Exploring the Rule-Based/Declarative/Logical Paradigm Through Prolog

We will now study Prolog as a means of exploring the rule-based paradigm (which has several commonly used names). Note that the previous paradigms, imperative or procedural and functional, have involved the idea of *commands* to the machine. In the declarative paradigm, the idea is to declare facts and have the machine draw conclusions from them.

This is an intriguing idea! In a software development effort using imperative or functional languages, we first figure out what we want the program to do, and then do a magical, annoying, difficult thing known as "programming" or "coding" to tell the machine what actions to take in order to accomplish the goals. In the dream of the declarative paradigm, we very precisely state what we want the program to accomplish, and then it figures out how to accomplish that on its own.

> Due to lack of time, we will barely begin to learn about Prolog, but it should still be a powerful experience as you experience a truly different paradigm from the ones you are used to.
>
> We should also note that the extremely powerful language (and environment) *Mathematica* follows the rule-based paradigm, which demonstrates that serious work can be accomplished with languages from this paradigm.

Basics of Prolog

We will use SWI-Prolog as our vehicle for briefly exploring the declarative paradigm.

The free online book entitled "Learn Prolog Now" seems nice, so you are encouraged to read through the first parts of it to supplement this written chapter and accompanying videos. Just search online for `learn prolog now`). We will cover most of what is in Chapters 1–6.

You should go to `www.swi-prolog.org` and download a version of Prolog for your platform.

The general process is to use a text editor to write your "program" and then to start up the `swipl` application. This application has a prompt that allows you to state facts and queries.

If you have made a file `test.pl`, you can load it into the database of rules by doing

`consult('test.pl').` (or `consult('test').`)

You can see a listing of all the rules by doing

`listing.`

You can see a listing of all the rules for a predicate `f` by doing

`listing(f).`

You can terminate the session by doing

`halt.`

All Prolog rules have the form

`<left> :- <right>`

with the semantics "if `<right>` is satisfied then `<left>` will be satisfied," where `<left>` and `<right>` are *predicates*. A predicate is something of the form `f(x,y,z)` or `f(A,B,C)`.

Identifers starting with lowercase letters are atoms that represent fixed entities, namely predicates or objects. Identifiers starting with uppercase letters represent variables.

A single underscore symbol is used to represent an *anonymous* variable. It can be used when we want to say that we don't care what value is substituted for a variable. Two or more underscores used in the same predicate can take on different values.

If `<right>` is empty, then we can write:

`<left>`

to say that `<left>` is simply true. Such an unconditional rule is known as a "fact."

The right hand side of a rule can be a simple predicate, or it can be a sequence of two or more predicates separated by commas. To satisfy something like

`<pred1>, <pred2>, <pred3>`

Prolog first tries to satisfy `<pred1>`, then `<pred2>`, and finally `<pred3>`. Thus, the comma behaves like an "and" operator.

You can also use semi-colons to separate predicates, with Prolog trying to satisfying one or the other, thus accomplishing a form of "or" operator. This is just an abbreviation for using several separate rules.

The key idea is that when you ask Prolog to satisfy a predicate involving variables, it tries (using a mechanism to be described shortly) to find rules that let it replace variables by atoms in order to "prove" that the predicate is satisfied.

⇒ Watch the video `prolog1.mp4` in which I make a database of rules in a file named `family.pl` about a famous family. These rules will use predicates such as `father`, `mother`, `parent`, and so on, with atoms `homer`, `marge`, and so on representing people in the family. After making the database, we'll start up Prolog and ask it to satisfy some queries.

The Unification Mechanism

To understand how Prolog works, you must understand its algorithm for trying to satisfy a predicate. We will describe this here for reference, but may not understand it until we have more experience.

Given a goal, Prolog scans the list of rules in order until it finds the first one whose left part (head) matches the goal, with a suitable substitution of specific atoms for variables. Then it looks at the right part (tail) and takes the first clause as its new goal. It continues

in this way until it finds a substitution for each variable that satisfies the entire tail, in which case it has satisfied the original goal, or fails to be able to satisfy one of the goals listed in the tail with the current values of the variables. Note that failure means that it goes through the entire list of rules and can't find one whose head can be made to match the current sub-goal. In the event of failure to satisfy a sub-goal, Prolog backtracks to a previous goal and continues down the list of rules looking for a different rule to apply.

Yikes! Note that each goal is either directly satisfied by finding a clause with no right part—a "fact"—that matches it with suitable substitution of variables, or spawns a sub-goal, to which the entire unification mechanism is applied recursively.

The Prolog interpreter has the feature that after satisfying a query, you can hit `enter` to be done with it, or hit a semi-colon to ask for another solution.

It can be useful, especially when trying to understand the unification mechanism, to trace certain predicates. If you do

```
trace(f).
```

then Prolog will detail its attempts to satisfy any goal involving the predicate `f`.

⇒ If we had more time and energy in this rather bizarre moment in history, I would put a question on Test 3 to see how well you understand the unification mechanism, but instead let's just look at one example in the video `video2.mp4`, which is similar to a test question from last time I taught the course.

Here are some Prolog rules for two predicates:

```
edge(a,b).
edge(a,d).
edge(b,c).
edge(d,b).
edge(d,e).
edge(e,c).

path(X,Y) :- edge(X,Y).
path(X,Z) :- edge(X,Y), path(Y,Z).
```

In the video we will try to figure out by hand the results of each of the queries below, and will then see what Prolog actually does (assuming we keep hitting semi-colon on each until all the possibilities are exhausted).

```
edge(X,b).

edge(X,Y).

path(X,Y).
```

Built-in Predicates

Prolog has some built-in predicates, especially involving arithmetic, that allow us to satisfy predicates using numeric atoms instead of the unification mechanism.

For example, you can do something like

```
X is 2*3.
```

And Prolog will respond with

X = 6.

Example: the Factorial Function

Here is a reasonable-looking collection of rules to implement the factorial function (in `Prolog/factorial.pl` at the course web site):

```
% factorial(N,F) means F is N!

factorial(0,1).

factorial(N,F) :- factorial(N1,F1), N1 is N-1, F is N*F1.
```

⇒ In the video `prolog3.mp4` I will look at how these rules misbehave and fix them.

Lists

⇒ The video `prolog4.mp4` talks about the following material, focusing on programming with lists in Prolog.

Because lists are such a powerful data structure, Prolog supports special predicate forms for working with lists.

You can represent, say, a four element list as `[a,b,c,d]`. A list is viewed as a single value in terms of substituting for variables in the unification mechanism. The empty list is `[]`.

You can use the notation `[H|T]` to represent a list with `H` as its first element and `T` as the rest of the list. This notation allows us to effectively do any sort of recursive stuff in Prolog that we did in Lisp.

⇒ Write a predicate `ourMember` such that `ourMember(X,Y)` is satisfied if `X` is a member of the list `Y`.

⇒ Write a predicate `len` such that `len(X,N)` is satisfied if the list `X` has `N` items.

We can actually use as many initial items in a list as needed, as in

`[A,B,C|X]`

to match a list with `A`, `B`, and `C` as, respectively, the first, second and third items of a list, with `X` as the rest of the list (which may be empty).

Some Examples of Prolog Programming

Here are some fancier examples of Prolog programming.

The Zebra Puzzle

⇒ Look online at the Wikipedia entry for the "zebra puzzle" (referenced as appearing in *Life International* in 1962) and look at how facts such as those are given in this Prolog code (available at the course web site):

```
% right(X,Y,L) means "X is immediately to the
%   right of Y in the list L"
right(X,Y,L) :- append(_,[Y,X|_], L).

%  nextTo(X,Y,L) means "X and Y are adjacent in the list L"
nextTo(X,Y,L) :- right(X,Y,L).
nextTo(X,Y,L) :- right(Y,X,L).

% the 5 tuples have the format
%   [country of origin, pet, beverage of choice, brand of cigarettes,
%          color of house ]

start(S) :- length(S,5),
  member([english,_,_,_,red],S),
  member([spanish,dog,_,_,_],S),
  member([_,_,_,coffee,green],S),
  member([ukrainian,_,_,tea,_],S),
  nextTo([_,_,_,_,green],[_,_,_,_,ivory],S),
  member([_,snails,oldGold,_,_], S ),
  member([_,_,kool,_,yellow],S),
  S=[_,_,[_,_,_,milk,_],_,_],
  S=[[norwegian,_,_,_,_],_,_,_,_],
  nextTo([_,_,chesterfield,_,_],[_,fox,_,_,_],S),
  nextTo([_,_,kool,_,_],[_,horse,_,_,_],S),
  member([_,_,luckyStrike,orangeJuice,_],S),
  member([japanese,_,parliaments,_,_],S),
  nextTo([norwegian,_,_,_,_],[_,_,_,_,blue],S),
  member([_,zebra,_,_,_],S),
  member([_,_,_,water,_],S).

```

⇒ Test this program and find all the solutions that it produces. Then study the code and figure out how it works.

Here is a sorting algorithm implemented in Prolog:

```
% qsort(A,B) means that A quicksorted yields B

qsort([],[]).

qsort([H|T],S) :- partition(H,T,L,R),
                  qsort(L,L1),
                  qsort(R,R1),
                  app(L1,[H|R1],S).

%  partition(A,B,C,D) means
%   A is an item, B is partitioned into C and D with
%   all guys in C < A and all the guys in D >= A

partition(P, [A|X], [A|Y], Z ) :- A<P,
                                  partition(P,X,Y,Z).

partition(P, [A|X], Y, [A|Z] ) :- A>=P,
                                  partition(P,X,Y,Z).

partition(P,[],[],[]).

%  app(X,Y,L) means L is the list X followed by the list Y
app([],Y,Y).
app([H|T],Y,[H|W]) :- app(T,Y,W).
```

⇒ Test this program and verify that it seems to work. Then study the code and figure out how it works. A good technique is to write down precisely the meaning of each predicate, such as has been done with the predicate `app`.

Exercise 16 (practicing recursive, list-based Prolog)

Here are a couple of fairly simple programs for you to create.

a. Write rules to implement a Prolog predicate `locate( Target, List, Index )` that means "`Target` is at position `Index` in the `List`," assuming that `List` is a list of one or more integers that includes `Target`. Use 0 for the first position in the list.

   For example, the query

   ```
   locate( 3, [1, 2, 3, 4, 5, 6], Index).
   ```

   should produce `Index = 2`.

b. Write rules to implement a Prolog predicate `mix( X, Y, M )` that means "`M` is a list formed by alternating items from lists `X` and `Y`, which are assumed to have the same number of items."

   For example, the query

   ```
   mix( [1,2,3], [4,5,6], M ).
   ```

   should produce `M = [1,4,2,5,3,6]`.

   Email me your rules in an attached file (both parts can be in the same file unless you use overlapping names of predicates for the two parts with different meanings) named `exercise16.pl`.

Project 7 (a slightly more interesting Prolog program)

Your job on this project is to create a text file named `project7.pl` that contains some Prolog rules such that I can consult your file and be able to enter any list of numbers and have the next number in the sequence (assuming that the given numbers come from evaluation of a polynomial and that there are enough of them—your rules don't have to check for this) be figured for me by doing something like

```
nextItem( [146, 394, 882, 1730, 3082], N).
```

with Prolog responding

```
N = 5106
```

Your `nextItem` rules must work by doing a technique that works when the given numbers come from a polynomial, which is to do subtractions successively as shown here for the example sequence given above:

146 394 882 1730 3082

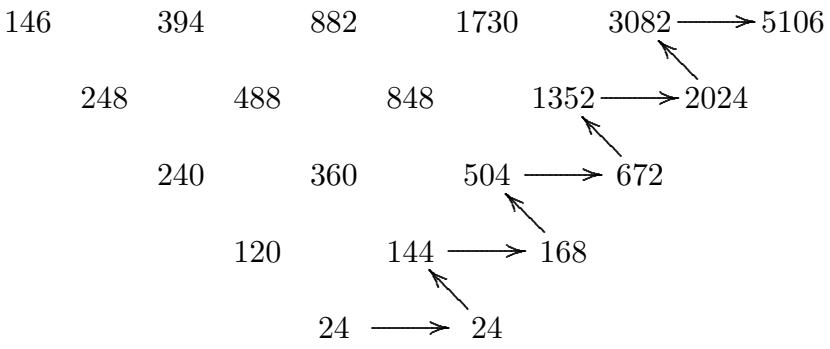
248 488 848 1352

240 360 504

120 144

24

Once a row with only one item is reached, then we know (given the assumptions that the original sequence came from a polynomial and that it contained enough numbers to be fair for the degree of polynomial) that the pattern for that row is constant, so we know the next value in that row. Given the next value in a row, we can add it to the last value in the row above to compute the next value in the row above. For the example, we figure these next values in each row:



Your `project7.pl` must only use fundamental Prolog constructs—no built-in predicates other than arithmetic and list syntax (using square brackets, commas, and the vertical bar).

As a possible hint, you might want to give rules for these predicates (in addition to the `nextItem` predicate):

> `rowBelow( A, B)`
>
> meaning that `A` is a list of numbers and `B` is the row below it in the chart, and
>
> `myLast( A, X )`
>
> meaning that `X` is the last item in the list `A`.

As usual, email `shultzj@msudenver.edu` with your file `project7.pl` as an attachment.
