Definitions and inferences

Much human intelligence consists of conclusions or *inferences* drawn from facts, instead of facts themselves. We'll now show how inference methods can be represented, including a famous one called *inheritance*. That is, we'll formalize chains of reasoning.

Rules for definitions

If we ask a lot of queries of a database, especially queries about related things, we may be typing the same expressions over and over again. Nearly all programming languages have subroutines or procedures to modularize knowledge, group it into its natural pieces. Prolog has them too, and they're called *rules* or *definitions*. Rules are a way to create the ability to query new predicates without specifying new facts, by defining the new predicate names in terms of old predicate names.

Prolog rules have a *left side* and a *right side*, and the symbol **:-** in between (it is supposed to look like a backward arrow, but you need a good imagination). Read the symbol as "if". To its left side is a single predicate expression, usually with variables as arguments. To its right side is a query, with possibly multiple predicate expressions combined with the comma ("and"), semicolon ("or") and **not** symbols. For instance:

```
gray_enterprise :- part_of(enterprise,X), color(X,gray).
```

This says that predicate expression **gray_enterprise** succeeds if the query

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

succeeds. In other words, **gray_enterprise** is true if the Enterprise is part of something that is gray in color. So **gray_enterprise** is a code representing a query.

More formally, the **gray_enterprise** rule defines a new predicate of no arguments called **gray_enterprise**. When queried:

```
?- gray_enterprise.
```

the interpreter succeeds (answers **yes**) if it can succeed in querying the right side of the **gray_enterprise** rule. Otherwise the interpreter fails (answers **no**). It's like the right side of the rule is substituted for the left side whenever it occurs in a query. The variable **X** is a *local variable* in the rule, a variable whose value will be "thrown away" when the rule is done, and will not be printed out in any query answer. Any Prolog variable that appears only on the right side of a rule is called local to that rule.

But **gray_enterprise** isn't too useful because it only covers one color. What we really want is a rule that will *tell us* what color something is. We can do this with a variable on the left side of the rule, a variable representing the color:

```
color_enterprise(C) :- part_of(enterprise,X), color(X,C).
```

C is a *parameter variable* of the rule, and a value for it will be "returned" as a result of executing the rule. So if we query

```
?- color_enterprise(C).
```

when our database is that of Section 3.1 and the interpreter uses the **color_enterprise** rule, it will type

```
C=gray
```

in reply. Rules can have any number of such parameter variables, as well as constants (ordinary words and numbers), as arguments on their left sides. So if we wanted an even more general color rule, one that would tell us the color for many objects, we could say:

```
color_object(X,C) :- part_of(X,Y), color(Y,C).
```

Here **X** and **C** are parameter variables, and **Y** is a local variable. So values for **X** and **C** only will be returned.

Figure 4-1 summarizes our terminology about rules. You can think of local variables as being existentially quantified ("there exists some X such that something is true"), and parameter variables as universally quantified ("for every X something is true")--see Appendix A for more about quantification, an important concept in logic. Actually, the distinction of "local" from "parameter" variables is misleading, because the opposite of "local" is "global", and there aren't any true "global" variables in Prolog. Rules just give a shorthand for queries, and a variable **X** is one query is always different from a variable **X** in a separate query. The only way to get anything like a global variable in Prolog is to assert a fact.

Here are more rule examples. This defines X to be the son of Y if X is the child of Y and X is male:

```
son(X,Y) :- child(X,Y), male(X).
```

This defines X to be the "superboss" of Y if X is the boss of the boss of Y:

```
superboss(X,Y) :- boss(X,Z), boss(Z,Y).
```

This defines something X you own to be stolen if it isn't present and you haven't given it to someone:

```
stolen(X) :- owns(you,X), not(present(X)), not(given(you,X,Y)).
```

This defines the previous example predicate **color_object** in a different way, using **a_kind_of** instead of **part_of**. It says that the color of X is C if X is a type of Y and the color of Y is C.

```
color_object(X,C) :- a_kind_of(X,Y), color(Y,C).
```

Usually predicates defined by rules can be used in queries just like fact predicates. This means that we can usually make variables any of the arguments to defined-predicate expressions in queries, a powerful feature. (A few exceptions to this will be discussed in later chapters: some rules with **not**s, the arithmetic and **is** constructs of Chapter 5, and the "cut" ("**!**") of Chapter 10.) That is, we can designate arbitrary inputs and outputs. This means that Prolog is fundamentally more powerful than conventional programming languages, which usually require all but one argument (the output variable) to a procedure to be filled in at the time of the call. But Prolog rules, the Prolog equivalent of procedures, rarely impose such requirements.

Rule and fact order

Rules can go into a Prolog database just like facts. And the interpreter then uses both those rules and facts to answer queries. But we must worry a little about rule and fact order.

To see why, consider combining several reasoning methods in the same rule, as for instance three ways of establishing the color of something:

```
color_object(X,C) :- color(X,C); (part_of(X,Y), color(Y,C));
(part_of(X,Y), part_of(Y,Z), color(Z,C)).
```

(Queries and rules can take more than one line; the period at the end indicates where they stop.) But that's poor rule-writing style, since it's hard to read the right side. Instead, we can write three separate rules with the same left side:

```
color_object(X,C) :- color(X,C).
color_object(X,C) :- part_of(X,Y), color(Y,C).
color_object(X,C) :- part_of(X,Y), part_of(Y,Z), color(Z,C).
```

Now each rule's right side gives *sufficient* conditions for the left-side predicate expression to be true, but not *necessary* conditions. That is, each describes some but not all the situations for which **color_object** succeeds. When each rule is very specific, we have *definition by examples*.

The order of those three rules matters. Whenever a query on predicate **color_object** with two arguments is issued, the rules will be tried to answer the query in database order, just as if they were facts. The order shown is probably the best, because the simplest rule will be tried first, then the next-simplest, then the most complex (where we measure "complexity" by the number of expressions).

Facts and rules can be freely intermingled in a Prolog database; the overall order is all that matters. Facts should generally come first, though, if there are both facts and rules for a predicate name. That's because facts require less symbol matching than rules--and a lot less than some of the recursive rules we'll discuss later in this chapter. Putting facts before recursive rules of the same predicate is also the standard way to write recursive programs (see Appendix B).

Rules as programs

So this is how we'll write a Prolog program: by giving a list of facts and rules (definitions) that explain about some area of human knowledge. To match a query, rules and facts will be considered in order, somewhat like sequential execution of the lines of a program in a conventional computer language. "And"s on the right sides of queries will be initially be done in left-to-right order, like multiple procedure calls in a line of a conventional language. And like those languages, we can have "procedure" (rule) hierarchies; that is, a rule right side can use predicate names defined in rules, including its own left-side predicate name (the last is called *recursion*).

This mapping of predicate expressions to actions means that we can model *procedures* in Prolog, not just facts. For instance, here's a way to describe a daily agenda for a student:

```
day_agenda :- wakeup, classes, lunch, classes, dinner, study, sleep.
wakeup :- late, dress, travel(home,campus).
wakeup :- not(late), shower, dress, breakfast, travel(home,campus).
classes :- class, class, class.
class :- check_schedule, go_room, take_notes.
```

This notation is useful for describing processes, even if we never define what the basic actions (like **take_notes**) are. A computer doesn't need to know *everything* to be intelligent, just the important things.

(However, we'll introduce a better way to describe sequences of actions in Chapter 9.)

Rules in natural language

As we said in Section 2.8, an artificial-intelligence system is often built from natural-language (e.g. English) specifications, either oral or written. Rules can be specified several ways in English. The easiest to spot is the "if...then" statement:

> If a vehicle floats on water, then it's a ship.

which could become the Prolog rule (note the reversed order):

```
ship(X) :- vehicle(X), floats(X,water).
```

But the "then" is often omitted:

> If a vehicle floats on water, it's a ship.

"Define" or "assume" often signals a rule, taking the things in the opposite (Prolog) order:

> Define a ship as anything that floats on water.
> Assume as a ship anything that floats on water.

"Anything", "anyone", "any", "something", "some", and "a" in such definitions often map to Prolog variables.

Besides expressing facts, the verb "to be" can express rules:

> A ship is any vehicle that floats on water.
> Ships are water-floating vehicles.

The borderline between facts and rules can be fuzzy, but generally speaking, use a rule when variable bindings seem possible. The "a" and "any" in the first sentence suggest a variable, and hence a rule.

When an "and" occurs in the "if" part of a definition, we can just put commas into the right side of the rule. For instance

> Something that is a vehicle and floats on water is a ship.

takes the form of the preceding **ship(X)** rule. If an "and" occurs in the "then" part of a definition, multiple conclusions hold for some situation. Prolog doesn't allow "and"s in rule left sides, but we can write multiple rules with the same right side and different left sides. So

> If a vehicle floats on water and is gray, then it is a ship and of military origin.

becomes

```
ship(X) :- vehicle(X), floats(X,water), gray(X).
origin(X,military) :- vehicle(X), floats(X,water), gray(X).
```

Rules without right sides

The right side of a rule represents sufficient conditions for the left-side predicate expression to be true. What if a rule doesn't have anything on its right side? Then we'd be saying that nothing is necessary to make the left side true, that the left side always true. In other words, a fact. So facts are just a special case of rules. That's why Prolog facts and rules are put together in one big database: they're the same thing.

But rules can have variables. What would it mean for a fact to have a variable? Consider:

```
part_of(X,universe).
```

If you think of that as a rule with no right side, it says that for any **X**, nothing is necessary to say that **X** is part of the universe. In other words, every **X** is part of the universe. So using a term from logic (see Appendix A), facts with variables are *universally quantified*.

Postponed binding

An interesting consequence of the Prolog interpreter's handling of rules is "postponed" binding of variables. This is interesting because most other programming languages can't do anything like this. Suppose we query:

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

Here the first argument is bound (that is, it's an input) and the second argument is unbound (that is, it's an output). If there's a fact in the database

```
color_object(enterprise,gray).
```

then the query can immediately bind **C** to **gray**. But if there are no **color_object** facts, only rules with **color_object** on their left sides, or if a rule is first in the database, the binding may be delayed. For instance, suppose the interpreter picks the rule

```
color_object(X,C) :- part_of(X,Y), color_object(Y,C).
```

The variable **C** won't be bound when the rule is invoked, and it won't be bound in the **part_of** (which doesn't mention **C**), so if it's ever bound it won't be until the recursive query of **color_object**. But this query may require other recursive calls. It may take a long time for an appropriate **color_object** fact to be found, and hence a long time for variable **C** to get bound.

In fact, some query variables may *never* get bound to values in successful queries. Consider:

```
recommended(X,J) :- not(bad_report(X)).
```

a rule that recommends person **X** for job **J** if they haven't had a bad report recently. Here **J** is never bound if the query's second argument isn't initially bound. Most Prolog dialects will invent a name like "**_14**" for such unbound variables, to print out if they must.

Postponing of binding in Prolog interpreters has the advantage that binding is only done when truly necessary to answer a query. This saves on the computer's overhead cost of binding, for one thing. Furthermore, *multiway* reasoning becomes easier, reasoning for which we don't know in advance which arguments to a predicate expression (procedure) will be bound (or be inputs). Multiway reasoning means the same rule can be used many ways. We'll look more into this in the next chapter.

Backtracking with rules

As you remember from Chapter 3, backtracking is the "backing up" the interpreter does when it can't find a match for a predicate expression in a query. When predicates in a query have rule definitions, the interpreter acts somewhat as if the right side of the definition were typed as part of a query instead of the single predicate expression that is the left side. (We say "somewhat" because variables in a rule are local in meaning, and parameter variables must get bound.) So backtracking into a predicate expression for a rule-defined predicate means returning to the last expression in the rule. Tracing backtracking with rules is hard because we move both left and right (through backtracking) and up and down (through procedure calls and returns). This is a common problem with powerful computer languages such as Prolog: simply because they're powerful, it's hard to follow everything they're doing.

Here's an example that isn't too hard. Suppose we have two kinds of facts about an organization: the department each employee works for and the manager of each department. Suppose we define a **boss** predicate of two parameter arguments **B** and **E**, that says that **B** is the boss of **E** if **B** manages a department **D** in which **E** is an employee; **D** will be a local variable. Assume that Tom works in the sales department, Harry works in the production department, Dick manages the sales department, and Mary manages the production department. Then the Prolog database is:

```
department(tom,sales).
department(harry,production).
manager(dick,sales).
manager(mary,production).
boss(B,E) :- department(E,D), manager(B,D).
```

Now suppose we want to find a boss different from Tom's. It will be **X** in the query:

```
?- boss(X,Y), not(boss(X,tom)).
```

Notice that the **not** must come second in the query because **X** must be bound.

Let us trace execution (see Figure 4-2).

1. The first predicate expression in the query matches only a rule in the database, no facts, so the first job to do is searching for a match for the first expression on the right side of the rule, **department(E,D)**. This can be matched to the first fact, with **E=tom** and **D=sales**.

2. Moving on to the second predicate expression in the rule, **manager(B,D)**, the interpreter finds a match in the third fact with **B=dick**, so the rule succeeds and the first expression in the original query succeeds. So **X=dick** and **Y=tom**.

3. Moving to the second and last expression in the query, **not(boss(X,tom))**, the interpreter tries to find if Dick is the boss of Tom. It has no memory of what it just proved, so it goes back to the rule.

4. Again, both predicate expressions on the right side of the rule can match the same facts, so the rule succeeds. But the condition in the original query is the opposite ("not" or negation) of this, so the second half of the original query fails and the interpreter backtracks to the first expression. (There's never a new way to satisfy a **not** when it fails.)

5. Backtracking into a predicate expression satisfied previously by a rule means that the interpreter must go to the last (rightmost) expression on the right side of the rule and see if it can find another match there. But there is no other boss of the sales department so it must now backtrack to the first expression of the rule, **department(E,D)** with unbound **B** and **E**.

6. Fortunately for this there is another choice: **E=harry** and **D=production**.

7. With this success, the interpreter can start moving right again. It considers the second predicate expression in the rule. And it can find a match of **B=mary** (remember, **D=production** now). So the rule succeeds, and thus the first expression of the query succeeds with **X=mary** and **Y=harry**.

8. Now in the second query expression, it must check that it is not true that Mary is the boss of Tom. To do this, it tries to prove the **boss** rule fails with **B=mary** and second argument **tom**.

9. In the first expression on the right side of the rule, it can match **D** to sales, but there is no fact that Mary manages the sales department. So the rule fails, and since there is no other rule or fact for the **boss** predicate, the second **boss** expression in the query fails.

10. But since there's a **not** in front of this expression, the whole query succeeds. So **X=mary** is the answer we needed.

To help in debugging, most Prolog interpreters will automatically print out abbreviated trace information if you ask. To ask in most Prolog dialects, query the built-in predicate **trace** of no arguments. To stop the tracing, query the built-in predicate **notrace** of no arguments. Some other debugging facilities of most Prolog dialects are described in Appendix D.

Transitivity inferences

Certain rule forms occur frequently in artificial intelligence. A very important form states *transitivity* of a two-argument predicate. For instance, consider bosses in an organization. If your boss has a boss in turn, that big boss is your boss too. If that big boss has a boss in turn, that even bigger boss is your boss too. So bossing relationships form chains, and that's transitivity.

Formally, a relationship predicate **r** is transitive if this rule is correct:

```
r(X,Y) :- r(X,Z), r(Z,Y).
```

(See Figure 4-3.) This says that if predicate **r** holds from some **X** to some **Z**, and also from **Z** to some **Y**, the predicate always also holds from **X** to **Y**. This rule can be used recursively too; that is, it can refer to itself on its right side, not just to **r** facts. So the rule can follow indefinitely relationship long chains. For instance, suppose these facts are placed in front of the rule in the Prolog database:

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

Then if we query

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

no facts match, so the interpreter will use the rule, and will first query

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

For this, **Z** can match **b** in the first fact. The interpreter will then query

```
?- r(b,d).
```

There's no fact stating this either, so it must use the rule again recursively. For this new call of the rule, **X=b** and **Y=d**, so the next query is

```
?- r(b,Z).
```

This new **Z** is different from the previous **Z**, since each recursive call has its own variables (see Appendix B), in the same way that **X**'s in different rules represent different **X**'s. For this new query, the interpreter can bind **Z** to **c** since there is a **r(b,c)** fact, and then the second part of the rule becomes the query:

```
?- r(c,d).
```

That predicate expression is a fact. So the rule succeeds in proving **r(b,d)**. And thus it succeeds in proving **r(a,d)**, the original query.

Many of the relationship predicates in Section 2.6 are transitive: **a_kind_of**, **part_of**, **right_of**, **during**, and **ancestor**, for instance. Some example applications:

--If the Vinson is a kind of carrier, and the carrier is a kind of ship, then the Vinson is a kind of ship.

--If the electrical system is part of the car, and the battery is part of the electrical system, then the battery is part of the car.

--If the Vinson is north of the Enterprise, and the Enterprise is north of the Kennedy, then the Vinson is north of the Kennedy.

--If during the day Monday you had a meeting with your boss, and during the meeting you found out you got promoted, then during the day Monday you found out you got promoted.

--If a number X is greater than a number Y, and Y is greater than a number Z, then X is greater than Z.

And a graphical example is shown in Figure 4-4. Here we are representing facts about a pile of blocks on a table. A block is above another block if it is resting on that block. Block b is above block a, block c is above block b, and block d is above block c. Hence by transitivity, block d is above block a.

Why is transitivity important? Because it can save fact space, by figuring out "indirect" facts from a few direct ones. So it reduces redundancy. Transitivity will work best when we store facts relating only the "closest" things--like for **part_of**, the very smallest thing B that still contains thing A. That's because transitivity explains relationships between things farther apart from the same relationships between things closer together, not the other way around.

Inheritance inferences

An even more important rule form for artificial intelligence is the *inheritance* form. Consider a bureaucratic organization. If it has only one business address, then that is the business address of all the employees. It wouldn't make sense for a computer to keep a separate business address fact for each employee; that would mean a lot of unnecessary facts. Instead it should store a single address fact with the name of the organization, and reason from that. This reasoning is called inheritance; we say the address *inherits* from organization to employee.

Inheritance always involves two predicates, a property predicate and a relationship predicate. Formally, we say property predicate **p** inherits with respect to relationship predicate **r** if this rule is correct:

```
p(X,Value) :- r(X,Y), p(Y,Value).
```

(See Figure 4-5.) That is, we can prove that property **p** of **X** has value Value if we can prove the **Y** is related to **X** by predicate **r**, and **Y** does have value Value for property **p**. (This generalizes the rule for predicate **color_object** in Section 4.1.) Sometimes we use the term "inheritance" when **p** is a relationship predicate too. Like the transitivity rule, the inheritance rule can be used recursively--that is, the **p** on the right side can use the rule itself to achieve its ends--but this isn't too common because the **r** predicate can recurse too.

Inheritance frequently occurs when the relationship predicate (**r** in the preceding) is **a_kind_of**. That is, if you want to know some property **p** of some **X**, find some **Y** that Xis an example of, and **Y**'s value is **X**'s value too. Some examples:

--If people are animals, and animals eat food, then people eat food.

--If the Enterprise is a kind of ship, and ships float on water, then the Enterprise floats on water.

But inheritance can occur with relationship predicates besides **a_kind_of**:

--If the hood is part of my car's body, and my car's body is gray, then the hood is gray too.

--If the U.S. Navy is part of the U.S. Government, and U.S. Government is everywhere mired in bureaucratic inefficiency, then the U.S. Navy is everywhere mired in bureaucratic inefficiency.

--If the Enterprise is at Norfolk, and Captain Kirk is on the Enterprise, then Captain Kirk is at Norfolk.

And a semantic network example is shown in Figure 4-6. Here we have facts about a particular truck called **truck_4359** and its battery. The truck has a location (**nps_north_lot**) and an owner (**nps**). These properties inherit *downward* to the battery via the **part_of** relationship to the battery, and that's what those two dotted lines from the battery node mean. ("Downward" means from a larger or more general thing to a smaller or more specific thing.) In other words, the location of the battery of a truck comes from the location of the whole truck, and the owner of the battery comes from the owner of the whole truck. Inheritance can also proceed upward from the battery. The status property of the battery (**defective**) inherits upward, because we can say that the truck is defective when its battery is defective.

Generally, properties that represent universal or "for every X" quantification (see Appendix A) inherit downward, while properties that represent existential or "there exists an X" quantification inherit upward. For instance, the "animals eat food" example is downward: this says that for every X an animal, it eats food, and so since people are animals, they eat food too. But consider the property of people that some of them live in

North America, i.e. that there exist some people that live in North America. Then since people are animals, some animals live in North America, so that existential property inherits upward.

Inheritance of a property need not be absolute--there can always be exceptions. For instance, the battery may be removed from the truck for repair, so even though the truck location is the NPS north lot, the battery location isn't. For such an exception, the location of the battery can be put into the database in front of the inheritance rule, so a query on the truck's battery location will find the fact first. So an inheritance rule is a *default*, general-purpose advice that can be overridden.

Inheritance is often associated with transitivity, because when the relationship predicate **r** is transitive, the interpreter is actually following a transitivity chain to follow an inheritance chain. For instance (see Figure 4-7), suppose we have the facts

```
a_kind_of(vinson,carrier).
a_kind_of(carrier,ship).
a_kind_of(ship,vehicle).
purpose(vehicle,transportation).
```

and a rule for inheritance of **purpose** with respect to **a_kind_of**:

```
purpose(X,V) :- a_kind_of(X,Y), purpose(Y,V).
```

and a rule for transitivity of **a_kind_of**:

```
a_kind_of(X,Y) :- a_kind_of(X,Z), a_kind_of(Z,Y).
```

Suppose we query:

```
?- purpose(vehicle,P).
```

We can use the transitivity rule to prove first **a_kind_of(vinson,ship)**, then **a_kind_of(vinson,vehicle)**; the latter inferred relationship is shown by a dotted link in the Figure. Then the inheritance rule succeeds, proving **purpose(vinson,transportation)**; this is also shown by a dotted link.

Thus inheritance is useful for the same reason as transitivity: it can extrapolate an explicit set of facts to a much larger implicit set of facts, by inferences with rules. It usually makes sense, then, to only store "nonredundant" property facts that can't be inherited (though it won't give wrong answers to store redundant information, just increase the size of the database and slow other queries a little). So we typically should store property facts about properties for the *most general* things having those properties. This is because inheritance usually goes downward from more general things to more specific things.

Some implementation problems for transitivity and inheritance

Unfortunately, peculiarities of Prolog interpreters can prevent transitivity and inheritance rules from working properly. The problem is a possible *infinite loop*, a frequent danger in Prolog programming. An infinite loop is when the same thing is done over and over forever when trying to answer a query. Typically, a rule calls itself forever or a set of rules call one another in cycle forever. (A good clue is the error message "Out of stack space" when running a simple program.) You can get into infinite loops in other programming languages too, but it's easier with Prolog because of its emphasis on recursion and complicated backtracking.

To avoid infinite loops for inheritance, we must pay careful attention to database order. We must put any rules about inheriting values for some property after facts about values of that property. We should also use

```
p(X,Value) :- r(X,Y), p(Y,Value).
```

instead of

```
p(X,Value) :- p(Y,Value), r(X,Y).
```

The second rule can cause an infinite loop in which **p** continually calls itself with the same arguments. This won't happen at first, if facts that can match the **p** query come before this rule in the Prolog database; but whenever the proper answer is **no** (either when the only possible answer to the query is **no**, or when we type enough semicolons after other answers), the interpreter will recurse forever instead of saying **no**. This is because Prolog executes *top-down* like other programming languages: the left side of a rule is matched to some query, and then the right side provides new things to match. So be careful with inheritance rules. In fact, as much British detective fiction demonstrates, inheritances can lead to murder.

But it's tougher to eliminate the infinite loop with the transitivity rule:

```
r(X,Y) :- r(X,Z), r(Z,Y).
```

This works fine when the **r** relationship is provable from the facts, but will never say **no** when **r** is unprovable. For instance, query

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

won't say **no** when there are no facts in the database involving either **a** or **b**: the first predicate expression on the right side has no choice but to call on this rule itself. Here reordering the expressions on the right side of the rule won't change anything, because both have predicate **r**. Instead we must rewrite the rule as two, and use another predicate name, one for queries only. For instance, consider transitivity for the **boss** relationship, where **boss(X,Y)** means that person **X** is the boss of person **Y**. To describe indirect bosses, we'll use a new predicate **superior** of two arguments:

```
superior(X,Y) :- boss(X,Y).
superior(X,Y) :- boss(X,Z), superior(Z,Y).
```

This will avoid the infinite loop because the only recursion is in the second rule, and the form of the second rule is the same as the first (better) form for inheritance rules given previously.

A similar trick can be used to state symmetry or *commutativity* of a two-argument predicate. Examples of commutative predicates are the **equals** predicate for numbers, the "distance-between" predicate for places, and the "is-friends-with" predicate for people. The obvious rule can easily cause infinite loops:

```
p(X,Y) :- p(Y,X).
```

Querying a **p2** instead avoids this problem:

```
p2(X,Y) :- p(X,Y).
p2(X,Y) :- p(Y,X).
```

provided we don't use **p2** in a recursive rule itself.

A longer example: some traffic laws

Rules can do many things. As we have seen, rules can define new predicates and extend the power of old predicates. Rules can also state policies, prescriptions of what to do in particular situations. Here is an example of the representation of California traffic laws. These are the laws about signal lights, for both vehicles and pedestrians, from the *California Driver's Handbook*, 1985. The letters in brackets are paragraph codes for later reference.

> [A] New Signals-Note: California is adopting red arrows and yellow arrows in addition to green arrows, as signals for vehicle traffic. This is what the colors of traffic lights mean: A red light or arrow means "STOP" until the green appears. A flashing RED traffic light or arrow means exactly the same as a stop sign, namely STOP! But after stopping, proceed when safe, observing the right-of-way rules.
>
> [B] A GREEN light or arrow means "GO", but you must let any vehicles or pedestrians remaining in the intersection when the signal changes to green, get through before you move ahead. Look to be sure that all cross traffic has stopped before you enter the intersection.
>
> [C] A YELLOW light or arrow warns you that the red signal is about to appear. When you see the yellow light or arrow, you should stop if you can do so safely. If you can't stop, look out for vehicles that may enter the intersection when the light changes. A flashing YELLOW light or arrow is a warning of a hazard. Slow down and be especially alert.
>
> [D] A lighted GREEN ARROW, by itself or with a RED, GREEN or YELLOW light, means you may make the turn indicated by the green arrow. But give the right-of-way to pedestrians and vehicles which are moving as permitted by the lights. The green arrow pointing left allows you to make a "protected" left turn; oncoming traffic is stopped by a red light as long as the green arrow is lighted.
>
> [E] If the green arrow is replaced by a flashing yellow light or arrow, slow down and use caution; make the move which the green arrow would permit, only when safe for you and others.
>
> [F] If the green arrow is replaced by a flashing red light or arrow, stop for either signal; then go ahead when it's safe to do so.
>
> [G] NEW SIGNALS--Note: California is adopting international symbols to guide pedestrians at street crossings. An upraised hand (orange) means the same as the "WAIT" or "DON'T WALK" sign. A walking person symbol (white) means the same as the "WALK" sign.
>
> [H] Special signs for walkers: The "DON'T WALK" or "WAIT" or upraised hand sign, if flashing, warns the walker that it is safe to cross, first yielding to any vehicles which were still in the crossing when the light changed.
>
> [I] At a crossing where there are no special pedestrian signals, walkers must obey the red, yellow, or green lights or arrows. But walkers facing a green arrow must not cross the street.

We can represent the preceding text as a set of Prolog rules in four groups: rules about arrows, rules about cars, rules about pedestrians, and *default* rules (the last two **action** rules). As we said, defaults are weak prescriptions for general cases, only used if more specific advice cannot be found; for instance, the first default rule says that if nothing prevents you from going forward, go forward.

In listing these rules, we follow a standard Prolog convention of putting blank lines between groups of rules with the same left-side predicate name; this makes reading long programs easier. We also use comments; anything between the symbols "/*" and "*/" is treated as a comment and ignored by the Prolog interpreter. The codes at the beginning of lines reference the relevant paragraph(s) of the *Driver's Handbook* text. Note that the text and rules do not correspond exactly; in fact, some "obvious" rules included are nowhere in the text. What's "obvious" to people isn't always so obvious to computers.

```
/* ------ Rules for arrow lights ------ */
/* A */ action(car,stop) :- light(yellow_arrow,Direction),
  safe_stop_possible.
/* D,E */ action(car,yield_and_leftturn) :-
  light(yellow_arrow,left), not(safe_stop_possible).
/* D,E */ action(car,yield_and_rightturn) :-
  light(yellow_arrow,right), not(safe_stop_possible).
/* D */ action(car,yield_and_leftturn) :- light(green_arrow,left).
/* D */ action(car,yield_and_rightturn) :- light(green_arrow,right).
/* ------ Rules for regular lights ------ */
/* A */ action(car,stop) :- light(red,steady).
/* A */ action(car,stop_and_go) :- light(red,flashing).
/* C */ action(car,stop) :- light(yellow,steady),
  safe_stop_possible.
/* C */ action(car,yield_and_go) :- light(yellow,steady),
  not(safe_stop_possible).
/* B */ action(car,yield_and_go) :- light(green,steady).
/* C */ action(car,slow) :- light(yellow,flashing).
/* A */ action(car,stop) :- light(red_arrow,Direction).
/* ------ Rules for pedestrian lights ------ */
action(pedestrian,stop) :- pedhalt(steady).
/* I */ action(pedestrian,stop) :- not(pedsignals), greenfacing.
action(pedestrian,stop) :- pedhalt(flashing),
  safe_stop_possible.
/* H */ action(pedestrian,yield_and_go) :- pedhalt(flashing),
  not(safe_stop_possible).
/* H */ action(pedestrian,yield_and_go) :- pedgo(steady).
/* ------ Default rules ------ */
/* I */ action(pedestrian,A) :- not(pedsignals), not(greenfacing),
action(car,A).
action(X,go) :- not(action(X,stop)),
  not(action(X,stop_and_go)).

/* ------ Rules defining the special terms ------ */
/* G */ pedhalt(State) :- light(wait,State).
/* G */ pedhalt(State) :- light(dont_walk,State).
/* G */ pedhalt(State) :- light(hand,State).


/* G */ pedgo(State) :- light(walk,State).
/* G */ pedgo(State) :- light(walking_person,State).


pedsignals :- pedhalt(State).
pedsignals :- pedgo(State).


greenfacing :- light(green_arrow,right),
  not(clockwise_cross).
```

```
greenfacing :- light(green_arrow,left),
  clockwise_cross.
```

To understand programs like this, it's often a good idea to draw a *predicate hierarchy* (actually, a lattice) showing which predicates refer to which other predicates within their definitions. This helps distinguish *high-level* predicates about abstract things from *low-level* predicates about details. Figure 4-8 shows the predicate hierarchy for this program.

The rules of this program define legal actions in particular traffic situations. To use them, we query the predicate **action** with unbound variables (outputs) for one or both of its arguments. The first argument to **action** represents whether you are a pedestrian or in a car, and the second argument represents a legal action in the situation. So for instance, **action(car,stop)** means that it is legal for the car to stop. The most useful way to query **action** is to make the first argument an input and the second argument an output. So to know what is legal for a car, query

```
?- action(car,X).
```

and the variable **X** will be bound to a description of some action legal with the current facts; to know what is legal for a pedestrian, query

```
?- action(pedestrian,X).
```

To check if some particular action is legal, fill in both arguments. If a sequence of actions is legal, instead of just one, the program puts the characters "**_and_**" between the actions to form one long word.

To describe a situation, three kinds of facts are needed:

--**light(<kind-of-light>,<light-status>)**: this is the main kind of fact. The first argument is either:

red (a circular red light)
yellow (a circular yellow light)
green (a circular green light)
red_arrow
yellow_arrow
green_arrow
wait (the word "wait" lit up)
dont_walk (the words "don't walk" lit up)
hand (a picture of a human palm)
walk (the word "walk" lit up)
walking_person (a picture of a walking person)

The second argument describes the condition of the light; this is either **left** or **right** for the arrow lights, and either **steady** or **flashing** for all the other lights.

--**safe_stop_possible**: this predicate of no arguments asserts that you can stop quickly and safely.

--**clockwise_cross**: if you are a pedestrian, this predicate of no arguments asserts that the path by which you will cross a street is clockwise with respect to the center of that street.

For example, here are the facts for when you are in a car approaching an intersection with plenty of room to stop, and you see a steady yellow light and a green arrow light pointing to the left:

```
safe_stop_possible.
light(yellow,steady).
light(green_arrow,left).
```

Rule order matters, since it establishes priorities among actions. Rules for arrows should go before rules for other kinds of lights, because arrows override other lights showing. Within each of the three main groups of rules--arrows, regular lights, and pedestrian lights--the rules for stopping should go first to handle malfunctions of the lights in which more than one light is lit. Finally, defaults should go last, like the last two **action** rules.

Running the traffic lights program

Here are some examples of the program working. Suppose our database is the example facts just given. Suppose we ask what a car can do:

```
?- action(car,X).
```

The first three rules fail because there are no facts about yellow arrows. But a green arrow to the left is visible, so the fourth rule succeeds, and we get

```
X=yield_and_leftturn
```

In other words, in this situation it is legal to yield and then make a left turn. Now if we type a semicolon, the interpreter will try to find an alternative action. It starts from where it left off in the rules, so it next examines the fifth rule. Neither the fifth, sixth or seventh rules are satisfied by the three facts, but the eighth rule does succeed. So we get

```
X=stop
```

In other words it is legal to stop your car. If we type a semicolon once more, the query will fail (the last rule fails because it is legal to stop the car, as we just showed). So there are only two alternatives.

For another example, suppose a pedestrian walking along a sidewalk comes to an intersection where they wish to cross the street in a clockwise direction with respect to the center of the intersection. Suppose that in the direction the pedestrian wants to go there is a steady green light and a flashing white picture of a walking person. The representation is:

```
clockwise_cross.
light(green,steady).
light(walking_person,flashing).
```

To discover what we can do, the query is

```
?- action(pedestrian,X)
```

None of the first twelve rules succeed because this query specifies a pedestrian. For the thirteenth rule to succeed, a **pedhalt** rules must succeed, but the lights we observe are not any of the three **pedhalt** types. So the thirteenth rule fails, and similarly the fourteenth and fifteenth. For the sixteenth rule we must check the **pedgo** rules. The first **pedgo** rule fails, but the second matches our last fact with the **State=flashing**. So the sixteenth rule succeeds, and we get typed out

```
X=yield_and_go
```

which recommends to the pedestrian to first yield the right-of-way, then go ahead.

This program should be invoked repeatedly, say every second to decide what to do for that second. And each second something must delete old facts and write new facts into the Prolog database. This general approach is followed by most real-time artificial intelligence programs. For speed, each updating can be done by a separate concurrent processor, to avoid interfering with inference. Notice that we have ignored the biggest obstacle to making this program practical, the recognizing of lights in the real world. This is addressed in the "vision" subarea of artificial intelligence.

Declarative programming

The traffic lights program may not seem much like programs in other programming languages. Programming in Prolog (and much artificial intelligence programming, whatever the language) is different in style from most programming. The style is programs in lots of small modular pieces, pieces smaller than the usual subroutines and procedures of other languages. The emphasis is on writing correct pieces, and not on putting the pieces together. In writing and debugging each piece, the emphasis is on whether it makes sense by itself and whether it is logically correct, not how it is used--the "what" instead of the "how". This is called *declarative* programming | REFERENCE 1|. .FS | REFERENCE 1| It's also close to what is called *nondeterminism* or *nondeterministic programming*. .FE

This bothers some students. They feel they don't really understand artificial intelligence programs because there's often no clear, easily understandable sequence in which programs do things. Prolog interpreters for instance work from top to bottom through facts and rules with the same predicate name, but for each new predicate in a query, they jump to the first occurrence of that predicate name, and jump around in quite complicated ways when they backtrack. And Prolog's operation represents one of the simpler ways to do artificial intelligence programs, as we'll see in Chapter 6. A programmer accustomed to flowcharts may find this bewildering.

In reply, one can say that artificial intelligence solves hard problems, problems on which conventional software-engineering techniques (including numerical ones) struggle, and for which artificial-intelligence methods seem to be the only ones that work. And there are no clear criteria for "intelligent behavior", so in designing intelligent programs it would seem more important to ensure that the individual pieces of the program make sense rather than imposing some grand (necessarily controversial) organizing scheme. Usually, researchers don't want artificial intelligence programs to be brilliant, just not dumb, and concentrating on the pieces helps avoid being dumb. Also, the well-known software technique of recursion (see Appendix B and Chapter 5) is best understood in a declarative way, and artificial intelligence often uses recursion.

This still may not sound too reassuring, but remember there are lots of ways to program. As we say in California, stay mellow.

Keywords:

rule

left side of a rule
right side of a rule
local variable in a rule
parameter variable of a rule
hierarchies of rules
postponed binding
transitivity of a predicate
inheritance of a predicate
commutativity of a predicate
infinite loop
database order
default
declarative programming

.SH Exercises

4-1. (E) What's the difference between "**:-**" and logical implication? In other words, what's the difference between **a :- b.** and the logical statement

"If b is true, then a is true."

4-2. (A) (a) Suppose in a database you had a rule whose right side was guaranteed to be always true. How could you get the effect of such a rule in a simpler way?

(b) Suppose in a database you had a rule whose right side was always guaranteed to be false for any assignment of bindings. How could you get the effect of such a rule in a simpler way?

4-3. Write rules for reasoning about genealogies (or "family trees"). Assume the genealogy is represented by facts of the single four-argument form

```
child(<name_of_father>,<name_of_mother>,<name_of_child>,<sex>).
```

Define the following new predicates based on the **child** facts:

father(X,Y), meaning **X** is the father of **Y**
mother(X,Y), meaning **X** is the mother of **Y**
son(X,Y), meaning **X** is the son of **Y**
grandfather(X,Y), meaning **X** is the grandfather of **Y**
sister(X,Y), meaning **X** is the sister of **Y**
uncle(X,Y), meaning **X** is the uncle of **Y**
ancestor(X,Y), meaning **X** is the ancestor of **Y**
half_sister(X,Y), meaning **X** is the half-sister of **Y**

4-4. (P) Figure 4-9 is a picture of three stacks of blocks on a table.

(a) Represent in Prolog the facts describing the relationships between the blocks. Use two predicates, one that says a block is on another, and one that says a block is immediately to the left of another. Only give facts of the second type for blocks at the bottom of piles.

(b) Type your facts into a file.

(c) Now ask the Prolog interpreter:

--what blocks are on block B;
--what blocks block A is on;
--what blocks are on other blocks;
--what blocks are on blocks that are immediately to the left of other blocks.

(d) Define a new predicate **above** which is true when a block is anywhere in the stack above another block. Define a new predicate **stackleft** when a block is in a stack that is immediately to the left of the stack of another block. Put these into your file, and load the whole thing.

(e) Now ask the Prolog interpreter:

--what blocks are above other blocks;
--what blocks are either above block F or in a stack immediately to the left of its stack;
--what blocks are above other blocks but are not in a stack immediately to the left of any block.

4-5. (A) Consider the rules:

```
a(X) :- not(b(X)).


b(X) :- not(c(X)).
```

Assuming predicate **b** is never referenced except in the preceding, and is never queried directly, and there are no facts with predicate **b**, would it be equivalent to define a single rule

```
a(X) :- c(X).
```

Why?

4-6. (A) (a) Consider the following to be true:

Clint is the mayor.
The mayor is respected.

From this we can conclude that:

Clint is respected.

But now suppose:

Clint is the mayor.
The mayor is an elected office.

From this we would seem to be able to conclude:

Clint is an elected office.

What is the fallacy in reasoning?

(b) Consider the following to be true:

Clint is a movie star.
Clint is mayor.

From this it makes sense to conclude:

A movie star is mayor.

But now suppose as true:

John is a little stupid.
John is mayor.

It is fallacious then to conclude:

A little stupid is mayor.

What is the fallacy?

4-7. (A,E) Explain using Prolog concepts the different meanings of the word "and" in the following.

(a) A food crop is anything that is a plant and provides substantial human nutrition.

(b) A department chairman is a professor and is responsible to the provost.

(c) To put out a fire, your options are to apply flame retardants and to let it burn itself out.

(d) Tom and Sue are managers.

(e) Tom and Sue are friends.

4-8. Consider this query:

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

used with this database:

```
a(1,2).
a(3,5).
a(R,S) :- b(R,S), b(S,R).


b(1,3).
b(2,3).
b(3,T) :- b(2,T), b(1,T).
```

Without using the computer, what is the first answer found to the query?

4-9. Inheritance involves a thing and a property. Suppose the thing is a computer program. Give two different examples of relationship predicates that are involved in inheritance, and the corresponding properties.

4-10. (A) Consider the type predicate **some** of two arguments **<set>** and **<property>** which is true whenever some members of **<set>** have **<property>**. So for instance **some(people,american)** means that some people are American. Consider inheritance involving the "set containment" relationship predicate.

(a) Does **some** inherit upward (to an including set), downward (to an included set), both, or neither? (Note: inheritance is only "yes" or "no", never "maybe".)

(b) Write the inheritance for part (a) as a Prolog rule.

(c) Consider the similar predicate **all** which is true whenever all members of **<set>** have **<property>**. So for instance **all(people,american)** means that all people are American. Does it inherit upward, downward, both, or neither?

(d) Consider the similar predicate **most** which is true whenever most members of **<set>** have **<property>**. Does it inherit upward, downward, both, or neither?

4-11. (R,A) Suppose we have facts about the components of a car. Suppose:

--**front_of(X,Y)** means part **X** of the car is in front of part **Y** (towards the headlights);
--**inside_of(X,Y)** means part **X** of the car is inside (contained in) part **Y**.

(a) Is **inside_of** transitive?

(b) Does **front_of** inherit with respect to **inside_of**? If so, in which direction?

(c) Why could it be more useful for a program to have a description of where things are under the hood of a car in terms of **front_of**, **right_of**, and **above** facts instead of in terms of Cartesian (x, y, and z) coordinates?

4-12. Consider the accounting department of some organization you know. For each of the following properties, say whether it inherits (1) upward, to some entity including the accounting department, (2) downward, to some entity included in the accounting department, (3) both directions, or (4) neither direction.

(a) There are crooks in it.

(b) Half the employees in it are crooks.

(c) Regulations controlling crookedness are enforced.

(d) All crooks in it have been caught and punished.

4-13. (R,A,P) (a) Represent the following facts in Prolog (binary-relationship predicates are recommended). Represent what the words mean, not what they say; each different word shouldn't necessarily be a different predicate name or argument. Type your Prolog facts into a computer file.

An Acme hotplate has a cord and a body.
Part of the body of an Acme hotplate is the heating element.
The heating element is metal.
Another part of the body of an Acme hotplate is the cover.
The cover has a knob.

Plastic is always used for knobs.
One of the things a cord consists of is a wire.
Metal comprises the wire.
Part of the cord is an insulater.
The insulater is fiber.

(b) Start up the Prolog interpreter, load the file, and ask

--what things are stated to contain metal;
--what things are stated to be parts of the body of a hot plate.

(c) Define rules for transitivity of **part_of** and upward inheritance of material with respect to **part_of**, rules that can be applied to your answer to part (a). Put these in the file.

(d) Now ask:

--what things contain plastic;
--what things do not contain fiber;
--what things containing metal also contain fiber.

4-14. Suppose we have **a_kind_of** facts about 31 objects. Suppose by transitivity on these facts we can show that object **a** is a kind of object **b**. Suppose the facts have no redundancy (two different routes between the same two objects by following arrows) and no cycles (ways to leave an object and return by following arrows).

(a) What is the minimum number of times that you could successfully use the recursive transitivity rule (the rule, not any fact) proving this? Use the two-rule form in Section 4.10.

(b) What is the maximum number?

(c) Suppose 30 of the 31 objects appear once and only once each as the first argument to the **a_kind_of** facts. Suppose 16 of the 31 objects never appear as a second argument and suppose the rest appear exactly twice each as a second argument. Of the remaining 15 that do appear as a second argument, 8 of them appear in facts with the 16; of the remaining 7, 4 appear in facts with the 8; of the remaining 3, 2 appear in facts with the 4; and the remaining one appears in facts with the 2. What data structure does the semantic network resemble?

(d) For the situation in part (c), what is the maximum number of times the recursive transitivity rule could be successfully used to prove that object **a** is a kind of object **b**?

(e) Suppose **b** has property **v**, and suppose that **a** inherits this value because **a** is a kind of **b**. Assume as before there are 31 objects total. What is the maximum number of times the inheritance rule (of the form of the first one in Section 4.9) could be successfully used proving this?

4-15. (a) Represent the following as Prolog facts and rules (definitions). Hint: represent what these mean, not what they literally say. Be as general as you can.

A VW Rabbit is a VW.
Tom's car is a VW Rabbit.

> Dick's car is a VW Rabbit.
> A VW has an electrical system.
> Part of the electrical system is the alternator.
> The alternator is defective on every VW.

(b) Write Prolog inference rules that will allow conclusion that Tom's car or Dick's car is defective. Hint: you need to define transitivity and inheritance for concepts in part (a).

(c) Prove that Dick's car is defective, given the facts and rules of parts (a) and (b). (Don't prove it the way a Prolog interpreter would--omit dead ends.)

4-16. (P) Write Prolog definitions for the California speed laws that follow, as extracted from the *California Driver's Handbook*. Your top-level predicate should be called **limit**, with one argument, an output variable. The program should set that variable to the legal maximum speed under the current conditions. The current conditions should be specified as Prolog facts.

Don't worry too much about the predicates you use here; there's much room for personal taste. Instead, worry about the *order* of Prolog definitions and facts. Note you must handle the situation in which you may have seen several road signs recently, some perhaps contradictory, and you must decide which ones apply. Assume though that any other facts (like the presence of children) apply to the immediate vicinity. Warning: you'll find the laws unclear about certain situations; just pick something reasonable in those cases.

> "The maximum speed limit in California is 55 miles per hour. Other speed limit signs tell you the highest speed at which you can expect to drive with safety in the places where the signs are set up...."
>
> "In business or residence districts, 25 miles per hour is the speed limit unless signs show other limits. When you see a "SCHOOL" sign, the speed limit is 25 miles per hour while children are outside or are crossing the street during school hours. The 25 m.p.h. limit applies at all times when a school ground is unfenced and children are outside, even though the road is posted for a higher speed. Lower speed must be obeyed if posted...."
>
> "When you come within 100 feet of a railroad crossing and you cannot see the tracks for 400 feet in both directions, the limit is 15 m.p.h. This limit does not apply if the crossing is controlled by gates, a warning signal or a flagman."
>
> "The 15 m.p.h. limit also applies at blind intersections where you cannot see for 100 feet both ways during the last 100 feet before crossing, unless yield or stop signs on the side streets give you the right of way--also in any alley."

As tests, run the program to find the limit at each point in the following scenario:

(a) You enter a residential district. A sign says 55 m.p.h.

(b) You are still in the residential district. You come to a SCHOOL sign, and students are on the sidewalks. The time is within school hours.

(c) A speed limit sign says 35 m.p.h. You enter an alley.

4-17. (E) Explain in what ways legal definitions are different from Prolog definitions. Is this a weakness of legal definitions, a weakness of Prolog, both, or neither? Should the two evolve closer together?

4-18. (E) Definitions of terms in politics don't seem to be very much like Prolog terms. For instance, what one nation calls an unprovoked military attack may be considered by the attacking nation as "claiming what is rightfully theirs" or "preventing terrorism". These excuses are not arbitrary but are supported by dictionary definitions. What makes political definitions so much more slippery than the Prolog definitions of this chapter?

4-19. There are several part-whole relationships, even for the same object. Consider a piece of rock consisting of | 10 sup 24 | molecules of silicon dioxide, whose chemical formula is |Si O sub 2|.

(a) Give an interpretation of **part_of** for which the **color** property inherits to any part of this rock.

(b) Give an interpretation of **part_of** for which the **number_of_molecules** property inherits to any subpart.

4-20. Consider the following proof that God does not exist. Take the following statement as true from the definition of God:

> God saves those who can't save themselves.

(a) Write this as a Prolog rule whose left side and right side both refer to a **saves** predicate of two arguments.

(b) Suppose the person saved is God. Show the bindings in the rule, and explain what the rule becomes in this case.

(c) As you can see, there's a serious logical problem here. Does this prove God doesn't exist? Why?

Go to book index