CSC 231 Spring 2015 **Wake Forest University**

Lab 8: Introduction to Prolog

Overview of Lab

This lab is intended to gently introduce you to Prolog. Prolog is a logic-based language, unlike Erlang (functional) or C++ or Java (imperative/state-based). This lab will introduce the basics of Prolog – *facts, questions, variables,* and *definitions.* A more detailed analysis of the Prolog process will be covered in later labs. For now, try to get a hang of the syntax basics.

Installation of Prolog Interpreter

The Prolog interpreter that will be used for the next few labs is SWI-Prolog. If you are using bally, prolog is already installed. If you are using your own computer, you will likely have to install prolog. See me if that is the case.

Running the Prolog Interpreter

To run Prolog under Ubuntu, type *prolog* on the command line. This will place you inside the Prolog interpreter. To exit, type *Control-D*.

You should actually use two windows in working with the Prolog interpreter. You will need to edit text files to hold facts and definitions in one window (using gedit or sublime for example). Once you have a set of facts and definitions made, you should then load these facts and definitions into the interpreter in another window and run queries/programs against the loaded information.

Comments

Comments in Prolog are written as ***/* comment here */***.

Facts

Very simply put, Prolog is a language that supports writing programs whose focus is on objects and the description of relationships between objects.

Essentially, you will be writing axioms from which further statements can be proven through inference. The end process of logic programming is that you state a goal, and then the implementation system uses a search through the axioms and an application of inference rules to determine whether or not that goal can be satisfied.

At a basic conceptual level, programming in Prolog consists of three steps:

1) declaring some facts about objects and their relationships
2) defining rules (definitions/predicates) about objects and their relationships which are applied over the set of facts
3) asking questions about objects and their relationships that can be answered by the facts and application of the rules

Facts are a core part of the Prolog language. In most texts, you will hear of the Prolog system working with a *database* of *clauses*. *Clauses* are individual facts, entered by you and assumed to be true for your domain, and the *database* is the collection of all such facts. A *fact* is essentially a statement that consists of two parts: a relationship and a set of objects involved in that relationship. As an example, the following fact indicates that the object “john” has the relationship “likes” with the object “mary”.

likes(john, mary).

The Prolog rules that constrain how you can write facts are as follows:

1) The names of all relationships and objects must begin with a lowercase letter.
2) The relationship is written first and the objects are separated by commas and enclosed within parentheses.
3) A period must come at the end of the fact.

Other than the above syntax rules, any relationships can be described. The semantics and meaning are up to you. There can be an arbitrary number of objects used as arguments – sometimes only one is needed, such as in the following facts:

valuable(diamonds).
valuable(gold).

Order of objects *does* matter, as the following statements describe two different facts:

likes(john,mary).	John likes Mary.
likes(mary,john).	Mary likes John.

The database does not even need to contain statements that are true in the real world – they just need to represent statements that are true in the domain you are interested in reasoning over. You should save all the facts that you are interested in working with in a text file. To load the facts into Prolog so that they can be reasoned over, use the following command at the interpreter prompt:

['filenameToLoad'].

replacing *filenameToLoad* with the name of the file you are interested in using.

Questions

Once a database of facts is established in Prolog, it is possible to answer questions about the facts. After loading a set of facts into the interpreter, questions can be directly entered in the interpreter. To ask the most basic questions, use the same syntax as if you were entering a fact. As an example, if you want to know if there is a fact that indicates *john likes mary*, the following question could be entered:

Interpreter >> likes(john,mary).

If a fact that matches the question exists in the database, then a ***yes/true*** answer is returned from the Prolog interpreter. If a matching fact can't be found, a ***no/false*** answer is returned.

An Example
At this point, let's define a basic family relationship database and query that database. In a file called ***lab8Family.plg*** (available through Sakai), the following facts have been entered.

male(william).
male(billy).
female(mary).
female(laura).
parent(william,billy).
parent(william, laura).
parent(mary,billy).
parent(mary,laura).

Questions that might be of interest, in both English format and Prolog format, and the responses provided by Prolog, are:

Is william male?

```
male(william).       // does a fact exist that states that william is male?
true                 // would this still hold if there was another fact
                     // female(william). ?
                     // Yes! We just have to verify there is a(ny) fact supporting the
                     // original question.
```

Is laura female?
female(laura).
true

Is william female?
female(william).
false

Is william the parent of laura?
parent(william, laura).
true

Is billy the parent of laura?
parent(billy,laura).
false

Conjunctions

Conjunctions allow more complicated queries (questions) to be asked. Think of a conjunction as an AND of facts to match. A conjunction is entered in the Prolog interpreter by entering queries separated by commas.

An Example

Using conjunctions, we can start to query things such as: “is william the parent of billy and laura?” (alluding to the fact that laura and billy might be siblings).

parent(william, laura), parent(william, billy).
true

Another example is the question: “are mary and william both parents of billy?” (alluding to the fact that mary and william might be husband and wife).
parent(mary, billy), parent(william, billy).
true

A third conjunction question: Are william, mary, billy and laura all male?
male(william), male(mary), male(billy), male(laura).
false

How Prolog Answers Queries

Prolog answers queries by searching through the database of facts. Given a simple single-fact query, you can think of the Prolog search as checking the query against every fact in the database. If a matching fact is found, a ***yes/true*** is returned. If all facts have been checked and none match the query, a ***no/false*** answer is returned.

Things get more complicated when a conjunction is in the query. To handle conjunctions, the interpreter attempts to match each part of the query from left to right. It will take the first part of the query and look for a match. If no match is found for the first part, no possible match can be found satisfying the conjunction (relate this to: having one part of an AND be false makes the whole thing false) so the search stops. If a match is found, the second part of the conjunction is then used in searching for a match, starting back at the beginning of the database. This continues until a match is found, wherein successive parts of the query to the right are then searched for. If no match is found for the second item, the first part of the query is returned to and a search from where the original match was found is started as there may be other matches to the first query that do allow matches in the remaining queries. It is through this **backtracking** technique that a conjunction query can be successfully answered.

Variables

The basic questions described above are limited in that they can just check whether or not matching facts exist. More interesting questions require the use of variables in place of objects. If we would like to find all things that John likes, the following question can be posed to the interpreter.

likes(john, X).

Note that variables use uppercase letters and that they can be arbitrarily long words.

The interpreter, when asked a question with variables, will return, one-by-one, all facts that match the question. On each matching fact, the appropriate variables (in this case, the variable X, but in general, any variables occurring in the query) will be instantiated to each object in the fact that matches and the state of those variables will be printed to the screen.

If you are only interested in a single answer, you can hit ***return*** to go back to the interpreter prompt after the first answer is given. If you would like to see more answers, hit a semicolon **;** instead of return.

The backtracking search described above allows multiple answers to be found. Hitting a return effectively says "You can stop searching", hitting a semicolon says "I'm interested in more answers - please backtrack and find more matches and tell me what matched." Don't be surprised if Prolog sometimes returns the same answer twice – a goal may be satisfied through alternative ways of satisfying its subcomponents.

One could also add a fact to the database of ***likes(john, X).*** This is essentially making the statement that john likes anything and everything. A question, ***likes(john,math)***, would be satisfied during the database search because the *math* constant and *X* variable would match up.

An Example

To find all parents of billy, the following question with a variable would work. The value that the variable X takes on in each case where the statement can be satisfied is presented:

parent(X,billy).
X = william ;
X = mary ;
false [note this last false indicates there are no more answers/ways of satisfying the query]

In variable based queries, you should hit a semicolon after an answer has been presented. This will continue to show you all possible answers. The semicolons are printed as I hit them on the keyboard.

To find all males,

male(X).
X = william ;
X = billy ;
false

If you want to get out early (that is, only see one or two answers, for example), hit return instead of semicolon.

Inference Statements/Definitions

If we are just limited to facts and questions, then we are really limited to just making simple inferences about the facts that we have entered. To detail all the possible knowledge about a domain, we would have to explicitly add by hand many, many facts. To deal with this, Prolog allows a programmer to enter ***inference statements*** or ***definitions***. A definition is a logical statement that describes how to infer a fact from other facts. Definitions typically use variables so that they can represent statements over arbitrary objects.

As an example, given that facts dealing with a ***mother*** predicate over two arguments already exist in the database, a grandmother inference rule can be defined as follows:

grandmother(X,Y) :- mother(X,Z), mother(Z,Y).
grandmother(X,Y) :- mother(X,Z), father(Z,Y).

Note that multiple “definitions” for the same predicate can be given, much like in Erlang where multiple function implementations can be defined. Definitions are entered in the same text file as the facts that you are using in the domain.

It is also possible to write recursive definitions, such as the following code.

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).

However, you should be careful in the recursion, opting to make the recursive call deep enough into the query such that its parameters have been initialized to the point that infinite recursion won’t occur. For example, one doesn’t want to write ***ancestor(X,Y) :- ancestor(Z,Y), parent(X,Z)*** as answering a query(X,Y) with un-instantiated parameters requires an immediate call to the same query with two un-instantiated parameters. By re-ordering the inference rule to ***ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y)***, the ***ancestor(Z,Y)*** query will be instantiated with Z already set (given the previous matching of X and Z against a ***parent*** fact).

An Example
To continue with the family examples already discussed, definitions for mother might be defined as:

mother(X,Y) :- parent(X,Y), female(X).

Questions using the mother definition:

mother(mary,laura).
true
mother(william,billy).
false
mother(william,laura).
false

mother(mary,X).
X = billy ;
X = laura ;
false

mother(X,billy).
X = mary ;
false

A rule for defining siblings could be finding two objects who share a parent but which aren’t the same object:

siblings(X,Y) :- parent(Z,X), parent(Z,Y), X \= Y. (\= is not equals)

siblings(billy,laura).
true

siblings(william,mary).
false

siblings(william,billy).
false

Operators
Equality and arithmetic operators are built into Prolog. There are some syntactical differences in their usage from languages you might be familiar with. They are defined below:

X = Y	***equals***
X \= Y	***not equals***
X < Y	***less than***
X > Y	***greater than***
X =< Y	***less than or equal***
X >= Y	***greater than or equal***

Note that we are allowed to use infix notation (we could however, if we wanted, write these in the standard Prolog format ***>(X,Y)***).

2 > 3.
false

2 < 3.
true

2 = 3.
false

2 \= 3.
true

Standard arithmetic operators are also built in and can be used in infix notation. They are defined below:

X + Y	***addition***
X – Y	***difference***
X * Y	***product (multiplication)***
X / Y	***quotient (division) – floating point answers***
X // Y	***quotient(division) – integer answers***
X mod Y	***remainder – integer answers***

In Prolog, however, these are not automatically executed. To execute an arithmetic operator, a statement of the following form is required:

<***variable> is <value> <operator> <value>.***

The <> are not required – they are used to indicate you should replace <something> with an appropriate entry.

Z is 2 + 3.
Z = 5 ; // this statement is satisfied when Z = 5

Note that these mathematics operations, with a variable on the left-hand-side of the ***is*** and an expression on the right-hand-side, are always satisfied if in a conjunction and the value of the expression is the value assigned to the variable for the rest of the conjunction. Also important to note is that the right-hand-side of an *is* expression must always be instantiated – all variables in use must have a value before the expression can be evaluated. This prevents doing things such as ***5 is X+3.*** and expecting the system to return X is 2.
This leads to a style of programming where, to retrieve the results of doing a mathematics operation, particularly arithmetic, it is fairly common to pass in a return-value variable to a function which will be set to hold the answer returned from the function. So, instead of functions having return values explicit, we pass in a parameter we want to hold the returned value. The value of this variable will be printed to the screen when a "match" (the eventual assignment of a value) for the variable is found during the search over the database of facts and rules. Here is an example of using a return-value variable.

function(InputVariableA,InputVariableB,OutputVariableC) :- OutputVariableC is InputVariableA <someMathOperation> InputVariableB.

An Example
A very simple use of the style discussed above is the following example that makes use of comparison operators and math operators to write a function which only generates an answer when the two input values are positive.

```
sumIfBothPositive(A,B,C) :- A > 0, B > 0, C is A+B.

sumIfBothPositive(2,3,C).
C = 5;

sumIfBothPositive(-1,3,C).
false
```

Think about how these functions work to return an answer. Remember, Prolog is searching a database of facts, looking for matches. Definitions are "matched" when the queries on the RHS of the definition are all matched. If a definition is written as:

```
sumIfBothPositive(A,B,C) :- A > 0, B > 0, C is A+B.
```

and then called with:

```
sumIfBothPositive(2,3,C).
```

the 2 and 3 will be used to replace the A and B automatically. Thus, the query to match is 2 > 0, 3 > 0, C is 2+3. This will match instantaneously, as 2 is > 0, 3 is > 0, and the ***is*** statement always matches, settings the variable before the ***is*** to the value computed after the ***is*** keyword. The system will then print out to you what values of C satisfy the statement (C = 5). A negative value for A or B will lead to either ***A > 0*** or ***B > 0*** not being satisfied and the whole conjunction not being satisfied.

Using this style of mathematical programming, a definition for determining the number of n-choose-r combinations [an appropriate formula is (n!) / ((r!)((n-r)!))] could be written using the following code: ***combinations(A,B,C) :- factorial(A,X), factorial(B,Y), D is A-B, factorial(D,Z), C is X/(Y*Z)***, assuming factorial has been written correctly.

Note how this uses D, X, Y and Z to hold internal, temporary values, and how it sets C, the output value, to the result of doing a mathematics operation. It in effect computes factorial(A)/(factorial(B)*factorial(A-B)).

After these inference rules ("definitions") are entered in Prolog (via a text file), the following query results in the following correct output:

```
combinations(10,4,C).
C = 210;
No
```

Having touched on recursion earlier, and knowing that functions like ***factorial*** require recursion, let's think about recursive math problems. For math problems, we'll typically want to store the answer in an output variable. Also, for many math problems, the recursion base case is very simple in that the answer is exactly known.

Assume we are implementing ***power(A,B,C)***, where B is >= 0 and C should represent $A^{B.}$ A correct implementation in Prolog is:

power(A,0,1).
power(A,B,C) :- B > 0, Y is B-1, power(A,Y,Z), C is A * Z.

Note that we have a fact that says any variable, raised to the 0 power, is 1.
Then, we have another definition that says, if B is > 0, then we can compute A^B by computing A^{B-1} and multiplying the result that comes back by A. You will see this pattern a lot in mathematical recursion problems.

Note the check to make sure that B > 0 in line 2 of this definition. This is to prevent a match of a query against both the first and second rules (that is, to prevent something matching the base case being also satisfied by the 2nd rule).

The Prolog interpreter will actually complain about the previous implementation. For the line ***power(A,0,1).***, it will state

Warning:
Singleton variables: [A]

It is basically complaining that there is a variable defined, but it is never used. You have already encountered this in the Erlang language. A way to work around this problem (but which isn't required) is to change A to the anonymous variable, represented by an _ [an underscore]. Think of this as a **don't-care** variable.

Using the _ , the definition of power would look like this and the interpreter won't complain:

power(_,0,1).
power(A,B,C) :- B > 0, Y is B-1, power(A,Y,Z), C is A * Z.

One final note – do not use "is" as assignment; use it only to request that a math operation is triggered!

Takehome Problems:
A file with a set of basic facts and definitions you can build upon can be found on the class website as ***lab8Takehome.plg***.

1. Update the facts about families already defined in ***lab8Takehome.plg*** to represent your family or some family for which you know a lot of information. Familiarity with the family is important so you can test the correctness of your implementation on this problem. Extend the definitions you've already been given to include **9 new definitions: father(X,Y), daughter(X,Y), son(X,Y), sister(X,Y), brother(X,Y), grandfather(X,Y), aunt(X,Y), uncle(X,Y), and cousin(X,Y) [for aunt and uncle – only include those defined by blood relationship – your fathers sister, not your father's brother's wife].** If your definition requires multiple arguments, make the initial argument be the key argument (i.e. for son(X,Y), X should represent the object that is the son, and Y the parent).

2. Implement the following functions with base case facts and recursive definitions.
   1) isnatural(X) : X is a natural number if X = 0, or if X is > 0 and X-1 is a natural number.

   2) factorial(A,B): If A is 0, B is 1. For any number A > 0, B is the result of multiplying A times the result of computing the factorial of A-1.

   3) fibonacci(A,B): If A is 0 or 1,B is 1. For any number A > 1, B is the sum of the results from calling fibonacci on A-1 and fibonacci on A-2.

3. Translate the requirements for the BS in CS major (not the BA or minor) found on the following webpage: http://college.wfu.edu/cs/undergraduate-program/requirements-for-the-computer-science-major so that, given a list of facts of the form ***taken(Student,Dept,CourseNumber)***, I can use a function you write called ***fulfilledCSMajor(Student)*** and receive a ***true*** or ***false*** answer indicating whether or not the student has completed the requirements for the WFU BS in CS major. The BS in CS major requires 38 hours in CS – 26 of

these are from specific class required in the core (111→ 241, 399). The additional 12 hours you should consider as equivalent to "having completed four different courses at or above the 300 level". There are also requirements to take three math classes: 112, 117, and an option from the following math classes: 121, 205, or 206.

Your answers will likely be conjunction heavy. Try to write definitions that help abstract out some of the work. The ***lab8Takehome.plg*** file has three students you can use for testing: studentA, studentB, and studentC. studentA has taken enough courses to fulfill the BS major requirements, while studentB and studentC have not taken enough to fulfill the major requirements. Note that the courses that these particular students have taken are not exactly those that should be listed as the requirements for the major, instead they are examples of a set of courses which either fulfill or don't fulfill enough of the requirements.

Due Date: Thursday, 3/26/2015 at midnight (things submitted by 5pm, Wednesday, 3/25/2015 I will get back to you by the next lab; the others will get a best effort – probably over the weekend)

To submit via Sakai:

- **An updated lab8Takehome.plg file which contains the facts and definitions required to answer the above questions.**
- **A separate text file lab8Descriptions.txt containing a brief English description of your implementations for the questions above and why you believe your code correctly implements the desired functionality.**