First Order Logic

Sentence ::= Atom
    | (Sentence Connective Sentence)
    | ∀ Variable . Sentence
    | ¬ ∃ Variable . Sentence
    | ¬ Sentence

Atom ::= Proposition
    | Predicate(Term, ...)

Term ::= Constant
    | Variable
    | Function(Term, ...)

Dr. Ehab Al-Shaer/ Network Security
First-Order Logic – network examples

All machines are reachable
\( \forall x . (\text{machine}(x) \supset \text{reachable}(x)) \)

No machine is not reachable.
\( \neg \exists x . (\text{machine}(x) \land \neg \text{reachable}(x)) \)

Every machine has a router that can directly forward to.
\( \forall x . (\text{machine}(x) \supset \exists y . (\text{router}(y) \land \text{forward}(x,y))) \)

Every node is connected to it’s successor.
\( \forall x . (\text{node}(x) \supset \exists y . \text{nexthop}(x,y) \land \text{connected}(x,y)) \)

No machine is more than 5 hops away.
\( \neg \exists x \exists y . \text{reachable}(x) \land \text{loc}(x,y) \land \text{less_than}(y, FIVE) \)
First-Order Logic –

network security examples

All MS Window machines are patched
\[ \forall x \ . \ (\text{machine}(x) \land \text{window}(x) \implies \text{patched}(x)) \]

No unpatched machine is not remotely exploitable.
\[ \neg \exists x \ . \ (\text{machine}(x) \land \neg \text{patched}(x) \land \neg \text{rem exploitable}(x)) \]

Every exploitation to DB server can lead to attack all db clients interacting with this server
\[ \forall x, y \ . \ (\text{machine}(x) \land \text{dbserver}(x) \land \text{compromized}(x) \land \text{dbclient}(y) \land \text{connected}(y, x) \implies \text{compromized}(y)) \]

Every machine has a browser that can connect to the internet
\[ \forall x \ . \ (\text{machine}(x) \implies \exists y \ . \ (\text{browser}(x, y) \land \text{internet access}(y))) \]
**Unification**

Can resolve clauses if can unify one pair of literals

- Same predicate, one positive, one negative
- Match variable(s) to other variables, constants, or complex terms (function symbols)
- Carry bindings on variables through to all the other literals in the result

\[
\begin{align*}
(Mortal(HENRY)) & & (-Mortal(y) \lor Fallible(y)) \\
(Fallible(HENRY)) & & (Mortal(y) \rightarrow Fallible(y))
\end{align*}
\]
You always hurt the ones you love.
Politicians love themselves.
Therefore, politicians hurt themselves.

\[-\text{love}(x,y) \lor \text{hurt}(x,y)\]
\[\text{love}(x,y) \rightarrow \text{hurt}(x,y)\]
\[-\text{politician}(z) \lor \text{love}(z,z)\]
\[\text{politician}(z) \rightarrow \text{love}(z,z)\]
\[-\text{politician}(z) \lor \text{hurt}(z,z)\]
You always hurt the ones you love.
Politicians love themselves.
Therefore, politicians hurt themselves.

\[ \neg \text{love}(x, y) \lor \text{hurt}(x, y) \]
\[ \neg \text{politician}(z) \lor \text{love}(z, z) \]

rename “z” as “w” so that no clauses have variables with the same name

\[ \neg \text{politician}(w) \lor \text{hurt}(w, w) \]
happy.pl

happy(X) :- rich(X).
happy(X) :- loves(X,Y),happy(Y).
loves(X,Y) :- spouse(X,Y).
loves(X,Y) :- mother(X,Y).

rich(bill).
spouse(melinda,bill).
mother(elaine,melinda).
mother(mary,bill).
rich(paul).
mother(barbara,henry).

QUERIES:

?- happy(bill).

YES

?- happy(henry).

NO

?- happy(Z).
Exercise: The Mini Zebra Puzzle

There are three houses in a row on street. Each house is inhabited by a man of a different nationality, who has a different pet, and drinks a different beverage.

The Spaniard own a dog.
The Ukranian drinks tea.
The man in the third house drinks milk.
The Norwegian lives next to the tea drinker.
The juice drinker owns a fox.
The fox is next door to the dog.

Question: Who owns the zebra?
Prolog Limitations

• Only handles definite clauses (exactly one positive literal per clause)
  • Cannot express e.g. happy(bill) v happy(henry)
• Tree-shaped proofs means some sub-steps may be repeatedly derived
  • DATALOG: does forward-chaining inference and caches derived unit clauses
• Interpreter can get into an infinite loop if care is not taken in form & order of clauses
What is Prolog?

- PROgrammation et Logique.
- Edinburgh syntax is the basis of ISO standard.
- High-level interactive language.
- Logic programming language.

- Based on Horn Clauses
  - \((\text{parent}(X,Z) \land \text{ancestor}(Z,Y)) \supset \text{ancestor}(X,Y)\)
What is Prolog? (2)

- Programming languages are of two kinds:
  - **Procedural** (BASIC, ForTran, C++, Pascal, Java);
  - **Declarative** (LISP, Prolog, ML).

- In procedural programming, we tell the computer **how** to solve a problem.
- In declarative programming, we tell the computer **what** problem we want solved.

- (However, in Prolog, we are often forced to give clues as to the solution method).
What is Prolog used for?

- **Good at**
  - Grammars and Language processing,
  - Knowledge representation and reasoning,
  - Unification,
  - Pattern matching,
  - Planning and Search.
  - i.e. Prolog is good at Symbolic AI.

- **Poor at:**
  - Repetitive number crunching,
  - Representing complex data structures,
  - Input/Output (interfaces).
Basic Elements of Prolog

- Our program is a database of **facts** and **rules**.

- Some are always true (**facts**):
  
  \[
  \text{father( john, jim).}
  \]

- Some are dependent on others being true (**rules**):
  
  \[
  \text{parent( Person1, Person2 ) :-}
  \text{father( Person1, Person2 ).}
  \]

- To run a program, we ask questions about the database.
Prolog in English

Example Database:

John is the father of Jim.
Jane is the mother of Jim.
Jack is the father of John.

Person 1 is a parent of Person 2 if
    Person 1 is the father of Person 2 or
    Person 1 is the mother of Person 2.

Person 1 is a grandparent of Person 2 if
    some Person 3 is a parent of Person 2 and
    Person 1 is a parent of Person 3.

Example questions:

Who is Jim's father?
Is Jane the mother of Fred?
Is Jane the mother of Jim?
Does Jack have a grandchild?
Prolog in Prolog

Example Database:

John is the father of Jim.
Jane is the mother of Jim.
Jack is the father of John.

Person 1 is a parent of Person 2 if
Person 1 is the father of Person 2 or
Person 1 is the mother of Person 2.

Person 1 is a grandparent of Person 2 if
some Person 3 is a parent of Person 2 and
Person 1 is a parent of Person 3.

Example questions:
Who is Jim's father?
Is Jane the mother of Fred?
Is Jane the mother of Jim?
Does Jack have a grandchild?

Example Database:

father( john, jim ).
mother( jane, jim ).
father( jack, john ).

parent( Person1, Person2 ) :-
    father( Person1, Person2 ).
parent( Person1, Person2 ) :-
    mother( Person1, Person2 ).
grandparent( Person1, Person2 ) :-
    parent( Person1, Person3 ),
    parent( Person3, Person2 ).

Example questions:
?- father( Who, jim ).
?- mother( jane, fred ).
?- mother( jane, jim ).
?- grandparent( jack, _ ).
Using Prolog

1. Strawberry Prolog
2. SWI-prolog
Prolog Fundamentals
Anatomy of a Program

- Prolog programs are made up of **facts** and **rules**.
- A **fact** asserts some property of an object, or relation between two or more objects.
  
  e.g. `parent(jane, alan).`

  Can be read as “Jane is the parent of Alan.”

- **Rules** allow us to infer that a property or relationship holds based on preconditions.
  
  e.g. `parent(X, Y) :- mother(X, Y).`

  = “Person X is the parent of person Y if X is Y’s mother.”
Predicate Definitions

- Both facts and rules are predicate definitions.

- ‘Predicate’ is the name given to the word occurring before the bracket in a fact or rule:

  \[ \text{parent}(\text{jane,alan}). \]

- By defining a predicate you are specifying which information needs to be known for the property denoted by the predicate to be true.

*Dr. Ehab Al-Shaer/ Network Security*
Clauses

- Predicate definitions consist of *clauses*.
  = An individual definition (whether it be a fact or rule).

  e.g.  
  
  mother(jane,alan).  = Fact

  parent(P1,P2):- mother(P1,P2).  = Rule

- A clause consists of a *head*
- and sometimes a *body*.
  - Facts don’t have a body because they are always true.
Arguments

- A predicate head consists of a *predicate name* and sometimes some *arguments* contained within brackets and separated by commas.

  \[ \text{mother}(\text{jane}, \text{alan}). \]

- A body can be made up of any number of *subgoals* (calls to other predicates) and *terms*.

- **Arguments** also consist of *terms*, which can be:
  - Constants e.g. jane,
  - Variables e.g. Person1, or
  - Compound terms.
Terms: Constants

Constants can either be:

- **Numbers:**
  - integers are the usual form (e.g. 1, 0, -1, etc), but
  - floating-point numbers can also be used (e.g. 3.0E7)

- **Symbolic (non-numeric) constants:**
  - always start with a **lower case alphabetic character** and contain any mixture of letters, digits, and underscores (but no spaces, punctuation, or an initial capital).
    - e.g. abc, big_long_constant, x4_3t).

- **String constants:**
  - are anything between single quotes e.g. ‘Like this’.

*Dr. Ehab Al-Shaer/ Network Security*
Terms: Variables

- **Variables always start with an upper case alphabetic character or an underscore.**
- Other than the first character they can be made up of any mixture of letters, digits, and underscores.

  e.g. X, ABC, _89two5, _very_long_variable

- There are no “types” for variables (or constants) – a variable can take any value.

- All Prolog variables have a “local” scope:
  - they only keep the same value within a clause; the same variable used outside of a clause does not inherit the value (this would be a “global” scope).

*Dr. Ehab Al-Shaer/ Network Security*
Naming tips

- Use real English when naming predicates, constants, and variables.
  
  e.g. “John wants to help Somebody.”
  
  Could be: \texttt{wants(john,to\_help,Somebody)}.
  
  Not: \texttt{x87g(j,\_789)}.

- Use a \textbf{Verb Subject Object} structure:
  
  \texttt{wants(john,to\_help)}.

- **BUT** do not assume Prolog Understands the meaning of your chosen names!

  - You create meaning by specifying the body (i.e. preconditions) of a clause.
A Prolog program consists of **predicate definitions**.
A predicate denotes a property/relationship between objects.
Definitions consist of **clauses**.
A clause has a **head** and a **body** (**Rule**) or just a **head** (**Fact**).
A head consists of a **predicate name and arguments**.
A clause body consists of a conjunction of **terms**.
Terms can be **constants**, **variables**, or **compound terms**.
We can set our program **goals** by typing a command that unifies with a clause head.
A goal unifies with clause heads in order (top down).
**Unification** leads to the instantiation of variables to values.
If any variables in the initial goal become instantiated this is reported back to the user.
Lists

- Lists are *recursively defined* structures.

  “An empty list, [], is a list.
  A structure of the form [X, …] is a list if X is a term and […] is a list, possibly empty.”

- This recursiveness is made explicit by the bar notation
  - \([\text{Head}|\text{Tail}]\) (‘|’ = bottom left PC keyboard character)

- **Head** must unify with a single term.
- **Tail** unifies with a list of any length, including an empty list, [].
  - the bar notation turns everything after the Head into a list and unifies it with Tail.
Head and Tail

| ?-[a,b,c,d]= [Head | Tail]. | ?-[a,b,c,d]= [X | Y | Z]. |
Head = a,
Tail = [b,c,d]? 
yes

| ?-[a] = [H | T]. |
H = a,
T = []; 
yes
Z = []? yes

| ?-[a,b,c]= [W | X | Y | Z]]. |
W = a,
X = b,
Y = c,
Z = [c,d];
yes

| ?-[] = [H | T]. |
no

| ?-[a | b | c | []] = List. |
List = [a,b,c]?
yes
List Processing Predicates: Member/2

- **Member/2** is possibly the most used user-defined predicate (i.e. you have to define it every time you want to use it!)
- It checks to see if a term is an element of a list.
  - It returns **yes** if it is
  - And **fails** if it isn’t.

<table>
<thead>
<tr>
<th>?- member(c, [a, b, c, d]).</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
</tr>
</tbody>
</table>

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

- It 1\textsuperscript{st} checks if the Head of the list unifies with the first argument.
  - If yes then succeed.
  - If no then fail first clause.
- The 2\textsuperscript{nd} clause ignores the head of the list (which we know doesn’t match) and recurses on the Tail.
List Processing Predicates: Member/2

|?- member(ringo, [john, paul, ringo, george]).

Fail(1): member(ringo, [john|_]).
(2): member(ringo, [_|paul, ringo, george]):-
   Call: member(ringo, [paul, ringo, george]).
      Fail(1): member(ringo, [paul|_]).
      (2): member(ringo, [ _|ringo, george]):-
         Call: member(ringo, [ringo, george]).
            Succeed(1): member(ringo, [ringo|_]).

1) member(H, [H|_]).
2) member(H, [ _|T]):-
   member(H, T).
Quick Aside: Tracing Prolog

- To make Prolog show you its execution of a goal type trace. at the command line.
  - Prolog will show you:
    - which goal is Called with which arguments,
    - whether the goal succeeds (Exit),
    - has to be Redone, or Fails.
  - The tracer also indicates the level in the search tree from which a goal is being called.
    - The number next to the goal indicates the level in the tree (top level being 0).
    - The leftmost number is the number assigned to the goal (every new goal is given a new number).

- To turn off the tracer type notrace.
| ?- trace.
| ?- member(ringo, [john, paul, ringo, george]).
| 1  Call: member(ringo, [john, paul, ringo, george]) ?
| 2  2 Call: member(ringo, [paul, ringo, george]) ?
| 3  3 Call: member(ringo, [ringo, george]) ?
| 3  3 Exit: member(ringo, [ringo, george]) ?
| 2  2 Exit: member(ringo, [paul, ringo, george]) ?
| 1  1 Exit: member(ringo, [john, paul, ringo, george]) ?
| yes
|
| ?- member(stuart, [john, paul, ringo, george]).
| 1  1 Call: member(ringo, [john, paul, ringo, george]) ?
| 2  2 Call: member(ringo, [paul, ringo, george]) ?
| 3  3 Call: member(ringo, [ringo, george]) ?
| 4  4 Call: member(stuart, [george]) ?
| 5  5 Call: member(stuart, []) ?  \[ ] does not match [H|T]
| 5  5 Fail: member(stuart, []) ?
| 4  4 Fail: member(stuart, [george]) ?
| 3  3 Fail: member(ringo, [ringo, george]) ?
| 2  2 Fail: member(ringo, [paul, ringo, george]) ?
| 1  1 Fail: member(ringo, [john, paul, ringo, george]) ?
| no

Dr. Ehab Al-Shaer/ Network Security
Collecting Results

- When processing data in Prolog there are three ways to collect the results:
  1. Compute result at base case first, then use this result as you backtrack through the program.
  2. Accumulate a result as you recurse into the program and finalise it at the base case.
  3. Recurse on an uninstantiated variable and accumulate results on backtracking.

- These all have different uses, effect the order of the accumulated data differently, and require different degrees of processing.

*Dr. Ehab Al-Shaer/ Network Security*
We want to define a predicate, `length/2`, which takes a list as its first argument and returns a number as the second argument that is equal to the length of the list.

We can use recursion to move through the list element by element and `is/2` to count as it goes.

```
listlength([],0).
listlength([_|T],N1):-
    listlength(T,N),
    N1 is N+1.
```

To make a counter we need to initialise it at a value i.e. zero.

As the counter increases during backtracking it needs to be initialised in the base case.
Compute lower result first: trace.

?- listlength([a,b,c],N).
1 1 Call: listlength([a,b,c],_489) ?
2 2 Call: listlength([b,c],_1079) ?
3 3 Call: listlength([c],_1668) ?
4 4 Call: listlength([],_2257) ?
4 4 Exit: listlength([],0) ?
5 4 Call: _1668 is 0+1 ?
5 4 Exit: 1 is 0+1 ?
3 3 Exit: listlength([c],1) ?
6 3 Call: _1079 is 1+1 ?
6 3 Exit: 2 is 1+1 ?
2 2 Exit: listlength([b,c],2) ?
7 2 Call: _489 is 2+1 ?
7 2 Exit: 3 is 2+1 ?
1 1 Exit: listlength([a,b,c],3) ?

N = 3 ? yes
Why compute lower result first?

```prolog
listlength([],0).
listlength([_|T],N1):-
    N1 is N+1,
    listlength(T,N).
```

```prolog
|?- listlength([a,b,c],N).
Instantiation error in is/2
fail
```

```prolog
|?- listlength([a,b,c],0).
1 Call: listlength([a,b,c],0) ?
  2    2 Call: _1055 is 0+1 ?
  2    2 Exit: 1 is 0+1 ?
  3    2 Call: listlength([b,c],1) ?
  4    3 Call: _2759 is 1+1 ?
  4    3 Exit: 2 is 1+1 ?
  5    3 Call: listlength([c],2) ?
  6    4 Call: _4463 is 2+1 ?
  6    4 Exit: 3 is 2+1 ?
  7    4 Call: listlength([],3) ?
  7    4 Exit: listlength([],3) ?
  5    3 Exit: listlength([c],2) ?
  3    2 Exit: listlength([b,c],1) ?
  1    1 Exit: listlength([a,b,c],0) ?
yes
```

*Dr. Ehab Al-Shaer/ Network Security*
head(H,[H|_]).
tail(T,[_|T]). % T is list
first(F,[F|_]). % the same as head
last(L,[L]).
last(L,[H|T]):- last(L,T).

prefix([],_).
prefix([H|T1],[H|T2]):- prefix(T1,T2).
suffix(S,S).
suffix([H|T],L):- suffix(T,L).

member(X,[X|_]).
member(X,[_|T]):- member(X,T).
nth_member(1,[M|_],M).
nth_member(N,[_|T],M):- N>1, N1 is N-1, nth_member(N1,T,M).
append([],L,L).
append([H|T],L,[H|LT]):- append(T,L,LT).
add2end(X,[H|T],[H|NewT]):- add2end(X,T,NewT).
add2end(X, [], [X]).
insert(X,[],[X]).
insert(X,[Y|T],[X,Y|T]):- X <= Y.
insert(X,[Y|T],[Y|NT]):- X > Y, insert(X,T,NT).
delete(X,[X|T],T). delete(X,[Y|T],[Y|NT]):- delete(X,T,NT).
**Program subset(Subset, List).**
subset([Head|Tail], List) :- member(Head, List), subset(Tail, List).
subset([], _). % The empty list is a subset of all lists

**intersection(List1, List2, Intersection).**
intersection([H|T], L, [H|U]) :- member(H,L),
intersection(T,L,U).
intersection([_|T], L, U) :-
intersection(T,L,U).
intersection(_,_,[]).

**Usage: reverse(List, ReversedList).**
reverse(List, Reverse) :- reverse_accumulate(List, [], Reverse).
reverse_accumulate([], List, List).
reverse_accumulate([Head|Tail], Accumulate, Reverse) :-
reverse_accumulate(Tail, [Head|Accumulate], Reverse).