An Introduction to Logic

Propositions

Prolog is founded upon ideas from logic, and the idea of a logical proof. Logic is a formal system of reasoning about what is true and what is false. The propositional calculus is a simple system in which propositional variables, such as P and Q stand for whole sentences which are considered either true or false. So a sentence like “5 is an integer” is true because of the nature of numbers. A sentence like “a dog is a mammal” is true because of the definition of what it is to be a mammal, and the nature of dogs. However, a sentence like “it is raining” can be either true or false depending on the state of the world when the sentence is uttered.

Logical Connectives

If we let P stand for “it is raining” and Q for “the sky is cloudy”, we can express the fact that the two sentences are true together using a logical “and”. So the compound sentence “it is raining and the sky is cloudy” can be written as $P \wedge Q$ using the connective for and. The idea of “or” can be expressed as well. So the sentence “I am tired or I am hungry” could be written as $P \vee Q$ using the connective for “or”. Logical negation turns true into false and false into true. So the sentence “it is not raining” can be written as $\neg P$ using the unary connective for “not”. These three connectives make it possible to form very complex expressions such as $\neg(P \vee Q) \wedge \neg R$ which express equally complex compound sentences.

Truth tables

The exact meaning of the logical connectives are given in truth tables, which express all the possible combinations of true and false, and the outcome for each connective. The table for not is simple since it only has tow possibilities. The tables for and and or have four possibilities each:

$\neg$	P
false	true
true	false

The column with the not symbol turns the truth of P upside-down.

P	$\wedge$	Q
true	true	true
true	false	false
false	false	true
false	false	false

P	$\vee$	Q
true	true	true
true	true	false
false	true	true
false	false	false

Logical Implication

Truth can also be made conditional, i.e. the truth of one sentence can be dependent on another. Thus the sentence "if it is cloudy then it will rain" can be expressed as an implication: $P \Rightarrow Q$. An implication is only false when the left-hand sentence is true and the right-hand one false. We can express this as a truth table:

P	$\Rightarrow$	Q
true	true	true
true	false	false
false	true	true
false	true	false

Note that an implication is "trivially true" when the left-hand sentence is false.

Logical Proof

There are a set of intuitive rules that derive true results from other true (compound) propositions. For example, if it known that $P \wedge Q$ is true, the clearly *P* is true (and also *Q*). If $P \vee Q$ is true, and *P* is false, then *Q* must be true. If $\neg\neg P$ is true, then *P* is true. The most useful of these rules, and the one that Prolog uses is called modus ponens. If it is known that $P \Rightarrow Q$ is true and also that *P* is true, then *Q* must be true. This rules is very often written as:

$$\frac{\begin{array}{c}P \Rightarrow Q\\ P\end{array}}{Q}$$

Use of this rule can give rise to proofs where the truth of some proposition is dependent on a sequence of applications of the rule. For instance, if we know that $P \Rightarrow Q, Q \Rightarrow R, P$ are all true, then clearly *R* is true after the application of modus ponens twice. Put another way, if we want to prove that *R* is true, we need to know that *P* is true first, and that is only because Q is also true. Prolog swaps round the implication, but essentially follows the same thinking. The same proof would be written in Prolog as:

```
    r :- q.
    q :- p.
    p.
```

We can then prove r as a goal by first setting up q as a subgoal, then setting up p as a further subgoal and finally finding p to be a fact. This can also be done with compound subgoals. E.g. if $P \wedge Q \Rightarrow R, R \wedge S \Rightarrow T, P, Q, S$ are all true, then clearly T is true. In Prolog form this is:

```
t :- r,s.
r :- p,q.
p.
q.
s.
```

Relations and Predicates

Some sentences can be broken down because they express properties of or relationships among objects. For instance, the cat is black, says that the object 'cat' has a property of 'black'. John loves Jane is a relationship between John and Jane of 'loves'. In logic, such relationships are called predicates and are expressed in a sort of 'functional' way: black(cat) and loves(John,Jane). It is then possible to express generalities using this form. A statement such as All elephants are gray is not a statement about one particular elephants, but about the whole set of elephants. Similarly, a statement like Some bears are white is not a statement about a particular bear, but about the existence of white bears. These two forms are captured in logic by quantifiers. In order to use a quantifier, we need a set of objects about which we can make a statement. This needs a variable, and the quantifiers are said to quantify 'over' that variable. The universal quantifier captures the meaning of all: $\forall x.elephant(x)$ says that everything is an elephant. Clearly this is false in the real world. However, we can make it true by making it conditional: $\forall x.hasTrunk(x) \Rightarrow elephant(x)$. This says that every thing which has a trunk is an elephant. It is read as 'for all x, if hasTrunk(x) is true, then elephant(x) is true. The existential quantifier expresses the meaning of some: $\exists x.elephant(x)$. Clearly this is true in the real world, since there is at least one things that is an elephant. To restrict the usage of the quantifier, we often put another property in its scope: $\exists x.gray(x) \wedge elephant(x)$ says that there is at least one gray thing that is also an elephant. The existential forms are often read as "there exists an x such that …".
The addition of relations (predicates), variables and quantifiers turns logic propositional into predicate logic, and it really this that gives rise to Prolog.

Prolog and Predicate Logic

We can express complex proofs, still using modus ponens, in predicate logic. For instance, if $\forall x.P(x) \Rightarrow Q(x), P(a)$ are both true, where a is any object, then clearly $Q(a)$ is also true. The form $\forall x.P(x) \Rightarrow Q(x)$ is often read as 'all Ps are Qs', and since a has property P, then it also has property Q. Prolog expresses universal rules like these using variables:

```
q(X) :- p(X).
```

which are implicitly universally quantified. When Prolog matches a goal to a rule, it is instantiating the variable. So in:

```
q(X) :- p(X).
p(a).
```

if we have a goal q(Y), then this is true if p(Y) is true, and this is true because Y can be instantiated to the object a. Prolog therefore proves logical statements by setting them up as goals, attempting to match the goals to facts or rules, and using the equivalent of modus ponens to set up subgoals. A more complex program like:

```
p(X,Y) :- q(X),r(Y).
q(X) :- s,t(X).
r(a).
s.
t(b).
```

can be used to prove many things. We can prove for instance that q(b) is true by using the rule for q. We can also prove p(b,a) is true by using the rule for p and the rule for q. Two questions occur: exactly what can Prolog represent?, and what are the limitations of this? It turns out that Prolog can only represent part of predicate logic. What this means is that some proofs which are possible in logic cannot be proved using Prolog. However, the limitations of this are quite minor. Prolog can represent proofs using a wide variety of forms, and these are perfectly adequate for many purposes. The limitations are given by the form of a rule. The head of the rule (the left-hand side) can only be a single predicate. This subset of predicate logic is called Horn clause logic because we can only express universally quantified expression of the form
$\forall x,\ldots z.P(x,\ldots z) \wedge Q(x,\ldots z) \wedge \cdots \Rightarrow R(x,\ldots z)$
There can be any number of variables, and predicates with any number of arguments, and any number of anded terms on the left of the implication, but there can only be a single term on the right-hand side. (Existential quantifiers can also be handled, but only using logical functions, which we will not describe here). The other big limitation is that negation is not represented. This also means that any statement involving not is difficult to express. Again, the limitation can be minimized, but it is still there.

Is Prolog really Logic

Well, yes and no. Its form of proof is very close to chaining through universally quantified implications using modus ponens, but this is not all of logic. The best answer is to understand its limitations, and work within them to produce working programs. AS a complete example, consider a ‘family relationship’ program that can prove things about family members and how they are related.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),ancestor(Z,Y).
mother(X,Y) :- parent(X,Y),female(X).
father(X,Y) :- parent(X,Y),male(Y).
sibling(X,Y) :- mother(M,X),mother(M,Y)
sibling(X,Y) :- father(F,X),father(F,Y).
child(X,Y) :- parent(Y,X).
cousin(X,Y) :- parent(P1,X),parent(P2,Y),sibling(P1,P2).
parent(john,mary).
parent(jill,mary).
parent(bill,john).
parent(sarah,jake).
parent(jane,sarah).
parent(bill,sarah).
male(john).
male(bill).
male(jake).
female(mary).
female(jill).
female(sarah).
```

Using the program above, we can show that cousin(mary,jake) and that ancestor(bill,mary) among many other things. The goal stack for cousin(mary,jake) is:

```
cousin(mary,jake)
  | X=mary,Y=jake
parent(P1,mary),parent(P2,jake),sibling(P1,P2)
  | P1=john
parent(P2,jake),sibling(john,P2)
  | P2=sarah
sibling(john,sarah)
  | X=john,Y=sarah
mother(M,john),mother(M,sarah)
  | X=M,Y=john
parent(M,john),female(M),mother(M,sarah)
  | M=bill
female(bill),mother(bill,sarah)
  | fail, so backtrack to parent(M,john)
  | fail, so backtrack to sibling(john,sarah)
father(F,john),father(F,sarah)
  | F = bill
parent(bill,john),male(bill),father(bill,sarah)
  |
male(bill),father(bill,sarah)
  |
father(bill,sarah)
  | X=bill,Y=sarah
parent(bill,sarah),male(bill)
```

```
  | male(bill)
yes
```

The goal stack for ancestor(bill,mary) is:

```
ancestor(bill,mary)
  | X = bill,Y=mary
parent(bill,mary)
  | fail, so backtrack to ancestor(bill,mary)
  | X=bill,Y=mary
ancestor(bill,Z),ancestor(Z,mary)
  | X=bill,Y=Z
parent(bill,Z),ancestor(Z,mary)
  | Z=john
ancestor(john,mary)
  | X=john,Y=mary
parent(john,mary)
  |
yes
```