Chapter 12 - The Logic-Oriented Model

The logic-oriented model is introduced in this chapter. It is briefly described, then discussed in more detail through the introduction of the Prolog language. The implementation of the logic-oriented model in database query languages is illustrated, using the language SQL.

12.1 Introduction to Logic Language Model

The logic model for programming languages permits the expression of programs in a form similar to symbolic logic. This process is essentially the same as writing a program that proves a theorem. In order to understand how this works, let us examine how a theorem is proved.

First, a theorem must be proved from a set of known facts that are accepted as true. These are called axioms in mathematics. The statement to be proved is then considered a goal, and the given information is manipulated using rules of logical inference until the goal statement is demonstrated to be true.

A program written using a programming language of the logic model consists of a set of axioms and a goal statement, both expressed in the syntax of the language. The rules of inference are then applied to determine whether the axioms are sufficient to ensure the truth of the goal statement or not. The manner in which the rules of inference are applied to derive the goal from the axioms is not expressed in the language, but is assumed to occur automatically. In this way, programs written in the logic model express a goal and the facts from which this goal is to be derived but do not express the method by which this derivation takes place. The derivation method may be produced by the execution of the program if the goal is derivable.

As an example, consider the logic model approach to finding the greatest common divisor of 16 and 56. We first state our axioms that define the greatest common divisor:

Axiom 1: gcd(N,1) = 1 for all N ≥ 1.
Axiom 2: gcd(N,M) = gcd(M,N mod M) for N ≥ 1 and M ≥ 1.
Goal: gcd(56,16)

The axioms state a definition of the greatest common divisor and the goal is the result we wish to derive. This is the entire program in a logic model language. It is unnecessary for the programmer to express how the program should go about determining the validity of the goal statement.

In contrast to the imperative model--where the programmer must describe how a problem is solved--under the logic model, the programmer describes the goal, and the system searches for a solution in the form of a proof that verifies that goal.

12.2 Prolog -- A Logic-Oriented Language

Prolog is a language that is based very closely on the logic model. It was created in 1972 by Alain Colmerauer in Marseilles, France. Since its creation, it has been extensively used in com-
puter science research in Europe and has been adopted as the core language of the Japanese Fifth Generation Project. It is of particular interest to the artificial intelligence community, where it has found most of its applications. This Section describes the logic-oriented model using Prolog as an example of this model. Chapter 12.4 describes some of the non-logic-oriented features of Prolog and significant implementation issues.

12.2.1 Basic Components

An **object** is the most basic component of Prolog and can be any specific object at all. It might be a number, a person, a set of books, a program, a list of names, a sequence of characters, or anything that can be conceptually abstracted into a single item. Note that an object can be composite in nature such as the list, set or sequence mentioned in the preceding sentence. The meaning of an object is completely determined by its use within the program--that is, through its logical relationships with other objects. In the pure logic model, there is no binding of type to an object.

An individual object in Prolog is represented by a sequence of characters that begins with a lowercase letter or a digit. We use the convention that any sequence of digits represents a number object that corresponds to the sequence interpreted as a decimal integer. This can be done without violating the typeless property of the logic model, because such objects are assumed to carry with them the entire set of arithmetic relations that are necessary to define them.

**Relations** represent some quality, attribute, or relationship of one or more objects. Relations are given names that begin with a lowercase letter and are expressed by the relation name, which is followed by a list of object names separated by commas and enclosed in parentheses. Some examples of relations are

```
father(tom,jane)
larger(2,1)
male(robert)
cost(carl,17246)
```

A variable in Prolog is represented by a sequence of characters beginning with an uppercase letter. A variable, when used in the context of a relation, indicates that the relation holds for all objects in the space of objects that are defined. For example, the relation

```
same_age(X,X)
```

indicates that every object is related to itself by the relation SAME_AGE.

A program written in Prolog may contain three kinds of statements--a goal, facts, and rules--all terminated with a period. The program must contain exactly one goal, but may include any number of facts and rules. A goal is expressed by `?-` followed by a fact. There are two types of goals, depending on whether the embedded fact contains any variables or not. A goal with no variables is simply a request to determine whether the fact embedded in the goal can be derived as true from the facts and rules present in the program. For example,
?-father(john,sue).

will result in true if john can be derived as the father of sue according to the facts and rules given, and results in false if this relation cannot be derived. Note that we do not say the relation is "not true". Rather, it cannot be derived as true from the information present in the program.

The second form of a goal, that containing one or more variables, requests that all object tuples be listed if, when substituted into the goal for the corresponding variables, they cause the goal to be derivably true. For example, the goal

?-father(X,sue).

might return the result

X = john

Furthermore, the goal statement

?-adjacent(X,california).

might return

X = oregon
X = nevada
X = arizona

and

?-completed(john,COURSE,GRADE).

might return

COURSE = cs100   GRADE = a
COURSE = cs200   GRADE = b
COURSE = ma110   GRADE = b
COURSE = en200   GRADE = c

A fact is a relation between objects and indicates that the given relation does indeed hold. For example,

father(john,sue).

states that john is the father of sue. If a fact contains variables, then the fact holds for all objects in the object space of this program. For example,
female(PERSON).

specifies that all objects that are used within the containing program have the property FEMALE. Similarly

knows(X,barb).

indicates that the relation knows holds between all objects and barb.

Finally, a rule consists of two parts, a fact that is the conclusion of the rule and a list of facts that form the hypotheses for the rule. The general form is

c :- h₁, h₂, ..., hₙ

The meaning of such a rule is that fact c is derived to be true if facts h₁, h₂, ..., hₙ are all derivably true. For example, the rule

parent(john,sue) :- father(john,sue).

states that john is a parent of sue if john is a father of sue. Furthermore, the use of variables is common in the case of rules in order to generalize the rule to hold for all possible objects. The preceding rule could be generalized to

parent(X,Y) :- father(X,Y).

There may be many ways of deriving the same fact, and thus the same conclusion might be present in many rules. For example,

parent(X,Y) :- mother(X,Y).

would also be an appropriate rule.

Finally, some examples of rules containing multiple hypotheses are

mother(X,Y) :- parent(X,Y),female(X).

grandmother(X,Y) :- mother(X,Z),parent(Z,Y).

Note that in the last example, the variable Z is used in the hypotheses even though it does not appear in the conclusion. Such a variable is used to tie together facts in the hypotheses. This last rule could be stated: "X is a grandmother of Y if X is the mother of Z and Z is the parent of Y for some Z." In other words, a variable found only in the hypotheses is handled differently from one appearing in the conclusion of a rule. To further define this difference, if variables X₁, X₂, ..., Xₙ are used in the conclusion and variables Y₁, Y₂, ..., Yₘ are used in the hypotheses but not in the conclusion, the rule
\[ c(X_1, X_2, \ldots, X_n) : - h(X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots Y_m). \]

is interpreted as

For all objects \( X_1, X_2, \ldots, X_n \), \( c \) holds if there exist objects \( Y_1, Y_2, \ldots, Y_m \) such that \( h \) holds.

Figure 12.1 shows the BNF definition of the subset of Prolog we have introduced in this section.

Figure 12.1 BNF Representation of Prolog subset

\[
\begin{align*}
\text{program} & ::= \langle\text{goal}\rangle. \\
& \{ (\langle\text{rule}\rangle - \langle\text{fact}\rangle). \}
\end{align*}
\]

\[
\begin{align*}
\text{fact} & ::= \langle\text{constant}\rangle - \langle\text{variable}\rangle - \\
& \langle\text{relation-name}\rangle ( \langle\text{fact}\rangle \{,\langle\text{fact}\rangle\} )
\end{align*}
\]

\[
\begin{align*}
\text{rule} & ::= \langle\text{fact}\rangle : - \langle\text{fact}\rangle \{,\langle\text{fact}\rangle\}
\end{align*}
\]

\[
\begin{align*}
\text{goal} & ::= \text{-}\langle\text{fact}\rangle .
\end{align*}
\]

\[
\begin{align*}
\text{constant} & ::= \langle\text{lowercase letter}\rangle \{\langle\text{character}\rangle\} - \\
& \langle\text{digit}\rangle \{\langle\text{character}\rangle\}
\end{align*}
\]

\[
\begin{align*}
\text{variable} & ::= \langle\text{uppercase letter}\rangle \{\langle\text{character}\rangle\}
\end{align*}
\]

\[
\begin{align*}
\text{relation-name} & ::= \langle\text{lowercase letter}\rangle \{\langle\text{character}\rangle\}
\end{align*}
\]

12.2.2 Example Program in Prolog

Now we develop an extensive example program in our Prolog subset to illustrate how programming takes place under the logic model. This program consists of the description of relations between people, machines, types of hardware, and types of software. The objective of our program is to determine if a given person can run a specific software product.

Our example includes five types of objects, described in Figure 12.2, and five relations, described in Figure 12.3. Figure 12.4 lists a set of facts in Prolog syntax that are a part of the program. In particular, this figure defines specific relations among the objects included in the program using the three relations type, runs, and access. The other two relations, can_run and can_use, are relations that are defined by rules rather than facts. Rules for the example program are given in Figure 12.5. The first rule, which defines can_use, can be stated in words as follows:
For all objects P and SW, P can use SW if there exists an object MACH such that P has access to MACH and MACH can run SW.

This statement indicates that the relation can_use holds between two objects if an object can be found that satisfies both facts on the right-hand side of the rule.

Figure 12.2 Objects used in example program

Machines (MACH) = {ibmpc, mac}

Software (SW) = {spreadsheet, pascal, basic, smalltalk}

Hardware (HW) = {mach1, mach2, mach3}

Memory Size (MEM1, MEM2) in Kbytes = set of integers

Person (P) = {sue, jerry, sam}

Figure 12.3 Relations used in example program

runs(HW, SW, MEM)
    The hardware HW runs software SW if it has at least MEM Kbytes of memory.

spec(MACH, HW, MEM)
    Machine MACH is of type HW and has memory size MEM.

access(P, MACH)
    Person P has access to machine MACH.

can_run(MACH, SW)
    Machine MACH can run software SW.

can_use(P, SW)
    Person P can use software SW.

Figure 12.4 Facts for example program in Prolog

F1:  spec(mach1, ibmpc, 320).
F2:  spec(mach2, mac, 1000).
F3:  spec(mach3, ibmpc, 640).
F4:  runs(ibmpc, spreadsheet, 500).
F5:  runs(ibmpc, basic, 128).
F6:  runs(ibmpc, pascal, 256).
F7:  runs(mac, basic, 200).
F8:  runs(mac, smalltalk, 1000).
F9:  access(sue,mach1).
F10:  access(jerry,mach3).
F11:  access(sam,mach1).
F12:  access(sam,mach2).
Figure 12.5 Rules for example program in Prolog

R1: can_use(P,SW) :- access(P,MACH),
    can_run(MACH,SW).

R2: can_run(MACH,SW) :- spec(MACH,HW,_MEM1),
    runs(HW,SW,_MEM2),
    MEM1>=MEM2.

For example, the goal

?-can_use(sue,basic).

is satisfied if there is some object MACH for which both access(sue,MACH) and can_run(MACH,basic) are satisfied. By examining the facts in Figure 12.4, we see that the only MACH for which the first fact is satisfied is mach1. For the remaining subgoal, we may then assume that MACH is mach1 since that is the only object that satisfies the first subgoal. This is referred to as binding the variable MACH to mach1. The second clause of the rule is itself a rule, namely, can_run. Since MACH has to be mach1 to satisfy the first subgoal, we now need to determine if can_run(mach1,basic) is satisfied. This clause is satisfied if all three of the subgoals for can_run are satisfied for some specific objects MEM1, MEM2, and HW. In our case, the three subgoals will be

spec(mach1,HW,MEM1).
spec(mach1,ibmpc,320).
runs(HW,SW,MEM2).
runcs(ibmpc,basic,128).
MEM1>=MEM2.
320>=128.

The last clause requires further explanation, because it does not appear to be a relation, but rather a comparison. In reality, it is a relation that applies to two numeric objects. Here the relation MEM1>=MEM2 holds precisely when MEM1 is greater than or equal to MEM2. The first two subgoals can be matched by rules found in our program (Figure 12.4), and the third subgoal is matched by the definition of >=.

spec(mach1,ibmpc,320).
runs(ibmpc,basic,128).
320>=128.

with variable bindings HW=ibmpc, MEM1=320, and MEM2=128. Therefore, the rule

can_run(mach1,basic).

is satisfied; hence, so is the original goal
can_use(sue,basic).

Figure 12.6 illustrates the preceding process using a derivation tree. In this tree, each rule becomes a parent node for all the subgoals on its right-hand side. Variables that are bound are indicated by assignments labeling the edges of the tree. This tree shows the path by which the goal is verified, but it does not indicate the method by which this is done. We will examine this process later.

Figure 12.6 Derivation Tree

Goal: ?-can_use(sue,basic)

access(sue,MACH)  can_run(MACH,basic)

MACH:=mach1

access(sue,mach1)  FACT

spec(mach1,HW,MEM1)  runs(ibmpc,basic,MEM2)  320>=128

HW:=ibmpc
MEM2:=128

MEM1:=320

spec(mach1,ibmpc,320)  FACT

runs(ibmpc,basic,128)

The goal just considered contained no variables and hence was either satisfied or not satisfied. The more complex form of a goal— that containing one or more variables—asks for all tuples of objects that, when substituted for the variables, satisfy the goal. For example, our example program might have as its goal

?-can_use(sue,X).

This goal is asking for all objects that represent software products that sue can use. After some examination, we could deduce that the rules in Figure 12.4 yield two possible values of X,

\[
\begin{align*}
X &= \text{basic} \\
X &= \text{pascal}
\end{align*}
\]

12.2.3 The Process of Deduction

At the heart of a logic programming language is a processor commonly known as the infer-
Inference engine, whose role is to take all of the provided facts and rules and derive new facts from them. In particular, the inference engine is asked to derive new facts based on the goal that is given, in the hope that the goal will be one of those derived facts. In this section, we will study how the inference engine operates.

The two primary operations used by an inference engine to derive new facts from given facts and rules are resolution and unification. Resolution says that if we are given two rules, with a given fact \( f \) on the left-hand side of one rule and \( f \) also on the right-hand side of the other rule, a new rule can be derived whose left-hand side consists of the union of the two left-hand sides with \( f \) removed, and whose right-hand side consists of the union of the right-hand sides with \( f \) removed. Stated in generality using Prolog syntax, the two rules

\[
\begin{align*}
  f & : - a_1, a_2, \ldots, a_n \\
  g & : - f, b_1, b_2, \ldots, b_m
\end{align*}
\]

resolve to the new rule

\[
  g : - a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m
\]

If you follow the logic of these statements carefully, you can verify that the result of resolution is logically valid.

Consider the following example of resolution. Suppose we are given

\[
\begin{align*}
  \text{play\_ball}(X) & : - \text{not\_raining}, \text{not\_working}(X). \\
  \text{not\_working}(X) & : - \text{weekend}.
\end{align*}
\]

We can derive the following new rule by resolution over the relation \( \text{not\_working}(X) \):

\[
\text{play\_ball}(X) \text{ if not\_raining, weekend}.
\]

Notice that this is a valid logical deduction from the two given rules, which is always the case for resolution, because it is verifiably valid by rules of logic.

A special case of resolution arises when we observe that a fact is a degenerate occurrence of a rule. For example, the fact

\[
\text{csmajor}(harry).
\]

is a special case of the rule

\[
\text{csmajor}(harry) [:- \text{anything}].
\]

The ":- anything" is enclosed in brackets in the preceding rule to indicate that it need not be present and is included only to illustrate that a fact is a special case of a rule.

If this new rule is resolved with the rule
invite(harry) :- csmajor(harry),senior(harry).

the result is

invite(harry) :- [anything,]senior(harry).

or equivalently

invite(harry) :- senior(harry).

Similarly, if the two rules are

csmajor(harry) [:- anything].
invite(harry) :- csmajor(harry).

then resolution derives the fact

invite(harry) [:- anything].

Therefore, resolution can be used to derive a new rule from two rules or from a rule and a fact. Resolution can also derive a new fact from a rule and a fact.

Unification is the derivation of a new rule from a given rule through the binding of variables. For example, the rule

invite(X) :- csmajor(X),senior(X).

unifies to the new rule

invite(harry) :- csmajor(harry),senior(harry).

under the binding of harry to the variable X.

The derivation of new facts, including goal facts, can be expressed through a sequence of resolutions and unifications. Consider, for example, the derivation of the goal

?-can_use(sue,basic).

which is illustrated in Figure 12.6. As a series of resolutions and unifications, it might be derived as follows:

Unify R1 with SW=basic,P=sue,MACH=mach1
R3: can_use(sue,basic) :-
    access(sue,mach1),can_run(mach1,basic).
Unify R2 with MACH=mach1,SW=basic,MEM1=320,MEM2=128
R4: can_run(mach1,basic) :-
    spec(mach1,ibmpc,320),
12.2.4 Implementation considerations

You may have observed that we have said nothing about how the inference engine decides which unification or resolution to perform next in the derivation of a goal. This omission has been intentional, because that is part of the "how" in the solution of the problem and we have attempted to avoid "how" in our logic model. The programmer simply states the goal, facts, and rules in Prolog, and the inference engine determines the satisfiability of the goal. It is fine to consider the program as independent of the implementation of the inference engine when working with logic model, but, in practice, efficiency considerations usually require some knowledge of the implementation. The development of an appropriate derivation of a goal usually requires a search for the appropriate sequence of unifications and resolutions, and the choice of strategy used in that search might determine whether the derivation is accomplished in a reasonable amount of time or not. In some cases, an unwisely chosen search strategy could lead to non terminating search that never finds a valid derivation, even when one is possible.

The derivation of a goal can proceed in either of two ways. First, all possible resolutions and unifications could be performed until the goal is derived. This process has the disadvantage that many irrelevant facts and rules are derived before the desired goal appears. In fact, since there may be an infinite number of derivable facts and rules, the desired goal may never be derived.

The more useful approach is to start with the goal and work backward by identifying facts and rules from which the goal can be derived. In particular, this backward process can occur in two ways: the expansion of a goal by applying a rule and the reduction of a goal by applying a fact.

First, let's examine expansion by applying a rule. At any point, our goal is a list of facts, all of which must be satisfied for the goal to be satisfied. We represent facts by lowercase letters and assume that our present goal is

\[ ?- a, b, c. \]
If a rule is present that can be unified with fact \( a \)--that is, if the conclusion of the rule is the same as fact \( a \) when a variable is bound to an object--then the goal can be expanded according to that rule. For example, suppose the rule

\[ a : \neg d, e, f. \]

is present in the database of facts and rules. Then the original goal can be expanded to

\[ ?-d, e, f, b, c. \]

This process is valid, because \( ?-a, b, c \) can be derived if \( ?-d, e, f, b, c \) can be derived by an application of the preceding rule.

Note that this process can be summarized by

\[ a \text{ and } (a : \neg d, e, f) \rightarrow d, e, f. \]

where the \( b \) and \( c \) parts of the goal are simply carried along as additional subgoals. This process is the reversal of the resolution that would occur if the arrow went the other way. In other words, because we are starting with the goal and working backwards, each step is a reversed derivation step.

The second type of step--reduction using a fact--is the elimination of one of the facts in the goal because that fact matches a fact given in the program, possibly after the binding of one or more variables. For example, if the goal is

\[ ?-a, b, c. \]

and the fact \( a \) is present in the database, then this goal can be reduced to

\[ ?-b, c. \]

Thus, \( ?-a, b, c \) can be derived if \( ?-b, c \) can be derived, because \( a \) is a given fact.

Now we can apply this backward process to our derivation of \( ?-can\_use(sue,\text{basic}) \) examined earlier.

1. \( ?-can\_use(sue,\text{basic}). \) /R1/P=sue,SW=\text{basic}
2. \( ?-access(sue,MACH),can\_run(MACH,\text{basic}). /F9/MACH=mach1\)
3. \( ?-can\_run(mach1,\text{basic}). \) /R2
4. \( ?-\text{spec}(mach1,HW,\text{MEM1}),\text{runs}(HW,\text{basic},\text{MEM2}),\text{MEM1}>=\text{MEM2}. /F1/hw=IBMPC,mem1=320\)
5. \( ?-\text{runs(ibmpc,basic,\text{MEM2}),320}>=\text{MEM2}. /F10/\text{MEM2}=128\)
6. \(-320}>=128. /success\)

The logical argument then follows that if goal 6 is satisfied, then so is goal 5, if goal 5 is satisfied, then so is goal 4, and so on. Because goal 6 is a verified fact, the original goal (goal 1) is also satisfied.
In the above derivation, this process went more smoothly than we have a right to expect. To illustrate a possible difficulty, consider the goal

?can_use(jerry,spreadsheet).

Our process can proceed as before

1. ?-can_use(sam,spreadsheet). /R1/P=jerry,SP=spreadsheet
2. ?-access(sam,MACH),can_run(MACH,spreadsheet).
   /F12/MACH=mach2
3. ?-can_run(mach2,spreadsheet). /R2
4. ?-spec(mach2,HW,MEM1),runs(HW,spreadsheet,MEM2),MEM1>=MEM2.
   /F2/HW=mac,MEM1=1000
5. ?-runs(mac,spreadsheet,MEM2),1000>=MEM2.

Now the first subgoal at step 5 cannot be satisfied. This does not mean, however, that the original goal cannot be satisfied, because careful examination of the problem will indicate that it can be. This difficulty arises because at step 2, both F10 and F11 apply, and we made the wrong choice for completing our derivation. Had we chosen F11, we would have been successful.

This points to the necessity, in implementing this derivation process, of backing up to try other alternatives whenever we fail to satisfy a subgoal. The process that we will outline here will back up, one level at a time, until we reach a level where there is an alternate choice for satisfying the leading subgoal. If no such level exists, the search is completed and the derivation fails. This backing-up process is especially necessary when the original goal contains variables, so that all bindings of the variables must be found which lead to successful derivations. The derivation of the goal ?-can_use(X,spreadsheet) utilizing the backing-up process just described is illustrated in Figure 12.7. Notice that at each step the first goal (subgoal) in the list is either expanded by a rule or satisfied by a fact; in the case where neither can occur, the process backs up to the immediately preceding set of subgoals, where it searches for matches beyond those already attempted.

For example, after the first failure at goal 6, the process backs up to goal 5, where an alternative to M=500 is required. None being found, the process backs up to goal 4, where an alternative to HW=ibmpc and M1=320 is required. Again, none is found, so the process backs up to goal 3. Because goal 3 causes no variables to be bound, it is not possible to try other alternative bindings here, so the process immediately backs up to goal 2. At goal 2, a match can be found other than X=sue, M=mach1. The next such match being X=jerry, M=mach2, these bindings are made and the process proceeds forward once again. You can trace the rest of the process through Figure 12.7.

In summary, a logic-oriented language is implemented through a simple process of reverse resolution and unification that results in the generation of subgoals from the original goal. Because there may be several alternative subgoals generated by a given goal, the process may require backtracking to try all possible substitutions. This process can become quite time-consuming in practice, and the efficiency of the process can be affected by the order in which alternatives are considered.
Furthermore, in the case of recursive rules—that is, rules whose right-hand side contains the same relation as expressed on the left-hand side—the order of search can determine whether the derivation will terminate or not. For example, consider the following addition to our example Prolog program. First we add the new relation written_in where

\[
\text{written_in}(X,Y) \quad \text{means software } X \text{ is written in language } Y.
\]

Then, a simplified new rule can be added to our program:

\[
\text{can_run}(MACH,SW) :- \text{can_run}(MACH,L), \text{written_in}(SW,L).
\]
Now if we add the fact

\[ \text{written_in(spreadsheet,pascal)}. \]

and assume that our new rule for \text{can_run} is attempted before any other rule for \text{can_run} in our database search space, we generate the following derivation:

1. ?-\text{can_run}(X,\text{spreadsheet}).
2. ?-\text{can_run}(X,L),\text{written_in}(\text{spreadsheet},L).
3. ?-\text{can_run}(X,L'),\text{written_in}(L,L'),\text{written_in}(\text{spreadsheet},L').
   ...

This example illustrates that the order in which rules and facts are matched in the derivation can have a major impact on the resulting search process. In a pure logic programming environment, like Prolog, the assumption is made that if there is a valid derivation, the system will find it and that the order in which rules and facts occur is irrelevant. In practice, this concept is very difficult to implement, and actual languages do perform their searches for derivations in a way that is dependent on the order of declarations in the program.

12.3 Database query languages

One popular implementation of the logic model for a programming language is the query language of a relational database management system. In order to illustrate the correspondence between a relational query language and the logic model, we first briefly introduce the fundamentals of a relational database management system. Then we describe a relational query language, SQL, and, finally, we detail the query language-logic model correspondence.

12.3.1 Relational database management systems

The fundamental entity of a relational database management system is the \text{relation}, which can be viewed as a table of rows and columns, where each row, called a \text{tuple}, represents an object, and each column, called an \text{attribute}, represents a property of the object. For example, Figure 12.8 represents the relation \text{STUDENT} where each tuple is a single student and each column an attribute of students. This particular table has four tuples, representing four students, and five attributes, called \text{NAME}, \text{CLASS}, \text{SEX}, \text{MAJOR}, and \text{AGE}.

A database consists of one or more relations. Data are manipulated, stored in the tables, and retrieved from the tables via commands written in a \text{query language}. Many tools are provided for the organization of the data for efficient data entry, efficient retrieval, and formatting of reports and screens, but, because these are not pertinent to our purposes here, we will not discuss them. Our focus is on the query language as it relates to the logic model for programming languages.
12.3.2 Query language SQL

SQL is the most commonly used relational query language. Many features of SQL are omitted from our discussion because our intent here is to illustrate its basic adherence to the logic model. The fundamental features described in this section are those required to create relations, add tuples to relations, define views, and selectively retrieve information from relations.

Figure 12.8 Relation STUDENT

<table>
<thead>
<tr>
<th>Attributes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CLASS</td>
</tr>
<tr>
<td>Sue</td>
<td>SR</td>
</tr>
<tr>
<td>John</td>
<td>JR</td>
</tr>
<tr>
<td>Jerry</td>
<td>FR</td>
</tr>
<tr>
<td>Sally</td>
<td>FR</td>
</tr>
</tbody>
</table>

First, the creation of a relation is specified by a CREATE statement. The creation in SQL of the relation STUDENT shown in Figure 12.8 is accomplished by the statement

```
CREATE STUDENT (NAME CHAR(20),
                 CLASS CHAR(2),
                 SEX CHAR(1),
                 MAJOR CHAR(10),
                 AGE INTEGER)
```

The CREATE statement consists of the word CREATE followed by the name of the relation and the list of attributes, with each attribute's name and type specified. When a relation is created, it is assumed to be empty, containing no tuple.

Tuples can be added to a relation through the INSERT statement. One simple form of such a statement is

```
INSERT INTO STUDENT (NAME, CLASS, SEX, MAJOR, AGE) VALUES ('Sue', 'SR', 'F', 'Math', 21)
```

This statement includes the name of the relation, the list of attribute names, and a list of values that correspond positionally to the attributes named.

To illustrate views and retrievals in SQL, we need to define a second relation, ATHLETE, in our database. This relation contains three tuples and two attributes, and is shown in Figure 12.9. It is assumed that tuples in STUDENT and ATHLETE represent the same person when their NAME attribute values are identical.
Figure 12.9 Relation ATHLETE

Attributes
NAME    SPORT
John    Swimming
Sally   Soccer
Joe     Baseball

Retrievals are ways of specifying tables of data that are to be returned to the user. For example, one might wish to find the names of all students older than 20. Such a retrieval is expressed in SQL by

```
SELECT NAME
FROM STUDENT
WHERE AGE > 20
```

Here, the word SELECT is followed by the name of the attribute to be retrieved, the word FROM is followed by the name of the relation from which retrieval takes place, and WHERE is followed by a condition that is used to make the selection.

It is possible to retrieve more than one field using the SELECT statement. In the preceding example, if we want the ages and names of students be retrieved, we could write

```
SELECT NAME, AGE
FROM STUDENT
WHERE AGE > 20
```

We can also base the retrieval on more than one relation. Suppose we wish to retrieve the names and ages only of students who are also athletes. We could do this with

```
SELECT STUDENT.NAME, STUDENT.AGE
FROM STUDENT, ATHLETE
WHERE STUDENT.NAME = ATHLETE.NAME
```

The qualifiers need to be placed on NAME and AGE after SELECT because there are two relations involved in the retrieval. Therefore, attribute names need to specify the relation from which they are to be retrieved.

In addition, the condition for retrieval can be a complex logical expression. If we wish to retrieve the names and ages of all students who are athletes and older than 20, we can write

```
SELECT STUDENT.NAME, STUDENT.AGE
FROM STUDENT, ATHLETE
WHERE STUDENT.NAME = ATHLETE.NAME AND STUDENT.AGE > 20
```

A view in SQL is conceptually the same as a relation, consisting of tuples and attributes. A
view, however, is not actually found within the storage of the computer. Rather, it is derived from existing relations and the definition of the view. For example, the view `STUDENT_ATHLETE` can be created by the statement

```sql
CREATE VIEW STUDENT_ATHLETE
AS SELECT STUDENT.NAME, STUDENT.AGE, ATHLETE.SPORT
FROM STUDENT, ATHLETE
WHERE STUDENT.NAME = ATHLETE.NAME
```
This view has three attributes, two selected from STUDENT and one selected from ATHLETE. The rule for forming tuples is that tuples are formed only when the NAME field of STUDENT equals the NAME field of ATHLETE. Therefore, view STUDENT_ATHLETE consists of the two tuples shown in Figure 12.10.

Figure 12.10 View STUDENT_ATHLETE

<table>
<thead>
<tr>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
</tr>
<tr>
<td>John</td>
</tr>
<tr>
<td>Sally</td>
</tr>
</tbody>
</table>

Views can then be used as if they were actual tables in retrievals. For example, all student-athletes older than 20 could be retrieved by

```sql
SELECT NAME, AGE
FROM STUDENT_ATHLETE
WHERE AGE > 20
```

The contents of the hypothetical table associated with a VIEW change automatically as its constituent tables are modified.

Many other facilities are present in SQL, but those described above are sufficient for the purposes of the following section, which shows how SQL adheres to the logic model of a programming language.

### 12.3.3 SQL as a logic language

Most logic model capabilities can be specified in SQL through the `INSERT`, `VIEW`, and `SELECT` facilities. For example, the assertion of facts corresponds directly to the insertion of tuples into a relation, with the tuple defining an occurrence of a relation just like a fact does. A view is the SQL way of defining a rule, because it is the creation of a relation that is formed through logical manipulation of other relations. Finally, the `SELECT` query in SQL corresponds to a goal in the logic model, with the variables in the goal corresponding to the retrieved attributes of the view.
Figure 12.11 SQL expression of logic program in Figures 12.2-12.5

CREATE TABLE RUNS (MACH CHAR(5),
  SW CHAR(11)),
MEM INTEGER);
CREATE TABLE SPEC (MACH CHAR(5),
  HW CHAR(5),
MEM INTEGER);
CREATE VIEW CAN_RUN
  AS SELECT SPEC.MACH,RUNS.SW
  FROM SPEC,RUNS
  WHERE SPEC.HW=RUNS.HW AND
  SPEC.MEM>=RUNS.MEM;
CREATE VIEW CAN_USE
  AS SELECT ACCESS.PERSON,CAN_RUN.SW
  FROM ACCESS,CAN_RUN
  WHERE ACCESS.MACH=CAN_RUN.MACH;
INSERT INTO SPEC (MACH,HW,MEM)
VALUES ('MACH1','IBMPC',320);
...

To illustrate this correspondence, we return to the Prolog program given in Figures 12.2 through 12.5. Its expression in SQL is found in Figure 12.11.

We next show how goals can be duplicated by retrievals in SQL. Consider the goal

?-can_use(sue,spreadsheet).

In SQL, this can be accomplished by the retrieval

SELECT PERSON,SW
FROM CAN_USE
WHERE PERSON = 'Sue' AND SW = 'Spreadsheet'

Goals involving variables in the logic model are even more easily implemented in SQL. For example,

?can_use(X,spreadsheet).

would be expressed in SQL by

SELECT PERSON
FROM CAN_USE
WHERE SW = 'Spreadsheet'

and the two-variable goal

can-use (X, Y).

is expressed by

SELECT PERSON, SW
FROM CAN_USE

SQL and other relational query languages have some major limitations as representatives of
the logic model. First, the definition of relations binds each attribute to a specific type, in violation
of the typeless definition of objects in the logic model. The use of views is also much more limited
than the construction of rules, particularly when rules are recursively defined. Views in SQL may
not be defined in terms of themselves. Similarly, multiple definitions of the same rule can lead to
some complex logical manipulation and the creation of auxiliary views in SQL.

12.4 Non-Logic Model Features of Prolog

There are a number of features of Prolog that violate the spirit of the logic model. These gen-
erally provide imperative capabilities, which permit a more useful or efficient implementation of
the logic model in practice.

The first variance is that the order of search for a derivation is in the order that the rules and
facts are entered in the program. This order does make the efficiency and the termination of deri-
vations dependent on the ordering within the program and, as discussed in Section 12.2, is in vio-
lation of the strict logic model, where all “how” information is excluded. This distinction will be
illustrated by examples that appear later in this chapter.

The most famous of the nonlogic features of Prolog is the cut. The purpose of the cut is to
control backtracking during the derivation process. Therefore, it is very much a part of the “how”
related to a Prolog program. As an illustration of the use of the cut, let us add a second rule for
can_run to our Prolog program given in Figure 12.5, namely, the one corresponding to the rule
introduced in Section 12.2 regarding the fact that one software product may be written in the lan-
guage of another. If we make a more intelligent choice for the order of the two clauses in the
hypothesis, our two rules are

\[
\begin{align*}
\text{can_run} (\text{MACH}, \text{SW}) & :\text{spec} (\text{MACH}, \text{HW}, \text{MEM}1), \\
& \text{runs} (\text{HW}, \text{SW}, \text{MEM}2), \\
& \text{MEM}1 \geq \text{MEM}2. \\
\text{can_run} (\text{MACH}, \text{SW}) & :\text{written_in} (\text{SW}, \text{L}), \\
& \text{can_run} (\text{MACH}, \text{L}).
\end{align*}
\]

If we run the corresponding program with the goal

?-can_use(X,spreadsheet).

we will obtain the following output:
Notice that Jerry is derived twice, indicating that the program is doing some unnecessary work. Let's see why this happened. The object ‘Jerry’ satisfies can_use once because Jerry has access to mach3, which can run a spreadsheet directly. But the goal can also be derived from the second rule for can_run, because Pascal can run on mach3 as well and the spreadsheet is written in Pascal.

The cut is considered to be a subgoal that is universally satisfied, but that cannot be backed up over. It then stands as a one-way gate for the subgoal generation process, which, once passed, cannot be backed through.

**Figure 12.12 Derivation of a goal in Prolog using cut**

1. ?-can_use(X,spreadsheet).
   R1/\SW:=spreadsheet
2. ?-access(X,M),can_run(M,spreadsheet).
   F9/M:=mach1,X:=sue
3. ?-can_run(mach1,spreadsheet).
   R2
4. ?-spec(mach1,HW,M1),runs(HW,spreadsheet,M2),M1>=M2,!.
   F1/\HW:=ibmpc,M1:=320
5. ?-runs(ibmpc,spreadsheet,M2),320>=M2,!.
   F4/M2:=500
6. ?-320>=500,!.    Fails
7. ?-runs(ibmpc,spreadsheet,M2),320>=M2,!.
   Fails
8. ?-spec(mach1,HW,M1),runs(HW,spreadsheet,M2),M1>=M2,!.
   Fails
9. ?-can_run(mach1,spreadsheet).
   R3
10. ?-written_in(spreadsheet,L),can_run(mach1,L).
    F13/L:=pascal
11. ?-can_run(mach1,pascal).
    R2
12. ?-spec(mach1,HW,M1),runs(HW,pascal,M2),M1>=M2,!.
    F1/\HW:=ibmpc,M1:=640
13. ?-runs(ibmpc,pascal,M2),640>=M2,!.
    F4/M2:=500
14. ?-640>=500,!.
    Fails
15. ?-! Skip backup because of cut - 5. completed
    satisfied X=sue
16. ?-written_in(spreadsheet,L),can_run(mach1,L).
    Fails
17. ?-can_run(mach1,spreadsheet).
    Fails
18. ?-access(X,M),can_run(M,spreadsheet).
    F10/X:=jerry,M:=mach3
19. ?-can_run(mach3,spreadsheet).
    R2
20. ?-spec(mach3,HW,M1),runs(HW,spreadsheet,M2),M1>=M2,!.
    F3/M1:=640,\HW:=ibmpc
21. ?-runs(ibmpc,spreadsheet,M2),640>=M2,!.
    F4/M2:=500
22. ?-640>=500,!.
    Fails
23. ?-! Skip backup because of cut - 3. completed
    satisfied X=jerry
24. ?-access(X,M),can_run(M,spreadsheet).
    F11/X:=sam,M:=mach1
25. ?-can_run(mach1,spreadsheet).
    R2
26. ?-spec(mach1,HW,M1),runs(HW,spreadsheet,M2),M1>=M2,!.
    F1/\HW:=ibmpc,M1:=320
27. ?-runs(ibmpc,spreadsheet,M2),320>=M2,!.
    F4/M2:=500
28. ?-320>=500,!.
    Fails
29. ?-runs(ibmpc,spreadsheet,M2),320>=M2,!.
    Fails
30. ?-spec(mach1,HW,M1),runs(HW,spreadsheet,M2),M1>=M2,!.
    Fails
31. ?-can_run(mach1,spreadsheet).
    R3
32. ?-written_in(spreadsheet,L),can_run(mach1,L).
    F13/L:=pascal
33. ?-can_run(mach1,pascal).
    R2
34. ?-spec(mach1,HW,M1),runs(HW,pascal,M2),M1>=M2,!.
    F1/\HW:=ibmpc,M1:=320
35. ?-runs(ibmpc,pascal,M2),320>=M2,!.
    F6/M2:=256
36. ?-320>=256,!.
    satisfied
37. ?-! Skip backup because of cut - 3. completed
    satisfied X=sam
38. ?-written_in(spreadsheet,L),can_run(mach1,L).
    Fails
39. ?-can_run(mach1,spreadsheet).
    Fails
40. ?-access(X,M),can_run(M,spreadsheet).
    F12/X:=sam,M:=mach2
2. ?-access(X,M),can_run(M,spreadsheet).                           Fails
1. ?-can_use(X,spreadsheet).                                      Fails

In the preceding example, the first rule for can_run could be rewritten with the cut as follows:

can_run(MACH,SW) :- spec(MACH,HW,MEM1),
                  runs(HW,SW,MEM2),
                  MEM1 >= MEM2,
                  !.

After the first three subgoals are satisfied, the cut is then satisfied as well, completing the
derivation of can_run. After the entire derivation is completed, the cut then prevents the backing
up process from trying other alternatives for can_run, so unnecessary checking is avoided,
because it is irrelevant how many ways a computer can_run a software product.

Now the goal

?-can_use(X,spreadsheet)

will find only one derivation for each of the three people; the derivation process is displayed in
Figure 12.12. The derivation process, when backing up to a cut, will automatically produce a fail-
ure for all subgoals that have arisen since the generation of the subgoal that contains the cut.

Prolog also includes relations called asserta and assertz, which add new facts to the
beginning or end of the search list. These relations, when included within a rule, imply that the
logical derivation actually changes the facts. In pure logic programming, the derivation should
have no effect on the set of facts that hold. Similarly, the relation retract, which removes a fact
from the collection, is not a part of the logic model for the same reason.

Prolog also provides input and output facilities. By the structure of the language, these must
be specified within a derivation through the testing of built-in rules as subgoals. For example, the
goal of the form read(Variable) will bind the variable to the next data value appearing in the
input stream. A more direct assignment of a numeric value to a variable is possible through the
use of the infix operator ‘is’. Similarly, write(value) writes the specified value to the out-
put file. These input/output commands violate the spirit of pure logic programming by permitting
a derivation to “do” something as a side effect.

Finally, Prolog adds to the pure logic model by including data structures within the language.
You will recall that the pure logic model included only objects that were atomic items whose
meanings were not a part of the language.

Prolog permits structures and lists. We discuss only the list feature here. A list in Prolog is
very similar to a list in Scheme as detailed in Chapter 11. The Prolog list is defined as an ordered sequence of objects and lists. A list is written using its elements, which are separated by commas and enclosed in brackets. For example,

\[
[\ ] \\
[a,b,c] \\
[a,b,[c,d],e]
\]

are all lists. The first example illustrates the empty list while the third shows that list elements may be lists themselves. The head of a list is the first element in the list. The tail of the list is the list that remains after the first element is removed. Examples of the heads and tails of lists are shown here:

<table>
<thead>
<tr>
<th>List</th>
<th>Head</th>
<th>Tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>[a]</td>
<td>a</td>
<td>[ ]</td>
</tr>
<tr>
<td>[a,b]</td>
<td>a</td>
<td>[b]</td>
</tr>
<tr>
<td>[[x,y],a]</td>
<td>[x,y]</td>
<td>[a]</td>
</tr>
<tr>
<td>[[x,y],[a,b]]</td>
<td>[x,y]</td>
<td>[[a,b]]</td>
</tr>
</tbody>
</table>

In Prolog, the head and the tail of the list can be specified by writing a list as

\[
[X \mid Y]
\]

If this is in place for a variable binding, X and Y will be bound to the head and tail, respectively, of the list. For example, a relation that defines the head of a list can be written as

\[
\text{head}([H \mid T], H).
\]

Then the goal

?\text{-head}([a,b,c],X)

returns

\[
X = a
\]

Similarly, tail can be written

\[
\text{tail}([H \mid T],T).
\]

Both head and tail can be found in one rule by

\[
\text{list}([H \mid T],H,T).
\]

Note that this last fact can be used to find the head or tail of a list as in

?\text{-list}([a,b,c],X,Y)

X = a
Y = [b, c]

or to splice a list together as in

?-list(L,a,[b,c]).
L = [a,b,c]

12.5 Example Program in Prolog: Quicksort

Quicksort is conveniently implemented as a recursive rule in Prolog. We assume that the general form of the quick rule is

quick(UnsortedList,SortedList)

where both variables represent lists, the first serving as the input and the second the sorted output. A sample goal using quick might then be

?-quick([12,19,9,4,14,16],X)

from which Prolog derives

X = [4,9,12,14,16,19]

The quick rule is defined using two other rules, append and partition. The rule append is of the form

append(List1,List2,AppendedList)

where List1 and List2 are the two lists to be appended and AppendedList is the result. For example, given

?-append([1,2,3],[4,5],X)

Prolog will derive

X = [1,2,3,4,5]

The rule Partition is defined by

partition(Pivot,List,LessThanPivot,NotLessThanPivot)

Here, Pivot is any object and List is any list of objects. The rule partition results in all objects in List which are less than the pivot value being placed in LessThanPivot while all objects greater than or equal to the pivot are placed in the list NotLessThanPivot.

We will specify append and pivot later. Given their availability, we can define quick as follows:

quick([],[]).
quick([Pivot | Remainder],SortedList) :-
    partition(Pivot,Remainder,BeforePivot,AfterPivot),
quick(BeforePivot,SortedBefore),
quick(AfterPivot,SortedAfter),
append(SortedBefore, [Pivot|SortedAfter], SortedList).

The first rule for quick is the recursion stopper. It says that the empty list when sorted is the empty list. The second rule indicates that Pivot is bound to the first object in the input list, and Remainder is bound to the rest of the list. SortedList is bound within the subgoals.

There are four subgoals for this second rule. The first is that partition is derived for list Remainder with pivot element Pivot. The two sublists generated by the partitioning are bound to BeforePivot and AfterPivot.

Next, the subgoal quick is recursively derived twice with BeforePivot as input and SortedBefore as output and with AfterPivot as input and SortedAfter as output. These two derivations result in SortedBefore and SortedAfter being bound to the corresponding two sorted sublists. These sublists are then joined together with the pivot element in between by the append subgoal, binding the resulting list to SortedList.

The predicate append is easily constructed. Its form is

\[
\begin{align*}
\text{append}([], \text{List2}, \text{List2}).
\text{append}([\text{Head}|\text{Tail}], \text{List2}, [\text{Head}|\text{NewList}]) & :\text{- append(Tail, List2, NewList)}.
\end{align*}
\]

The first rule says that the empty list appended to any list is that list itself. The second rule is recursive and indicates how a list can be appended based on the application of append on its tail.

Next we examine the rules to implement partition.

\[
\begin{align*}
\text{partition}(\text{Pivot}, [], [], []). \\
\text{partition}(\text{Pivot}, [\text{Head}|\text{Tail}], [\text{Head}|\text{BeforePivot}], \text{AfterPivot}) & :\text{-} \\
& \\text{Head<}\text{Pivot}, \\
& \text{partition}(\text{Pivot}, \text{Tail}, \text{BeforePivot}, \text{AfterPivot}), \\
& \!.
\end{align*}
\]

partition(Pivot, [Head|Tail], BeforePivot, [Head|AfterPivot])  :-
partition(Pivot, Tail, BeforePivot, AfterPivot).

The first of the preceding rules indicates that the empty list is partitioned into two empty lists, no matter what the pivot value. This serves as the recursion stopper. The second rule says that if the head of the list being partitioned is less than the pivot and if the tail of the list is partitioned into BeforePivot and AfterPivot, then the original list is partitioned by the list formed when Head is attached to BeforePivot and that list is attached to AfterPivot. The cut is included in this definition to prevent attempting the third rule for partition when the second one is derived. The third rule is then only reached if Head is greater than or equal to Pivot and its meaning is obvious.

The combination of the definitions for quick, append, and partition form a complete version of quicksort.

12.6 Example Program in Prolog: Finding the Shortest Path

Problems that require searching are good candidates for solution in Prolog since the built-in search strategy of Prolog can be effectively put to use. In this section we examine definitions that solve the problem of finding the shortest path between two nodes in a directed graph.

We assume that nodes are atomic objects in our solution and paths are lists of nodes. For
example, a path from $a$ to $b$ passing through $g$ and $h$ successively is represented by the list

$$[a, g, h, b]$$

Lengths of the edges between two nodes and the lengths of paths will be represented by integers.

A graph is represented by a set of adjacent relations where

$$\text{adjacent}(X,Y,L)$$

means there is a directed edge from node $X$ to node $Y$ of length $L$. For example, Figure 12.14 illustrates a directed graph and the set of adjacent relations required to represent that graph in Prolog. The problem that we wish to solve is this: Given a directed graph and two nodes, $A$ and $B$, of that graph, find the shortest path from $A$ to $B$.

We begin with a simpler problem, that of finding all paths from $A$ to $B$. The Prolog predicate used to do this is

$$\text{trip}(\text{Start}, \text{Finish}, \text{Path}, \text{Length})$$

where $\text{Start}$ and $\text{Finish}$ are the two given terminating nodes, $\text{Path}$ is a derived path, and $\text{Length}$ is the total length of $\text{Path}$. The relation $\text{trip}$ derives all paths from $\text{Start}$ to $\text{Finish}$ that do not make repeat visits to any nodes, i.e., paths that do not cross themselves.

Before we describe the rules for trip, we first need to define two relations that greatly facilitate trip’s definition. The first is a general list predicate called $\text{not_in}$, which is of the form

$\text{not_in}(\text{Node}, \text{Path})$

where $\text{Node}$ is an atomic node object and $\text{Path}$ is a list of nodes representing a path in the directed graph. This rule derives successfully if $\text{Node}$ is not one of the members of $\text{Path}$ and fails to derive if $\text{Node}$ is a member of $\text{Path}$. It is defined by

$$\text{not_in}(\text{Node}, []) :- !. $$
$$\text{not_in}(\text{Node}, [\text{Head}|\text{Tail}]) :- \text{Node}<>\text{Head}, $$
$$\text{not_in}(\text{Node}, \text{Tail}). $$

We are now prepared for the workhorse of this solution, a relation called path, which is described by

$$\text{path}(\text{Start}, \text{SubPath}, \text{SPLength}, \text{FinalPath}, \text{FPLength})$$

where $\text{Start}$ is a node, $\text{SubPath}$ is a path, and $\text{SPLength}$ is a length, all of which are provided to the derivation. $\text{FinalPath}$ and $\text{FPLength}$ are a path and length that are derived. $\text{FinalPath}$ is a path starting at node $\text{Start}$ and finishing with $\text{SubPath}$. The derived length of $\text{FinalPath}$ is $\text{FPLength}$.

The rules for path are

$$\text{path}(\text{Start}, [\text{Start}|\text{Tail}], L, [\text{Start}|\text{Tail}], L). $$
$$\text{path}(\text{Start}, [\text{Head}|\text{Tail}], \text{SPLength}, \text{Path}, \text{FPLength}) :- $$
$$\text{adjacent}(\text{NewHead}, \text{Head}, \text{EdgeLength}), $$
$$\text{not_in}(\text{NewHead}, \text{Tail}), $$
NewLength is EdgeLength+SPLength,  
path(Start,[NewHead,Head|Tail],NewLength,Path,PLength).

This definition requires some further explanation. The first rule simply states that if the given subpath already begins at node Start, we are done, and that subpath is the final path and its length the final length. This is the recursion stopper.

The second rule is more complex. It says that a subpath can be built upon if a new head can be added that is adjacent to the present head, the new head is not in the present subpath, and a path can be built from Start using the newly created subpath (made by adding the new head). The operator is, which is used in the third subgoal, is a nonlogic operator that always succeeds and causes the variable NewLength to be immediately bound to the specified sum. The recursion specified in this rule then causes all possible paths to be derived.

The rule that derives all possible paths from Start to Finish is now defined by

\[
\text{trip}(\text{Start}, \text{Finish}, \text{Path}, \text{Length}) \leftarrow \text{path}(\text{Start}, [\text{Finish}], 0, \text{Path}, \text{Length}).
\]

In other words, there is a path Path of length Length from node Start to node Finish if the Path can be built from the subpath of length zero consisting only of node Finish and starts at node Start.

In order to clarify the actions of these rules, we examine a simple derivation in Figure 12.14. Note how the Prolog search for solutions results in all solutions being found. The last attempt to use the edge from d to b illustrates how the not_in predicate is used to prevent paths that cross themselves.

The next challenge, now that we can find all paths between two nodes, is to derive the shortest of these paths. This solution is derived in a way that illustrates the expressive power of Prolog. The solution is simply stated as

\[
\text{mintrip}(\text{Start}, \text{Finish}, \text{MinPath}, \text{MinLength}) \leftarrow \\
\text{trip}(\text{Start}, \text{Finish}, \text{MinPath}, \text{MinLength}), \\
\text{smallest}(\text{Start}, \text{Finish}, \text{MinLength}).
\]

This result says that MinPath is the shortest path if that path results in a successful derivation of trip and if the as yet undefined rule smallest can also be derived. The relation smallest is derived if there is no trip from Start to Finish shorter than MinLength. The easiest way to determine this is to again generate all trips from Start to Finish and fail the predicate smallest in any case where the generated trip length is greater than the value of MinLength. If no failure is generated, the predicate will succeed. This is written in Prolog as

\[
\text{smallest}(\text{Start}, \text{Finish}, \text{MinLength}) \leftarrow \\
\text{trip}(\text{Start}, \text{Finish}, \text{Path}, \text{PLength}), \\
\text{PLength}<\text{MinLength}, \\
!, \\
\text{fail}. \\
\text{smallest}(\text{Start}, \text{Finish}, \text{MinLength}).
\]

This definition uses the built-in Prolog predicate fail, which fails whenever it is encountered. In the first rule, when a path is found with length shorter than MinLength, immediately we want to fail the derivation. The predicate fail does this and the cut preceding fail prevents
any attempt to derive an alternative rule for smallest.

Notice that in the derivation of mintrip for two nodes, Start and Finish, if there are N different paths (as derived by trip) from Start to Finish, for each path, smallest will try to compare by generating paths and comparing lengths until a path is found that is shorter than the present path, or until all paths have been compared.

We can see that the order in which paths are generated can make a big difference in the efficiency of the derivation. Consider the example of Figure 12.13. It happens, in this case, that the paths are generated in increasing order by path length. The comparisons for shortest are then listed as follows:

<table>
<thead>
<tr>
<th>Test path</th>
<th>Test against</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a,d]</td>
<td>[a,d],[a,b,d],[a,c,b,d]</td>
</tr>
<tr>
<td>[a,b,d]</td>
<td>[a,d],fails</td>
</tr>
<tr>
<td>[a,c,b,d]</td>
<td>[a,d],fails</td>
</tr>
</tbody>
</table>

In total, 8 trips were generated in this derivation. If the trips had been generated in decreasing order of length, then 3 trips would need to be compared to each test path, and the total trips generated would be 12 as follows:

<table>
<thead>
<tr>
<th>Test path</th>
<th>Test against</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a,c,b,d]</td>
<td>[a,c,b,d],[a,b,d],[a,d]</td>
</tr>
<tr>
<td>[a,b,d]</td>
<td>[a,c,b,d],[a,b,d],[a,d]</td>
</tr>
<tr>
<td>[a,d]</td>
<td>[a,c,b,d],[a,b,d],[a,d]</td>
</tr>
</tbody>
</table>

For general N, these figures are $3N-1$ for increasing length generation and $N^2+N$ for decreasing, making the difference between $O(N)$ and $O(N^2)$ solutions. In practice, the solution presented cannot guarantee the $O(N)$ solution, but the fact that the paths are generated in the order of increasing number of nodes in the path makes the increasing order of length more plausible and leads to a reasonably efficient solution.

**Figure 12.13 Prolog representation of a directed graph**

```
adjacent(a,b,5).
adjacent(a,c,6).
adjacent(a,d,8).
adjacent(b,d,4).
adjacent(c,b,7).
adjacent(d,b,4).
```

**Figure 12.14 Derivation of path from node a to node d**

```
path(a,[d],0,P,L).
adj(a,d,8),not_in(a,[]),NewLength is 8+0,path(a,[a,d],8,P,L).
/P:=[a,d],L:=8
adj(b,d,4),not_in(b,[]),NewLength is 4,path(a,[b,d],4,P,L).
  adj(a,b,5),not_in(a,[d]),NewLength is 5+4,path(a,[a,b,d],9,P,L).
    /P:=[a,b,d],L:=9
adj(c,b,7),not_in(c,[d]),NewLength is 7+4,path(a,[c,b,d],11,P,L),
```
adj(a, c, 6), not_in(a, [b, d]), NewLength is 6+11, path(a, [a, c, b, d], 17, P, L).
\[ P := [a, c, b, d], L := 17 \]
adj(d, b, 4), not_in(d, [d]), ...
\[ \text{FAILS} \]

Terms - Chapter 12

- axiom
- object
- relation
- binding
- inference engine
- resolution
- unification
- backtracking
- relation
- tuple
- attribute
- query language
- cut
- relation
- list

Discussion Questions - Chapter 12

1. The logic language model separates the statement of the problem from the method used to solve it. What are the advantages and disadvantages of this approach?

2. To write programs for other programming language models—for example, the imperative model—you would use a design method like top-down design. How does the logic-oriented model change your problem-solving methodology? What type of method would be useful for logic-oriented programming?

3. “Programs” in a logic-oriented programming environment consist of sets of facts and rules and follow-up goals that are solved by the inference engine. Determine the types of problems you could solve with this approach. What types of problems are best suited for logic-oriented programming?

4. How do you suppose programs are “debugged” in a logic-oriented programming environment? Are bugs easier or harder to spot than with imperative or functional programs?

5. Discuss the usefulness of the principles of resolution and unification in everyday logic. Give examples of their application.

6. Does SQL use resolution or unification? In what way?

7. Describe the implementation restrictions on the inference engine that might have to be applied with SQL.

8. The logic language model separates the statement of the problem from the method used to solve it. What are the advantages and disadvantages of this approach?

9. Prolog and SQL take very different approaches to implementing the logic model. Compare
the two approaches and discuss their relative advantages.

10. Discuss the usefulness of the principles of resolution and unification in everyday logic. Give examples of their application.

11. Discuss the cut feature of Prolog. What kinds of programs could not be written if the cut were missing? Is the cut a convenience or a necessity?

12. Why would the search order for facts and rules in the knowledge base have to be stipulated for Prolog?

13. Discuss uses for `asserta` and `assertz`. How could `retract` be used?

14. Discuss the advantages and disadvantages of using many facts and few rules versus few facts and many rules in a Prolog program.

**Exercises - Chapter 12**

1. Give the result of applying resolution and unification to the following groups of clauses

   (a) \[ a(T) :- b, c(T) \]
   \[ b :- d, e \]

   (b) \[ a(T) :- b(T), c(T) \]
   \[ c(x) :- d(x), e \]

   (c) \[ a(T) :- b(T), c(T) \]
   \[ c(x) \]

   (d) \[ a(T) :- b(T), c(T) \]
   \[ b(x) :- d(x) \]
   \[ c(t) :- d(t) \]

   (e) \[ a :- b(T) \]
   \[ b(T) :- c(x), d(T) \]
   \[ c(T) :- b(T), e(T) \]
   \[ e(x) \]

   (f) \[ a(x) :- b(y), c \]
   \[ b(T) :- c, d(T) \]
   \[ d(y) :- a(y), e \]

2. Construct logical goals and rules for `prime(X)` that will verify that `X` is a prime number (for unbound `X`).

3. Using the Facts and Rules given in Figures 12.4 and 12.5, construct the derivation tree of the following:
(a) can_use(jerry, pascal).

(b) can_use(sam, smalltalk).

4. Consider the facts and rules from Figures 12.4 and 12.5. Which of the following goal definitions determine if one could maintain software on a particular machine?

(a) can_maintain(P, SW) :- can_use(P, SW), written_in(SW, L), knows(P, L).

(b) can_maintain(P, SW) :- acess(P, M), written_in(SW, L), can_run(M, SW).

(c) can_maintain(P, SW) :- written_in(SW, L), can_use(P, SW), knows(P, L).
    knows(P, L) :- access(P, L).

(d) can_maintain(P, SW) :- written_in(SW, L), can_use(P, SW),
    knows(P, L).
    knows(P, L) :- access(P, L), runs(M, L, MEM).

5. Derive the following goals using backtracking. Your answers should be given in a manner similar to Figure 12.7.

(a) can_use(sam, X).

(b) can_use(X, basic).

(c) Can_run(X, Y).

6. Given the facts and rules of Figures 12.4 and 12.5, why does the following query fail?

?- can_use(jerry, smalltalk).

7. Consider the Prolog knowledge base below:

parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
ancestor(X, Z) :- parent(X, Z).
ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).
sibling(X, Y) :- mother(M, X), mother(M, Y),
                  father(F, X), father(F, Y), X \= Y.

father(albert, jeffrey).
mother(alice, jeffrey).
father(albert, george).
mother(alice, george).
father(george, cindy).
mother(mary, cindy).

The question posed to the Prolog system
?- ancestor(X,cindy), sibling(X, jeffrey).

produces an answer of

X = george

Show the derivation of this answer in detail by tracing the derivation of the solution using resolution, unification and backtracking.

8. Let’s add the following rules and facts to the knowledge base from the preceding exercise:

   grandchild(C,G) :- parent(G, C1), parent(C1, C).
   greatgrandchild(C,G) :- parent(G, C1), grandchild(C, C1).

   father(jim, fred).
   mother(beth, fred).
   father(jim, ann).
   mother(beth, ann).
   father(fred, joyce).
   mother(helen, joyce).
   father(fred, albert).
   mother(helen, albert).

   If the goal is

   greatgrandchild(X, Y).

   how many answers do we get? What are they?

9. Use the rules of Exercises 1 and 2 and write a general greatgrandchild goal. That is, the goal

   greatgrandchild(X, Y, 1)

   will find all great grandchildren, whereas greatgrandchild(X, Y, 2) will find all great-great grandchildren, and so forth.

10. Show a derivation of

    head([[a, b], x, y], H)

    to produce H = [a, b].

11. Show a derivation of

    tail([[a, b], x, y], T)

    to produce T = [x, y].

12. Consider the append predicate given in section 12.5. Based on this predicate, develop a member predicate that verifies that an item is in a list.

Laboratory Exercises -- Chapter 12

1. Write a program that is a parser for the Prolog subset described in Figure 12.1.

2. Write a program to perform resolution. Assume that facts are represented by a single upper-case letter and the rule is input as the letter for the consequence fact followed by the letter for the hypothesis facts. For example, the rule
\[ f := a, b, c \]

is input to the program as

\[ fabc \]

Your program is to input two rules of this form and output the rule derived by resolution. Note that the order in which the hypothesis facts appear in the input is arbitrary.

3. Implement the quicksort algorithm given in Section 12.5 and verify that it does what it claims.

4. Implement a solution to the Towers of Hanoi puzzle. Your solution should define the predicate `hanoi` by

\[
\text{hanoi} (N) :- \text{move} (N, \text{left}, \text{center}, \text{right})
\]

to move \( N \) discs from tower \( \text{left} \) to tower \( \text{center} \) using tower \( \text{right} \) to assist. Implement this recursively by moving \( N-1 \) from \( \text{left} \) to \( \text{right} \), writing a message for moving one from \( \text{left} \) to \( \text{center} \), then moving \( N-1 \) from \( \text{right} \) to \( \text{center} \).

5. Using the list structure, implement a Stack structure. Operations should have an invocation similar to

\[
\text{pop}([a, b, c], I)
\]

and \( I \) would contain \( a \) after the operation.

6. Repeat Laboratory Exercise 5 using a queue instead of a stack, and `dequeue` instead of `pop`.

7. Write a program that “learns”—that is, interactively adds to the knowledge base when it has been determined that knowledge is insufficient. For example, write a Prolog program to learn how to guess animals. The program could be “preprogrammed” with some rules and facts and questions based on those rules and facts, such as:

\[
\text{Does the animal fly? or Does the animal have four legs?}
\]

Notice that this program is interactive. Upon running out of questions, the program should request new information: What is the animal name and what question distinguishes this animal from a <name of animal that has the same answers to all present questions>?

8. Verify your answers to Exercises 10 and 11 by writing “head” and “tail” in Prolog and demonstrating their use.

9. Implement the family tree facts and rules scattered throughout this chapter in Prolog. Verify that our answers work correctly. In addition, implement a new rule

\[
\text{ancestor}(X, Y)
\]

that checks that \( X \) is some ancestor of \( Y \).