Chapter 3

Introduction to First-Order Logic

This chapter introduces the basics of logic needed to understand some of the foundations of Prolog and logic programming. We start with Propositional Logic and then move to first-order logic.

A logic normally has three components: a syntax that defines the well-formed sentences of the language of the logic, a model-theoretic semantics that defines what models are appropriate for the logic language and how models make the well-formed sentences true or false, and a theory of deduction that gives rules that apply to sentences that determine some aspects of their truth. We will look first at propositional logic and its components, and then at first-order logic.

3.1 Propositional Logic

Propositional logic takes simple declarative sentences as basic and combines them with sentential operators: and, or and not (and perhaps others). For example, we might have basic sentences like "John is going to the store", "It is raining", and "It is daytime". Propositional logic has basic connective words: and, or, and not. We can combine the basic sentences to make larger sentences using the basic connective words. For example, we could say: "John is going to the store and it is daytime", or "It is not raining or it is daytime".

3.1.1 Syntax

In propositional logic we are interested is the connective words, and not particularly in how meaning is given to the basic declarative sentences. So we will simply represent the basic sentences by letters: p,q,r,s,.... We construct more complex sentences using the connectives to get sentences like: not(p), p or q, p or (q and not(p)), etc.

So the syntax of propositional logic is very simple, and can be given by the following grammar:

```
Sent --> Conjsent
Sent --> Sent or Conjsent
Conjsent --> Primesent
Conjsent --> Conjsent and Primesent
Primesent --> Basicsent
Primesent --> not Primesent
Basicsent --> Prop_sym
Basicsent --> ( Sent )
```

3.1.2 Semantics

The meaning of a sentence in a world is a truth value: true or false. I.e., given a world, each sentence is either true or false. So we will think of a world as assigning to each basic sentence a truth value. I.e., a world determines a truth assignment to the propositional symbols (which represent the basic sentences). So a given set of n propositional symbols can distinguish 2^n different worlds, one for each truth assignment to those propositional symbols.

Given the meanings of the propositional symbols, we want to give meanings to the compound sentences, those made up from basics propositional symbols using the propositional connectives. So let P be the set of proposition symbols, and M:P-¿true,false be a truth assignment (associated with some possible world.) We can extend M to the set of all sentences, as follows:

```
M(and(S1,S2)) = true, if M(S1)=true and M(S2)=true,
              = false, otherwise
M(or(S1,S2)) = true, if M(S1)=true or M(S2)=true, or both
              = false, otherwise
M(not(S)) = true, if M(S)=false
          = false, otherwise.
```

Given a propositional sentence, we can consider its truth value in each possible world (i.e., each truth assignment to the propositional symbols in the sentence.) For example, consider (not p or q).

```
p     | q     || not p or q
=======================
true  | true  || true
true  | false || false
false | true  || true
false | false || true
```

Here we have set out all four of the truth assignments (worlds) for the basic propositional symbols, p and q, one in each row. And then in the third column, we have entered the corresponding truth value of the compound sentence (not p or q) in that world in the third column. This is known as a truth table for (not p or q), and we can make a truth table for any (complex) sentence in the propositional logic. If the sentence has n different propositional variables, its truth table will have 2^n rows.

A sentence whose truth table has a last column of all *true is called a* **tautology. One whose last column is all *false is called a* contradiction. (A sentence that is a contradiction is said to be unsatisfiable since no world can satisfy it, i.e., make it true.) For example, (b or not b) is a tautology. And (p and not p) is a contradiction. Notice that if an arbitrary sentence Q is a tautology, then (not (Q)) is a contradiction.**

Tautologies are of particular interest. Say we know that (not (A) or B) is a tautology, for some particular sentences A and B. Say further that we don't know everything about our current world, but we do know that it makes A true. Then we also know that our current world, whatever it is, also makes B true. (Exercise, why?) We often write a formula of the form (not(A) or B) as (A -¿ B), read A implies B, or if A then B. We can extend the logic with this implication as a new connective.

3.1.3 Deduction

So to be able to reason about what is true in our current world, without know everything about the world, it is useful to know the tautologies. Now we can always determine if a sentence is a tautology by building its truth table. But a truth table is exponentially large. Is there a way to determine whether a sentence is a tautology without having to construct the entire truth table, just by looking at the sentence itself and maybe related sentences?

There are many different ways to tackle this problem. We will look at one particular approach. First we will consider sentences that have identical truth tables; such sentences are called equivalent. There are a number of important equivalences; some useful ones are listed here:

1. **A == not not A**
2. **A or (B and C) == (A or B) and (A or C)**
3. **not(A and B) == not(A) or not(B)**
4. **not(A or B) == not(A) and not(B)**

Now given any sentence, we can use these equivalences, iteratively, to transform it into an equivalent sentence of the following form, (called conjuctive normal form): all "not" operators are immediately over propositional symbols, and no "or" is inside the scope of an "and. For example, not(A and (B or not(C))), can be transformed as follows:

```
not(A and (B or not(C)))
== not(A) or not(B or not(C))
== not(A) or (not(B) and not(not(C)))
== not(A) or (not(B) and not(C))
== (not(A) or not(B)) and (not(A) or not(C))
```

We call a propositional symbol or the negation of a propositional symbol a "literal". So for any sentence we can find an equivalent sentence which is a conjunction of disjunctions of literals. We can think of a sentence in this conjunctive normal form as a set of "clauses", one clause for each conjunct, and each clause consisting of a set of literals. So for the example above, not(A and (B or not(C))) is equivalent to the set of clauses not(A),not(B),not(A),not(C). A set of clauses is true if all the clauses are true (corresponding to the conjunctions in the conjunctive normal form) and a single clause is true if any one (or more) of the literals in it is true (corresponding to the disjunction in the equivalent sentence.) A literal is true in the expected cases: if it is a propositional symbol then it is true just in case the propositional symbol is true; if it is the not of a propositional symbol, then it is true just in case the propositional symbol is false.

Now we will present a method (called resolution) to determine whether a set of clauses is a contradiction, i.e. is unsatisfiable. That means that every row of its truth table is false. Given such a method, we can use it to determine whether an arbitrary sentence is a tautology, as follows. We take the formula, put a not around it, convert it to an equivalent sentence in conjunctive normal form (using the equivalences above) and then apply resolution, which will tell us whether it is a contradiction. If it is, then the original sentence is a tautology.

A mentioned resolution works on clauses. The idea is given a set of clauses, to choose two from the set that can be resolved, form their resolvant, which is a new clause, and add the resolvant to the original set of clauses. And continue doing this until the empty clause is generated, or until no new clauses can be generated. If the empty clause is generated, then the original set of clauses is a contradiction.

So we need to define the "resolvant" of two clauses. Given two clauses, one of which contains a proposition symbol P and the other of which contains the literal not(P), these can be resolved and the resolvant is all the literals in the two clauses excluding the literals P and not(P). For example, the two clauses: p,not(q) and not(p),not(q),r, can be resolved on p in the first clause and not(p) in the second, producing the resolvant: not(q),r. Notice that we treat the clauses as sets and thus don't have multiple copies of a literal in a clause.

Do a larger example:

If we are given a set of clauses that is satisfiable, i.e. for which there exists a truth assignment that makes them all true, then if we form a resolvant from that set and add it back to the set, that new set of clauses is also satisfiable. This follows because consider the truth assignment that makes the entire set of clauses true; it must make the two clauses that we resolved true. And it must make the propositional symbol we resolved on either true or false. If it makes it true, then it makes its negation false. Then for the clause containing the negation to be true, some literal in it other than that negation must be true. That literal is in the resolvant and so the resolvant is true. Alternatively the satisfying truth assignment might make the proposition symbol we resolve on false. If so, then some other literal in its clause must be true (since the clause is true.) And that literal will be in the resolvant making the resolvant true. So adding resolvants preserves satisfiability of a set of clauses. If we eventually can get the empty clause, which is unsatisfiable, then the original set of clauses must be unsatisfiable. (If you're not clear on why the empty clause must be unsatisfiable, you can back up one step, and see that the only way we can generate the empty clause through resolution is by resolving two singleton clauses of the form p and not(p). An it's easy to see that no truth assignment can make both these clauses true.) So this argument shows that our tautology testing procedure is sound, i.e., if it says a sentence is a tautology, then it is indeed a tautology.

But we would also like for our procedure to be complete, i.e., if indeed the original sentence is a tautology, then our procedure will indeed tell us so. It turns out that resolution is indeed complete. The proof is not difficult so we will look at it.

Consider an arbitrary sentence. Let p1, p2, p3, ..., pn be all the proposition symbols that appear in it. Consider the following tree built from these symbols:

[]