First-Order Logic

Chapter 8

(Please turn your mobile devices to silent. Thanks!)

Last Time

- Logical agents apply inference to a knowledge base to derive new information and make decisions
- Entailment vs. Inference
- Two ways to prove a query
  1. Application of inference rules
  2. Model checking
- Soundness and Completeness as conditions for inference
- Resolution is complete for propositional logic in CNF
- Forward, backward chaining are linear-time, complete for Horn clauses

Outline

- Why FOL?
- Syntax and semantics of FOL
- Using FOL
- Wumpus world in FOL
- Knowledge engineering in FOL

Pros and cons of propositional logic

- ☺ Propositional logic is declarative
- ☺ Propositional logic allows partial/disjunctive/negated information
  - (unlike most data structures and databases)
- ☺ Propositional logic is compositional:
  - meaning of $B_{1,1} \wedge P_{1,2}$ is derived from meaning of $B_{1,1}$ and of $P_{1,2}$
- ☺ Meaning in propositional logic is context-independent
  - (unlike natural language, where meaning depends on context)

- ☹ Propositional logic has very limited expressive power
  - (unlike natural language)
  - E.g., cannot say "pits cause breezes in adjacent squares“
    - except by writing one sentence for each square

First-order logic

- Whereas propositional logic assumes the world contains facts,
- first-order logic (like natural language) assumes the world contains
  - Objects: people, houses, numbers, colors, baseball games, wars, …
  - Relations: red, round, prime, brother of, bigger than, part of, comes between, …
  - Functions: father of, best friend, one more than, plus, …

One to one mapping

Syntax of FOL: Basic elements

- Constants KingJohn, 2, NUS,...
- Predicates Brother, >,...
- Functions Sqrt, LeftLegOf,...
- Variables x, y, a, b,...
- Connectives $\neg$, $\Rightarrow$, $\wedge$, $\vee$, $\Leftrightarrow$
- Equality =
- Quantifiers $\forall$, $\exists$

Atomic sentences

Atomic sentence = *predicate* ($term_1$,...,$term_n$)
or $term_1$ = $term_2$

Term = *function* ($term_1$,...,$term_n$)
or *constant* or *variable*

Functions can be viewed as complex names for constants

- E.g., *Brother(KingJohn,RichardTheLionheart) > (Length(LeftLegOf(Richard)), Length(LeftLegOf(KingJohn)))*

Complex sentences

- Made from atomic sentences using connectives

$$\neg S,\ S_1 \wedge S_2,\ S_1 \vee S_2,\ S_1 \Rightarrow S_2,\ S_1 \Leftrightarrow S_2,$$

e.g.,

Sibling(KingJohn,Richard) $\Rightarrow$ *Sibling (Richard,KingJohn)*

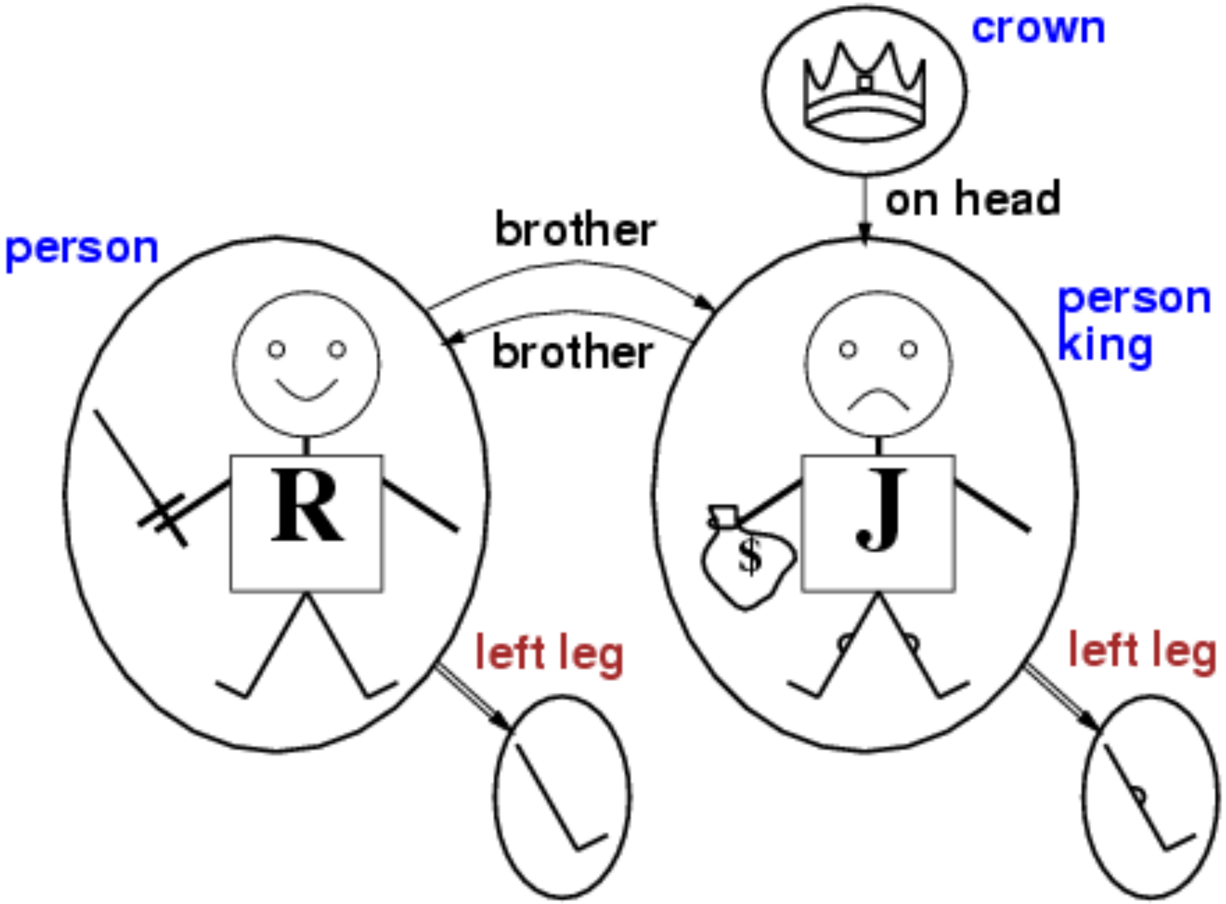
$>(1,2) \vee \leq (1,2)$

$>(1,2) \wedge \neg >(1,2)$

Truth in first-order logic

- Sentences are true with respect to a model and an interpretation
- Model contains objects (domain elements) and relations among them
- Interpretation specifies referents for
  constant symbols → objects
  predicate symbols → relations
  function symbols → functional relations
- An atomic sentence *predicate*($term_1,\ldots,term_n$) is true
  iff the objects referred to by $term_1,\ldots,term_n$
  are in the relation referred to by *predicate*

Model Example



Universal quantification

- $\forall$<*variables*> <*sentence*>

Everyone at NUS is smart:
$\forall x\ At(x,NUS) \Rightarrow Smart(x)$

- $\forall x$ *P* is true in a model *m* iff *P* is true with *x* being each possible object in the model

- Roughly speaking, equivalent to the conjunction of instantiations of *P*

$$
\begin{array}{ll}
 & At(KingJohn,NUS) \Rightarrow Smart(KingJohn) \\
\wedge & At(Richard,NUS) \Rightarrow Smart(Richard) \\
\wedge & At(NUS,NUS) \Rightarrow Smart(NUS) \\
\wedge\ \ldots &
\end{array}
$$

A common mistake to avoid

- Typically, $\Rightarrow$ is the main connective with $\forall$

- Common mistake: using $\wedge$ as the main connective with $\forall$:

  $\forall x\ At(x,NUS) \wedge Smart(x)$

  means

Existential quantification

- $\exists$<*variables*> <*sentence*>

Someone at NUS is smart:
$\exists x$ At(x,NUS) $\wedge$ Smart(x)

- $\exists x\ P$ is true in a model *m* iff *P* is true with *x* being some possible object in the model

- Roughly speaking, equivalent to the disjunction of instantiations of *P*

  At(KingJohn,NUS) $\wedge$ Smart(KingJohn)
  $\vee$ At(Richard,NUS) $\wedge$ Smart(Richard)
  $\vee$ At(NUS,NUS) $\wedge$ Smart(NUS)
  $\vee$ ...

Another common mistake

- Typically, $\wedge$ is the main connective with $\exists$

- Common mistake: using $\Rightarrow$ as the main connective with $\exists$:

$$\exists x\ At(x,NUS) \Rightarrow Smart(x)$$

is true if

Properties of quantifiers

- $\forall x\ \forall y$ is the same as $\forall y\ \forall x$
- $\exists x\ \exists y$ is the same as $\exists y\ \exists x$

- $\exists x\ \forall y$ is not the same as $\forall y\ \exists x$

  $\exists x\ \forall y\ Loves(x,y)$
  - "There is a person who loves everyone in the world"

  $\forall y\ \exists x\ Loves(x,y)$
  - "Everyone in the world is loved by at least one person"

- Quantifier duality: each can be expressed using the other
- $\forall x\ Likes(x,IceCream)$ $\quad\quad \neg\exists x\ \neg Likes(x,IceCream)$
- $\exists x\ Likes(x,Broccoli)$ $\quad\quad \neg\forall x\ \neg Likes(x,Broccoli)$

Equality

- $term_1 = term_2$ is true under a given interpretation if and only if $term_1$ and $term_2$ refer to the same object

- E.g., definition of *Sibling* in terms of *Parent*:

  $\forall x,y\ Sibling(x,y) \Leftrightarrow [\neg(x = y) \land \exists m,f\ \neg (m = f) \land Parent(m,x) \land Parent(f,x) \land Parent(m,y) \land Parent(f,y)]$

Using FOL

In the kinship domain:

- Brothers are siblings

  $\forall$x,y *Brother(x,y)* $\Rightarrow$ *Sibling(x,y)*
- One's mother is one's female parent

  $\forall$m,c *Mother(c)* = m $\Leftrightarrow$ *(Female(m)* $\wedge$ *Parent(m,c))*
- “Sibling” is symmetric

  $\forall$x,y *Sibling(x,y)* $\Leftrightarrow$ *Sibling(y,x)*

Using FOL

The set domain:

- $\forall s\ Set(s) \Leftrightarrow (s = \{\}\ ) \lor (\exists x, s_2\ Set(s_2) \land s = \{x|s_2\})$
- $\neg \exists x, s\ \{x|s\} = \{\}$
- $\forall x, s\ x \in s \Leftrightarrow s = \{x|s\}$
- $\forall x, s\ x \in s \Leftrightarrow [\ \exists y, s_2\} (s = \{y|s_2\} \land (x = y \lor x \in s_2))]$
- $\forall s_1, s_2\ s_1 \subseteq s_2 \Leftrightarrow (\forall x\ x \in s_1 \Rightarrow x \in s_2)$
- $\forall s_1, s_2\ (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \land s_2 \subseteq s_1)$
- $\forall x, s_1, s_2\ x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \land x \in s_2)$
- $\forall x, s_1, s_2\ x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \lor x \in s_2)$

Interacting with FOL KBs

- Suppose a wumpus-world agent is using an FOL KB and perceives a smell and a breeze (but no glitter) at *t=5*:

  `Tell`(KB,Percept([Smell,Breeze,None],5))
  `Ask`(KB,∃a BestAction(a,5))

  i.e., does the KB entail some best action at *t=5*?

- Answer: *Yes*, {*a/Shoot*} ← substitution (binding list)

- Given a sentence *S* and a substitution σ,
- *S*σ denotes the result of plugging σ into *S*; e.g.,
  S = Smarter(x,y)
  σ = {x/Hillary,y/Bill}
  *S*σ = Smarter(Hillary,Bill)

- `Ask`(KB,S) returns some/all σ such that KB ╞ σ

KB for the wumpus world

- Perception
  - $\forall t,s,b\ Percept([s,b,Glitter],t) \Rightarrow Glitter(t)$
- Reflex
  - $\forall t\ Glitter(t) \Rightarrow BestAction(Grab,t)$

Deducing hidden properties

- $\forall x,y,a,b$ *Adjacent*$([x,y],[a,b]) \Leftrightarrow$
  $[a,b] \in \{[x+1,y], [x-1,y],[x,y+1],[x,y-1]\}$

Properties of squares:

- $\forall s,t$ *At*$(\text{Agent},s,t) \wedge \text{Breeze}(t) \Rightarrow \text{Breezy}(s)$

Squares are breezy near a pit:

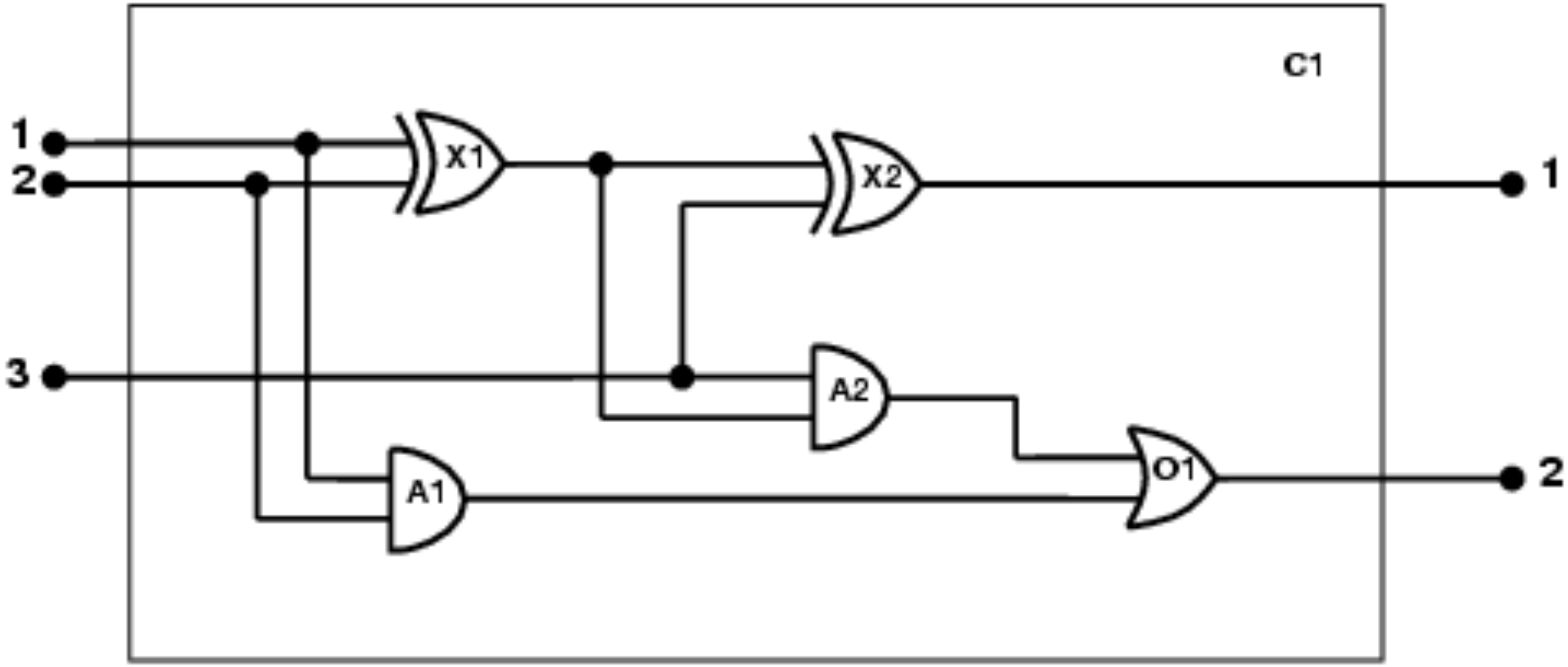
- Diagnostic rule - infer cause from effect
  $\forall s\ \text{Breezy}(s) \Rightarrow \exists r\ \text{Adjacent}(r,s) \wedge \text{Pit}(r)$
- Causal rule - infer effect from cause
  $\forall r\ \text{Pit}(r) \Rightarrow [\forall s\ \text{Adjacent}(r,s) \Rightarrow \text{Breezy}(s)\ ]$

Knowledge engineering in FOL

1. Identify the task
2. Assemble the relevant knowledge
3. Decide on a vocabulary of predicates, functions, and constants
4. Encode general knowledge about the domain
5. Encode a description of the specific problem instance
6. Pose queries to the inference procedure and get answers
7. Debug the knowledge base

The electronic circuits domain

One-bit full adder



The electronic circuits domain

1. Identify the task
   - Does the circuit actually add properly? (circuit verification)

2. Assemble the relevant knowledge
   - Composed of wires and gates; Types of gates (AND, OR, XOR, NOT)
   - Irrelevant: size, shape, color, cost of gates

3. Decide on a vocabulary
   - Alternatives:

     $Type(X_1) = XOR$

     $Type(X_1, XOR)$

     $XOR(X_1)$

The electronic circuits domain

4. Encode general knowledge of the domain
   - $\forall t_1,t_2$ Connected$(t_1, t_2) \Rightarrow$ Signal(t_1) = Signal(t_2)
   - $\forall t$ Signal$(t) = 1 \vee$ Signal$(t) = 0$
   - $1 \neq 0$
   - $\forall t_1,t_2$ Connected$(t_1, t_2) \Rightarrow$ Connected(t_2, t_1)
   - $\forall g$ Type(g) = OR $\Rightarrow$ Signal(Out(1,g)) = 1 $\Leftrightarrow \exists n$ Signal(In(n,g)) = 1
   - $\forall g$ Type(g) = AND $\Rightarrow$ Signal(Out(1,g)) = 0 $\Leftrightarrow \exists n$ Signal(In(n,g)) = 0
   - $\forall g$ Type(g) = XOR $\Rightarrow$ Signal(Out(1,g)) = 1 $\Leftrightarrow$ Signal(In(1,g)) $\neq$ Signal(In(2,g))
   - $\forall g$ Type(g) = NOT $\Rightarrow$ Signal(Out(1,g)) $\neq$ Signal(In(1,g))

The electronic circuits domain

5. Encode the specific problem instance

Type(X_1) = XOR
Type(A_1) = AND
Type(O_1) = OR

Type(X_2) = XOR
Type(A_2) = AND

Connected(Out(1,X_1),In(1,X_2))
Connected(Out(1,X_1),In(2,A_2))
Connected(Out(1,A_2),In(1,O_1))
Connected(Out(1,A_1),In(2,O_1))
Connected(Out(1,X_2),Out(1,C_1))
Connected(Out(1,O_1),Out(2,C_1))

Connected(In(1,C_1),In(1,X_1))
Connected(In(1,C_1),In(1,A_1))
Connected(In(2,C_1),In(2,X_1))
Connected(In(2,C_1),In(2,A_1))
Connected(In(3,C_1),In(2,X_2))
Connected(In(3,C_1),In(1,A_2))

The electronic circuits domain

6. Pose queries to the inference procedure

What are the possible sets of values of all the terminals for the adder circuit?

$\exists i_1,i_2,i_3,o_1,o_2$ Signal(In(1,C_1)) = $i_1 \wedge$ Signal(In(2,C_1)) = $i_2 \wedge$ Signal(In(3,C_1)) = $i_3 \wedge$ Signal(Out(1,C_1)) = $o_1 \wedge$ Signal(Out (2,C_1)) = o_2

7. Debug the knowledge base

May have omitted assertions like $1 \neq 0$

Summary

- First-order logic:
  - objects and relations are semantic primitives
  - syntax: constants, functions, predicates, equality, quantifiers
- Increased expressive power: sufficient to define wumpus world

PROgramming in LOGic

A crash course in Prolog

Slides edited from William Clocksin's versions at Cambridge Univ.

What is Logic Programming?

- A type of programming consisting of facts and relationships from which the programming language can draw a conclusion.
  - In *imperative programming* languages, we tell the computer what to do by programming the procedure by which program states and variables are modified.
  - In contrast, in *logical programming*, we don't tell the computer exactly what it should do (i.e., how to derive a conclusion). User-provided facts and relationships allow it to derive answers via logical inference.

- Prolog is the most widely used logic programming language.

Prolog Features

- Prolog uses **logical variables**. These are not the same as variables in other languages. Programmers can use them as ‘holes’ in data structures that are gradually filled in as computation proceeds.
- **Unification** is a built-in term-manipulation method that passes parameters, returns results, selects and constructs data structures.
- Basic control flow model is **backtracking**.
- **Program clauses and data** have the same form.
  - A Prolog program can also be seen as a relational database containing rules as well as facts.

Example: Concatenate lists a and b

In an imperative language

```
list procedure cat(list a, list b)
{
   list t = list u = copylist(a);
   while (t.tail != nil) t = t.tail;
   t.tail = b;
   return u;
}
```

In a functional language

```
cat(a,b) ≡
 if b = nil then a
else cons(head(a), cat(tail(a),b))
```

In a declarative language

```
cat([], Z, Z).
cat([H|T], L, [H|Z]) :- cat(T, L, Z).
```

Outline

- General Syntax
- Terms
- Operators
- Rules
- Queries

Syntax

- .pl files contain lists of clauses
- *Clauses* can be either *facts* or *rules*

Predicate, arity 1 (male/1)

```
male(bob).
male(harry).
child(bob,harry).
son(X,Y):-
      male(X),child(X,Y).
```

Terminates a clause

Argument to predicate

Indicates a rule

“and”

No space between functor and argument list

Complete Syntax of Terms

N.B. : case of **V**ariables and **t**erms and **c**onstants switched from FOL

Term

Constant

Names an individual

Atom

alpha17
gross_pay
john_smith
dyspepsia
+
=/=
'12Q&A'

Number

0
1
57
1.618
2.04e-27
-13.6

Compound Term

Names an individual that has parts

likes(john, mary)
book(dickens, Z, cricket)
f(x)
[1, 3, g(a), 7, 9]
-(+(15, 17), t)
15 + 17 - t

A **list** is made of a terms, separated by commas and enclosed by brackets.

Variable

Stands for an individual unable to be named when program is written

X
Gross_pay
Diagnosis
_257

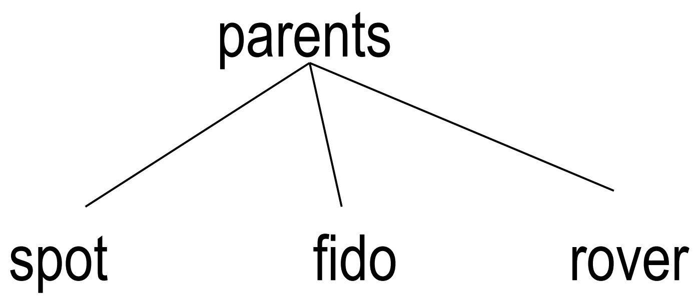
Compound Terms

The parents of Spot are Fido and Rover.

parents(spot, fido, rover)

Functor (an atom) of arity 3.   *components (any terms)*

It is possible to depict the term as a tree:



Examples of operator properties

Prolog has shortcuts in notation for certain operators (especially arithmetic ones)

Position	Operator Syntax	Normal Syntax
Prefix:	-2	-(2)
Infix:	5+17	+(17,5)

Associativity: left, right, none.
X+Y+Z is parsed as (X+Y)+Z
because addition is left-associative.

Precedence: an integer.
X+Y*Z is parsed as X+(Y*Z)
because multiplication has higher precedence.

These are all the same as the normal rules of arithmetic.

Rules

- Rules combine facts to increase knowledge of the system

```
son(X,Y):-
     male(X),child(X,Y).
```

- X is a son of Y if X is male and X is a child of Y

Interpretation of Rules

Rules can be given a declarative reading or a procedural reading.

Form of rule:	H :- G_1, G_2, …, G_n.
Declarative reading:	"That H is provable follows from goals G_1, G_2, …, G_n being provable."
Procedural reading:	"To execute procedure H, the procedures called by goals G_1, G_2, …, G_n are executed first."

Queries

- Prolog is interactive; you load a KB and then ask queries
- Composed at the ?- prompt
- Returns values of bound variables and yes or no

```
?- son(bob, harry).
yes
?- king(bob, france).
no
```

Another example

```
likes(george,kate).
likes(george,susie).
likes(george,wine).

?- likes(george,X)
X = kate
;
X = susie
;
X = wine
;
no
```

Answer: kate or susie or wine or false

Quantifiers

When a variable appears in the specification of a database,
the variable is universally quantified. Example:

likes(susie,Y)

One interpretation:
'Susie likes everyone'

For the existential quantifier one may do two things:

a. Enter the value directly into the database
   likes(george,Z) becomes likes(george,wine)
b. Query the interpreter
   ?- likes(george,Z) returns a value for Z if one exists

Points to consider

- Variables are bound by Prolog, not by the programmer
  - You can't assign a value to a variable.
- Successive user prompts ; cause the interpreter to return all terms that can be substituted for X.
  - They are returned in the order found.
  - Order is important

';' means Or
',' means And

- PROLOG adopts the **closed-world assumption**:
  - All knowledge of the world is present in the database.
  - If a term is not in the database assume is false.
  - Prolog's '**yes**' = I can prove it, '**no**' = I can't prove it.

Two things to think about:
When would the closed-world assumption lead to false inferences?
When would the different ordering of solutions cause problems?

Blank spaces to fill in on this slide

Queries

- Can bind answers to questions to variables
- Who is bob the son of? (X=harry)

?-

- Who is male? (X=bob, harry)

?-

- Is bob the son of someone? (yes)

?-

_ = Anonymous variable, don't care what it's bound to.

Lists

- The first element of a list can be separated from the tail using operator |

Example:

Match the list [tom,dick,harry,fred] to

[X|Y]	then X = tom and Y = [dick,harry,fred]
[X,Y|Z]	then X = tom, Y = dick, and Z = [harry,fred]
[V,W,X,Y,Z|U]	will not match
[tom,X|[harry,fred]]	gives X = dick

Example: List Membership

- We want to write a function member that works as follows:

?- member(a,[a,b,c,d,e])
yes
?- member(a,[1,2,3,4])
no
?- member(X,[a,b,c])
X* = *a
;
X* = *b
;
X* = *c
;
no

Can you do it?

Function Membership Solution

Define two predicates:

- member(X,[X|T]).
- member(X,[Y|T]) :- member(X,T).

A more elegant definition uses anonymous variables:

- member(X,[X,_]).
- member(X,[_|T]) :- member(X,T).

Again, the symbol _ indicates that the contents of that variable is unimportant.

Notes on running Prolog

You will often want to load a KB on invocation of Prolog

- Use “consult(‘mykb.pl’).” at the “?-” prompt.
- Or add it on the command line as a standard input “pl < mykb.pl”

If you want to modify facts once Prolog is invoked:

- Use “assert(p).”
- Or “retract(p).” to remove a fact

Prolog Summary

- A Prolog program is a set of specifications in FOL. The specification is known as the database of the system.
- Prolog is an interactive language (the user enters queries in response to a prompt).
- PROLOG adopts the closed-world assumption

- How does Prolog find the answer(s)? We return to this next week in **Inference in FOL**