Module 6

Knowledge Representation and Logic – (First Order Logic)

6.1 Instructional Objective

- Students should understand the advantages of first order logic as a knowledge representation language
- Students should be able to convert natural language statement to FOL statements
- The student should be familiar with the following concepts of first order logic
  - syntax
  - interpretation
  - semantics
  - semantics of quantifiers
  - entailment
  - unification
  - Skolemization
- Students should be familiar with different inference rules in FOL
- Students should understand soundness and completeness properties of inference mechanisms and the notions of decidability
- Students should learn in details first order resolution techniques
- The use of search in first order resolution should be discussed, including some search heuristics

At the end of this lesson the student should be able to do the following:

- Represent a natural language description as statements in first order logic
- Applying inference rules
- Implement automated theorem provers using resolution mechanism

Lesson 13

First Order Logic - I

6.2 First Order Logic

6.2.1 Syntax

Let us first introduce the symbols, or alphabet, being used. Beware that there are all sorts of slightly different ways to define FOL.

6.2.1.1 Alphabet

- Logical Symbols: These are symbols that have a standard meaning, like: AND, OR, NOT, ALL, EXISTS, IMPLIES, IFF, FALSE, =.
- Non-Logical Symbols: divided in:
    - Constants:
        - Predicates: 1-ary, 2-ary, .., n-ary. These are usually just identifiers.
        - Functions: 0-ary, 1-ary, 2-ary, .., n-ary. These are usually just identifiers. 0-ary functions are also called **individual constants**.

      Where predicates return true or false, functions can return any value.

    - Variables: Usually an identifier.

  One needs to be able to distinguish the identifiers used for predicates, functions, and variables by using some appropriate convention, for example, capitals for function and predicate symbols and lower cases for variables.

6.2.1.2 Terms

A Term is either an individual constant (a 0-ary function), or a variable, or an n-ary function applied to n terms: F(t1 t2 ..tn)
[We will use both the notation F(t1 t2 ..tn) and the notation (F t1 t2 .. tn)]

6.2.1.3 Atomic Formulae

An Atomic Formula is either FALSE or an n-ary predicate applied to n terms: P(t1 t2 .. tn). In the case that "=" is a logical symbol in the language, (t1 = t2), where t1 and t2 are terms, is an atomic formula.

6.2.1.4 Literals

A Literal is either an atomic formula (a **Positive Literal**), or the negation of an atomic formula (a **Negative Literal**). A **Ground Literal** is a variable-free literal.

6.2.1.5 Clauses

A Clause is a disjunction of literals. A **Ground Clause** is a variable-free clause. A **Horn Clause** is a clause with at most one positive literal. A **Definite Clause** is a Horn Clause with exactly one positive Literal.

Notice that implications are equivalent to Horn or Definite clauses:

(A IMPLIES B) is equivalent to ((NOT A) OR B)
(A AND B IMPLIES FALSE) is equivalent to ((NOT A) OR (NOT B)).

6.2.1.6 Formulae

A Formula is either:

- an atomic formula, or
- a **Negation**, i.e. the NOT of a formula, or
- a **Conjunctive Formula**, i.e. the AND of formulae, or
- a **Disjunctive Formula**, i.e. the OR of formulae, or
- an **Implication**, that is a formula of the form (formula1 IMPLIES formula2), or
- an **Equivalence**, that is a formula of the form (formula1 IFF formula2), or
- a **Universally Quantified Formula**, that is a formula of the form (ALL variable formula). We say that occurrences of variable are **bound** in formula [we should be more precise]. Or
- a **Existentially Quantified Formula**, that is a formula of the form (EXISTS variable formula). We say that occurrences of variable are **bound** in formula [we should be more precise].

An occurrence of a variable in a formula that is not bound, is said to be **free**. A formula where all occurrences of variables are bound is called a **closed formula**, one where all variables are free is called an **open formula**.

A formula that is the disjunction of clauses is said to be in **Clausal Form**. We shall see that there is a sense in which every formula is equivalent to a clausal form.

Often it is convenient to refer to terms and formulae with a single name. **Form** or **Expression** is used to this end.

6.2.2 Substitutions

- Given a term s, the result [**substitution instance**] of **substituting a term t in s for a variable x**, s[t/x], is:
    - t, if s is the variable x
    - y, if s is the variable y different from x
    - F(s1[t/x] s2[t/x] .. sn[t/x]), if s is F(s1 s2 .. sn).

- Given a formula A, the result (**substitution instance**) of **substituting a term t in A for a variable x**, A[t/x], is:
    - FALSE, if A is FALSE,
    - P(t1[t/x] t2[t/x] .. tn[t/x]), if A is P(t1 t2 .. tn),
    - (B[t/x] AND C[t/x]) if A is (B AND C), and similarly for the other connectives,
    - (ALL x B) if A is (ALL x B), (similarly for EXISTS),
    - (ALL y B[t/x]), if A is (ALL y B) and y is different from x (similarly for EXISTS).

The substitution [t/x] can be seen as a map from terms to terms and from formulae to formulae. We can define similarly [t1/x1 t2/x2 .. tn/xn], where t1 t2 .. tn are terms and x1 x2 .. xn are variables, as a map, the **[simultaneous] substitution of x1 by t1, x2 by t2, .., of xn by tn**. [If all the terms t1 .. tn are variables, the substitution is called an **alphabetic variant**, and if they are ground terms, it is called a **ground substitution**.] Note that a simultaneous substitution is not the same as a sequential substitution.

6.2.3 Unification

- Given two substitutions S = [t1/x1 .. tn/xn] and V = [u1/y1 .. um/ym], the **composition** of S and V, S . V, is the substitution obtained by:
    - Applying V to t1 .. tn [the operation on substitutions with just this property is called **concatenation**], and
    - adding any pair uj/yj such that yj is not in {x1 .. xn}.

  For example: [G(x y)/z].[A/x B/y C/w D/z] is [G(A B)/z A/x B/y C/w].

  Composition is an operation that is associative and non commutative

- A set of forms f1 .. fn is **unifiable** iff there is a substitution S such that f1.S = f2.S = .. = fn.S. We then say that S is a **unifier** of the set.
  For example {P(x F(y) B) P(x F(B) B)} is unified by [A/x B/y] and also unified by [B/y].

- A **Most General Unifier** (MGU) of a set of forms f1 .. fn is a substitution S that unifies this set and such that for any other substitution T that unifies the set there is a substitution V such that S.V = T. The result of applying the MGU to the forms is called a **Most General Instance** (MGI). Here are some examples:
-

FORMULAE	MGU	MGI
(P x), (P A)	[A/x]	(P A)
(P (F x) y (G y)), (P (F x) z (G x))	[x/y x/z]	(P (F x) x (G x))
(F x (G y)),	[(G u)/x y/z]	(F (G u) (G y))

```
(F (G u) (G z))
--------------------------------------------------------------
(F x (G y)),           Not Unifiable
(F (G u) (H z))
--------------------------------------------------------------
(F x (G x) x),         Not Unifiable
(F (G u) (G (G z)) z)
--------------------------------------------------------------
```

This last example is interesting: we first find that (G u) should replace x, then that (G z) should replace x; finally that x and z are equivalent. So we need x->(G z) and x->z to be both true. This would be possible only if z and (G z) were equivalent. That cannot happen for a finite term. To recognize cases such as this that do not allow unification [we cannot replace z by (G z) since z occurs in (G z)], we need what is called an Occur Test . Most Prolog implementation use Unification extensively but do not do the occur test for efficiency reasons.

The determination of Most General Unifier is done by the **Unification Algorithm**. Here is the pseudo code for it:

```
FUNCTION Unify WITH PARAMETERS form1, form2, and assign RETURNS MGU,
where form1 and form2 are the forms that we want to unify, and assign
is initially nil.

1. Use the Find-Difference function described below to determine the
   first elements where form1 and form2 differ and one of the elements
   is a variable. Call difference-set the value returned by Find-
   Difference. This value will be either the atom Fail, if the two
   forms cannot be unified; or null, if the two forms are identical; or
   a pair of the form (Variable Expression).

2. If Find-Difference returned the atom Fail, Unify also returns Fail
   and we cannot unify the two forms.

3. If Find-Difference returned nil, then Unify will return assign as
   MGU.

4. Otherwise, we replace each occurrence of Variable by Expression in
   form1 and form2; we compose the given assignment assign with the
   assignment that maps Variable into Expression, and we repeat the
   process for the new form1, form2, and assign.


FUNCTION Find-Difference WITH PARAMETERS form1 and form2 RETURNS pair,
where form1 and form2 are e-expressions.

1. If form1 and form2 are the same variable, return nil.

2. Otherwise, if either form1 or form2 is a variable, and it does not
   appear anywhere in the other form, then return the pair (Variable
   Other-Form), otherwise return Fail.

3. Otherwise, if either form1 or form2 is an atom then if they are the
   same atom then return nil otherwise return Fail.
```

```
4. Otherwise both form1 and form2 are lists.
   Apply the Find-Difference function to corresponding elements of the
   two lists until either a call returns a non-null value or the two
   lists are simultaneously exhausted, or some elements are left over
   in one list.

   In the first case, that non-null value is returned; in the second,
```
nil is returned; in the third, Fail is returned

6.2.4 Semantics

Before we can continue in the "syntactic" domain with concepts like Inference Rules and Proofs, we need to clarify the Semantics, or meaning, of First Order Logic.

An **L-Structure** or **Conceptualization** for a language L is a structure M= (U,I), where:

- U is a non-empty set, called the **Domain**, or **Carrier**, or **Universe of Discourse** of M, and
- I is an **Interpretation** that associates to each n-ary function symbol F of L a map

  I(F): UxU..xU -> U

  and to each n-ary predicate symbol P of L a subset of UxU..xU.

The set of functions (predicates) so introduced form the **Functional Basis (Relational Basis)** of the conceptualization.

Given a language L and a conceptualization (U,I), an **Assignment** is a map from the variables of L to U. An **X-Variant** of an assignment s is an assignment that is identical to s everywhere except at x where it differs.

Given a conceptualization M=(U,I) and an assignment s it is easy to extend s to map each term t of L to an individual s(t) in U by using induction on the structure of the term.

Then

- **M satisfies a formula A under s** iff
  - A is atomic, say P(t1 .. tn), and (s(t1) ..s(tn)) is in I(P).
  - A is (NOT B) and M does not satisfy B under s.
  - A is (B OR C) and M satisfies B under s, or M satisfies C under s. [Similarly for all other connectives.]
  - A is (ALL x B) and M satisfies B under all x-variants of s.
  - A is (EXISTS x B) and M satisfies B under some x-variants of s.
- **Formula A is satisfiable in M** iff there is an assignment s such that M satisfies A under s.
- **Formula A is satisfiable** iff there is an L-structure M such that A is satisfiable in M.

- **Formula A is valid or logically true in M** iff M satisfies A under any s. We then say that M is a **model** of A.
- **Formula A is Valid or Logically True** iff for any L-structure M and any assignment s, M satisfies A under s.

Some of these definitions can be made relative to a set of formulae GAMMA:

- **Formula A is a Logical Consequence of GAMMA in M** iff M satisfies A under any s that also satisfies all the formulae in GAMMA.
- **Formula A is a Logical Consequence of GAMMA** iff for any L-structure M, A is a logical consequence of GAMMA in M. At times instead of "A is a logical consequence of GAMMA" we say "GAMMA entails A".

We say that formulae A and B are (logically) **equivalent** iff A is a logical consequence of {B} and B is a logical consequence of {A}.

EXAMPLE 1: A Block World

Here we look at a problem and see how to represent it in a language. We consider a simple world of blocks as described by the following figures:

```
                                 +--+
                                 |a |
                                 +--+
                                 |e |
     +--+                        +--+
     |a |                        |c |
     +--+    +--+                +--+
     |b |    |d |    ======>     |d |
     +--+    +--+                +--+
     |c |    |e |                |b |
    ---------------         --------------------
```

We see two possible states of the world. On the left is the current state, on the right a desired new state. A robot is available to do the transformation. To describe these worlds we can use a structure with domain U = {a b c d e}, and with predicates {ON, ABOVE, CLEAR, TABLE} with the following meaning:

- ON: (ON x y) iff x is immediately above y.
  The interpretation of ON in the left world is {(a b) (b c) (d e)}, and in the right world is {(a e) (e c) (c d) (d b)}.
- ABOVE: (ABOVE x y) iff x is above y.
  The interpretation of ABOVE [in the left world] is {(a b) (b c) (a c) (d e)} and in the right world is {(a e) (a c) (a d) (a b) (e c) (e d) (e b) (c d) (c b) (d b)}
- CLEAR: (CLEAR x) iff x does not have anything above it.
  The interpretation of CLEAR [in the left world] is {a d} and in the right world is {a}

- TABLE: (TABLE x) iff x rests directly on the table.
  The interpretation of TABLE [in the left world] is {c e} and in the right world id {b}.

Examples of formulae true in the block world [both in the left and in the right state] are [these formulae are known as **Non-Logical Axioms**]:

- (ON x y) IMPLIES (ABOVE x y)
- ((ON x y) AND (ABOVE y z)) IMPLIES (ABOVE x z)
- (ABOVE x y) IMPLIES (NOT (ABOVE y x))
- (CLEAR x) IFF (NOT (EXISTS y (ON y x)))
- (TABLE x) IFF (NOT (EXISTS y (ON x y)))

Note that there are things that we cannot say about the block world with the current functional and predicate basis unless we use equality. Namely, we cannot say as we would like that a block can be ON at most one other block. For that we need to say that if x is ON y and x is ON z then y is z. That is, we need to use a logic with equality.

Not all formulae that are true on the left world are true on the right world and viceversa. For example, a formula true in the left world but not in the right world is (ON a b). Assertions about the left and right world can be in contradiction. For example (ABOVE b c) is true on left, (ABOVE c b) is true on right and together they contradict the non-logical axioms. This means that the theory that we have developed for talking about the block world can talk of only one world at a time. To talk about two worlds simultaneously we would need something like the Situation Calculus that we will study later.