**Artificial Intelligence I**

**Prolog**

**Objectives**

In this Lab, you will practice:

* using the SWI prolog system to run Prolog programs
* reading SWI prolog error messages and repairing errors
* using SWI prolog to perform queries
* writing Prolog programs to store facts and rules

(Note: Download and install [SWI Prolog](http://www.swi-prolog.org/download/stable))

SWI-Prolog offers a comprehensive free Prolog environment.

Knowledge databases that you might build with first-order logic suffer from a fatal problem: the logic is so sophisticated that finding answers efficiently (or even finding an answer at all) becomes difficult or impossible. First-order logic is so powerful that it can express almost anything, including very tricky or paradoxical statements. It's not easy to design algorithms to work with such systems.

Prolog is a programming language that allows us to "program" with declarative knowledge. It puts limitations on the kinds of logical statements we can write. These limitations are essential for allowing Prolog programs to work efficiently and always provide an answer (or determine there is no answer).

Because Prolog is a "Turing-complete" language, we can write any program in Prolog. Of course, a program that is easy to write in Java or C++ will, in all likelihood, be *very* difficult to write in Prolog, although it is possible. Prolog programs often feel like they're programs written in "reverse," since they are written declaratively rather than procedurally.

**Some quick concepts and examples**

There is probably no better way to introduce Prolog than to show some examples. But first, a few important notes:

* Prolog programs have two parts: a database (of facts and rules), and an interactive "query" tool; the database must be typed into a file.
* Prolog databases are "consulted" (loaded), and then the query tool is used to "make queries" (ask questions) about the database.
* How queries are answered is generally beyond the control of the programmer; Prolog uses a depth-first search to figure out how to answer queries; this is (generally) non-negotiable.
* "Programs" written in Prolog are "executed" by performing queries.

Let's make a database by creating some facts:

**parent**(tom, liz).

**parent**(bob, ann).

**parent**(bob, pat).

**parent**(pat, jim).

**female**(pam).

**male**(tom).

**male**(bob).

**female**(liz).

**female**(pat).

**female**(ann).

**male**(jim).

We type that into a file, maybe named family.pl, and "consult" the file in the query tool.

Now what? Well, we can ask questions:

?- female(pam).

true.

Ok, it's true that pam is a female.

?- parent(pat, ann).

false.

Right, pat is not ann's parent.

Now let's use variables. Variables always start with capital letters. Objects (like pam) must not.

?- parent(X, ann).

X = bob.

So Prolog has found an assignment for X that makes the statement true. We call parent a predicate (a 2-ary predicate because it accepts two inputs), and we always write queries by first stating a predicate.

Let's ask, "Who are all the parents?"

?- parent(P, C).

P = tom,

C = liz ;

P = bob,

C = ann ;

P = bob,

C = pat ;

P = pat,

C = jim.

(I typed the ; — we type that to get the next answer.)

If I don't actually care who the children are, I can replace C with \_ like so:

?- parent(P, \_).

P = tom ;

P = bob ;

P = bob ;

P = pat.

We get bob twice because he's the parent of two people.

Now let's ask, "Who are the female parents?"

?- parent(P, \_), female(P).

P = pat.

The comma means "AND." So the query requires that P is a parent and a female.

**Defining complex predicates (rules)**

The predicates shown above (parent, male, and female) are always true. They are called "facts."

A "rule," instead, is true only when some conditions are met. These conditions look like the last query above: they typically require several predicates to be true. The last query above is basically what defines a "mother." Let's make a rule (in our database file, family.pl):

**mother**(X) :- parent(X, \_), female(X).

Now for fathers. In the database, we add:

**father**(X) :- parent(X, \_), male(X).

Why the false at the end? The ; basically means "throw away the answer you just gave, and try again." Prolog finds a parent X (probably pat), but then that X does not satisfy the second part of the father rule, which is written male(X), so the answer is false for that X.

Notice that pat is the parent of jim and bob is the parent of pat. That makes bob a grandparent. Let's write a query to that effect:

?- parent(X, Y), parent(Y, \_).

X = bob,

Y = pat.

See? bob is a grandparent. Let's make a rule:

**grandparent**(X) :- parent(X, Y), parent(Y, \_).

And here are two queries:

?- grandparent(bob).

true.

?- grandparent(jim).

false.

Let's try defining the sister relationship:

**sister**(X, Y) :- parent(Z, X), parent(Z, Y), female(X), female(Y).

And some queries:

?- sister(ann, pat).

true.

?- sister(bob, pat).

false.

?- sister(X, Y).

X = Y, Y = liz ;

X = Y, Y = ann ;

X = ann,

Y = pat ;

X = pat,

Y = ann ;

X = Y, Y = pat ;

false.

What does X = Y, Y = liz mean? It means liz is, apparently, her own sister:

?- sister(liz, liz).

true.

Let's prevent that by adding another clause to the sister predicate (\= means "not equal"):

**sister**(X, Y) :- parent(Z, X), parent(Z, Y), female(X), female(Y), X \= Y.

Now, we get the correct behavior:

?- sister(liz, liz).

false.

?- sister(X, Y).

X = ann,

Y = pat ;

X = pat,

Y = ann ;

false.

These rules are essentially the following logical statements:

$$(\forall x)(\exists y)((Parent(x, y) \wedge Female(x)) \rightarrow Mother(x))$$

$$(\forall x)(\exists y)((Parent(x, y) \wedge Male(x)) \rightarrow Father(x))$$

$$(\forall x)(\exists y)(\exists z)((Parent(x, y) \wedge Parent(y, z)) \rightarrow Grandparent(x))$$

$$(\forall x)(\forall y)(\exists z)((Parent(z, x) \wedge Parent(z, y) \wedge Female(x) \wedge Female(y) \wedge x \neq y) \rightarrow Sister(x, y))$$

Notice how they all have the same form. Essentially, the variables in the rule (X or both X and Y in each of these cases) have a \(\forall\) in front (because the rule is true for all variables X or X and Y, should certain conditions hold), and the variables inside the rule have \(\exists\) in front because you only need to find one value for each variable to make the rule true. Notice we have \(\rightarrow\) not \(\leftrightarrow\) because there may be more than one way to satisfy a rule. For example,

**engineer**(X) :- mechanical\_engineer(X).

**engineer**(X) :- chemical\_engineer(X).

**engineer**(X) :- electrical\_engineer(X).

*% etc. etc.*

This is how we do "or" in Prolog: we make new rules. In each rule, we use commas for "and," but (generally) do not write "or" inside a rule.

**Negation as failure**

Negation is a little bit funny in Prolog. That's because negation is a little bit funny in life. How does anyone prove a negative statement? "I never read that book" — how can that be proved?

Let's say we want a predicate like "this person is not a father." Well, we know how to figure out who *is* a father: we have the father rule. How can we define a "is not a father" rule?

The first thing to note is Prolog does not allow us to use negatives in facts. We cannot write "not father(jim)."

Instead, to prove a negative in Prolog, we use a little trick: we assume everything stated in the database is *everything that exists* (this is the "closed-world assumption"). Anything not stated in the database does not exist. So, we can "prove" that jim is not a father by *failing* to prove that jim is a father (with our father rule).

**not\_a\_father**(X) :- \+father(X).

Some queries:

?- not\_a\_father(bob).

false.

?- not\_a\_father(jim).

true.

The \+ is the "negation operator." The idea is that \+father(X) is true whenever father(X) fails to be proved (recall that Prolog may search *every possibility* in order to prove something; thus, negation can be computationally expensive).

We cannot use \+ on the left-side of a rule or written as a fact because it makes no sense to say, "this will fail to be proven always." Only positive statements can be on the left of a rule or written as a fact.

This connection between "failing to prove" and "negation" is called "negation as failure." Negation as failure means, "proving the negation of something is equivalent to failing to prove that something" and depends on a closed-world assumption.

**negation-as-failure** *Also* **negation-by-failure** *n.* *Logic programming* **1** The reasonable assumption that *P* is false if one has failed to prove that *P* is true. **2** The unreasonable assumption that *P* is false if others have failed to prove that *P* is true. **3** *Drugs* The ineffectiveness of the "Just Say No!" campaign. — *The computer contradictionary*

**Lists in Prolog**

Besides atoms (ann, pat, bob, etc.) we can create lists. Let's make a simple predicate that says whether or not something is at the head (front) of a list:

**head**(X, [X|\_]).

The code [X|\_] means there is a list that starts with X and the rest (after the |) I don't care about (hence the \_). Here are some uses of this predicate:

?- head(a, [a, b, c]).

true.

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

X = a.

?- head(a, X).

X = [a|\_G228].

The \_G228 is a variable that Prolog created to represent the rest of the list; it has made no commitment about what \_G228 is, hence it can only describe it as variable \_G228.

Predicates can be recursive, which is exactly what is needed for the membership predicate:

*% X is a member of a list if the list only has the element X in it*

**member**(X, [X]).

*% X is a member of a list if the list starts with X*

*% (technically, the above statement is not needed; this*

*% statement suffices)*

**member**(X, [X|\_]).

*% Finally, X is a member of a list if it's a member of*

*% the tail of the list*

**member**(X, [\_|Tail]) :- member(X, Tail).

Example usage (I'm typing . after some responses because I don't want other possible answers):

?- member(a, [a]).

true .

?- member(a, [a, b, c]).

true .

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

X = a ;

X = b ;

X = c .

?- member(a, X).

X = [a|\_G231] ;

X = [\_G230, a|\_G234] ;

X = [\_G230, \_G233, a|\_G237] ;

X = [\_G230, \_G233, \_G236, a|\_G240] ;

X = [\_G230, \_G233, \_G236, \_G239, a|\_G243] .

In that last case, Prolog is *generating* lists that have a as a member somewhere. Pretty weird, huh? It's finding some way to make the predicate true.

**Prolog for databases**

Now we'll investigate using Prolog for representing and answering queries about data. Let's build a database of people and families. We want to say that certain people are born on certain dates, have jobs with particular incomes, are married, have children, etc.

We'll start with a predicate, family, that says a family is made up of an ID (a number), two parents, and zero-or-more children (a list). Each person has a first name and last name, birth date, and is either unemployed or has a job with a particular salary.

**family**(10392,

person(tom, fox, born(7, may, 1960), works(cnn, 152000)),

person(ann, fox, born(19, april, 1961), works(nyu, 65000)),

*% here are the children...*

[person(pat, fox, born(5, october, 1983), unemployed),

person(jim, fox, born(1, june, 1986), unemployed),

person(amy, fox, born(17, december, 1990), unemployed)]).

**family**(38463,

person(susan, rothchild, born(13, september, 1972), works(osu, 75000)),

person(jess, rothchild, born(20, july, 1975), works(nationwide, 123500)),

*% here are the children...*

[person(ace, rothchild, born(2, january, 2010), unemployed)]).

There are two families. Note that person is not a predicate, it's just a way of writing data. We have no way of asking person(X, Y, ...) because it's not a predicate. To access person data, we have to use the family predicate. Here are some examples:

?- family(38463, X, Y, Z).

X = person(susan, rothchild, born(13, september, 1972), works(osu, 75000)),

Y = person(jess, rothchild, born(20, july, 1975), works(nationwide, 123500)),

Z = [person(ace, rothchild, born(2, january, 2010), unemployed)].

?- family(38463, person(FirstName, LastName, \_, \_), \_, \_).

FirstName = susan,

LastName = rothchild.

?- family(ID, person(susan, rothchild, \_, \_), \_, \_).

ID = 38463.

Using this family predicate, we just put a variable in a location that we want to retrieve information about (such as family ID or a first name / last name of a person), and put \_ for all the slots we don't care about.

Now let's make it a little more interesting. Here is a married predicate that says if two people are married:

**married**(FirstName1, LastName1, FirstName2, LastName2) :-

family(\_, person(FirstName1, LastName1, \_, \_),

person(FirstName2, LastName2, \_, \_), \_).

**married**(FirstName1, LastName1, FirstName2, LastName2) :-

family(\_, person(FirstName2, LastName2, \_, \_),

person(FirstName1, LastName1, \_, \_), \_).

And its use:

?- married(tom, fox, ann, fox).

true .

?- married(X, Y, ann, fox).

X = tom,

Y = fox .

?- married(X, Y, A, B).

X = tom,

Y = fox,

A = ann,

B = fox ;

X = susan,

Y = rothchild,

A = jess,

B = rothchild .

Here is a predicate for calculating household income. We can calculate (simple math) using the is keyword:

**householdIncome**(ID, Income) :-

family(ID, person(\_, \_, \_, works(\_, Income1)),

person(\_, \_, \_, works(\_, Income2)), \_),

Income is Income1 + Income2.

And its use:

?- householdIncome(38463, X).

X = 198500.

?- householdIncome(X, 198500).

X = 38463.

?- householdIncome(X, 999).

false.

?- householdIncome(X, Y).

X = 10392,

Y = 217000 ;

X = 38463,

Y = 198500.

To facilitate further queries, we'll create an exists predicate that we can use to retrieve details about a person. It just pulls person information out of the various places it may appear in a family fact.

**exists**(Person) :- family(\_, Person, \_, \_).

**exists**(Person) :- family(\_, \_, Person, \_).

**exists**(Person) :- family(\_, \_, \_, Children), member(Person, Children).

We can use it like so:

?- exists(X).

X = person(tom, fox, born(7, may, 1960), works(cnn, 152000)) ;

X = person(susan, rothchild, born(13, september, 1972), works(osu, 75000)) ;

X = person(ann, fox, born(19, april, 1961), works(nyu, 65000)) ;

X = person(jess, rothchild, born(20, july, 1975), works(nationwide, 123500)) ;

X = person(pat, fox, born(5, october, 1983), unemployed) ;

X = person(jim, fox, born(1, june, 1986), unemployed) ;

X = person(amy, fox, born(17, december, 1990), unemployed) ;

X = person(ace, rothchild, born(2, january, 2010), unemployed).

?- exists(person(X, fox, \_, \_)).

X = tom ;

X = ann ;

X = pat ;

X = jim ;

X = amy .

?- exists(person(FirstName, LastName, born(\_, \_, 1972), \_)).

FirstName = susan,

LastName = rothchild .

?- exists(person(FirstName, LastName, \_, unemployed)).

FirstName = pat,

LastName = fox ;

FirstName = jim,

LastName = fox ;

FirstName = amy,

LastName = fox ;

FirstName = ace,

LastName = rothchild.

Let's get a little more sophisticated. Here is an interesting query:

*% Find all people born after 1980 who do not work at OSU*

?- exists(person(FirstName, LastName, born(\_, \_, Year), Job)),

Year > 1980, Job \= osu.

FirstName = pat,

LastName = fox,

Year = 1983,

Job = unemployed ;

FirstName = jim,

LastName = fox,

Year = 1986,

Job = unemployed ;

FirstName = amy,

LastName = fox,

Year = 1990,

Job = unemployed ;

FirstName = ace,

LastName = rothchild,

Year = 2010,

Job = unemployed.

Let's add a householdSize predicate. It "counts" the number of children and adds 2 (for the parents):

**householdSize**(ID, Size) :-

family(ID, \_, \_, Children),

length(Children, ChildrenCount),

Size is 2 + ChildrenCount.

We can use an interesting built-in predicate, findall, that finds a list of objects that match some "goal." Here is its use in a query:

*% First part of findall is what to collect,*

*% second part is the "goal" (a predicate),*

*% third part is the name of the resulting list*

?- findall(FirstName, exists(person(FirstName, \_, \_, \_)), Result).

Result = [tom, susan, ann, jess, pat, jim, amy, ace].

Of course, we can count the list:

?- findall(FirstName, exists(person(FirstName, \_, \_, \_)), Result),

length(Result, N).

Result = [tom, susan, ann, jess, pat, jim, amy, ace],

N = 8.

If we get a list of numbers, we can sum or average them. We'll need to load a Prolog library, however. Here is some code in a database file:

:- **use\_module**(library(lists)). *% load lists library for sumlist predicate*

**average**(List, Avg) :-

sumlist(List, Sum),

length(List, N),

Avg is Sum / N.

Here is a use:

?- average([1, 3, 4.2, 4.5], Avg).

Avg = 3.175.

Ok, now let's grab household incomes and average them. In query mode:

?- findall(Income, householdIncome(\_, Income), Result), average(Result, Avg).

Result = [217000, 198500],

Avg = 207750.