Artificial Intelligence through Prolog by Neil C. Rowe

Appendix G: Answers to selected exercises

(Exercises with answers were marked with an "A".)

2-1. The **color** fact is better, because colors are less based on judgment and context than sizes are. We can define colors by frequency ranges of light waves, but there are no absolute standards of size: a "big" ship is a lot bigger than a "big" molecule. So if you want to refer to sizes in an artificial intelligence program, you probably should use something more precise.

2-2. Answer (ii). For format (i) the predicate names are just as general as the argument names, meaning many separate predicate names are necessary. And there's no internal indication with format (i) that these are supposed to be memo facts. But format (iii) is too general--any fact can be written this way--and the predicate "fact" isn't informative. Also, (ii) is typical of the predicates used in the chapter.

2-5. The speaker is forgetting that arithmetic operations are functions, and the function result must be an explicit argument in predicate expressions. So one-argument operations become two-argument predicate expressions, and two-argument operations become three-argument expressions. Other than that, the speaker's argument is pretty much correct: logic is part of mathematics, and Prolog is closely related to logic.

2-6. Write each **ship** fact as five two-argument subfacts, each with first argument the original first argument of the **ship** fact. (That is, use the ship name as a *key*.) The second argument will be:

--ship location for predicate name **location**;
--time at location for predicate name **time**;
--date at location for predicate name **date**;
--ship color for predicate name **color**;
--ship commander for predicate name **commander**.

So for instance

```
ship(enterprise,15n35e,1200,160285,gray,j_kirk).
```

becomes

```
location(enterprise,15n35e).
time(enterprise,1200).
date(enterprise,160285).
color(enterprise,gray).
commander(enterprise,j_kirk).
```

2-7. You might want to reason about a hypothetical situation, a situation not currently true. Advance reasoning might help you plan what to do if the situation occurs, saving you time then and letting you figure the best thing to do. Reasoning about hypothetical situations is also useful for debugging programs. For such reasoning to work, the set of hypothetical facts must be consistent: they can't be false in the sense of being self-contradictory.

3-1. (a) Use DeMorgan's Law:

```
?- not(a), not(b).
```

(b) Use DeMorgan's Law and rearrange:

```
?- (a;b), (not(a);not(b)).
```

3-2.(a)

--What job bosses the boss of a CS professor?
--What person is Rowe's boss?
--Who is the incumbent dean of either IP or IPS? (querier probably doesn't know which is the proper abbreviation)
--What person either has the provost or the IPS dean as a boss?
--What job other than Shrady's is bossed by the superintendent?

(b) First answers found, in order:

```
X=cschairman
Y=dean_ips

X=csprofessor
Y=cschairman
Z=lum

X=marshall

J=cschairman
P=lum

P=director_milops
```

3-5. (a) NM (every possible pair).

(b) N (N-1 for each new choice of **X**, and then one for the final query failure).

(c) NM (the two parts of the query are independent).

(d) The minimum of M and N.

(e) N--it's just like part (b).

(f) 0. The arguments to **p** facts may be totally different from the arguments to **q** facts.

3-6. (a) "Joe" is a constant in the queries, and "someone" is a variable. Every time a new Prolog query is asked, the variable bindings from the previous query are thrown away, and so this only affects the "someone" query. The mistake or *fallacy*, is in assuming such bindings persist.

(b) Assuming that a student gets a single grade on a single test:

```
?- grade(P,1,G), not(grade(P,1,a)).
```

(c) No, this isn't correct. Different people may have taken the two tests: some students may have dropped the course and others may have added the course in between. So the meaning of "class" may have changed.

3-9.(a) **X=a, Y=d, Z=e**. The first three **r** facts all eventually fail when used to match **X** and **Y** in the first predicate expression in the query. **X=a** with **Y=b** fails in the third query expression, **X=a** with **Y=c** fails in the fourth query expression, and **X=b** with **Y=a** fails in the second query expression.

(b) Four times: for **Y=b** and **Z=c**, **Y=b** and **Z=d**, **Y=c** and **Z=d**, and **Y=c** and **Z=c**. Notice that even though the matching for the second predicate expression can't affect the success of the third expression, the Prolog interpreter will always backtrack there from the third because it's the immediately previous expression--that is, the interpreter always backtracks *chronologically*.

4-2. (a) Make the left side a fact.

(b) Delete the rule entirely--it never does any good.

4-5. No, not if **a** is queried with an unbound argument and there is some **c** fact in the database, say **c(3)**. Then the one-rule form binds **X** to 3 in **a**, but the two-rule form can't. But when **a** is queried with a bound argument, the two forms give the same answers.

4-6. (a) Two different meanings of the word "mayor" are being confused: the job description and the person currently occupying that office. These meanings can't be equated. We'll discuss this distinction more in Chapter 12.

(b) The phrase "a movie star" acts like a noun, whereas "a little stupid" acts like an adjective. So "Clint is a movie star" must be represented by an **a_kind_of** relationship predicate, whereas "John is a little stupid" must be represented by a property predicate.

4-7. (a) A definition with two things "and"ed together on the right side.

(b) Two different facts about a department chairman (notice they do not define "chairman", just give properties of one).

(c) Here the "and" is more like an "or". So you could write two rules with the same left side (representing "a way to put out a fire") and different right sides for the different methods. Alternatively but less directly, you could write two facts asserting two different "fire methods".

(d) Two **a_kind_of** facts.

(e) This is an unusual situation in English, an *idiom* in which the words don't mean what they literally say. Here Tom and Dick are not only friends, but implied friends of one another. So two interrelated facts are being asserted, or just a disguised relationship-predicate fact.

4-10.(a) upward only. If some of Set have Property, then those same items will be present in a set including Set.

(b) No basis conditions are needed because facts will provide them. The induction step looks like this:

```
some(Set,Property) :- contains(Set,Set2), some(Set2,Property).
```

(c) downward only. If all of Set have Property, then all the items of any subset of Set have the same property.

(d) neither way. "Most" is a statistical generalization about a set, and there's never any guarantee that a statistical generalization about one set applies to a related set. (A subset might just happen to contain atypical items.)

4-11. (a) Yes, something inside something else inside a third thing is inside the third thing.

(b) Yes, it in inherits downward in one sense. If A is in front of B, and B contains C, then A is in front of C. However, downward inheritance doesn't work the other way: if A is in front of B and B is inside C, then A is not necessarily in front of C because C could contain A too: let A=battery, B=motor, and C=entire car.

(c) To better talk to people (as when giving them advice) because people understand nonnumeric descriptions better. And if people were given coordinates to locate things, they would need to do a lot of measurements from reference points, which is bothersome. Furthermore, it's tricky to compute object relationships from Cartesian coordinates describing object shapes; a lot of mathematics is necessary, especially when it won't do to just store the center of irregularly shaped objects. So it would be awkward for the computer to store coordinates internally and convert them to physical relationships to talk to people.

On the other hand, Cartesian coordinates are more precise descriptors than relationship predicates: you can compute more accurate spatial relationships, ways to reach parts, etc. Cartesian coordinates might take less memory space than relationship information since you wouldn't have to represent information about an object more than once; for N objects in a car there are |N * (N - 1)| relationships between any two, though inheritance could cut that down. And there are lots of ways to compress numbers to make storing them even more efficient, like keeping relative coordinates of parts. It's not true that relationship predicates are more car-independent (that is, the same description could apply to many different cars) than Cartesian coordinates, since coordinates could have associated tolerance intervals to make them more fuzzy. But "user-friendliness" is the chief disadvantage of Cartesian coordinates.

4-13. (a) The point of this part of the problem is to note the varied forms the same predicate can take. Only two predicates are needed: the **part_of** of Section 2.6, and a **contains** predicate that says that some of an object is a particular material (if you wanted to be fancy, you could have an additional predicate indicating that *all* of an object is of a certain material, but that's not necessary for the questions in part (d)). Note the statements refer to an Acme hotplate, so somehow you should say this.

```
part_of(acme_hotplate_cord,acme_hotplate).
part_of(acme_hotplate_body,acme_hotplate).
part_of(acme_hotplate_heating_element,acme_hotplate_body).
part_of(acme_hotplate_cover,acme_hotplate_body).
part_of(acme_hotplate_knob,acme_hotplate_cover).
part_of(acme_hotplate_wire,acme_hotplate_cord).
part_of(acme_hotplate_insulater,acme_hotplate_cord).


contains(acme_hotplate_heating_element,metal).
contains(acme_hotplate_knob,plastic).
contains(acme_hotplate_wire,metal).
contains(acme_hotplate_insulater,fiber).
```

(b)

```
?- contains(X,metal).
X=acme_hotplate_heating_element;
X=acme_hotplate_wire;
no.
?- part_of(X,acme_hotplate_body).
X=acme_hotplate_heating_element;
X=acme_hotplate_cover;
no.
```

(c)

```
part(X,Y) :- part_of(X,Y).
part(X,Y) :- part_of(X,Z), part(Z,Y).


contains(X,M) :- part_of(Y,X), contains(Y,M).
```

If you use a predicate for things entirely composed of a particular material--call it **all_contains**--then you need one more rule:

```
contains(X,M) :- all_contains(X,M).
```

(d) The difficulty in the second question is that any **not** must refer to a bound variable. The apparent context is parts of the Acme hotplate. So first define:

```
thing(X) :- part_of(X,Y).
thing(X) :- not(part_of(X,Y)), part_of(Y,X).
```

That is, a "thing" is anything mentioned in **part_of** facts. Here are the results for all three questions:

```
?- contains(X,plastic).
X=acme_hotplate_knob;
X=acme_hotplate;
X=acme_hotplate_body;
X=acme_hotplate_cover;
no.

?- thing(X), not(contains(X,fiber)).
X=acme_hotplate_body;
X=acme_hotplate_heating_element;
X=acme_hotplate_cover;
X=acme_hotplate_knob;
X=acme_hotplate_wire;
no.

?- contains(X,metal), contains(X,fiber).
X=acme_hotplate;
X=acme_hotplate;
X=acme_hotplate_cord;
no.
```

5-1. Note no **is** or **=** is needed.

```
max(X,Y,Z,X) :- not(Y>X), not(Z>X).
max(X,Y,Z,Y) :- not(X>Y), not(Z>Y).
max(X,Y,Z,Z) :- not(X>Z), not(Y>Z).
```

5-2. Write as facts instead of variables. Easy fact representation is a big advantage of Prolog over languages like Pascal.

```
translate(1,integer_overflow).
translate(2,division_by_zero).
translate(3,unknown_identifier).
```

In general, **is** and **=** should rarely be used for anything other than arithmetic and an occasional local variable. Instead, use the automatic facilities of Prolog for checking and binding values.

5-8.(a) With rules for inferring both starts and ends from the other, we must be careful to avoid infinite loops. We can do this by defining new predicates **inferred_end** and **inferred_start**.

```
inferred_end(E,T) :- end(E,T).
inferred_end(E,T) :- start(E,Tstart), duration(E,Dur),
  T is Tstart + Dur.

inferred_start(E,T) :- start(E,T).
inferred_start(E,T) :- end(E,Tend), duration(E,Dur), T is Tend - Dur.
```

(b) The second event must be completely over before the first event starts or else **after** is not a proper term to use in English.

```
after(E1,E2) :- inferred_end(E2,E2end), inferred_start(E1,E1start),
  E1start > E2end.
```

(c) The usual English meaning of **during** is that an event occurred entirely inside the duration of the other.

```
during(E1,E2) :- inferred_start(E1,E1start), inferred_start(E2,E2start),
  inferred_end(E1,E1end), inferred_end(E2,E2end),
  E1start > E2start, E1end < E2end.
```

(d) The rules should come after all the facts, so facts will always be preferred to answer queries. The order of the rules among themselves doesn't matter except for the two **inferred_start** and **inferred_end** rules, because order of rules having the same predicate name is the only thing the Prolog interpreter considers--it indexes facts and rules with the same predicate name.

5-9.(a) **speed(<gear_number>,<speed_in_rpm>)**. Positive speeds can be clockwise, negative counterclockwise.

(b)

```
speed(G,S) :- same_shaft(G,G2), speed(G2,S).
```

(c)

```
speed(G,S) :- meshed(G,G2), teeth(G,TG), teeth(G2,TG2), speed(G2,S2),
  S is 0 - ( S2 * TG2 / TG ).
```

(d) Use the following database:

```
speed(g1,5000).
```

```
same_shaft(g2,g1).


meshed(g3,g2).
meshed(g4,g2).


teeth(g1,100).
teeth(g2,30).
teeth(g3,60).
teeth(g4,90).


speed(G,S) :- same_shaft(G,G2), speed(G2,S).
speed(G,S) :- meshed(G,G2), teeth(G,TG), teeth(G2,TG2), speed(G2,S2),
  S is 0 - ( S2 * TG2 / TG ).
```

and issue this query:

```
?- speed(g4,S).
```

The one **speed** fact doesn't apply, so the first **speed** rule is tried. But there is no **same_shaft** fact with **g4** as first argument, so the rule fails. Now the second **speed** rule is tried. Gear g2 can be matched to **g2**, with **TG=90** and **TG2=30**. Then we must find the speed of **G2=g2**. No fact applies, but we can use the first **speed** rule: **g2** is the first argument to a **same_shaft** fact having **g1** as second argument, so its speed is the speed of **g1**. And a fact says the speed of **g1** is 5000. So the speed of g2 is 5000, and we can do the calculation in the second "speed" rule as

```
S = 0 - ( 5000 * 30 / 90 ) = -1667 rpm
```

and the variable **S** in the original query is bound to that number.

(e) You could get infinite loops if you had extra redundant facts, like if you had both **meshed(g2,g1)** and **meshed(g1,g2)**. You could also get infinite loops if you had more than one inference rule of either the previous types, as for instance if you had

```
speed(G,S) :- same_shaft(G,G2), speed(G2,S).
speed(G,S) :- same_shaft(G2,G), speed(G2,S).
```

Generally speaking, gears are intended to transmit power in one particular direction, so you only need one rule. You can encode that direction in the **meshed** facts.

(f) The gears wouldn't turn or would self-destruct; such a contradiction of rotation speeds is an impossibility.

5-14. (a)

```
member(X,L) :- append(L2,[X|L3],L).
```

(b)

```
last(X,L) :- append(L2,[X],L).
```

(c)

```
deleteone(I,L,DL) :- append(L1,[I|L2],L), append(L1,L2,DL).
```

(d)

```
before(X,Y,L) :- append(L2,[X|L3],L), append(L4,[Y|L5],L3).
```

5-17. (a) It replaces every other word in a list with the word "censored", binding the result to the second argument. The rule (third line) just says if you can find an **X** and **Y** at the front of your list first argument, change the **Y** to the word "censored" after recursively doing the same on the rest of the list.

That's a declarative analysis. A procedural understanding comes from figuring out what happens with simple example lists:

```
?- mystery([],Z).
Z=[]
?- mystery([a],Z).
Z=[a]
?- mystery([a,b],Z).
Z=[a,censored]
```

For the last example, the third rule applies for the first time. So **X** is matched to **a**, **Y** is matched to **b**, and **L** is matched to **[]**, the empty list. The rule involves a recursive call with first argument **L**, but **L** is the empty list, it's the same situation as the first example preceding, and **M** is bound to the empty list too. So the rule says to take **M** and **X** and the word "censored" and assemble them into the result, the list whose first two items are **X** and "censored", and whose remainder is **M**. But **M** is empty, so the result is **[a,censored]**.

Now consider the next hardest case, an example list with three items:

```
?- mystery([a,b,c],Z).
```

The third rule must be again used. It matches **X** to **a**, **Y** to **b**, and **L** to **[c]**, the list containing only **c**. The rule says to do a recursion with **[c]** the first argument. But to solve that the second rule will do, so **M** is bound to **[c]**. So the answer is assembled from **X**, the word "censored", and that **M**, and the result for **Z** is **[a,censored,c]**.

Now consider an example list with four items:

```
?- mystery([a,b,c,d],L).
```

In the third rule, **X** is matched to **a**, **Y** is matched to **b**, and **[c,d]** is matched to **L**. The recursion applies mystery to the two-item list **[c,d]**, which by analogy to the preceding two-item example gives **[c,censored]**. So we assembled the answer from **X**, the word "censored", and the list **[c,censored]**, and the result for **Z** is **[a,censored,c,censored]**. So it looks like the program changes every alternate word in a list to the word "censored".

(b) One for even-length lists, one for odd-length. The rule lops off two items from the list on each recursion, so you can get into two different final situations.

5-19. (a) |N + 1| times, once for every item in the list plus once for the empty list; all these invocations will fail.

(b) |(N - 1) / 2|. There are 0 calls if the item is first in the list, one call if the item is second in the list, two calls if the item is third in the list, and so on. These terms form an arithmetic series. The average value of an arithmetic series is the average of the first and last numbers, in this case 0 and N - 1.

5-22. Use a list to keep arguments previously used, and just check to make sure a new value found is not in the list. At every recursion, add the new argument to the list. So:

```
a3(X,Y) :- a2(X,Y,[]).


a2(X,Y,L) :- a(X,Y).
a2(X,Y,L) :- not(member(X,L)), a(X,Z), a2(Z,Y,[X|L]).


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

Then you query **a3** to compute relationships by transitivity. For instance, suppose you have the database:

```
a(r,s).
a(s,t).
a(t,r).
```

(For instance, predicate **a** might mean "equals" or "is near".) Then the query

```
?- a(r,t).
```

will succeed as it should, as will

```
?- a(r,r).
```

But the query

```
?- a(r,u).
```

will fail as it should.

6-1.(a) Forward chaining means reasoning from facts to goals. The first fact is **c**, and it is mentioned in rules R5 and R8, so expressions in those rules are matched in that order. The right side of R8 is eliminated, so **b(X)** is made a new fact. (X isn't bound, but that's OK.) The "b(X)" goes at the front of the facts, so we now can match expressions in rules R1 and R3, but neither of those rules is eliminated yet.

So we pick the next fact **l**. This matches the right side in R7, and **g(X)** is made a new fact at the front of the fact list. This in turn matches an expression in R2. We next pick fact **e(a)** and this matches an expression in R5 only. So since **c** was already matched, R5 succeeds, and the fact **d** is asserted. But **d** is a goal, so we stop.

(b) Now we reason from goals to facts. The **f(X)** is the first goal (hypothesis), invoking R2, and we must prove **a(X)**. This in turn invokes R3, which invokes R8. R8 succeeds because **c** is a fact. But **i** is not a fact (nor are there any rules for it), so R3 fails. R3 is the only rule for proving **a(X)**, so R2 fails too.

So we try the next goal, **d**. R4 is the first applicable rule, and as before, **i** is not a fact nor provable, so R4 fails. We thus invoke R5 to prove **d**. The fact **e(a)** matches **e(X)**, and **c** is a fact, so **d** is proved.

(c) Yes, because if **c** and **j(b)** come before **e(a)** in the initial facts, the alternative goal **k(b)** will be proved first instead.

(d) No, fact order doesn't affect backward chaining unless there are facts with the same predicate name, which isn't true here. (If some of the facts had the same predicate name, different variables might be bound in the final goal, a problem that doesn't arise unless goals have variables.)

6-5. (a) **u**, **b**, **m(12)**, **a**. Steps in order:

1. Fact **r** creates new rules **t :- s.** and **u :- v.**
2. Fact **v** creates the new rule **a :- t**., and proves the fact **u** .
3. Fact **u** creates the new rule **a :- b, not(t).**
4. Fact **c** proves the fact **b**.
5. Fact **b** creates the new rules **a :- not(t).** and **m(X) :- n(X).**
6. Fact **n(12)** proves **m(12)**.
7. Fact **m(12)** proves nothing.
8. No facts remain unexplored, so we examine the rule with the **not**. Since **t** is not a fact, the **not** succeeds, and this proves **a**.
9. Fact **a** proves nothing.

(b) **b**, **u**, **m(12)**, **a**. Fact order doesn't matter here since all facts have different predicate names. Steps in order:

1. The first three rules fail because all mention at least one thing that doesn't match a fact.
2. The fourth rule succeeds since **c** is a fact, so the new fact **b** is asserted. The rule is deleted.
3. The fifth rule fails.
4. The sixth and last rule succeeds because **v** and **r** are both facts. So new fact **u** is asserted. The rule is deleted.
5. The first two rules again fail.
6. The third rule succeeds because **n(12)** and **b** are now facts. New fact **m(12)** is asserted. This rule is not deleted because it contains a variable on its left side.
7. No further rules succeed.
8. Now the **not**s are considered. In the second rule, **t** is not a fact, so **not(t)** succeeds, as well as **b**

and **a**. So fact **a** is asserted.

9. No further rules succeed.

(c) Index the predicate names on right sides of rules, and index **not**'s. Then you need only check rules that contain predicate names of facts proved on the last cycle, and you can find **not**s fast. Also you can also delete rules that succeed whose left sides do not contain variables.

6-7. The minimum of L and S. Each cycle must prove something to keep things going. There are only L different things provable, so L is an upper bound. But each cycle must match also at least one new predicate name on a rule right side so that a new rule can succeed. So if there are S right-side names, S is an upper bound too.

6-8.(a) Backward chaining means reasoning from hypotheses to facts. Here the hypotheses are actions the robot may take, which we must consider in the order given. Hence the first hypothesis we examine is "turn around", and the rule order is:

> R1 fails, turn-around hypothesis fails, try stop-and-wait hypothesis, R3, R4, R16, R8 fails (no moving branches visible), R9 succeeds (so second object is an animal), R16 fails (four branches not present), R17 succeeds (so second object is a person), R4 succeeds, R3 fails (because robot isn't "beneath" anything), R5 succeeds (the only other way to achieve the stop-and-wait hypothesis, and it succeeds because we've already proved the second object is an animal).

You could also put R10 and R11 before first invocation of R16, if you interpreted "person or vehicle" in rule R4 as a single concept.

(b) Now we take the facts, in order, and reason about what other facts they imply:

> --Fact F1 matches expressions in rules R13, R14, and R16.
>
> --Fact F2 matches expressions in rule R13.
>
> --Fact F3 matches expressions in rule R12.
>
> --Fact F4 matches expressions in rule R12, and that rule is completely satisfied. Hence the first object is an obstacle, and call that fact F4.1.
>
> --Fact F4.1 matches expressions in rules R6, R13, R14, and R15. Now R13 is completely satisfied. Hence the object to the right is a bush, and call this fact F4.2.
>
> --Fact F4.2 matches expressions in rules R2 and R3.
>
> --Fact F5 matches expressions in rules R4 and R5.
>
> --Fact F6 matches expressions in rules R9 and R17, and R9 is satisfied. Hence the second object is an animal, and call that fact F6.1.
>
> --Fact F6.1 matches expressions in R16 and R17, and R17 is completely satisfied. Hence the second object is a person, and call that fact F6.2.

--Fact F6.2 matches expressions in rule R4, and that rule is completely satisfied. Hence we must hide, and call that fact F6.3.

--Fact F6.3 matches expressions in rule R2, and that rule is completely satisfied. Hence we turn towards the bush and move a short distance.

(c) The backwards chaining used a fixed-rule-priority conflict resolution, and forward chaining used a fixed-fact-priority (with focus-of-attention) in addition to fixed-rule-priority. An alternative for forward chaining would be to add new facts at the end of the facts, while we still fetch from the front as with a queue. This would be good when we want to infer everything possible from a set of facts. An alternative for backward chaining would be a "biggest-rule-first" strategy in which the rule with the most conditions in its "if" part is tried first. Concurrency and partitioning ideas could work for both forward and backward control structures. We could also use a "most-related-rule" criterion, if we define "relatedness" for any rule pair.

6-9. (a) No, since a conclusion may depend on the absence of some fact. That fact may no longer be absent when a file is loaded in later.

(b) Yes, because caching just makes things already found easier to verify. Provability of conclusions is not affected, just efficiency.

(c) No, since new facts could be asserted that are unreachable by backward chaining. So a conclusion that depends on the absence of something might suddenly go from being true to being false.

6-12. (a) Forward chaining sounds better, since the system only need reason about sensor readings that changed from its last analysis (assuming we cache the conclusions from the last analysis), and changes to sensor readings will be relatively rare. And probably a lot of different goals (conclusions) can be reached.

(b) It's true that forward chaining caches automatically, so extra caching would seem unnecessary. But the best way to use this rule-based system is to run it repeatedly every second or so. Then conclusions learned during the last second could be useful in reasoning during this second--especially conclusions based on things that don't change often--so caching the previous conclusions could be useful. Also, caching could mean storing (in the rule-based system database) facts about the condition of sensors instead of directing the computer to look at its input/output ports to find those values. This also will be good because many sensors will infrequently change with time.

(c) Virtual facts don't make sense with forward chaining. If you suggested backward chaining in part (a), virtual facts might make sense if you interpret them as "queries" to sensor input/output ports. If you interpret them as queries to the user, they don't make sense because that would be an big imposition on the user.

(d) Yes, since there are clear categories of rules that don't interact much. For instance, the rules for handling burglars might not be in the database all the time, but only loaded when there is good evidence that a burglar is present. Rules for fires and other emergencies are similar. However, the rules for *recognizing* potential burglar and fire situations, that decide to load the detail-analyzing code, must be kept in main memory at all times.

(e) This is a closer judgment than the other parts of this question, but decision lattices are probably not a good idea. In the first place, you don't want to use them in a way so that you actually ask a human questions--that would be an imposition. But the decision lattice could "ask" questions of the sensors to get their readings.

The main difficulty is in building the decision lattice in the first place. There will be many different situations to handle, and it's hard to figure out what questions distinguish situations well, especially when houses are all different and you don't know what situations you'll get into; and it's the very rare emergency situations that are most critical to program properly. Furthermore, decision lattices are hard to modify, and hard to partition into modules. So for the advantages of a decision lattice, it would seem better to go with an and-or-not lattice, for which the translation from rules to lattice is much easier.

(f) Yes. Improved speed in inferencing is an obvious advantage over noncompiled forms. And-or-not lattices have the advantage of being easy to partition if the original rule-based system was easy to partition, like this one; partitioning a decision lattice is much harder, and since this application can easily be complex, inability to partition into modules is a serious disadvantage. And-or-not lattices are easier to modify than decision lattices, since they correspond closely to rules. And-or-not lattices could also standardize certain commonly-made inferences, so these could be mass-produced and a smaller, more house-specific computer could handle the rest of the reasoning. For instance, the conditions for suspecting a burglar and loading burglar-analysis rules are pretty much standard for all houses and with all burglars: check for unusual noises and for windows and doors being forced open, especially at night and when the house is not occupied. As another example, equipment requiring complicated continuous monitoring like an aquarium could be monitored according to a special and-or-not lattice supplied by the aquarium manufacturer, and only reports of major problems passed on to the central processor. It's true that and-or-not lattices have trouble handling variables, but this rule-based system doesn't much need them: use no-argument predicates to represent sensor values and value ranges.

7-1. Here's one way:

```
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')), askifnot(device_on).
diagnosis('Cord not in socket properly') :- power_problem,
  klutz_user, askif(just_plugged), askifnot(in_socket).
diagnosis('Internal break in the wiring') :-
  power_problem, jarring.
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 or switches')).
diagnosis('Motor burned out') :- askif(smell_smoke),
  mechanical_problem.
diagnosis('Something blocking the mechanical operation') :-
  mechanical_problem.
diagnosis('Kick it, then try it again') :- mechanical_problem.
diagnosis('Throw it out and get a new one').


power_problem :- askif(device_dead).
power_problem :- askif(has('knobs or switches')),
  askifnot(knobs_do_something).
```

```
power_problem :- askif(smell_smoke), not(heating_element).


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).


jarring :- askif(dropped).
jarring :- askif(jarred).

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(code_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 ').
questioncode(dropped,'Has the appliance been dropped recently').
questioncode(jarred,'Has the appliance been violently jarred recently').
```

7-2. Use the same idea of **coded_diagnosis** at the end of Section 7.4. Query instead of **diagnosis**:

```
better_diagnosis(D) :- diagnosis(D), not(found(D)), asserta(found(D)).
```

Now if the user types a semicolon after a diagnosis, the program will backtrack through the **asserta** and **found** to try to find another, but if D gets rebound to any previous value, the **not** will fail and the interpreter returns to "diagnosis". This also works for any order of diagnosis rules; rules for the same diagnosis need not be together.

7-5. (a) One unit of time is needed if the first cached item matches the query, two units of time if the second matches the query, three units of time if the third, and so on. Since we were told the probabilities are mutually exclusive, the probability that no item in the cache matches the query is |1 - KP|. Then the condition for the problem is

```
P + 2P + 3P + ... + KP + ( 1 - KP ) ( R + K )    <   R
```

The left side of the inequality contains an arithmetic series, so we can substitute its formula and get

```
P K ( K + 1 ) / 2   +   ( 1 - KP ) K   <    KPR
```

or

```
( K + 1 ) / 2   +   ( 1 / P )   -   K   < R
```

```
( 2 + P - PK ) / 2P   <   R
```

Since |KP < .1|, it's also true that |P < .1|, so we can approximate the inequality by just

```
PR > 1
```

(b) Analysis is similar, but the sum over the cache items has a different formula:

```
P + ( 2 * P / 2 ) + ( 3 * P / 3 ) + ... ( K * P / K )   =   KP
```

so the criterion becomes

```
KP + ( R + K ) ( 1 - P log sub 2 ( K + 1 ) )   <   R
```

or

```
K ( P + 1 )   <   P ( R + K ) log sub 2 ( K + 1 )
```

If |PK < .1| then because |P < PK| we know P is negligible compared to 1, so we can approximate this by

```
1    <   P ( ( R / K ) + 1 ) log sub 2 ( K + 1 )
```

7-11. (a)

```
or(P,Q) :- call(P).
or(P,Q) :- call(Q).
```

(b)

```
if(P,Q,R) :- call(P), call(Q).
if(P,Q,R) :- not(call(P)), call(R).
```

(c)

```
case(P,N,L) :- call(P), item(N,L,I), call(I).


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

8-3.(a) So **<prob1>** is |200 / 500 = 0.4|, and **<prob2>** is |800 / 1600 = 0.5|. The 1200 flat tires are irrelevant.

(b) The ratio 70/101 is approximately 0.69, and this is closest to 0.7, the independence-assumption combination (obtained from |1 - (1 - 0.4) (1 - 0.5)|). Conservative assumption gives |"maxfunction" (0.4 , 0.5) = 0.5|, and liberal gives |0.4 + 0.5 = 0.9|.

8-4.(a) conservative, since the function is the maximum of the two arguments.

(b) liberal. It can't be conservative since the values in the table are better than the previous table. It can't be independence-assumption because the combination of two "probably"s is a "definitely", something impossible with the independence formula (that is, the independence formula can't give a 1.0 unless one of its inputs is 1.0). The arithmetic (see part (c)) is consistent with liberal combination too.

(c) Let |x| be the probability of "probably not", |y| the probability of "probably". Then the table and the liberal-combination formula say that |x + x = y| and |minfunction (x + y , 1.0) = 1.0 |. So |2x = y| and |minfunction (3x , 1.0) = 1.0 |. Hence |x > 0.3333| from the last equation. But |y| must be less than 1.0 ("probably" should be less certain than "definitely") so |2x < 1.0| and |x < 0.5|. So |x| can be anything between 0.3333 and 0.5, and |y| must be correspondingly between 0.6667 and 1.0. (Notice that an infinity of possible answers doesn't mean *any* answer is right.)

8-5. No. There is no problem with either backward chaining, forward chaining, or any hybrid. Facts must always have probability values filled in (or else be certain, implying a probability of 1.0), so any rules combining those probabilities will work no matter how they're used.

8-6.(a) You must be careful to count separately the length of the list and the number of probabilities greater than 0.5. One way to do it:

```
orcombine(PL,P) :- numbigprobs(PL,Nbig), length(PL,N), P is Nbig / N.


numbigprobs([],0).
numbigprobs([P|PL],N) :- numbigprobs(PL,N), P < 0.5.
numbigprobs([P|PL],N) :- numbigprobs(PL,N2), P >= 0.5, N is N2 + 1.


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

Another way is to define a **censor** predicate that removes probabilities from a list that are less than 0.5:

```
orcombine(PL,P) :- length(PL,N), censor(PL,CPL), length(CPL,Nbig),
  P is Nbig / N.


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


censor([],[]).
censor([P|PL],[P|PL2]) :- censor(PL,PL2), P >= 0.5.
censor([P|PL],PL2) :- censor(PL,PL2), P < 0.5.
```

(b) Some answers:

--There are abrupt changes in the total probability when probabilities vary slightly from 0.5. It doesn't seem reasonable that any evidence combination method have such abrupt "jumps" as probabilities vary slightly.

--The number 0.5 is arbitrary--there's no good reason for it.

--The total probability with this method can be less than the "conservative" combination value; for example, **[0.4,0.4]** has 0 total with the method, 0.4 with the "conservative" method. But the conservative method gives the smallest logically reasonable value, so this method isn't always reasonable.

--This method is inaccurate when few probabilities are combined. If there's only one probability in the list presented for combination, then the cumulative probability is either 0 or 1, totally ignoring the probability of that one item.

--This method can't handle "cascades" very easily. That is, if the result of this method is an input to a combination in another rule, we've lost the important information about the number of probabilities contributing to the first result, which should greatly affect how much it should be counted for in the second combination.

8-9. (a) Newspapers and television report unusual and striking events, so you don't hear about the many more people that spent their money on the lottery and didn't win anything.

(b) In this example, successes in the lottery are broadcast much more strongly than failures, causing overestimation of the probability of success. This "biased sampling" problem is a chief difficulty in building rule-based systems from data: you must be careful there aren't reasons why certain data is overreported or underreported. For instance, embarassment might make people underestimate the frequency with which they forget to plug appliances in.

9-2. For inheritance, the goal condition is to find some other target object that has the property value you want, and has an inheriting link, like **a_kind_of**, to a given object. Looking for such a target object is a search starting from the given object. There can be several inheriting links for an object, each representing an alternative direction to explore, and these represent branches. And one choice affects later choices.

9-10. Just subtract K from the values of the old evaluation function to get a new evaluation function which is guaranteed to be a lower bound. Then you can use A* search with the optimality guarantee.

9-11.(a) See Figure G-1.

(b) Choose the state (among those so far discovered at some point in the problem) that has lowest evaluation function value. State sequence: a, d, c, b, f, e, g, h.

(c) Add the cost to reach a state to the evaluation function. Costs to states: a: 0, b: 2, c: 5, d: 9, e: 7, f: 11, g: 7, h: 11. Sum of cost and evaluation function: a: 10, b: 10, c: 12, d: 14, e: 13, f: 15, g: 16, h: 25. State sequence: a, b, c, e, d, f, g, h.

9-13. (a) All **[<left-bank>,<right-bank>]**, where **<left-bank>** and **<right-bank>** are lists together containing the symbols man, lion, fox, goose, corn, and boat. And **<left-bank>** is things on the left side of the river, **<right-bank>** things on the right.

(b) The start is **[[man,lion,fox,goose,corn,boat],[]]** and goal is **[[],[man,lion,fox,goose,corn,boat]]**.

(c) Man and boat; man, lion, and boat; man, fox, and boat; man, goose, and boat; man, corn, and boat; man, fox, corn, and boat; or man, goose, corn, and boat move from one side of the river to the other.

(d) The starting state has only one successor, **[[lion,goose],[man,fox,corn,boat]]**, which has three successors of its own (besides the starting state):

```
[[man,lion,goose,boat],[fox,corn]].
[[man,lion,fox,goose,boat],[corn]].
[[man,lion,goose,corn,boat],[fox]].
```

(e) |(1 + 3) / 2 = 2|

(f) Each thing can be in one of two places, and there are five things (the man must always be with the boat), hence |2 sup 5 = 32|.

(g) No, subparts interact.

9-15.(a) The set of all possible coating patterns.

(b) No, there are many different states (coating patterns) that satisfy the goal conditions, and no easy way to find them. (If you knew any of them, you wouldn't need to search.) So backward search isn't possible, and hence bidirectional search isn't either. (Notice that despite similarities, this problem is quite different from city-route planning: the goal is a configuration of coatings, not a location.)

(c) It always decreases. Every branch involves coating a grid cell, thus removing it from the pool of coatable cells. As you proceed, previously-coatable cells may become uncoatable because they would connect coated regions that shouldn't be connected, so the branching factor may decrease by more than one per level, but it will always decrease.

(d) Many heuristics are possible, including:

> --Prefer to coat cells adjacent to the last cell you coated (a "focus of attention" heuristic).
> --Prefer to coat cells that lie on a straight line between points you want to connect.
> --As soon as a desired connection is made between two points, prefer not to coat any cells within five cells of either of those points.
> --Prefer to coat cells that continue a straight line (e.g., coat [X,Y] if [X-1,Y] and [X-2,Y] are both coated).

10-1.(a) Change the definition of **depthsearch** to read

```
depthsearch(Start,Ans) :- depthsearch2(Start,[Start],Ans),
  print_reverse(Ans).
```

and include the definition

```
print_reverse([]) :- !.
print_reverse([X|L]) :- print_reverse(L), write(X), nl.
```

(b) Add to the front of all state lists the name of the last operator used to reach that state (or "none" for the starting state). (This assumes that states are represented by lists, as in most applications.) Then modify all **successor**, **goalreached**, **eval**, and **cost** predicate definitions to ignore the first term of the input state description, and modify all **successor** predicate definitions to set the first item of the output successor state to an operator name appropriate to the purpose of each such rule or fact.

(c) Replace the **member** in **not(member(Newstate,Statelist))** by **permutedmember**, defined as:

```
permutedmember(L1,L2) :- subset(L1,L2), subset(L2,L1), !.
subset([],[]) :- !.
subset([X|L1],L2) :- member(X,L2), !, subset(L1,L2).
```

where **member** is defined as before.

10-5. Refer to states as a pair of numbers representing fluid quantities in the size-5 and size-7 glasses respectively. So [3,4] means 3 units in the size-5 glass and 4 units in the size-7 glass.

(a) The states form a circle (excluding [5,7] which is only reachable when the goal amount is 5 or 7). Two solutions are always found for goal amounts 1, 2, 3, 4, and 6, one from going around the circle clockwise and one counterclockwise, both stopping at the first state satisfying the goal condition. (To see this behavior, it's important to avoid duplicating the same state at different places in the search graph.)

(b) The breadth-first state graph is augmented by extra links to the states [5,0], [0,7], and [5,7]. Links to the first occur from states [5,X] and [X,0] for certain X, to the second from [X,7] and [0,X] for certain other X, and to the third from [X,7] and [5,X] for certain other X. On such transitions, the search "starts over" in the opposite direction around the circle (so if movement was clockwise before the transition, afterwards it is counterclockwise). Only one such transition can occur in a solution path, except when the goal amount is 5 or 7.

(c) The eight rules make four pairs, for which the first rule of each pair refers principally to the first glass, the second rule to the second glass. The first pair empties one glass into another, the second pair fills one glass from another, the third pair fills a glass from the faucet, and the fourth pair empties a glass into the drain.

(d) Delete the second **goal_reached** fact.

10-7. Branch-and-bound search is just an A* search in which the evaluation function is always zero. So just to use the definition

```
eval(S,0).
```

with the A* program. That will work fine if the problem-independent file is always loaded before the problem-dependent file. But if you can't be sure, you can remove the **eval** predicate from the **add_state** predicate definition:

```
add_state(Newstate,Pathlist) :- not(agenda(Newstate,P,C)),
  not(oldagenda(Newstate,P2,C2)), cost([Newstate|Pathlist],Cnew),
  asserta(agenda(Newstate,[Newstate|Pathlist],Cnew)).
```

10-10. (a) Represent states as a list of pairs, where the first element of each pair is the name of a chemical and the second element is its amount in moles. Define **increase** and **remove** predicates that change the amount of a chemical in a state list. (Write "remove" to fail when it can't find the specified chemical in the state, so you don't need a **member** predicate.)

```
go(Ans) :- search([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],Ans).
```

```
goalreached(S) :- member([cacl2,N],S), N >= 1.


successor(S,[[C,Nsum]|S3]) :- remove(C,N1,S,S2),
  remove(C,N2,S2,S3), Nsum is N1 + N2.
successor(S,S6) :- remove(cl2,Ncl2,S,S2), remove(h2o,Nh2o,S2,S3),
  N is Ncl2 - Nh2o, N >= 0, increase(cl2,N,S3,S4),
  increase(hclo,Nh2o,S4,S5), increase(hcl,Nh2o,S5,S6).
successor(S,S6) :- remove(cl2,Ncl2,S,S2),
  remove(h2o,Nh2o,S2,S3), N is Nh2o - Ncl2, N >= 0,
  increase(h2o,N,S3,S4), increase(hclo,Ncl2,S4,S5),
  increase(hcl,Ncl2,S5,S6).
successor(S,S4) :- remove(h2o2,N,S,S2), member([mno2,X],S2),
  HalfN is N / 2, increase(h2o,N,S2,S3),
  increase(o2,HalfN,S3,S4).
successor(S,S6) :- remove(caco3,Ncaco3,S,S2),
  remove(hcl,Nhcl,S2,S3), HalfNhcl is Nhcl/2, Ncaco3 >= HalfNhcl,
  increase(cacl2,HalfNhcl,S3,S4), increase(co2,HalfNhcl,S4,S5),
  increase(h2o,HalfNhcl,S5,S6).
successor(S,S6) :- remove(caco3,Ncaco3,S,S2),
  remove(hcl,Nhcl,S2,S3), HalfNhcl is Nhcl/2, Ncaco3 < HalfNhcl,
  increase(cacl2,Ncaco3,S3,S4), increase(co2,Ncaco3,S4,S5),
  increase(h2o,Ncaco3,S5,S6).
successor(S,S5) :- remove(h2,Nh2,S,S2), remove(cl2,Ncl2,S2,S3),
  Nh2 >= Ncl2, N is Nh2-Ncl2, increase(h2,N,S3,S4),
  TwiceNcl2 is 2*Ncl2, increase(hcl,TwiceNcl2,S4,S5).
successor(S,S5) :- remove(h2,Nh2,S,S2), remove(cl2,Ncl2,S2,S3),
  Nh2 < Ncl2, N is Ncl2-Nh2, increase(cl2,N,S3,S4),
  TwiceNh2 is 2*Nh2, increase(hcl,TwiceNh2,S4,S5).
successor(S,S7) :- remove(hcl,Nhcl,S,S2),
  remove(mno2,Nmno2,S2,S3), QuarterNhcl is Nhcl/4,
  QuarterNhcl >= Nmno2, N is QuarterNhcl - Nmno2,
  increase(hcl,N,S3,S4), TwiceNmno2 is 2*Nmno2,
  increase(mncl2,Nmno2,S4,S5), increase(h2o,TwiceNmno2,S5,S6),
  increase(cl2,Nmno2,S6,S7).
successor(S,S7) :- remove(hcl,Nhcl,S,S2),
  remove(mno2,Nmno2,S2,S3), QuarterNhcl is Nhcl/4,
  QuarterNhcl < Nmno2, N is Nmno2 - QuarterNhcl,
  increase(mno2,N,S3,S4), HalfNhcl is Nhcl/2,
  increase(mncl2,QuarterNhcl,S4,S5),
  increase(h2o,HalfNhcl,S5,S6), increase(cl2,Nmno2,S6,S7).

increase(Chemical,Amount,S,[[Chemical,Newamount]|NewS]) :-
  failing_delete([Chemical,Oldamount],S,NewS),
  Newamount is Oldamount + Amount, !.
increase(Chemical,Amount,S,[[Chemical,Amount]|S]).


remove(Chemical,Amount,S,NewS) :-
  failing_delete([Chemical,Amount],S,NewS).


failing_delete(X,[X|L],L) :- !.
failing_delete(X,[Y|L],[Y|L2]) :- failing_delete(X,L,L2).
```

(b)

```
?- go(Answer).
(11) 11 Call: successor([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],_8)
(11) 11 Back to: successor([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],_8)
(11) 11 Back to: successor([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],_8)
(11) 11 Back to: successor([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],_8)
(11) 11 Exit: successor([[cl2,2],[mno2,1],[caco3,1],[h2o2,2]],
  [[o2,1],[h2o,2],[cl2,2],[mno2,1],[caco3,1]])
(63) 19 Call:
  successor([[o2,1],[h2o,2],[cl2,2],[mno2,1],[caco3,1]],_67)
(63) 19 Back to:
  successor([[o2,1],[h2o,2],[cl2,2],[mno2,1],[caco3,1]],_67)
(63) 19 Exit:
  successor([[o2,1],[h2o,2],[cl2,2],[mno2,1],[caco3,1]],
  [[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]])
(112) 28 Call:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],_123)
(112) 28 Back to:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],_123)
(112) 28 Back to:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],_123)
(112) 28 Back to:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],_123)
(112) 28 Back to:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],_123)
(112) 28 Exit:
  successor([[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],
  [caco3,1]],[[h2o,1],[co2,1],[cacl2,1],[hclo,2],[cl2,0],[o2,1],[mno2,1]])

Answer=[[[h2o,1],[co2,1],[cacl2,1],[hclo,2],[cl2,0],[o2,1],[mno2,1]],
       [[hcl,2],[hclo,2],[cl2,0],[o2,1],[mno2,1],[caco3,1]],
       [[o2,1],[h2o,2],[cl2,2],[mno2,1],[caco3,1]],
       [[cl2,2],[mno2,1],[caco3,1],[h2o2,2]]]

yes
```

(c) Cost could be the monetary cost of the chemicals needed for all the reactions so far, and the evaluation function could be the average cost of a chemical times the number of chemicals not yet in the reaction vessel but desired in the goal state. Or cost could be the danger of doing a sequence of reactions, and the evaluation function an estimate of the remaining danger based on the goal state.

(d) In the real world, all possible chemical reactions occur simultaneously. To model this for our rules, all that apply should work in parallel somehow. A little of each reaction could be done, time-sharing between reactions cyclically. (To be more precise, reactions should proceed at different speeds determined by *equilibrium constants* for each reaction, which can be modeled by "biased" time-sharing.).

10-12. (a) 16 states, if the goal state is on the final (leaf) level.

(b) 16 as well. With best-first search, the agenda can contain states at many levels. But if it does contain states at levels other than level 4 (the leaf level) then we could substitute in the successors of those states to get a bigger possible agenda. Therefore the largest agenda must contain only level 4 (leaf states), and there are 16 of those.

10-15. This is what the **foriterate** predicate definition does, so just query:

```
call(P,K) :- foriterate(call(P),K).
```

So just use the two-argument version of **call** instead of the usual one-argument form. If for instance you wanted to print out the third value of **X** to which the expression **a(X)** can be matched, you would query:

```
?- call(a(X),3), write(X).
```

11-1. These correspond to the **recommended** definitions. And rules for such household hints should go in front of default **recommended** rules for the same situations.

11-4. You need new predicates to describe the condition of the plate--let's say **on** and **off** with argument "plate". A starting state for the testing can be

```
[closed(case),closed(top),inside(batteries),defective(batteries),
  defective(light),unbroken(case),on(plate)]
```

with the goal

```
[ok(batteries),ok(light),closed(case),closed(top)]
```

Define new operators **remove_plate** and **replace_plate** this way:

```
recommended([off(plate)],remove_plate).
recommended([on(plate)],replace_plate).


precondition(remove_plate,[open(top),on(plate)]).
precondition(replace_plate,[open(top),off(plate)]).


deletepostcondition(remove_plate,[on(plate)]).
deletepostcondition(replace_plate,[off(plate)]).


addpostcondition(remove_plate,[off(plate)]).
addpostcondition(replace_plate,[on(plate)]).
```

To relate the new predicates and operators to the code written before, modify the preconditions of **replace_light** and **assemble_top**:

```
precondition(replace_light,[off(plate),open(top)]).
precondition(assemble_top,[on(plate),open(top)]).
```

11-6. Means-ends analysis is recursive, and recursion requires a stack when implemented in a computer. Laying out the parts in the order you remove them from the car is like keeping a stack, so you can "pop" them--that is, put them back into the car--in the reverse of the order they came out. It's not fair to say the parts are preconditions to operators, however; they're just props used in achieving preconditions, and the real preconditions are abstract notions.

11-7. Just write two different **precondition** facts, one for each set of preconditions. Put the set you think more likely to succeed first. If it can't be satisfied, backtracking will take the second set of preconditions. Nothing we've said requires a unique **precondition** rule for each operator.

12-3.(a) A value inherited must be for a slot that's filled in for the "purchase order" frame. Unfortunately, the frame is quite abstract, so most of the slots that you could imagine for it wouldn't have values (though those slots themselves will inherit downward via slot inheritance). One answer would be a **purpose** slot in a purchase order, always filled in with the value "purchasing_something". Another answer would be a **where_obtained** slot always filled with value "stockroom", meaning you can get blank purchase orders from the stockroom.

(b) It inherits from P2 too. (That doesn't necessarily mean that P2 has its units slot filled in--P2 could be inheriting that value itself. But that's still "inheritance from P2".)

12-4. Use frames to represent slots themselves. Have the name of each possible slot be the name of a slot-defining frame, in which we have a special slot named **value** to hold the value. Each slot-defining frame can have a slot for each qualifying slot of the original slot, with values filled in as appropriate. Every time you refer to the slot name, you can just inherit information from its slot-defining frame.

12-5. (a) The value "Floodge Manufacturing Inc." in the manufacturer's name slot, since it must be filled in similarly in any subtype of computer made by Floodge.

(b) The slot **ID-number**, since every computer must have an ID number, but each computer has a different value for it.

(c) Every Floodge computer has a CPU. So every personal computer made by Floodge also has a CPU. The CPU of a personal computer made by Floodge is an example of a part-kind inheritance, inference of a new frame. This new frame is different from the CPU frame for all Floodge computers, because for instance personal computers have slower CPUs.

12-9. The relationship of the Navy to military-organizations is **a_kind_of**, whereas the relationship of NPS to the Navy is more like **part_of**. Or you could say the Navy and military organizations are intensions, while NPS is more like an extension.

12-13. The problem of multiple inheritance for qualifying slots should be simpler than the original multiple inheritance problem, since they usually have few different values. So you can just assign priorities of inheritance direction as suggested for the user models example. In other words, any "qualifying-qualifying" slot for inheritance method will be filled with the same value, this direction-priority method.

13-1. We have two orderings to deal with: the order of the three rules, and the order of the two terms in the first rule. The second order involves an "and" and these should be ordered with the thing most likely to fail first. So **c(X)** should come before **b(X)** since 0.7 is less than 0.8. Since we're told that all the probabilities are independent, the probability of that first rule succeeding is just the independence-assumption "andcombine" of 0.7 and 0.8, or 0.56. This 0.56 is greater than 0.1, but less than 0.6. So since we should put the rules with greatest probability of success first, we should put the **e** rule first, then the rule with **b** and **c**, and then the rule with **d**.

13-4.(a) There's a mistake or bug somewhere, because the existence of the picture implies there must be some possible interpretation of it. Maybe our constraints are unreasonable, or maybe the characteristics of the regions of the picture were described incorrectly.

(b) This suggests an optical illusion, a situation for which the human eye could see two possible interpretations. Optical illusions can arise at random, particularly with pictures of geometrical figures and pictures in which objects have lined up in unusual ways.

(c) This suggests not enough constraints or too-weak constraints to solve the problem. That's not necessarily the programmer's fault, many real-world pictures can give this behavior, like those with few sharp boundaries.

13-6. We must first create possibility lists, using the three meal types given and any single-variable constraints. The first constraint is the only single-variable one. So we have the following possibilities (using "m1" for meal 1, "B1" for breakfast on day one, "L2" for lunch on day two, etc.)

```
B1: [m1]
L1: [m1,m2,m3]
D1: [m1,m2,m3]
B2: [m1,m2,m3]
L2: [m1,m2,m3]
D2: [m1,m2,m3]
```

Now we do the multivariable part of relaxation starting with the B2 variable. (The instructions also suggest that we start with B2 in the first step, but that doesn't make the difference to the result that it does for this step.) We retrieve the possible values of a variable, and check which of them satisfy constraints C2, C3, and C4. Here is one way:

1. Pick variable B2. Possibility m1 is impossible by constraint 4.

2. Pick variable L2 by the focus-of-attention heuristic. Then m1 is impossible by constraint 4.

3. Pick L1 by the same heuristic. It cannot be m1 by constraint 2. And it can't be m3 be constraint 4. So it must be m2. Possibility lists are now:

   ```
   B1: [m1]
   L1: [m2]
   D1: [m1,m2,m3]
   B2: [m2,m3]
   L2: [m2,m3]
   D2: [m1,m2,m3]
   ```

4. Pick variable L2. Then m2 can be eliminated by constraint 4 so the only remaining possibility is m3.

5. Pick B2. It cannot be m3 by constraint 3 so it must be m2.

6. Pick D1. It cannot be m2 by constraint 2, and it cannot be m3 by constraint 3 so it must be m1.

7. Pick D2. It cannot be m3 by constraint 3, and it cannot be m1 be constraint 1,so it must be m2.

The final possibilities have been reduced to one for each variable, as follows: B1=m1, L1=m2, D1=m1, B2=m2, L2=m3, D2=m2.

13-13. (a) 162 times, to get the answer **X=4, Y=3, Z=5**:

1. 3 backtracks from **n(Y)** to **n(X)**, since four **X** values must be tried.

2. 17 backtracks from **n(Z)** to **n(Y)**, |(3 * 5) + 3 - 1|. (The five values for **Y** must be cycled through three times, then the third value succeeds.)

3. 89 backtracks from **X>Y** to **n(Z)**, |(17 * 5) + 5 -1|. (Similar reasoning to the preceding.)

4. 29 backtracks from the last predicate expression **g(X,Y,Z)** to **not(f(X,X))**, | (6 * 5) - 1 |. (Six (X,Y) pairs are tried: (2,1), (3,1), (3,2), (4,1), (4,2), and (4,3).)

5. 24 backtracks from **not(f(X,X))** to **X>Y**, |29 - 5|. (The same as the previous answer except for the **X=2** and **Y=1** case ruled out.)

(b) The **g** predicate is the hardest to satisfy, and the > comparison is next hardest, so write the query as

```
g(X,Y,Z), X>Y, not(f(X,X)), n(X), n(Y), n(Z).
```

This will backtrack a mere three times: all three from the **X>Y** to **g(X,Y,Z)**. The same answer is found.

(c) 39 times. First figure out where to backtrack to. For predicate expression P, backtrack to the last predicate expression previous to P that binds a variable in P, or the immediately previous expression if no expression binds a variable in P.

--**n(Y)** should backtrack to **n(X)**
--**n(Z)** should backtrack to **n(Y)**
--**X>Y** should backtrack to **n(Y)**
--**not(f(X,X))** should backtrack to **n(X)**
--**g(X,Y,Z)** should backtrack to **n(Z)**

Let's now simulate dependency-based backtracking:

1. We match **X=1**, **Y=1**, and **Z=1**.

2. **X>Y** fails; we go back to **n(Y)**. (1 backtrack)

3. We pick **Y=2**, then return to **X>Y**, which fails again. Back to **n(Y)**. (1 backtrack)

4. We pick **Y=3**, then return to **X>Y**, which fails again. Back to **n(Y)**. (1 backtrack)

5. This repeats until **Y=5**, whereon we fail **n(Y)**. We return to **n(X)** and pick **X=2**. (3 backtracks)

6. We now pick **Y=1**, and **X>Y** succeeds.

7. But **f(X,X)** succeeds, so **not(f(X,X))** fails, so we backtrack to **n(X)**. (1 backtrack)

8. We now take **X=3**. We skip **n(Y)** and **n(Z)** since they could not have caused the last failure.

9. **X>Y** holds, as does **not(f(X,X))**. But **g(X,Y,Z)** fails, so we backtrack to **n(Z)**. (1 backtrack)

10. **Z=2** is chosen, we return to **g(X,Y,Z)**, but that doesn't work either. We return to **n(Z)**. (1

backtrack)

11. We alternate repeatedly between **n(Z)** and **g(X,Y,Z)** until **n(Z)** fails, whereupon we return to **n(Y)**. (4 backtracks)

12. We pick **Y=2**, **Z=1**. **X>Y** succeeds, but **g(X,Y,Z)** fails. Return to **n(Z)**. (1 backtrack)

13. None of the other values for **Z** will make **g(X,Y,Z)** succeed. Fail **n(Z)**. (5 backtracks)

14. We pick **Y=3**. **X>Y** fails. Similarly for **Y=4** and **Y=5**. So **n(Y)** fails. (4 backtracks)

15. We pick **X=4**, **Y=1**, **Z=1**. **X>Y** and **not(f(X,X))** succeed, but **g(X,Y,Z)** fails. (1 backtrack)

16. **Z=2** is tried and fails **g(X,Y,Z)**, followed by **Z=3**, **Z=4**, **Z=5**. So **n(Z)** fails. (5 backtracks)

17. **Y=2** is tried and fails **g(X,Y,Z)** after all **Z** are tried. (6 backtracks)

18. **Y=3** is tried. **Z=1**, **Z=2**, **Z=3**, and **Z=4** are tried and fail; **Z=5** succeeds. (4 backtracks)

(d) First create possibility lists for **X**, **Y**, and **Z**, using the **n** and **f** predicate expressions:

```
X: [1,3,4,5]
Y: [1,2,3,4,5]
Z: [1,2,3,4,5]
```

The only multivariable constraints are **X>Y** and **g(X,Y,Z)**.

1. Pick variable **X** first since it has the fewest possibilities. **X=1** is not possible with **X>Y**. And **X=5** is not possible with **g(X,Y,Z)**. So **X=3** or **X=4** only. (Note it's not fair to say that **X** cannot be 3 now because that could only be shown by considering the two constraints together, **X>Y** and **g(X,Y,Z)**. Pure relaxation throws away bindings from one constraint when considering another constraint.)

2. Pick variable **Y**. **Y=3** is the only way to satisfy constraint **g(X,Y,Z)**.

3. Return to variable **X**. Now by **X>Y**, **X** cannot be 3, so it must be 4.

4. Go to variable **Z**, the only remaining variable with multiple possibilities. It must be 5 by **g(X,Y,Z)**.

13-14. (a) Whenever a unique value is chosen for a variable the **unique_values** predicate can exploit that new value. So **unique_values** becomes harder to satisfy, and it must do more backtracking. This slows the program down. When we start with two values for e, things aren't slowed down until o is found to be 0, at which point there are fewer other unassigned variables to worry about, so the slowdown is less dramatic.

(b) Delete the definition of **unique_values**, and delete its call from the definition of the **satisfiable** predicate. It isn't necessary.

(c) The predicate **duplication** catches most of the bad assignments of variables to values, but **unique_values** catches a few cases that **duplication** misses. For instance, if each of variables A, B, and C have possibilities of 4 or 5, and each variable has a distinct value, there's no way that the conditions can be met; **unique_values** will catch this.

13-18.(a) A state is any set of possibility lists for the variables, where possibility lists are subsets of the original possibility lists in the starting state, plus a list of which variables are "active". It can be represented as a list of lists.

(b) Depth-first search. It keeps moving ahead, never returning to a previous state. There's no point in returning to a previous state because you keep learning useful things as you proceed, things that can potentially help in future study.

(c) No, because you can't predict what intermediate states you'll get into.

(d) The branching factor always stays the same or decreases by 1. A state transition means either elimination of a single possibility for a variable or no elimination.

(e) If the sum of the number of possibilities on all initial possibility lists is N, then there are |2 sup N| possibility list configurations obtainable by crossing out 0 to N possibilities. But we haven't included the active variables. There are |2 sup V| configurations of these for each possibility-list configuration, so the total number of states is the product of these two numbers, or |2 sup {N + V}|. (Some of those states have zero possibilities for some variables, but such states are possible when the input has errors.)

14-1. **X** binds to 5, and **Y** binds to **X**, so **Y** is bound to 5 too. So the **possible**s cancel, giving as resolvent

```
state(3,A,5); state(3,6,5).
```

But the second predicate expression is covered by the first, so we can rewrite this as

```
state(3,A,5).
```

That is the only resolution possible.

14-2. (a) Suppose:

> **t** represents your having travel money;
> **d** represents your department chairman approval;
> **s** represents sponsor approval;
> **b** represents the boss of your department chairman approving;
> **f** represents discretionary funds being available.

Then we can write the initial rules as

```
t :- d, s.
t :- d, b, f.
```

(b)

```
t; not(d); not(s).
t; not(d); not(b); not(f).
```

(c)

```
(d,s);(d,b,f) :- t.
```

or equivalently:

```
d,(s;(b,f)) :- t.
```

(d) The second preceding can be translated to

```
(d, (s;(b,f))); not(t).
```

Using the distributive law of "or" over "and":

```
(d; not(t)), (s; (b,f); not(t)).

d; not(t).
s; (b,f); not(t).
```

The first of those two is a clause, but we still need to use the distributive law of "or" over "and" on the second. Finally, we have these three clauses:

```
d; not(t).
s; b; not(t).
s; f; not(t).
```

(e) Just resolve the fact **t** with the clauses in part (d), giving:

```
d.
s; b.
s; f.
```

Since we did all possible resolutions involving all given information, those are all the clauses derivable.

(f) Just resolve the clause **not(t)** with the clauses in part (b), giving:

```
not(d); not(s).
not(d); not(b); not(f).
```

14-5. Resolvent clauses are only guaranteed to *include* the truth, not give a precise specification of the truth as the "and" of the two resolvents would. The statement "b or c" does include all the cross-hatched region in the figure, the "and" of the input clauses, which is all that is required.

14-9. (a) The number of possible pairs of clauses or |N (N - 1) / 2|. (Note that resolution is a commutative operation, hence the division by 2.)

(b) Each successful resolution increases by one the number of clauses, so after K resolutions there will |N + K| clauses and |(N + K) (N + K - 1) / 2| pairs. K of these pairs will have already been used, so an upper bound on the branching factor is |((N + K) (N + K - 1) / 2) - K |.

15-1. The real test is not the opinion of any human being, but whether the car works correctly when the supposed problem is fixed. So the gold standard is the minimum change necessary to get the car working right again. (It's the "minimum" because you can always get any faulty car working again by simply replacing every part with a nonfaulty part.)

15-3. In the confusion matrix, examine the cell pairs symmetric about the diagonal running northwest-southeast (but not including the cells of the diagonal itself.) Look for any pair where (1) one number is much larger than the other, and (2) the cell with the larger number is in a row corresponding to diagnosis rules that precede the rules that correspond to the row for the other diagnosis; such a pair suggests the order of the rows should be reversed. This clue is further strengthened if the other cells in the same two rows and two columns are small, except for the diagonal terms.

15-6. (a) Not a good idea by the third and fourth overall criteria of Section 15.10.

(b) Not a good idea by the first, second and sixth criteria.

(c) Not a good idea by the second and third criteria.

(d) Not a good idea by the third, fourth, and fifth criteria.

M-1. (a) Yes, backtracking on failure is the whole point.

(b) No, the whole point of forward chaining is to keep moving forward from facts to new conclusions. Choosing new facts and new rules is better done by iteration, not backtracking.

(c) No, since you store things on an agenda and just pick the best thing on the agenda at each point, a form of iteration. The agenda item selected might be something in a far away and/or deeper part of the search lattice than the last agenda item, so you're not necessarily "going back" to anything.

(d) Yes, dependency-based backtracking is included in this.

(e) Yes, because you might have several directions (links) you could follow to find a value for some slot. If one path doesn't pan out, you should backtrack and try another.

(f) No, because it's just an example of forward chaining.

M-3. Horn Clause: f. (the definition)

generate-and-test: i. (variable **X** is generated, then tested)

constraint: m. (equivalent terminology)

forward chaining: o. (that's the whole idea)

default: b. (the definition)

depth-first control structure: q. (essential even for those depth-first activities that aren't searches)

intension: p. (the definition)

difference table: h. (the main purpose)

near miss: n.

Skolem Function: j.

caching: a. (the definition)

dependency: r. (the definition for predicate expressions in the same query)

multiple inheritance: c. (the chief difficulty)

decision lattice: g.

Bayes' Rule: d. (it can be proved from probability theory)

heuristic: e. (a famous example)

hierarchical reasoning: l. (hierarchy is in the levels of recursion)

agenda: k. (the definition)

M-4.(a) 1, and-or-not lattice. This sounds like a pretty straightforward application for expert systems; few variables will be needed. So the lattice will be feasible, and will be a simple (and reasonably efficient) implementation method. You could build the lattice with logic gates on a VLSI chip that could be put into a small hand-held device. Dependency-based backtracking is probably too fancy a technique for this problem; you must wait for the user to answer questions, so the extra speed of dependency-based backtracking will usually be wasted. You shouldn't need to backtrack anyway since this sounds good for forward chaining; there are few facts (the problem mentions "simple questions"), and probably low "fanout" from facts (multiple references to the same fact in different rules should be rare). A breadth-first resolution strategy is a very slow method of reasoning only appropriate when you want to be sure to reach every valid conclusion, including conclusions with "or"s; here we only want a single conclusion about which tax form to use. Finally, there's no such thing as "confusion-matrix conflict resolution"; conflict resolution is how you decide what to do next in a rule-based system, and confusion matrices are statistical summaries useful in analyzing experiments with system performance.

(b) 4, concurrency. You can only ask questions one at a time, and there's probably low fanout of facts to rules, so there isn't much opportunity for concurrency. (Things are quite different for expert systems with real-time inputs, like the "smart house" in Exercise 6-12.) Virtual facts would be essential if you used backward chaining--you don't want to ask the same question twice. Inference is a general term for all reasoning, and this system must reason. A depth-first control structure is good for many expert systems including this one, regardless of whether they do backward, forward, or hybrid chaining, because it's easy to implement.

(c) 2, **repeat**. There's not much reason to repeat anything here; after all, an and-or-not lattice sound good, and that can't accommodate iteration very well. The **asserta** or **assertz** would be necessary for virtual facts. The **write** would be necessary to ask questions of the user. The **>** could be used in making the needed monetary-amount comparisons.

M-5. (a) 1, best-first search. Deviation from a straight line is a good (and necessary) evaluation function; we clearly don't want to lurch a lot, or we couldn't walk very fast. But there's no obvious cost function; speed is not important, according to the problem statement. And there's no such thing as "weight-first search".

(b) 2, caching only. Caching is helpful for reexamining the same ground as robot moves forward. Means-ends analysis isn't helpful because there's only one operator, foot placement. So all the tricks that means-ends analysis uses to select operators based on logical conditions don't have anything to work with. Furthermore, the problem is quantitative: it depends on numeric things like the coordinates of a square, the positions of the other legs, the center of gravity of the robot, the direction of motion of the robot, and so on. Means-ends analysis is intended for non-quantitative problems.

(c) 2, **assertz**. No reason to use queues here, about the only reason to ever use an **assertz**. The **is** is always useful with numeric calculations, and the cut and **repeat** predicates could help make search efficient, just as they did for the best-first program in Section 10.9.

Go to book index