Prolog

- Language constructs
  - Facts, rules, queries through examples
- Horn clauses
  - Goal-oriented semantics
  - Procedural semantics
- How computation is performed?
- Comparison to logic programming
Logic Programming vs Prolog

• Logic programming languages are not procedural or functional. Based on predicate calculus (remember 198:205?) -- represent using predicates/relations:

```java
class Student extends Person {int year; float gpa; major}
```
can be represented, among others, as

```prolog
student(_), person(_), inYear(_,_), hasGpa(_,_),...
```
```prolog
person(X):- student(X).
person(X):- student(X).
```
```prolog
student(jane). %% jane = new Student();
inYear(jane,3). %% jane.year = 3;
```

• Separate logic from control:
  • Separate the **What** (logic) from the **How** (control)
  • Programmer declares what facts and relations are true
  • System determines how to use facts to solve problems
  • State relationships and query them as in logic
Logic Programming

- Computation engine: theorem-proving
  - Uses unification, resolution, backward chaining, backtracking
- Programming style: uses recursion, like Scheme
- Problem description is higher-level than imperative languages
Prolog

• As database management
  – Start with program as a database of facts
  – Simple queries with constants and variables ("binding"), conjunctions and disjunctions
  – Add to program rules to derive additional facts
  – Two interpretations
    • Declarative: based on logic
    • Procedural: searching for answers to queries
      – Search trees and rule firings can be traced
Facts

likes(eve, pie).  food(pie).
likes(al, eve).  food(apple).
likes(eve, tom).  person(tom).
likes(eve, eve).
Queries (Asking Questions)

likes(eve, pie).
likes(al, eve).
likes(eve, tom).
likes(eve, eve).

food(pie).
food(apple).
person(tom).

?-likes(al, eve).

yes

?-likes(al, pie)

no

no

no

no

?-likes(person, food).

?-likes(al, Who).

Who=eve

?-likes(eve, W).

W=pie

W=tom

W=eve

no

no

no
Harder Queries

likes(eve, pie).  food(pie).
likes(al, eve).  food(apple).
likes(eve, tom).  person(tom).
likes(eve, eve).

?-likes(A,B).
A=eve, B=pie ; A=al, B=eve ; ...
?-likes(D,D).
D=eve ; no
?-likes(eve,W), person(W).
W=tom
?-likes(al,V), likes(eve,V).
V=eve ; no

and
Harder Queries

likes(eve, pie). food(pie).
likes(al, eve). food(apple).
likes(eve, tom). person(tom).
likes(eve, eve).

?-likes(eve, \texttt{W}), likes(\texttt{W}, V).
\texttt{W}=\texttt{eve}, V=pie ; \texttt{W}=\texttt{eve}, V=tom ; \texttt{W}=\texttt{eve}, V=\texttt{eve}

?-likes(eve, \texttt{W}), person(\texttt{W}), food(\texttt{V}).
\texttt{W}=\texttt{tom}, V=pie ; \texttt{W}=\texttt{tom}, V=apple or

?-likes(eve, V), (person(\texttt{V}); food(\texttt{V})).
\texttt{V}=\texttt{pie} ; \texttt{V}=\texttt{tom} ; no

?-likes(eve, \texttt{W}), \texttt{\backslash+n}likes(al, \texttt{W}).
\texttt{W}=\texttt{pie} ; \texttt{W}=\texttt{tom} ; no

same binding
rules
likes(eve, pie).  food(pie).
likes(al, eve).  food(apple).
likes(eve, tom).  person(tom).
likes(eve, eve).

• What if you want to ask the same question often?

Add a rule to the database:
\[ \text{rule1:-likes(eve,V),person(V).} \]

?–rule1.

yes
Rules

likes(eve, pie).  food(pie).
likes(al, eve).  food(apple).
likes(eve, tom).  person(tom).
likes(eve, eve).

rule1:-likes(eve,V),person(V).
rule2(V):-likes(eve,V),person(V).

?-rule2(H).
H=tom ; no
?-rule2(pie).
no

Note rule1 and rule2 are just like any other predicate!
Queen Victoria Example

male(albert). \hspace{1cm} a fact
female(alice). \hspace{1cm} Facts are put in a file.
male(edward).
nfemale(victoria).
parents(edward,victoria,albert).
parents(alice,victoria,albert).
?- [family]. \hspace{1cm} loads file
yes
?- male(albert). \hspace{1cm} a query
yes
?- male(alice).
no
?- parents(edward,victoria,albert).
yes
?- parents(bullwinkle,victoria,albert).
no

cf Clocksin and Mellish
Problem: facts alone do not make interesting programs possible. Need variables and deductive rules.

?-female(X).  % a query or proposed fact
X = alice ;  % asks for more answers
X = victoria ;  % if user types <return> then no more answers given
no % when no more answers left, return no

Variable X has been unified to all possible values that make female(X) true.
  - Performed by pattern match search

Variables capitalized, predicates and constants are lower case
Queen Victoria Example, cont.

sister_of(X,Y):-
    female(X),parents(X,M,F),parents(Y,M,F).

?- sister_of(alice,Y).
Y = edward

?- sister_of(alice, victoria).
no
Horn Clauses (logical foundations)

• A Horn Clause is: \( c \leftarrow h_1 \land h_2 \land h_3 \land \ldots \land h_n \)
  – **Antecedents** (h’s): conjunction of zero or more conditions which are *atomic formulae* in predicate logic
  – **Consequent** (c): an atomic formula in predicate logic

• **Meaning of a Horn clause:**
  – *The consequent is true if the antecedents are all true*
  – \( c \) is true if \( h_1, h_2, h_3, \ldots, \) and \( h_n \) are all true

\[ \text{likes(calvin,hobbes)} \leftarrow \text{tiger(hobbes), child(calvin)}. \]
Horn Clauses

• In Prolog, a Horn clause $c \leftarrow h_1 \land \ldots \land h_n$ is written $c ::= h_1 , \ldots , h_n$.
• Horn Clause is a Clause
• Consequent is a Goal or a Head
• Antecedents are Subgoals or Tail
• Horn Clause with No Tail is a Fact
  male(edward). dependent on no other conditions
• Horn Clause with Tail is a Rule
  father(albert,edward) :-
    male(edward), parents(edward,M,albert).
Horn Clauses

• Variables may appear in the antecedents and consequent of a Horn clause:

  – \( c(X_1, \ldots, X_n) :- h(X_1, \ldots, X_n, Y_1, \ldots, Y_k). \)

    For all values of \( X_1, \ldots, X_n \), the formula \( c(X_1, \ldots, X_n) \) is true if there exist values of \( Y_1, \ldots, Y_k \) such that the formula \( h(X_1, \ldots, X_n, Y_1, \ldots, Y_k) \) is true

  – Call \( Y_i \) an auxiliary variable. Its value will be bound to make consequent true, but not reported by Prolog, because it doesn’t appear in the consequent.
Declarative Semantics

• Prolog program consists of facts and rules

• Rules like

  sister_of(X,Y):-female(X),parents(X,M,F),parents(Y,M,F).

correspond to logical formulas

\[ X,Y \cdot \text{sister_of}(X,Y) \quad \exists M,F \cdot \text{female}(X), \text{parents}(X,M,F), \text{parents}(Y,M,F). \]

/* X is the sister of Y, if X is female, and there are M and F who are X’s parents, and Y’s parents */

  – Note that variables not in head are existentially quantified

• A query is a conjunction of atoms, to be proven

  – If query has no variables and is provable, answer is yes

  – If query has variables, proof process causes some variables to be bound to values (called a substitution); these are reported
Example

?-sister_of(X,Y):
    female(X),parents(X,M,F),parents(Y,M,F).

?-sister_of(alice,Y).
Y = edward

?-sister_of(X,Y).
X = alice
Y = edward  ;
X = alice
Y = alice  ;
no

What’s wrong here?

Example shows
- subgoal order of evaluation
- argument invertability
- backtracking
- computation in rule order
Procedural Semantics

?-sister_of(X,Y):
    female(X),parents(X,M,F),parents(Y,M,F).

• **First** *find* an X to make female(X) true
• **Second** *find* an M and F to make parents(X,M,F) true for that X.
• **Third** *find* a Y to make parents(Y,M,F) true for those M, F
• This algorithm is recursive; each *find* works on a new “copy” of the facts+rules. eventually, each find must be resolved by appealing to facts.
• Process is called *backward chaining*.
• Variables are *local*;
  • (every time rule is used, new names for X,Y,M,F)
Prolog Rule Ordering and Unification

• Rule ordering (from first to last) used in search
• Unification requires all instances of the same variable in a rule to get the same value
• Unification does not require differently named variables to get different values: sister_of(alice, alice)
• All rules searched if requested by successive typing of ;
Example

\[ \text{sis}(X,Y) :- \text{female}(X), \text{parents}(X,M,F), \\
\quad \text{parents}(Y,M,F), \neg (X = Y). \]

?- \text{sis}(X,Y).  \textit{last subgoal disallows } X, Y \textit{ to have same value}

X = alice
Y = edward  ;
no

= \text{means } \textit{unifies with}

== \text{means } \textit{same in value}
Negation as Failure

- \( \lnot(P) \) succeeds when \( P \) fails
  - Called negation by failure, defined:

\[
\text{not} \left( X \right) : - X, !, \text{fail}. \\
\text{not} \left( _ \right).
\]

- Which means

  - if \( X \) succeeds in first rule, then the rule is forced to fail by the last subgoal (\text{fail}). we cannot backtrack over the cut (\text{!}) in the first rule, and the cut prevents us from accessing the second rule.

  - if \( X \) fails, then the second rule succeeds, because “\( _ \)” (or don’t care) unifies with anything.
Transitive Relations

parents(jane,sally,bob). parents(john,sally,bob).
parents(sally,mary,al). parents(bob,ann,mike).
parents(mary,lee,joe).

Y is ancestor of X

ancestor(X,Y):- parents(X,Y,_).
ancestor(X,Y):- parents(X,_,Y).
ancestor(X,Y):- parents(X,Z,_),ancestor(Z,Y).
ancestor(X,Y):- parents(X,_,Z),ancestor(Z,Y).

?-ancestor(jane,X).
X= sally ;
X= bob ;
X= mary ;
X= al ;
X= lee ;
X= joe ;
X=ann ;
X=mike ;
no
Logic Programming vs Prolog

• Logic Programming: **Nondeterministic**
  – Arbitrarily choose rule to expand first
  – Arbitrarily choose subgoal to explore first
  – Results don't depend on rule and subgoal ordering

• Prolog: **Deterministic**
  – Expand first rule first
  – Explore first (leftmost) subgoal first
  – Results may depend on rule and subgoal ordering
Mineral Prolog Syntax

<rule> ::= (<head> :- <body> .) | <fact> .
<head> ::= <predicate>
<fact> ::= <predicate>
<body> ::= <predicate> { , <predicate> }
<predicate> ::= <functor> (<term> { , <term> })
<term> ::= <integer> | <atom> | <variable> | <predicate>
<query> ::= ?- <predicate>