```
?- ancestor(irvin,elizabeth).
yes
?- ancestor3(irvin,elizabeth).
```

This query invokes a new query

```
ancestor3(Z,elizabeth), parent(irvin,Z).
```

which invokes

```
ancestor3(Z1,elizabeth), parent(Z,Z1), parent(irvin,Z).
```

which invokes

```
ancestor3(Z2,elizabeth), parent(Z1,Z2), parent(Z,Z1), parent(irvin,Z).
```

which invokes ...

The eventual result is a message such as

"Out of local stack during execution; execution aborted."

The problem with this last definition of the ancestor relation is the left recursion with uninstantiated variables in the first clause. If possible, the leftmost goal in the body of a clause should be nonrecursive so that a pattern match occurs and some variables are instantiated before a recursive call is made.

Lists in Prolog

As a special notational convention, a list of terms in Prolog can be represented between brackets: [a, b, c, d]. As in Lisp, the head of this list is a, and its tail is [b, c, d]. The tail of [a] is [ ], the empty list. Lists may contain lists: [5, 2, [a, 8, 2], [x], 9] is a list of five items.

Prolog list notation allows a special form to direct pattern matching. The term [H | T] matches any list with at least one element:

H matches the head of the list, and

T matches the tail.

A list of terms is permitted to the left of the vertical bar. For example, the term [X,a,Y | T] matches any list with at least three elements whose second element is the atom a:

X matches the first element,

Y matches the third element, and

T matches the rest of the list, possibly empty, after the third item.

Using these pattern matching facilities, values can be specified as the intersection of constraints on terms instead of by direct assignment.

Although it may appear that lists form a new data type in Prolog, in fact they are ordinary structures with a bit of "syntactic sugar" added to make them easier to use. The list notation is simply an abbreviation for terms constructed with the predefined "." function symbol and with [ ] considered as a special atom representing the empty list. For example,

[a, b, c] is an abbreviation for .(a, .(b, .(c, [ ])))

[H | T] is an abbreviation for .(H, T)

[a, b | X] is an abbreviation for .(a, .(b, X))

Note the analogy with the relationship between lists and S-expressions in Lisp. In particular, the "list" object [a | b] really represents an object corresponding to a dotted pair in Lisp—namely, .(a,b).

List Processing

Many problems can be solved in Prolog by expressing the data as lists and defining constraints on those lists using patterns with Prolog's list representation. We provide a number of examples to illustrate the process of programming in Prolog.

1. Define last(L,X) to mean "X is the last element of the list L".

   The last element of a singleton list is its only element.

   ```
   last([X], X).
   ```

   The last element of a list with two or more elements is the last item in its tail.

   ```
   last([H|T], X) :- last(T, X).
   ```

   ```
   ?- last([a,b,c], X).
   X = c
   yes
   ?- last([ ], X).
   no
   ```

Observe that the "illegal" operation of requesting the last element of an empty list simply fails. With imperative languages a programmer must test for exceptional conditions to avoid the run-time failure of a program. With logic programming, an exception causes the query to fail, so that a calling program can respond by trying alternate subgoals. The predicate last acts as a generator when run "backward".

```
?- last(L, a).
L = [a];
L = [_5, a];          % The underline indicates system-generated variables.
L = [_5, _9, a];
L = [_5, _9, _13, a] ...
```

The variable H in the definition of last plays no role in the condition part (the body) of the rule; it really needs no name. Prolog allows **anonymous variables**, denoted by an underscore:

```
last([_|T], X) :- last(T, X).
```

Another example of an anonymous variable can be seen in the definition of a father relation:

```
father(F) :- parent(F, _), male(F).
```

The scope of an anonymous variable is its single occurrence. Generally, we prefer using named variables for documentation rather than anonymous variables, although anonymous variables can be slightly more efficient since they do not require that bindings be made.

2. Define member(X,L) to mean "X is a member of the list L".

   For this predicate we need two clauses, one as a basis case and the second to define the recursion that corresponds to an inductive specification.

   The predicate succeeds if X is the first element of L.

   ```
   member(X, [X|T]).
   ```

   If the first clause fails, check if X is a member of the tail of L.

   ```
   member(X, [H|T]) :- member(X,T).
   ```

   If the item is not in the list, the recursion eventually tries a query of the form member(X,[ ]), which fails since the head of no clause for member has an empty list as its second parameter.

3. Define delete(X,List,NewList) to mean
   "The variable NewList is to be bound to a copy of List with all instances of X removed".

   When X is removed from an empty list, we get the same empty list.

   ```
   delete(X,[ ],[ ]).
   ```

   When an item is removed from a list with that item as its head, we get the list that results from removing the item from the tail of the list (ignoring the head).

   ```
   delete(H,[H|T],R) :- delete(H,T,R).
   ```

If the previous clause fails, X is not the head of the list, so we retain the head of L and take the tail that results from removing X from the tail of the original list.

```
delete(X,[H|T],[H|R]) :- delete(X,T,R).
```

4. Define union(L1,L2,U) to mean
   "The variable U is to be bound to the list that contains the union of the elements of L1 and L2".

   If the first list is empty, the result is the second list.

```
union([ ],L2,L2).                                    % clause 1
```

   If the head of L1 is a member of L2, it may be ignored since a union does not retain duplicate elements.

```
union([H|T],L2,U) :- member(H,L2), union(T,L2,U).    % clause 2
```

   If the head of L1 is a not member of L2 (clause 2 fails), it must be included in the result.

```
union([H|T],L2,[H|U]) :- union(T,L2,U).              % clause 3
```

   In the last two clauses, recursion is used to find the union of the tail of L1 and the list L2.

5. Define concat(X,Y,Z) to mean "the concatenation of lists X and Y is Z". In the Prolog literature, this predicate is frequently called append.

```
concat([ ], L, L).                                   % clause α
concat([H|T], L, [H|M]) :- concat(T, L, M).          % clause β
```

```
?- concat([a,b,c], [d,e], R).
R = [a,b,c,d,e]
yes
```

The inference that produced this answer is illustrated by the search tree in Figure A.4. When the last query succeeds, the answer is constructed by unwinding the bindings:

```
R = [a | M] = [a | [b | M1]] = [a,b | M1] = [a,b | [c | M2]]
  = [a,b,c | M2] = [a,b,c | [d,e]] = [a,b,c,d,e].
```

Figure A.5 shows the search tree for another application of concat using semicolons to generate all the solutions.

To concatenate more than two lists, use a predicate that joins the lists in parts.

```
concat(L,M,N,R) :- concat(M,N,Temp), concat(L,Temp,R).
```

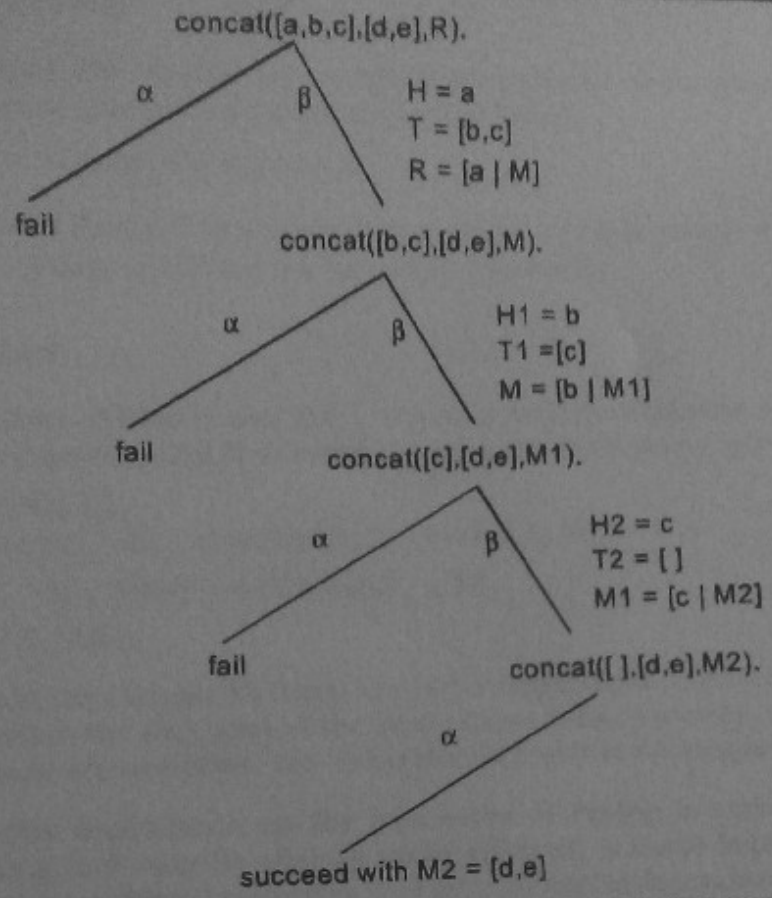

Figure A.4: A Search Tree for concat

No confusion results from using the same name for this predicate, since the two versions are distinguished by the number of parameters they take (the arities of the predicates).

6. Define reverse(L,R) to mean "the reverse of list L is R".

```
reverse([ ], [ ]).
reverse([H|T], L) :- reverse(T, M), concat(M, [H], L).
```

In executing concat, the depth of recursion corresponds to the number of times that items from the first list are attached (cons) to the front of the second list. Taken as a measure of complexity, it suggests that the work done by concat is proportional to the length of the first list. When reverse is applied to a list of length n, the executions of concat have first argument of lengths, n-1, n-2, ..., 2, 1, which means that the complexity of reverse is proportional to n^2.

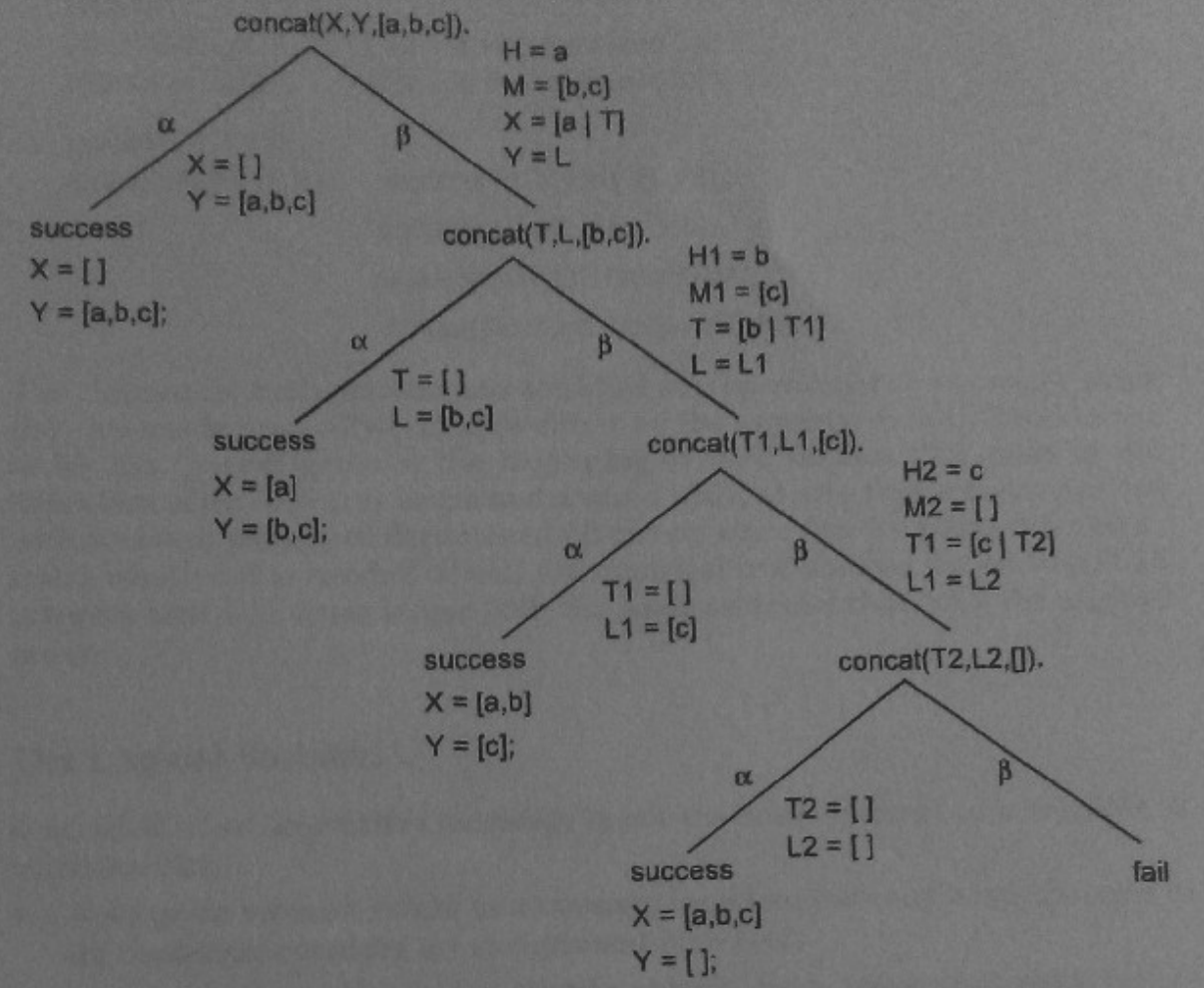

Figure A.5: Another Search Tree for concat

7. An improved reverse using an accumulator:

```
rev(L, R) :- help(L, [ ], R).

help([ ], R, R).
help([H|T], A, R) :- help(T, [H|A], R).
```

The predicate help is called n times if the original list is of length n, so the complexity of rev is proportional to n. Observe that the predicat help is tail recursive.

Sorting in Prolog

A few relations are needed for comparing numbers when sorting a list of numbers (equal and not equal are described later):

M < N, M =< N, M > N, M >= N.

These relations demand that both operands be numeric atoms or arithmetic expressions whose variables are bound to numbers.

Insertion Sort

If a list consists of head H and tail T, the idea with the insertion sort is to sort the tail T (recursively) and then insert the item H into its proper place in the tail.

```
insertSort([ ], [ ]).
insertSort([X|T], M) :- insertSort(T, L), insert(X, L, M).

insert(X, [H|L], [H|M]) :- H<X, insert(X, L, M).
insert(X, L, [X|L]).
```

Observe that the clauses for insert are order dependent. The second clause is executed when the first goal of the first clause fails—namely, when H>=X. If these clauses are switched, the definition of insert is no longer correct.

Although this dependence on the rule order of Prolog is common in Prolog programming and may be slightly more efficient, a more logical program is constructed by making the clauses that define insert independent of each other:

```
insert(X, [ ], [X]).
insert(X, [H|L], [X,H|L]) :- X=<H.
insert(X, [H|L], [H|M]) :- X>H, insert(X,L,M).
```

Now only one clause applies to a given list. The original clause insert(X, L, [X|L]). must be split into two cases depending on whether L is empty or not.

Quick Sort

The quick sort works by splitting the list into those items less than or equal to a particular element, called the **pivot**, and the list of those items greater than the pivot. The first number in the list can be chosen as the pivot. After the two sublists are sorted (recursively), they are concatenated with the pivot in the middle to form a sorted list.

The splitting operation is performed by the predicate partition(P, List, Left, Right), which means P is a pivot value for the list List, Left = { X∈List | X≤P }, and Right = { X∈List | X>P }.

```
partition(P, [ ], [ ], [ ]).
partition(P, [A|X], [A|Y], Z) :- A=<P, partition(P, X, Y, Z).
partition(P, [A|X], Y, [A|Z]) :- A>P, partition(P, X, Y, Z).

quickSort([ ], [ ]).
quickSort([H|T], S) :-  partition(H, T, Left, Right),
                        quickSort(Left, NewLeft),
                        quickSort(Right, NewRight),
                        concat(NewLeft, [H|NewRight], S).
```

The clauses for both partition and quickSort can be entered in any order since they are made mutually exclusive either by the patterns in their head terms or by the "guard" goals at the beginning of their bodies. The goals in the definition of partition may be turned around without affecting correctness but with a severe penalty of diminished efficiency since the recursive call will be made whether it is needed or not. An empirical test showed the sorting of 18 integers took 100 times longer with the goals switched than with the original order.

The Logical Variable

A variable in an imperative language is not the same concept as a variable in mathematics:

1. A program variable refers to a memory location that may have changes in its contents; consider an assignment N := N+1.
2. A variable in mathematics simply stands for a value that once determined will not change. The equations $x + 3y = 11$ and $2x - 3y = 4$ specify values for x and y—namely, x=5 and y=2—which will not be changed in this context. A variable in Prolog is called a **logical variable** and acts in the manner of a mathematical variable.
3. Once a logical variable is bound to a particular value, called an **instantiation** of the variable, that binding cannot be altered unless the pattern matching that caused the binding is undone because of backtracking.
4. The destructive assignment of imperative languages, where a variable with a value binding is changed, cannot be performed in logic programming.
5. Terms in a query change only by having variables filled in for the first time, never by having a new value replace an existing value.
6. An iterative accumulation of a value is obtained by having each instance of a recursive rule take the values passed to it and perform computations of values for new variables that are then passed to another call.

7. Since a logical variable is "write-once". It is more like a constant identifier with a dynamic defining expression as in Ada (or Pelican) than a variable in an imperative language.

The power of logic programming and Prolog comes from using the logical variable in structures to direct the pattern matching. Results are constructed by binding values to variables according to the constraints imposed by the structures of the arguments in the goal term and the head of the clause being matched. The order that variables are constrained is generally not critical, and the construction of complex values can be postponed as long as logical variables hold their places in the structure being constructed.

Equality and Comparison in Prolog

Prolog provides a number of different ways to compare terms and construct structures. Since beginning Prolog programmers often confuse the various notions of equality and related predicates, we provide a brief overview of these predicates.

Unification

"T1 = T2" Succeed if term T1 can be unified with term T2.

```
| ?- f(X,b) = f(g(a),Y).
X = g(a)
Y = b
yes
```

Numerical Comparisons

"=:=", "=\=", "<", ">", "=<", ">="

Evaluate both expressions and compare the results.

```
| ?- 5<8.
yes
| ?- 5 =< 2.
no
| ?- N =:= 5.
! Error in arithmetic expression: not a number (N not instantiated to a number)
no
| ?- N = 5, N+1 =< 12.
N = 5                    % The unification N = 5 causes a binding of N to 5.
yes
```

Forcing Arithmetic Evaluation (is)

"N is Exp" Evaluate the arithmetic expression Exp and try to unify the resulting number with N, a variable or a number.

```
| ?- M is 5+8.
M = 13
yes
| ?- 13 is 5+8.
yes
| ?- M is 9, N is M+1.
M = 9
N = 10
yes
| ?- N is 9, N is N+1.
no                                  % N is N+1 can never succeed.
| ?- 6 is 2*K.
! Error in arithmetic expression: not a number (K not instantiated to a number)
no
```

The infix predicate is provides the computational mechanism to carry out arithmetic in Prolog. Consider the following predicate that computes the factorial function:

The factorial of 0 is 1.

```
fac(0,1).
```

The factorial of N>0 is N times the factorial of N-1.

```
fac(N,F) :- N>0, N1 is N-1, fac(N1,R), F is N*R.
```

```
| ?- fac(5,F).
F = 120
yes
```

Identity

"X == Y" Succeed if the terms currently instantiated to X and Y are literally identical, including variable names.

```
| ?- X=g(X,U), X==g(X,U).
yes
| ?- X=g(a,U), X==g(V,b).
no
| ?- X\==X.                          % "X \== X" is the negation of "X == X"
no
```

Term Comparison (Lexicographic)

"T1 @< T2", "T1 @> T2", "T1 @=< T2", "T1 @>= T2"

```
| ?- ant @< bat.
yes

| ?- @<(f(ant),f(bat)).          % infix predicates may also be entered
yes                              %   as prefix
```

Term Construction

"T =.. L" L is a list whose head is the atom corresponding to the principal functor of term T and whose tail is the argument list of that functor in T.

```
| ?- T =.. [@<,ant,bat], call(T).
T = ant@<bat
yes

| ?- T =.. [@<,bat,bat],call(T).
no

| ?- T =.. [is,N,5], call(T).
N = 5,
T = (5 is 5)
yes

| ?- member(X,[1,2,3,4]) =.. L.
L = [member,X,[1,2,3,4]]
yes
```

Input and Output Predicates

Several input and output predicates are used in the laboratory exercises. We describe them below together with a couple of special predicates.

get0(N) N is bound to the ascii code of the next character from the current input stream (normally the terminal keyboard). When the current input stream reaches its end of file, a special value is bound to N and the stream is closed. The special value depends on the Prolog system, but two possibilities are:

26, the code for control-Z or

-1, a special end of file value.

put(N) The character whose ascii code is the value of N is printed on the current output stream (normally the terminal screen).

see(F)	The file whose name is the value of F becomes the current input stream.
seen	Close the current input stream.
tell(F)	The file whose name is the value of F becomes the current output stream.
told	Close the current output stream.
read(T)	The next Prolog term in the current input stream is bound to T. The term in the input stream must be followed by a period.
write(T)	The Prolog term bound to T is displayed on the current output stream.
tab(N)	N spaces are printed on the output stream.
nl	Newline prints a linefeed character on the current output stream.
abort	Immediately terminate the attempt to satisfy the original query and return control to the top level.
name(A,L)	A is a literal atom or a number, and L is a list of the ascii codes of the characters comprising the name of A. **| ?- name(A,[116,104,101]).** A = the **| ?- name(1994,L).** L = [49, 57, 57, 52]
call(T)	Assuming T is instantiated to a term that can be interpreted as a goal, call(T) succeeds if and only if T succeeds as a query.

This Appendix has not covered all of Prolog, but we have introduced enough Prolog to support the laboratory exercises in the text. See the further readings at the end of Chapter 2 for references to more material on Prolog.