CSE305 Week 14: Prolog, Past, and Future

[The lecture as originally planned, putting in these notes some of what was meant to be shown between Sebesta's chapter 16 slides and Prolog on `timberlake`.]

A rough way to convey the norm of "declarativeness" is to associate some attitudes to programming language styles in regard to a specification of code requirements.

- "Develop Spec, Then Program": *traditional programming languages*. (In O-O languages, the class hierarchy integrates the spec.)
- "Develop Spec Hand-in-Hand With Program"---via
  - Rapid Prototyping: *scripting languages* ("form follows function")
  - Magnifying Type Into Spec: *functional languages* ("function follows form")
- "The Spec Is The Program": *declarative languages*.

On the third point, we can consider SQL as well as Prolog to be a declarative language. (See April 23, 2023 item at *365 Data Science*.) Both SQL and Prolog are principally set up as "query systems" where there is a database of items and rules, and the user states a goal---ideally without expending work on an algorithm to achieve it. They system then finds an algorithm by which to solve the problem.

SQL is based on the relational database model, but is not so overtly set up for *inference* of relations the way Prolog is. In most frequent use, information flows from the system to the submitted query. Prolog promotes more back-and-forth between them.

What we will emphasize about Prolog is the *analysis of information flow*. Let's start with the kind of simple examples in Sebesta section 16.6 with this in mind---I've added a couple to the end.

```
female(shelley).
male(bill).
female(mary).
male(jake).
father(bill,jake).
father(bill,shelley).
mother(mary,jake).
mother(mary,shelley).
father(jake,tina).
mother(shelley,robert).
```

These are in a file `SebEx.prolog` in the Prolog/ folder on the course webpage and also in `~regan/cse305/LANGUAGES/PROLOG/LECREC23/` on timberlake. If you copy it to your home system or your own working folder on `timberlake`, you can load it by doing

```
prolog
```

and then enter at the *query prompt*, which may look like | ?-

```
| ?- consult('SebEx.prolog').
```

always ending in a period. On Timberlake and many systems you can also do `['SebEx.prolog'].` You will get a compile message---ignore any warnings about clauses not being together.

The query prompt is only a query prompt. If you want to enter new facts or rules, you can enter `consult(user).` or `[user].` to get to the entry prompt, which doesn't have a ? in it. You can enter facts and rules with new names, but can't "extend" old definitions---you will get a warning about trying to redefine something with a lot of confusing options. Sticking with the example file and the query prompt for now, try the query

```
father(bill,mary).
```

Prolog says `no`. But to

```
father(bill,jake).
```

it's a yes, since we entered it as a fact. Same for `male(bill).` and `male(jake).` But

```
male(robert).
```

gets a `no` because we didn't enter it as a fact, regardless of our idea of real-world truth. This again is the **Closed World Assumption** of Prolog coupled with **Negation as Failure** (failure to prove). Now try

```
mother(mary,X).
```

Prolog fills in `X = jake` and then repeats the query ? prompt on the same line. If you want to see if `mother(mary,X)` holds for anyone else, type **;** ---whereupon Prolog comes up with `X = shelley`. A further **;** makes Prolog answer **no**. This is because Prolog has reached the end of its search. If you do `mother(X,shelley)` you will get only `mary`; you will not get `robert` because the `mother` relation is not symmetric. However, let us do:

```
mother(X,Y).
```

You will get all three possibilities this way---your output may look like this:

```
| ?- mother(X,Y).
X = mary,
Y = jake ? ;
```

```
X = mary,
Y = shelley ? ;
X = shelley,
Y = robert ? ;
no
```

Now let us change our mindset away from `mother` and `father` being built out of individual facts and think of them as "global predicates" that are given as rule definitions. One way this could come about is if our database were initially set up this way (SebEx2.prolog):

```
female(shelley).
male(bill).
female(mary).
male(jake).
parent(bill,jake).
parent(bill,shelley).
parent(mary,jake).
parent(mary,shelley).
parent(jake,tina).
parent(shelley,robert).
mother(X,Y) :- female(X), parent(X,Y).
father(X,Y) :- male(X), parent(X,Y).
```

It is OK to do **`consult(user).`** or just **`[user].`** and add the rule

```
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
```

Here it is OK to leave Y "unbound"---Prolog treats it existentially as something to fill in to try to satisfy the predicate.

Now `mother(X,Y)` and `father(X,Y)` feel like general functions. The question is, how will we intend to use them?

1. We will give `X` as a definite value and try to find a child `Y`.
2. We will give `Y` as a definite value and try to find a mother or father `X`.
3. We will leave both `X` and `Y` unspecified and be happy to find any combination that works.

The last is most general, but a single answer may be least useful. We usually want answers for a particular person. This brings us to the information flow topic.

Flow and Cut

In fact, we can think of the three modes of use, say specifically for mother(X,Y), as corresponding to three different functions. At least they have three different Prolog header comments according to which of three modes a variable has: **+** meaning expected as read-only input, **-** meaning write-only output, and **?** for neither **+** nor **-**, could be used either way and Prolog can re-jigger it at will:

1. % `mother(+X, -Y)`: given X, find Y such that X is mother of Y, or no if none exists.
2. % `mother(-X, +Y)`: given Y, find X such that X is mother of Y, or no if none exists.
3. % `mother(?X,?Y)`: function should work when X and Y are left as variables to fill.

The single code above works in all three modes. But suppose we had coded `mother(X,Y)` this way:

```
mother2(X,Y) :- female(X), !, parent(X,Y).
```

In mode 1 this is innocuous: if we give a definite value for `X` then once we check `female` we never need to re-check it, so the cut `!` is technically justified. But in mode 2 where `Y` is given, the system will try only one female (in a perhaps-arbitrary order) and may miss the correct one. Likewise in mode 3 with repetition, you will get only one mother's children.

OK, here the cut is completely unnecessarily---and it should be used sparingly when at all---but issues of flow are unavoidable in other cases. One is that Prolog arithmetic is (supposed to be) unidirectional:

```
Y is X + 3.
```

requires `X` to have a known value at the time of invocation. Prolog will not not take Y and infer that X is Y - 3. (Or does it?) This also shows that the idea of flow has its own complications:

> X may start out as a "free variable" but we have to be able to tell that by the time this is executed it will have been given a value.

Thus "**is**" is assignment, not equality, but even as assignment it has limitations:

```
X is X + 3.
```

does not do `X += 3`. By itself it is an instantiation error, which points up a crazily immediate difference from the OCaml interactive system (and that of Python etc.):

> Prolog variables do not retain their values between queries.

Within a rule they do: **`X is 4, X is X + 3`**. does not give an instantiation error, but says no because now **`is`** does get treated as the equality predicate. Safe policy: treat **`is`** as right-to-left value flow only. Here is an example from Sebesta of how arithmetic guides flow:

```
speed(ford, 100).
speed(chevy, 105).
speed(dodge, 85).
speed(volvo, 80).
time(ford, 20).
time(chevy, 21).
time(volvo, 24).

distance(X,Y) :- speed(X, Speed), time(X, Time), Y is Speed * Time.
```

This will give Y = 2000 as the answer to `distance(ford,Y).` and Y = 2205 for `distance(chevy,Y).` But `distance(dodge,Y)` just gives **`no`** because there is no way to satisfy the middle term on the right-hand side of the rule from the given facts.

The point about *flow* here is that you can get the distance Y given a definite value for the car X. But can you give a distance Y and get the car that travels that distance? Here it is OK because Speed and Time will be always be instantiated by the time `Y is Speed * Time` is reached. Prolog will backtrack thru the car data trying to find numbers that multiply to Y. What it won't do is try to factor a given Y like 2205. Nevertheless, the intended usage pattern can be crisply summarized as:

```
% distance(+X, -Y): Distance traveled by car X.
```

Lists and Recursion and Flow

Literal lists in Prolog use square brackets and comma not semicolon. To "save" a list in a variable, give it as argument to an "atomic predicate":

```
newList([apple, prune, grape, kumquat]).
```

Unlike in OCaml, you can save multiple lists to the "same variable" by entering

```
newList([apricot, peach, pear]).
```

Prolog uses vertical bar **`|`** not **`::`** to pattern-match on lists. Another difference is that to emulate **`x::y::rest`**, Prolog does **`[X , Y | Rest]`**. The vertical bar thus separates elements from the sub-list. Here is a predicate to append two lists (could also be called concatenate):

```
% append(L1,L2,L3): L1@L2 = L3.
append([],L,L).
append([H | Tail1], L2, [H | Tail3]) :- append(Tail1, L2, Tail3).
```

This can work in multiple ways, but the functional usage pattern is **`append(+L1,+L2,-L3)`**. If you do

`append(X,Y,[a,b,c]).`, Prolog will laboriously iterate through all the possible ways of breaking the list into two pieces.

```
rev([], []).
rev([H | Tail], L) :- rev(Tail, Result), append(Result, [H], L).
```

Note that we did have to put `H` in brackets to make it a list, not a bare element---Prolog keeps some record of types under the hood. This definition clearly speaks the flow **`rev(+L1, -L2)`**.

There is one subtlety with `member` that Sebesta does not mention:

```
member(E, [E | _]).
member(E, [H | Tail]) :- member(E, Tail).
```

If and when the base-case clause is satisfied, we might not want to backtrack to see if maybe `Tail` has duplicate copies of the element `E`. We could stop that by inserting a cut:

```
member(E, [E | _]), !.
member(E, [H | Tail]) :- member(E, Tail).
```

This does seem to have the intended usage `member(+E, +L)`, but now we mean that we expect `E` to be already given when the rule is encountered. Now hark back to the idea of defining

```
mother3(X,Y) :- member(X,L), parent(X,Y)
```

where L is a list of females. Now maybe we don't want to cut after member. It depends on whether the intended usage pattern is `mother3(+X, -Y)` or `mother3(-X, +Y)`. If the former, the cut could be put in as

```
mother3(X,Y) :- member(X,L), !, parent(X,Y)
```

rather than in the definition of `member` itself. All this makes the analysis of information flow for efficient and well-targeted Prolog systems more complicated than the simple norm of "declarativeness" would seem to entail.

Caveats of Prolog

[Much already noted above; see Sebesta's slides for summary.]

[The intended "global summary" of the course to end this lecture will be presented as exam review.]