CS 430 Programming Languages
Spring 2016
Module 16 Lab

JAMES MADISON UNIVERSITY

1. Open two terminal windows. Create a directory for your Prolog work and cd to it in both windows (for example, my working directory is creatively called prolog).

2. Open an editor in one window on a file called lab16.pl. (The extension pl is the standard extension for Prolog programs.) Cut and paste the following into the lab16.pl file. Save the file.

```
%
% CS 430 Module 16 Lab facts and rules
%

direct_ancestor(scheme,lisp).
direct_ancestor(commonlisp,lisp).
direct_ancestor(haskell,scheme).
direct_ancestor(java,smalltalk).
direct_ancestor(java,cpp).
direct_ancestor(cpp,c).
direct_ancestor(cpp,smalltalk).
direct_ancestor(smalltalk,simula).
direct_ancestor(simula,algol).
direct_ancestor(csharp,java).
direct_ancestor(c,algol).
direct_ancestor(pascal,algol).
direct_ancestor(ada,pascal).
direct_ancestor(algol,fortran).
```

   The first three lines of this file are comments (now you know how to make comments in Prolog). The fifth and later lines state **facts** about the ancestry of programming languages using a syntax similar to predicate logic. The first fact states that a direct ancestor of Scheme is Lisp. Note that the predicate name and all the language names start with lower case letters. In Prolog, these are called **atoms**. **Variables** start with upper-case letters. Note also that each fact ends with a dot. Every fact or rule in Prolog must end with a dot; if you forget it, nothing will happen until you type it in, or you will get some sort of compilation error.

3. In the other window start up GNU Prolog with the command gprolog. You should see a few lines and then the prompt | ?-. This means that the interpreter is ready for you to query the system.

4. You can leave the interpreter at any time by typing control-D or by typing halt. and enter.

5. So far there are no facts in the system, so we have to tell Prolog to read the file that we just made. There is a built-in predicate called consult to do this for us. Type

consult('lab16.pl'). followed by a return (don't forget the dot). This should load the facts in the named file, indicated by a compilation message and a line with yes on it. You need the single quotes because dots have special meaning (they terminate facts and rules), so we need the quotes to include the dot in the atom naming the file. Fortunately, we can use the default extension and just type consult(lab16). Try it. If we had changed the file, the new predicates would have superseded the old when we reloaded it. Finally, we can abbreviate this by just typing [lab16]. (with the dot). Try that too.

6. Type listing. and return to see what facts are rules are in the database.
7. Now we can make queries. First lets see whether cpp is a direct ancestor of java by typing direct_ancestor(java, cpp). and hitting the enter key. Now see whether java is a direct ancestor of cpp. Now see whether python is a direct ancestor of ruby. Of course gprolog does not know anything about python and ruby because there are no facts about them. Prolog assumes that if a predicate is not a fact in its database, then it is false.
8. We can ask more interesting things. For example, we can see what direct ancestors java has by typing direct_ancestor(java,X). The variable in the second argument slot will cause Prolog to try to find a way to replace the variable with an atom that will satisfy one of the rules. Associating an atom with a variable is called **instantiation**, and matching atoms and predicates with one another is called **unification.** Prolog will succeed in this effort and show you a value for X that will make this statement true. If you type return, it will stop trying, but if you type ; (semicolon) Prolog will attempt another replacement for X. Try it. Eventually, it will run out of alternatives that work.
9. What happens if you ask for a direct ancestor of a language that does not have one recorded in the database? Try it.
10. You can also ask for all the atoms with a particular direct ancestor. Try direct_ancestor(X,algol).
11. Finally, you can ask for all the atoms in the direct_ancestor relation. How?
12. Facts are nice, but they don't get us far. Lets add some rules. Copy and paste the following at the end of lab16.pl and save the file.

```
ancestor(A,C) :- direct_ancestor(A,C).
ancestor(A,C) :- direct_ancestor(A,B), ancestor(B,C).
```

13. Reconsult the file by typing [lab16]. and return. A shortcut: you can use the arrow keys to get to previous queries in gprolog. List the database.
14. The first line in the box above is a **rule**. It is read: A has ancestor C if A has direct ancestor C. In other words, any direct ancestor is also an ancestor. The second line is another rule. It is read: A has ancestor C if A has direct ancestor B and B has ancestor C. In other words, any ancestor of a direct ancestor of A is also an ancestor of A. Note that :- is read as *if*, and the comma is read as *and*. All rules have this form in Prolog, called *Horn clauses*. There can only be one predicate to the left of the :- operator, called the **head**

of the rule. Multiple predicates may appear to the right of the :- operator separated by commas. The variables make this rule general so that it will apply to all atoms.

15. With these definitions, we can now ask other questions. For example, formulate a query to ask for all the ancestors of java. Use the semicolon to go through the entire list. Note that some languages turn up more than once. This is because Prolog is finding more ways to satisfy the ancestor rules, and some of these involve the same languages in the other ways. For example, smalltalk is a direct ancestor of java, so it turns up in the list right off the bat because of the first rule. But cpp is also a direct ancestor of java, and smalltalk is a direct ancestor of cpp, so because of the second rule, smalltalk turns up as an ancestor of java again.

16. Now find all the languages of which lisp is an ancestor.

17. Now find all the pairs of languages in the ancestor relation.

18. Prolog is (supposed to be) a declarative language, which means facts and rules are stated and then Prolog determines whether queries can be satisfied. In such a language, the rules of logic should hold, which means, among other things, that the order in which facts and rules are stated, and the order of conjuncts in a conjunction, should not matter. But, in fact, order is everything in Prolog because of the way the inference engine works. Prolog programmers must constantly be aware of the order in which the inference engine will consider facts and rules in an effort to satisfy a query—Prolog programming is more akin to procedurally programming the inference engine than it is to declaratively populating a database.

19. A query is taken as a **goal** that the inference engine must satisfy. The inference engine will begin trying to match the goal with the first fact or rule with the same predicate as the goal, and go down the list of facts or rules in order. If the goal matches a fact, then the query is satisfied. If the goal matches the head of a rule, then the clauses on the right hand side of the :- operator are taken as sub-goals to be satisfied from left to right, and the matching algorithm begins again at the first fact or rule with the same predicate as the sub-goal. More and more sub-goals may be generated. When all the sub-goals of a goal are satisfied, the goal is satisfied. If a sub-goal cannot be satisfied, then the inference engine backs up to the previously satisfied sub-goal and tries to satisfy it in a different way. This is called **backtracking**. Eventually, either all the sub-goals (and hence the goal) are satisfied, or the inference engine tries but fails to satisfy all sub-goals in every possible way, in which case the main goal fails.

20. To illustrate this, change the definition of ancestor in lab16.pl to the following and save the file.

```
ancestor(A,C):-ancestor(B,C), direct_ancestor(A,B).
ancestor(A,C):-direct_ancestor(A,C).
```

    Reconsult lab16.pl and ask for the ancestors of Java. What happens?

21. By the rules of logic, this definition means exactly the same as before. But lets think about what the inference engine does. When given the query ancestor(java,X) it matches this goal with the head of the first rule in the box, with A=java and C=X (this means that C and X will **share** and both be instantiated to the same values). This generates two sub-goals, to be satisfied in order: ancestor(B,C), and direct_ancestor(java,B). In attempting to satisfy the first sub-goal, the inference engine will match ancestor(B,C) with the head of the first rule, with A=B and C=C, generating the two sub-sub-goals ancestor(B,C), and direct_ancestor(A,B). It is not hard to see why the stack overflows. This also shows why order is extremely important in Prolog programming.

22. Change the rules for ancestor back to the way they were, restart gprolog, reconsult the file, and find the ancestors of java to make sure everything is ok. (In vi, you can just type u in edit mode several times to undo the changes you made to the program file.)

23. Write a rule to define the predicate cousins(X,Y), which is true if X and Y have an ancestor in common. Hint: you can use the same variable in several conjuncts to indicate that the same individual is involved. Add it to lab16.pl, save the file, and consult it. Type the query cousins(java,cpp).

24. Notice that if you type a semicolon after a result from the last query, you will get the same answer over and over. This is because the inference engine is finding more and more common ancestors, so there are more and more ways to satisfy this predicate. You can tell the inference engine to stop backtracking with a special goal called **cut** indicated by the exclamation point (!). A cut always succeeds as a goal, but it stops the inference engine from backtracking at the point where it occurs. The effect of the cut is to fix the sub-goals to its left at their value when the cut is encountered. If you place a cut at the end of your definition of the cousins predicate, it will only give one answer when it is called. Try it (you must separate the cut from other goals with a comma and end the rule with a period, just as before).

25. So far we have been using only two-place predicates, but predicates can have zero or more arguments. Define the predicate common_ancestor(X,Y,Z) that succeeds is Z is a common ancestor of X and Y. Try it out on java and cpp.

26. Notice that if you put in a cut at the end of the definition, then you only got one common ancestor, but that if you left it off, then you could get all of them. Notice also that you could redefine cousins in terms of common_ancestor. There are often many ways to define predicates.

27. Now lets make a one-place predicate called progenitor that is true of a language if it has no ancestors but at least one descendent. It pretty easy to see how to say that a progenitor must have a descendent, but how do we say that it does not have an ancestor? There is no true negation in Prolog, but there is a way to approximate it. There is a special built-in predicate called **not** and written \+ that applies to other predicates. It succeeds if its argument fails. So the predicate \+ancestor(X,Y) succeeds if ancestor(X,Y) fails. Now write progenitor and try it out.

28. You may have noticed that when you consult your program file with the progenitor predicate definition in it you get a warning about singleton variables. A **singleton variable** is one that appears only once in a clause. This is not really a problem, but it does sometimes indicate a mistake, so Prolog flags it with a warning. To make the warning go away, you can replace each singleton variable with the **anonymous variable**, written _. Modify your definition of progenitor by replacing singleton variables with the anonymous variable, reconsult, and test.

29. You can use the anonymous variable in queries as well. In this case, Prolog refrains from indicating the instantiation(s) of the variable. For example, try common_ancestor(java,cpp,_). Before (if you recall), the common ancestors were listed; now they are not.

30. The last topic in this lab is testing. You can build tests into your program by making predicates that succeed only if other predicates succeed. This also provides a good examples of a zero-place predicate. First, lets make a test for cousins. Type the predicate test_cousins :- cousins(java,cpp), \+cousins(java,lisp). that succeeds just in case java and cpp are cousins and java and lisp are not. Consult lab16.pl and type the goal test_cousins. If it succeeds, the test passes.

31. Write similar tests for common_ancestor and progenitor, and then write an overall test goal such as test :- test_cousins, test_common_ancestor, test_pregenitor, !. The cut at the end makes the test either succeed once or fail. Now when you consult the file you merely have to type test. and enter to run all the tests.