LOGIC PROGRAMS

A logic program is defined as a finite set of rules (clauses)

$\boxed{\mathbf{A \leftarrow B_1, B_2, \ldots\ B_n.}}$ **In Prolog $\leftarrow$ is :-**

where LHS is the head and RHS is the body of the rule containing goals B_i

Note that this is the general form for all rules, facts, queries and called a clause (actually Horn clause).

- With $n = 0$ i.e. no body we have a unit clause, which is just a fact.
- With no head, we have a conjunctive set of goals

Answering a query is equivalent to determining whether the goal is a logical consequence of the program, using deduction rules

Lists and Recursion

List is a binary structure:

- first argument is an element, the head of the list
- second argument is recursively the rest or tail of the list
- written as [X | Y] where X is the head and Y is tail of the list
- empty list denoted by [ ]

%list(Xs) ← Xs is a list.
list([ ]).
list([X | Xs]) ← **list(Xs).**

Convention is that if X is head of a list Xs (plural) denotes tail

Proof tree of ?-list ([a, b, c]):

list([a, b, c]) ------ list([b, c]) ------ list([c]) ------ list([ ])

%member(Element, List) ← Element is an element of List
member(X, [X | Xs]).
member(X, [Y | Ys]) ← member(X, Ys).

- Declaratively, X is an element of a list if it is either the the head of the list or if it is a member of the tail of the list.

Example Append

%append(Xs, Ys, XsYs) :- XsYs is the result of concatenating Xs and Ys

```
append([ ], Ys, Ys ).
append( [ X | Xs ], Ys, [ X | Zs ]) ← append( Xs, Ys, Zs).
```

Proof tree for ?- append([a, b], [c, d], [a, b, c, d]).

$$
\begin{array}{c}
\text{append([a, b], [c, d], [a, b, c, d])} \\
\downarrow \\
\text{append([b], [c, d], [b, c, d])} \\
\downarrow \\
\text{append([ ], [c, d], [c, d])}
\end{array}
$$

Example Reverse/2 = 2 arguments

%reverse(List, RevList) ← RevList is the result of reversing List

```
reverse( [ ], [ ] ).
reverse( [ X | Xs ], Zs ) ← reverse( Xs, Ys),
        append( Ys,[ X ], Zs).
```

reverse([1],R).

reverse([2,1],R).

reverse([3,2,1],R).

Example Reverse/3 = 3 arguments

Examples so far have built recursive structures top-down with no access to partial structures. An extra argument can be used to build bottom-up:

```
%reverse( List, Tsil) :- Tsil is the result of reversing List
reverse( Xs, Ys) :-  reverse( Xs, [ ], Ys).
reverse( [X|Xs], Revs, Ys) :-  reverse( Xs, [X|Revs], Ys).
reverse( [ ], Ys, Ys).
```

reverse/3 is introduced with 2nd arg as an accumulator.

```
Trace:      reverse([a,b,c],Xs)
                reverse([a,b,c], [ ], Xs)
                    reverse([b,c], [a], Xs)
                        reverse([c], [b,a], Xs)
                            reverse([ ], [c,b,a], Xs)
                                Xs = [c,b,a]
                            TRUE
```

Note that 3rd argument is carried through recursion, and instantiated to reversed list

Arithmetic in Prolog

For efficiency uses built-in operators e.g. X is 3 + 5

Example: %factorial(N, F) :- F is the integer N factorial

```
factorial( N, F) :-  N > 0, N1 is N - 1,
                     factorial(N1, F1), F is N * F1.
factorial( 0, 1).
```

- cannot be used with 1st argument as variable

Iteration using Recursion

factorial/4 = 4 arguments

Example:

```
factorial( N, F) :- factorial( 0, N, 1, F).
factorial( I, N, T, F) :-  I < N, I1 is I + 1,
              T1 is T * I1, factorial( I1, N, T1, F).
factorial( N, N, F, F).
```

- computes : I is 0; T is 1;
  WHILE I < N DO
  I is I + 1; T is T * I END;
  RETURN T.
- I and T are values of loop vars before (I + 1)th iteration
- I is loop counter and T the accumulator
- 4th arg of factorial/4 is instantiated to the solution

The cut !

- reduces search space by dynamically pruning search tree
- prunes computation paths that contain solutions
- usually understood procedurally rather than declaratively
- is a goal which succeeds and commits Prolog to all choices made since parent goal was unified with head of clause that ! occurs in

- does not affect goals appearing to its right in the clause

$$\boxed{A \text{ :- } B_1, \ldots B_k, !, B_{k+2}, \ldots B_n}$$

if failure occurs in goal $B_{k+2}, \ldots B_n$ backtracking goes back only as far as the !. If no goal to the right of ! succeeds, search proceeds from last choice before unification with A

Cuts that do not change declarative meaning of program if removed are called 'green cuts'. In contrast 'red cuts' do change the meaning and require care when used.

Negation as failure using the red cut

- uses system predicate fail that always fails

    %not X : - if X is not provable {assumes not X equiv to not(X)}

    not X :- X, !, fail.
    not X.

    If X succeeds not X fails
    ELSE not X succeeds

    - note that meaning depends on rule order

Problems with nonground goals

unmarried_student(X) :- not married(X), student(X).
student(bill).
married(joe).
?- unmarried_student(X). fails since *not married(X)* fails ; with $X = joe$,
and the 'expected' solution $X = bill$ not found

Problem is that unmarried_student(bill) succeeds

STATE-SPACE SEARCH

Depth-first search is built into Prolog

solve_dfs(*State*,*History*,*Moves*) ←
Moves is a sequence of moves to reach a desired final state from the current *State*, where *History* contains the states visited previously.

```
solve_dfs(State,History,[ ]) ←
    final_state(State).
solve_dfs(State,History,[Move|Moves]) ←
    move(State,Move),
    update(State,Move,State1),
    legal(State1),
    not member(State1,History),
    solve_dfs(State1,[State1|History],Moves).
```

Testing the framework

```
test_dfs(Problem,Moves) ←
    initial_state(Problem,State), solve_dfs(State,[State],Moves).
```