INTRODUCTION TO PROLOG

10.8.13

PROLOG is a logic programming language. This is a class of languages based on logic, distinct from functional programming languages (such as LISP) and procedural programming languages (like JAVA and C++). Most people are familiar with the latter, but less so with functional and logic languages. PYTHON is actually a mix, with lots of functional aspects, as well as procedural. Both functional and logic languages are sometimes called declarative since their programs simply state (declare) what kinds of things can be done rather than how to do them (the compiler or interpreter figures out how); this is especially dramatic in the logic-programming case.

PROLOG uses Horn clause wffs only (recall that theses are disjunctions of literals, at most one of which is positive), and usually written in these forms:

\[
pred(Vars) : - pred(Vars), \ldots, pred(Vars).
\]

and

\[
pred(constants).
\]

This is less expressive the full FOL; for instance, \(P \lor Q\) is not a Horn clause. But for many purposes Horn clauses are enough.

Here the \(-\) acts like a backward arrow, \(\beta\), and can be read “…if…”. The idea is that it gives a rule for proving the left-hand-side, namely: establish the right-hand-side. This typically involves finding suitable values for the variables (Vars) to make that happen, and that is what PROLOG does. PROLOG tries to find bindings for the variables that will satisfy your description of the problem that is in your program.

Here is a trivial example:

Example 1

\[
p(X) : - q(X).
\]

\[
q(c).
\]

These two lines constitute a complete PROLOG program. It would be saved in a file and loaded into PROLOG when we want to use it. The program has no instructions at all, just information in the form of one fact (c has property q) and one rule: for any value of \(X\), \(p(X)\) will be true if \(q(X)\) is. Clearly these beg for modus ponens to be used. But PROLOG won’t do anything unless we give it something to establish, a so-called query. We do this by typing what we want it to try to prove, in this case perhaps \(p(c)\). So we’d type \(p(c)\) and PROLOG obliges by trying to make it true. It would check what it knows (the rule and the fact in the program above) and combine them (using resolution, not MP) together with the negation of \(p(c)\), to try to derive a contradiction. In this case it would succeed, and it would tell us so by returning “true”.
If instead we had typed the query \( p(X) \) — thereby asking PROLOG to find a way to make this true -- by finding values of \( X \) that satisfy \( p(X) \) -- then it would search (again by resolution) and in the process determine that \( X \) must be \( c \), and it would return the result \( X=c \).

PROLOG programmers do not need to think about data structures (not much, at least), nor flow of control, searching, sorting, etc. But they do often need to think very hard about how to describe the problem carefully, not only in enough detail and clarity but also so that PROLOG can proceed in a left-to-right top-to-bottom effective search (yes, search, but PROLOG does it automatically) for ways to get variable values that work.

How does one actually use PROLOG? At UMCP, go to the GRACE system (ssh grace.umd.edu) and then use your university login info. Once logged in, type `swipl` (for SWI PROLOG, one popular version); this will start up the prolog system, and you will see some welcoming text and then the symbol `?`, which is prolog’s prompt. You’ll need to load your program (which will have been typed into a file, say `myprog.pl`). Do this by entering

```
?- [myprog]. %don’t type the ?, prolog does that.
```

Note that `%` starts a comment. Also note the period at the end; this is essential. It tells prolog that you are finished with your input. Next we tell prolog what we want it to try to prove (make true) for us.

At this point we need more serious examples.

**Example 2**

```
parent_of(X,Y) :- father_of(X,Y).
parent_of(X,Y) :- mother_of(X,Y).

sibling_of(X,Y) :- parent_of(X,Z), parent_of(Y,Z).

mother_of(joe,mary).
father_of(sue,bill).
mother_of(sue,mary).
```

```
?- parent_of(X,Y). % find all child-parent pairs
X = sue,
Y = bill ;
X = joe,
Y = mary ;
X = sue,
Y = mary.
```
Prolog has found three pairs: sue/bill, joe/mary, and sue/mary.

?- sibling_of(sue,X).
X=sue ;
X=joe ;
X=sue.

This needs explanation. First, sibling_of(X,Y) has been declared as true if a Z can be found that is parent of both X and Y. That is, X and Y are siblings if they have a common parent. But nothing in that says X and Y have to be different, and since Sue has the same parents as Sue, then Sue is her own sibling. Same with Joe. Finally, since prolog knows both the father and the mother of Sue, then prolog has two ways to prove Sue is Sue’s sibling.

This last result – getting the same answer in more than one way – can be stopped (if we want to stop it), by means of the “cut” operator, written as “!” . We add it to the right of something else, such that, if prolog succeeds in making it true, we don’t want prolog to try to make it true again. So we could use the following rewrite of the sibling_of line above:

sibling_of(X,Y) :- parent_of(X,Z), parent_of(Y,Z), !.

But now the query gives too little:
?- sibling_of(sue,X)
X=sue.

Joe is left out. This is because, once one answer is found, the cut told prolog to stop looking. The cut is in fact a very useful tool, but it did not do what we want in this case.

One also might want to prevent prolog from considering people as their own siblings. But negation is tricky in prolog, since it is not in Horn clause form. That is, we might wish to assert something like not-sibling_of(X,X) to mean X cannot be X’s sibling. But prolog has no direct way to do this.

What prolog does have is the ability to declare that something has failed, with the fail predicate. For instance, we can revise the sibling rule this way:

sibling_of(X,Y) :- parent_of(X,Z), parent_of(Y,Z),
                diff(X,Y).

The above seems to say that we now can only prove people are siblings if they are different. But what does “different” mean? This is not a built-in prolog predicate. So we
have to declare how to prove when X and Y are different, which is done in the additional
lines:

diff(X,X) :- !, fail.
diff(X,Y).

The first line says that prolog should not try to prove diff(X,X) more than once, and that
it fails. So this blocks diff(X,X) from ever being proven. The second line says diff(X,Y)
is true, period. They seem to be contradictory.

But prolog goes in a top-to-bottom, left-to-right fashion. So in processing the query
sibling_of(sue,X), it will try to prove the diff(sue,X) predicate by first matching it with
diff(X,X); if it can do that (i.e., if sue and its choice of a possible X are the same symbol)
then the cut (!) says that it should never try to prove diff again (during the processing of
the current query) with those same two arguments. But it can try other arguments, such as
sue and joe, and diff(sue,joe) succeeds: diff(X,X) will not match it (it never even reaches
the cut) and so prolog goes on to try the diff(X,Y) line below it, and it is always true.

This would not have worked if we reversed the two lines.

There is a quicker way to do the same thing as

sibling_of(X,Y) :- parent_of(X,Z), parent_of(Y,Z), diff(X,Y).
diff(X,X) :- !, fail.
diff(X,Y).

namely:
sibling_of(X,Y) :- parent_of(X,Z), parent_of(Y,Z),
                not((X,Y)).

Prolog has not as a special predicate that works as a cut-fail combo, exactly as we made
diff work with two extra lines of code. And = is a special predicate as well, that gets
prolog to try to show its two arguments are the same; it does not do arithmetic in order to
show this. So =(2,1+1) will not succeed as a query, but =(1,1) and ==(a,a) and ==(X,X) will;
and so will ==(X,a) when prolog tries the value a for X.

The next example makes clear that prolog is by no means just geared toward what one
might think of as logic problems.

Example 2
factorial_of(0,1).
factorial_of(N,Ans) :-
    N>0,
N1 is N-1,  %Note: “is” gets prolog to eval numerical expressions like N-1.
factorial_of(N1,PrevAns),
Ans is PrevAns*N.

OK, this doesn’t look very different from what one would write in almost any language -- PYTHON, for instance. But on closer inspection, the PROLOG code describes (declares) what it means for one variable, Ans, to be the factorial of another, N. Still, that meaning – in this case – pretty much contains the basis for computing Ans from N.

The BIG difference is that in PROLOG we are using factorial_of not as a function but rather a predicate: it says that N and Ans are related in a certain way, specified by the four indented lines below it. And then it is up to PROLOG to figure out from this what, for instance, Ans must be if factorial_of(5,Ans) is to be true. That is, we use the program by typing a query such as factorial_of(5,Ans); this asks PROLOG to make it true, by finding a value of the variables (only Ans in this case) that it can prove from what information is in the program.

?- factorial_of(5,Ans).
   Ans=120

Also note that we can ask other forms of query, such as
?- factorial_of(3,6).
   True .

We might also ask the query “backwards”, try to find what number has 120 as its factorial:

?-factorial_of(X,120).
ERROR: </2: Arguments are not sufficiently instantiated

PROLOG has complained. It’s not that one cannot ask such things, but that in this case the program does not pin X down enough for PROLOG to have a decent chance of finding a successful value of X before it runs out of space. That is, there are so many things X could be, that PROLOG does not have a good handle on where to start looking. There are ways to nudge prolog to look in a more constrained way, but we will not go into that here.

But in many cases (we will see another one below) this kind of backward-seeming query is just fine, and it illustrates a power of logic-programming that procedural or functional languages do not so readily provide. In fact, the above example of siblings already showed us this. We can ask (and get answers to) who are the siblings of sue, who are the parents of sue, whom mary is the parent of, and so on. Prolog really just tries to make any query true, no matter which arguments are variables.
Example 3 (Lists -- these may be especially useful in Project 1)

Next we introduce predicates for various list-processing notions. Prolog has a special notation for lists: [a,b,c,d,e,…]; and also [X|Y] represents a list whose first element is X, and where Y is the rest of the list. Thus [a,b,c,d,e,…] is the same list as [a | [b,c,d,e,…]].

The following predicates declare what it means for one list to be the tail of another, for an item to be the first element of a list, etc.

tail_of([X|Y], Y). %That’s it! This is not built-in; we just defined it!

So we can query

?- tail_of([a,b,c,d],T). % and we get:
T = [b,c,d]

first_of([X|Y], X).
?- first_of([a,b,c,d], F).
F = a

second_in(List , Ans) :-
    tail_of(List, T),
    first_of(T, Ans).

?- second_in([a,b,c,d],S).
S = b

Etc.

% The next rules specify what is meant by a member of a list.

member_of([X|Y], X).
member_of(List, Memb) :-
    tail_of(List, T),
    member_of(T, Memb).

% OR an alternative version:

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

It turns out that member is already a built-in predicate in SWI-PROLOG, so we don’t really need either of the member_of versions above. But member works in reverse from our member_of: member([b,[a,b,c]]) is true, not member([a,b,c],b).
Next we create a predicate that describes what it means for a list to be the result of removing an item from another list:

\[
\begin{align*}
\text{removal_of}(\text{Item}, [\text{Item}\mid \text{Tail}], \text{Tail}). \\
\text{removal_of}(\text{Item}, [\text{First}\mid \text{Tail}], [\text{First}\mid \text{RestOfTail}]) & : \text{removal_of}(\text{Item}, \text{Tail}, \text{RestOfTail}).
\end{align*}
\]

?- removal_of(b, [a,b,c], L).
L = [a,c]

Note that we can also use removal “backwards” (as we wanted to earlier with factorial_of), asking what item X was removed from [a,b,c] to leave [a,c]:

?- removal_of(X, [a,b,c], [a,c]).
X = b

But if we try asking what list L item b was removed from to get [a,c], we get a surprise:

?- removal_of(b, X, [a,c]).
X = [b,a,c] ;
X = [a,b,c] ;
X = [a,c,b] ;
False.

What has happened here is that PROLOG has diligently worked to find whatever it can substitute for X to make the query true. First it finds the list [b,a,c] is one possibility, since removing b from it does leave [a,c] as needed. Then it waits, and if we type a semicolon (;) it keeps looking, and finds [a,b,c]; and finally also [a,c,b]. Notice that is has tried putting the missing b in front, then in the middle, and finally in back – and not because we told it to. PROLOG handles all that kind of detail on its own. The “False” at the end means it tried a fourth time (because we typed another semicolon) and failed.

The following describes what it is for a list to have another item stuck on at its end:

\[
\begin{align*}
\text{appendedto}([], \text{Elem}, [\text{Elem}]). \\
\text{appendedto}([\text{X}\mid \text{Y}], \text{Elem}, [\text{X}\mid \text{Tail}]) & : \text{appendedto}(\text{Y}, \text{Elem}, \text{Tail}).
\end{align*}
\]

?- appendedto([a,b,c], c, L).
L = [a,b,c,c].

SWI-PROLOG has a built-in append predicate, that describes a further list (not a further element) being appended to the end of another, to produce a third list, as in

?- append([a,b,c], [d,e], L).
L = [a,b,c,d,e]

And here is prolog code describing what it means to be the reverse of a list:

reverse_of([],[]).
reverse_of([X|T], Rev) :-
    reverse_of(T, TRev),
    appended_to(TRev, X, Rev).

?- reverse_of([a,b,c,d], X).
X = [d,c,b,a].

?- reverse_of([X,b], [b,a]).
X = a.

While we are on the topic of lists, here is another program that uses lists as a workaround to avoid having to go to a lot of trouble with fancy prolog features that would otherwise be needed. This program determines whether a typed input integer is prime – as long as it is between 1 and 400. It also uses read and write, which are more-or-less self-evident (but later on these are revisited with some explanation). The list of integers from 2 to 20 in the 5th line below are the ones used as possible factors of the input value N.

composite(A) :-
    template(B), B < A,
    0 is A mod B, !.

template(A) :-
    member(A, [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]).

testforprime(A) :-
    prime(A),
    write('Yes, prime! '),
    main.
testforprime(_),-
    write('Not prime. '),
    main.

main :-
    write('Please type an integer between 1 and 400: '),
    read(A),
    integer(A),
    A>1,
    testforprime(A),
    write('Yes '),
write(A),
write(' is prime. ').

prime(A) :-
  integer(A),
  not(toobig(A)),
  not(composite(A)).

toobig(A) :-
  A>400,
  write('Your N is too large; try again with N < 401').

Example 4

One can also solve the **8 queens problem** quite easily using PROLOG. Here is one way to do it (taken from *Prolog Programming for Artificial Intelligence*, by Ivan Bratko). The predicate solution (S) means S is a correct list of non-attacking positions X/Y. This is an incremental solution, meaning it regards the empty list as a solution, but uses another predicate “template” to force a final answer to be complete and to have a convenient format. That is, the user asks the double-query “?- template(S), solution(S)” to force prolog to find an S that not only is a solution but that has all eight parts, where each part gives the column-number X along with its row Y. The notation X/Y does not refer to division but is just a pairing of X with Y. The template line at the bottom forces X to be each of 1,2,…,8 in turn, if the query is template(S), solution(S).

solution([ ]). solution([X/Y | Others]) :- % First queen at X/Y, other queens at Others
  solution(Others),
  member(Y, [1,2,3,4,5,6,7,8]),
  noattack(X/Y, Others). % First queen does not attack others

noattack(_, []). % Nothing to attack
noattack(X/Y, [X1/Y1 | Others]) :-
  Y =\= Y1, % Different Y-coordinates
  Y1-Y =\= X1-X, % Different diagonals
  Y1-Y =\= X-X1,
  noattack(X/Y, Others).

template([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).

This will print out, one after another (as long as we keep typing semicolons) all the solutions.
Eight-queens is one of the most amazing examples of the power of logic programming. In just a few simple lines one describes the problem in detail, and that’s it! Prolog does the rest.

You can have fun – and get useful experience – by cutting and pasting the above programs into files in your grace account, then typing swipl, then loading a file (or many files), and asking queries.

Input/Output in PROLOG -- simple aspects (keyboard input and monitor output)

Clearly we need a way to get info into a running prolog system, and info back out. Yet everything is supposed to happen as an attempt by prolog to find a way to get a query to be true. So we’ll need some predicates that do double-duty (other actions, aka “side-effects”). We already saw three: is, fail and cut.

Another is write. write(…) is a predicate that always succeeds (is always treated as true), but that has a side-effect: it prints out its argument (if it something that can be written). But where to put it in the program?

Factorial example, again:

factorial_of(0,1).
factorial_of(N,Ans) :-
    N>0,
    N1 is N-1,
    factorial_of(N1,PrevAns),
    Ans is PrevAns*N,

myprog(N) :-
    write(‘The factorial of ‘), write(N), write(‘ is ‘), write(Ans).

Now the query
    myprog(5)
will result in this output:
    The factorial of 5 is 120

While myprog(5) does call (as a sub-query) factorial_of(5,Ans), and while the latter does find Ans=120, prolog knows this is not what the user asked, and so it does not print the Ans=120. It just sets Ans to be 120 and goes back to finishing up the original myprog query, which does cause printing to happen due to the write predicates in it.

That is, we have written a “wrapper” that calls factorial_of and then puts some writes at the end. In effect, myprog is what is often called “main”, and it will probably be a key programming construct for your first project. Asking prolog to prove main(…) will
involve prolog in calling (proving) various subparts of the project, including I/O parts. And the subparts can also call yet further sub-subparts, and so on, in good modular programming style.

[By the way, one way to halt prolog it to type \texttt{ctrl-d}. But another more elegant way is to use the predicate \texttt{halt} (with a period, of course).]

\textbf{For Project 1}, your code will need to have a way to add new information to itself. That is, when a user “teaches” something, such as “John is a human”, in the form \texttt{[john, is, a, human]}, your code should from then on have that stored as a fact of some sort, perhaps as \texttt{learned([john, is, a, human])} where \texttt{learned} is a new predicate. Fortunately, prolog has a way for a program to add facts (and even rules) to itself.

The basic predicate that is used to add new items to your program (which we shall refer to generically as the KB), is this: \texttt{assert}. Here is a fairly illustrative example of how it can be used within a program:

\begin{verbatim}
:- dynamic good/2.
mypred(X) :- cond1(X),cond2(X,Y), assert(good(X,Y)).
cond1(joe).
cond2(joe,sue).
\end{verbatim}

Here we announce in the first line that the predicate \texttt{good} is one that may have new facts asserted about it at some point. This warns prolog to be ready to allow changes in what \texttt{good} applies to. The second line then says \texttt{mypred} can be proven if certain other predicates can be made true (\texttt{cond1} and \texttt{cond2}) and also the assert predicate -- but the latter \texttt{always} succeeds, and has the side-effect of inserting its argument as a new fact in the KB. So, given the above four lines in our program, if we load it into prolog then we can present these queries and get the indicated results:

\begin{verbatim}
?- mypred(X).
X = joe.

?- good(X,Y).
X = joe, Y = sue.
\end{verbatim}

If we want to \textit{look} at the KB (i.e., our program at this) at the point in time, we use the predicate \texttt{listing}, and here is what happens:

\begin{verbatim}
?- listing.

mypred(A) :-
    cond1(A),
\end{verbatim}
cond2(A, B),
  assert(good(A, B)).

cond2(joe, sue).

:- dynamic good/2.

good(joe, sue).

cond1(joe).

In general, using listing (followed by a period) as a query will show us our code. But if the code has been run and if it involved an assert, then it will have been changed, so the listing now will be different, as we see above: good(joe,sue) is now there, but it was not in the original KB (the actual program we loaded). Nor is the new fact in the actual file holding the code – that did not change, only the runtime version. (There are ways to save it permanently, but we will not go into that here. Also note that the internally stored version (printed by listing) has renamed the variables and also re-ordered the lines a bit; this is normal and is not due to the assert feature.)

One more feature is this: how to enter user data from the keyboard. This is done with the read predicate. The following brief program illustrates how it works:

hello_world:-
  write(`What is your name?`),
  read(X),
  write(`Hello, `), write(X).

If we load this program, we can get the following:

?- hello_world.
What is your name? 'Denise'.
Hello, Denise
true.

If we leave out the quote-marks then the capital D will cause prolog to treat Denise as a variable, and the write(X) will print _L135 or some such internal coding for a variable. But if we use quotes or lowercase keyboard input, the write will be the same as what we type.

While you may use read in your project code, it is not necessary. You can instead have a special predicate that the user types, such as

input(…).
where the … is whatever is needed for your purposes. For instance:

input(['avwilliams', 'is', 'a', 'building']).

Just be sure that your documentation makes *very clear* exactly how your code is to be used, so the user (that is, the grader!) knows exactly how to run it and use it.