Prolog Statements

- Prolog statements consist of facts, rules, and queries.
- Example of facts (written as headless Horn clauses)
  
  ```
  male(tony).
  male(greg).
  female(nikki).
  female(karen).
  ```

- Example of rules (written as headed Horn clauses)
  
  ```
  sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X), X \= Y.
  ```

- Facts and Rules are stored in a file with extension .pl

Loading the Knowledge Base

- Suppose the facts and rules are stored in a file called `c:\classes\prolog\gender.pl`

- To load the file:

  ```
  | ?- ['C:/classes/prolog/gender'].
  
  or

  | ?- consult('C:/classes/prolog/gender').
  ```
Prolog Statements

- Goal/Query statements
  - Also in the form of headless Horn clauses
  - `?- male(tony).`
  - yes

  - `?- female(X).
    X = nikki ? ;
    X = karen
    yes`

Query/Goal Statement

- A query can just be an assertion of a fact
  - Prolog will try to establish whether that fact can be shown to be true

- Alternatively, a query can contain variables
  - Prolog will try to find values for the variables that make the query true

- Variables are written with initial uppercase letters.
  - Prolog will print the variable values one by one. After each one, type
    - ";" to see the next value (solution),
    - "RETURN" to accept the last value (solution), or
    - "a" to print all values

Inference Process

- Given a goal/query, how does Prolog match the goal against the propositions in the database?

- In general, there are two approaches:
  - Forward chaining (bottom-up resolution)
    - System begin with the facts and rules of the database and attempt to find a sequence of matches that lead to the goal
  - Backward chaining (top-down resolution)
    - System begin with the goal and attempts to find a sequence of matching propositions that lead to some set of original facts in the database

- Prolog implementation uses backward chaining
Inference Process (Example)

Example:
- Given the following facts and rules:
  
  - \texttt{father(bob).
  
  - \texttt{man(X) :- father(X).}
  
- Given the following goal/query: \texttt{man(bob)}.

Forward chaining:
- Start with the first proposition: \texttt{father(bob)}.
- Goal is inferred by matching \texttt{father(bob)} to right side of the second rule
- This leads to the goal \texttt{man(bob)}.

Inference Process (Example)

Example:
- Given the following facts and rules:
  
  - \texttt{father(bob).
  
  - \texttt{man(X) :- father(X).}
  
- Given the following goal/query: \texttt{man(bob)}.

Backward chaining:
- Start with the goal: \texttt{man(bob)}.
- Match the goal to the left-hand side of second proposition
  - This would instantiate \texttt{X} to \texttt{bob}
- Match the right side of the second proposition \texttt{(father(bob))} to the first proposition

Inferencing Process of Prolog

What if there are more than one rule that matches a proposition? Which rule should be used next?
- \texttt{father(bob)}.
- \texttt{man(X) :- married(Y,Z), son(X,Y).}
- \texttt{man(X) :- father(X,Y).}
- \texttt{man(X) :- brother(X,Y).}

Solution Search Approaches
- Depth-first
  - finds a complete sequence of propositions-a proof for the first subgoal before working on the others
- Breadth-first
  - works on all subgoals of a given goal in parallel
- Backtracking
  - when the system finds a subgoal it cannot prove, it reconsiders the previous one to attempt to find an alternative solution and then continue the search-
  - multiple solutions to a subgoal result from different instantiations of its variables
Simple Arithmetic

- Prolog supports integer variables and integer arithmetic
  \[ 7 + 5 = 12 \]
  produces an answer equals to 12

- \( A = 5 \).

- How about this statement: \( \text{Sum} \equiv \text{Sum} + \text{Number} \)
  - Since \( \text{Sum} \) is not instantiated, the reference to it on the RHS is undefined and the clause fails

Simple Arithmetic Continued

- Given the following facts:
  
  \[
  \begin{align*}
  & \text{speed( ford, 100).} \\
  & \text{speed(chevy, 105).} \\
  & \text{speed(dodge, 95).} \\
  & \text{time(ford, 20).} \\
  & \text{time(chevy, 21).} \\
  & \text{time(dodge, 24).}
  \end{align*}
  \]

- We can calculate distance with this rule:
  
  \[
  \text{distance}(X,Y) :- \text{speed}(X, \text{Speed}), \\
  \text{time}(X, \text{Time}), Y \equiv \text{Speed} * \text{Time}
  \]

- Query:
  
  \[
  \text{| ?- distance(chevy, Chevy\_Distance).} \\
  \text{will instantiate Chevy\_Distance to 105*21 = 2205}
  \]

Tracing the Process

- Prolog has a built-in structure named \texttt{trace} that displays instantiations of values to variables at each step during the attempt to satisfy a given goal
- \texttt{trace} is used to understand and debug Prolog pgms

- Tracing model
  - describes Prolog execution in terms of 4 events
    - Call, which occurs at the beginning of an attempt to satisfy a goal
    - Exit, which occurs when a goal has been satisfied
    - Redo, which occurs when backtrack causes an attempt to resatisfy a goal
    - Fail, which occurs when a goal fails
  - In Gnu prolog:
    - \texttt{trace} \% This will switch on the trace mode
    - \texttt{Notrace} \% This will switch off the trace mode
Tracing the Process

Consider the following example database and goal:
likes(jake, chocolate).
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).
\[\text{likes(jake, X), likes(darcie, X).} \quad \text{This is the goal/query}\]

Trace Process:
1. Call: likes(jake, _0)?
   - Exit: likes(jake, chocolate)
2. Call: likes(darcie, chocolate)?
   - Fail: likes(darcie, chocolate)
3. Redo: likes(jake, _0)?
   - Exit: likes(jake, apricots)
4. Call: likes(darcie, apricots)?
   - Exit: likes(darcie, apricots)

\[X = \text{apricots}\]

List Structures

Prolog uses the syntax of ML and Haskell to specify lists.
List elements are separated by commas, and the entire list is delimited by square brackets.
Example: [apple, prune, grape, kumquat]
Example: [] means empty list

Like Haskell, Prolog uses the special notation
\[[ X \mid Y ]\] to indicate head X and tail Y
head and tail correspond CAR and CDR in LISP

Example:
Consider the following facts and rules:
\text{newlist([apple, apricot, pear, banana]).}
\text{first(X) :- newlist([X|Y]).}

In query mode,
\text{first(X).}

Statement instantiates X with apple
List Structures

Other Examples:

```
newlist ([X|Y])
  This will instantiate X to apple and Y to [apricot, pear, banana]
```

```
newlist ([apple, apricot|X]).
  This will instantiate X to [pear, banana]
```

List Structures

Append:
```
?- append([bob, jo], [jake, darcie], X).
produces the output X = [bob, jo, jake, darcie]
```

Reverse:
```
?- reverse([bob, jo, jake, darcie], X).
produces the output X = [darcie, jake, jo, bob]
```

member:
```
?- member(bob, [darcie, jo, jim, bob, alice]).
The system responds with the answer Yes
```

Writing your own append function:
```
?- myappend([], [banana, eggs, milk], List).
List = [banana, eggs, milk]
```
```
?- myappend([], List, [banana, eggs, milk]).
List = [banana, eggs, milk]
```
```
?- myappend(List1, List2, [banana, eggs, milk]).
List1 = [];
List1 = [banana, eggs, milk] ?;
List1 = [banana] ?;
List1 = [eggs, milk] ?;
List1 = [banana, eggs] ?;
List2 = [milk] ?;
List1 = [banana, eggs, milk]
List2 = [] ?;
```

Organization of Programming Languages - Cheng (Fall 2004)
**List Structures**

Writing your own append function:

```prolog
myappend([], List, List).
myappend([Head|Tail1], List2, [Head|List3]) :- myappend(Tail1, List2, List3).
```

- The first proposition specifies when the empty list is appended to any other list, that other list is the result.
- The second proposition specifies the characteristics of the new list.

Appending the list [Head1|Tail1] to List2 produces the list [Head1|List3] only if List3 is obtained by appending Tail1 and List2.

**List Structures**

Write your own reverse function

```prolog
myreverse([], []).
myreverse([Head|Tail], List) :-
  myreverse(Tail, Result),
  append(Result, [Head], List).
```

**List Structures**

Write your own member function:

```prolog
mymember(Element, [Element|_]).
mymember(Element, [_|List]) :-
  member(Element, List).
```
Size

?- size([a,b,c], N).
N = 3 ?
yes

Note: There is a predefined function called length which is the same as size

size([], 0).
size([H | T], N) :-
    size(T, N1), N is N1+1.

Towers of Hanoi

The object is to move the disks, one at a time, from the left peg to the right peg.
You are allowed to use the middle peg.
At no stage are you allowed to place a bigger disk on top of a smaller one.
Towers of Hanoi

- Move top disk from left to right
- Move top disk from left to middle

Towers of Hanoi

- Move top disk from left to right
- Move top disk from left to middle
- Move top disk from right to middle

Towers of Hanoi

- Move top disk from left to right
- Move top disk from left to middle
- Move top disk from right to middle
- Move top disk from left to right
Towers of Hanoi

- Move top disk from left to right
- Move top disk from left to middle
- Move top disk from right to middle
- Move top disk from left to right
- Move top disk from middle to left

Towers of Hanoi

- Move top disk from left to right
- Move top disk from left to middle
- Move top disk from right to middle
- Move top disk from left to right
- Move top disk from middle to left
- Move top disk from middle to right

Towers of Hanoi

- Move top disk from left to right
- Move top disk from left to middle
- Move top disk from right to middle
- Move top disk from left to right
- Move top disk from middle to left
- Move top disk from middle to right

- Move top disk from left to right
- Move top disk from left to middle
- Move top disk from right to middle
- Move top disk from left to right
- Move top disk from middle to left
- Move top disk from middle to right

- Move top disk from left to right
- Move top disk from left to middle
- Move top disk from right to middle
- Move top disk from left to right
- Move top disk from middle to left
- Move top disk from middle to right

- Move top disk from left to right
- Move top disk from left to middle
- Move top disk from right to middle
- Move top disk from left to right
- Move top disk from middle to left
- Move top disk from middle to right

- Move top disk from left to right
- Move top disk from left to middle
- Move top disk from right to middle
- Move top disk from left to right
- Move top disk from middle to left
- Move top disk from middle to right
Suppose we have \( N \) disk on the source pole.

The idea is that if we can move the top \( N-1 \) disks onto the middle pole, then we can simply move the largest disk at the bottom onto the destination pole.

By applying the same technique that we use to move the top \( N-1 \) disks, we can move the \( N-1 \) disks onto the dest pole.

---

move(1, X, Y, Z) :-
    write(' Move top disk from '), write(X),
    write(' to '), write(Y), nl.
move(N, X, Y, Z) :-
    N > 1, M is N - 1, move(M, X, Z, Y),
    move(1, X, Y, Z), move(M, Z, Y, X).
| ?- move(3, left, right, middle).  

---
Solution

\[ \text{move(3, left, right, middle).} \]
Move top disk from left to right
Move top disk from left to middle
Move top disk from right to middle
Move top disk from left to right
Move top disk from middle to left
Move top disk from middle to right
Move top disk from left to right

true ?

Quick Sort (Exercise)

Resolution Order Control

- Prolog always matches in the same order:
  - starting at the beginning and at the left end of a given goal
- User can affect efficiency of resolution by ordering the propositions to optimize a particular application
  - If certain rules are more likely to succeed than others during an execution, placing those rules first makes the program more efficient
Control of Backtracking

- The cut operator (!) is used to control backtracking.
- The (!) is actually a goal which always succeeds immediately, but it cannot be re-satisfied through backtracking.
- For example, the goal
  - A, B, !, C, D.
- would follow a left to right unification process up to C. If A and B succeed but C fails, we know it is a waste of time to backtrack to resatisfy A or B, so the whole goal fails if we use ! in the place immediately before C.

Problems with Control

- The ability to tamper with control flow in a Prolog program is a deficiency because it is detrimental to one of the advantages of logic programming: programs do not specify how to find solutions.
- Using the ability for flow control, clutters the simplicity of logic programs with details of order control to produce solutions.
- Frequently used for the sake of efficiency.

Negation

- Consider the example
  - parent(bill, jake).
  - parent(bill, shelley).
  - sibling(X, Y) :- parent(M, X), parent(M, Y).

?- sibling(X, Y).
Prolog's result is
X = jake
Y = jake
Negation

Consider the example

\[
\text{parent(bill, jake).} \\
\text{parent(bill, shelley).} \\
\text{sibling(X, Y) :- parent(M, X), parent(M, Y), X \neq Y.}
\]

?-sibling(X,Y).

Prolog's result is

\[
X = \text{jake} \\
Y = \text{shelley}
\]

Negation not a Logical Not

Negation means resolution cannot satisfy the subgoal

Logical Not cannot be a part of Prolog primarily because of the form of the Horn clause

\[
B : A_1 \land A_2 \land \ldots \land A_n
\]

If all the \(A\) propositions are true, it can be concluded that \(B\) is true. But regardless of the truth or falseness of any or all of the \(A\)s, it cannot be concluded that \(B\) is false. Prolog can prove that a given goal is true, but it cannot prove that a given goal is false.

Expert Systems

Expert Systems are designed to emulate human expertise. They consist of a database of facts, an inferencing process, some heuristics about the domain, and some friendly human interface.

Prolog provides the facilities of using resolution for query processing, adding facts and rules to provide the learning capability, and using trace to inform the user of the reasoning behind a given result.