1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

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

Arithmetic and lists in Prolog

Before going any further, we need to introduce two additional features of Prolog that will come in handy in writing future programs, arithmetic and lists. These give rules new capabilities. As we've seen already, rules can:

1. define new predicates in terms of existing predicates
2. extend the power of existing predicates (as with inheritance rules)
3. recommend what to do in a situation (as with the traffic lights program)

To these, we'll now add:

4. quantify and rank things
5. store, retrieve, and manipulate sets and sequences of data items

Arithmetic comparisons

Prolog has built-in arithmetic comparison predicates. But their predicate expressions are written differently from those shown so far: they're written in the *infix* notation of mathematics. The predicate name comes between the arguments, like this:

3 > 4 means 3 is greater than 4
15 = 15 means 15 equals 15
X < Y means **X** is less than **Y**
Z >= 4 means **Z** is greater than or equal to 4
PPPP =< 3 means PPPP is less than or equal to 3

We'll usually put spaces around infix symbols to make them easier to see, but it's not required. As an example, here's the definition of a predicate that checks if a number is positive:

```
positive(X) :- X > 0.
```

With this definition in our database, it could be used like this:

```
?- positive(3).
yes
?- positive(-6).
no
```

Here's the definition of a predicate that checks if its first argument is a number lying in the range from its second to its third argument, assuming all arguments are bound:

```
in_range(X,Y,Z) :- X >= Y, X =< Z.
```

Using this definition, the query

```
?- in_range(3,0,10).
```

gives the response **yes**.

Arithmetic assignment

Like any computer language, Prolog has arithmetic computations and assignment statements. Arithmetic assignment is done by expressions with the infix **is** predicate. Querying these peculiar expressions has the side effect of binding some variable to the result of some arithmetic computation. For instance

```
X is ( 2 * 3 ) + 7
```

binds (assigns) **X** to the value 13 (2 times 3 plus 7). The thing to the left of the **is** must be a variable name, and the stuff to the right must be an algebraic formula of variables and numeric constants, something that evaluates to a number. The algebraic formula is written in standard infix form, with operations **+**, (addition), **-** (subtraction), ***** (multiplication), and **/** (division). We'll often put spaces around these symbols to make them more readable. The algebraic formula can have variables only if they're bound to values, as in

```
Y is 2, X is Y * Y.
```

where **Y** is first bound to 2, and then **X** is bound to 4. A practical example is this definition of the square of a number, intended to be a function predicate:

```
square(X,Y) :- Y is X * X.
```

If this rule is in the Prolog database, then if we query

```
?- square(3,Y).
```

(that is, if we ask what the square of 3 is), the Prolog interpreter will type

```
Y=9
```

Notice that since predicate expressions aren't functions in Prolog, we can't write anything like

```
f(X,Y) + g(X,Z)
```

even if **f** and **g** are function predicates, because expressions only succeed or fail; expressions don't have "values". Instead, to add the two function values we must say something like

```
f(X,Y), g(X,Z), T is Y + Z
```

Another warning: don't confuse **=** with **is**. The **=** is a purely logical comparison of whether two things are equal. (Originally intended for numbers, it also works for words.) The **is** is an operation, an arithmetic assignment statement that figures out a value and binds a variable to it.

Reversing the "is"

A serious weakness of arithmetic, which makes it different from everything else in Prolog we've talked about so far, is that it isn't multiway or reversible. For instance, if we have the preceding definition of **square** in our database, and we query

```
?- square(X,9).
```

wondering what number squared is 9, the interpreter will refuse to do anything because the right side of the **is** statement refers to an unbound variable. This is different from having a bunch of arithmetic facts in *prefix* form like

```
square(0,1).
square(1,1).
square(2,4).
square(3,9).
```

for which we could query **square(3,Y)** or **square(X,9)** or even **square(X,Y)**

and get an answer. Similarly, for the preceding definition of **positive**, the query

```
?- positive(X).
```

won't work: the interpreter can only do a **>** comparison when both things are bound to numbers. So it will complain and refuse to do anything.

The Prolog interpreter's excuse for its behavior is that function inversion and other such *multiway reasoning* is hard to do in general, and sometimes is impossible. A square of a number is easy to compute, but a square root requires iterative approximation and a lot more code. And there are an infinity of positive numbers; where should an interpreter start when asked to give one? Artificial intelligence requires flexible reasoning capable of going in many different directions--people seem to do it. So it's desirable to get around the interpreter's limitations.

One way is to provide additional rules for a predicate definition. Helpful in this is the built-in Prolog predicate **var** of one argument, which succeeds if that argument is an unbound variable, and fails otherwise. As an example of its use, consider a **better_add** predicate of three arguments which says the sum of the first two arguments is the third argument. If all three arguments are bound (inputs), then it checks the addition. If the first two arguments are bound, it binds the third to their sum. If the first and third arguments are bound, it binds the second to the difference of the third and the first. Similarly if the second and third arguments are bound, it binds the first to the difference of the third and second. Here's the code (**Z** is a temporary-storage variable):

```
better_add(X,Y,S) :- not(var(X)), not(var(Y)), not(var(S)),
  Z is X + Y, Z=S.
better_add(X,Y,S) :- not(var(X)), not(var(Y)), var(S),
  S is X + Y.
better_add(X,Y,S) :- not(var(X)), var(Y), not(var(S)),
  Y is S - X.
better_add(X,Y,S) :- var(X), not(var(Y)), not(var(S)),
  X is S - Y.
```

We can't handle two arguments unbound; then there's an infinite number of possibilities for the bindings. But at least the preceding handles three more cases than the Prolog **is** can handle by itself.

The **in_range** predicate of Section 5.1 can provide another example of a predicate enhancement. That predicate checked whether its first argument (an input number) was between the second and third arguments (input numbers too). We can improve **in_range** so that an unbound first argument will make it *generate* a number between other two arguments, and generate further numbers on backtracking. To make things easier, we'll assume all numbers will be integers. Here's the definition of this **integer_in_range**:

```
integer_in_range(X,Y,Z) :- not(var(X)), not(var(Y)), not(var(Z)),
  X >= Y, X =< Z.
integer_in_range(X,Y,Z) :- var(X), not(var(Y)), not(var(Z)),
  Y =< Z, X is Y.
integer_in_range(X,Y,Z) :- Y =< Z, Y2 is Y + 1,
integer_in_range(X,Y2,Z).
```

This is a *tail-recursive* program of a form we'll use many times in this chapter. (Again, see Appendix B to review recursion.) The first rule handles the case handled before. The second rule says if **X** is unbound and **Y** and **Z** are bound, and we want to generate an integer on the range **Y** to **Z**, we can always pick **Y**. Otherwise (if a semicolon is typed), the third rule is used. It "crosses out" **Y** from the range by increasing the lower limit of the range by 1, and generates an integer from this new, narrower range. If the range ever decreases so much that it disappears, all the rules fail. So if we query

```
?- integer_in_range(X,1,10).
```

the interpreter first replies **X=1**; then if we type a semicolon, **X=2**; then if we type a semicolon, **X=3**; and so on up to 10.

Lists in Prolog

Another important feature of Prolog is linked-lists. Every argument in a predicate expression in a query must be anticipated and planned for. To handle sets and sequences of varying or unknown length, we need something else: linked-lists, which we'll henceforth call just *lists*.

Lists have always been important in artificial intelligence. Lisp, the other major artificial intelligence programming language, is almost entirely implemented with lists--even programs are lists in Lisp. The extra space that lists need compared to arrays (see Appendix C) is more than compensated in artificial intelligence applications by the flexibility possible.

Square brackets indicate a Prolog list, with commas separating items. For example:

```
[monday,tuesday,wednesday,thursday,friday,saturday,sunday]
```

(Don't confuse square brackets "[]" with parentheses "()"; they're completely different in Prolog. Brackets group lists and parentheses group arguments.) Lists can be values of variables just like words and numbers. Suppose we have the following facts:

```
weekdays([monday,tuesday,wednesday,thursday,friday]).
weekends([saturday,sunday]).
```

Then to ask what days are weekdays, we type the query

```
?- weekdays(Days).
```

and the answer is

```
Days=[monday,tuesday,wednesday,thursday,friday]
```

We can also bind variables to items of lists. For instance, if we query

```
?- weekends([X,Y]).
```

with the preceding facts in the database, we get

```
X=saturday, Y=sunday
```

But that last query requires that the weekends list have exactly two items; if we query

```
?- weekends([X,Y,Z]).
```

we get **no** because the query list can't be made to match the data list by some binding.

We can work with lists of arbitrary length by the standard methods for linked-pointer list manipulation. We can refer to any list of one or more items as **[X|Y]**, where **X** is the first item and **Y** is the rest of the list (that is, the list of everything but the first item in the same order) | REFERENCE 1|. .FS | REFERENCE 1| In the language Lisp, **X** is called the *car* and **Y** is called *cdr* of the list. .FE We'll call "**|**" the *bar* symbol. Note that **[X|Y]** is quite different from **[X,Y]**; the first can have any nonzero number of items, whereas the second must have exactly two items. Note also that **X** and **Y** are different data types in **[X|Y]**; **X** is a single item, but **Y** is a list of items. So **[X|Y]** represents an uneven division of a list.

Here are some examples with the previous weekdays and weekends facts.

```
?- weekdays([A|L]).
A=monday, L=[tuesday,wednesday,thursday,friday]

?- weekdays([A,B,C|L]).
A=monday, B=tuesday, C=wednesday, L=[thursday,friday]

?- weekends([A,B|L]).
A=saturday, B=sunday, L=[]
```

The "**[]**" is the list of zero items, the *empty list* | REFERENCE 2|. .FS | REFERENCE 2| Called *nil* in the language Lisp. .FE

Defining some list-processing predicates

Let's write some famous list-processing programs (summarized in Figure 5-1). Programs requiring many lines of code in conventional programming languages can often be quite short in Prolog because of its declarative nature. We'll define mostly function predicates. Following the convention of Section 2.9, the function result is the last argument of their predicate expressions.

First, here's a definition of a predicate that computes the first item of an indefinitely long list:

```
first([X|L],X).
```

This definition is a fact, not a rule--but remember, facts are just rules with no right side. So **X** stands for any item, and **L** stands for any list.

Here's a definition of the last item of a list:

```
last([X],X).
last([X|L],X2) :- last(L,X2).
```

The first line says that the last item of a list of one item is that item. The second line says the last item of any

other nonempty list is the last item of the list formed by removing the first item. This is a tail-recursive program with the first line the *basis step* (simple nonrecursive case) and the second line the *induction step* (recursive case). Tail recursion is the standard way to define list-processing predicates in Prolog, with each recursion chopping one item off a list.

We can use **first** and **last** just like the predicate definitions in Chapter 4, to work on data we type in. For instance:

```
?- last([monday,tuesday,wednesday,thursday,friday],X).
X=friday
```

We can also use it on lists in the database by doing a database retrieval first. Suppose we have a database fact:

```
weekdays([monday,tuesday,wednesday,thursday,friday]).
```

Then we could find out the last weekday by:

```
?- weekdays(L), last(L,X).
L=[monday,tuesday,wednesday,thursday,friday], X=friday
```

As another example of a list-processing definition, consider **member(X,L)** which is true if item **X** is a member of list **L**. We can give the following fact and rule (note the order: the recursive part of a definition should always come after the nonrecursive, to avoid an infinite loop) | REFERENCE 3|:

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

.FS | REFERENCE 3| Recursive list-processing predicate definitions, and many other recursive definitions too, can be made considerably more efficient by Prolog's built-in "cut" predicate (symbolized by "**!**"), to be explained in Section 10.7. For instance, a better version of **member** for most purposes is: **member(X,[X|L]) :- !.**
member(X,[Y|L]) :- member(X,L). .FE The fact says that **X** is a member of any list where it is the first item; there's no need to check the rest of the list then. Otherwise, figure out if **X** is a member of the rest of the list, and that's the answer. Notice there's no need to give conditions under which **member** fails, like

```
member(X,[]) :- 1=2.
```

Since 1 can never equal 2, this rule never succeeds. But in Prolog, failure failing when we use a rule means the same thing as no rule at all. So the immediately preceding rule is completely useless.

We just gave a declarative explanation of the **member** predicate. For a *procedural* explanation (though we emphasize again that this is not the best way to understand recursive programs), consider the query

```
?- member(dick,[tom,dick,harry]).
```

The first line of the **member** definition fails because **dick** is not **tom**. So the second line is used, creating a recursive call

```
?- member(dick,[dick,harry]).
```

for which the first line succeeds. So the original query gives **yes**.

The **member** definition will also work when the first argument is unbound (an output). Then the program *generates* members of a list in succession under backtracking, something quite useful for artificial-intelligence programs. Consider the query:

```
?- member(X,[tom,dick,harry]).
```

When the interpreter executes this, the first line of the program can match **X=tom**, and this binding is printed out. If we now type a semicolon, we request a different binding, forcing the interpreter to use the second rule. So this recursive call is executed:

```
?- member(X,[dick,harry]).
```

And for this new query the first line can succeed, giving the result **X=dick**. If we type another semicolon, we'll be querying

```
?- member(X,[harry]).
```

and we'll get **X=harry**; and if we type yet another semicolon, we'll be querying

```
?- member(X,[]).
```

and we'll get **no**.

Here's a predicate **length(L,N)** that computes length **N** of a list **L**:

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

Remember, **[]** represents the *empty list*, the list with no members. The first line says the empty list has length zero. The second line says that the length of any other list is just one more than the length of the list created by removing the first item. For instance:

```
?- length([a,b,c,d],N).
N=4
```

Here's a predicate **max(L,M)** that computes the maximum of a list **L** of numbers:

```
max([X],X).
max([X|L],X) :- max(L,M), X > M.
max([X|L],M) :- max(L,M), X =< M.
```

The first line says the maximum of a list of one item is that item. The second line says that the first number in a list is the maximum of the list if it's greater than the maximum for the rest of the list. The third line says the maximum of a list is the maximum for all but the first item of the list if neither of the first two rules applies. For instance:

```
?- max([3,7,2,6,1],N).
N=7
```

List-creating predicates

Suppose we want to delete every occurrence of some item from a list, creating a new list. We can do this with a predicate **delete** of three arguments: (1) the item **X** we want to get rid of (an input), (2) the initial list **L** (an

input), and (3) the final list **M** (an output). And we'll assume that's the only pattern of inputs and outputs we'll ever use. For instance:

```
?- delete(b,[b,a,b,b,c],M).
M=[a,c]
```

To define this, we could write:

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

But there's a better way to write the last rule: we can move the **[Y|M]** list to the left side. This is good because (1) the **is** is unnecessary because left sides can also bind variables, and (2) **is** isn't completely reversible, and we'd like a more multiway program. So we could use instead:

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

This works the same, even with the third argument unbound, because nothing can be done with the **[Y|M]** on the left side until the right side is executed and **M** is bound. So the construction of **[Y|M]** remains the last thing done by the third rule.

You may be puzzled why the **not(X=Y)** in the third line is necessary. We could write

```
delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- delete(X,L,M).
```

The **delete** predicate never fails for its first two arguments bound; one of those three rules must always succeed. So if the second line fails, the left side must be at fault, and **X** and **Y** must be different, right? Yes, but only the *first* time through. If we ever backtrack into this **delete**, we'll be in trouble because backtracking would use the third rule for some situation in which it used the second rule previously. For instance:

```
?- delete(b,[a,b,a,b,c],L).
L=[a,a,c] ;
L=[a,a,b,c] ;
L=[a,b,a,b,c] ;
no
```

So be careful in Prolog programming: just because something works OK for its first answer doesn't mean it will work OK on backtracking to get new answers.

Next, here's a useful predicate that "appends" (concatenates) one list to another. It has three arguments: the first list, the second list, and the combined list.

```
append([],L,L).
append([X|L1],L2,Lnew) :- append(L1,L2,L3), Lnew is [X|L3].
```

As with **delete**, we can rewrite the last rule to eliminate the awkward and undesirable **is**, moving the **[X|L3]** to the left side:

```
append([],L,L).
```

```
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

This says first that the anything appended on the empty list is that thing itself. Otherwise, to append some nonempty list having first item **X** to a second list **L2**, append the *rest* of that first list (without **X**) to **L2**, and then put **X** in front of that. Study this revised definition carefully; it's a good example of how the style of Prolog programming differs from the style of most other programming.

Figure 5-2 shows an example using **append** with the first two arguments bound (inputs) and the third argument unbound (an output), for the query

```
?- append([gas,oil],[tires,battery,radiator],Things_to_check_before_trip).
```

The nested boxes represent the rule invocation environments created with each recursion. The outer one holds the parameter and local variables for the initial invocation of **append**. This invocation recursively calls itself, creating the middle box, with its own distinct variables. This invocation of **append** in turn recursively calls itself, resulting in the inner environment (box) with yet more distinct variables. The processing state at this point is depicted in the Figure. Now:

1. The first line of **append** says to bind its third argument to its value of **L**, **[tires,battery,radiator]**, and the invocation of the inner box succeeds.

2. Returning next to the middle box, **[tires,battery,radiator]** is the value of **L3**, and its **X** is **oil**, so **[X|L3]** is **[oil,tires,battery,radiator]**. So the invocation of the middle box succeeds.

3. Returning to the outer box, **[oil,tires,battery,radiator]** is the value of *its* **L3**, so its **[X|L3]** is **[gas,oil,tires,battery,radiator]**. The outer box succeeds.

4. So the original third argument **Things_to_check_before_trip** is bound to **[gas,oil,tires,battery,radiator]**.

Like the **member** predicate and many other predicates defined without arithmetic, **append** will work several ways. In fact, it will work seven ways (see Figure 5-3). For instance, it will handle the case in which the third argument is bound (an input) but the first and second arguments are unbound (outputs). Then the first two arguments are bound to binary partitions (breakings-in-half) of the third argument. So

```
?- append(L1,L2,[tom,dick,harry]).
```

gives the following if you keep typing a semicolon:

```
L1=[], L2=[tom,dick,harry];
L1=[tom], L2=[dick,harry];
L1=[tom,dick], L2=[harry];
L1=[tom,dick,harry], L2=[];
no.
```

The other rows in Figure 5-3 show other ways **append** can be used. Basically, we've got seven quite different programs in one. This comes about from the declarative interpretation of the definition: it describes conditions that hold when its third argument is the result of appending the first two arguments, not how to do it. Again, it describes "what" instead of "how".

Combining list predicates

List predicate definitions can refer to other list predicates. For instance, we can use **member** to define a **subset** predicate that determines whether all the members of some list **L1** are members of some list **L2**. Here it is, and we print **member** as well to refresh your memory.

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


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

We leave a blank line between the rule groups with the same predicate name, following a convention (also followed in the traffic lights program of Section 4.11). Here's an example use of the program:

```
?- subset([b,c],[a,c,e,d,b]).
yes
```

Here's a program for sorting lists of numbers into increasing order, *insertion sort* in particular:

```
sort([],[]).
sort([X|L1],L2) :- sort(L1,L3), insert_item(X,L3,L2).


insert_item(X,[],[X]).
insert_item(X,[Y|L],[X,Y|L]) :- X < Y.
insert_item(X,[Y|L1],[Y|L2]) :- X >= Y, insert_item(X,L1,L2).
```

The first argument to **sort** is an unsorted input list, and the second argument is the output sorted list. The first argument to **insert_item** is an input item, the second argument an input list, and the third argument is the result of inserting that item into that list, an output. For instance:

```
?- sort([3,2,7,4],L).
L=[2,3,4,7]
```

Redundancy in definitions

Basis conditions in a recursive definition are simple cases that don't require recursion. We always need at least one basis condition in a recursion, but we can have more than one. For instance, we could define length of a list this way:

```
length([],0).
length([X],1).
length([X,Y],2).
length([X,Y,Z|L],N) :- length(L,N2), N is N2 + 3.
```

instead of equivalently:

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

But the extra lines can speed calculation when our lists are very short on the average: we're able to answer many queries without recursion. Here's a similar alternative to the **member** definition of Section 5.5:

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

Here the second line is redundant, since its case can be covered by a slightly modified third line. But if lists are often, or the things we're looking for are usually towards the front of the lists, the preceding definition may be faster than the original definition.

Such modifications are a kind of *caching*, a concept that occurs in many disguises in artificial intelligence, and which will reappear in a quite different form in Chapter 6. Caching means asserting unnecessary or redundant facts to improve efficiency. The idea is to waste a little space (the extra facts) in the hope of improving calculation speed. Caching doesn't always improve speed significantly, so to justify it you first need to do experiments, or do some mathematical analysis like that we'll do in Section 9.14 and 13.3.

An example: dejargonizing bureaucratese (*)

Natural language (human language) is a major subarea of artificial intelligence research. Lists and list processing routines are the obvious way to represent and use sentences and languages in Prolog. As an example, consider a critical technical problem facing the United States today: the translation of bureaucratic jargon into real English. Bureaucratic organizations typically use their own terminology to make their accomplishments look a little less pathetic than they really are. It would be useful to take a sentence of such bureaucratic jargon, expressed as a Prolog list, and convert it to understandable everyday English. Such a translation program might be used routinely on government documents.

For instance, "impact" is often misused as a verb, as in "The study will impact the department." When so used, it can be replaced by the simpler and more standard English word "affect". Similarly, "adversely impact" and "negatively impact" can be replaced by "hurt". "Transition" is also misused as a verb, as in "The project will transition to phase 3," and can be replaced by "change". "Evaluate options" and "consider options" can be changed to "study", and "under advisement" and "under consideration" to "being studied". You can probably recall many more examples. These substitutions usually but not always have the same meanings, so the sentence created by applying them should always be carefully double-checked.

It's easy to write a Prolog program for this, once English sentences are represented in Prolog list format. First we represent the substitution pairs as facts. For example:

```
substitution([adversely,impact],[hurt]).
substitution([negatively,impact],[hurt]).
substitution([impact],[affect]),
substitution([will,transition],[will,change]).
substitution([must,transition],[must,change]).
substitution([to,transition],[to,change]).
substitution([consider,options],[study]).
substitution([evaluate,options],[study]).
substitution([under,advisement],[being,studied].
substitution([under,consideration],[being,studied].
substitution([expedite],[do]).
substitution([expeditiously],[fast]).
substitution([will,secure],[will,close]).
substitution([must,secure],[must,close]).
substitution([prioritize],[rank]).
```

The first argument contains the original words, and the second argument the words to be substituted. Note that extra words in the first argument narrow the applicability of the substitution, but reduce the possibility of making mistakes.

Next we define a predicate that recurses through the sentence list, like **member** and **delete** defined in Sections 5.5 and 5.6 respectively:

```
dejargonize([],[]).
dejargonize(L,NL) :- substitution(S,NS), append(S,L2,L),
  append(NS,L2,L3), dejargonize(L3,NL).
dejargonize([X|L],[X|L2]) :- dejargonize(L,L2).
```

The first line sets the basis condition as the empty list, and the last line recurses through the list. The middle two lines do the work of substitution. They check through the substitution facts for one whose first argument matches the front of the list (using the **append** predicate according to the third line of Figure 5-3, for second argument unbound), and substitute (using **append** according to the second line of Figure 5-3, for the third argument unbound), and recurse on the new list. Here's an example:

```
?- dejargonize
([we,must,consider,options,to,transition,expeditiously],L).
L=[we,must,study,to,change,fast]
```

Despite the somewhat frivolous nature of this program, the idea of substituting into strings of words is important in much natural language work, and we'll look some more into it at the end of the next chapter.

Keywords:

infix
arithmetic assignment
is
prefix
multiway reasoning
recursion
tail recursion
basis of a recursion
induction step of a recursion
lists
Lisp
linked-pointer list representation
the bar symbol
the member predicate
the delete predicate
the append predicate
caching

Exercises

***5-1. (A,P) Define a Prolog predicate max(X,Y,Z,M) that says that M is the maximum of the three input numbers X, Y*, and Z. Use only ">" to compare numbers.**

5-2. (R,A) A student familiar with Pascal was writing a compiler in Prolog. This required translating

an error code number into an English description, so they wrote rules like this:

```
translate(Code,Meaning) :- Code=1, Meaning is integer_overflow.
translate(Code,Meaning) :- Code=2, Meaning is division_by_zero.
translate(Code,Meaning) :- Code=3, Meaning is unknown_identifier.
```

This is poor Prolog programming style. How can it be improved?

5-3. (P) Write a better_divide like better_add that handles similar cases for division. Have the program prevent division by zero.

5-4. To figure out a tax amount, you subtract the deductions from the gross and multiply by the tax rate (expressed as a decimal number less than 1). Using the better_add predicate defined in Section 5.3 and an analogous predicate better_multiply that you define yourself, write a single Prolog rule that can be used to answer all the following questions by a single query each:

> **--If my gross is 1,000, my deductions 270, and my tax rate 0.15, what is my tax?**
>
> **--If my tax was 170 at a tax rate of 0.17, with no deductions, what was my gross?**
>
> **--If my tax was 170 at a tax rate 0.17 and a gross of 1500, what amount of deductions did I take?**
>
> **--What tax rate would result in a tax of 80 on a gross of 1200 with 400 in deductions?**

If your Prolog dialect can handle decimal numbers, show your program works correctly for the preceding questions.

5-5. (P) (This requires a Prolog dialect with floating-point numbers.) Define a predicate square(X,Y) that says that its second argument is the square of its first argument (the result of multiplying the first argument by itself). Using the built-in var predicate, have it handle each of the four cases of binding of its arguments:

> **--if both arguments are bound, it checks that the second argument is the square of the first;**
>
> **--if the first argument is bound and the second argument unbound, it computes the square of the first argument;**
>
> **--if the first argument is unbound and the second argument is bound, it computes an approximation of the first argument within 0.001 by bisection search or some other iterative method from numerical analysis;**
>
> **--if both arguments are unbound, it generates all possible pairs of positive integers and their squares starting with 1.**

5-6. (P) (a) Define a new predicate integer_better_add, like better_add but able to handle the case in which its first two arguments are unbound (outputs), finding all pairs of integers that sum to a bound (input) integer third argument.

(b) Use part (a) to write a program to generate three-by-three magic squares that have a given number

as characteristic sum. (A *magic square* is a two-dimensional array of integers such that the sum of eight things--the three columns, the three rows, and the two diagonals--is the same.)

5-7. Consider this query:

```
a(X,Y), not(c(X)), d(X,Y).
```

Suppose our Prolog database contains this, in order:

```
d(X,Y) :- X > 1, Y > 1.


a(0,1).
a(0,2).
a(2,1).
a(M,N) :- b(P,Q), b(Q,P), M is P + 1, N is Q + 1.


c(0).


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

(a) Without using the computer, what is the first answer found to the query by a Prolog interpreter?

(b) Without using the computer, what is the second answer found to the query (when you type a semicolon after the first answer)?

5-8. (R,A) Suppose we have Prolog facts about named events in terms of the following three predicates:

> **start(<event>,<time>), an event started at a particular time**
> **end(<event>,<time>), an event ended at a particular time**
> **duration(<event>,<length>), an event lasted for a particular length of time**

We may have one, two, or all three of these facts about some event, and we can't know in advance which we will have if we have one or two.

(a) Write Prolog rules to infer an end time for some event when there is no end(Event,Time) fact for it, and to infer a start time when there is no start(Event,Time) fact for it.

(b) Define a new Prolog predicate after(Event1,Event2) which is true when its first argument Event1 definitely happened after its second argument Event2.

(c) Define a new Prolog predicate during(Event1,Event2) which is true when its first argument Event1 definitely happened while its second argument Event2 was happening.

(d) Explain where in a Prolog database of facts these rules should go for things to work right.

5-9. (R,A) The transmission of a car contains gears that transmit power to the car's wheels. You can infer the speed and direction of the wheels, given facts about what gears are connected to what other

gears and what is driving them.

Assume gears are labeled g1, g2, g3, and so on. Assume the number of teeth on each gear is specified by facts of the form

```
teeth(<gear_name>,<number_of_teeth>).
```

Assume all rigid connections between gears on the same rigid shaft are specified by facts of the form

```
same_shaft(<gear_name_1>,<gear_name_2>).
```

Assume that all meshed (teeth-to-teeth) connections between gears are specified by facts of the form

```
meshed(<gear_name_1>,<gear_name_2>).
```

(a) We want to reason about the rotational speed and direction of gears. Give a good format for such facts for each gear.

(b) Anytime two gears are rigidly attached to the same rigid shaft, their rotational speeds are the same. Write a Prolog rule that can infer the rotational speed of one such gear on a shaft from the known rotational speed of another such gear. Use the fact format from part (a).

(c) Anytime two gears are connected or "meshed" the product of the number of teeth and the rotational speed for each gear is the same, except that one gear rotates in the opposite direction from the other. Write a Prolog rule that can infer the rotational speed of a gear from the rotational speed of a gear meshed with it, assuming the number of teeth on both gears is known. Use the fact format from part (a).

(d) Suppose gear g1 has a rotational speed of 5000 rpm in a clockwise direction. Suppose it is on the same shaft as g2, g2 is meshed to g3, and g2 is meshed to g4. Suppose g1 has 100 teeth, g2 has 30 teeth, g3 has 60 teeth, and g4 has 90 teeth. Give a Prolog query that will figure out the rotational speed and direction of gear g4 from a database of these facts. Then show the steps that the Prolog interpreter will take to answer that query. Note: you must write the facts with the correct order of arguments in order for your inference rules to apply properly to them.

(e) Explain how infinite loops could happen when reasoning about gears, if you weren't careful in specifying the facts.

(f) Suppose for some arrangement of gears and specified gear speeds you find gear g8 has a speed of 1200 rpm by one chain of reasoning, and 800 rpm by another chain of reasoning. What does this mean for the gears?

5-10. (P) Define a predicate convert that does units conversion for length measurement; for instance, it converts a measurement in feet to meters. The convert predicate should take four arguments: a number (an input), the units for that number (an input), the units to which you want to convert (an input), and the result number (an output). Handle the following units: meters, decimeters, centimeters, millimeters, decameters, kilometers, inches, feet, yards, and miles. Hint: don't write a separate rule for every pair of possible units, but chain inferences.

5-11. Consider representing maps of highway routes in Prolog. Suppose you don't have much memory

space, so representing squares of the map as a two-dimensional array is out of the question. Instead, you want to store just information about what towns are connected by which highways segments (assume segments have names and numbers, not unique, like "California 62"), and distances. Suppose highway segments are in different counties, states, and countries, and we want to remember which. Suppose we also store the different maximum speed limits for different states and countries. Assume each route is in a single county (you can create imaginary towns on county lines to ensure this). Assume highway segments meet only at towns.

(a) Give formats for the fact predicates you need.

(b) Give a rule or rules for inferring the maximum speed limit on a road.

(c) Give a rule or rules for inferring a distance (not necessarily the shortest, that's hard) between two towns. Don't worry about getting into a infinite loop; assume the Prolog interpreter is smart enough not to visit the same town twice in a route (we'll explain how to do this in Chapter 10).

(d) Suppose there are R routes (connecting two towns each), C counties, S states, and K countries. How many facts do you save with the inference in part (b)? Make reasonable assumptions if necessary.

(e) Suppose there are R routes (connecting two towns each) and T towns. Approximately how many facts do you save with the inference in part (c)? Perhaps consider different arrangements of towns. Make reasonable assumptions if necessary.

5-12. (P) Define a completely multidirectional inference_distance of three arguments. Its first argument is a kind of the second argument. Its third argument is the number of linking a_kind_of facts that must be followed to get from the first argument to the second. By "completely multidirectional" we mean able to handle any pattern of bindings of the arguments. Assume there are only a_kind_of facts, no rules, and the facts don't have any cycles, and there is only one route between any two things. (The inference distance concept is important in psychology, because some psychologists believe that humans have semantic networks in their heads and that the speed of human reasoning is proportional to the inference distance.)

5-13. (E) Consider a Prolog definition written to implement a function, in other words written to be used when all arguments but the last are bound (inputs), which binds its last argument to some unique value. In abstract mathematical terms, what characteristics must the function have to be easily used multidirectionally, that is with last argument bound and other arguments unbound?

5-14. (A,P) (a) Using a single call to append and no other predicates, implement the member predicate. (Hint: the fifth row of Figure 5-3 is closest to what you need.)

(b) Using a single call to append and no other predicates, implement the last predicate.

(c) Using just two calls to append and no other predicates, implement a deleteone predicate that removes a single occurrence of some item from a list.

(d) Using just two calls to append and no other predicates, implement a before predicate of three arguments, that succeeds if its first two arguments are both members of its list third argument, and where the first argument item occurs before the second argument item.

5-15. (E) Figure 5-3 is missing a row for the case in which all variables are unbound. Explain why for this case the predicate definition will not work properly.

5-16. Consider the delete predicate defined in the chapter:

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

(a) Suppose you rewrite it as

```
delete(X,[],[]).
delete(X,[X|L],M) :- delete(X,L,M).
delete(X,[Y|L],[Y|M]) :- delete(X,L,M).
```

What happens when you query this predicate with second and third arguments bound and first argument unbound?

(b) What happens if you change the second line to

```
delete(X,[X|L],L).
```

and query this predicate with first and second arguments bound and third argument unbound?

5-17. (A) (a) Suppose that Prolog predicate mystery is queried with its first argument a bound (input) list and its second argument unbound (an output). Describe what the predicate mystery does, in a sentence of 20 or less English words. (Hint: try it on sample lists.)

```
mystery([],[]).
mystery([X],[X]).
mystery([X,Y|L],[X,censored|M]) :- mystery(L,M).
```

(b) Most recursive list-processing definitions have only one basis condition. Why does the preceding definition need two?

5-18. In the language Logo and other languages with *turtle geometry* primitives, there are special commands to control a plotting pen. The pen has a position (measured in millimeters in a coordinate system) and a direction it is pointing in the plane of the paper (measured in degrees). Two built-in predicates manipulate the pen: forward(X) which moves the pen forward a distance X in the direction it is pointing, and right(X), which turns the pen around X degrees (without moving its location) so it points in a new direction. This program draws spirals, increasing the length it moves forward each step:

```
spiral(Side,Angle,Increment) :- forward(Side), right(Angle),
  Side2 is Side + Increment, spiral(Side2,Angle,Increment).
```

(a) Draw what the query

```
?- unknown(4).
```

causes to be drawn. Assume the pen is pointing north at the start. Indicate the lengths of lines in your drawing. Here is the definition:

```
unknown(1) :- right(90).
unknown(N) :- Nm1 is N-1, unknown(Nm1), N2 is 5-N,
  forward(N2), unknown(Nm1).
```

(b) Draw what the query

```
?- flake(3).
```

causes to be drawn. Again assume the pen is pointing north at the start. Here is the definition:

```
flake(1) :- forward(1).
flake(N) :- Nm1 is N-1, flake(Nm1), right(-60), flake(Nm1),
  right(120), flake(Nm1), right(-60), flake(Nm1).
```

Hint: this one is too complicated to figure out procedurally--reason about the declarative meaning of the rules.

5-19. (A) Consider use of the member predicate with both arguments bound, and with the second (list) argument N items long.

(a) How many times is the rule invoked when the answer to the query is no?

(b) Suppose the answer to the query is yes and suppose the item being searched for is equally likely to appear at any position in the list. How many times on the average will the rule be invoked now?

5-20. Consider our "dejargonizing" program. Each substitution of one set of words for another takes one line to define. Explain how to compress this set of substitutions so one line can handle many substitutions. (Hint: variables can be items in lists.)

5-21. (E) What is wrong with the following "proof" that all horses are the same color? "One horse is obviously the same color as itself. Suppose for some N, every set of N horses is the same color. Then consider some horse Snowy, a white horse, for purposes of argument. Consider the set of N+1 items formed by including Snowy in an arbitrary set of N horses. Now if we take out some other horse than Snowy from this set (call him Blacky), we have a set of N horses. By the induction assumption, these horses are all the same color. But Snowy is white. Therefore all the other horses in the original set must have been white. So put Blacky back in and take out some other horse (besides Snowy, call him Alabaster) to create a new set of N horses. This new set must also be all of the same color, so since it includes Snowy, the color must be white. But Blacky is in the set too, and so must be white too. Hence every set of N+1 horses must be white. Hence if any set of N horses is all the same color, any set of N+1 horses is of the same color. Hence by recursive argument, all horses are the same color."

5-22. (A,P) Write a modified transitivity rule (for let's say transitivity of a(X,Y)) that can't ever get into infinite loops when answering a query of the form

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

Show your program working on some sample data that contains cycles with respect to the a predicate. (A "cycle" means a closed loop in the semantic network.) Hint: use an extra list argument.

5-23. (P) Create a Prolog database and query it appropriately to create poems of the following form:

A A B C C B D D D D E E B

Here each capital letter stands for a class of nonsense words that must rhyme together. B represents one-syllable words, and the other letters represent two-syllable words. In addition, the poem cannot have duplicate words. Here is an example poem:

uga buga ru batta hatta nu fitty pitty witty ditty garra farra tu

It will help to define a predicate different of two arguments that says whether two words are identical. Present your resulting poem as the value of a single variable, a list of lists for which each list is a line of the poem. Type semicolons to see what similar poems you get.

Go to book index