First Order Logic and Prolog

Plan
• First Order Logic

Announcements
• Assignment2 mailed out last week. Questions?

First-Order Logic

Propositional logic has the virtues of simplicity and genericity (two things we want in a representation language), but also has some severe limitations. Consider our vacuum world. We introduced propositional symbols $D_1, \ldots, D_n$ to represent whether each of the $n$ rooms was dirty. How would we say the following things?

• All the rooms are clean.
• Any room that the dog has been in is dirty.
• The dog is in some room.
• All the upstairs rooms are clean, except the bathrooms.
• Rooms next to dirty rooms have a dusty smell.
• No two adjacent rooms are both dirty.
• The vacuum should visit every dirty room.
• When vacuuming a room, check whether any nearby rooms are dirty.

The point is that representing general knowledge using propositional logic can be cumbersome and expensive. Introducing a new concept like NextTo can require $n^2$ new propositions, and recall that inference is exponential in the number of propositional symbols. And for infinite domains, it is impossible. The problem with propositional logic is that all we have are propositions.

What we need is a language that allows us to put more structure on facts. This requires that we make some further assumptions about the structure of the world (an ontology), but these won't be very severe. The assumption of first-order logic is that we can describe the world in terms of objects and relations. For example, an object could be a room, and a relation (unary) could be whether it is dirty or not. A binary relation involving rooms might be adjacency, or perhaps relative cleanliness (cleaner-than). Starting from these basic building blocks, we can build complex propositional
expressions without an explosion of primitives.

**Syntax**

First-order logic (FOL) is sometimes called first-order predicate calculus (FOPC) because its atomic sentential unit is a predicate. A predicate describes a relation, and is written as a predicate symbol applied to a series of terms. For example, the atomic sentence Dirty(Room1200) is built from the predicate symbol Dirty applied to a single term, the constant Room1200. Predicate formulas can be binary, for example, NextTo(Room1301, Room1303), or of any arity whatsoever. These formulas express relations, in this case, that objects Room1301 and Room1303 are in the NextTo relation.

In addition to constant terms, we can also form terms using functions. For example, we might have an occupants function that can be used to describe the set of occupants of a room (or other objects) without actually giving these occupants their own name. Then, for example, we could write atomic sentences like Sleepy(occupants(Room1200)). Note that constants can be viewed simply as functions with no arguments - we could have also written Sleepy(492students), for example.

As in propositional logic, atomic formulas can be combined into more complex formulas using the standard connectives.

So far, all we have is a somewhat neater way to write down propositions. What gives FOL its real power is its ability to quantify over variables. Consider our example from before, “All the rooms are clean”. With what we have so far, the best we can do is write a long conjunction of predicate expressions with each of the room constants. But FOL offers a much more direct way:

$$\forall x. \neg Dirty(x)$$

if all the objects are rooms, or more generally,

$$\forall x. Room(x) \Rightarrow \neg Dirty(x).$$

In these sentences, \(x\) is a variable term, and \(\forall\) is the universal quantifier. The sentence is equivalent to a big conjunction, where \(x\) is replaced in the formulas by every object in the universe (hence the term universal).

FOL has one other quantifier, the existential:
∃x. DogIn(x), or ∃x. Room(x) ∧ DogIn(x).

A sentence is well-formed as long as all its variables are quantified (sometimes universal by default). We will get a better feel for quantifiers when we practice using them a bit later.

FOL offers one more important built-in feature, namely a special equality predicate symbol. An expression of the form term = term is an atomic sentence, just as if "=" were a binary predicate (but written in infix notation).

**Semantics**

Now that we know what FOL lets us write down, what does it all mean? The semantics of a language is the connection between sentences and facts in the world. For FOL, we start by assuming that we can interpret the world as consisting of objects and relations. Terms in FOL denote objects, and predicate symbols the names of relations. Thus a sentence of the form

\[ \text{Predicate}(\text{term}_1, \ldots, \text{term}_n) \]

is true iff the tuple \( \langle \text{term}_1, \ldots, \text{term}_n \rangle \) is in the relation named by Predicate.

For example, an interpretation of the Dirty predicate dictates which rooms are in the (unary) Dirty relation, and an interpretation of NextTo specifies the pairs of rooms that are next to each other.

The equality predicate is treated specially. Whereas the meaning of other predicate symbols depends on the interpretation, the interpretation of "=" is fixed. Specifically, the sentence \( \text{term}_1 = \text{term}_2 \) is true iff \( \text{term}_1 \) and \( \text{term}_2 \) denote the same object.

Now we have specified the meaning of atomic sentences. The meaning of composite sentences formed by connectives is defined by truth tables, just as in propositional logic.

A sentence with a universally quantified variable is true iff it is true no matter which object the variable stands for. For example,

\[ \forall x. \text{Room}(x) \Rightarrow \neg \text{Dirty}(x) \] is true if \( \text{Room}(x) \Rightarrow \neg \text{Dirty}(x) \) is true for all objects \( x \). Notice that this sentence happens to be trivially true for any \( x \) that is not a room, by the nature of implication. This is a very standard
pattern for the use of universal quantification.

An existentially quantified sentence is true as long as it is true for some object that the variable could stand for. For instance, \( \exists x. Room(x) \land DogIn(x) \) requires that there is some \( x \) such that \( x \) is a room with a dog in it. Consider the meaning of the alternatives,

\[
\exists x. Room(x) \Rightarrow DogIn(x), \text{ or } \forall x. Room(x) \land \neg Dirty(x)
\]

Writing FOL Sentences

To get a feel for the use of FOL, let’s write down some general knowledge. Why? Because TELLing an agent stuff is the way we program it!

Before starting, a word about symbol names. As you know, in a programming language such as LISP, the actual symbols used as variables don’t matter. So the program

\[
(defun foo (x) (+ x 100))
\]

and the program

\[
(defun foo (dont-call-me-x) (+ dont-call-me-x 100))
\]

are identical. Similarly, in FOL, the sentence

\[
\forall x. MyFriend(x) \Rightarrow SendBirthdayCard(x)
\]

means the same thing as

\[
\forall x. P0001(x) \Rightarrow P0002(x)
\]

or even

\[
\forall x. MyEnemy(x) \Rightarrow SendBirthdayCard(x)
\]

It is only our intended meaning of \( MyFriend \) that has the meaning we are thinking of. What makes \( MyFriend \) different from \( MyEnemy \) in the knowledge base is its connection to other things. We should keep this in mind when translating from English to FOL.

Go around the room, translating sentences as time permits.
Politicians are untrustworthy.

For all x: Politician(x) => ~Trustworthy(x)

Fort Hare students are sometimes sports fans.

Exists y: Student(y, FortHare) & Fan(y, Sports)

If Bush wins the election .... what?

Wins(Bush, Election) => For all y: ~(y=Bush) => Loses(y)

People who are tall, blond, and from California either drive BMWs or they can’t afford a car.

For all x: (Tall(x) & Blond(x) & From(x, CA)) => (Drives(x, BMW) V ~Exists y: Drives(x, y))

**Inference using Modus Ponens**

When we talked about propositional logic, we discussed several inference rules, for example Modus Ponens:

\[
\frac{\alpha \rightarrow \beta, \alpha}{\beta}
\]

Modus Ponens is a sound inference rule in FOL, as are all the propositional inference rules we enumerated. For first-order inference we will need special ways to treat variables and quantifiers, but before we get to that let us explore inference using Modus Ponens in the simpler, variable-free case.

We know the following: That RoomA is next to RoomB, and that if RoomA is next to RoomB, then the rooms are not apart. What can we conclude?

Nextto(A, B)

Nextto(A, B) => ~Apart(A, B)

--------

~Apart(A, B)
Of course, an advantage we cited for using FOL was that we could introduce the use of variables. So, for example, we could say that any rooms that are next to each other are not apart:

\[ \text{Forall } x, y: \text{Nextto}(x, y) \implies \neg \text{Apart}(x, y) \]

Assuming \text{Nextto}(A, B) what can we conclude?

Nextto(x, y)?? No!

Nextto(A, B)!

We can even do Generalized Modus Ponens:

\[ \text{NT}(A, B) \]

\[ \text{NT}(B, C) \]

\[ \text{Forall } x, y, z: \text{NT}(x, y) \land \text{NT}(y, z) \implies \text{Apart}(x, z) \]

yields \text{Apart}(A, C)

What this says is that once we know one of the antecedent conjuncts, then the remaining conjuncts will be sufficient to imply the conclusion. Note the original modus ponens is a special case with \( m = 1 \).

Returning to the example, what if it had instead known:

\[ \text{NT}(A, B) \land \text{NT}(C, B) \]

Can we conclude \text{Apart}(A, B)?

Couldn’t do it, because \( y \) would have to change values! Leads to what we can and can’t do to match terms to each other - otherwise known as unification.

**Unification**

Unify \( \text{NT}(A, B) \) with \( \text{NT}(x, y) \)

Unify \( \text{NT}(A, x) \) with \( \text{NT}(y, B) \)

Unify \( \text{NT}(A, x) \) with \( \text{NT}(x, B) \)
Point out the need for standardizing apart

Notion of Most General Unifier: What if we had:

\[ NT(x,y) \land NT(y,z) \Rightarrow \text{Apart}(y,w) \]?

Well, since the left side is true, the right side is true. Since the quantification is universal, we could have w bound to anything. Should we bind it to something like “anger”? Why not?

**Prolog**

One convenient feature of the KB we are using is that all sentences are of the form \((a_1 \land a_2 \land \ldots \land a_m) \Rightarrow b\), where b and each of the \(a_i\) are positive literals. Sentences of this form are called Horn sentences, and a KB consisting exclusively of Horn sentences is called (you guessed it) a Horn KB.

For Horn KBs, it turns out that modus ponens is all we need. That is, modus ponens is both sound and complete for Horn KBs. To find all consequences of a set of Horn sentences, we can simply run modus ponens until it is no longer applicable. This sort of inference process is called forward chaining.

Alternately, we can attempt to prove a particular sentence by backward chaining. To do this, we find which implications in the KB conclude the sentence, and recursively try to find proofs for their antecedent conjuncts.

[Generate the AND/OR tree for the Nextto problem.]

Declarative notions. We declare facts, and declare rules. We don’t tell the program “how” to get an answer, we just give it a bunch of facts and rules.

All of this can be done using a logic programming language called Prolog. Note, though, that Prolog introduces other features that push it beyond Horn KBs. In particular, it permits negated conditions, and has to deal with what it means to prove that something is not true. We’ll see this in a while.

Go into prolog and use Bratko’s predecessor examples.

```prolog
?- [user].
| : parent(pam, bob).
```
\begin{verbatim}
| : parent(tom, bob).
| : parent(tom, liz).
| : parent(bob, ann).
| : parent(bob, pat).
| : parent(pat, jim).
| :
% user compiled 0.01 sec, 1,320 bytes

Yes
?- parent(pam, bob).

Yes
?- parent(pam, pat).

No
?- parent(X, pat).

X = bob ;

No
?- parent(X, bob).

X = pam ;

X = tom ;

No
?- parent(tom, X).

X = bob ;

X = liz ;

No
?- parent(X, Y).

X = pam
Y = bob ;

X = tom
Y = bob ;

X = tom
Y = liz ;

X = bob
Y = ann ;
\end{verbatim}
\[ X = \text{bob} \\
Y = \text{pat}; \\
X = \text{pat} \\
Y = \text{jim}; \]

\[ \text{No} \]

\[- \text{parent}(X, X). \]

\[ \text{No} \]

Simply by using facts and unification, we can do some potentially interesting things. This especially holds true when we get to interesting data structures.

For example, we can define a point relation as holding over 2 coordinates: point(1, 2) could be true if these coordinates define a point. Then a line segment connects two points: seg(point(1,2),point(3,4)). Finally, we can have unary predicates about line segments. For example, a segment is vertical if the x values are the same:

Vertical(seg(point(X,Y),point(X,Y1))). \text{<Note missing paren in book!>}

Horizontal(seg(point(X,Y),point(X1,Y))).

?- [user].
| vertical(seg(point(X,Y),point(X,Y1))).
Warning: Singleton variables: \{Y, Y1\}
| horizontal(seg(point(X,Y),point(X1,Y))).
Warning: Singleton variables: \{X, X1\}
| % user compiled 0.01 sec, 124 bytes

Yes
?- vertical(seg(point(1,1),point(1,2))).

Yes
?- vertical(seg(point(1,1),point(2,Y))).

No
?- horizontal(seg(point(1,1),point(2,Y))).

Y = 1;

No
?- vertical(seg(point(2,3), P)).
P = point(2, _G364) ;

No
?- vertical(S),horizontal(S).

S = seg(point(_G353, _G354), point(_G353, _G354)) ;

No

From this last case, notice that we can give a “goal” (query to prove) that is a conjunction. More generally, just like in Norvig’s Prolog, we can give a “fact” that is a rule:

?- [user].
|    parent(pam,bob).
|    offspring(X,Y) :- parent(Y,X).
|    parent(bob,ann).
Warning: Clauses of parent/2 are not together in the source-file
| % user compiled 0.00 sec, 152 bytes

Yes
?- offspring(bob,X).

X = pam ;

No
?- offspring(X,Y).

X = bob
Y = pam ;
X = ann
Y = bob ;

No

Finally, rules can have a combination of conditions:

?- [user].
|    parent(pam, bob).
|    parent(tom, bob).
|    parent(bob, ann).
|    offspring(X,Y) :- parent(Y,X).
|    female(pam).
|    female(ann).
|    male(bob).
|    male(tom).
| daughter(X) :- female(X), offspring(X,Y). |
WARNING: Singleton variables: [Y] |
| % user compiled 0.00 sec, 740 bytes |

Yes
?- daughter(X).

And Prolog has built-in a number of other capabilities. We’ll talk about one or two here.

One is that you can also have negation:

male(X) :- not(female(x))
?- [user].
| : male(X) :- not(female(X)).
| : male(bob).
| : female(ann).
| : % user compiled 0.00 sec, -284 bytes

Yes
?- male(X).
X = bob ;
No
?- female(Y).
Y = ann ;
No
?- male(joe).
Yes

The other is that you can also have functions (data structures):
?- [user].
| : married(ed).
| : married(john,sue).
| : married(X,spouseof(X)) :- married(X).
| :
% user compiled 0.00 sec, -140 bytes

Yes
?- married(X).

X = ed ;

No
?- married(X,Y).

X = john
Y = sue ;

X = ed
Y = spouseof(ed) ;

No