Contents

Chapter 11

Defeasible Prolog

11.1. NONMONOTONIC REASONING AND PROLOG

If our reasoning is monotonic, the set of conclusions we draw from the information we have only gets larger as we get more and more information. Once we reach a conclusion, no additional information will cause us to reject it. When our reasoning is nonmonotonic, we may reject an earlier conclusion on the basis of new information.

Human reasoning is notoriously nonmonotonic. We make plans based on what we expect to happen, but we constantly revise our expectations, and our plans, as events unfold. For example, we make a date with a friend to meet for lunch on Friday. Then on Wednesday we learn that our friend had to leave town suddenly due to a family emergency. We revise our expectations and no longer believe our friend will keep his luncheon date.

The Prolog inference engine is nonmonotonic because of the way it handles negation. As we know, the query `\+ Goal.` succeeds in Prolog if and only if the query `?- Goal.` fails. Consider this two-clause knowledge base:

```
flies(X) :- bird(X), \+ penguin(X).
bird(chilly).
```

The first clause says that something flies if it is a bird and we cannot prove that it is a penguin. The second says that Chilly is a bird. With this knowledge base, the query

```
?- flies(chilly).
```

succeeds. But if we add the fact `penguin(chilly)`, the same query will then fail because we are no longer unable to prove that Chilly is a penguin. With more information, Prolog rejects a goal it earlier accepted.

People often accept and use rules of thumb like "Birds fly," principles that they realize have exceptions. We use these rules, perhaps even unconsciously, unless we have reason to believe we are dealing with a case where the rule does not apply. We do not usually list all of the exceptions as part of the rule, and we realize that there may be many exceptions we don't know about. However, we can sometimes represent such rules in Prolog using negation-as-failure to list the exceptions. We shall see that this method will not work in all cases.

11.2. NEW SYNTAX FOR DEFEASIBLE REASONING

Generalizations like "Birds fly" and "Matches burn when they are struck" apply to usual, typical, or normal cases. Unusual, atypical, or abnormal cases are exceptions to the rule. We will say one of these generalizations is *defeated* when we have information that tells us we are dealing with an exception to the rule. Because they can be defeated, we will call these generalizations *defeasible rules.*

Although Prolog can perform some kinds of nonmonotonic reasoning, Prolog rules are not defeasible. The rule about birds from the last section is not a defeasible rule because it is not affected by anything not stated in the rule itself. Prolog concludes that something flies if it satisfies the conditions of the rule no matter what else is in the knowledge base. A defeasible rule, on the other hand, is a rule that cannot be applied to some cases even though those cases satisfy its conditions, because some knowledge elsewhere in the knowledge base blocks it from applying.

This might lead one to think that a defeasible rule is simply an inexplicit Prolog rule. Not so. Some instances of defeasible reasoning cannot be reproduced in ordinary Prolog. As an example, suppose Farmer, Hunter and Baker are all running for the same elective office. Someone might say, "I presume Farmer will be elected, but if he isn't, then Hunter will be." How could we represent this assertion in Prolog?

We might represent the presumption that Farmer will win by the fact

```
will_be_elected(farmer).
```

and we might represent the expectation that Hunter will win if Farmer doesn't by the rule

```
will_be_elected(hunter) :- \+ will_be_elected(farmer).
```

But this doesn't really capture the opinions expressed. First, the original statement was a presumption which might be defeated, but there is no way we can avoid the conclusion that Farmer will win the election once we have the fact about Farmer in our database. Second, there isn't much point in having the rule about Hunter at all since its condition can never be satisfied so long as we have the fact about Farmer.

Perhaps our mistake is in the way we have represented the presumption. Instead of translating it into a Prolog fact, perhaps it should be a rule. But what rule? What is the condition for this rule? Maybe it should be

```
will_be_elected(farmer) :- \+ will_be_elected(hunter).
```

Now the two rules about Farmer and Hunter have exactly the same form and the same strength, but our presumption was that Farmer will be elected. Furthermore, the two rules taken together will cause Prolog to loop endlessly because each of them always calls the other when Prolog is given the query

```
?- will_be_elected(X).
```

Let's add some detail to our example. Suppose our political prognosticator goes on to say, "Of course, Farmer won't win the election if Baker withdraws and throws her support to Hunter." How shall we represent this in Prolog? The obvious way to do it is

```
\+ will_be_elected(farmer) :- withdraws(baker),
                               supports(baker,hunter).
```

But this is also obviously wrong since it is not a well-formed Prolog clause. The built-in predicate `\+` cannot occur in the head of a Prolog clause. Typically, a Prolog interpreter will understand this rule as an attempt to redefine the predicate `\+`. Some interpreters will display an error message saying that this is not allowed.

What we need is a new way to represent defeasible rules and presumptions and an inference engine that knows how to use them. We also need a negation operator that is different from negation-as-failure so we can represent rules that tell us when something is positively not the case rather than just that we cannot *prove* that it is the case. These negative rules are needed to tell us when we have an exception to a defeasible rule, but they are desirable in their own right as well.

It's easy to represent defeasible rules. We arbitrarily pick a new operator to join the head and the body of the rule. We will use the operator `:=/2`. With this operator, we represent the defeasible rule that birds fly as

```
flies(X) := bird(X).
```

We will also introduce the negation operator `neg/1`. Unlike `\+`, `neg` can appear in a the head of a rule. Thus, we represent the claim that Farmer will not be elected if Baker withdraws from the election and throws her support to Hunter as

```
neg will_be_elected(farmer) :- withdraws(baker),
                               supports(baker,hunter).
```

Now comes the challenging part: building an inference engine that knows how to use these rules properly. Before we can do this, we must investigate further how reasoning with defeasible rules works.

Representing presumptions requires a bit more thought than representing ordinary defeasible rules. A presumption looks something like a defeasible fact. In Prolog the fact

```
yellow(sun).
```

is equivalent to the rule

```
yellow(sun) :- true.
```

The same relationship should hold between presumptions and defeasible rules. We will represent the presumption that Farmer will be elected by the defeasible rule

```
will_be_elected(farmer) := true.
```

This allows us to distinguish between presumptions and facts.

There is one other kind of statement that is important in understanding our use of defeasible rules. These are statements like "Farmer might not win if there is a poor voter turn-out." How do these *might* rules work? If there is a poor voter turn-out, does it follow that Farmer does not win? No. The rule is too weak to support this conclusion. But if there is a poor voter turn-out, our presumption that Farmer will win gets defeated. Then the inference engine should conclude neither that Farmer will win nor that Farmer will not win. The sole purpose of these 'might' rules is to interfere with conclusions that we might otherwise draw. They do not themselves allow us to draw any new conclusions.

This points out that there are two different ways that a defeasible rule can be defeated. First, we can have an ordinary Prolog rule or another defeasible rule that supports the opposite conclusion. When this happen, we say that the second rule *rebuts* the first rule. But that's not what is happening with these rules that use the word 'might'. They do not support the opposite conclusion or, for that matter, *any* conclusion. They simply point out a situation in which the rules they defeat might not apply. Rather than rebut, they *undercut* the rules they attack. So a defeasible rule might be either *rebutted* or *undercut*. Of course, two defeasible rules with opposite conclusions offer rebuttals for each other. It remains to be seen under what circumstances these rebuttals are successful. The other kind of defeater, the *might* defeater that does not itself support any conclusion, we will call an *undercutting* defeater or simply a *defeater*.

Of course, we will need some way to represent defeaters as well as defeasible rules and presumptions. We will use a new operator `:^` to connect the head and the body of a defeater. We represent the defeater in our election example as

```
neg will_be_elected(farmer) :^ voter_turn_out_is_poor.
```

We want an inference engine that recognizes defeasible rules, presumptions, defeaters and negations using `neg`, and knows how to use them. To make the work of the defeasible inference engine easier, we will place some restrictions on the new kinds of rules we are introducing. We will not allow the use of disjunctions (`;`), cuts (`!`), or negation-as-failure (`\+`) in any of these rules. Then we don't have to tell our inference engine what to do when it finds these in a rule. By restricting the use of cuts and negation-as-failure, we restrict procedural control of the search for solutions and make it difficult to adopt a procedural style while using defeasible rules and defeaters. We assume that the procedural parts of our programs will still be written in ordinary Prolog. The use of disjunction in a defeasible reasoning system, on the other hand, raises special problems that we will not consider here.

11.3. STRICT RULES

Note that not all rules used in reasoning are defeasible. For example, the rule 'Penguins are birds' can have no exceptions. This is because 'bird' is part of the definition of 'penguin'. In general, rules which say that members of one *natural kind* are also members of another are nondefeasible or *strict* rules. Other examples include 'Gold is a metal' and 'Bats are mammals'. We won't go into exactly what it means to say that something is a natural kind, and it isn't quite right to say that 'bird' is part of the definition of the word 'penguin', but the general idea should be clear enough.

Other examples of a strict rules are 'Squares are rectangles' and 'Marxists are communists'. These rules are strict quite literally by definition. Of course, Marxists and communists aren't natural kinds (nor are squares and rectangles on most accounts), but that's why the relations between the terms are literally definitions. These are groups that man designed rather than groups found naturally.

Some other strict rules involve spatial and temporal regularities, e.g., 'Every living person born in Atlanta was born in Georgia' and 'Every living person born before 1950 is more than forty years old today.'

Besides the kinds of rules listed here, most other rules are defeasible. Even strong causal claims like 'Litmus paper turns red when it is dipped in acid' should be treated as defeasible, as should strong regularities like 'Members of the John Birch Society are conservative.' No causal generality can ever be completely confirmed, and of course a group like the John Birch Society might be infiltrated by the A.C.L.U. or some other liberal group.

We will represent strict rules as ordinary Prolog rules. Whenever the antecedent condition of a strict rule is derivable, so is its consequent. But suppose the condition of a strict rule is only *defeasibly* derivable? Suppose, for example, that we have evidence that some animal is a bat and we also have evidence that it is a bird, but our evidence for neither claim is conclusive. Then the conditions for the two strict rules

```
mammal(X) :- bat(X).
neg mammal(X) :- bird(X).
```

are both defeasibly derivable in the case of our mystery animal. What should we conclude? If our strict rules are correct, then some of our evidence must be incorrect. But we can't tell whether it is our evidence that the animal is a bat or our evidence that it is a bat that is flawed. We do know, though, that the animal can't both *be* a mammal and *not be* a mammal. In this situation, we should refrain from drawing any conclusion about whether the animal is a mammal. We may still be in the peculiar situation of concluding at least tentatively that it is both a bat and a bird, but we do a certain amount of damage control by stopping short of the blatant contradiction that it both is and is not a mammal.

There are two lessons to be learned from this example. First, we have to distinguish between what is strictly derivable and what is only defeasibly derivable. A conclusion is strictly derivable if it is derivable from facts and strict rules alone. We will say that any conclusion that is strictly derivable is also (at least) defeasibly

Figure 11.1 The Pennsylvania Dutch Example.

derivable, but other conclusions that depend on presumptions and defeasible rules are also defeasibly derivable. Second, to limit the damage when we have conflicting evidence for competing strict rules, we will allow strict rules to be rebutted by other strict rules in at least those cases where the condition for the rebutted strict rule is only defeasibly derivable.

What about cases where strict rules compete with defeasible rules? Let's consider another example. Pennsylvania Dutch is a dialect of German. Ordinarily, native speakers of Pennsylvania Dutch are born in Pennsylvania. Of course, anyone born in Pennsylvania was born in the United States. But typically, native speakers of a German dialect are not born in the United States. Hans speaks Pennsylvania Dutch. We represent all this information as follows;

```
native_speaker(X,german_dialect) :- native_speaker(X,pa_dutch).
born(X,pennsylvania) := native_speaker(X,pa_dutch).
born(X,usa) :- born(X,pennsylvania).
neg born(X,usa) := native_speaker(X,german_dialect).
native_speaker(hans,pa_dutch).
```

We can also represent this information in a graph, as we have done in Figure 11.3 using $\rightarrow$ for strict rules and $\Rightarrow$ for defeasible rules. We use bars through the arrows to show when the negation of the term at the head of the arrow is indicated by the rule.

First, we notice that Hans must definitely be a native speaker of a German dialect. Second, we have evidence that Hans was born in Pennsylvania and no evidence to the contrary. So we conclude defeasibly that Hans was in fact born in

Pennsylvania. But now we have conflicting evidence in the strict rule saying Hans was born in the United States and the defeasible rule saying that he was not. Even though the condition of the strict rule is only defeasibly derivable while the condition of the defeasible rule is strictly derivable, the strict rule is superior. So we conclude defeasibly that Hans was born in the United States. Another way to put this is to strengthen our previous observation: *a strict rule can only be defeated if its condition is only defeasibly derivable, and then only by a fact or by another strict rule that rebuts it.*

11.4. INCOMPATIBLE CONCLUSIONS

We call simple Prolog clauses that do not contain the functor `:-` *atomic* clauses or simply *atoms*. We call atoms and negations of atoms formed using the predicate `neg` *literals*. Where `Atom` is a Prolog atom, we say that `Atom` and `neg Atom` are *complements* of each other. As we remarked earlier, both positive and negative literals may occur as the heads of rules in d-Prolog. And rules whose heads are complements of each other may rebut each other.

This is not the only way that two rules might conflict. It is impossible for one and the same animal to be both a bird and a mammal, and it is impossible for one and the same man to be both a Marxist and a capitalist. (We are not talking about *membership* is some Marxist group, now, but actually *being* a Marxist.) But of course `bird(animal)` and `mammal(animal)` are not complements of each other, nor are `marxist(man)` and `capitalist(man)` Yet we would surely want to say that evidence that an animal is a bird conflicts with any evidence that it is a mammal, and evidence that a man is a Marxist conflicts with any evidence that he is a capitalist. How can we indicate this?

Consider the example of Ping who was born in the People's Republic of China but who emigrated to this country where she now owns a restaurant. Is Ping a capitalist? We might represent the situation as follows:

```
capitalist(X) := owns(X,restaurant).
neg capitalist(X) :- marxist(X).
marxist(X) := born(X,prc).
neg marxist(X) :- capitalist(X).
owns(ping,restaurant).
born(ping,prc).
```

This set of d-Prolog rules and facts certainly captures the incompatibility of being both a Marxist and a capitalist. But consider what would happen if we asked whether we could defeasibly derive that Ping is a capitalist. Since she owns a restaurant, we have evidence that she is a capitalist. But the rule we would use is defeasible; so we must make sure that it is not defeated. We find the competing rule that says she is not a capitalist if she is a Marxist. This will defeat our rule unless we can show that there is no way to derive that Ping is a Marxist, i.e., that the rule is not *satisfied*. But since Ping was born in the People's Republic of China, we have evidence that she is a Marxist. Of course, this also depends on a defeasible rule; so we will still be okay if we can show that this rule is defeated. We have a competitor in the strict rule that says that capitalists are not Marxists. So now we need to show is that Ping is a capitalist in order to show that Ping is a capitalist! We are in a loop and have no way to get out.

Our solution to this problem is similar to the solution to the problem of using biconditionals discussed in Chapter 6. We will introduce a predicate `incompatible/2` which will tell d-Prolog when two clauses that are not complements of each other are nevertheless incompatible. So in our examples, we would use the clauses

```
incompatible(bird(X),mammal(X)).
```

and

```
incompatible(marxist(X),capitalist(X)).
```

When we implement our defeasible logic, we must tell the d-Prolog inference engine how to use this information without looping.

We say two clauses `Clause1` and `Clause2` are *contraries* in a d-Prolog database if one is the complement of the other or if either of the two clauses

```
incompatible(Clause1,Clause2).
```

or

```
incompatible(Clause2,Clause1).
```

is in the database. We say that two rules *compete* or *conflict* with each other and that they are *competitors* if their heads are contraries. These definitions are implemented in Prolog as follows:

```
contrary(Clause1,Clause2) :-
        incompatible(Clause1,Clause2).

contrary(Clause1,Clause2) :-
        incompatible(Clause2,Clause1).

contrary(Clause1,Clause2) :-
        comp(Clause1,Clause2).

comp(neg Atom,Atom) :-
        !.

comp(Atom,neg Atom).
```

11.5. SUPERIORITY OF RULES

Let's consider the Chinese restaurant example again. We can represent this information by the graph in Figure 11.5. Here we have two defeasible rules, each rebutting the other. But, intuitively, what would we conclude about someone who emigrated from the People's Republic of China to the United States and opened a restaurant?

Figure 11.2 The Chinese Restaurant Example.

We would conclude that this person is probably a capitalist. We would consider the evidence of running a business for profit superior to the evidence of place of birth. So our capitalist rule is superior to our Marxist rule.

We need a way to tell d-Prolog that one rule is superior to another. For this, we introduce the predicate `sup/2`. In our example, we add the clause

```
sup((capitalist(X) := owns(X,restaurant)),
    (marxist(X) := born(X,prc))).
```

We say Rule1 is *inferior* to Rule2 if Rule2 is superior to Rule1. The importance of this notion to our defeasible logic is that a defeasible rule cannot be rebutted by an inferior defeasible rule. In our example, the Marxist rule is rebutted by the capitalist rule, but the capitalist rule is not rebutted by the Marxist rule. So we conclude that `capitalist(ping)` is defeasibly derivable.

Similarly, a defeasible rule cannot be undercut by an inferior defeater. For example, people normally can walk. But if someone is very young, he or she might not yet walk.

```
walks(X) := human(X).
neg walks(X) :^ infant(X).
```

But suppose we incorporate into this same database information about animals other than humans. Antelope, for example, can walk within hours of birth. We would not want our defeater for young creatures to undercut the rule for deer.

```
walks(X) := deer(X).
sup((walks(X) := deer(X)),(neg walks(X) :^ infant(X))).
```

Figure 11.3 A 'Tweety Triangle'.

So the rule about normal deer locomotion is superior to the defeater about infant locomotion and is not undercut by it. But the rule about human locomotion is not superior to the defeater and is therefore undercut by it.

11.6. SPECIFICITY

Sometimes we can tell that one rule is superior to another without being told. Consider the following example.

```
flies(X) := bird(X).
neg flies(X) := penguin(X).
bird(X) :- penguin(X).
penguin(opus).
```

Is Opus a bird? Definitely. Does Opus fly? Apparently, not. An example like this has been around for some time involving a bird named Tweety. The corresponding graph is shown in Figure 11.3. Because of the shape of the graph, this kind of example is often called a 'Tweety Triangle'.

Remember that defeasible rules tell us about the normal or typical cases. Typical birds fly. But a penguin is a special kind of bird. Because of our strict rule, we immediately know that something is a bird as soon as we know that it is a penguin. The converse, of course, is not true: many birds are not penguins. So any rule for penguins is more specific than any rule for birds. Penguin rules take into account more information than do bird rules. When one rule is more specific than another, the first rule is superior to the second.

How can we tell that penguin rules are more specific than bird rules? By noticing that the strict rule

```
bird(X) :- penguin(X).
```

is in our database. But the connection between birds and penguins might have been less direct and it might have involved defeasible rules as well as strict rules. Here is another example.

```
conservative(X) := affiliate(X,chamber_of_commerce).
neg conservative(X) := theatrical_agent(X).
busnessperson(X) := theatrical_agent(X).
affiliate(X,chamber_of_commerce) := businessperson(X).
theatrical_agent(Taylor).
```

In this case, the rule for nonconservativism is more specific than and therefore superior to the rule for conservativism. That's because we have a chain of rules linking the condition for nonconservativism (being a theatrical agent) to the condition for conservativism (being a business person). Put another way, we can *defeasibly derive* that Taylor is an affiliate of the Chamber of Commerce from her activities as a theatrical agent, but we cannot defeasibly derive how she makes her living from her membership in the Chamber of Commerce.

It is not enough to establish specificity that we have rules linking the condition of one competing rule to the condition of another. If those links are defeasible, then they must be undefeated or specificity cannot be inferred. As an example of what can go wrong, assume that college students are normally adults and that normally adults are employed, but that college students normally are not employed. Furthermore, employed persons typically are self-supporting while college students typically are not self-supporting. Finally, we stipulate that Jane is a college student who is employed. Does Jane support herself? The rules and facts relevant to this example are:

```
adult(X) := college_student(X).
neg employed(X) := college_student(X).
neg self_supporting(X) := college_student(X).
employed(X) := adult(X).
self_supporting(X) := employed(X).
college_student(jane).
employed(jane).
```

The corresponding graphic representation is shown in Figure 11.6. We have two conflicting rules, one for college students and another for employed persons. Is either more specific than the other? Since college students are normally adults and adults are normally employed, it might appear at first glance that the rule for college students is more specific than the rule for employed persons. However, rules about college students are not really more specific than rules about employed persons because the link from college students to employed persons is itself defeated. Or to put it differently, we cannot defeasibly derive that Jane is employed from her status as a college student because we have a rule that says college students normally are

Figure 11.4 The College Student Example.

not employed. This rule is more specific than the rule for adults which occurs in the path from college students to employed persons. So neither rule about self-support is more specific, and each rule is rebutted by the other.

This points out a crucial fact about our defeasible reasoning. Most of the time, we are using everything we know as the basis for our conclusions. But sometimes, when we are trying to figure out which rules are more specific, we use only the information in the conditions of the rules together with other rules that may link these. When we do this, we ignore any facts or presumptions we may have available and rely only on the conditions of the two rules being compared and the other rules available to us. For example, part of the information we have in the college student example is the fact that Jane is employed. If we are allowed to use this, then we can infer that Jane is employed from any other information at all. But that certainly doesn't mean that it follows even defeasibly from Jane's being a college student that she is employed.

In summary, then, at any given time we may be reasoning from part of our information. We will call these different sets of information *knowledge bases*. Normally we will be reasoning from the entire d-Prolog database. We will call this the *root* knowledge base. When we test to see whether one rule is more specific than another, we limit ourselves to the clauses in the condition of one of the two rules being compared together with all the strict rules, defeasible rules, and defeaters in the root knowledge base. We leave out any facts or presumptions in the root knowledge base during this part of our reasoning. Remembering that the strict rules, defeasible rules, and defeaters from the root knowledge base get added in, we can think of each condition of a rule as itself determining a knowledge base. It will be important in designing our inference engine to remember that we must always keep track of the knowledge base in use at each stage of reasoning.

```
% DPROLOG.PL
%
% A defeasible inference engine for Prolog.
%

:- nl, nl,
   write('d-Prolog - A Defeasible Extension of Prolog'), nl, nl,
   write('"preempt" toggles preemption of defeaters.'), nl,
   nl, nl,
   op(1100,fx,@),
   op(900,fx,neg ),
   op(1100,xfy,:=),
   op(1100,xfy,:^).

:- dynamic (neg)/1, (:=)/2, (:^)/2.
:- multifile (neg)/1, (:=)/2, (:^)/2.

%
% @(+Goal)
%   Succeeds if Goal is defeasibly derivable from
%   the d-Prolog knowledge base.
%

@ Goal :- def_der(root,Goal).

%
% strict_der(+KB,+Goal)
%   Succeeds if Goal is derivable from the ordinary Prolog
%   facts and rules in the d-Prolog knowledge base.
%

strict_der(root,Goal) :-
      % A goal is derivable from the complete d-Prolog
      % knowledge base if and only if it succeeds.
        !,
        call(Goal).

strict_der(KB,(First,Rest)) :-
      % A conjunction is strictly derivable if and only if
      % both of its conjuncts are strictly derivable.
        !,
        strict_der(KB,First),
        strict_der(KB,Rest).

strict_der(KB,Goal) :-
      % A goal is strictly derivable from a knowledge base
```

```
      % other than the complete knowledge base if it is
      % a contained in that knowledge base.
        conjunct(Goal,KB).

strict_der(KB,Goal) :-
      % A goal is strictly derivable from a knowledge base
      % other than the complete knowledge base if it is
      % the head of an ordinary Prolog rule in the complete
      % knowledge base whose nonempty condition is strictly
      % derivable from that knowledge base.
        clause(Goal,Condition),
        Condition \== true,
        strict_der(KB,Condition).

%
% def_der(+KB,+Goal)
%   Succeeds if Goal is defeasibly derivable from the
%   knowledge base KB.
%

def_der(_,true) :- !.
      % `true' is defeasibly derivable from any
      % knowledge base.

def_der(KB,(First,Rest)) :-
      % A conjunction is defeasibly derivable if
      % and only if both conjuncts are defeasibly
      % derivable.
        !,
        def_der(KB,First),
        def_der(KB,Rest).

def_der(_,Goal) :-
        functor(Goal,Predicate,Arity),
        system(Predicate/Arity),
      % A goal with a built-in functor is defeasibly
      % derivable if and only if it succeeds.
        !,
        Goal.

def_der(KB,Goal) :-
      % A goal is defeasibly derivable if it is
      % strictly derivable.
        strict_der(KB,Goal).

def_der(_,Goal) :-
```

```
      % A goal is defeasibly derivable if it is
      % the head of an ordinary Prolog rule whose
      % condition is defeasibly derivable and which
      % is not rebutted.
        clause(Goal,Condition),
        Condition \== true,
        def_der(KB,Condition),
        \+ rebutted(KB,(Goal :- Condition)).

def_der(KB,Goal) :-
      % A goal is defeasibly derivable if it is
      % the head of a defeasible rule whose
      % condition is defeasibly derivable and which
      % is neither rebutted nor undercut.
        def_rule(KB,(Goal := Condition)),
        def_der(KB,Condition),
        \+ rebutted(KB,(Goal := Condition)),
        \+ undercut(KB,(Goal := Condition)).

def_der(KB,Goal) :-
        preemption,
      % If defeater preemption is enabled, then
      % a goal is defeasibly derivable if it is
      % the head of a defeasible rule whose condition
      % is defeasibly derivable, provided every
      % rebutting or undercutting defeater for that
      % rule is itself rebutted by a strict rule or
      % a superior defeasible rule.
        def_rule(KB,(Goal := Condition)),
        \+ (contrary(Goal,Contrary1),
             strict_der(KB,Contrary1)),
        def_der(KB,Condition),
        \+ (contrary(Goal,Contrary2),
             clause(Contrary2,Condition2),
             Condition2 \== true,
             def_der(KB,Condition2)),
        \+ (contrary(Goal,Contrary3),
             def_rule(KB,(Contrary3 := Condition3)),
             def_der(KB,Condition3),
             \+ (preempted(KB,(Contrary3 := Condition3)))),
        \+ (contrary(Goal,Contrary4),
             (Contrary4 :^ Condition4),
             def_der(KB,Condition4),
             \+ (preempted(KB,(Contrary4 :^ Condition4)))).

%
```

```
% contrary(+Clause1,-Clause2)
%   Discovers Clause2 which either is the complement of
%   Clause1 or is incompatible with Clause1.
%

contrary(Clause1,Clause2) :-
        incompatible(Clause1,Clause2).

contrary(Clause1,Clause2) :-
        incompatible(Clause2,Clause1).

contrary(Clause1,Clause2) :-
        comp(Clause1,Clause2).

%
% comp(+Clause1,-Clause2)
%   Succeeds if Clause1 is neg Clause2 or if Clause2 is
%   is neg Clause1.
%

comp(neg Atom,Atom) :-
        !.

comp(Atom,neg Atom).

%
% rebutted(+KB,+Rule)
%   Succeeds if the Rule is defeated in the knowledge
%   base KB by a rebutting defeater (an ordinary
%   Prolog rule or a defeasible rule to which the Rule
%   is not superior).
%

rebutted(KB,Rule) :-
      % Any rule is rebutted if a contrary of
      % its head is strictly derivable.
        Rule =.. [_,Head,_],
        contrary(Head,Contrary),
        strict_der(KB,Contrary),
        !.

rebutted(KB,Rule) :-
      % Any rule may be rebutted by an ordinary
      % Prolog rule with a contrary head.
        Rule =.. [_,Head,_],
        contrary(Head,Contrary),
```

```
        clause(Contrary,Body),
        Body \== true,
        def_der(KB,Body).

rebutted(KB,(Head := Body)) :-
      % Defeasible rules may be rebutted by other
      % defeasible rules with contrary heads.
        contrary(Head,Contrary),
        def_rule(KB,(Contrary := Condition)),
        def_der(KB,Condition),
        \+ sup_rule((Head := Body),(Contrary := Condition)),
        !.

%
% undercut(+KB,+Rule)
%   Succeeds if the Rule is defeated in the knowledge
%   base KB by an undercutting defeater.
%

undercut(KB,(Head := Body)) :-
      % Only defeasible rules may be undercut by pure
      % defeaters.
        contrary(Head,Contrary),
        (Contrary :^ Condition),
        def_der(KB,Condition),
        \+ sup_rule((Head := Body),(Contrary :^ Body)),
        !.

%
% sup_rule(+Rule1,+Rule2)
%   Succeeds if the body of Rule2 is defeasibly derivable
%   from the body of Rule1, but the body of Rule1 is not
%   defeasibly derivable from the body of Rule2. The user
%   can also force superiority by adding clauses for the
%   predicate `sup'.
%

sup_rule(Rule1,Rule2) :-
        sup(Rule1,Rule2).

sup_rule((_ := Body1),(_ := Body2)) :-
        def_der(Body1,Body2),
        \+ def_der(Body2,Body1).

sup_rule((_ := Body1),(_ :^ Body2)) :-
        def_der(Body1,Body2),
```

```
        \+ def_der(Body2,Body1).

%
% conjunct(+Clause1,+Clause2)
%

conjunct(Clause,Clause) :- !.

conjunct(Clause,(Clause,_)) :- !.

conjunct(Clause,(_,Rest)) :-
        conjunct(Clause,Rest).

%
% def_rule(+KB,+Rule)
%   Succeeds if KB is the entire d-Prolog knowledge base
%   and Rule is any defeasible rule in KB, or if Rule is a
%   defeasible rule in the d-Prolog knowledge base and the
%   body of Rule is not the atom `true'.
%

def_rule(root,(Head := Body)) :-
        !,
        (Head := Body).

def_rule(_,(Head := Body)) :-
        (Head := Body),
        Body \== true.

%
% preempted(+KB,+Rule)
%   Succeeds if the Rule is defeated in the knowledge
%   base KB by a superior rebutting defeater (an ordinary
%   Prolog rule or a defeasible rule which is superior
%   to the Rule.)
%

preempted(KB,Rule) :-
      % Any rule is preempted if a contrary of
      % its head is strictly derivable.
        Rule =.. [_,Head,_],
        contrary(Head,Contrary),
        strict_der(KB,Contrary),
        !.

preempted(KB,Rule) :-
```

```
      % Any rule may be preempted by an ordinary
      % Prolog rule with a contrary head.
        Rule =.. [_,Head,_],
        contrary(Head,Contrary),
        clause(Contrary,Body),
        Body \== true,
        def_der(KB,Body).

preempted(KB,(Head := Body)) :-
      % Defeasible rules may be preempted by superior
      % defeasible rules with contrary heads.
        contrary(Head,Contrary),
        def_rule(KB,(Contrary := Condition)),
        def_der(KB,Condition),
        sup_rule((Contrary := Condition),(Head := Body)),
        !.

%
% preempt
%   Toggles the preemption of defeaters feature between
%   enabled and disabled.
%

preempt :-
        retract(preemption),
        !,
        write('Preemption is disabled.'), nl.

preempt :-
        assert(preemption),
        write('Preemption is enabled.'), nl.

:- abolish(preemption/0).

:- consult('dputils.pl').
```

11.7. DEFINING STRICT DERIVABILITY IN PROLOG

The defeasible inference engine will be built around a special predicate `@/1` just as the inference engine for biconditionals in Chapter 6 was built around the special predicate `prove/1`. For any `Goal`, we will want the query

```
?- @(Goal).
```

to succeed if the defeasible inference engine can derive `Goal` from the database, including defeasible rules, presumptions, defeaters and negations with `neg`. If the inference engine cannot prove `Goal` using these resources, we will want the query to fail. Basically, then, the inference engine must define the predicate `@`.

We will make `@` a prefix operator so we can express the above query without parentheses as

```
?- @ Goal.
```

In fact, we define four new operators `:=/2`, `:^/2`, `neg/1`, and `@/1` using the built-in predicate `op/3` discussed in Chapter 6. File DPROLOG.PL thus begins with a procedure `init/0` containing the `op` declarations, followed immediately by a query invoking it. If the `op` declarations were not executed at this time, Prolog would be unable to understand the rest of the clauses in the file.

Remember that a literal may be either strictly or defeasibly derivable either from the complete d-Prolog database (the root knowledge base) or from the literals in the condition of some rule (together with the strict rules, defeasible rules, and defeaters in the root knowledge base). The query

```
?- @ Goal.
```

succeeds if `Goal` is at least defeasibly derivable from the root knowledge base. We need another binary predicate that takes both the knowledge base and the goal as arguments. Thus, we define `@` by

```
@ Goal :- def_der(root,Goal).
```

Before defining `def_der/2`, we will attack the easier task of defining the predicate for strict derivability, `strict_der/2`. For the root knowledge base, strict derivability is just ordinary derivability in Prolog.

```
strict_der(root,Goal) :-
        !,
        call(Goal).
```

We include the cut because we do not want the inference engine to apply any later clauses upon backtracking since these are intended only for knowledge bases other than the root knowledge base.

A literal is strictly derivable from a knowledge base `KB` if it is *in* `KB` or if it is the head of a strict rule in the root knowledge base whose condition is strictly derivable from `KB`. Since conditions may be conjunctions of literals, we will handle this case first. A conjunction of literals is strictly derivable from `KB` if each conjunct is strictly derivable from `KB`.

```
strict_der(KB,(First,Rest)) :-
        !,
        strict_der(KB,First),
        strict_der(KB,Rest).
```

We put this clause first and include a cut so the inference engine will not try to apply any of the clauses intended for literals to a complex condition before it has broken the condition into individual literals.

For any knowledge base KB other than the root knowledge base, we have two cases left to consider. One is where the literal we are trying to derive is *in* KB and the other is where it is the head of a strict rule whose condition is strictly derivable from KB. We have a separate clause for each of these situations. Remember that any knowledge base other than the root knowledge base will be identified with the condition of some other rule. So it will be a conjunction of literals. A literal is *in* a conjunction of literals if it is one of the conjuncts.

```
strict_der(KB,Goal) :-
        conjunct(Goal,KB).

strict_der(KB,Goal) :-
        clause(Goal,Condition),
        Condition \== true,
        strict_der(KB,Condition).
```

We eliminate rules in the second clause above whose conditions are the built-in predicate `true` because these rules are the facts in the root knowledge base. Remember that if we don't do this, then any fact in the root knowledge base will be strictly derivable from any rule condition whatsoever. So, for example, if we know that Opus is both a bird and a penguin, we would be able to show that being a penguin *follows* from being a bird. Since we certainly don't want to be able to do that, we have the restriction.

The auxilliary predicate `conjunct/2` is easily defined.

```
conjunct(Clause,Clause) :- !.

conjunct(Clause,(Clause,_)) :- !.

conjunct(Clause,(_,Rest)) :-
        conjunct(Clause,Rest).
```

What we have done in defining strict derivability is to call the Prolog interpreter for the root knowledge base and to define a Prolog interpreter in Prolog for other knowledge bases.

11.8. D-PROLOG: PRELIMINARIES

Before looking at how we apply defeasible rules, we must attend to some other details.

Remember that a presumption is a defeasible rule of the form

```
Goal := true.
```

In using presumptions, an inference engine will eventually contend with the query `?- def_der(KB,true)`. Of course, this query should succeed. So the first clause in the definition of `def_der/2` is simply

```
def_der(KB,true) :- !.
```

As with strict derivability, the inference engine will sometimes need to show that the condition of a rule is defeasibly derivable. To do this, it must break the condition down into its individual literals.

```
def_der(KB,(First,Rest)) :-
        !,
        def_der(KB,First),
        def_der(KB,Rest).
```

When the inference engine encounters a built-in Prolog predicate, it must evaluate the goal containing that predicate using the built-in definition of the predicate regardless of the knowledge base upon which the derivation is based.

```
def_der(_,Goal) :-
        functor(Goal,Predicate,Arity),
        built_in(Predicate/Arity),
        !,
        Goal.
```

Of course, a literal is *at least* defeasibly derivable from a knowledge base if it is *strictly* derivable from that knowledge base.

```
def_der(KB,Goal) :-
        strict_der(KB,Goal).
```

The last clause before looking at defeasible rules defines how strict rules may be involved in defeasible derivations. Since strict rules can't be undercut, they apply if their conditions are at least defeasibly satisfied and they are not rebutted.

```
def_der(KB,Goal) :-
        clause(Goal,Condition),
        Condition \== true,
        def_der(KB,Condition),
        \+ rebutted(KB,(Goal :- Conditional)).
```

We eliminate strict rules with `true` as their condition since these are facts and cannot be used in deriving a literal from the condition of another rule in testing for specificity.

How can a strict rule be rebutted? It is rebutted if a contrary of its head is *strictly* derivable (when we know *conclusively* that the head of the rule is false.) Otherwise, it can only be rebutted by another strict rule.

```
rebutted(KB,Rule) :-
        Rule =.. [_,Head,_],
        contrary(Head,Contrary),
        strict_der(KB,Contrary),
        !.

rebutted(KB,Rule) :-
```

```
        Rule =.. [_,Head,_],
        contrary(Head,Contrary),
        clause(Contrary,Body),
        Body \== true,
        def_der(KB,Body).
```

Notice that these two clauses apply to *all* rules, including defeasible rules. Naturally, any set of circumstances that will rebut a strict rule will also rebut a defeasible rule.

11.9. USING DEFEASIBLE RULES

Under what circumstances can we apply a defeasible rule? As an example, let's take the simple rule

```
flies(X) := bird(X).
```

To use this rule to conclude tentatively that Opus flies, we need to establish three things:

1. We conclude at least tentatively that Opus is a bird.
2. Our rule is not rebutted.
3. Our rule is not undercut.

An initial attempt at representing this as a general principle in Prolog is simple. First we find a defeasible rule whose head unifies with our goal. Then we apply the three tests listed above.

```
def_der(KB,Goal) :-
        def_rule(KB,(Goal := Condition)),
        def_der(KB,Condition),
        \+ rebutted(KB,(Goal := Condition)),
        \+ undercut(KB,(Goal := Condition)).
```

We must define which rules belong in a knowledge base because we only use presumptions when we are reasoning from the root knowledge base. This gives us a simple definition of `def_rule/2`:

```
def_rule(root,(Head := Body)) :-
        !,
        (Head := Body).

def_rule(KB,(Head := Body)) :-
        (Head := Body),
        Body \== true.
```

Of course, we need to specify the conditions in which a defeasible rule is rebutted or defeated. Recall from the last section that any rule, strict or defeasible, is rebutted if a contrary of its head is either strictly derivable or is the head of some strict rule that is at least defeasibly satisfied. But defeasible rules can also be rebutted by other competing, noninferior defeasible rules with contrary heads.

```
rebutted(KB,(Head := Body)) :-
        contrary(Head,Contrary),
        def_rule(KB,(Contrary := Condition)),
        def_der(KB,Condition),
        \+ sup_rule((Head := Body),(Contrary := Condition)),
        !.
```

We use a cut because we are never interested in knowing that there are different ways to rebut a rule. If it is rebutted at all, then it cannot be applied.

One rule is superior to another if we have a clause for the predicate `sup/2` that indicates this, or if the first rule is more specific than the second. The later situation holds when the body of the second rule is defeasibly derivable from the body of the first, but at least one literal in the body of the first rule is not defeasibly derivable from the body of the second.

```
sup_rule(Rule1,Rule2) :-
        sup(Rule1,Rule2).

sup_rule((Head1 := Body1),(Head2 := Body2)) :-
        def_der(Body1,Body2),
        \+ def_der(Body2,Body1).
```

Defeasible rules are undercut by defeaters in much the same way they are rebutted by other defeasible rules. The only difference is that a defeater does not support an argument for a contrary conclusion.

```
undercut(KB,(Head := Body)) :-
        contrary(Head,Contrary),
        (Contrary :^ Condition),
        def_der(KB,Condition),
        \+ sup_rule((Head := Body),(Contrary :^ Body)),
        !.
```

Once again, the cut is used because we only need to know that there is at least one way that the rule is undercut. Since superiority also applies to defeaters, we need another clause for `sup_rule/2`:

```
sup_rule((Head1 := Body1),(Head2 :^ Body2)) :-
        def_der(Body1,Body2),
        \+ def_der(Body2,Body1).
```

11.10. PREEMPTION OF DEFEATERS

Arguably, our requirement that we cannot apply a defeasible rule unless it is superior to every competing defeasible rule or defeater that is satisfiable is too strong. We will look at an example that illustrates the problem with this requirement. To begin, we know that southerners are typically conservatives and conservatives are typically Republicans, even though southerners typically are not Republicans. Furthermore,

Figure 11.5 Twin Tweety Triangles.

movie stars are typically rich and rich folks are typically Republicans, even though movie stars typically are not Republicans. Kim Bassinger is a southern movie star. The d-Prolog representation of all this information is as follows:

```
[1]  republican(X) := conservative(X).
[2]  republican(X) := rich(X).
[3]  neg republican(X) := southerner(X).
[4]  neg republican(X) := movie_star(X).
[5]  conservative(X) := southern(X).
[6]  rich(X) := movie_star(X).
     southerner(kim).
     movie_star(kim).
```

The question, of course, is whether we should conclude that Kim Bassinger is a Republican. The intuitively correct answer is that she is not.

The structure of the rules is perhaps easier to see in the graph in Figure 11.10. You will probably recognize this as an example of twin Tweety Triangles. When we look at these rules, we see that [3] is superior to [1] (since it is more specific) but not to [2], while [4] is superior to [2] (since it is more specific) but not to [1]. So neither of the two rules supporting the intuitively correct rule that Kim is not a Republican is superior to all competitors. How can we save the situation?

Based on examples like this, we can insist that any defeasible rule that is itself rebutted by a superior defeasible rule loses its capacity to rebut any other rule. Then since [1] is rebutted by the superior rule [3] and [2] is rebutted by the superior rule [4], neither [1] nor [2] has the capacity to rebut either [3] or [4]. In this case, we say

Art work to be provided later.

Figure 11.6 An Example of an Undercut Defeater.

that the rules [1] and [2] have been *preempted* as defeaters.

Perhaps a defeasible rule should be preempted as a potential defeater if it is either rebutted or undercut by a superior rule. This, however, appears to be too strong as the following modification of our original Tweety Triangle shows. Remember that birds normally fly, that penguins normally don't fly, and that penguins are birds. Now suppose we develop a genetically altered penguin with large wings and correspondingly large 'flight' muscles. Can these genetically altered penguins fly? We should probably take the information that a particular penguin has this genetic alteration as a reason to suspend our usual judgment that it can not fly. We are assuming, of course, that the genetic alteration envisioned is not so great that the creatures cease to be penguins. They are still still penguins albeit strange ones. Suppose Folio is one of these genetically altered penguins. (A folio is a kind of opus, isn't it?) In the graphic representation of this example (Figure 11.10), we use a wavy arrow for the defeater. The following is the d-Prolog representation of the same example:

```
flies(X) := bird(X)
neg flies(X) := penguin(X).
flies(X) :^ ga_penguin(X).
bird(X) :- penguin(X).
penguin(X) :- ga_penguin(X).
ga_penguin(folio).
```

The important point to note about this example is that we are inclined to *withhold* judgment about whether Folio can fly; we are not inclined to infer even tentatively that he *can* fly. But if the penguin rule were preempted by the undercutting defeater for genetically altered penguins, then it could no longer rebut the bird rule and we could then use the bird rule to infer that apparently Folio flies. Since this runs contrary to our intuitions about this kind of example, we conclude that potential defeaters are not preempted when they are only undercut by superior defeaters. They must actually be rebutted by a superior rule to be preempted.

Putting all of this together we get another, rather complicated, clause for the predicate `def_der/2`:

```
def_der(KB,Goal) :-
[1]     preemption,
[2]     def_rule(Goal := Condition),
[3]     \+ (contrary(Goal,Contrary1),
             strict_der(KB,Contrary1)),
[4]     def_der(KB,Condition),
[5]     \+ (contrary(Goal,Contrary2),
             clause(Contrary2,Condition2),
             Condition2 \== true,
             def_der(KB,Condition2)),
[6]     \+ (contrary(Goal,Contrary3),
             def_rule(KB,(Contrary3 := Condition3)),
             def_der(KB,Condition3),
             \+ (rebutted(KB,(Contrary3 := Condition3)))),
[7]     \+ (contrary(Goal,Contrary4),
             (Contrary4 :^ Condition4),
             def_der(KB,Condition4),
             \+ (rebutted(KB,(Contrary4 :^ Condition4)))).
```

We will examine these conditions one at a time.

This clause is considerably more complex than our earlier clauses for applying defeasible rules. If we add it to our inference engine, Prolog will have much more work to do to check to see whether a literal is defeasibly derivable. At the same time, twin Tweety Triangles are relatively unusual. So we give the d-Prolog developer the option of enabling or disabling preemption of defeaters. If it is enabled, the clause `preemption` will be found in the database and condition [1] of our rule will be satisfied. Otherwise, condition [1] will fail and Prolog will ignore the rest of the clause. We define a predicate `preempt/0` which toggles between enabling and disabling preemption of defeaters, and by default preemption is disabled when d-Prolog is loaded.

```
preempt :-
        retractall(preemption),
        !,
        write('Preemption is disabled.'), nl.

preempt :-
        assert(preemption),
        write('Preemption is enabled.'), nl.
```

```
:- retractall(preemption).
```

Condition [2] finds any available defeasible rule that we might try to apply, condition [3] makes sure that no contrary of the head of the rule (after unifying with the goal) is strictly derivable, and condition [4] checks to see if the rule is satisfiable. The rest of our conditions check to make sure that the rule is not defeated in one of several ways. Condition [5] checks to see that the rule is not rebutted by any strict rule (which can never be preempted). Conditions [6] and [7] check to see that every defeasible rule or defeater that might otherwise rebut or undercut the rule is preempted. The definition for the auxiliary predicate `preempted/2` is just like the definition for `rebutted/2` except that in the last clause, the rebutting defeasible rule must be superior to the rule it preempts rather than just not inferior.

```
preempted(KB,Rule) :-
        Rule =.. [_,Head,_],
        contrary(Head,Contrary),
        strict_der(KB,Contrary),
        !.

preempted(KB,Rule) :-
        Rule =.. [_,Head,_],
        contrary(Head,Contrary),
        clause(Contrary,Body),
        Body \== true,
        def_der(KB,Body).

preempted(KB,(Head := Body)) :-
        contrary(Head,Contrary),
        def_rule(KB,(Contrary := Condition)),
        def_der(KB,Condition),
        sup_rule((Contrary := Condition),(Head := Body)),
        !.
```

This completes the definition of our defeasible inference engine.

```
% DPUTILS.PL
%
% Utilities for d-Prolog.
%

:- op(1100,fx,@@).

:- dynamic have_seen_predicate_before/2.
:- dynamic dPrologDictionary/2.

%
% @@(+Goal)
%   Tests to see if Goal is strictly or defeasibly
%   derivable from the d-Prolog knowledge base. Goal
%   may not contain an uninstantiated variable.
%

@@ Goal :-
        Goal =.. [_|ArgumentList],
        member(Argument,ArgumentList),
        var(Argument),
        write('Improper argument for @@.'), nl,
        write('Argument contains an uninstantiated variable.'),
        nl,
        !,
        fail.

@@ Goal :-
        strict_der(root,Goal),
        contrary(Goal,Contrary),
        Contrary,
        !,
        write('definitely, yes -'), nl,
        write('and definitely, no - contradictory.'), nl.

@@ Goal :-
        strict_der(root,Goal),
        !,
        write('definitely, yes.'), nl.

@@ Goal :-
        contrary(Goal,Contrary),
        Contrary,
        !,
        write('definitely, no.'), nl.
```

```
@@ Goal :-
        def_der(root,Goal),
        !,
        write('presumably, yes.'), nl.

@@ Goal :-
        contrary(Goal,Contrary),
        def_der(root,Contrary),
        !,
        write('presumably, no.'), nl.

@@ _ :-
        write('can draw no conclusion.'), nl.

%
% member(?X,?Y)
%   Succeeds if X is a member of the list Y.
%

member(X,[X|_]).

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

%
% dlisting(+Predicate)
%   Lists all ordinary Prolog clauses, defeasible rules, and
%   defeaters in memory which have heads of the form
%   Predicate(...Arguments...) or of the form
%   neg Predicate(...Arguments...).
%

dlisting(Predicate) :-
        listing(Predicate),
        fail.

dlisting(Predicate) :-
        clause(neg Head,Body),
        functor(Head,Predicate,_),
        pprint(neg Head,' :-',Body),
        fail.

dlisting(Predicate) :-
        (Head := Body),
        dfunctor(Head,Predicate,_),
        pprint(Head,' :=',Body),
        fail.
```

```
dlisting(Predicate) :-
        (Head :^ Body),
        dfunctor(Head,Predicate,_),
        pprint(Head,' :^',Body),
        fail.

dlisting(Predicate) :-
        clause(incompatible(Clause1,Clause2),Body),
        dfunctor(Clause1,Predicate,_),
        pprint(incompatible(Clause1,Clause2),' :-',Body),
        fail.

dlisting(Predicate) :-
        clause(incompatible(Clause1,Clause2),Body),
        dfunctor(Clause2,Predicate,_),
        pprint(incompatible(Clause1,Clause2),' :-',Body),
        fail.

dlisting(Predicate) :-
        clause(sup(Clause1,Clause2),Body),
        rule_functor(Clause1,Predicate,_),
        pprint(sup(Clause1,Clause2),' :-',Body),
        fail.

dlisting(Predicate) :-
        clause(sup(Clause1,Clause2),Body),
        rule_functor(Clause2,Predicate,_),
        pprint(sup(Clause1,Clause2),' :-',Body),
        fail.

dlisting(_).

%
% dlist
%   dlists all predicates in the d-Prolog dictionary.
%

dlist :-
        dPrologDictionary(Predicate,_),
        dlisting(Predicate),
        fail.

dlist.

%
```

```
% dfunctor(+Clause,-Predicate,-Arity)
%   Returns the d-Prolog Predicate of Clause
%   together with its Arity.
%

dfunctor(neg Clause,Predicate,Arity) :-
        functor(Clause,Predicate,Arity),
        !.

dfunctor(Clause,Predicate,Arity):-
        functor(Clause,Predicate,Arity).

%
% rule_functor(+Rule,-Predicate,-Arity)
%   Returns the d-Prolog Predicate of the head of
%   the Rule together with its Arity.
%

rule_functor((Head :- _),Predicate,Arity) :-
        dfunctor(Head,Predicate,Arity),
        !.

rule_functor((Head := _),Predicate,Arity) :-
        dfunctor(Head,Predicate,Arity),
        !.

rule_functor((Head :^ _),Predicate,Arity) :-
        dfunctor(Head,Predicate,Arity),
        !.

%
% pprint(+Head,+Operator,+Body)
%   A formatting routine for printing ordinary Prolog clauses,
%   defeasible rules, and defeaters.
%

pprint(Head,' :-',true) :-
        !,
        write(Head), write('.'), nl.

pprint(Head,Operator,Clause) :-
        write(Head), write(Operator), nl,
        pprint(Clause).

pprint((First,Rest)) :-
        !,
```

```
        write('    '), write(First), write(','), nl,
        pprint(Rest).

pprint(Clause) :-
        write('    '), write(Clause), write('.'), nl.

%
% rescind(+Predicate/+Arity)
%   Removes all ordinary Prolog clauses, defeasible rules,
%   or defeaters whose heads are of the form
%   Predicate(...Arguments...) or of the form
%   neg Predicates(...Argument...).
%

rescind(Predicate/Arity) :-
        abolish(Predicate/Arity),
        make_variable_list(Arity,[],VariableList),
        Clause =.. [Predicate|VariableList],
        retractall(Clause),
        retractall(neg Clause),
        retractall((Clause := _)),
        retractall((neg Clause := _)),
        retractall((Clause :^ _)),
        retractall((neg Clause :^ _)),
        retractall(incompatible(Clause,_)),
        retractall(incompatible(_,Clause)),
        retractall(incompatible(neg Clause,_)),
        retractall(incompatible(_,neg Clause)),
        retractall(sup((Clause := _),_)),
        retractall(sup(_,(Clause := _))),
retractall(sup((neg Clause := _),_)),
        retractall(sup(_,(neg Clause := _))),
        retractall(sup((Clause :^ _),_)),
        retractall(sup(_,(Clause :^ _))),
retractall(sup((neg Clause :^ _),_)),
        retractall(sup(_,(neg Clause :^ _))),
        retractall(dPrologDictionary(Predicate,Arity)).

%
% rescindall
%   Removes from memory the entire d-Prolog knowledge base,
%   provided it was loaded into memory using reload.
%

rescindall :-
        dPrologDictionary(Predicate,Arity),
```

```
        rescind(Predicate/Arity),
        fail.

rescindall.

%
% dload(+Filename)
%   Consults a file containing a d-Prolog knowledge base.
%

dload(Filename) :-
        concat([Filename,'.dpl'],NewFilename),
see(NewFilename),
        repeat,
        read(Term),
        execute_term(Term),
        add_to_memory(Term),
        remember_predicate(Term),
        Term == end_of_file,
        seen,
        !.

%
% reload(+Filename)
%   Reconsults a file containing a d-Prolog knowledge base.
%

reload(Filename) :-
        concat([Filename,'.dpl'],NewFilename),
see(NewFilename),
        repeat,
        read(Term),
        execute_term(Term),
        rescind_previous_clauses(Term),
        add_to_memory(Term),
        Term == end_of_file,
        retractall(have_seen_predicate_before/2),
        seen,
        !.

%
% execute_term(+Term)
%   During a dload or a reload, executes any queries
%   read from a d-Prolog knowledge base.
%
```

```
execute_term((:- Goal)) :-
        Goal,
        !,
        fail.

execute_term(_).

get_term(Handle,Term) :-
        read(Handle,Term),
        !.

get_term(_,end_of_file).

%
% add_to_memory(+Clause)
%   Adds clauses to memory during a reload.
%

add_to_memory((:- _)) :- !.

add_to_memory(end_of_file) :- !.

add_to_memory(Term) :-
        assertz(Term).

%
% remember_predicate(+Term)
%   Adds new predicates to the dPrologDictionary
%   during a dload.
%

remember_predicate((:- _)) :- !.

remember_predicate(end_of_term) :- !.

remember_predicate(Rule) :-
        rule_functor(Rule,Predicate,Arity),
        add_to_dictionary(Predicate,Arity),
        !.

remember_predicate(sup(Rule1,Rule2)) :-
        rule_functor(Rule1,Predicate1,Arity1),
        add_to_dictionary(Predicate1,Arity1),
        rule_functor(Rule2,Predicate2,Arity2),
        add_to_dictionary(Predicate2,Arity2),
        !.
```

```
remember_predicate(incompatible(Goal1,Goal2)) :-
        dfunctor(Goal1,Predicate1,Arity1),
        add_to_dictionary(Predicate1,Arity1),
        dfunctor(Goal2,Predicate2,Arity2),
        add_to_dictionary(Predicate2,Arity2),
        !.

remember_predicate(Goal) :-
        dfunctor(Goal,Predicate,Arity),
        add_to_dictionary(Predicate,Arity).

%
% add_to_dictionary(+Predicate,+Arity)
%   Adds a clause to dPrologDictionary/2 for
%   Predicate and Arity if one is not already
%   present.
%

add_to_dictionary(Predicate,Arity) :-
        \+ dPrologDictionary(Predicate,Arity),
        assert(dPrologDictionary(Predicate,Arity)).

add_to_dictionary(_,_).

%
% rescind_previous_clauses(+Term)
%   Removes from memory all ordinary Prolog clauses,
%   defeasible rules, or defeaters that were not loaded
%   during the current invocation of reload and that,
%   from the perspective of the d-Prolog inference engine,
%   are clauses for the same predicate as Term.
%

rescind_previous_clauses((:- _)) :- !. % Query to execute.

rescind_previous_clauses(end_of_file) :- !.

rescind_previous_clauses((Head :- _)) :-
        rescind_previous_clauses(Head),
        !.

rescind_previous_clauses((Head := _)) :-
        rescind_previous_clauses(Head),
        !.
```

```
rescind_previous_clauses((Head :^ _)) :-
        rescind_previous_clauses(Head),
        !.

rescind_previous_clauses(incompatible(X,Y)) :-
        rescind_previous_clauses(X),
        rescind_previous_clauses(Y),
        !.

rescind_previous_clauses(sup(Rule1,Rule2)) :-
        rescind_previous_clauses(Rule1),
        rescind_previous_clauses(Rule2),
        !.

rescind_previous_clauses(Clause) :-
        dfunctor(Clause,Predicate,Arity),
        \+ have_seen_predicate_before(Predicate,Arity),
        asserta(have_seen_predicate_before(Predicate,Arity)),
        rescind(Predicate/Arity),
        remember_predicate(Clause),
        !.

rescind_previous_clauses(_).

%
% make_variable_list(N,Sofar,List)
%   Creates a list with N-many anonymous variables in
%   its head and Sofar as its tail.
%

make_variable_list(0,X,X) :- !.

make_variable_list(N,OldList,VariableList) :-
        M is N - 1,
        make_variable_list(M,[_|OldList],VariableList).

%
% dictionary
%   Prints a list of all predicates and their arities that
%   occur in a d-Prolog knowledge base loaded into memory
%   using dload or reload.
%

dictionary :-
        dPrologDictionary(Predicate,Arity),
        write(Predicate),
```

```
        write('/'),
        write(Arity),
        nl,
        fail.

dictionary :- !.

%
% contradictions
%   Finds all contradictory goals for the d-Prolog knowledge
%   base that succeed, displays them, and stores them as
%   clauses for the predicate contradictory_pair/2.
%

contradictions :-
        abolish(contradictory_pair/2),
        fail.

contradictions :-
        dPrologDictionary(Predicate,Arity),
        make_variable_list(Arity,[],VariableList),
        Clause1 =.. [Predicate|VariableList],
        clause(Clause1,_),
        contrary(Clause1,Clause2),
        contradictions_aux(Clause1,Clause2),
        assert(contradictory_pair(Clause1,Clause2)),
        write(Clause1), write(' - '), write(Clause2), nl,
        fail.

contradictions.

contradictions_aux(X,Y) :-
        \+ contradictory_pair(X,Y),
        \+ contradictory_pair(Y,X),
        X,
        Y,
        !.

%
% whynot(+Goal)
%   Explains why Goal is not defeasibly derivable.
%

whynot(Goal) :-
        Goal, nl,
        write('Why not indeed!'), nl,
```

```
        write(Goal),
        write(' is strictly derivable!'), nl.

whynot(Goal) :-
        (@ Goal), nl,
        write('Why not indeed!'), nl,
        write(Goal),
        write(' is defeasibly derivable!'), nl.

whynot(Goal) :-
        \+ initial_evidence(Goal), nl,
        write('There are no rules in the database for '), nl,
        write(Goal), nl,
        write('whose antecedent is defeasibly derivable.'), nl.

whynot(Goal) :-
        clause(Goal,Body),
        (@ Body),
        contrary(Goal,Opposite),
        Opposite, nl,
        write('The antecedent of the strict rule'), nl, nl,
        pprint(Goal,' :- ',Body), nl,
        write('is defeasibly derivable; but'),
        write(Opposite), nl,
        write('is strictly derivable.'), nl, nl.

whynot(Goal) :-
        def_rule(root,(Goal := Body)),
        (@ Body),
        contrary(Goal,Opposite),
        Opposite, nl,
        write('The antecedent of the defeasible rule'), nl, nl,
        pprint(Goal,' := ',Body), nl,
        write('is defeasibly derivable; but'),
        write(Opposite), nl,
        write('is strictly derivable.'), nl, nl.

whynot(Goal) :-
        def_rule(root,(Goal := Body)),
        (@ Body),
        contrary(Goal,Opposite),
        (Opposite,Tail),
        Tail \== true,
        (@ Tail), nl,
        write('The antecedent of the defeasible rule'), nl, nl,
        pprint(Goal,' := ',Body), nl,
```

```
        write('is defeasibly derivable;'), nl,
        write('but the condition of the strict rule'), nl, nl,
        pprint(Opposite,' :- ',Tail), nl,
        write('is also defeasibly derivable.'), nl, nl.

whynot(Goal) :-
        def_rule(root,(Goal := Body)),
        (@ Body),
        contrary(Goal,Opposite),
        def_rule(root,(Opposite := Tail)),
        (@ Tail),
        \+ sup_rule((Goal := Body),(Opposite := Tail)),
        nl,
        write('The antecedent of the defeasible rule'), nl, nl,
        pprint(Goal,' := ',Body), nl,
        write('is defeasibly derivable; but this rule is not'), nl,
        write('superior to the rule'), nl, nl,
        pprint(Opposite,' := ',Tail), nl,
        write('whose antecedent is also defeasibly derivable.'), nl, nl.

whynot(Goal) :-
        def_rule(root,(Goal := Body)),
        (@ Body),
        contrary(Goal,Opposite),
        defeater((Opposite :^ Tail)),
        (@ Tail),
        \+ sup_rule((Goal := Body),(Opposite :^ Tail)), nl,
        write('The antecedent of the defeasible rule'), nl, nl,
        pprint(Goal,' := ',Body), nl,
        write('is defeasibly derivable; but this rule is not'), nl,
        write('superior to the defeater ;'), nl, nl,
        pprint(Opposite,' :^ ',Tail), nl,
        write('whose antecedent is also defeasibly derivable.'), nl, nl.

%
% initial_evidence(+Goal)
%   Succeeds if there is a rule supporting Goal that
%   is satisfied, i.e., whose body is at least
%   defeasibly derivable.
%

initial_evidence(Goal) :-
        clause(Goal,Body),
        (@ Body).

initial_evidence(Goal) :-
```

```
        def_rule(root,(Goal := Body)),
        (@ Body).

%
% concat(+List,-String)
%   Concatenates a List of strings into a single
%   String.
%

concat(List,String) :-
concat_aux(List,[],String).

concat_aux([],Chars,String) :-
name(String,Chars).

concat_aux([First|Rest],Chars,String) :-
name(First,List),
append(Chars,List,NewChars),
!,
concat_aux(Rest,NewChars,String).
```

11.11. DEFEASIBLE QUERIES AND EXHAUSTIVE RESPONSES

The query `?- @ Goal`, where `Goal` contains no variables, evokes a response of `yes` or `no` from d-Prolog. If the answer is `yes`, we do not know whether defeasible rules were used in the derivation of `Goal`; if the answer is `no`, we do not know if d-Prolog is able to show that `Goal` is positively false (i.e., that `?- @ neg Goal` succeeds) or merely that d-Prolog is unable to defeasibly derive `Goal`. To get all this information, we need to try some combination of the following four queries:

1. `?- Goal.`
2. `?- neg Goal.`
3. `?- @ Goal.`
4. `?- @ neg Goal.`

A utility predicate is provided to make this process simpler.

The first line in DPUTILS.PL is a query that declares a new operator, `@@/1`. This operator is used to ask d-Prolog to provide an exhaustive response to a defeasible query without variables. It would be very difficult to give an exhaustive response to a query containing variables since different responses would be appropriate for different instantiations of the variables. So the first clause in the definition of `@@` checks to see if an uninstantiated variable occurs in the goal. If it does, the user is informed of the problem and the query succeeds. We should remember here that disjunctions, cuts, and negations-as-failure also may not appear in the argument for

`@@` just as they may not appear in the argument for `@`. If this restriction is violated, then the user will not be warned but the behavior of the inference engine is unpredictable.

Having determined that the goal contains no uninstantiated variables, the next clause tests to see if both the goal and some contrary of the goal are strictly derivable. If so, then there is a contradiction in the knowledge base and the user is so informed. If the system does not find a contradiction, it tests again to see if the goal is strictly derivable. If so, the response to the user is

```
definitely, yes.
```

If not, the system tests to see if any contrary of the goal is strictly derivable. If so, the response to the user is

```
definitely, no.
```

If neither the goal nor any contrary is strictly derivable, the system checks to see if the goal or any contrary is defeasibly derivable. If so, the response is, respectively,

```
presumably, yes.
```

or

```
presumably, no.
```

We do not have to worry about getting both answers here since the only way that a goal and a contrary of that goal can be both defeasibly derivable is if both are strictly derivable. That case is handled by an earlier clause.

Finally, the system may be unable to strictly or defeasibly derive either the goal or any contrary of the goal. Then the response to the user is

```
can draw no conclusion.
```

11.12. LISTING DEFEASIBLE PREDICATES

In response to the query

```
?- dlisting(pred).
```

d-Prolog lists all clauses which d-Prolog recognizes as belonging to the predicate `pred`. So `dlisting/1` is the d-Prolog equivalent of the built-in Prolog predicate `listing`. We need a special utility to list the d-Prolog clauses for a predicate because Prolog will rcognize the clauses

```
neg p.
p := q.
neg p :^ r.
sup((p := q),(neg p :^ r)).
incompatible(p,(neg s)).
```

as belonging, respectively, to the predicates `neg/1`, `:=/2`, `:^/2`, `sup/2`, and `incompatible/2`. Prolog will list none of these clauses in response to the query

```
?- listing(p).
```

However, the query

```
?- dlisting(p).
```

will list these clauses and any others in memory that belong to the predicate `p` from the perspective of d-Prolog.

The definition of `dlisting` includes many clauses to cover all the possible cases, but there is nothing very difficult to understand about any of these clauses and we will not discuss them in detail. The pridicate `dlisting` also uses the auxiliary predicate `pprint/1` to format rules.

Another utility predicate, `dlist/0`, will perform a listing for all predicates in the d-Prolog dictionary. This dictionary is created by the two utilities `dload/1` and `reload/1` described in the next section.

11.13. CONSULTING AND RECONSULTING D-PROLOG FILES

As we saw in the previous section, Prolog does not recognize the same clauses for a predicate that the d-Prolog inference engine recognizes. We also have to take this difference between Prolog and d-Prolog into account when we reconsult a d-Prolog file. Normally all the clauses for a given predicate are contiguous in a file, i.e., clauses for one predicated are not interrupted by clauses for any other predicate. Many Prologs expect this during a reconsult and discard all clauses currently in memory for a predicate whenever a new block of clauses for that predicate is encountered. For example, if a file containing only the three clauses `odd(1)`, `even(2)`, and `odd(3)` were reconsulted by a Prolog that behaves in this way, clause `odd(1)` would not be in memory after the reconsult. Because the clauses for the predicate `odd` are interrupted by a clause for `even`, Prolog treats the clause `odd(3)` as though it were a clause for a new predicate and deletes all clauses for `odd` currently in memory. So Prolog adds `odd(1)` to memory, then deletes it before adding `odd(3)` to memory during the reconsult.

The problem, of course, is that Prolog sees all defeasible rules as clauses for the predicates `:=`, all undercutting defeaters as clauses for the predicate `:^`, and so on. If we reconsult a file containing the clauses

```
red(X) := apple(X).
yellow(X) := banana(X).
grape(this_fruit).
purple(X) := grape(X).
```

only the last defeasible rule in the file will be in memory after the reconsult.

We solve this problem by developing our own reconsult utility `reload/1` defined in DPUTILS.PL. The proper query to reconsult a d-Prolog file is

```
?- reload(filename).
```

Here `filename` should not include an extension; `reload` expects the file to have the extension `.DPL` for d-Prolog. The utility reads terms from the specified file one by one. First it tries to execute them. This allows the user to include queries in a file which are executed whenever the file is consulted or reconsulted. Next, if the term is a clause and not a query, `reload` determines the d-Prolog predicate for the term. If it has not seen a term for the predicate before, it stores the predicate and its arity in a clause for the predicate `have_seen_predicate_before/2` and it uses the utility predicate `rescind/1` (see next section) to remove all d-Prolog clauses for the predicate from memory. It also stores the predicate and and its arity in a clause for the predicate `dPrologDictionary`. Finally the utility adds the clause to memory. It processes each term in the file in this manner until it encounters an `end_of_file` marker. Then it removes all clauses for `have_seen_predicate_before` from memory but leaves the d-Prolog dictionary in memory for later use by other utilities explained below.

Consulting a file is a much simpler operation since nothing has to be removed from memory during a consult. But we nevertheless provide a d-Prolog utility `dload/1` to consult d-Prolog files. The proper query to consult a d-Prolog file is

```
?- dload(filename).
```

Once again, no extension should be included in `filename` and `dload` expects the file to have the extension `.DPL`. The only difference between using `dload` and `consult` for d-Prolog files is that `dload` builds the d-Prolog dictionary used by other d-Prolog utilities.

The two utilities `dload` and `reload` take longer to consult or to reconsult a file than do the built-in utilities `consult` and `reconsult`. This is to be expected since they are doing more work.

11.14. THE D-PROLOG DICTIONARY

In the last section, we described how the d-Prolog utility predicates `dload` and `reload` build a d-Prolog dictionary as they consult or reconsult d-Prolog files. The dictionary consists of the clauses for the predicate `dPrologDictionary/2`. Each clause has the form

```
dPrologDictionary(Pred,Arity).
```

where `Pred` is the name of some d-Prolog predicate loaded into memory using `dload` or `reload`, and `Arity` is the arity of that predicate.

The first utility to use the d-Prolog dictionary is the predicate `dictionary/0`. d-Prolog lists all predicates and their arities stored in the d-Prolog dictionary in response to the query `?- dictionary.`

11.15. RESCINDING PREDICATES AND KNOWLEDGE BASES

On occasion a user may want to remove all clauses for a particular predicate from memory. For many Prolog implementations there is a built-in predicate `abolish/1` that will remove all clauses from memory for a Prolog predicate. The appropriate query for a predicate `Pred` with arity `Arity` is

```
?- abolish(Pred/Arity).
```

Of course, we will run into the same complication here that we do with the built-in predicate `listing`: many clauses which belong to the predicate `Pred` from the perspective of d-Prolog will belong to the predicates `neg`, `:=`, `:^`, `sup`, and `incompatible` from the perspective of Prolog.

A predicate `rescind/1` is defined in the file DPUTILS.PL which fills this gap. The query

```
?- rescind(Pred/Arity).
```

will remove from memory all d-Prolog clauses which belong to the predicate `Pred` with arity `Arity`.

A user may also want to eliminate all d-Prolog predicates from memory without removing the d-Prolog inference engine itself. The predicate `rescindall/0`, defined in DPUTILS.PL, removes all `dPrologDictionary` clauses and all clauses for predicates recorded in the d-Prolog dictionary. So all d-Prolog clauses loaded into memory using the d-Prolog utility predicates `dload` and `reload` can be removed from memory with a single, simple command.

11.16. FINDING CONTRADICTIONS

An important difference between a Prolog and a d-Prolog program is that a Prolog program can never contain a contradiction while contradictions are possible in d-Prolog programs. The only kind of negation available in Prolog is negation-as-failure represented by the built-in predicate `\+`. The only way a query

```
?- \+ Goal.
```

can succeed is if the query

```
?- Goal.
```

fails. This is, after all, exactly what negation-as-failure means. The clause `\+ Goal` says that `Goal` cannot be proved, not that it is false.

In d-Prolog we have implemented a positive negation operator `neg`. The only way the query

```
?- neg Goal.
```

can succeed is if there is a fact or rule in memory whose head unifies with `neg Goal`. But nothing is simpler than to construct a contradictory program in d-Prolog. A simple example is:

```
p.
neg p.
```

A more interesting example is:

```
p.
q.
incompatible(p,q).
```

The first example is contradictory since it says that the same atom is both true and false. The second is contradictory since it says both that two atoms are true and that they cannot both be true.

A d-Prolog program containing a contradiction must contain some false statements. In general, a d-Prolog developer will not intend to incorporate contradictions into his or her program. They are more likely to occur as a result of the interactions of several rules. While d-Prolog is expressively more powerful because contradictions can be expressed in it, contradictions are in principle a bad thing. So when we have them, we want to be able to find them. Once we find them, we can begin the task of figuring out which of our facts or rules is incorrect. This can be a difficult task, but it is a problem in knowledge acquisition rather than a problem in logic programming. Our job is only to provide a tool to find the contradictions.

The d-Prolog utility `contradictions/0`, defined in DPUTILS.PL, does this for us. In response to the query

```
?- contradictions
```

d-Prolog constructs dummy queries from the predicates found in the d-Prolog dictionary. Then it tries to derive strictly both the query and any contrary of the query. If it succeeds, it informs the user. The d-Prolog dictionary is essential to this process since otherwise there would be no way to generate the test queries.

As it finds contradictions, d-Prolog displays them to the user and stores them in clauses for the predicate `contradictory_pair/2`. These clauses are not removed from memory after the contradictions have been identified; they are left in memory for the convenience of the user. It can take a long time to check a large program for contradictions and it is faster to check the clauses for `contradictory_pairs` than to run the utility again. However, whenever the utility is invoked it removes all clauses for `contradictory_pairs` from memory before testing for contradictions. Thus any contradictions corrected by the user will not show up again.

Notice that besides the contradictory pairs, a great deal more may be displayed when the `contradictions` utility is invoked. d-Prolog will call every query it can construct from the predicates in the d-Prolog dictionary. Any queries that use `write` will produce text on the screen when `contradictions` is used. More importantly, any other input or output operations included in your code will be performed, including writes to files using `tell/1` or changes in the contents of memory using `assert/1` or `retract/1`. So great care should be used with the `contradictions` utility. Still, with care, it can be a valuable tool for testing d-Prolog knowledge bases for consistency.

11.17. A SPECIAL EXPLANATORY FACILITY

A simple Prolog query either contains `\+` or it doesn't. In either case, there is only one possible explanation for why a query failed. If it does not contain `\+`, then it failed because it did not unify with any fact or with the head of any rule whose body succeeded. If it contains `\+`, then it failed because the clause following the `\+` succeeded. Although simple in principle, it is often difficult to work through a complex Prolog program carefully to see how the clauses interact and why a query fails.

With d-Prolog, however, defeasible queries can fail in new and interesting ways. They can fail because no rule supporting them is satisfied, but they can also fail even though some rule supporting them is satisfied because that rule is rebutted or undercut. This is why the rules are called defeasible.

The last of the utilities defined in DPUTILS.PL is the predicate `whynot/1`. Given a query of the form

```
?- whynot(Goal).
```

the utility will respond in one of the following ways.

1. If the query `?- @ Goal` succeeds, `whynot` reports this and stops.

2. If there is no fact that will unify with `Goal` and no rule supporting `Goal` that is satisfiable, `whynot` reports this and stops. The auxiliary predicate `initial_evidence/1` is used here.

3. If there is a strict or defeasible rule supporting `Goal` which is satisfiable, `whynot` displays this rule, finds the competing rule or fact that defeats it, and displays it as well.

It takes several clauses to cover all the cases, but the clauses are not difficult to understand once one understands the d-Prolog inference engine.

11.18. A SUITE OF EXAMPLES

The file KBASES.DPL contain a representative selection of d-Prolog examples that illustrate many of the features discussed in this chapter. You must load DPROLOG.PL before loading this file. Otherwise, Prolog will not understand the operators `:=`, `:^`, `neg`, and `@`, and you will see various error messages. We suggest that you always use `dload` or `reload` to load KBASES.DPL or any other d-Prolog files.

The first predicate defined in KBASES.DPL is `examples/0` which simply prints a list of the examples that have been loaded. This predicate is invoked in a query at the end of the file so the user will immediately see the list of examples as soon as the file is loaded.

The Tweety Triangle, the Chinese Restaurant, the Pennsylvania Dutch, and the Twin Tweety Triangle examples discussed earlier in this chapter are included in KBASES.DPL. Also included are an example of inheritance with exceptions (Naked Nautilus), an example of multiple inheritance (modified Nixon Diamond), an example of defeasible temporal reasoning (Yale Shooting Problem), and a complex example involving several features of defeasible reasoning (Election). We will look at some of these examples in detail.

```
% KBASES.DPL - a sampler of d-Prolog knowledge bases

:- multifile (neg)/1, (:=)/2, (:^)/2.

examples:-
nl,
write('Tweety Triangle is loaded.'), nl,
        write('Chinese Restaurant example is loaded.'), nl,
write('Pennsylvania Dutch example is loaded.'), nl,
        write('Twin Tweety Triangle is loaded.'), nl,
write('Naked nautilus example is loaded.'), nl,
write('Modified Nixon Diamond is loaded.'), nl,
write('Yale Shooting Problem is loaded.'), nl,
write('Election example is loaded.'), nl, nl.

%
% Tweety Triangle, plus some refinements involving
% undercutting defeaters.
%

flies(X):=  bird(X).
neg flies(X):= penguin(X).
neg flies(X):^ sick(X),bird(X).

flies(X) :^ ga_penguin(X).
flies(superman) := true.

bird(tweety).
bird(X):- penguin(X).

penguin(opus).
penguin(X) :- ga_penguin(X).

ga_penguin(folio).

%
% Chinese Restaurant Example - illustrating use of
% the predicate incompatible/2.
%

incompatible(marxist(X),capitalist(X)).

capitalist(X) := owns(X,restaurant).

marxist(X) := born(X,prc).
```

```
owns(ping,restaurant).

born(ping,prc).

%
% Pennsylvania Dutch Example - showing superiority
% of strict rules over defeasible rules.
%

native_speaker(X,german_dialect) :- native_speaker(X,pa_dutch).
native_speaker(hans,pa_dutch).

born(X,pennsylvania) := native_speaker(X,pa_dutch).
born(X,usa) :- born(X,pennsylvania).
neg born(X,usa) := native_speaker(X,german_dialect).

%
% Twin Tweety Triangles - an example of preemption
% of defeaters.
%

republican(X) := conservative(X).
neg republican(X) := southerner(X).
conservative(X) := southerner(X).

republican(X) := rich(X).
neg republican(X) := movie_star(X).
rich(X) := movie_star(X).

southerner(kim).
movie_star(kim).

%
% The Naked Nautilus - an example of inheritance with
% exception.
%

has_shell(X) := mollusc(X).
neg has_shell(X) := cephalopod(X).
has_shell(X) := nautilus(X).
neg has_shell(X) := naked_nautilus(X).

mollusc(X) :- cephalopod(X).
mollusc(fred).

cephalopod(X) :- nautilus(X).
```

```
nautilus(X) :- naked_nautilus(X).

%
% Nixon Diamond - an example of multiple inheritance
% with resolution of the conflict forced by specifying
% superiority of the rule for Republicans.
%

pacifist(X) := quaker(X).
neg pacifist(X) := republican(X).

quaker(nixon).

republican(nixon).

sup((neg pacifist(X) := republican(X)),
         (pacifist(X) := quaker(X))).

%
% The Yale Shooting Problem - an example involving
% temporal persistence.
%

holds(alive,s).
holds(loaded,s).

holds(F,result(E,S)) := holds(F,S).

incompatible(holds(alive,S),holds(dead,S)).

holds(dead,result(shoot,S)) :=
        holds(loaded,S),
        holds(alive,S).

%
% The Election Example - an extended example of
% some complexity.
%

runs(farmer).
runs(hunter).
runs(baker).
runs(crow).
runs(crane).
neg runs(dove):=true.
```

```
backs(hawk,crow).
neg backs(X,Y) :- backs(X,Z),
                     Y \== Z.

neg nominate(_,X) :- neg runs(X).

nominate(free_traders,farmer):=
runs(farmer).
neg nominate(free_traders,farmer):=
        neg backs(butcher,farmer).
nominate(free_traders,hunter):=
runs(hunter),
        neg nominate(free_traders,farmer).
nominate(free_traders,baker):=
runs(baker),
        neg nominate(free_traders,farmer),
        neg nominate(free_traders,hunter).

nominate(isolationists,dove):=
        runs(dove).
nominate(isolationists,crow):=
runs(crow),
        neg runs(dove),
        neg nominate(free_traders,farmer).
nominate(isolationists,crane):=
runs(crane),
        neg nominate(isolationists,dove),
        neg nominate(isolationists,crow).

elected(Candidate1):=
        nominate(free_traders,Candidate1),
        nominate(isolationists,Candidate2),
        backs(hawk,Candidate2).
elected(dove):=
        nominate(isolationists,dove),
        nominate(free_traders,Candidate),
        neg backs(butcher,Candidate).

:- examples.

p.
neg p.
```

11.19. SOME FEATHERED AND NON-FEATHERED FRIENDS

This example contains a number of claims about what normally flies and what normally does not fly. In English, these claims are:

- Normally, birds fly.
- Normally, penguins don't fly.
- A sick bird might not fly.
- A genetically altered penguin might fly.
- Presumably, Superman flies.

We also have the following information about what are penguins and what are birds:

- Tweety is a bird.
- All penguins are birds.
- Opus is a penguin.
- All genetically altered penguins are penguins.
- Folio is a genetically altered penguin.

With this example in memory, we conduct the following dialog with d-Prolog:

```
?- @ flies(X).

X = tweety ->;

X = superman ->;

no
?- @ neg flies(X).

X = opus ->;

no
?- assert(sick(tweety)).

yes
?- assert(sick(superman)).

yes
?- @ flies(X).

X = superman ->;
```

```
no
?- @@ flies(opus).
presumably, no.

yes
?- @@ flies(folio).
can draw no conclusion.

yes
?-
```

Notice that when we tell d-Prolog that Tweety and Superman are sick, it no longer concludes that Tweety can fly although it continues to conclude that Superman can fly. This is because the undercutting defeater only applies to sick birds. Since the system cannot infer that Superman is a bird, it does not use the defeater in Superman's case. Notice also that the system can draw no conclusion about Folio. The rule for genetically altered penguins undercuts the rule for penguins, but the undercut rule can still rebut the rule for birds.

11.20. INHERITANCE REASONING

A kind of reasoning often found in expert systems is *inheritance* reasoning. Our genetically altered penguin is an example of inheritance reasoning. Since genetically altered penguins are penguins and penguins are birds, genetically altered penguins *inherit* characteristics from both penguins and birds. Two common problems with inheritance reasoning are exceptions and multiple inheritance. Additional examples of inheritance reasoning illustrating these two problems and how they might be resolved in d-Prolog are included in KBASES.DPL.

Molluscs are a large group of sea creatures including oysters and clams. Squid and octopuses make up a subcategory of molluscs called cephalopods. A nautilus is a kind of cephalopod. It looks like a squid with a shell. The shell might be long and spiraling or coiled like a typical snail shell. For the example, we invent a special kind of nautilus which has lost its shell. We call this creature a *naked nautilus.* An interesting feature of this sequence of groups and subgroups is that molluscs and nautili normally have shells while cephalopods and naked nautili normally do not. The situation is represented graphically in Figure 11.20 and in d-Prolog by the clauses in the Naked Nautilus example in KBASES.DPL.

We start with the information that Fred is a mollusc. We subsequently learn that he is a cephalopod, then that he is a nautilus, and finally that he is a naked nautilus. The following dialog with d-Prolog shows how these disclosures should affect our beliefs about whether Fred has a shell.

```
?- @@ has_shell(fred).
presumably, yes.

yes
```

Figure 11.7 The Naked Nautilus

```
?- assert(cephalopod(fred)).

yes
?- @@ has_shell(fred).
presumably, no.

yes.
?- assert(nautilus(fred)).

yes
?- @@ has_shell(fred).
presumably, yes.

yes
?- assert(naked_nautilus(fred)).

yes
?- @@ has_shell(fred).
presumably, no.

yes
```

The Naked Nautilus is an example where we have a linear hierarchy of groups with each group directly inheriting from its immediate supergroup. We can think of properties flowing from each group to its subgroups. Multiple inheritance, on the other hand, involves cases where the inheritance relations between the groups and individuals form a nonlinear structure. An example is the Nixon Diamond illustrated graphically in Figure 11.20. Here the individual Nixon inherits from two groups, Republicans and Quakers, and neither group is more specific than the other. Furthermore, members of one group normally have a characteristic that members of the other group normally do not. So Nixon inherits from multiple groups with incompatible characteristics. In general, we can draw no conclusions in a Nixon Diamond since the two inheritance rules rebut each other. But in d-Prolog we can force resolution of the conflict by specifying with a clause for the predicate `sup` which of the two rules should dominate. Notice that resolving the conflict in this way does not involve a claim that Republicans are normally Quakers or that Quakers are normally Republicans. Neither rule becomes more specific than the other. Instead, stipulating superiority of the Republican rule simply declares that for purposes of determining whether a person is a pacifist, being a Republican counts for more than being a Quaker. For another case where Republicans and Quakers tend to differ in their characteristics, we might specify that being a Quaker is more important or we might leave the issue unresolved.

Figure 11.8 The Nixon Diamond

11.21. TEMPORAL PERSISTENCE

Things tend to remain the same and yet eventually everything changes. We expect to find our car where we parked it, to find our home where we left it, to find our friend's hair the same color as the last time we saw it. Yet we also believe that if we abandoned our car it would eventually be moved, that our house will someday be demolished, and that our friend's hair will turn gray or even disappear over the course of the years. The tendency things have to stay the same we will call *temporal persistence*. This is a kind of inertia which sooner or later will be overcome by the forces of change.

Temporal persistence is a very important feature of the world, one that we rely upon routinely. Suppose you want a green chair in your bedroom and you have a green chair in your living room. You might try to achieve your goal by moving the chair from the living room to the bedroom. But how can you infer that this action will accomplish your goal? How do you know that after you move the chair it will still be green? Why do you even believe that it will still be a chair? More to the point from the perspective of artificial intelligence, how can a machine infer this? It may have a rule that says that after something is moved from location A to location B, it is in location B in the resulting situation. But it seems the system must also have a rule that says that green things remain green and chairs remain chairs when they are moved. And other rules that says chairs remain chairs and things maintain their locations when they are painted. And so on. It would appear that we will need more rules telling us what does *not* change as the result of some event than we do to tell us what changes.

This problem, which is a problem of knowledge representation, was dubbed the *Frame Problem* by McCarthy and Hayes (1969). Specifically, the problem is to represent the information a system needs to make all the necessary inferences about what remains unchanged when some event occurs. The kinds of rules we have mentioned which state that some feature of a situation remains unchanged after some event are called *frame axioms*. The point that Hayes and McCarthy make is that it looks like we will need lots and lots of frame axioms, probably more than all our other rules combined.

A proposal to solve the Frame Problem is to formulate one very general temporal persistence principle that says, essentially, that everything tends to stay the same. Then, unless we can show that something would change as a consequence of some event having occurred, we are justified in inferring that it has not changed. We can formulate such a principle using McDermott's (1982) Situation Calculus together with d-Prolog.

In the Situation Calculus situations are indicated by constants or as the situations that result from the occurrence of certain events in other situations. For example, we could indicate an initial situation by the constant `s`. If some event `e1` occurs at `s`, we would represent the resulting situation by `result(e1,s)`. If in this situation another event `e2` were to occur, the resulting situation would be represented by `result(e2,result(e1,s))`.

Besides representing situations and the events that produce them, we also want to represent what holds in different situations. In the initial situation `s` of the last paragraph, for example, it may be the case that a particular chair is green and that it is in the living room. We could represent this information by the following clauses:

```
holds(chair(c),s).
holds(green(c),s).
holds(in(c,living_room),s).
```

Now suppose the event e1 is the event of the chair being moved from the living room to the bedroom. Then we believe that in the resulting situation, the chair is still a chair and is still green, but it is no longer in the living room.

```
holds(chair(c),result(e1,s)).
holds(green(c),result(e1,s)).
holds(neg in(c,living_room),result(e1,s)).
```

We now have enough machinery to formulate our temporal persistence principle. It will be a defeasible rule containing variables indicating that it applies to all situations, events, and fluents (possible states that may or may not hold in a situation):

```
holds(F,result(E,S)) := holds(F,S).
```

Basically, this rule says that if some fact F holds in situation S, then presumably it also holds after event E occurs. This single principle should be the only frame axiom we need since it applies to all fluents, all events, and all situations.

This solution to the Frame Problem, or something very much like it, was proposed by McDermott (1987). But he later decided it didn't work because of some very simple examples that did not easily submit to this treatment using the nonmonotonic formalisms with which McDermott was familiar. One famous example constructed by Hanks and McDermott (1987) has come to be called the Yale Shooting Problem. We include a version of the YSP in KBASES.DPL which is simplified but which nevertheless captures the problematic features of the original.

Our version of the YSP concerns Frankie and Johnnie, lovers of ballad fame. Frankie shot Johnnie for his infidelity. We come in late in the story. Frankie enters Johnnie's empty room with a loaded gun. She waits until he returns, then points the gun at his head and pulls the trigger. Intuitively, we do not expect Johnnie to survive this story. How do we represent the story in d-Prolog so the system can infer the expected outcome? Like this:

```
holds(alive,s).
holds(loaded,s).

holds(F,result(E,S)) := holds(F,S).

incompatible(holds(alive,S),holds(dead,S)).

holds(dead,result(shoot,S)) :=
        holds(loaded,S),
        holds(alive,S).
```

The first two clauses say that in the initial situation, s, Johnnie is alive and Frankie's gun is loaded. The next is our temporal persistence principle. The fourth says that Johnnie can't be both alive and dead in the same situation. The final clause is a causal rule that says that normally Johnnie's status would be changed from alive to dead if a loaded gun were fired at his head. In the brief act of the story that we witness only two events occur: Frankie waits until Johnnie arrives and then she fires the gun at him. The crucial query, then, is

```
?- @@ holds(alive,result(shoot,result(wait,s))).
```

d-Prolog gives the correct answer: presumably, Johnnie is not alive at the end of the story.

Notice that it is important in the last clause of the example that `holds(alive,S)` is a condition. Otherwise, this defeasible rule would not be more specific than the temporal persistence principle in the appropriate instance and the two rules would rebut each other. It is reasonable to include this condition since we are trying to represent a causal rule. Causal rules tell us how events tend to change things. So the condition of the rule naturally includes the feature of the situation which will be changed. This is all that is needed for the solution to work.

Why did McDermott's nonmonotonic logic and the other nonmonotonic formalisms with which he was familiar fail to resolve the example so simply? Because these systems allow *multiple extensions* of a knowledge base. Basically, an extension is any completion of the knowledge base that violates a minimum number of the defeasible rules in the knowledge base. In the YSP, there are at least three ways we can complete the story that only violate one defeasible rule.

1. Violate the persistence principle going from the first situation to the second so the gun is no longer loaded when Johnnie arrives.

2. Violate the causal principle going from the second situation to the last so Johnnie is still alive at the end of the story.

3. Violate the persistence principle going from the second situation to the last so Johnnie is dead at the end of the story.

Thus, the story has multiple extensions. In the first two extensions, Johnnie survives. Why do we prefer the third extension, the only one that d-Prolog supports? Because we have a *reason* to reject the persistence principle in this case: it conflicts with the more specific causal principle.

11.22. THE ELECTION EXAMPLE

The object of the Election Example is to deduce who will represent two political parties in a presidential campaign and who will in fact be elected. Farmer, Hunter, and Baker seek the nomination of the nomination of the Free Trade Party; Crow and Crane seek the nomination of the Isolationist Party. Dove is an Isolationist who has been discussed as a potential candidate, but it is presumed that she will not run. Hawk, a highly-visible but discredited Isolationist, backs Crow. We stipulate no one supports two different candidates, and we presume of any individual that he or she will not be nominated. We represent this information with the following clauses:

```
runs(farmer).
runs(hunter).
runs(baker).
runs(crow).
runs(crane).
```

```
neg runs(dove):=true.

backs(hawk,crow).
neg backs(X,Y) :- backs(X,Z),
                  Y \== Z.

neg nominate(_,Person) := neg runs(Person).
```

It is presumed that Farmer will represent the Free Traders in the election (if he runs as we know he will), although he cannot get the nomination without Butcher's support. If Farmer should for some reason not get the nomination, Hunter is expected to get it. And if both Farmer and Hunter fail, the nomination is expected to go to Baker.

```
nominate(free_traders,farmer):=
        runs(farmer).
neg nominate(free_traders,farmer):=
        neg backs(butcher,farmer).
nominate(free_traders,hunter):=
        runs(hunter),
        neg nominate(free_traders,farmer).
nominate(free_traders,baker):=
        runs(baker),
        neg nominate(free_traders,farmer),
        neg nominate(free_traders,hunter).
```

The situation is murkier for the Isolationists. Presumably, Dove would get the nomination if she ran. But Crow will apparently get the nomination if, as expected, Dove does not run, and if the Free Traders do not nominate Farmer. If for some reason neither Crow nor Dove is nominated by the Isolationists, then the nomination will go to Crane.

```
nominate(isolationists,dove):=
        runs(dove).
nominate(isolationists,crow):=
        runs(crow),
        neg runs(dove),
        neg nominate(free_traders,farmer).
nominate(isolationists,crane):=
        runs(crane),
        neg nominate(isolationists,dove),
        neg nominate(isolationists,crow).
```

We expect that the Free Trader candidate will win the election if Hawk, a well-known but discredited Isolationist, backs the Isolationist candidate. We might call this the 'kiss of death' rule. Second, we expect Dove to win if she is nominated and if the Free Trader candidate does not have Butcher's support. This is as much as we can predict regarding the actual outcome of the election.

```
elected(Candidate1):=
        nominate(free_traders,Candidate1),
        nominate(isolationists,Candidate2),
        backs(hawk,Candidate2).
elected(dove):=
        nominate(isolationists,dove),
        nominate(free_traders,Candidate),
        neg backs(butcher,Candidate).
```

With this information, d-Prolog can deduce that apparently Farmer will receive the Free Trader nomination, but it can reach no conclusion about either the Isolationist nominee or the eventual winner in the election. If we add the information that Butcher backs Baker, d-Prolog concludes that apparently Hunter will receive the Free Trader nomination, Crow will receive the Isolationist nomination, and Hunter will be elected. If we add the further information that, contrary to expectation, Dove decides to run, then d-Prolog concludes that apparently the Free Traders will nominate Hunter, the Isolationists will nominate Dove, and the country will elect Dove. The corresponding dialog with d-Prolog looks like this:

```
?- @ nominate(Party,Candidate).

Party = free_traders
Candidate = farmer ->;

no
?- @ elected(President).

no
?- assert(backs(butcher,baker)).

yes
?- @ nominate(Party,Candidate).

Party = free_traders
Candidate = hunter ->;

Party = free_traders
Candidate = hunter ->;

Party = isolationists
Candidate = crow ->;

Party = isolationists
Candidate = crow ->;

no
?- @ elected(President).
```

```
President = hunter ->;

President = hunter ->;

President = hunter ->;

President = hunter ->;

no
?- assert(runs(dove)).

yes
?- @ nominate(Party,Candidate).

Party = free_traders
Candidate = hunter ->;

Party = free_traders
Candidate = hunter ->;

Party = isolationists
Candidate = dove ->;

no
?- @ elected(President).

President = dove ->;

President = dove ->;

President = dove ->;

President = dove ->;

no
```

Careful examination of the information represented in the example should convince you that these are the correct conclusions. However, it is not a simple task to sort through the information and reach the correct conclusions. This example shows d-Prolog's ability to handle complex combinations of interacting rules.

Exercise 11.22.1

Notice that after `backs(butcher,baker)` is asserted, d-Prolog offers the solution

```
Party = free_traders
Candidate = hunter
```

twice in response to the query

```
?- @ nominate(Party,Candidate).
```

Explain this. If one clause is deleted from the definition of the predicate `def_der/2`, this solution is not repeated. Which clause can we eliminate to accomplish this? Despite the fact that it sometimes produces duplicate queries, we do not want to delete this clause. Why not?

Exercise 11.22.2

When we assert clauses indicating that Butcher supports Baker, Hawk supports Dove, and Dove is running, d-Prolog deduces that both Hunter and Dove are elected President. We can avoid this by adding a clause for the predicate `incompatible/2` which has a condition using the predicate `runs/1`. When done properly, d-Prolog can no longer deduce who will be elected with the information indicated here. What will the new clause for `incompatible` be?

11.23. D-PROLOG AND THE CLOSED WORLD ASSUMPTION

In working with databases, we often make the Closed World Assumption. The CWA is the assumption that if some piece of data were correct, it would be included in the database. For example, consider a database in Prolog containing information about employees of some company. The database will include such information as employees' names, social security numbers, position, salary, birth date, etc. Suppose we want to know whether Jones is a secretary. When we query the database

```
?- position(jones,secretary).
```

we get the response `no`. We conclude that Jones is not a secretary since, if he were, that fact would have been in the database and the query would have succeeded.

Consider another example. We want to know all employees who are not secretaries and who make less than $25,000 per year. The appropriate query is

```
?- salary(X,Y), Y < 25000, \+ position(X,secretary).
```

If the subgoal `\+ position(X,secretary).` succeeds, then `X` must not be a secretary. But this is only so if the CWA is correct for the predicate `position/2`. Notice that the variable `X` is bound by the time the subgoal containing `\+` is called. This is important since otherwise the subgoal will only succeed if no one in the company is a secretary.

Negation-as-failure is only appropriate where the CWA is justified. But when the CWA is justified, we can use `\+` in Prolog to invoke it. This would seem to present a problem for d-Prolog. The CWA is often very useful, but the d-Prolog inference engine will not interpret `\+` correctly. The solution to this problem is simple, and in fact d-Prolog gives us additional control over when we do and when we do not make the Closed World Assumption. In our example, we invoke the CWA for the predicate `position/2` by including in our database the presumption

```
neg position(X,Y) := true.
```

The presumption, then, is that nobody has *any* position in the company. Of course, presumptions are always overriden by facts. They are also overriden by any competing defeasible rule whose body is something other than `true`. So the presumption that nobody has any position in the company is overriden by any positive evidence whatsoever that some individual holds some position in the company. We can then rephrase our earlier query about employees making less than $25,000 who are not secretaries as

```
?- @ salary(X,Y), Y < 25000, neg position(X,secretary).
```

In some cases, we may not be justified in making the CWA. Then we will not add the corresponding presumption. Suppose, for example, the senior partner in our fictional company has entered data in the database concerning whether the other employees like chocolates. Her reason for doing this is to help her keep track of what to give the other employees for Christmas. If they like chocolates, that's what she gives them. If they don't like chocolates, she gives them something else. If she doesn't know, she tries to work chocolates into the conversation and find out. But she would not *assume* that someone does not like chocolates just because the database doesn't include the information that they *do* like chocolates. That's just not the kind of information one can assume would be included. The partner will have to enter an explicit fact to show that a particular employee does not like chocolate.

The Closed World Assumption example in KBASES.DPL includes information about positions, salaries, and attitudes toward chocolates for all the people working in the small dental clinic Dewey, Pullem, and Howe. The CWA is made only for the predicate `position`.

Exercise 11.23.1

(Project) Write a set of Prolog rules, defeasible rules, and defeaters for a predicate `prescribe/2` that captures the following principles for prescribing medication.

1. Normally, prescribe a medication for a diagnosed illness if the medication is effective in treating that illness.
2. Normally, don't prescribe a medication for a diagnosed illness if there are counter-indications (allergies, etc.).
3. Normally, prescribe a medication for a diagnosed illness even if the medication is counter-indicated if the medication is effective in treating the illness and the patient's condition is critical (the patient will die if not treated).
4. Never prescribe a counter-indicated medication for an illness if there is another medication that is effective in treating the illness and is not counter-indicated.

Exercise 11.23.2

(Project) Rewrite the CONMAN knowledge base in the file MDC.PL as a set of Prolog rules, defeasible rules, and defeaters for the predicate `diagnosis/1`. Use the prescription rules from the preceding exercise as part of your knowledge base.

11.24. BIBLIOGRAPHICAL NOTES

There is a growing literature on non-monotonic and defeasible reasoning. (Ginsberg 1987) is a collection of early papers, and (Gabbay and Hogger 1994) contains many recent papers. The inference engine d-Prolog is based on the theoretical work reported in (Nute 1993, 1994).