Intro to Prolog

Prolog, which stands for PROgramming in LOGic, is the most widely available language in the logic programming paradigm using the mathematical notions of relations and logical inference. Prolog is a declarative language rather than procedural, meaning that rather than describing how to compute a solution, a program consists of a data base of facts and logical relationships (rules) that describes the relationships which hold for the given application. Rather then running a program to obtain a solution, the user asks a question. When asked a question, the run time system searches through the data base of facts and rules to determine (by logical deduction) the answer.

Often there will be more than one way to deduce the answer or there will be more than one solution, in such cases the run time system may be asked to backtrack and find other solutions. Prolog is a weakly typed language with dynamic type checking and static scope rules.

Prolog is typically used in artificial intelligence applications such as natural language interfaces, automated reasoning systems and expert systems. Expert systems usually consist of a data base of facts and rules and an inference engine, the run time system of Prolog provides much of the services of an inference engine.

Prolog Syntax

Prolog is based on facts, rules, queries, constants, and variables. Facts and rules make up the database while queries drive the search process. Facts, rules, and queries are made up of constants and variables. All prolog statements end in a period.

Facts

A fact is a proposition and begin with a lowercase alphabetic character and ends in a period. Here are some sample facts:

```
sunny.
```

This says that *sunny* is true.

```
superhero(spiderman).
```

This says that spiderman is a superhero. Note the lowercase. This distinction is important, because we'll use an initial uppercase letter to indicate a variable.

```
eats(spiderman, pizza).
```

This says that spiderman eats pizza.

Each fact that we enter describes the logical "world" that comprises the database of knowledge we can then reason over.

Rules

A rule is an implication just like we used in forward chaining. In a rule, we can use boolean operators to connect different facts. The symbols used in prolog are as follows:

Predicate Calculus	Prolog	
$\wedge$	,	(comma)
$\vee$	;	(semicolon)
$\leftarrow$	:-	"if", note direction is left, not $\rightarrow$
$\neg$	not	

Here are some examples:

humid :- hot, wet.	Same as: hot $\wedge$ wet $\rightarrow$ humid
pleasant :- not(humid) ; cool.	Same as: $\neg$humid $\vee$ cool $\rightarrow$ pleasant

```
likes(bruno, spinach) :- not(likes(ted, spinach)).
```

Bruno likes spinach if Ted does not like spinach. Soon we'll extend this using variables instead of just spinach.

Variables

Variables are denoted in prolog by identifiers that start with an uppercase letter. For example:

```
likes(bruno, Food) :- not(likes(ted, Food)).
```

This says that Bruno likes any food that Ted does not like. Note that this is quite different from:

```
likes(bruno, food) :- not(likes(ted, food)).
```

The second statement is an atom named food, while the first is a variable that can represent any number of possible values.

Consider the following rules and facts:

```
likes(joe,Food) :-
      contains_cheese(Food),
      contains_meat(Food).
likes(joe,Food) :-
      greasy(Food).
```

```
likes(joe,chips).
contains_cheese(macaroni).
contains_cheese(lasagna).
contains_meat(lasagna).
greasy(french_fries).
```

In processing these rules, Prolog will **unify** the right hand side of the rule with any atoms that match the predicate. For the first rule, Food could be either macaroni or lasagna since both fit the criteria of contains_cheese. But then we AND this with contains_meat which leaves only lasagna. From these facts we can conclude that Joe likes chips, lasagna (cheese + meat), and french fries (greasy).

Queries

To query the database we can use prolog in an interpretive manner. Queries are generally made at the ?- prompt and consist of a predicate. For example, given the above data:

```
?- contains_meat(salad).
No

?- contains_meat(lasagna).
Yes

?- likes(joe,chips).
Yes

?- likes(joe, lasagna)
Yes

?- likes(joe, macaroni)
No
```

We can also make queries that include variables in them. Prolog will **instantiate** the variables with any valid values, searching its database in left to right depth-first order to find out if the query is a logical consequence of the specifications. Whenever Prolog finds a match, the user is prompted with the variables that satisfy the expression. If you would like to have Prolog continue searching for more matches, type ";" (meaning NO match yet). This may require prolog to backtrack to find some other matching expressions. If you are satisfied with the match and would like Prolog to stop, type "y" (meaning Yes, accept).

Here is a query that finds all foods that contain cheese:

```
?- contains_cheese(X).
X = macaroni ;
```

X = lasagna ;
No

Since I hit ";" each time to not accept the matches, Prolog exhausts the possible foods with cheese and returns no. If I type "y" instead Prolog will return yes:

?- contains_cheese(X).
X = macaroni
Yes

We could query the database to find all the foods that Joe likes to eat:

?- likes(joe, X).
X = lasagna ;
X = french_fries ;
X = chips ;
No

We could also query the database to find all the people that like to eat lasagna:

?- likes(X, lasagna).
X = joe ;
No.

Right now nobody in the database likes macaroni so we get the following:

?- likes(X, macaroni).
No

Using SWI Prolog

We've now covered enough that we can write somewhat interesting programs. In this class we will be using a free implementation of prolog called SWI Prolog. It is already installed on bigmazzy and the Windows machines. If you want to install it on your own Windows or Linux box, you can download it from http://www.swi-prolog.org/

Here we'll show how to get started with SWI Prolog in the Unix environment. It is very similar under Windows (in fact the Windows implementation just launches a Unix-like shell). In Windows, files are loaded relative to the prolog directory selected as the 'Home' directory during installation.

To start, type **swipl** to invoke the SWI Prolog interpreter.

mazzy> **swipl**
Welcome to SWI-Prolog (Version 4.0.11)
Copyright (c) 1990-2000 University of Amsterdam.

```
Copy policy: GPL-2 (see www.gnu.org)

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

At this point, the interpreter is ready for you to type in queries. We have no predicates entered though, so we can enter them by typing either
consult(user). or the shortcut of **[user].**

We can then continue by typing in the facts and rules we would like:

```
?- [user].
|: likes(joe,Food) :-
|:        contains_cheese(Food),
|:        contains_meat(Food).
|: likes(joe,Food) :-
|:        greasy(Food).
|:
|: likes(joe,chips).
|: contains_cheese(macaroni).
|: contains_cheese(lasagna).
|: contains_meat(lasagna).
|: greasy(french_fries).
|: % user compiled 0.00 sec, 2,336 bytes

Yes
?-
```

To end the user input, hit **control-d**. Back from the ?- prompt we can now make our queries:

```
?- likes(joe, X).
X = lasagna ;
X = french_fries ;
X = chips ;
No
```

To exit the prolog interpreter, hit **control-d** again.

```
% halt
```

It is often inconvenient to have to enter our data every time we start prolog and go back and forth between user mode and query mode. We can direct prolog to load its facts from a file instead of from the keyboard. To do so, but all of the prolog code you want into a

file in the working directory. SWI Prolog recognizes extensions of ".pl" as being prolog code. To load the file in prolog type either:

consult(filename). or **[filename].**

This will be the most common way you will input your data to prolog.

For example:

```
?- [myfile].
% myfile compiled 0.00 sec, 1563 bytes.
Yes
```

You can also use the "File" menu to select the prolog file to open and load.

Types and Expressions

Prolog is a weakly typed language. We have the following simple data types:

boolean , integer, real, atoms (character sequences)

Since a variable could be of many possible types, there are predicates to test what type a variable is:

var(V)	true if V is a variable
nonvar(NV)	true if NV is not a variable
atom(A)	true if A is an atom
integer(I)	true if I is an integer
real(R)	true if R is a floating point number
number(N)	true if N is an integer or real
atomic(A)	true if A is an atom or a number

Arithmetic expressions are evaluated using the built-in predicate **is** which operates in an infix fashion:

```
?- X is 4*4.
X = 16
```

It is important to note that we cannot use the = symbol here. The equals symbol unifies a variable with some value, but does not evaluate the value:

```
?- X = 4*4.
X = 4*4
```

We have not evaluated 4 times 4. Instead X is bound to 4*4, just like we could bind X to something like macaroni, cheese, or foo. In this case, we happened to choose a value that looks like an arithmetic expression.

We have available your standard arithmetic operators, with some syntax changes:

+ addition	- subtraction	* multiplication	
/ real division	// integer division	mod modulus	** power

Boolean Predicates allow us to compare values to one another. There are several interesting boolean predicates in Prolog that are not available in other languages, but we'll only cover the basics here.

```
A = B                ; Unify A with B
```

Unification is not the same as assignment. It sets A equal to the matching pattern from B. It is the same as:

```
food(cheese).
?- food(X).
X = cheese;
```

In this case, when we query food(X) we unify or match A with cheese. The same thing is happening when we write A = B, e.g.:

```
A = foo ; A = bar.
A = foo;
A = bar;
No
```

Here Prolog is searching for matches that satisfy A. The user says "no" to discovered matches so none is found.

Here are some other booleans:

```
A == B               ; A identical to B
A \= B               ; A not identical to B
A < B                ; Numeric less than
A > B                ; Numeric greater than
A =< B               ; Numeric <=, note that there is no <=
A >= B               ; Numeric >=, note that there is no =<
```

There is another form of <, > using terms, that we won't cover here.

User I/O

To output a string or variable use write:

write(X)	outputs value of X to screen
write('foo')	outputs foo

The value returned by write is always true.

To read into a variable use read:

read(X).

Read waits for the user to input a value, and what the user inputs is bound to X.

Normally we won't use read or write very often, but in some cases it may be useful, especially for debugging.

Comments

User comments are on lines preceded by %. These will be ignored by the interpreter or compiler.

Functions

Prolog does not provide explicit function types, but we can use rules or predicates to give us the same functionality. We can then use as arguments to our function the different variables that are passed in. With the aid of unification we will be able to manipulate these variables in clever ways to serve as both input and outputs depending on the context.

Let's define a predicate that returns the minimum of two other values. Since we don't have functions that can return values, we must return the value in a third parameter. What will really happen is Prolog will bind the third parameter to a value that satisfies the predicate.

```
minimum(M,N,M) :- M =< N.
minimum(M,N,N) :- N =< M.
```

We could invoke this as:

```
?- minimum(4, 5, X).
X = 4;

?-minimum(10, 3, X).
X = 3;
```

The first invocation unifies M=4, N=5, and M with X = 4 and checks if 4<=5. Since it is this expression is true. Since this is true, we get back as an option X = 4. If we said no to this value, prolog would try the second definition. M = 4, N=5, and N is unified with X. We check to see if 5 <= 4 which it is not, so this returns false.

The second invocation works the other way around, where the first definition will fail but the second will succeed. In this way we have defined a function to determine the minimum by expressing what we want declaratively, but perform no procedural instructions. Prolog does the procedural search for us.

It is important to note that Prolog always tries to satisfy the predicates listed in order from top to bottom when they have the same name. You can take advantage of this behavior by placing the predicates you wish to try first up at the top of the file.

Example Programs

Now that we have mastered functions, let's try some sample programs.

Factorial

For the first one, lets try defining factorial. We must do this recursively, starting with a base case and moving on up, where n! equals n*(n-1)! given the base case of 0! equals 1.

```
factorial(0,1).          %% Defines that 0! equals 1
factorial(A,B) :-        %% Compute A!  return value via B
    A > 0,
    C is A-1,            %% Note "is" for arithmetic expression
    factorial(C,D),      %% Compute A-1 factorial, get value in D
    B is A*D.            %% Our return variable gets set to A*D
```

We can now use this to compute the factorial:

```
?- factorial(4, X).
X = 24
```

Financial Advisor

Next let's implement the financial advisor we wrote earlier as an exercise. We came up with these rules:

Adequate savings: At least $5000 in the bank for each dependent.
Adequate income: must be steady and supply at least $15000 per year plus $4000 for each dependent.

Prolog:

```
min_savings(Dependents, Amount) :-
  Amount is 5000 * Dependents.

min_income(Dependents, Amount) :-
  Amount is (4000 * Dependents) + 15000.
```

Strategies:

1. Savings_account(inadequate) → Investment(savings).
2. Savings_account(adequate) ^ Income(adequate) → Investment(stocks).
3. Savings_account(adequate) ^ Income(inadequate) → Investment(combination).

Prolog:

```
investment(savings) :-
  savings(inadequate).

investment(stocks) :-
  savings(adequate), income(adequate).

investment(combo) :-
  savings(adequate), income(inadequate).
```

Logic:

4. $\forall x$ Amount_saved(x) ^ $\exists y$ (dependents(y) ^ greater(x,MinSavings(y))) → Savings_Account(adequate)
5. $\forall x$ Amount_saved(x) ^ $\exists y$ (dependents(y) ^ not(greater(x,MinSavings(y)))) → Savings_Account(inadequate)

5a. $\neg\exists y$ dependents(y) → Savings_Account(adequate)

6. $\forall x$ Earnings(x, steady) ^ $\exists y$ (dependents(y) ^ greater(x,MinIncome(y))) → income(adequate).
7. $\forall x$ Earnings(x, steady) ^ $\exists y$ (dependents(y) ^ Not(greater(x,MinIncome(y)))) → income(inadequate).
8. $\forall x$ Earnings(x,unsteady) → income(inadequate).

8a. $\forall x$ Earnings(x,steady) ^ $\neg\exists y$ dependents(y) → income(adequate)

Prolog:

```
savings(adequate) :-
  amount_saved(Amount),
  numDependents(Dependents),
  min_savings(Dependents, SavingsMin),
  Amount > SavingsMin.
```

```
savings(adequate) :-
  numDependents(Dependents),
  Dependents == 0.

savings(inadequate) :-
  amount_saved(Amount),
  numDependents(Dependents),
  min_savings(Dependents, SavingsMin),
  Amount =< SavingsMin.

income(adequate) :-
  earnings(AmountEarned, steady),
  numDependents(Dependents),
  min_income(Dependents, IncomeMin),
  AmountEarned > IncomeMin.

income(inadequate) :-
  earnings(AmountEarned, steady),
  numDependents(Dependents),
  min_income(Dependents, IncomeMin),
  AmountEarned =< IncomeMin.

income(adequate) :-
  numDependents(Dependents),
  Dependents == 0.

income(inadeqate) :-
  earnings(AmountEarned, unsteady).
```

We can jump start this with something like:

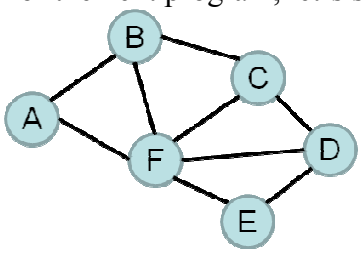
```
amount_saved(22000).
numDependents(3).
earnings(25000, steady).
```

The result is:

```
?- investment(X).
X = combo ;
```

Graph Coloring

For the next program, let's solve the graph coloring problem:



```
color(A,B,C,D,E,F):-
        diff(A,B),
        diff(A,F),
        diff(B,C),
        diff(C,D),
        diff(D,E),
        diff(E,F),
        diff(D,F),
        diff(C,F),
        diff(B,F),
        write(A),
        write(B),
        write(C),
        write(D),
        write(E),
        write(F).

diff(red, blue).
diff(blue, red).
diff(green, red).
diff(red, green).
diff(green, blue).
diff(blue, green).
```

The program runs through the big list of AND's. When something is false, the program backtracks and tries another value. This process is continually repeated until finally we come across some values that match our constraints and make the right hand side true.

Lists

Given what we've covered, we can already write fairly sophisticated programs. However, to build more complex data types, we can make use of prolog's built in lists. To manipulate the lists, we must use recursion since prolog lacks iterative structures.

A list is denoted by square brackets, [].

> [] is the empty list.
> [1] is the list with the element 1 inside it.
> [3,1] is the list with elements 3 and 1 inside it.
> [foo, bar, zot] is the list with foo, bar, and zot in it.
> [[3,1], [2,2], [4,4]] is the list with three sublists

By embedded lists within lists, we can create fairly complex data structures.

The trickier part comes in matching lists. Prolog uses the vertical bar symbol, |, to separate the head element from the tail of the list. For example, if:

[a, b, c] is matched to [X|Y] then X=a, Y=[b, c]
[a, b, c] is matched to [X,Y | Z] then X=a, Y=b, Z = [c]
[a, b, c] is matched to [W, X,Y|Z] then W=a, X=b, Y=c, Z=[]
[a, b, c] does not match [V,W,X,Y | Z] since the number of elements differs.

Here is how we can construct lists and access elements of the list:

```
head([X|Y],X).
tail([X|Y],Y).
add(X,R,[R|X]).
```

Head takes the list as the first argument, and returns the first element of the list as the second argument.

Tail takes the list as the first argument, and returns the tail of the list as the second argument.

Add takes a list as the first argument and an element to add as the second argument and returns the list with the appended element as the third argument. Notice that we are using [R|X] constructively in this case, while previous examples used it to pick apart an existing list.

For example:

```
?- add([],a,X), add(X,b,Y), add(Y,c,Z), tail(Z,TAIL),
head(Z,HEAD).
```

X = [a]
Y = [b, a]
Z = [c, b, a]
TAIL = [b, a]
HEAD = c

This created the list Z equal to [c, b, a] (we put new elements on the front). Then we extracted the tail and the head from this list.

Prolog includes a predicate to test for membership of a list. It is member:

```
?- member(x,[a,b,x,y,z]).
Yes
?- member(x,[a,b,c]).
No
```

Although this is built-in, it is a useful exercise to write our own membership function. We can do so recursively fairly simply:

```
mymember(X, [X|T]).            %% If X is the head, then true.
mymember(X, [Y|T]) :- mymember(X, T). %% Recurse on tail of list
```

The recursive part of this function traces mymember with smaller and smaller lists each time. If X is a member of the list, eventually it will be the head of the list and the first predicate will be true. Otherwise, this function will recurse through the entire list without finding the element and give back false.

Notice that in the recursive definition, we declared a variable Y to refer to anything that is in the head position of the list. However, we didn't use the variable anywhere on the right side of the rule. We just wanted some variable there that is not equal to X. In cases like this, we can instead use something called the anonymous variable. It is denoted by an underscore and serves as a placeholder for a variable whose value we really don't care about. We can rewrite this as:

```
mymember(X, [ _ |T]) :- mymember(X, T).
```

The functionality will be identical to before. This is a good programming practice where applicable, since it tells the programmer and interpreter that certain variables are used solely for pattern-matching purposes, but not for variable binding.

Example Program: Farmer, Wolf, Goat, Cabbage

The farmer, wolf, goat, and cabbage problem is similar to the missionaries and cannibals problem. The farmer wants to get his wolf, goat, and cabbage across a river from the west to the east side, but the boat only has space for himself plus one other occupant. If the farmer leaves the wolf and goat alone, the wolf will eat the goat. If the farmer leaves the goat and cabbage alone, the goat will eat the cabbage. How can the farmer get everyone and everything across the river to the other side without loss? Only the farmer is allowed to take the boat.

We can model this problem by making moves from one state to the next. We must know what the goal state is and what states lead to the loss of the goat or the cabbage. We will model states through the 4-tuple composed of e/w. The first element indicates what side of the river the farmer is on. The second element indicates what side the wolf is on, the third indicates what side the goat is on, and the last indicates what side the cabbage is on.

For example:	w,w,w,w	is the start state, everyone is on the west side
	w,e,w,e	farmer and goat on west, wolf and cabbage on east
	e,e,e,e	goal state, everyone on the east side

Our strategy will be to make moves and then remember each state that we visit by storing the state into a list. Before we visit a new state, we'll check to see if it already exists in the list of visited states. If so, we won't visit that state. This will eliminate the possibility of forming infinite loops in our search.

Here is a predicate that will tell us the opposite of a side:

```
opp(e,w).
opp(w,e).
```

For example, opp(e,X). gives us X=w, which is the opposite of e.

Next we'll make the predicate unsafe that tells us what states are dangerous.

```
% wolf eats goat
unsafe(e,w,w,_).
unsafe(w,e,e,_).
% goat eats cabbage
unsafe(e,_,w,w).
unsafe(w,_,e,e).
```

Note the use of the anonymous variable.

Next, if we ever reach the goal state, print out a message:

```
% Goal state
move(state(e,e,e,e),_) :- write('Goal!'), nl.
```

The possible moves we can make are to take the farmer and wolf, take the farmer and goat, take the farmer and cabbage, or take the farmer by himself. First is take the farmer and the wolf. We can only make this move if the farmer and wolf are on the same side. The first argument is the state while the second argument is a list that will contain all of the states we have visited so far. Initially we'll set this equal to `[]`:

```
% Take farmer and wolf
move(state(F,F,G1,C1),L) :-
      opp(F,F2),
      opp(F,W2),
      G2 = G1,
      C2 = C1,
      not(unsafe(F2,W2,G2,C2)),
      not(member(state(F2,W2,G2,C2),L)),
      move(state(F2,W2,G2,C2),[state(F,F,G1,C1)|L]),
      write('Take wolf to '), write(F2), write(W2),
write(G2), write(C2), nl.
```

First we get the opposite side for the farmer and wolf, since we'll be moving them. We could shorten this a bit by removing one of the opp() since we'll get the same answer each time. But this makes things a bit more clearer, since the new state we will move to will be represented by F2,W2,G2,C2. As long as this state is not unsafe and it is not in the list of previously seen states, then make a recursive call to move with the new state and then add the current state onto the list of visited states. If all of this succeeds, then write out the move to the screen.

The other moves are similar:

```
% Take farmer and goat
move(state(F,W1,F,C1),L) :-
      opp(F,F2),
      opp(F,G2),
      W2 = W1,
      C2 = C1,
      not(unsafe(F2,W2,G2,C2)),
      not(member(state(F2,W2,G2,C2),L)),
      move(state(F2,W2,G2,C2),[state(F,W1,F,C1)|L]),
      write('Take goat to '), write(F2), write(W2), write(G2),
write(C2), nl.
```

```
% Take farmer and cabbage
move(state(F,W1,G1,F),L) :-
      opp(F,F2),
      opp(F,C2),
      G2 = G1,
      W2 = W1,
      not(unsafe(F2,W2,G2,C2)),
      not(member(state(F2,W2,G2,C2),L)),
      move(state(F2,W2,G2,C2),[state(F,W1,G1,F)|L]),
      write('Take cabbage to '),write(F2), write(W2), write(G2),
write(C2), nl.

% Take farmer and nobody else
move(state(F,W1,G1,C1),L) :-
      opp(F,F2),
      C2 = C1,
      G2 = G1,
      W2 = W1,
      not(unsafe(F2,W2,G2,C2)),
      not(member(state(F2,W2,G2,C2),L)),
      move(state(F2,W2,G2,C2),[state(F,W1,G1,C1)|L]),
      write('Take farmer only to '), write(F2), write(W2),
write(G2), write(C2), nl.
```

Here is the program in action:

```
?- move(state(w,w,w,w),[]).
Goal!
Take goat to eeee
Take farmer only to wewe
Take cabbage to eewe
Take goat to weww
Take wolf to eeew
Take farmer only to wwew
Take goat to ewew
```

Much More to Prolog

There is more to Prolog than we covered here. Ultimately, what we have done is define what we want the outcome to be. Prolog has done the searching for us to tell us how to instantiate variables to satisfy our desired outcome.

Prolog programs generally do not run extremely efficiently; an equivalent program in C or C++ or Java will likely run much faster. However, some programs are much easier to code in Prolog but requires a different way to think about solving the problem than in an imperative language.