```
X=penguin
```

If we were to hit return at this point Prolog would print `yes.` and give a prompt. If instead we typed `;' (which in Prolog means logical OR just as `,' means logical AND) and return, Prolog would try to find another solution to our query. In this case, there isn't one so Prolog would respond `no.` If there had been another solution we could keep typing semicolon return until we had gone through all of them. Our last example of a query with this program is:

```
| ?-loves(Z,Y).
```

In this case, when Prolog goes through the table it will first match this goal with the head `loves(jane,X)` setting Z=jane and X=Y. It then tries to satisfy `flies(X)`. This matches with the head of the rule `flies(W)` where X=W. So we try to satisfy the tail of the rule beginning with `flies(W)`. This involves trying to satisfy the goals `bird(W), W \= ostrich, W \= penguin.`. Prolog first tries to satisfy `bird(W)` by setting W=ostrich but then this immediately fails the second goal. Prolog backtracks and tries to satisfy `bird(W)` with W=penguin this. The second goal is satisfied since penguin \= ostrich, but now the third goal fails. So Prolog backtracks and now tries W=seagull. This satisfies the other two goals. So the rule `flies(W)` where W=seagull is satisfied. So `loves(jane,X)` where X=W=seagull is satisfied and thus `loves(Z,Y)` is satisfied where Z=jane, Y=X=W=seagull. So Prolog returns the solution

```
Z=jane
Y=seagull
```

If we typed semicolon return Prolog would try to find another solution by trying to resatisfy `loves(jane, X)` with a different value of X. It can do this with Y=X=eagle. So the next solution Prolog would output is Z=jane, Y=eagle. Prolog can now no longer resatisfy `loves(jane,X)`. So it if we typed semicolon return it tries to find another rule or fact to match `loves(Z,Y)`. This matches the fact `loves(penguin,jane)`. So Prolog would output the solution Z=penguin, Y=jane. Finally, the last solution Prolog can find is by matching `loves(Z,Y)` with `loves(aadvark,jane)` so Prolog would output Z=aadvark, Y=jane.

Lists in Prolog

Prolog like Scheme supports lists. Below are some example lists:

```
[]
[a,b,c]
[dogs,cats,marbles,mix]
[root, [l1, l2], [l3]]
```

The first list above is the empty-list in Prolog. Notice lists in Prolog use brackets rather than the parentheses used in Scheme and also unlike Scheme we separate list elements by commas.

Here is a short program to append two lists:

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

The expression `[X|L1]` denotes the list with car X and cdr L1. We can use this predicate in more than one way. For example:

```
| ?- append([a,b],[c],Z).

Z = [a,b,c]
```

or

```
| ?- append(Y,[e,f,g],[a,b,c,d,e,f,g]).

Y = [a,b,c,d]
```

A brief explanation of how the first example works: As Prolog goes through the list of rules it tries to match `append ([a,b],[c],Z)` with the head of one of the rules involving `append`. The first head which matches comes from the second rule where we set X = a, L1=[b], L2=[c], Z=[X|L3]=[a|L3]. Prolog actually would use a new variable rather than L3 so as to prevent conflicts if the same rule is used again. Let's say it sets Z=[a|VAR1]. So we now try to satisfy the tail of this rule, which after substituting in for the values of L1, L2, VAR1 becomes the goal `append([b],[b],VAR1)`. Again the first head of a rule it matches with is the second rule, where we set X=b, L1=[], L2=[c], VAR1=[b|L3]. Again we rename the L3, so VAR1=[b|VAR2]. So we now try to satisfy `append([],[c],VAR2)`. This matches with the first prolog rule where L=[b], L=VAR2. Hence, we get VAR2=[b] and so VAR1=[b,c] and Z=[a,b,c]. the temporary values VAR1 and VAR2 are destroyed and Z=[a,b,c] is returned.

Built-in predicates:

Prolog has several useful built-in predicates which we now describe.

Comparison and Meta-Logical Predicates

var(X) -
: succeeds if X is an uninstantiated variable

atom(X) -
: succeeds if X is currently stands for an atom.

integer(X) -
: succeeds if X is currently stands for an integer.

real(X) -
: succeeds if X is currently stands for an real.

is_list(X) -
: succeeds if X is currently stands for a list.

X=Y -
: succeeds if X and Y are instantiated to the same value. Uninstantiated variables can be equal to anything.

X\=Y -
: succeeds if X and Y are not instantiated to the same value.

X==Y -
: like X=Y but can only succeed if all varibles instantiated

X\==Y -
: succeeds if X == Y fails.

X < Y or X >Y or X >= Y or X =< Y
: succeed if X and Y are numbers and the corresponding relation holds.

is Predicate and Arithmetic Expressions

The built-in pedicate `X=Y` is not assignment. It just checks if X and Y are instantiated to the same thing. For instance,

```
add(X,Y,Z) :- Z= X+Y.
```

On the query, `| ?- add(1,2,Z)` would return `Z= 1+2` not 3. Also the query, `| ?- add(1,2,3)` would return `no.` since the object 1+2 and 3 are not the same thing. To do assignment, one uses the `is` predicate. For instance,

```
new_add(X,Y,Z) :- Z is X+Y.
```

Now if we do `| ?- new_add(1,2,Z).` we get Z=3 and also `| ?- new_add(1,2,3).` returns `yes.` The right-hand side of an `is` predicate is allowed to be any arithmetic expression made up of +,-,*,/, mod. Notice unlike Scheme expression in Prolog are written in infix (I know this might take some getting used to). Parenthesis and operator precedence are used to

specify order of evaluation. Using `is` in a rule of a predicate restricts the way we can use this predicate. For instance, one might think one could get all the numbers which add to 3 by using the query `| ?- new_add(X,Y,3).` This will give an error however, since for Prolog to intepret the `is` predicate above it needs both X and Y to be instantiated.

I/O Predicates

display(X) -
: Prints X to the current output device (usually screen which is called `userout`).

nl -
: Write a newline to the current output device.

put(X) -
: prints a character X given as an ASCII integer to the current output device.

tab(X) -
: Insert X spaces on current line of current output device.

write(X) -
: Prints X to the current output device. Write pays attention to whether or not an operator should be in infix notation, display ignores this information.

get(X) -
: gets the next ASCII character of value >32 from the current input device. (On Unix systems have to watch out since buffering goes on even when input device is the keyboard.)

get0(X) -
: get next character from current input device.

read(X) -
: reads a term followed by a period from current input device. the variable X is bound only to the term and not the period

skip(X) -
: skip over X many characters from input device.

see(X) -
: set the file X to be the current input device. X=userin is the keyboard.

see(X) -
: set current input device to X.

seeing(X) -
: if X is a variable then X instantiated to name of current input device.

seen -
: close file return current input device to keyboard.

tell(X) -
: set the current output device to be the file X. The screen is the output device userout.

telling(X) -
: if X is a variable then X instantiated to name of current output device.

told -
: close file return current output device to screen.

Debugging Predicates

trace -
: Put Prolog in mode where each step of Prolog search printed out.

notrace -
: Takes Prolog out of trace mode.

spy P -
: Put a spy point on a predicate P. For example `| ?- spy dog/2.` would put a spy point on the 2-ary predicate `dog`.

nospy P -
: Removes a spy point from a predicate P.

debugging -
: turns on debugging mode so that whenever spied-on predicates are called their information is displayed.

nodebug -
: turn off debugging mode.

Miscellaneous Predicates

asserta(X) -
adds the clause X to the start of Prologs database of clauses.
assertz(X) -
adds the clause X to the end of Prologs database of clauses.
retract(X) -
delete first clause which matches X from prolog database of clauses. Only works on dynamically created clauses.
call(X) -
succeeds if goal X succeeds.
not(X) -
succeeds if goal X fails.

Cut

The symbol `!' in Prolog is called *cut*. It is used to limit the search space that a Prolog program searches over when executing a query. It is used in a rule as follows:

```
a :- b,c,d,!,e,f,g.
```

The effect of cut on the above rule is that in trying to satisfy it we allow backtracking among the goals b,c,d and we allow backtracking between the e,f,g but if we ever have to cross the cut to perform backtracking then the goal `a` fails even if there are other clauses beginning with a. As a first example, we can implement the built-in function `not(X)` as:

```
not(X) :- call(X),!,fail.
not(_).
```

The `_' is an anonymous variable it will match anything. The cut-fail combination is quite common in Prolog usage. Another example use of cut would be the following:

```
hd_of_state(bob,elbonia) :- !.
hd_of_state(anne,outland) :- !.
hd_of_state(gilbert,mordor) :- !.
hd_of_state(sally,frisia) :- !.
```

The above program might make sense if we knew in advance that we would never query `hd_of_state` with both slots as variables. We know each country only has one head of state, so once a query on `hd_of_state` had been satisfied we'd know there is no way to resatisfy this goal if we later have to backtrack over it. So using cut we could then fail immediately without searching the other `hd_of_state` clauses. In this same vein consider the program:

```
sum_up(1,1) :-!.
sum_up(N,X) :- N1 is N -1,
               sum_up(N1,X1),
               X is X1+N.
```

This program is supposed to `sum_up` the the numbers from 1 to N. So `sum_up(4,X)` return X=10. Using `sum_up(1,1).` rather than `sum_up(1,1) :-!.`, could lead to problems if we ever attempted to resatisfy the goal `sum_up(4,X)`. If we try to resatisfy `sum_up(4,X)`, if would first try to resatisfy `sum_up(1,1)` using the second `sum_up` clause causing an infinite loop. One last example of the use of cut is the following:

```
repeat.
repeat :-repeat.

some_predicate :- initialize,
                  repeat,
                  do_action,
```