Section 1.5: Logic Programming

January 30, 2008

Abstract

In this section we see an application of logic in a language called "Prolog" (for PROgramming in LOGic). Of particular interest is the "inference engine" used by the language to prove theorems (i.e. answer queries). With such a language it is easy to create an "expert system". We are also introduced to recursion.

1 PROLOG

Prolog (PROgramming in LOGic) is a declarative (rather than procedural) language, with its own inference rules which allow the user to pose interesting questions of a database of facts and rules. An SQL select statement **declares** the properties of the data to be extracted from a database, not the **procedure** for extracting the data. Similarly, Prolog declares facts and rules, but doesn't say how to check them. The questions are actually checked by applications of predicate logic.

While the book uses a PROLOG "pseudo-code", we will use a freely obtainable (GNU Lesser General Public License) version of prolog (SWI-Prolog) to get a better feeling for prolog. On our course website is a demo file that I will use today in class. I encourage you to download SWI-Prolog and experiment!

1.1 Prolog database

This is composed of facts and "rules" (which are also statements, or facts!). In terms of predicate logic, one creates predicate wffs on the fly, and establishes which elements of the domain make them true by passing exhaustively over the domain. For example,

animal(bear)

says that the predicate $A(x)$ representing the predicate wff "x is an animal" is true for the constant "bear".

Facts can be binary, as well (and n-ary in general, of course):

eat(bear,fox)

asserts that the predicate wff $E(x, y)$ given by "x eats y" is satisfied by bears and foxes (to the chagrin of the foxes, and the delight of the bears).

In addition, rules can be established to check whether logical connectives between facts are true. For example,

prey(x) **if** *eat(y,x)* **and** *animal(x)*

would tell us if an animal is eaten by something else. In fact, notice that it is expressed as an implication (begun by "if"): translated in more standard predicate wff form, it might read

$$E(y, x) \wedge A(x) \rightarrow Pr(x).$$

which Prolog treats as if it is universally quantified:

$$(\forall x)(\forall y)[E(y, x) \wedge A(x) \rightarrow Pr(x)].$$

(i.e., the compilation of the rule means that it is true for all x and for all y). Here is an example of a convention in which apparently free variables are actually quantified; as long as everyone is on board, not a problem!

So, as promised, the Prolog "rules" are simply more complex predicate wffs, rather than inference rules which it might use to prove theorems.

Once the database is created ("compiled"), we can move on to the important issue of posing interesting questions to the database.

1.2 Prolog queries and "proofs"

Our text lists "**is**" and "**which**" as examples of Prolog queries. The first tests an assertion, whereas the second asks for all instances which make a statement true.

Example: Practice 28, p. 61/65

deer!

Proofs are based on **Horn clauses**, which are simply implications expressed as disjunctions using the implication rule:

$$P \to Q \iff P' \vee Q$$

The right-hand side above is an example of a Horn clause: a wff composed of predicates or the negations of predicates joined by disjunctions, where at most one predicate is unnegated.

So all the facts, which were expressed either as existential instantiations or as implications with universal generalization, are expressible as Horn clauses. A general argument (turned into a Horn clause) looks like

$$P_1 \wedge P_2 \wedge \ldots \wedge P_n \to Q \iff P_1' \vee P_2' \vee \ldots \vee P_n' \vee Q$$

The *prey(x)* rule from above,

$$E(y,x) \wedge A(x) \to Pr(x),$$

is expressed as a Horn clause as

$$E(y,x)' \vee A(x)' \vee Pr(x).$$

Prolog uses Horn clauses to prove arguments by **resolution** (which is essentially disjunctive syllogism): it matches an unnegated predicate with a negated predicate in another rule.

For example, consider our rule *prey(x)*, and the request for those animals which are prey:

which(*x: prey(x)*)

How will Prolog operate? It seeks a rule with *prey* as an unnegated predicate (i.e. it is the consequent of an implication). It finds

$$E(y,x)' \vee A(x)' \vee Pr(x).$$

Prolog then seeks **un**negated $E(y,x)$ or $A(x)$ to collapse this argument using disjunctive syllogism. It finds, for example, that $E(bear, fox)$. Using universal instantiation, Prolog reckons that perhaps

$$E(bear, fox)' \vee A(fox)' \vee Pr(fox).$$

and combines the two to reduce the argument to

$$A(fox)' \vee Pr(fox).$$

When Prolog checks its list of facts, and encounters $A(fox)$, it will resolve with that above to conclude

$$Pr(fox).$$

Alternatively, this can also be thought of as *modus ponens*: $A(fox), A(fox) \to Pr(fox)$ resolves to $Pr(fox)$.

Example: watch the software SWI-Prolog resolve issues associated with Example 39, p. 64/68.

Example: Practice 29, p. 66/69

predator(x) if eat(x, y) and animal(y)
and animal(x)

The reason that Prolog can prove theorems in this way is that the domain is finite: when you create the database, it is only possible to specify a finite collection of facts, which means that Prolog need only test a finite set of instances for truth.

1.3 Recursive rules

Prolog offers us our first example of recursion, "... in which the item being defined is itself part of the definition...." While this is a very powerful and fascinating idea, it can go horribly awry (in the form of infinite loops). We'll see some shortly.

Some rules are recursive in nature: for example, in problem #13, p. 70/74, we're asked to consider a Prolog database for parts of an automobile engine. Parts have been classified into big and small, and a rule exists which determines "part-of". Now we're to write a rule for "component-of". Since a screw may be a part of a filter, which is a part of the fuel system, we need a rule which can dig down into the structure of the parts to discover the true "component-ness" of a part within a part.

Consider the rule defined as two rules, as follows:

component-of(x,y) **if** *part-of(x,y)*

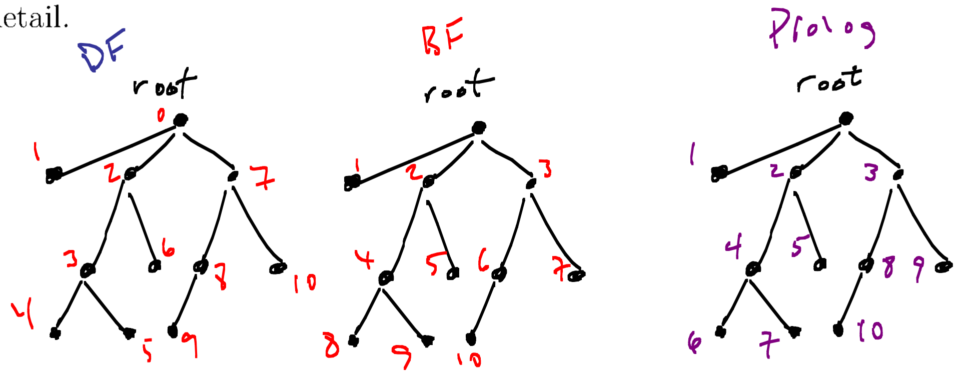
component-of(x,y) **if** *part-of(x,z)* **and** *component-of(z,y)*

The first definition gives us a base case: if x is a part of y, then it is certainly a component of y. On the other hand, the second definition allows us to determine that, since screw is a part of filter, and filter is a part of the fuel system, and fuel system is part of the engine, hence screw is a component of the engine.

Example: watch SWI-Prolog resolve *in-food-chain(bear,X)* (Example 40, p. 67/70).

Example: Practice 30, p. 68/72

Prolog uses a **depth-first** strategy for answering questions with recursion (that is, exploring the length of a path before coming back up to explore an adjacent path), rather than a **breadth-first** strategy. We'll explore both these search strategies later on in the course in greater detail.


DF
root
BF
root
Prolog
root