First-Order Logic (FOL)

- FOL consists of the following parts:
  - Objects/terms
  - Quantified variables
  - Predicates
  - Logical connectives
  - Implication

Objects/Terms

- FOL is a formal system that allows us to reason about the real world.
- It is therefore no surprise that at the core of FOL we objects and terms that describe objects in the real world such as:
  - phil, betty, fido -- objects
  - pair(bob,susan) -- term
  - [chicken,turkey,duck] -- term

First-Order Logic

- Quantified variables allow us to talk about *sets* of objects/terms
  - *Universally* quantified variables

    ∀X – for All objects X

  - *Existentially* quantified variables

    ∃Y – there Exists an object Y

First-Order Logic

- Predicates
  - Predicates are functions that map their arguments into true/false where the domain is some universe, say U, and the co-domain is the set of Boolean values { true, false }, e.g., for the predicate p we have:

    $p: U \rightarrow \{ \text{true}, \text{false} \}$

  - Example: human(X)
    - $\text{human}: U \rightarrow \{ \text{true}, \text{false} \}$
    - human(tree) = false
    - human(paul) = true
  - Example: mother(X,Y)
    - $\text{mother}: U \times U \rightarrow \{ \text{true}, \text{false} \}$
    - mother(betty,paul) = true
    - mother(giraffe,peter) = false
- Another way of looking at predicates is as *properties* of objects.
- Note: if we do not make another assumptions on the universe then the universe is usually taken as the set of all possible objects.

First-Order Logic

- We can combine predicates and quantified variables to make statements on sets of objects
  - $\exists X[mother(X,paul)]$
    - there exists an object X such that X is the mother of Paul
  - $\forall Y[human(Y)]$
    - for all objects Y such that Y is human

First-Order Logic

- Logical Connectives: and, or, not
  - $\exists F\ \forall C[\text{parent}(F,C) \text{ and } \text{male}(F)]$
    - There exists an object F for all objects C such that F is a parent of C and F is male.
  - $\forall X[\text{day}(X) \text{ and } (\text{rainy}(X) \text{ or } \text{snowy}(X))]$
    - For all objects X such that X is a day and X is either rainy or snowy.

First-Order Logic

- If-then rules: $A \Rightarrow B$
  - $\forall X \forall Y$[parent(X,Y) and female(X) $\Rightarrow$ mother(X,Y)]
    - For all objects X and for all objects Y such that if X is a parent of Y and X is female then X is the mother of Y.
  - $\forall Q$[human(Q) $\Rightarrow$ mortal(Q)]
    - For all objects Q such that if Q is human then Q is mortal.
- We can combine quantified variables, predicates, logical connectives, and implication into WFF's (well-formed formulas)

First-Order Logic

- Modus Ponens

human(socrates)

$\forall Q[\text{human}(Q) \Rightarrow \text{mortal}(Q)]$

$\therefore$ mortal(socrates)

We reason with FOL by asserting truths and then use the implications to deduce consequences of these assertions.

First-Order Logic

- WFFs can become very complicated, consider

$$\forall ABCD[(p(A) \Rightarrow k(B)) \Rightarrow (q(C) \Rightarrow k(D))]$$

- Very difficult to automate

Horn Clause Logic

In Horn clause logic the form of the WFFs is restricted:

$$P_1 \wedge P_2 \wedge \ldots \wedge P_{n-1} \wedge P_n \Rightarrow P_0$$

Single predicate in consequent

Conjunctions only!

Where P_0 , P_1 , P_2, … P_{n-1}, P_n are predicates over universally quantified variables.

Proving things is computation!

Use resolution to reason with Horn clause expressions - resolution mimics the modus ponens using horn clause expressions.

Advantage: this can be done mechanically (Alan Robinson, 1965)

"Deduction is Computation"

J. Alan Robinson: A Machine-Oriented Logic Based on the Resolution Principle. J. ACM 12(1): 23-41 (1965)

Basic Programs

- Prolog programs follow the FOL style: assert truth and use the rules/implications to compute the consequences of these assertions.

Valid Horn Clause

human(socrates)

$\forall Q[\text{human}(Q) \Rightarrow \text{mortal}(Q)]$

$\therefore$ mortal(socrates)

Basic Prolog Programs

- Prolog programs consist of fact (assumptions) and inference rules.
- As opposed to natural deduction, Prolog is based on FOL.
- We can execute Prolog programs by trying to prove things via queries.

Example: a simple program

```
male(phil).
male(john).
female(betty).
```

Facts, Prolog will treat these as true and enters them into its knowledgebase.

We execute Prolog programs by posing queries on its knowledgebase:

Prompt → ?-

```
?- male(phil).
    true - because Prolog can use its knowledgebase to prove true.
?- female(phil).
    false - this fact is not in the knowledgebase.
```

Prolog - Queries & Goals

A query is a way to extract information from a logic program.

Given a query, Prolog attempts to show that the query is a logical consequence of the program; of the collection of facts.

In other words, a query is a goal that Prolog is attempting to satisfy (prove true).

When queries contain variables they are existentially quantified, consider **!!**

```
?- parent(X,liz).
```

The interpretation of this query is: prove that there is at least one object X that can be considered a parent of liz, or formally, prove that

$$\exists x[parent(x,liz)]$$

holds.

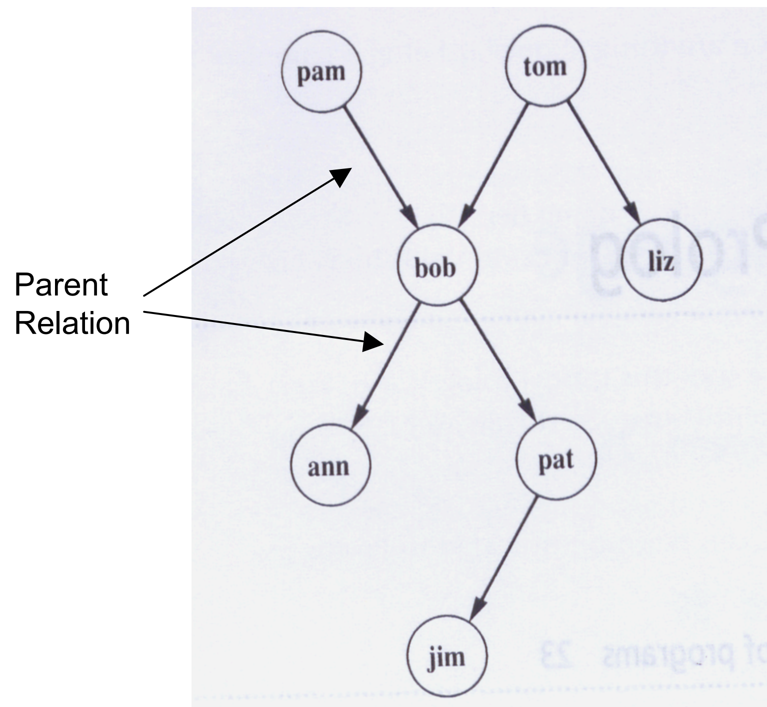
NOTE: Prolog will return all objects for which a query evaluates to true.

A Prolog Program

```
% a simple prolog program
female(pam).
female(liz).
female(ann).
female(pat).

male(tom).
male(bob).
male(jim).

parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
```



A Family Tree

Example Queries:

```
?- female(pam).
?- female(X).        ∃X[female(X)]?
?- parent(tom,Z).
?- father(Y).
```

Compound Queries

A compound query is the conjunction of individual simple queries.

Stated in terms of goals: a compound goal is the conjunction of individual subgoals each of which needs to be satisfied in order for the compound goal to be satisfied. Consider:

?- parent(X,Y) , parent(Y,ann).

or formally, show that the following holds,

$$\exists X,Y[parent(X,Y) \wedge parent(Y,ann)]$$

When Prolog tries to satisfy this compound goal, it will make sure that the two Y variables always have the same values.

Prolog uses unification and backtracking in order to find all the solutions which satisfy the compound goal.

Prolog Rules

Prolog rules are Horn clauses, but they are written "backwards", consider:

$$\forall X,Y[female(X) \wedge parent(X,Y) \Rightarrow mother(X,Y)]$$

is written in Prolog as

```
mother(X,Y) :- female(X), parent(X,Y) .
```

Implies ("think of $\Leftarrow$") — `:-`

"and" — `,`

head: `mother(X,Y)`

body: `female(X), parent(X,Y) .`

Prolog rules a implicitly universally quantified! !!

You can think of a rule as introducing a new "fact" (the head), but the fact is defined in terms of a compound goal (the body). That is, predicates defined as rules are only true if the associated compound goal can be shown to be true.

Prolog Rules

```
% a simple prolog program
female(pam).
female(liz).
female(ann).
female(pat).

male(tom).
male(bob).
male(jim).

parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).

mother(X,Y) :- female(X),parent(X,Y).
```

Queries:
?- mother(pam,bob).
?- mother(Z,jim).
?- mother(P,Q).

Prolog Rules

The same predicate name can be defined by multiple rules:

```
sibling(X,Y) :- sister(X,Y) .
sibling(X,Y) :- brother(X,Y).
```

Socrates Revisited

Consider the program relating humans to mortality:

```
mortal(X) :- human(X).
human(socrates).
```

We can now pose the query:

```
?- mortal(socrates).
```

True or false?

Declarative vs. Procedural Meaning

When interpreting rules purely as Horn clause logic statement → <u>declarative</u>

When interpreting rules as “specialized queries” → <u>procedural</u>

<u>Observation</u>: We design programs with declarative meaning in our minds, but the execution is performed in a procedural fashion.

Consider:

```
mother(X,Y) :- female(X),parent(X,Y).
```

Prolog Terms

- A term in Prolog is anything that cannot be considered a predicate
  - Simple object names, e.g. betty, john
  - Simple structures, e.g. couple(betty, john), in this case the important part here is that *couple* does not appear as a predicate definition
  - Lists

Lists & Pattern Matching

arity

- The unification operator: =/2
  - The expression A=B is true if A and B are terms and unify (look identical)

```
?- a = a.
  true
?- a = b.
  false
?- a = X.
  X = a
?- X = Y.
  true
```

Lists & Pattern Matching

- Lists – a convenient way to represent abstract concepts
  - Prolog has a special notation for lists.

[a]
[a,b,c]
[ ]

Empty List

[bmw, vw, mercedes]
[chicken, turkey, goose]

Lists & Pattern Matching

- Pattern Matching in Lists

```
?- [ a, b ] = [ a, X ].
X = b

?- [ a, b ] = X.
X = [ a, b ]
```

But:

```
?- [ a, b ] = [ X ].
no
```

The Head-Tail Operator: [H|T]

```
?- [a,b,c] = [X|Y];
X = a
Y = [b,c]

?- [a] = [Q|P];
Q = a
P = [ ]
```

Lists - the First Predicate

The predicate first/2: accept a list in the first argument and return the first element of the list in second argument.

```
first(List,E) :- List = [H|T], E = H;
```

Lists - the Last Predicate

The predicate last/2: accept a list in the first argument and return the last element of the list in second argument.

Recursion: there are always two parts to a recursive definition; the base and the recursive step.

```
last([A],A).
last([A|L],E) :- last(L,E).
```

Lists - the Append Predicate

<u>The append/3 predicate:</u> accept two lists in the first two parameters, append the second list to the first and return the resulting list in the third parameter.

```
append([ ], List, List).
append([H|T], List, [H|Result]) :- append(T, List, Result).
```

Prolog – Arithmetic

- Prolog is a programming language, therefore, arithmetic is implemented as expected.
- The only difference to other programming languages is that assignment is done via the predicate is rather than the equal sign, since the equal sign has been used for the unification operator.

Examples:

```
?- X is 10 + 5;
X = 15

?- X is 10 + 5 * 6 / 3;
X = 20
```

← Precedence and associativity of operators are respected.

Prolog – Arithmetic

Example: write a predicate definition for length/2 that takes a list in its first argument and returns the length of the list in its second argument.

```
length([ ], 0).
length(L, N) :- L = [H|T], length(T,NT), N is NT + 1.
```

Prolog – Arithmetic

<u>Example</u>: we can also use arithmetic in compound statements.

```
?- X is 5, Y is 2 * X.
X = 5
Y = 10
```

Prolog – I/O

- write(term)
  - is true if term is a Prolog term, writes term to the terminal.
- read(X)
  - is true if the user types a term followed by a period, X becomes unified to the term.
- nl
  - is always true and writes a newline character on the terminal.

☞ Extra-logical predicates due to the side-effect of writing/reading to/from the terminal.

Prolog – I/O

```
?- write(tom).
tom

?- write([1,2]).
[1, 2]

?- read(X).
|: boo.
X = boo

?- read(Q).
|: [1,2,3].
Q = [1, 2, 3]
```

Prolog I/O Prompt → |:

Prolog – I/O

<u>Example:</u> write a predicate definition for fadd/1 that takes a list of integers, adds 1 to each integer in the list, and prints each integer onto the terminal screen.

```
fadd([ ]).
fadd([ H | T ]) :- I is H + 1, write(I), nl, fadd(T).
```

Member Predicate

Write a predicate member/2 that takes a list as its first argument and an element as its second element. This predicate is to return true if the element appears in the list.

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

Exercises

(1) Define a predicate max/3 that takes two numbers as its first two arguments and unifies the last argument with the maximum of the two.

(2) Define a predicate maxlist/2 takes a list of numbers as its first argument and unifies the second argument with the maximum number in the list. The predicate should fail if the list is empty.

(3) Define a predicate ordered/1 that takes a list of numbers as its argument and succeeds if and only if the list is in non-decreasing order.

The ‘Cut’ Predicate

- The Cut predicate ‘!’ allows us to control Prolog’s backtracking behavior
- The Cut predicate forces Prolog to commit to a set of choice points
- Consider the following code:

```
different(A,B) :- A=B,!,fail.
different(_,_).
```

- Returns true if A and B are different and false if they are equal.

The Cut Predicate

```
a:-b,c,d.
c:-p,q,!,r,s.
c:-t.

b.
d.
p.
q:- ??.
r:- ??.
s.
t.

?- a.
```

- What would be the behavior if
  - q:-fail and r:-true
  - q:-true and r:-fail