1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

http://hdl.handle.net/10945/36984

Variables and queries

We can put facts into a computer. So what can we do with them? Well, we want to *reason* about facts and conclude new facts--what's called *inference*. For this we'll need the concepts of queries, variables, and backtracking.

Querying the facts

One thing we can do with facts in a computer is to look them up. This is the usual mode of *Prolog interpreters*, software that interprets and executes code written in the Prolog language | REFERENCE 1|: they wait for us to give them things they can try to look up. .FS | REFERENCE 1| Most Prolog-understanding software are interpreters like this and not *compilers*. A Prolog interpreter is not an "artificial intelligence program" but a tool to execute artificial-intelligence programs written in the Prolog language. .FE You're in this *query mode* when the Prolog interpreter types **?-** at the front of every line. Query mode is the way database query languages work, like SQL and QUEL.

To make this clearer, assume these facts (the semantic network example from Section 2.7) have been entered into a computer running a Prolog interpreter:

```
a_kind_of(enterprise,ship).
a_kind_of(kennedy,ship).
part_of(enterprise,u_s_navy).
part_of(kennedy,u_s_navy).
part_of(u_s_navy,u_s_government).
a_kind_of(u_s_government,government).
color(ship,gray).
location(enterprise,15n35e).
has(u_s_government,civil_service_system).
```

We call such a set of facts known to a Prolog interpreter a *Prolog database* or just *database*. As we'll explain shortly, databases can be loaded from files. The block diagram in Figure 3-1 summarizes these basics.

Now in query mode we can type

```
part_of(kennedy,u_s_navy).
```

so that what actually shows on the computer terminal will be

```
?- part_of(kennedy,u_s_navy).
```

Note the period; the interpreter will wait (forever if necessary) until we type it. Then the interpreter will type in reply the single word **yes** to acknowledge that the fact is in its database. If we ask instead

```
?- part_of(pequod,u_s_navy).
```

(again the **?-** is typed by the interpreter and not us), the computer will type the single word **no**. So **yes** means "I found it" and **no** means "I couldn't find it". We call a **yes** a query *success* and **no** a query *failure*. So to make the computer say **no** when a query is *false*, the database must include every truth about its subject, for otherwise **no** could mean incomplete data.

Queries with one variable

But this isn't too interesting. A query must give the precise fact we want to look up, including every argument. We might instead want to ask if a **part_of** fact has **enterprise** as its first argument and anything at all as its second argument. We can do this by querying

```
?- part_of(enterprise,X).
```

Read this as "Find me an "X" such that **part_of(enterprise,X)** is true," or simply as "What is the Enterprise part of?" The Prolog interpreter will go through its facts in order, trying to match each to the query. When it finds one that matches in predicate name and first argument, it will type "**X=**" followed by the fact's second argument, instead of typing **yes**. Or in technical jargon, it *binds* or *matches* **X** to a value and prints it. So for this query with the previous database, we will get

```
X=u_s_navy
```

or **X** is bound to **u_s_navy**.

X here is a *variable*. Prolog variables have similarities to variables in other programming languages, but also important differences we'll encounter as we proceed. Prolog variables are designated by a capitalized first letter in a word (followed by other letters and numbers, either capitalized or uncapitalized), and this is why in the last chapter we used lower case for other words in Prolog.

Variables can only be arguments in Prolog; they can't appear as predicate names (though we'll give a way around this limitation in Chapter 12). This means Prolog represents only *first-order logic*. First-order logic is sufficient for nearly all artificial-intelligence applications, so that's no big deal. First-order logic is a reason we insisted in Chapter 2 on predicate names more general than their argument names: variables pay off when they stand for lots of possibilities.

Multi-directional queries

A variable can appear anywhere among the arguments to a predicate expression in a query, with some exceptions to be discussed later. So we could also query with the previous database

```
?- part_of(X,u_s_navy).
```

and the Prolog interpreter will type back

```
X=enterprise
```

In other words, we can be flexible about which arguments are *inputs* (constants) and which are *outputs* (variables) in a query. This means Prolog can answer quite different questions depending on where we put variables in the query. This flexibility extends to calls of Prolog procedures (subroutines and functions) too, as you will see in Chapter 4, a big difference from most programming languages.

Matching alternatives

More than one thing (value) can match (bind) a query variable. The Prolog interpreter will find the first, print it out, and stop and wait If just one is sufficient, type a carriage return. But to see the next answer (if any),

type a semicolon ("**;**"). We can keep typing semicolons, and it will keep finding new matches, until it can't find any more and it must answer **no**. So for our example database if we query

```
?- a_kind_of(X,ship).
```

which means "Find me an X that's a kind of ship," the interpreter will first type

```
X=enterprise
```

and then if we type a semicolon it will type

```
X=kennedy
```

and then if we type a semicolon it will type **no**. The semicolon prints at the end of the line, so what this will all look like on the computer terminal will be:

```
?- a_kind_of(X,ship).
X=enterprise;
X=kennedy;
no
```

where we typed the two semicolons and the first line except for the "**?-** ", and the interpreter typed the rest.

We can have more than one variable in a query. If we were to query for our example database

```
?- part_of(X,Y).
```

("What things are part of other things?") and we kept typing semicolons, we would eventually see on the computer terminal some like this (some Prolog dialects format this output slightly differently):

```
X=enterprise, Y=u_s_navy;
X=kennedy, Y=u_s_navy;
X=u_s_navy, Y=u_s_government;
no
```

So semicolons find us every combination of bindings of the variables that satisfies a query. Since the Prolog interpreter works top to bottom through the database, the bindings will reflect database order.

Multi-condition queries

A Prolog interpreter also lets us specify that several different conditions must succeed together in a query. This lets us specify "chains of reasoning", like those so important to detectives in mystery fiction.

Suppose we wanted to know the gray-color of the Enterprise. If we type

```
?- color(enterprise,C).
```

we get **no** with our example database, because the color fact is about ships in general and not the Enterprise. This problem of information in the "wrong place" happens often in artificial-intelligence systems. Instead we can ask if there is some category or type T that the Enterprise belongs to, such that everything of type T has color C:

```
?- a_kind_of(enterprise,T), color(T,C).
```

This represents an "and" (*conjunction*) of two predicate expressions, both of which must succeed for the whole match to succeed. It works this way: we first try to answer the query

```
?- a_kind_of(enterprise,T).
```

Then for that particular **T**, we answer the query

```
?- color(T,C).
```

Using our example database, we first match **T** to **ship** in the first-listed fact. We then look for a **color** fact in which this **T** is the first argument, and the seventh-listed fact qualifies; we can then match **C** to **gray**. The Prolog interpreter now types out:

```
T=ship, C=gray
```

So commas between predicate expressions in a query line mean reusing the same values for any same-named variables. Commas are like a logical "and" since all the subqueries (predicate expressions) must succeed for the whole query to succeed. To make commas easier to spot, we'll often put spaces after them in queries; these spaces are ignored by the interpreter. (But don't put spaces in predicate expressions.)

As another example, suppose we want to know what the Enterprise is part of. We could say

```
?- part_of(enterprise,X).
```

and get **X=u_s_navy**, but that's not the only reasonable answer since the U.S. Navy is part of something else. So we could say:

```
?- part_of(enterprise,X), part_of(X,Y).
```

and get back **X=u_s_navy, Y=u_s_government**.

Logical "or" (*disjunction*) is represented by a semicolon instead of a comma. For instance

```
?- color(enterprise,C); color(ship,C).
```

asks for the color of the Enterprise if any is recorded, otherwise the color of ships in general. Parentheses can group which predicate expressions go with which others in "and"s and "or"s. So for instance the two conditions under which something is part of something else could be compressed into one with:

```
?- part_of(enterprise,X); (part_of(enterprise,Y), part_of(Y,X)).
```

This reads: "Find me an X such that either the Enterprise is part of it, or the Enterprise is part of some Y that is part of it." We won't use these "or" semicolons much, because (1) "and"s occur more often in applications, (2) they often require parentheses and so are hard to read, and (3) there is a better way to get the effect of an "or", to be discussed in the next chapter.

Figure 3-2 should help you keep straight the special symbols we've used so far, plus previewing a few to come.

Negative predicate expressions

So we have "and"s and "or"s. All we need to complete a Boolean algebra is a negation or "not". This is

accomplished by the built-in predicate **not** whose one argument is a predicate expression. (A *built-in* predicate is one with special meaning to the interpreter, a meaning not given by facts.) A **not** succeeds whenever querying its argument fails, a fails whenever querying its argument succeeds. So the query

```
?- not(color(enterprise,green)).
```

will succeed whenever there's no fact that the color of the Enterprise is green, and fail when there is such a fact. We'll extend the term "predicate expression" to include such **not** expressions too.

How will the Prolog interpreter ever be sure something is *not* true? Strictly speaking, it can't, since facts that directly say something is false are not permitted in Prolog (Chapter 14 discusses this further). So **not** is defined to mean the interpreter couldn't find a fact in its database--*negation-by-failure* or *the closed-world assumption*. Yet this is a curious and awkward interpretation of "not", not what we usually mean by the word in English. So we must be careful with **not** in Prolog. One big problem is that we can't, with a few exceptions, put unbound variables within a **not**. So this query won't work:

```
?- not(color(X,gray)), a_kind_of(X,ship).
```

(This asks for a ship **X** that isn't gray.) Instead we must reverse the order of the two things:

```
?- a_kind_of(X,ship), not(color(X,gray)).
```

Some query examples

Questions in English about a database often map directly into Prolog queries. Words like **is**, "are", "does", and "did" at the beginning of a question suggest queries without variables (yes/no queries). Words like "what", "which", "who", "where", "when", and "how" suggest variables.

Here are some examples. We assume the meanings of the **part_of**, **color**, **a_kind_of**, etc. predicates we've been assuming all along. (These queries print out additional variable values than those desired; Chapter 4 will explain how to prevent this.)

1. What things are part of gray things?

```
?- part_of(X,Y), color(Y,gray).
```

2. What things are part of parts of other things?

```
?- part_of(A,B), part_of(B,C).
```

3. What things are gray or blue?

```
?- color(T,gray); color(T,blue).
```

4. What isn't an example of a gray thing? (**Example** suggests the reverse of the **a_kind_of** relationship.)

```
?- a_kind_of(E,T), not(color(T,gray)).
```

5. What is the Enterprise, either directly or through one level of indirection?

```
?- a_kind_of(enterprise,J); (a_kind_of(enterprise,K), a_kind_of(K,J)).
```

6. What things of which the Enterprise is part, are themselves part of something that has a civil service system?

```
?- part_of(enterprise,U), part_of(U,V), has(V,civil_service_system).
```

Loading a database

How do we load a database of facts into a Prolog interpreter in the first place? This varies between implementations of Prolog interpreters, but usually we must first enter the facts we want into a text file, using an editor program. We exit the editor and start up the Prolog interpreter. We then query a special *built-in* loading predicate, called **consult** in this book. This **consult** is not something we must give facts for, but an internal Prolog name like **not**; it takes one argument, the name of a file to load into the Prolog interpreter's internal memory (database). From then on, the facts in that file will be used to answer queries.

For instance, suppose we use the editor to create a file called "test", containing:

```
boss(harry).
employee(tom).
employee(dick).
```

We can start the Prolog interpreter, type the query

```
?- consult(test).
```

and then type the query

```
?- employee(X).
```

and get

```
X=tom
```

from the first fact in the file that matches.

We can load more than one file into the Prolog interpreter, if several files contain useful facts. Just query **consult** again. New facts are put after the old facts, so you can get answers in a different order if you load the same files in a different order.

Backtracking

Let's consider in more detail how the Prolog interpreter answers complicated queries. To make this easier, consider for now queries with only commas ("and"s), no semicolons ("or"s) or **not**s.

Predicate expressions "and"ed in a query are first taken left to right. That is, the leftmost expression is tried first, then the second expression from the left (using whatever variable matches were found for the first) and so on. So predicate expressions in a query are initially done in order, like lines of a program in a conventional language like Pascal.

But suppose that a predicate expression fails--that is, no fact matching it can be found. If the expression has variables that were bound earlier in the query line, the fault may just be in the bindings. So the interpreter

automatically *backtracks* (goes back to the immediately previous expression in the query) and tries to find a different fact match. If it cannot, then *that* predicate expression fails and the interpreter backtracks to the previous one, and so on.

Anytime the Prolog interpreter cannot find another matching for the leftmost expression in a query, then there's no way the query could be satisfied; it types out the word **no** and stops. Anytime on backtracking it can find a new matching for some predicate expression, it resumes moving right from there as it did originally.

The purpose of backtracking is to give "second chances" to a query, by revising earlier decisions. Backtracking is very important in artificial intelligence, because many artificial-intelligence methods use intelligent guessing and following of hunches. Guesses may be wrong, and backtracking is a good way to recover then.

Here's an example:

```
?- part_of(X,Y), has(Y,civil_service_system).
```

which asks for an **X** that is part of some **Y** that has a civil service system. Assume the standard database example of this chapter. Then the only facts that will help with this query (the only facts with predicate names **part_of** and **has**) are:

```
part_of(enterprise,u_s_navy).
part_of(kennedy,u_s_navy).
part_of(u_s_navy,u_s_government).
has(u_s_government,civil_service_system).
```

Here in detail is what the Prolog interpreter does to answer this query:

1. It takes the first predicate expression in the query, and matches **X** to **enterprise**, and **Y** to **u_s_navy**. It stores the information that it has chosen the first fact to match the first expression.

2. It then moves to the second predicate expression, and tries to answer the *subquery*

   ```
   ?- has(u_s_navy,civil_service_system).
   ```

   That is, it "substitutes" in the value bound to variable **Y**. But the subquery fails since there's no such fact.

3. So it must backtrack, or return to the first predicate expression in the query. From its stored information, it knows it chose the first **part_of** fact last time, so now it tries the second, binding **X** to **kennedy** and **Y** to **u_s_navy**. It stores the information about what it chose.

4. It then tries to answer the subquery

   ```
   ?- has(u_s_navy,civil_service_system).
   ```

   This is the same query it did in step 2, but the interpreter is stupid and doesn't remember (Chapter 6 will explain how to force it to remember), so it checks the facts and fails to find anything again. The subquery fails.

5. So it backtracks again, to the first predicate expression in the query. It chose the second fact last time, so it now chooses the third (and last) **part_of** fact. So **X** is bound to **u_s_navy** and **Y** is bound to **u_s_government**.

6. The second expression is considered with the new binding for **Y**, and the interpreter tries to answer the subquery

```
?- has(u_s_government,civil_service_system).
```

And this succeeds because it's the fourth fact.

7. So both predicate expressions in the query succeed, and the whole query succeeds. The interpreter prints out the bindings that it found:

```
X=u_s_navy, Y=u_s_government
```

Notice that the interpreter wouldn't have had to backtrack if we just reversed the order of the query ("and" is commutative--see Appendix A):

```
?- has(Y,civil_service_system), part_of(X,Y).
```

because only one fact can match the **has** predicate expression. But that requires analyzing the facts in advance, and probably won't be true for a complete database for an application.

The automatic backtracking of the Prolog interpreter has both advantages and disadvantages. A big advantage is that combinatorial problems are easier to specify than with most computer languages, because the interpreter does more work for you. It also means that Prolog is a more flexible language than most: if you refer to an unbound variable in Pascal, Ada, PL/I, or Fortran, you get an error message and the program stops. The disadvantages are that Prolog programs run slower than those of other languages, and they're sometimes harder to understand and debug, because the language tries to do more.

A harder backtracking example: superbosses

Here is another backtracking example. It's trickier than the last because two predicate expressions both have alternatives. Furthermore, the same predicate name is used twice, and we have to distinguish the alternatives for each use.

Suppose we have facts about employees in an organization, represented with a two-argument predicate **boss**. Its first argument a boss, and its second argument is an employee of that boss. Take the following example database:

```
boss(dick,harry).
boss(tom,dick).
boss(ann,mary).
boss(mary,harry).
```

Suppose we want find "superbosses", people who are bosses of bosses. That is, those X that are a boss of some Y while at the same time Y is a boss of some Z. We can issue the query

```
?- boss(X,Y), boss(Y,Z).
```

and every match the interpreter finds for variable **X** will be a superboss. (Matches for **Y** and **Z** will also be found, but **X** is all that we want, according to the way we stated the problem.)

Let's trace query execution (summarized in Figure 3-3). As usual, assume facts are placed in the Prolog database in the order listed.

1. The first predicate expression in the query will match the first fact in the database, with **X=dick** and **Y=harry**.

2. Moving to the second predicate expression in the query, the interpreter searches for a **boss** fact with **harry** as its first argument. But there's no such fact in the database, so the second expression in the query fails.

3. So the interpreter backtracks, returning to the first expression to make another choice. Last time it used the first fact in the database, so this time it uses the second fact and sets **X=tom** and **Y=dick**.

4. Things proceed just as if these matchings happened originally. The interpreter goes to the second predicate expression, and searches for a **boss** fact where **dick** is the first argument. And yes, there is such a fact, the first fact in the database.

5. So **Z=harry**, and since we're at the end of the query, the query succeeds. Therefore Tom is a superboss. The interpreter types out **X=tom, Y=dick, Z=harry**.

Now we can explain better what typing a semicolon does after the Prolog interpreter types out a query answer (not to be confused with a semicolon in a *query:* it forces failure and backtracking. For instance, suppose after that answer **X=tom** we type a semicolon instead of a carriage return. What happens now is summarized in Figure 3-4, together with the previous events.

6. The interpreter will go back to what it just finished, the second expression of the query, and try to find a different match.

7. The old match for the second query expression was from the first fact, so now it examines the second, third, and fourth facts in order. Unfortunately, none have **dick** as their first argument, so the expression fails.

8. So the interpreter must return to the first predicate expression yet again. The first and second facts have been tried, so it uses the third fact and sets **X=ann** and **Y=mary**.

9. It resumes normal left-to-right processing and tries to find a match for the second query expression, starting at the top of the list of facts. (Each time it approaches a query predicate expression from the left, it starts at the top of the facts.) This means finding a fact where **mary** is the first argument, and indeed the fourth fact qualifies.

10. So **Z=harry**, and the entire query succeeds when **X=ann**, meaning that Ann is a superboss. The interpreter types out **X=ann, Y=mary, Z=harry**.

Backtracking with "not"s

Negated predicate expressions (expressions with a **not**) are easy with backtracking. Since they can't bind variables to succeed, they can be skipped in backtracking. For instance, we could add another expression to our superboss query to insist that the superboss not be the boss of Dick:

```
?- boss(X,Y), not(boss(X,dick)), boss(Y,Z).
```

Then when the interpreter executes the new query (see Figure 3-5):

1. The first predicate expression matches the first fact in the database as before, setting **X=dick** and **Y=harry**.

2. This binding of **X** satisfies the second condition, the **not** (Dick isn't his own boss).

3. For the third expression, there's no fact with **harry** as its first argument, so it fails. The interpreter backtracks to the immediately previous (second) expression.

4. But the second expression is a **not**, and **not**s always fail on backtracking, so the interpreter returns to the first expression and matches **X=tom** and **Y=dick**.

5. But this **X** now makes the second expression fail--there is a fact that Tom is the boss of Dick.

6. The interpreter returns to the first predicate expression and takes the next matching from the database, **X=ann** and **Y=mary**.

7. This **X** succeeds with the second expression since Ann isn't the boss of Dick.

8. The **Y** match succeeds with the third expression if **Z=harry**. So the interpreter reports that **X=ann, Y=mary, Z=harry**.

Notice the backtracking works differently if we rearrange the query into this equivalent form:

```
?- boss(X,Y), boss(Y,Z), not(boss(X,dick)).
```

This is processed very much like the original two-expression superboss query, except that the **not** expression forces failure instead of success for **X=tom**; we got the same effect by typing a semicolon after the first answer to the query in the last section. But the query won't work right if we reorder it as

```
?- not(boss(X,dick)), boss(X,Y), boss(Y,Z).
```

because now a **not** with an unbound variable is first, violating the guideline of Section 3.6.

The generate-and-test scheme

When a variable occurs more than once in a Prolog query, the Prolog interpreter chooses a value (binding) at the first occurrence, and then uses it in all other occurrences. So processing of the predicate expression containing the first occurrence "generates" a value, which is "tested" by the predicate expressions containing later occurrences. This idea is often used in artificial intelligence, and it's called the *generate-and-test* scheme or paradigm. Often the generating predicate expressions define the types of the variables, so their predicates are type predicates (Section 2.2).

Generate-and-test is a good way to attack problems for which we don't know any particularly good way to proceed. We generate potential solutions, and apply a series of tests to check for a true solution. This works well when it's hard to reason backward about a problem (from a statement of the problem to a solution), but it's easy to reason forward from a guess to a solution (or verify a proposed solution). An example is cryptography (decoding ciphers): an approach is to guess possible coding (encryption) methods, and see if any of them gives coded text resembling a coded message. Many other interesting problems work well for generate-and-test. But problems with well-defined solution methods, like many mathematical problems, aren't suitable for it.

Backtracking with "or"s (*)

Semicolons in queries ("or"s) are tricky for backtracking. We'll mostly ignore them in this book because, as we say, there's a better way to get their effect; but for the record, here's what happens. When a predicate expression before a semicolon succeeds, all the other expressions to the right that are "or"ed with it can be skipped. When such an expression fails, the next term to the right should be tried. If there aren't any more, the whole "or" should fail, which usually means backtracking to the left. So while backtracking with "and"s always goes left, backtracking with "or"s sometimes goes left and sometimes goes right.

Implementation of backtracking

Implementing backtracking requires allocation of a pointer (Appendix C defines pointers) for every predicate expression in a query, a pointer to where in the database the interpreter last found a match for a predicate expression. So Prolog is more complicated to implement than conventional higher-level languages like Pascal that only need extra storage in the form of a stack for procedure calls (Appendix C defines stacks too). Prolog needs a stack for this purpose too, as you'll see in Chapter 4.

Queries don't necessarily require inspection of facts in the database in sequence (*sequential search*). All Prolog interpreters *index* facts in some way, usually at least by predicate name. This means keeping lists of facts with the same predicate name, together with their addresses. So when the interpreter sees an **a_kind_of** predicate in a query, it need only search through the **a_kind_of** facts pointed to in the **a_kind_of** index list for a match. Figure 3-6 gives an example of a database and its index. More selective indexing (not standard in most Prolog dialects, though) can examine the arguments too.

Indexing by predicate name means that Prolog facts can go in many different orders and still provide exactly the same behavior. For instance, the facts in Figure 3-6

```
a_kind_of(enterprise,ship).
a_kind_of(kennedy,ship).
part_of(enterprise,u_s_navy).
part_of(kennedy,u_s_navy).
part_of(u_s_navy,u_s_government).
```

can be rearranged as

```
part_of(enterprise,u_s_navy).
a_kind_of(enterprise,ship).
part_of(kennedy,u_s_navy).
a_kind_of(kennedy,ship).
part_of(u_s_navy,u_s_government).
```

and all queries will give the same answers in the same order, as they will for

```
part_of(enterprise,u_s_navy).
part_of(kennedy,u_s_navy).
part_of(u_s_navy,u_s_government).
a_kind_of(enterprise,ship).
a_kind_of(kennedy,ship).
```

But this is only because the three **part_of** facts maintain their order and the two **a_kind_of** facts maintain their order. The following database will give answers in a different order than the preceding, though it gives the same answers:

```
a_kind_of(kennedy,ship).
part_of(u_s_navy,u_s_government).
part_of(enterprise,u_s_navy).
a_kind_of(enterprise,ship).
part_of(kennedy,u_s_navy).
```

The speed of query answering depends on how many facts are indexed for each predicate in a query; the more facts, the slower the queries. Queries will also be slower when variables appear multiple times in the query and there is no argument indexing. This situation, called a *join* in database systems, requires embedded iterative loops, and loops can take a lot of time. With joins, possibilities literally multiply. For our previous example

```
?- a_kind_of(enterprise,X), color(X,C).
```

if there are 100 **a_kind_of** facts and 50 **color** facts, 50,000 combinations must be tried to find all possible **X** and **C** pairs, as when we type a semicolon repeatedly or when there are no such **X** and **C**.

About long examples

We've studied several long examples in this chapter. Are all the examples of artificial intelligence like this? Yes, unfortunately. Artificial intelligence is a set of techniques for managing complexity, and you can only see its advantages in at least moderately complex problems.

This disturbs some students. They feel that since they can't get all of a long example into their heads at once, they can't really understand what's going on. One reply is to think of programming languages. There's a lot of activity behind the scenes that the programmer isn't usually aware of--parsing, storage management, symbol tables, stacks, type checking, register allocation, and optimization. But you don't need to know this to program. The complexity of artificial-intelligence examples comes from the need to explain, at least initially, similar behind-the-scenes details. Once you understand what details are necessary, you can ignore them as you program.

Most artificial-intelligence programs and systems do provide additional help for understanding complex program activities in the form of *explanation* facilities that summarize and answer questions about reasoning activity. These facilities provide program tracing, and answer questions like "Why did you conclude boss(tom,dick)?" and "Why didn't you conclude boss(dick,tom)?". More on this in Section 15.8.

Keywords:

inference
Prolog database
Prolog interpreter
query
success
failure
variable
binding variables
first-order logic
inputs
outputs
match
matching alternatives
database order
conjunction
match
disjunction
negation
not
built-in predicates
closed-world assumption
consult
backtracking
failure
indexing
generate-and-test

Exercises

3-1. (A) Good programming style in Prolog doesn't allow the argument to a not to be more than one predicate expression, and doesn't allow composite queries (queries not just a single predicate expression) "or"d together.

***(a) Using good programming style, write a Prolog query that is true if the "nor" of predicate expressions s a* and b (both of no arguments) is true. ("Nor" means the opposite of an "or").**

(b) Using good programming style, write a Prolog query that is true if the "exclusive or" of predicate expressions a and b is true. ("Exclusive or" means either is true but not both).

3-2. (R,A) Suppose you have the following Prolog database:

```
incumbent(csprofessor,davis).
incumbent(csprofessor,rowe).
incumbent(csprofessor,wu).
incumbent(csprofessor,zyda).
incumbent(cschairman,lum).
incumbent(dean_ips,marshall).
incumbent(provost,shrady).
incumbent(superintendent,shumaker).
incumbent(director_milops,bley).
bossed_by(csprofessor,cschairman).
bossed_by(cschairman,dean_ips).
bossed_by(orchairman,dean_ips).
```

```
bossed_by(dean_ips,provost).
bossed_by(provost,superintendent).
bossed_by(director_milops,superintendent).
```

(a) The incumbent predicate means that the person that is its second argument has the job description that is its first argument; the bossed_by predicate means that the boss of the first argument is the second argument. Paraphrase each of the following Prolog queries in English.

```
?- bossed_by(csprofessor,X), bossed_by(X,Y).
?- bossed_by(X,Y), incumbent(X,rowe), incumbent(Y,Z).
?- incumbent(dean_ip,X); incumbent(dean_ips,X).
?- incumbent(J,P), (bossed_by(J,provost); bossed_by(J,dean_ips)).
?- bossed_by(P,superintendent), not(incumbent(P,shrady)).
```

(b) Without using a computer, what will be the first answer found by a Prolog interpreter with the preceding database and with each query given?

3-3. Suppose two queries each represent an "and" of a number of predicate expressions. Suppose the expressions of query 1 are a subset of the expressions in query 2. How do the answers to query 1 relate to the answers to query 2?

3-4. The words "the" and "a" mean different things in English. What important feature of Prolog querying does the difference between them demonstrate in the following sentences?

> **Find a memo we sent headquarters last week. The memo reported on a board meeting last October 10. The board meeting was noisy, and this is mentioned in the memo.**

3-5. (A) Suppose in your Prolog database you have N one-argument facts for the predicate name p and M one-argument facts for the predicate name q.

(a) What is the maximum number of answers, not counting no, that you will get to the query

```
?- p(X), q(Y).
```

(b) How many total times will the Prolog interpreter backtrack from q to p for the situation in part (a), before it types no?

(c) What is the minimum number of answers to the query in part (a)?

(d) What is the maximum number of answers, not counting no, you will get to the query

```
?- p(X), q(X).
```

(e) How many total times will the Prolog interpreter backtrack from q to p for the situation in part (d), before it types no?

(f) What is the minimum number of answers to the query in part (d)?

3-6. (R,A) Suppose we keep in a Prolog database information about grades on two tests in a course.

(a) Suppose we ask if Joe got an A on test 1 and the Prolog interpreter says yes. Suppose we then ask if

Joe got an A on test 2 and it says yes. It seems fair to summarize this by saying Joe got A's on both tests 1 and 2. Now suppose we ask if someone got an A on test 1 and the interpreter says yes. We ask if someone got an A on test 2 and it says yes. It is unfair now to conclude that someone got an A on both test 1 and test 2. How is this situation different? How does this illustrate an important feature of Prolog querying?

(b) Suppose the database consists of facts of the form:

`grade(<person>,<test-number>,<grade>).`

Write a query that establishes if everyone in the class got an A on test 1, without using an "or" (semicolon). (Hint: Use the exact opposite.)

(c) Suppose you ask if everyone in the class got an A on test 1 and the Prolog interpreter says yes. Suppose you then ask if everyone in the class got an A on test 2 and it says yes. Can you conclude that everyone in the class got both an A on test 1 and an A on test 2? Why? Assume this is a real class at a real college or university.

3-7. Here's a summary of the current situation on the fictitious television soap opera *Edge of Boredom:*

> Jason and Phoebe are married, but Phoebe is in love with Perry. Perry doesn't love her because he is still married to Stacey, but Zack is romantically inclined toward Phoebe. He's in competition with Lane, who also loves Phoebe despite being married to Eulalie, whom Jason is feeling romantic about.

(a) Represent the basic meaning of these statements by facts using only two different predicate names. Notice that if X is married to Y, Y is married to X.

(b) A marriage is on the rocks if both its participants are in love with other people and not with each other. Which people are in marriages that are on the rocks? Show the necessary Prolog query and its result.

(c) A person is jealous when a person they love is loved by a third person, or a person is jealous when married to someone loved by a third person. Which people are jealous? Show the necessary Prolog query and its result.

3-8.(a) Consider the query

```
?- a(X,Y), b(X,Y).
```

with the database

```
a(1,1).
a(2,1).
a(3,2).
a(4,4).
b(1,2).
b(1,3).
b(2,3).
b(3,2).
b(4,4).
```

Without using a computer, what are all the answers that you will get to the query, in order (as you keep typing semicolons)?

(b) Without using a computer, what does this query print out (as you keep typing semicolons)?

```
?- a(X,Y), b(X,Y), a(Y,Y).
```

3-9. (A) Consider this Prolog query:

```
?- r(X,Y), s(Y,Z), not(r(Y,X)), not(s(Y,Y)).
```

with this database:

```
r(a,b).
r(a,c).
r(b,a).
r(a,d).
s(b,c).
s(b,d).
s(c,d).
s(c,c).
s(d,e).
```

(a) Without using a computer, what is the first answer found to the query? Hint: you don't have to do it Prolog's way.

(b) Without using a computer, how many times does a Prolog interpreter backtrack from the third to the second predicate expression to get this first answer?

3-10. Consider this Prolog database:

```
u(a,b).
u(b,b).
u(c,d).
u(c,a).
u(d,a).
u(d,c).
```

Now consider this Prolog query, without actually using a computer:

```
?- u(X,Y), u(Y,Z), not(u(X,Z)).
```

(a) How many times will a Prolog interpreter backtrack to the first query predicate expression **u(X,Y)** to find the first answer to this query?

(b) How many times will a Prolog interpreter backtrack to the second query predicate expression **u(Y,Z)** to find the first answer to this query?

(c) How many times will a Prolog interpreter backtrack to the third query predicate expression **not(u(X,Z))** to find the first answer to this query?

(d) How many further times will a Prolog interpreter backtrack to the first query predicate expression **u(X,Y)** to find the second answer to this query?

(e) How many further times will a Prolog interpreter backtrack to the second query predicate expression **u(Y,Z)** to find the second answer to this query?

3-11. (H) Design a good set of predicates for the following data about an organization and its employees. Assume you have to do this in Prolog. Try to be efficient: avoid duplicate data, empty data, and too many linking arguments, while keeping data access reasonably fast.

Assume we have an organization with departments, subdepartments, and projects. A subdepartment can belong to only one department, but a project can belong to more than one subdepartment or department (but most of the time only one). Employees belong to one subdepartment and one or more projects. Employees have a name, social security number, date of birth, address, and a list of successfully completed projects that they participated in since they joined the organization. Employees also are characterized by Personnel by "job skills" they have from a rough list (e.g. "can type", "has truck license", "experience in writing"). Projects have a name, code, starting date, projected or actual completion date, and the room number for the office of each employee on the project. Employees have only one office, but there may be more than one employee in the same office.

Design these predicates to answer these questions easily:

--Give the name, department, and office number for each employee on project 93521.
--Give the name, department, and office number for each employee on projects started last year.
--Give the people in department 43 who have typing skills.

3-12. Questions in English have subtleties that are sometimes hard to translate into Prolog queries. This became obvious in building the first *natural language front ends* to databases, computer programs that tried to answer, in English, questions about the database contents. Here are illustrations of two bugs discovered in those early programs. Try to explain what a program like a Prolog interpreter is missing when it makes such errors. (Brackets give our explanatory comments.)

(a)

Person: Can you tell me the commander of the Enterprise and his rank?
Computer: Yes. [That's all it types in response.]

(b)

Person: Who commands the Pequod?
Computer: Nobody. [That's strange, because every ship must have a commander.]
Person: Where is the Pequod currently?
Computer: Nowhere. [Strange ship this Pequod.]
Person: Does the Pequod exist?
Computer: No. [So that's the reason.]

Go to book index