C SC 245 — Introduction to Discrete Structures
Fall 2004 (McCann)

http://www.cs.arizona.edu/classes/cs245/fall04/

Quick–'n'–Dirty Prolog Tutorial

September, 2004 (McCann)

Background: Prolog, whose name is from the phrase "PROgramming in LOGic," is a special–purpose language. At the heart of Prolog is a logical inference engine that, based on the data supplied by the programmer, generates answers to questions posed by the user. Unlike procedural languages (e.g., Java), Prolog is a language in which statements are logical expressions.

`gprolog`: This tutorial is based on the GNU project's `gprolog`, an open–source implementation of Prolog. Each implementation of Prolog usually offers features and even syntax that differ from other implementations, so the examples here may not all work in another version of the language.

`gprolog` is available for a variety of systems, including Windows, UNIX, and most UNIX–like systems (including Linux). This tutorial will assume a UNIX–like environment and the basic command–line interface. In particular, it was written for University of Arizona Computer Science students who have no Prolog background, and for that reason U of A hardware will be named.

Prolog Program Components: Technically, you don't create a Prolog program; you create a Prolog database. In the database are two types of clauses: Facts and Rules.

As the name suggests, a fact is a single piece of information. To represent the fact that the sky is blue, you could use the clause `blue(sky).`. Similarly, `mammal(rabbit).` says that rabbits are mammals. Those are both examples of single argument, or unary, predicates. Facts can have multiple arguments; for example, `plays(john,hockey).` indicates that john plays hockey. Worth noting: Prolog constants are in lower case, variables are in upper case, domains are not defined explicitly, and entries always end with a period.

Rules are used to generate new information from facts, other rules, and even themselves. Rules have the form $\boxed{\text{head :- body.}}$, where the head and the body are clauses that typically use variables instead of constants (because rules are used to define general relationships).

For example, consider this rule: `grandparent(X,Z) :- parent(X,Y) , parent(Y,Z).` The rule says that X is the grandparent of Z when X is a parent of Y and Y is a parent of Z. (The comma represents a logical AND, the `parent()` clauses on either side are called subgoals, and X, Y, & Z are variables. Remember, variables are upper case.)

Rules may be recursive. Consider: `ancestor(X,Y) :- parent(Z,Y) , ancestor(X,Z).` This says that X is an ancestor of Y when Y has a parent Z and X is an ancestor of Z. If that's confusing, plug in your name for Y, one of your parent's names for Z (your mom, for example), and one of your mom's parent's names for X.

Notice that we don't explain to Prolog how to work with these rules. That's its problem, not ours. We just have to supply a logically complete database, and Prolog will take care of deciding which rules to use, and when, to answer a given question.

This brings us to the third type of clause: The query. When you want to ask Prolog a question, you supply a query. You can think of a query as being a rule without a body. Prolog will take that head and will attempt to find out if it is true or false. If the query has variables, Prolog will attempt to find all possible values that can be used in place of the variables to make the query true. Better yet, it will tell you what they are.

Here's how it works (very roughly!). Assume the following database:

```
parent(amy,bob).
parent(bob,cathy).
parent(bob,doug).
grandparent(X,Z) :- parent(X,Y) , parent(Y,Z).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(Z,Y) , ancestor(X,Z).
```

The query `parent(amy,bob).` will return true because Prolog can see that it is defined in the database as a fact. `ancestor(bob,doug).` is also true; bob is a parent of doug, and the first ancestor rule says that if you're a parent, then you're an ancestor, too. What about `ancestor(amy,cathy).`? This is true if Prolog can find a constant to replace Z such that `parent(Z,cathy)` is true AND such that `ancestor(amy,Z)` is also true. In this example, it will find that replacing Z with bob will make both subgoals true, and so the answer to the query is true.

Prolog can do more. Consider this query: `parent(bob,X).` With that form of query, we aren't interested in just a true/false answer; we want to know which constants exist in the database that can replace X to make the query true. By searching the facts, Prolog will find that X can be cathy or X can be doug. If you let it, Prolog will tell you both (or you can stop after seeing just the first response).

Prolog can do a LOT more than this example can show. But, this is sufficient for our purposes.

Using `gprolog`: The first step is to create the database. Traditionally, the database file has a .pl file extension, and `gprolog` follows this tradition. Using an editor (a really easy one to use from a terminal window under UNIX is `pico`; you can find a short command list on my web page), create a file with that extension and type in your facts and rules. For example, you might create a file named `family.pl` and type in the following:

```
parent(hank,ben).
parent(hank,denise).
parent(irene,ben).
parent(irene,denise).
parent(alice,carl).
parent(ben,carl).
parent(denise,frank).
parent(denise,gary).
parent(earl,frank).
parent(earl,gary).

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

Don't forget the period at the end of each line! Blank lines are OK. You can even put in comments; in Prolog, they start with a percent sign and stop at the end of the line.

With the file created, you can start `gprolog`. At the shell prompt, type: `gprolog` and press Enter. You'll see something like this:

```
GNU Prolog 1.2.16
By Daniel Diaz
Copyright (C) 1999-2002 Daniel Diaz
| ?-
```

The last line is the Prolog prompt; it's ready for you to type a command. To load your family database into Prolog, use this command at that prompt: `[family].` This will load the database from the `family.pl` file into Prolog. If Prolog finds that everything is in order, it will say something like this:

```
compiling /***/***/family.pl for byte code...
/***/***/family.pl compiled, 14 lines read - 1204 bytes written, 109 ms

yes
| ?-
```

If it's not happy, it will give an error message that probably won't make much sense. Don't panic! It will report the line in the file on which it found the error. Go back to your file, find that line, fix the error, and try again.

With the database loaded, you can type in queries. Here are a few to show you what will happen:

```
| ?- parent(hank,denise).

yes
| ?- parent(denise,hank).

no
| ?- grandparent(irene,frank).

true ? ;

no
| ?- parent(hank,X).

X = ben ? ;

X = denise

yes
| ?- grandparent(hank,X).

X = carl ? ;

X = frank ? ;

X = gary

yes
| ?-
```

When Prolog prints a response and follows it with a question mark, it is asking if you want it to keep searching for more answers. Typically, you do. If so, just press the semicolon, which tells Prolog to keep going.

When you're done and want to leave `gprolog`, just enter Ctrl–D (that is, hold down the Ctrl key on the keyboard, and press D). You'll be back at the shell prompt.

Tracing

Frequently, your database will cause Prolog to produce answers that you know aren't correct. This means that your facts or rules are incorrect, but finding out which ones are wrong can be difficult. To help, Prolog provides a tracing command that allows you to watch Prolog's reasoning as it happens. It's hard to follow until you see it a few times, but it's very helpful, not only for debugging, but also for learning about how Prolog works.

To turn on tracing, the command is `trace.` While tracing is on, you can issue any query, and Prolog will step you through its logical process. Here's what a trace of the `grandparent(hank,X).` query looks like:

```
| ?- trace.
The debugger will first creep -- showing everything (trace)

yes
{trace}
| ?- grandparent(hank,X).
      1    1  Call: grandparent(hank,_16) ?
      2    2  Call: parent(hank,_85) ?
      2    2  Exit: parent(hank,ben) ?
      3    2  Call: parent(ben,_16) ?
      3    2  Exit: parent(ben,carl) ?
      1    1  Exit: grandparent(hank,carl) ?

X = carl ? ;
      1    1  Redo: grandparent(hank,carl) ?
      2    2  Redo: parent(hank,ben) ?
      2    2  Exit: parent(hank,denise) ?
      3    2  Call: parent(denise,_16) ?
      3    2  Exit: parent(denise,frank) ?
      1    1  Exit: grandparent(hank,frank) ?

X = frank ? ;
      1    1  Redo: grandparent(hank,frank) ?
      3    2  Redo: parent(denise,frank) ?
      3    2  Exit: parent(denise,gary) ?
      1    1  Exit: grandparent(hank,gary) ?

X = gary

yes
{trace}
| ?-
```

Note that you press Return at the question marks after the trace output, but semicolons after the normal results output (as shown).

Starting at the top of the trace, note that Prolog uses its own internal labels to represent the variables. Prolog needs to find a constant that has hank as its parent; it finds ben. That takes care of the first subgoal of the grandparent rule. To satisfy the second subgoal it needs to find a constant that has ben as its parent. It finds carl, and reports him to us.

We typed a semicolon, which asked Prolog to look for more grandchildren. It finds that ben doesn't have any other children, but it finds that denise is another child of hank, and denise has a child named frank. Continuing, it finds another child of denise (gary), and that's all.

To turn tracing off, the command is `notrace.`

Connecting Prolog to Predicate Logic

You no doubt remember that in logic, we like to represent predicates with capital letters (e.g., $E(x, y)$ represents *x eats y*). Consider this implication: $(P(x, y) \wedge P(y, z)) \rightarrow G(x, z)$. This says: If $P(x, y)$ is true and $P(x, z)$ is also true, then $G(x, z)$ is true. This is exactly what our grandparent rule says in Prolog! The Prolog syntax is just backwards. Think of the `:-` symbol in Prolog as representing the word "if", and the connection becomes clear.

Internally, Prolog represents rules in a form known as a *Horn clause.* A Horn clause is a disjunction of predicates in which at most one of the predicates is not negated. Sounds odd, but it matches well with Prolog's needs. Consider the grandparent clause again:

```
grandparent(X,Z) :- parent(X,Y) , parent(Y,Z).
```

Rewritten as in our logical notation, including quantification:

$$\forall x\ \forall y\ \forall z\ ((P(x, y) \wedge P(y, z)) \rightarrow G(x, z))$$

We know that $p \rightarrow q$ is logically equivalent to $\neg p \vee q$, and so the above is equivalent to:

$$\forall x\ \forall y\ \forall z\ (\neg(P(x, y) \wedge P(y, z)) \vee G(x, z))$$

And De Morgan's Laws tell us that we can replace the AND with an OR as the negation is pulled through, producing a Horn clause:

$$\forall x\ \forall y\ \forall z\ (\neg P(x, y) \vee \neg P(y, z) \vee G(x, z))$$

Here's why having a Horn clause is important: Prolog has facts in its database about parents. By employing a simplified version of the resolution rule of inference, Prolog can use a fact to remove a predicate from the Horn clause and get closer to an answer. For example, for the query `grandparent(hank,X)`, the resulting Horn clause would be `not parent(hank,A) or not parent(A,B) or grandparent(hank,B)`. The database tells Prolog that `parent(hank,ben)` is a fact, and Prolog can resolve that fact with the Horn clause if A = ben. That assignment is called *unification*, and the result is the Horn clause `not parent(ben,B) or grandparent(hank,B)`. Finding that ben is the parent of carl lets Prolog use resolution and unification once more to conclude that `grandparent(hank,carl)` is true.

Yes, Virginia; there are good reasons to learn about logic if you want to be a computer scientist!

Hopelessly Stuck in Gatesville?

For those of you who are not quite ready to brave the world of UNIX, `gprolog` is available for Windows in the GS228 lab. I believe that there's an icon for it on the desktop, but if not, check the Start menu for "GNU Prolog".

One important difference from the UNIX version: When you load your database into the Windows version, the command syntax is different. For example, if `family.pl` is on the D: drive, your syntax would be `['d:family.pl'].` instead of just `[family].`. Otherwise, for what you need to worry about, I think everything's the same.

Want to Learn More?

There are lots of good texts on Prolog, and plenty of web pages with tutorials. Resolution and unification are often covered in artificial intelligence texts.