you’d still have to state separately that Cathy is female (she’s not a mother).

Which style is computationally more efficient depends on the kinds of queries to be answered. FAMILY.PL can answer “father” and “mother” queries more quickly, since they do not require any inference. But the representation that takes “parent” as basic can answer “parent” queries more quickly.

Unlike other knowledge representation languages, Prolog does not force the knowledge base builder to state information in a particular logical style. Information can be entered in whatever form is most convenient, and then appropriate rules can be added to retrieve the information in a different form. From the viewpoint of the user or higher-level rule issuing a query, information deduced through rules looks exactly like information entered as facts in the knowledge base.

Yet another style is sometimes appropriate. We could use a “data-record” format to encode the family tree like this:

\[ \text{person(cathy, female, michael, melody).} \]
\[ \text{person(michael, male, charles_gordon, hazel).} \]
\[ \text{person(melody, female, jim, eleanor).} \]

Each record lists a person’s name, gender, father, and mother. We then define predicates to pick out the individual pieces of information:

\[ \text{male}(X) :- \text{person}(X, \text{male}, _, _). \]
\[ \text{female}(X) :- \text{person}(X, \text{female}, _, _). \]
\[ \text{father}(\text{Father}, \text{Child}) :- \text{person}(\text{Child}, _, \text{Father}, _). \]
\[ \text{mother}(\text{Mother}, \text{Child}) :- \text{person}(\text{Child}, _, _|_, \text{Mother}). \]
The only advantage of this style is that the multiargument facts are often easy to generate from conventional databases, by simply printing out the data in a format that conforms to Prolog syntax. Human beings find the data-record format much less readable than the other formats, and it is, if anything, slower to process than a set of one- or two-argument facts.

Exercise 1.16.1

Databases often contain names and addresses. Take the names and addresses of two or three people and represent them as a set of Prolog facts. Many different approaches are possible; be prepared to justify the approach you have taken.

1.17. BIBLIOGRAPHICAL NOTES

Two indispensable handbooks of Prolog practice are Sterling and Shapiro (1994) and O'Keefe (1990); the former concentrates on theory and algorithms, the latter on practical use of the language.

There is a large literature on detection and prevention of endless loops in Prolog; see, for example, Smith, Genesereth, and Ginsberg (1986) and Bol (1991). Most loops can be detected, but there may be no way to tell whether the looping computation should succeed or fail.