Lab 1

6 Sep 2007

Today we'll get an introduction to the "logic programming" paradigm of Prolog. Since Thursday is Thursday this year, I don't really have time to give you a lot of background, so we'll make this lab a little more discovery-based. Don't be afraid to ask questions during the lab, of course, but try to understand what Prolog is doing here, and we'll talk about it more tomorrow.

1 Prolog

Prolog is a computer language quite different from any you've seen before; it is a language for specifying (a subset of) predicate logic. The power of Prolog is that the reasoning part is all built-in; the "program" you write is just a set of assertions about what is true. It maps directly to more conventional logic notation (which you may have seen elswhere, but which in any case we'll talk about tomorrow), though it appears superficially different.

The simplest 'sentence' of Prolog that you can write asserts that a certain predicate is true of a certain entity. For instance,

```
rainy(today).
```

indicates that the `rainy` predicate is true of `today`. We could also say

```
rainy(yesterday).
```

which asserts the same predicate to be true of a different object, or

```
cloudy(today).
```

NOTE: there are two important, if unobvious, syntactic features of these three lines. There are no spaces between the predicate and the parenthesis; and each line ends in a period. Getting either one wrong yields an unhelpful syntax error.

Create a new file named `weather.pl`, and type the above three lines into it. Save it. In another window, start the program `pl`, which opens the Prolog interpreter. It prints out a header message and then gives you a prompt: "`?-`". Load the file you just typed in using Prolog's square bracket operator:

```
?- [weather].
```

Did it compile correctly? Hopefully the answer is

```
Yes
```

but if not, go and fix the file (and reload it) before you continue.

Now that a few facts are loaded up, you can query Prolog about them.

```
?- rainy(today).

Yes
?- rainy(yesterday).

Yes
?- rainy(tomorrow).

No
```

Seems awful certain of that, eh? Think about that "no" answer—it's important, and we'll talk about it in class later.

Also, query whether `cloudy(yesterday)`. But we're about to fix that.

2 Rules

As a general rule, even if we don't have explicit evidence of cloudiness, if it rained on a given day we'd be prepared to say it was cloudy, too. To express that in Prolog, we need a new kind of thing: a rule. Actually, two: in order to express rules of any interest, we also need variables.

Speaking in English, we would probably express the rule as something like "if a day was rainy, it was also cloudy." If we had to avoid pronouns (like 'it'), we'd probably just introduce a variable: "For any given day X, if X was rainy, then X was cloudy." We express this in Prolog as

```
cloudy(X) :- rainy(X).
```

which is backwards from what you might expect; the mnemonic is that the symbol '`:-`' looks a very little bit like a left-pointing "if-then" arrow, if you kind of squint at it. Another syntactic note: it's actually built into the syntax that variables must at least start with a capital letter. (Conversely, non-variables like `today` and `yesterday` and for that matter `rainy` all have to start with a lowercase letter.)

So go ahead and add that rule to your file, and save it again. In the Prolog window, you'll have to reload the file (the up arrow is handy here), but now you should be able to get the expected answer

```
?- cloudy(yesterday).

Yes
```

even though that was never explicitly stated: Prolog is inferring the correct answer.

3 Variables in queries

You can actually use variables outside of rules. Make the query '`rainy(X).`' and see what happens (don't forget the period). It's giving you one possible value of X that would make that statement true. Hit enter again, and Prolog cheerfully confirms that `Yes`, it found a value that worked.

Now try '`cloudy(Y)`'. Note that the variable name doesn't have to be the same one used elsewhere. And this time, instead of hitting enter, press the semicolon key—this says you're not satisfied with that completion, so it tries a different one that also produces a true statement (either explicitly given as fact, or inferred via rules). Keep pressing the semicolon and eventually Prolog will tell you that `No`, it can't find any more ways to fill in the variable.

4 More complexity

Start a new file called `family.pl` and add to it the following facts:

```
mother(betty, alex).
mother(betty, chris).
mother(betty, jordan).
mother(debbie, loren).
```

Prolog lets you have predicates with as many arguments as you want; you just have to separate them by commas. The interpretation of these is that the entity represented by the first argument is the mother of the entity represented by the second. Save, load this file into the Prolog interpreter, and try to query whether `debbie` is `alex`'s mother, or who-all `betty` is the mother of.

The comma lets us make more complex predicates, but it also opens up a world of complexity in the rules, as well. Consider the following rule:

```
sibling(X,Y) :- mother(Z,X), mother(Z,Y).
```

Because you know the English-word meanings of 'sibling' and 'mother', you can probably see what this is supposed to do. The way it works is this: if Prolog can find a single entity to substitute for *both* occurrences of the variable Z, so that X and Y have the same mother, then X and Y must be siblings. Type that rule in, save, reload the file again, and this time make the following queries:

- Are `alex` and `jordan` siblings?
- Are `loren` and `chris` siblings?
- Who are all of the siblings of `jordan`?

Oops.

5 Operators

Mostly, Prolog relies on this fact-and-rule system to do everything, but some things are just a bit too clunky, so some operators are provided.

```
?- alex = alex.

Yes
?- alex = chris.

No
?- alex \= chris.

Yes
```

Why did they pick '`\=`' for their not-equals operator? I'm not sure. The `!` does something else, so maybe they wanted to reserve it. Back when Prolog was created, there wasn't really any one standard not-equals operator.

In any case, modify the `sibling` rule to be a non-reflexive relation, and test it.

6 More play

If you add the rules

```
father(frank, loren).
father(frank, alex).
```

and ask about siblingness, Prolog will tell you

```
?- sibling(loren, alex).

No
```

but that's not right. Add a rule that will fix it.

Add the following predicates to the system, one at a time, making sure to test each one (adding base facts as necessary to create test cases) before moving on. If there is some other predicate you want to implement that would make these more straightforward, go ahead and do that.

- grandfather
- cousin
- aunt
- ancestor (this one is tricky)

7 Handing in

Make sure you get through at least "cousin" before class tomorrow. Whatever you've finished of the family-tree rules by the end of the lab period, though, hand it in:

```
handin cs262 lab1 family.pl
```