Implementation of rule-based systems

Let's examine how different control structures for rule-based systems can be implemented in Prolog. We'll cover backward chaining, rule-cycle hybrid chaining, forward chaining, input and output routines, meta-rules, and decision lattices. This will mean a lot of details; the last section of this chapter brings together the key Prolog code.

Implementing backward chaining

Though Prolog is designed for backward chaining, there are many details for an implementer of a backward-chaining rule-based system to worry about when using Prolog, especially when the rule-based system is large. The traffic lights program in Section 4.11 was simple because traffic laws are *supposed* to be simple, so even people unenlightened enough never to have taken an artificial intelligence course can understand them. Many rule-based expert systems can't be so nice, like those that diagnose and fix hard problems. Such rule-based systems are often called *expert systems*, because they automate the role of human experts. We'll introduce some general-purpose programming aids for expert systems in this chapter.

The goal of most expert systems is to reach a diagnosis, which we'll assume is obtained by typing the query

```
?- diagnosis(X).
```

So what is X? It should describe a situation. We could connect words with underscores as before, but there's an alternative: we can put single quotation marks (apostrophes) around a string of words to force treatment of it as a unit. That is, a *character string*. An advantage of character strings is that they can start with capital letters and contain periods and commas, while words can't.

Here are some example diagnosis rules for an expert system:

```
diagnosis('fuse blown') :- doesnt_work, all_lights_out.
diagnosis('fuse blown') :- noise(pop).
diagnosis('break in cord') :- doesnt_work, cord_frayed.
```

Of course we must define those right-side predicates.

Implementing virtual facts in caching

One problem is those rules require advance entry of facts (often, many facts) so that rule right sides can work properly. As we mentioned in the last chapter, virtual facts (facts demanded only when needed) are a simple improvement. A good way to get them is to define a function predicate **ask** of two bound arguments. The first argument is an input, a string containing question text to be typed out on the terminal, and the second argument is an output, a variable to be bound to the question's answer that the user types.

```
ask(Q,A) :- write(Q), write('?'), read(A), nl.
```

Here **write**, **read**, and **nl** are Prolog predicates built-in in most implementations (see Appendix D); **write** prints its argument on the terminal, **read** reads something typed by the user and binds that something to the variable that is the **read**'s argument, and **nl** sends a carriage return to the terminal.

Now we never want to ask a user the same question twice; we should cache answers so we can reuse them. It's easy to add this feature to the **ask** predicate. We just use the **asserta** built-in predicate introduced in Section 6.1, which takes a fact as argument and adds it to the Prolog database. Using it, conclusions can be added to the database as they are discovered. We can stick the **asserta** at the end of the definition of the **ask** predicate:

```
ask(Q,A) :- write(Q), write('?'), read(A), nl, asserta(ask(Q,A)).
```

Then if the same question is asked again, the fact will be used to answer it instead of this rule. This works because facts put in the database with **asserta** are put in front of all other facts and rules with the same first predicate name. Here's an example of the use of **ask**:

```
diagnosis('fuse blown') :- ask('Does the device work at all',no),
  ask(Are the lights in the house off',yes).
```

This says to diagnose that the fuse is blown if (1) the user answers **no** when asked whether the device works at all, and (2) the user answers **yes** when asked whether all the lights in the house are off.

You should carefully phrase the questions to be issued by a rule-based system. In particular, avoid pronouns and other indirect references to things, since rules and questions may be invoked in hard-to-predict orders. Generally, though there are exceptions, phrase questions so a **yes** answer means unusual things are going on, while a **no** means things are normal. For instance, after "Are the lights in the house off?", don't ask "Is the fuse OK?" but "Is the fuse blown?". And be consistent in phrasing questions. After that question "Are the lights in the house off?", ask "Does the fuse look blown?" in preference to than "The fuse looks blown doesn't it?", to maintain the same verb-noun-adjective order.

Input coding

We could implement a big expert system this way, with diagnosis rules having **ask** predicates on their right sides. But this can require unnecessary code repetition. So two important coding tricks are used in large rule-based systems: coding of input (answers) and coding of output (questions and diagnoses).

Input coding groups user answers into categories. An important case is questions with only **yes** or **no** answers; expert systems often to rely on them for simplicity. We can define two new predicates **affirmative** and **negative**, which say whether a word the user typed is a positive or a negative answer respectively:

```
affirmative(yes).
affirmative(y).
affirmative(ye).
affirmative(right).
affirmative(ok).
affirmative(uhhuh).
negative(no).
negative(n).
negative(not).
negative(never).
negative(impossible).
negative(haha).
```

Then we can define a predicate **askif** of one input argument. It will be just like **ask** except it will have only one argument, the question, and it will succeed if that question is answered affirmatively and fail if the question is answered negatively. We can also fix it so that if an answer is neither positive nor negative (in other words, it is unclear), we will complain and ask for another answer.

```
askif(Q) :- ask(Q,A), positive_answer(A).
positive_answer(A) :- affirmative(A).
positive_answer(Qcode,A) :- not(negative(A)), not(affirmative(A)),
  write('Please answer yes or no.'), read(A2),
  retract(asked(Qcode,A)), asserta(asked(Qcode,A2)),
  affirmative(A2).
```

We can also define:

```
askifnot(Q) :- not(askif(Q)).
```

which saves some parentheses.

Users may not always understand a question. We can let them type a **?** instead of an answer, give them some explanatory text, and provide them another chance to answer:

```
ask(Q,A) :- asked(Q,A).
ask(Q,A) :- not(asked(Q,A)), write(Q), write('? '), read(A2),
  ask2(Q,A2,A).
ask2(Q,'?',A) :- explain(Q), ask(Q,A).
ask2(Q,A,A) :- not(A='?'), nl, asserta(asked(Q,A)).
```

where **explain** facts store explanatory text. Minor humanizing touches such as these can be immensely important to user satisfaction, while imposing little on the programmer.

Output coding

Another useful trick is to code questions, so we need not repeat their text at each mention in the rules. Codes for questions also make rules easier to read, and help prevent mistakes because it's easy to err in typing a long string of text (and with caching, every slightly different question is asked and cached separately). For this we can use a predicate **questioncode** of two arguments, a code word and a text string for the corresponding question. Here's an example from appliance diagnosis:

```
diagnosis('fuse blown') :- askif(device_dead), askif(lights_out).
diagnosis('fuse blown') :- askif(hear_pop).
diagnosis('break in cord') :- askif(device_dead),
  askif(cord_frayed).
questioncode(device_dead,'Does the device refuse to do anything').
questioncode(lights_out,
  'Do all the lights in the house seem to be off').
questioncode(hear_pop,'Did you hear a sound like a pop').
questioncode(cord_frayed,
  'Does the outer covering of the cord appear to be coming apart').
```

To handle this, we must redefine **ask**:

```
ask(Qcode,A) :- asked(Qcode,A).
ask(Qcode,A) :- not(asked(Qcode,A)), questioncode(Qcode,Q),
  write(Q), write('? '), read(A2), ask2(Q,Qcode,A2,A).
ask2(Q,Qcode,'?',A) :- explain(Qcode), ask(Qcode,A).
ask2(Q,Qcode,A,A) :- not(A='?'), asserta(asked(Qcode,A)).
```

A further refinement is to handle a class of related questions together. We can do this by giving arguments to output codes, as for instance using **hear(X)** to represent a question about hearing a sound **X**. Then to make the query we need string concatenation, something unfortunately not available in most Prolog dialects. But there is a simple shortcut to concatenation by writing a **questioncode** rule instead of a fact, that types extra words before succeeding:

```
questioncode(hear(X),X) :- write('Did you hear a sound like a ').
```

So to ask if the user heard a pop sound you use:

```
askif(hear(pop)).
```

which prints on the terminal as:

```
Did you hear a sound like a pop ?
```

Yet another coding trick is to code diagnoses. This isn't as useful as question coding, but helps when diagnoses are provable in many different ways. Diagnosis coding requires a new top-level predicate that users must query instead of **diagnosis**, as:

```
coded_diagnosis(D) :- diagnosis(X), diagnosis_code(X,D).
```

For instance, we could use:

```
diagnosis(fuse) :- ask('Does the device work at all',no),
  ask(Are the lights in the house off',yes).
diagnosis_code(fuse,'Fuse blown').
```

Then we could get this behavior:

```
?- coded_diagnosis(X).
Does the device work at all? no.
Are the lights in the house off? yes.
X=Fuse blown
```

Intermediate predicates

Building expert systems is straightforward when there are ten to a hundred rules, each with one to three expressions on the right side. But it can get confusing when, as in a typical expert system today, there are thousands of rules averaging ten expressions per right side. It just gets too difficult to keep track of all the symbols used, and to determine everything necessary for each rule. The solution is to frequently use intermediate predicates, predicates that occur on both the left and right sides of rules. Intermediate predicates can represent important simplifying generalizations about groups of facts. Viewing expert-system predicates as a hierarchy with diagnoses or final conclusions at the top and asked questions at the bottom, intermediate predicates are everything in between. Much of the intelligence and sophistication of expert systems can come from a good choice of intermediate predicates to reduce redundancy and simplify rules.

A useful intermediate predicate with appliance diagnosis is **power_problem**, a predicate that is true if the appliance is not getting any electricity for its innards. It's useful because appropriate diagnoses when it is true are quite different from those when it is false: a non-power problem must be in the device itself and can't be in the cord or external fuse. So **power_problem** can go into many rules, typically early in the right side of the rule to filter out inappropriate rule use. But **power_problem** clearly is an intermediate predicate, not a fact we can establish from a single question, because it has many different symptoms. And some symptoms are very strong, as when all the lights in the house went off when you have tried to turn on the device, or when the device stopped working when you moved its frayed cord slightly.

Generally speaking, intermediate predicates are needed for any important phenomena that aren't diagnoses. Here are some more ideas for intermediate predicates in appliance diagnosis:

--whether the problem is mechanical;
--whether the problem is in a heating element;
--whether the appliance has had similar problems before;
--whether you can improve things by adjusting the controls or buttons;
--whether danger of electrocution is present;
--whether anything unusual was done to the appliance lately (like being dropped or having liquids spilled on it);
--how much troubleshooting expertise the user has (note that intermediate predicates can have arguments).

Intermediate-predicate expressions won't be arguments to **askif**s, since they don't directly query a user. Caching is a good idea with intermediate predicates, even more so than caching of query answers (as we did with the **ask** predicate). A single intermediate-predicate fact can summarize many questions, and caching it saves having to ask all those questions over again. Nevertheless, caching of intermediate-predicate conclusions should not necessarily be automatic, as it only makes sense when a result might be reused.

An example program

Let's put together the ideas we've introduced in this chapter in a larger rule-based system for the diagnosis of malfunctions in small household appliances. Figure 7-1 shows some of the terminology, and Figure 7-2 gives the predicate hierarchy. We list rules in three groups: diagnosis (top-level) rules, intermediate predicate rules, and question-decoding rules. To get a diagnosis from this program, query **diagnosis(X)**. Typing a semicolon will then give you an alternative diagnosis, if any; and so on. So if several things are wrong with the appliance, the program will eventually find them all.

```
/* Top-level diagnosis rules */
diagnosis('fuse blown') :- power_problem, askif(lights_out).
diagnosis('fuse blown') :- power_problem, askif(hear(pop)).
diagnosis('break in cord') :- power_problem, askif(cord_frayed).
diagnosis('short in cord') :- diagnosis('fuse blown'),
  askif(cord_frayed).
diagnosis('device not turned on') :- power_problem,
  klutz_user, askif(has('an on-off switch or control')),
  askif(device_on).
diagnosis('cord not in socket properly') :- power_problem,
  klutz_user, askif(just_plugged), askif(in_socket).
diagnosis('foreign matter caught on heating element') :-
  heating_element, not(power_problem), askif(smell_smoke).
diagnosis('appliance wet--dry it out and try again') :-
  power_problem, klutz_user, askif(liquids).
diagnosis('controls adjusted improperly') :- klutz_user,
  askif(has('knobs, switches, or other controls')).
diagnosis('kick it, then try it again') :- mechanical_problem.
diagnosis('throw it out and get a new one') :-
  not(power_problem), askif(hear('weird noise')).
diagnosis('throw it out and get a new one').

/* Definitions of intermediate predicates */
power_problem :- askif(device_dead), askif(has(knobs)),
askif(knobs_do_something).
power_problem :- askif(device_dead), askif(smell_smoke).
klutz_user :- askifnot(handyperson).
klutz_user :- askifnot(familiar_appliance).
mechanical_problem :- askif(hear('weird noise')),
  askif(has('moving parts')).
heating_element :- askif(heats).
heating_element :- askif(powerful).

/* Question decoding */
questioncode(device_dead,'Does the device refuse to do anything').
questioncode(knobs_do_something,
  'Does changing the switch positions or turning
  the knobs change anything').
```

```
questioncode(lights_out,
  'Do all the lights in the house seem to be off').
questioncode(cord_frayed,
  'Does the outer covering of the cord appear to be coming apart').
questioncode(handyperson,'Are you good at fixing things').
questioncode(familiar_appliance,
  'Are you familiar with how this appliance works').
questioncode(device_on,'Is the ON/OFF switch set to ON').
questioncode(just_plugged,'Did you just plug the appliance in').
questioncode(in_socket,'Is the cord firmly plugged into the socket').
questioncode(smell_smoke,'Do you smell smoke').
questioncode(liquids,
  'Have any liquids spilled on the appliance just now').
questioncode(heats,'Does the appliance heat things').
questioncode(powerful,'Does the appliance require a lot of power').
questioncode(has(X),X) :- write('Does the appliance have').
questioncode(hear(X),X) :- write('Did you hear a ').
```

Here we use variables inside question codes for questions about components and sounds heard. We also use a *subdiagnosis* ('fuse blown') as input to another diagnosis, a useful trick.

Running the example program

Here's an actual run of this program (which requires definitions of **askif** and other predicates given earlier in this chapter). Note the same diagnosis is repeated when there are different ways to prove it.

```
?- diagnosis(X).
Does the device refuse to do anything? yes.
Do all the lights in the house seem to be off? no.
Does the appliance have knobs or switches? yes.
Does changing the switch positions or turning
the knobs change anything? no.
Do you smell smoke? yes.
Does the appliance heat things? no.
Does the appliance require a lot of power? no.
Did you hear a pop? yes.

X=Fuse blown;

X=Fuse blown;

X=Fuse blown;
Are you good at fixing things? no.
Does the appliance have an on-off switch or control? yes.
Is the ON/OFF switch set to ON? no.

X=Device not turned on;

Are you familiar with how this appliance works? no.

X=Device not turned on;

X=Device not turned on;

X=Device not turned on;

X=Device not turned on;

X=Device not turned on;

Did you just plug the appliance in? yes.
Is the cord firmly plugged into the socket? no.

X=Cord not in socket properly;

X=Cord not in socket properly;

X=Cord not in socket properly;

X=Cord not in socket properly;

X=Cord not in socket properly;

X=Cord not in socket properly;

Have any liquids spilled on the appliance just now? maybe.
Please type yes or no. no.

X=Controls adjusted improperly;

X=Controls adjusted improperly;
Did you hear a weird noise? no.

X=Throw it out and get a new one;

no
?- halt.
```

Partitioned rule-based systems

Intermediate predicates group related rules together, but they are only a conceptual grouping, more a help to understanding and debugging programs. A stronger way of grouping is putting rules into partitions that can't "see" one another. This is easy to do with Prolog by putting rules in separate files and only loading the files you need into the database. Loading is done with the **consult** built-in predicate in Prolog, a predicate of one argument which is the name of the file to load. So if the rule

```
a :- b, c.
```

is used for backward chaining, and we want whenever it succeeds for the file "more" to be loaded, we should rewrite it as

```
a :- b, c, consult(more).
```

Like most built-in predicates, **consult** always fails on backtracking since there's only one way to load a file.

Often one partition is designated the "starting" partition, loaded automatically when the rule-based system begins. It then decides which other partitions to load and invoke. If a loaded partition later decides it's not relevant (as when none of its rules fire), it can itself load another partition and start that one running.

Implementing the rule-cycle hybrid

Prolog's features make backward chaining easy. But it's also a general-purpose programming language, and can implement quite different control structures.

First consider the rule-cycle hybrid of backward and forward chaining, easier to implement than pure forward chaining and hence used in many simple expert systems. It can be done by writing each rule in a new form, a transformation of each backward chaining rule:

1. "and" a new **asserta** on the right end of the right side of the rule, whose argument is the left side of the rule;
2. "and" a new **not** on the left end of the right side of the rule, with the same argument;
3. then replace the left side of the rule by **r** (first renaming any **r** predicates already in the rules).

So these backward chaining rules:

```
a :- b.
c :- d, e, f.
```

become:

```
r :- not(a), b, asserta(a).
r :- not(c), d, e, f, asserta(c).
```

And two sample rules from the appliance diagnosis program:

```
diagnosis('fuse blown') :- power_problem, askif(lights_out).
power_problem :- askif(device_dead), askif(has(knobs)),
  askif(knobs_do_something).
```

become:

```
r :- not(diagnosis('fuse blown')), power_problem, askif(lights_out),
  asserta(diagnosis('fuse blown')).
r :- not(power_problem), askif(device_dead), askif(has(knobs)),
  askif(knobs_do_something), asserta(power_problem).
```

(We'll discuss later how to convert automatically.) So we replace our old rules with new rules whose only effect is caching of particular conclusions. Note that these new rules never call on other rules, even if there are intermediate predicates, because **r** is the only left side (since we made sure the predicate **r** doesn't occur in the rule-based system). We'll assume for now that no predicate expressions in the original rules contain **not**s, since they introduce complications. If we really need negatives, we can define fact predicates that stand for the opposite of other fact predicates.

Now we must cycle through the rules; that is, consider each rule in order, and go back to the first when we finish the last. Within each pass, we can force the Prolog interpreter to repeatedly backtrack to a query of **r**. A simple way is

```
?- r, 1=2.
```

Since the **=** can never be true, the interpreter will keep trying **r** rules, regardless of whether they succeed or fail. Eventually it will run out of all **r** rules and fail. Actually, there's a built-in Prolog predicate called **fail** that has exactly the same effect as **1=2**, so we can say equivalently

```
?- r, fail.
```

To give us a handle on this code, let's give it a name:

```
one_cycle :- r, fail.
```

To get the Prolog interpreter to repeat indefinitely a cycle through the rules, we might think that we could do the same **fail** trick, like

```
hybrid :- one_cycle, fail.
```

But this won't work because **one_cycle** won't ever return to the *top* of the list of rules. And **one_cycle** itself never succeeds, so the **fail** is useless. We could try

```
hybrid :- not(one_cycle), fail.
```

which answers the second objection but not the first: we need each call to **one_cycle** to be a fresh call. That suggests recursion:

```
hybrid :- done.
hybrid :- not(one_cycle), hybrid.
```

The **done** is a stopping condition that must be defined by the builder of the rule-based system. For diagnosis expert systems, it could be defined as

```
done :- diagnosis(X).
```

which says to stop whenever some diagnosis is proved.

The preceding definition of **hybrid** only checks once per cycle whether it is done. To stop sooner, we could put the check inside **one_cycle** like this:

```
hybrid :- done.
hybrid :- not(one_cycle), hybrid.
one_cycle :- r, done.
```

But this requires more calls to **done**, not a good idea if **done** is a complicated calculation.

Note: this approach can handle **not**s in rules, though differently from the algorithm in Section 6.4 since **not**s will be evaluated on every cycle. But as with the algorithm, any **not** must occur before any rule with the argument to the **not** as its left side, or we'll get wrong answers.

Implementing pure forward chaining (*)

Pure forward chaining requires yet another rule form. (See a Section 7.14 for how to rewrite rules automatically in this form.) Since pure forward chaining repeatedly finds and "crosses out" expressions on the right sides of rules, it would help to express rule right sides as lists, for then we can use our **member** and **delete** list-processing predicates from Chapter 5. We can do this by making rules a kind of fact, say using a **rule** predicate name. The first argument to **rule** can be the left side of the original rule, and the second argument the list of predicate expressions "and"ed on the right side. So these rules

```
a :- b.
c :- d, e, f.
g(X) :- h(X,Y), not(f(Y)).
```

become

```
rule(a,[b]).
rule(c,[d,e,f]).
rule(g(X),[h(X,Y),not(f(Y))]).
```

and the two sample rules from the appliance diagnosis program

```
diagnosis('fuse blown') :- power_problem, askif(lights_out).
power_problem :- askif(device_dead), askif(has(knobs)),
  askif(knobs_do_something).
```

become

```
rule(diagnosis('fuse blown'),[power_problem,askif(lights_out)]).
rule(power_problem,
  [askif(device_dead),askif(has(knobs)),askif(knobs_do_something)]).
```

For now, we'll assume that the rules don't contain **not**s.

We also must represent facts. Pure forward chaining requires that we identify all facts, distinguishing them from rules. We can do this by making each fact an argument to a predicate named **fact**, of one argument. Then to bind **F** to every fact in turn, we query

```
?- fact(F), done.
```

which will backtrack repeatedly into **fact**. For every fact **F**, we must find the rules whose right sides can match it, to derive new rules and possibly new facts. This suggests:

```
forward :- fact(F), not(pursuit(F)), done.
```

Unfortunately, we can't really implement "focus-of-attention" forward chaining this way, since we can't insert new facts just after the last fact we selected, only at the beginning (with **asserta**) and end (with **assertz**) of the database. To prevent fact reuse, we can delete facts once pursued. But deleted facts are still important to us (obtaining facts is the whole point of forward chaining) so we'll copy them into **usedfact** facts before we delete them. The revised code:

```
forward :- done.
forward :- fact(F), not(pursuit(F)), assertz(usedfact(F)),
  retract(fact(F)), forward.
```

(Remember from Section 6.1 that **retract** removes a fact from the database.) Then when we're done, all the things we learned plus all the starting facts are in the database as arguments to the **fact** and **usedfact** predicates.

The **pursuit** predicate can cycle through the rules like **one_cycle** did in the hybrid implementation:

```
pursuit(F) :- rule(L,R), rule_pursuit(F,L,R), fail.
```

For **rule_pursuit** we must search through the right side of a rule, deleting anything that matches the fact **F**; we can use the **member** and **delete** predicates of Sections 5.5 and 5.6 respectively. As you may dimly recall, **member** checks whether an item is a member of a list, and **delete** removes all occurrences of an item from a list. We need them both because **delete** always succeeds, and we'd like to fail when a match doesn't occur in the list. So (Figure 7-3):

```
forward :- done.
forward :- fact(F), not(pursuit(F)), assertz(usedfact(F)),
  retract(fact(F)), forward.
pursuit(F) :- rule(L,R), rule_pursuit(F,L,R), fail.
rule_pursuit(F,L,R) :- member(F,R), delete(F,R,Rnew),
  new_rule(L,Rnew).
new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).
```

The two **new_rule** lines say that when you've deleted everything on the right side of a rule, the left side is a new fact; otherwise just write a new, shorter and simpler rule. And here again are **member** and **delete**:

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- not(X=Y), delete(X,L,M).
```

As with the rule-cycle hybrid, you must define **done** appropriately. If you want to make sure that all possible conclusions are reached, use

```
done :- not(fact(X)).
```

which forces forward chaining to continue until there are no more **fact** facts, in which case everything learned (as well as the initial facts) is a **usedfact**. (Note this rule violates our advice in Section 3.6 to avoid **not**s whose variables aren't bound. Here we want to stop if there's any unexamined fact **X** remaining, a kind of existential quantification, so it makes sense. A universally quantified negation is equivalent to an existentially quantified unnegation; see Appendix A.) Alternatively, **done** can be defined to mean that one of a set of "goal" facts have been proved.

The **asserta** in the first **new_rule** rule is important because it forces focus-of-attention handling of new facts. That is, the last fact found will be the next fact pursued, like a stack data structure, because of the recursion. As we pointed out in the last chapter, the focus-of-attention approach often reaches interesting conclusions fast. If we change **asserta** to **assertz**, we get a queue instead of a stack, with new facts pursued only after old facts.

Amazingly, the program works just fine when rules contain variables. This is because the basis step (first line) in the **member** predicate definition can bind variables to succeed, and when it does those variables keep the same bindings through the rest of the rule. If there is more than one matching of a fact to a rule, the program will find each by backtracking to **member**. For instance, the rule

```
rule(a(X,Y),[b(X,Y),b(Y,X),b(X,X)]).
```

can match the fact **b(tom,dick)** to either the first or second predicate expression on its right side, giving two new rules:

```
rule(a(tom,dick),[b(dick,tom),b(tom,tom)]).
rule(a(dick,tom),[b(dick,tom),b(dick,dick)]).
```

When forward chaining with the preceding program is slow, a simple change can often speed things up. That is to delete the old rule when a new rule is formed. This speeds things up because the sum of the number of rules, the number of **fact**s, and the number of **oldfact**s stays constant instead of always increasing. To do this, we need only add a single expression to the **rule_pursuit** rule:

```
rule_pursuit(F,L,R) :- member(F,R), delete(F,R,Rnew),
  retract(rule(L,R)), new_rule(L,Rnew).
```

We can only do this safely for rule-based systems representable as and-or-not lattices, where there are either no variables or only variables that can take a single binding. Otherwise deletion will throw away still-possibly-useful rules, but this may not bother us if the odds are small they're still useful.

Forward chaining with "not"s (*)

As with hybrid chaining, we can avoid rules containing **not**s by substituting "unfact" predicate names representing the opposite of other predicate names. Or we can require that arguments to **not**s never be matchable to anything appearing on the left side of a rule (DeMorgan's Laws can get the rules into this form; see Appendix A). Then we rewrite the top level of the program to handle **not**s after it's done everything else:

```
full_forward :- forward, handle_nots.
handle_nots :- rule(L,R), member(not(X),R), not(usedfact(X)),
  not(fact(X)), delete(not(X),R2), new_rule(L,R2), handle_nots.
handle_nots :- forward.
```

This is not quite the algorithm in Section 6.2, but it's close.

General iteration with "forall" and "doall" (*)

The iteration method of the rule-cycle hybrid and forward chaining programs can be generalized. First, suppose we want to check whether some predicate expression **Q** succeeds for all possible variable values that satisfy some other predicate expression **P**; that is, we want to check *universal quantification* of **Q** with respect to **P**. We can do this by requiring that there be no way for **Q** to fail when **P** has succeeded previously, taking into account any bindings. We can use the built-in **call** predicate of Prolog, which queries a predicate expression given as argument:

```
forall(P,Q) :- not(somefailure(P,Q)).
somefailure(P,Q) :- call(P), not(call(Q)).
```

As an example, assume this database:

```
a(1).
a(2).
b(1).
b(2).
c(1).
c(2).
c(3).
d(1,5).
d(5,1).
```

Here are some example queries and their results:

```
?- forall(a(X),b(X)).
yes
?- forall(b(X),c(X)).
yes
?- forall(c(X),b(X)).
no
?- forall(c(X),d(X,Y)).
```

```
no
?- forall(d(X,Y),d(Y,X)).
yes
```

Similarly, we can define a predicate that repeatedly backtracks into predicate expression **P** until **P** fails:

```
doall(P) :- not(alltried(P)).
alltried(P) :- call(P), fail.
```

Assume this database:

```
a(1).
a(2).
a(3).
u(X) :- a(X), write(X).
v(X) :- u(X), Y is X*X, write(Y), nl.
```

Then here are two examples:

```
?- doall(u(X)).
123
yes
?- doall(v(X)).
1
4
9
yes
```

(Remember, **write** prints its argument on the terminal, and **nl** prints a carriage return.)

This **doall** is just what the forward chaining program accomplishes in the **pursuit** predicate. So we can rewrite the first four lines of the forward chaining program as

```
forward :- done.
forward :- fact(F), doall(pursuit(F)), assertz(usedfact(F)),
  retract(fact(F)), forward.
pursuit(F) :- rule(L,R), rule_pursuit(F,L,R).
```

instead of

```
forward :- done.
forward :- fact(F), not(pursuit(F)), assertz(usedfact(F)),
  retract(fact(F)), forward.
pursuit(F) :- rule(L,R), rule_pursuit(F,L,R), fail.
```

And in the rule-cycle hybrid

```
hybrid :- done.
hybrid :- doall(r), hybrid.
```

can be used instead of

```
hybrid :- done.
hybrid :- not(one_cycle), hybrid.
one_cycle :- r, fail.
```

And the code for handling **not**s in forward chaining given in Section 7.11 can be improved to

```
full_forward :- forward, doall(handle_not), forward.
handle_not :- rule(L,R), member(not(X),R), not(usedfact(X)),
  not(fact(X)), delete(not(X),R2), new_rule(L,R2).
```

instead of

```
full_forward :- forward, handle_nots.
handle_nots :- rule(L,R), member(not(X),R), not(usedfact(X)),
  not(fact(X)), delete(not(X),R2), new_rule(L,R2), handle_nots.
handle_nots :- forward.
```

The changes improve program readability.

Input and output of forward chaining (*)

Fact pursuit is only part of what we need for forward chaining. We must also handle input and output differently than with backward chaining. Backward chaining asked the user a question whenever an answer was relevant to some conclusion under study. This can mean that many irrelevant questions are asked before backward chaining hits on the right conclusions to try to prove. Forward chaining, on the other hand, focuses on a set of facts. The facts must get into the database somehow to start things off.

Two approaches are possible. First, give a *questionnaire*, a fixed set of questions to a user presented one at a time, and code answers into facts. Fixed questionnaires are common in the early part presented one at a time of medical diagnosis, when a doctor tries to get a broad picture of the health of a patient before moving on to specifics. Second (especially if most possible facts don't have arguments), give a *menu*, a set of questions presented simultaneously to the user, and ask which questions should be answered **yes**. For diagnosis applications, the menu can contain common symptoms. Menus are good when there are lots of possible facts, few of which are simultaneously relevant to a case.

Both are straightforward to implement in Prolog. Questionnaires can be done by a fixed sequence of calls to the **askif** predicate defined in Section 7.3. The answer to each question will cause the assertion of **asked** and **fact** facts. Menus can be implemented by an **ask_which** predicate | REFERENCE 1|: .FS | REFERENCE 1| It's interesting to compare this program to the forward-chaining program in Section 7.10: this represents more a Lisp style of programming, with recursion through lists and no expectation of backtracking. The forward-chaining program represents more a Prolog style of programming, with frequent backtracking and no lists. .FE

```
ask_which([A,B,C,D,E,F,G,H|L]) :-
screen_ask_which([A,B,C,D,E,F,G,H],[A,B,C,D,E,F,G,H]),
ask_which(L).
ask_which([]).
ask_which(L) :- length(L,N), N<9, N>0, screen_ask_which(L,L).
screen_ask_which([X|L],L2) :- length(L,N), length(L2,N2),
  N3 is N2 - N, write(N3), write(': '), questioncode(X,Q),
  write(Q), write('?'), nl, asserta(asked(X,no)),
screen_ask_which(L,L2).
screen_ask_which([],L2) :-
  write('Give numbers of questions whose answer is yes.'),
  read(AL), create_facts(AL,L2), nl.
create_facts([N|L],L2) :- item(N,L2,I), assertz(fact(I)),
  retract(asked(I,no)), asserta(asked(I,yes)), create_facts(L,L2).
create_facts([N|L],L2) :- not(item(N,L2,I)), create_facts(L,L2).
create_facts([],L2).
item(1,[X|L],X).
item(N,[X|L],I) :- N > 1, N2 is N-1, item(N2,L,I).
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
length([],1).
length([X|L],N) :- length(L,N2), N is N2+1.
```

Here's an example use, assuming the **questioncode** definitions for appliance diagnosis given earlier (Section 7.6):

```
?- ask_which
([device_dead,knobs_do_something,lights_out,cord_frayed,
handyperson,familiar_appliance,device_on,just_plugged,in_socket,
smell_smoke,liquids,heats,powerful,has(knobs),has('moving parts'),
has('knobs, switches, or other controls'),
hear(pop),hear('weird noise')]).
1: Does the device refuse to do anything?
2: Does changing the switch positions or turning
the knobs change anything?
3: Do all the lights in the house seem to be off?
4: Does the outer covering of the cord appear to be
coming apart?
5: Are you good at fixing things?
6: Are you familiar with how this appliance works?
7: Is the ON/OFF switch set to ON?
8: Did you just plug the appliance in?
Give numbers of questions whose answer is yes.[3,4,5,7].

1: Is the cord firmly plugged into the socket?
2: Do you smell smoke?
3: Have any liquids spilled on the appliance just now?
4: Does the appliance heat things?
5: Does the appliance require a lot of power?
6: Does the appliance have knobs?
7: Does the appliance have moving parts?
8: Does the appliance have knobs, switches, or other controls?
Give numbers of questions whose answer is yes.[6,8].

1: Did you hear a pop?
2: Did you hear a weird noise?
Give numbers of questions whose answer is yes.[].
yes
```

To verify that the correct facts were asserted, we can use Prolog's built-in **listing** predicate that takes one argument, a predicate name, and prints out every fact and rule with that name:

```
?- listing(fact).
fact(device_dead).
fact(lights_out).
fact(cord_frayed).
fact(handyperson).
fact(device_on).
fact(has(knobs)).
fact(has('knobs, switches, or other controls')).
yes
```

Fact order is very important to the operation and hence the efficiency of forward chaining (see Figure 6-3). Furthermore, if your definition of the **done** predicate is anything besides **not(fact(X))**, an incorrect fact order may prevent all correct conclusions. So it is a good idea to order question presentation from most important to least important, since facts are entered with **assertz**. To allow some user control, the preceding program asserts choices for each individual menu in user-supplied order. We set the menu size to eight questions in the program, but you can easily change it if you want to improve user control and your terminal screen is tall enough.

For most applications, you shouldn't put every possible fact into menus. You should put the things most common in rules in a first set of menus, and then run forward chaining and see what conclusions you reach. If no "interesting" conclusions are reached (interestingness could mean membership in a particular set), then present further menus to the user (perhaps about things mentioned on the right sides of the shortest remaining rules), assert new facts, and run forward chaining again. You can repeat the cycle as many times as you like. This is a *fact partitioning* trick for rule-based systems, as opposed to the *rule partitioning* trick in Section 7.8. For appliance repair for instance, the first menu could list key classificatory information, such as whether it has a heating element, whether it has mechanical parts, whether it gets electricity from an outlet or from batteries, and what part of the appliance is faulty. After forward chaining on these facts, further menus could obtain a detailed description of the malfunction.

Both menus and questionnaires should not "jump around" too much between topics as backward chaining often does. People are easily confused by rapid shifts in subject, and asking related questions, even irrelevant ones, makes them feel more comfortable. Doctors know this.

Ordering of output (conclusions) presentation is important too for forward chaining. Backward chaining just establishes one conclusion at a time, but forward chaining may establish a whole lot of interesting conclusions about some problem. In fact this is one of its advantages, that it can determine not one but multiple simultaneous problems are causing observed symptoms. If these conclusions are numerous, it's good to sort them by a fixed priority ordering.

Rule form conversions (*)

The rule forms needed for backward chaining, hybrid chaining, and forward chaining are all different. So it would be nice to automatically convert rules from one form to another. That means treating rules as data, something important in many different areas of artificial intelligence, and supported by most dialects of Prolog and Lisp.

Prolog does this with the **clause** predicate in most dialects. This has two arguments representing a left and a right side of a rule. A query on **clause** succeeds if its arguments can match the left and right sides of a database rule. (Remember, facts are rules without left sides.) So for instance

```
?- clause(a,R).
```

binds the variable **R** to the right side of the first rule it can find that has **a** as its left side. The arguments to **clause** can also be predicate expressions containing arguments themselves. So

```
?- clause(p(X),R).
```

will try to find a rule with a **p** predicate of one argument on its left side, and will bind that one argument (if any) to the value of **X**, and **R** to the query representing the right side of that rule | REFERENCE 2|. .FS | REFERENCE 2| Many Prolog dialects require that the predicate name in the first argument be bound, though arguments to that predicate name may have variables. If you have a list of all predicate names, you can iterate over them to do this. .FE

Facts are just rules with no left sides. And **clause** recognizes this, also matching facts in the database as well as rules. Then its first argument is bound to the fact, and the right side is bound to the word **true**. For example, if for the preceding query we have a fact **p(a)** in our database, one answer will be **X=a** with **R** bound to **true**.

To automatically convert a rule to forward chaining form, we can access it with **clause**, then convert the right side of the rule to a list. The second argument of the clause predicate is bound to a query, not a list, so we need a list conversion operation. This is called "univ" and is symbolized in most Prolog dialects by the infix predicate "**=..**", for which the stuff on the left side is a predicate expression and the stuff on the right side is an equivalent list. So we can say

```
?- clause(L,R), R =.. Rlist.
```

and **Rlist** is a list, as we need for forward chaining. Here's the full code for converting all rules in a Prolog database to forward-chaining form:

```
forward_convert :- clause(L,R), not(overhead_predicate(L)),
  R =.. Rlist, new_rule(L,Rlist), fail.
overhead_predicate(new_rule(X,Y)).
overhead_predicate(rule(X,Y)).
overhead_predicate(fact(X)).
new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).
overhead_predicate(forward_convert).
overhead_predicate(overhead_predicate(X)).
```

If your Prolog dialect doesn't allow **L** to be unbound, you can use:

```
forward_convert(All_predicates) :- member(L,All_predicates),
  forward_convert2(L).
forward_convert2(L) :- clause(L,R), R =.. Rlist, new_rule(L,Rlist), fail.
new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).
```

and you must query **forward_convert** with a list of all the predicates you want to convert.

But all this won't work in some Prolog dialects that treat commas and semicolons like infix operators. If the preceding doesn't work for you, try:

```
forward_convert(Preds) :- member(Pred,Preds), forward_convert2(Pred).
forward_convert2(Pred) :- clause(Pred,R), remove_commas(R,R2),
  new_rule(Pred,R2), fail.
remove_commas(true,[]).
remove_commas(S,[Y|L]) :- S=..[Comma,Y,Z], remove_commas(Z,L).
remove_commas(X,[X]) :- not(X=true), not(X=..[Comma,Y,Z]).
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).
```

We can apply the same approach to the hybrid control structure. We convert the right side of each backward-chaining rule to a list, insert a **not** expression at the front of this list, insert an **asserta** expression at the end of the list, and replace the original left side with the single symbol **r**. We then can use the **=..** operation in reverse (the left side unbound and the right side bound), converting the list of right-side expressions to a query. We enter this new rule into the Prolog database using **asserta** with this new rule as argument. Here's the full code:

```
hybrid_convert :- clause(L,R), R =.. Rlist,
  add_last(asserta(L),Rlist,Rlist2),
  R2 =.. [not(L)|Rlist2], asserta(r :- R2), fail.
add_last(X,[],[X]).
add_last(X,[Y|L],[Y|L2]) :- add_last(X,L,L2).
```

If your Prolog dialect doesn't allow **L** to be unbound, you can use:

```
hybrid_convert(All_predicates) :- member(L,All_predicates),
  hybrid_convert2(L).
hybrid_convert2(L) :- clause(L,R), R =.. Rlist,
  add_last(asserta(L),Rlist,Rlist2),
  R2 =.. [not(L)|Rlist2], asserta(r :- R2), fail.
add_last(X,[],[X]).
add_last(X,[Y|L],[Y|L2]) :- add_last(X,L,L2).
```

and if none of that works in your Prolog dialect, you can try this alternative hybrid-chaining program which doesn't directly convert Prolog rules but interprets them as needed:

```
hybrid(Leftsidelist) :- done.
hybrid(Leftsidelist) :- not(one_cycle(Leftsidelist)), hybrid(Leftsidelist).
one_cycle(Leftsidelist) :- member(P,Leftsidelist), clause(P,R),
remove_commas(R,R2), allfacts(R2), not(P), asserta(P), fail.
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
remove_commas(true,[]).
remove_commas(S,[Y|L]) :- S=..[Comma,Y,Z], remove_commas(Z,L).
remove_commas(X,[X]) :- not(X=true), not(X=..[Comma,Y,Z]).
allfacts([]).
allfacts([Term|L]) :- clause(Term,true), allfacts(L).
```

Indexing of predicates (*)

Prolog interpreters automatically index predicate names appearing on the left sides of rules, to help in backward chaining; that is, they keep lists of pointers to all rules with a particular predicate name, to speed finding them. But this won't help our rule-cycle hybrid and forward chaining implementations, for which we must do our own indexing if we want efficiency.

Let's take pure forward chaining as an example. Indexing could mean storing for each predicate name a list of which rule *right* sides it appears in. One way is to generalize the **rule** predicate of Section 7.10 to a **rules** predicate whose first argument is a predicate expression **P** and whose second argument is a list of pairs representing rules containing **P** on their right side. The first item of each pair is a rule left side, and the second item is a rule right side containing the specified predicate name. So the rules

```
a :- b, c.
d :- c, e, f.
```

are written as

```
rules(b,[[a,[b,c]]]).
rules(c,[[a,[b,c]],[d,[c,e,f]]]).
rules(e,[[d,[c,e,f]]]).
rules(f,[[d,[c,e,f]]]).
```

and just changing **pursuit**, the forward chaining program becomes:

```
forward :- done.
forward :- fact(F), doall(pursuit(F)), assertz(usedfact(F)),
retract(fact(F)), forward.
pursuit(F) :- rules(F,Rlist), member([L,R],Rlist), rule_pursuit(F,L,R).
rule_pursuit(F,L,R) :- member(F,R), delete(F,R,Rnew), new_rule(L,Rnew).
new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).
```

A rule with K expressions on its right side is repeated K times this way, so this indexing buys speed at the expense of space. The speed advantage comes from eliminating the **member** predicate in the earlier implementation; we have cached in advance the results of running **member**.

Clever indexing techniques have been developed in both artificial intelligence and database research, and much further can be done along these lines.

Implementing meta-rules (*)

Meta-rules are rules that treat rules as data, usually by choosing the one to use next (*conflict resolution*). Choosing means treating rules as data. So meta-rule implementation needs special rule formats like those for hybrid and pure-forward chaining, and can exploit **clause** and **=..** for rule-form conversion. We can represent meta-rules as rules with a special **prefer** predicate name on their left sides. The **prefer** will take four arguments: the left side of a rule 1, the right side of rule 1 (as a list), the left side of a rule 2, and the right side of rule 2 (as a list). It should succeed when rule 1 is preferable (to execute next) to rule 2. (Of course, none of your regular rules can have a **prefer** predicate name for this to work.) Here's an example:

```
prefer(L1,R1,L2,R2) :- length(R1,Len1), length(R2,Len2),
Len1 < Len2.
```

This says a rule with shorter right side should be preferred. Predicate **length** was defined in Section 5.5, and computes the number of items in a list:

```
length([],0).
length([X|L],N) :- length(L,N2), N is N2 + 1.
```

Another example:

```
prefer(a(X),R1,a(Y),R2) :- member(b(Z),R1), not(member(b(W),R2)).
```

This says that if two rules both conclude something about one-argument predicate **a**, one is preferred if it mentions one-argument predicate fact for the second) concluding about a one-argument predicate **a**, whether their arguments are constants or variables.

Meta-rules are most useful when they refer, unlike these two examples, to the current state of reasoning rather than the unvarying text of rules. As an example, consider this meta-rule to be used with pure forward chaining:

```
prefer(L1,R1,L2,R2) :- member(b,R1), not(member(b,R2)), fact(b).
```

This says to prefer a rule that mentions predicate expression **b** to one that doesn't, whenever **b** was proved a fact. A more useful meta-rule says to prefer the rule used most recently:

```
prefer(L1,R1,L2,R2) :- used(L1,R1,T1), used(L2,R2,T2), T1 > T2.
```

Here we assume **used** is a fact asserted after each successful rule application, stating the time a rule was used (times can be increasing integers). Such meta-rules permit flexible control adjustable to the processing context, something general-purpose control structures can't do by themselves.

Programmers can write these meta-rules at the same time they write the rules for an application. The set of meta-rules can express a lattice of orderings (or *partial ordering*) of rule preferences, but not necessarily a complete ordering or sorting. So it's possible for neither of two rules to be preferred to the other, in which case we can pick one arbitrarily.

Meta-rules enhance general-purpose control structures, and aren't a control structure by themselves. This means that meta-rule implementation is different for backward, hybrid, and forward chaining. With backward chaining, meta-rules pick a rule or fact (not necessarily the first in database order) to try to match to a predicate expression in a query or rule right side. With hybrid chaining, meta-rules pick a rule from the entire set of rules to run next. With pure forward chaining, meta-rules can both select a fact to try next and select a rule to try to match the fact to. As an example, here's an implementation of meta-rules with hybrid chaining.

```
metahybrid :- done.
metahybrid :- pick_rule(R), call(R), metahybrid.
pick_rule(R) :- clause(r,R), not(better_rule(R)).
better_rule(R) :- clause(r,R2), prefer(r,R2,r,R).
```

This assumes rules are written in the standard hybrid form with **r** on the left side and the two extra things on the right side. Rules are repeatedly executed (using the **call** predicate, which executes a query as if it were typed in) until the **done** condition is satisfied. A rule is chosen to execute only if no other rules are preferred to it according to the meta-rules.

Implementing concurrency (*)

Several Prolog dialects provide for concurrency of rules and expressions in rules. The emphasis is programmer's tools to indicate good places to do concurrency rather than automatic choice. For instance, a special "and" symbol, used in place of a comma, can specify "and"-parallelism on the right side of a rule. These approaches to concurrency are complicated and incompatible with each other, and we won't discuss them here.

Decision lattices: a compilation of a rule-based system (*)

Compilers and compilation are important concepts in computer science. Compilers take a program in an easy-to-read but slow-to-execute form and convert it to a more efficient, easier-to-interpret form. Compilation is often rule-based itself, especially often-complicated *code optimization*. But compilation techniques can also make artificial intelligence programs themselves more efficient. Since rule-based systems are further steps in power beyond traditional higher-level languages like Pascal, Ada, PL/1, and Fortran, secondary compilations are usually done before the primary compilation to machine language. These secondary compilations convert rules to the format of those languages: formats without references to backward, forward, or hybrid chaining, and with no backtracking, no postponing of variable bindings, and no multiway designation of inputs and outputs to predicate expressions. The decision lattice representation of rules introduced in Section 6.8 is one such secondarily-compiled format. It starts at some node in a lattice, and depending on how a user answers a question, it those to some next node in the lattice. Each question represents another branch point. When it gets to a node that is a *leaf* (a node from which it cannot go any further), it retrieves the conclusion associated with that node.

A decision lattice for a set of rules can be created systematically (albeit not algorithmically, since it involves some subjective judgment) from rules without "or"s on their right sides, by the following (similar to "automatic indexing" methods):

1. For every top-level or **diagnosis** rule, repeatedly substitute in the definitions for (right sides of) all intermediate predicates on its right side, until no more remain. If there is more than one rule proving an intermediate predicate, make multiple versions of the diagnosis rule, one for each possibility. (This is a useful compilation method even if you don't want a decision lattice; it's called *rule collapsing*.)

2. Examine the right sides of the new rules. Pick a predicate expression that appears unnegated in some rules and negated in an approximately equal number (the more rules it appears in, the better, and the more even the split, the better). Call this the partitioning predicate expression, and have the first question to the user ask about it. Create branches from the starting node to new nodes, each corresponding to a possible answer to this question. Partition the rules into groups corresponding to the answers, and associate each group with one new node (copies of rules not mentioning the predicate expression should be put into every group). Then remove all occurrences of the expression and its negation from the rules. Now within each rule group, separately apply this step recursively, choosing a predicate that partitions the remaining rules in the group best, and taking its questioning next.

An example will make this clearer. Suppose we have facts, **a**, **b**, **c**, **d**, and **e**, and possible diagnoses (final conclusions) **r**, **s**, **t**,**u**, and **v**. Suppose these are the rules:

```
r :- a, d, not(e).
s :- not(a), not(c), q.
t :- not(a), p.
u :- a, d, e.
u :- a, q.
v :- not(a), not(b), c.
p :- b, c.
p :- not(c), d.
q :- not(d).
```

For the first step (preprocessing), we substitute in the intermediate predicates **p** and **q**. (Intermediate predicates are just anything that occurs on both a left and a right side.)

```
r :- a, d, not(e).
s :- not(a), not(c), not(d).
t :- not(a), b, c.
t :- not(a), not(c), d.
u :- a, d, e.
u :- a, not(d).
v :- not(a), not(b), c.
```

For the second step we first examine right sides of rules to find something mentioned in a lot of rules that partitions them evenly as possible. Expression **a** is the obvious choice, because it is the only expression occurring in every rule. So we partition on **a**, deleting it from the rules, getting two rule sets:

```
r :- d, not(e).  /* Subdatabase for "a" true */
u :- d, e.
u :- not(d).

s :- not(c), not(d).  /* Subdatabase for "not(a)" true */
t :- b, c.
t :- not(c), d.
v :- not(b), c.
```

The first set will be used whenever the fact **a** is true, and the second set will be used whenever the fact **a** is false. In the first group **d** appears in all rules, so it can be the partitioning expression. Likewise, **c** can partition the second group. This gives four rule groups or *subdatabases*:

```
r :- not(e).           /* The "a, d" rule subdatabase */
u :- e.

u.                     /* The "a, not(d)" subdatabase */

t :- b.                /* The "not(a), c" subdatabase */
v :- not(b).

s :- not(d).           /* The "not(a), not(c)" subdatabase */
t :- d.
```

Three of the four groups are two-rule, for which one more partitioning gives a unique answer. The final decision lattice created by this analysis is shown in Figure 7-4.

Decision lattice compilations of rule-base systems can be easily written in any computer language, including Prolog. Give code names to every node in the decision lattice, including the starting and ending nodes. Then define a **successor** predicate of two arguments that gives conditions for one node to be followed by another node. For instance, for our previous example:

```
successor(n1,n2) :- askif(a).
successor(n1,n3) :- askifnot(a).
successor(n2,n4) :- askif(d).
successor(n4,u) :- askif(e).
successor(n4,r) :- askifnot(e).
successor(n2,u) :- askifnot(d).
successor(n3,n5) :- askif(c).
successor(n5,t) :- askif(b).
successor(n5,v) :- askifnot(b).
successor(n3,n6) :- askifnot(c).
successor(n6,t) :- askif(d).
successor(n6,s) :- askifnot(d).
```

Then we query a new **diagnosis** predicate defined:

```
diagnosis(Node,Node) :- not(successor(Node,X)).
diagnosis(D,Start) :- successor(Start,X), diagnosis(D,X).
```

For the preceding example we also need a way to query the facts:

```
questioncode(X,X) :- member(X,[a,b,c,d]), write('Is this correct: ').
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

And here's what happens:

```
?- diagnosis(X,n1).
Is this correct: a? no.
Is this correct: c? yes.
Is this correct: b? no.
X=v;
no
```

This program is a simple example of "search" programs, which we'll study in much more detail in Chapters 9, 10, and 11.

Summary of the code described in the chapter (*)

Warning: in using the following, make sure your own code does not redefine or even duplicate the definitions of any predicates used, or you can be in serious trouble.

The following rules assist question-asking for backward, forward, and hybrid chaining. The **ask** predicate asks the user a question, providing extra explanation if the user types a question mark, and returns the answer (after caching it). The **questioncode** and **explain** predicates must be provided by the programmer; the first decodes questions, and the second provides additional explanation for particular questions when the user has trouble understanding. The **askif** predicate handles yes/no questions; it succeeds if the user answers positively, fails if the user answers negatively, and requests another answer if the user answers anything else. Warning: do **abolish(asked,2)**) to erase memory before a new situation (problem), if you want to solve more than one situation.

```
/* Tools for questioning the user */
askif(Qcode) :- ask(Qcode,A), positive_answer(Qcode,A).
askifnot(Qcode) :- not(askif(Qcode)).


positive_answer(Qcode,A) :- affirmative(A).
positive_answer(Qcode,A) :- not(negative(A)),
  not(affirmative(A)), write('Please answer yes or no.'),
  read(A2), retract(asked(Qcode,A)),
  asserta(asked(Qcode,A2)), affirmative(A2).
```

```
ask(Qcode,A) :- asked(Qcode,A).
ask(Qcode,A) :- not(asked(Qcode,A)), questioncode(Qcode,Q),
  write(Q), write('? '), read(A2), ask2(Q,Qcode,A2,A).


ask2(Q,Qcode,'?',A) :- explain(Qcode), ask(Qcode,A).
ask2(Q,Qcode,A,A) :- not(A='?'), asserta(asked(Qcode,A)).

affirmative(yes).
affirmative(y).
affirmative(ye).
affirmative(right).
affirmative(ok).
affirmative(uhhuh).


negative(no).
negative(n).
negative(not).
negative(never).
negative(impossible).
negative(haha).
```

To do rule-cycle hybrid chaining, write the rules with **r** on their left sides, and an **asserta** of the original left side on the right end of the right side. No rules can contain **not**. Then query **hybrid**, using this program:

```
/* Problem-independent rule-cycle hybrid chaining */
hybrid :- done.
hybrid :- doall(r), hybrid.


doall(P) :- not(alltried(P)).


alltried(P) :- call(P), fail.
```

To do pure forward chaining, write the rules as facts with predicate name **rule**, for which the first argument is a left side and the second argument is the corresponding right side. No rules can contain **not**. Then query **forward**, defined this way:

```
/* Problem-independent forward chaining */
forward :- done.
forward :- fact(F), doall(pursuit(F)), assertz(usedfact(F)),
 retract(fact(F)), forward.


pursuit(F) :- rule(L,R), rule_pursuit(F,L,R).


rule_pursuit(F,L,R) :- member(F,R), delete(F,R,Rnew),
  new_rule(L,Rnew).


new_rule(L,[]) :- not(fact(L)), asserta(fact(L)).
new_rule(L,R) :- not(R=[]), asserta(rule(L,R)).


doall(P) :- not(alltried(P)).


alltried(P) :- call(P), fail.


member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).


delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- not(X=Y), delete(X,L,M).
```

When this program stops, the learned facts are left as assertions of **fact** and **usedfact** predicates. If you want to make sure that all possible conclusions are reached, use this definition of **done**:

```
done :- not(fact(X)).
```

If you want to speed up forward chaining, and your rule-based system can be represented as an and-or-not lattice, you can rewrite the preceding **rule_pursuit** rule as:

```
rule_pursuit(F,L,R) :- member(F,R), delete(F,R,Rnew), retract(rule(L,R)),
  new_rule(L,Rnew).
```

If you want to do forward chaining with predicate expressions having **not**s, first make sure the **not**s all refer to fact predicates. Then execute **full_forward**:

```
full_forward :- forward, doall(handle_not), forward.


handle_not :- rule(L,R), member(not(X),R), not(usedfact(X)),
  not(fact(X)), delete(not(X),R2), new_rule(L,R2).
```

Meta-rules can enhance other control structures. They can be written as rules with predicate name **prefer**, of four arguments (the left and right sides of two rules, respectively) that give conditions under which the first rule should be executed before the second rule. For meta-rules with hybrid chaining, execute the query **metahybrid** with this alternative code:

```
/* Problem-independent rule-cycle hybrid chaining using meta-rules */
metahybrid :- done.
metahybrid :- pick_rule(R), call(R), metahybrid.


pick_rule(R) :- clause(r,R), not(better_rule(R)).


better_rule(R) :- clause(r,R2), prefer(r,R2,r,R).
```

To implement menus as a way of getting facts to do forward chaining, execute **ask_which** with argument the list of question codes for the facts you want to check.

```
/* Menu generation for forward chaining */
ask_which([A,B,C,D,E,F,G,H|L]) :-
  screen_ask_which([A,B,C,D,E,F,G,H],[A,B,C,D,E,F,G,H]),
  ask_which(L).
ask_which([]).
ask_which(L) :- length(L,N), N<9, N>0, screen_ask_which(L,L).


screen_ask_which([X|L],L2) :- length(L,N), length(L2,N2),
  N3 is N2 - N, write(N3), write(': '), questioncode(X,Q),
  write(Q), write('?'), nl, asserta(asked(X,no)),
  screen_ask_which(L,L2).
screen_ask_which([],L2) :-
  write('Give numbers of questions whose answer is yes.'),
  read(AL), create_facts(AL,L2), nl.


create_facts([N|L],L2) :- item(N,L2,I), assertz(fact(I)),
  retract(asked(I,no)), asserta(asked(I,yes)), create_facts(L,L2).
create_facts([N|L],L2) :- not(item(N,L2,I)), create_facts(L,L2).
create_facts([],L2).


item(1,[X|L],X).
item(N,[X|L],I) :- N > 1, N2 is N-1, item(N2,L,I).


member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).


length([],1).
length([X|L],N) :- length(L,N2), N is N2+1.
```

To implement a decision lattice, draw the lattice and label all the nodes with unique names. Write rules defining the branches with the **successor** predicate, for example

```
successor(n5,n9) :- askif(device_dead).
successor(n5,n15) :- askifnot(device_dead).
```

which says that if you're at node n5, go to node n9 if the user responds positively to the **device_dead** question, otherwise go to node n15. Then to run the decision lattice call **diagnosis(D,<first-node>)**, which needs this definition:

```
diagnosis(D,Start) :- successor(Start,X), diagnosis(D,X).
diagnosis(Node,Node) :- not(successor(Node,X)).
```

Keywords:

diagnosis
character string
input coding
output coding
intermediate predicates
menu

.SH Exercises

7-1. (R,A,P) Improve the appliance-diagnosis program given in Section 7.6 so it can reach the following new diagnoses:

--the motor has burned out, for appliances that have motors;
--something is blocking the mechanical operation (like something keeping the motor from turning), for mechanical appliances;
--some internal wiring is broken (possible if appliance was dropped recently, or some other jarring occurred);

Add new questions and perhaps intermediate predicates to handle these diagnoses. Show your new program working.

7-2. (A,P) Figure out a simple way to prevent repeated printing of the same diagnosis in the backward-chaining appliance program as in Section 7.7. Show your method working on the appliance program.

7-3. (E) (a) Medical records often reference very-high-level and very-low-level concepts only, no intermediate concepts. Is this typical of other expert-system application areas?

(b) Suppose you must defend intermediate concepts in an expert system to your boss or representative of a funding agency. They could claim that intermediate concepts don't add capabilities to an expert system, just make its innards neater. Furthermore, intermediate concepts require extra design effort, and slow down operation by forcing reasoning to proceed by smaller steps. How would you reply in defense of intermediate predicates?

7-4. (P,H,G) Design a rule-based expert system to give debugging advice about Prolog programs. Use backward chaining in the manner of the appliance example, asking questions of the user. There is a lot of room for creativity in the choice of what the program knows about, but the program must help debug Prolog programs in some way. Be sure:

1. Your program is a rule-based expert system:

2. Your program contains at least 20 diagnosis rules (rules drawing a conclusion about what is wrong with a Prolog program), 12 of which have more than one kind of evidence "and"ed on their right side;

3. Your program uses at least three intermediate conclusions, conclusions that are neither equivalent to facts nor correspond to advice to the user;

4. Your program can reach at least seven different conclusions depending on the user's answers to the questions;

5. Three of the conclusions your program reaches appear on the left side of more than one rule (that is, there must be multiple paths to three conclusions);

6. Your program uses at least three kinds of history information, like how long the user has been programming Prolog, or whether parts of the user's program are already debugged;

7. The control structure of your program is not extensively "hard-wired" (for instance, there can't be a lot of branch specifications controlling where to go next);

8. As much as possible, all the variables and predicate names are semantically meaningful (that is, their function should be explained by their names; use numbers in your program only for (possibly) probabilities).

Use of probabilities or certainty factors is not necessary. Think about good ways to present your conclusions to the user.

7-5. (A) (For people who know something about probabilities.) Caching of facts in backward chaining is not always a good idea; it depends on the situation. Suppose we have rules for testing whether **a(X)** is true for some **X**, rules that require R units of time to execute on the average. Suppose to speed things up we cache K values of **X** for which the **a** predicate holds. That is, we place facts for those values in front of the rules that calculate **a(X)** as previously. Then to answer a query on the **a** predicate, we first sequentially search through all these cached facts, and use the rules only if we can't find a match in the cache.

(a) Suppose the probability is P that any one item in the cache matches an incoming query, and suppose the cache items are all different and do not have variables, so if one item matches the query no other item will. Under what approximate conditions will using the cache be preferable to not using the cache? Assume that |K < 0.1 / P|, and assume that each cache access requires one unit of time.

(b) Now assume the items of the cache are not all equally likely to be used. Often a *Zipf's Law* distribution applies, for which the probability of the most common item is P, the probability of the second most common item is P/2, the probability of the third most common item is P/3, and so on. Again assume cache items are mutually exclusive. Again assume that each cache access requires one unit of time. Under what approximate conditions is caching desirable now? Hint: the sum of |1 / I| from |I = 1| to |K| is approximately |log sub 2 (K + 1)|.

7-6. (P) Write a rule-based expert system to recommend cleaning methods for clothing. Obtain rules from a human "expert" on the subject. Use hybrid chaining in implementation for simplicity. Ask the user to give a single category that best describes the material they want to clean, and then ask additional questions as necessary to make its description more precise. Write at least twenty rules reaching at least ten different cleaning methods as conclusions. Use a lot of **a_kind_of** facts to classify the many kinds of clothing. You'll need to worry about rule order, and default rules will help.

7-7. The criteria for when to delete a rule in forward chaining were conservative: they miss other circumstances under which it would be good to delete rules. Define those circumstances, and explain how we could efficiently check for them.

7-8. (H,P) Write a program to diagnose malfunctions of a car using forward chaining in the pure form. Write at least fifty rules and use at least ten intermediate predicates. Use a repair manual as your source of expertise. Concentrate on some particular subsystem of the car to do a better job. Use the menu approach to defining facts. Provide for at least two invocations of forward chaining, and stop when one of a set of **diagnosis** facts is proved.

7-9. (H,P) Write a rule-based expert system to diagnose simple illnesses and medical problems (such as a "primary care" or "family practice" physician might handle). Use a medical reference book as your source of knowledge. Your program should be able to reach 30 different diagnoses, at least seven in more than one way, using at least five intermediate predicates. Run your rules with both backward chaining and forward chaining and compare performance. To do this, use the automatic conversion routines, or use the forward-chaining rule form and write your own backward-chainer. For forward chaining, let the user pick symptoms from a series of menus, and ask questions of the user directly for additional secondary facts. Show your rules working on sample situations.

7-10. (P) Write a rule-based expert system to choose a good way to graphically display data. Suppose as input this program loads a file of **data** facts of one argument, a list representing properties of some object. Suppose the output is a recommendation about whether to use bar graphs, pie graphs, two-dimensional plots, summary tables, etc. for particular positions in the data list. For instance, a recommendation might be to plot all the first items of data lists as an X coordinate against second items as a Y coordinate, and to draw a bar graph showing all the first items against third items. These graphing recommendations will need to pick subsidiary information about ranges to be plotted, binning (what values get grouped together), approximate dimensions of the display, extra lines, unusual values (*outliers*) that have been left out, etc. Generally you can find some graphical display for every pair of data item positions, but only some of these will be interesting and worth recommending. Interestingness can be defined in various ways, but should emphasize the unpredictability of the data: a nearly straight line makes an uninteresting graph. It will be useful to define a number of processing predicates, including a **column** predicate that makes a list of all the data values in position K in the data list, and predicate that applies a standard statistical test (look one up in a statistics book) to see if the values in two such columns are associated or correlated. You may want to consult a book giving recommendations for graphical display.

7-11. (A) The definitions of the **forall** and **doall** predicates both use the built-in Prolog predicate **call**, which takes a predicate expression as argument and queries it. (This is useful because the argument can be a variable bound within a program.) Use **call** to implement the following:

(a) **or(P,Q)** which succeeds whenever querying either **P** or **Q** succeeds

(b) **if(P,Q,R)** which succeeds whenever **P** succeeds then **Q** succeeds, or if **P** fails then **R** succeeds

(c) **case(P,N,L)** which succeeds whenever predicate expression number **N** of list **L** succeeds, where **N** is bound by executing predicate expression **P** (that is **N** must be a variable in **P** that is bound by **P**)

7-12. (H,P) Implement the other hybrid of backward and forward chaining mentioned in Section 6.4, the hybrid that alternates between forward and backward chaining. Show your program working on some sample rules and facts.

7-13. (P) Implement meta-rules for pure forward chaining. Use a **prefer** predicate like the one used for rule-cycle hybrid meta-rules.

7-14. (P) Consider meta-rules with pure forward chaining (the implementation is considered in Exercise 7-13). Assume there are no variables in the rules.

(a) Write a meta-rule to prevent the same conclusion from being reached twice.

(b) Write a meta-rule, and a little additional code to that written for Exercise 7-13, to prevent the same rule from being used twice in a row.

7-15. Convert the appliance-diagnosis program to a decision lattice. Try to minimize the number of questions required to a user.

7-16. (H,P,G) Write a rule-based system that, given the syllables of an English sentence in phonemic representation, figures out which syllables to stress. Such stress rules are relatively straightforward and can be found in many linguistics books. You'll need list-processing techniques from Chapter 5, and you must worry about the order the stress rules are applied.

7-17. (P) Write a program to give simple navigation commands to a mobile robot moving across a battlefield. Assume the battlefield is divided into squares, designated by integer X and Y coordinates ranging from X=0 to X=50 and from Y=0 to Y=50; the robot is not allowed to leave this area. The robot starts at location somewhere on the Y=0 line and it is trying to get to location (50,30). There are impassible bomb craters at (20,10), (10,25), and (20,40), all circles of radius 2. There is an impassible ravine extending along the Y=30 line from X=10 to X=50.

The robot begins moving at time 0. The rule-based system to control it should order one of only four actions: move one unit north, move one unit south, move one unit east, and move one unit west. Each move takes one unit of time. At time T=15 a bomb crater appears 4 units north of the robot; at T=25 a crater appears 3 units west; at T=30 a crater appears 3 units east; and at T=40 a crater appears 4 units east.

The robot should not be able to "see" craters and ravines until it is just at their edges. But suppose the robot knows the coordinates of the goal location and its current coordinates at all times, so it always knows where it is. Try to specify the robot's actions in general terms, that will work for any configuration of craters and ravines.

Try your program out with the robot starting at various locations along Y=0.

7-18. (E) Like most rule-based expert systems, our appliance diagnosis system doesn't reason about causal chains. In other words, it knows that a certain pattern of symptoms signal an underlying cause, but it doesn't know *why* the cause leads to the symptoms, the exact chains of cause and effect that explain each particular symptom. For instance, it knows that when a device isn't working at all, there might be a short in the cord; but it doesn't know the reason is that a short causes the resistance of a cord to be significantly lowered, causing a lot of electricity to flow into the cord, causing a lot of electricity to flow through a fuse, causing the metal conductor in the fuse to heat up, causing it to melt, causing the metal to flow, causing it to break the electrical connection, causing no electricity to go to the appliance cord, causing the appliance to not work at all. For what kinds of appliances and diagnosis circumstances could this lack of causal-chain reasoning be a problem? (We should use a quite different rule-based system then.) In what sense can the appliance expert system of this chapter be seen as a simplification of a more general kind of expert system?

Go to book index