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.**
- **Specify relations between objects**
  `larger(3,2)` `father(tom,jane)`
- **Separate logic from control:**
  - Separate the What (logic) from the How (control)
  - Programmer declares what facts and relations are true
    `father(X,jane):- male(X),parent(X,jane).`
  - System determines how to use facts to solve problems
  - State relationships and query them as in logic

Logic Programming vs Prolog

- **Computation engine: theorem-proving and recursion**
  - **Uses unification, resolution, backward chaining, backtracking**
- **Problem description is higher-level than imperative languages**

Prolog

- **As database management**
  - **Program is a database of facts**
  - **Simple queries with constants and variables (“binding”), conjunctions and disjunctions**
  - **Rules to derive additional facts**
  - **Two interpretations**
    - **Declarative: related to 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).
```

predicates

constants

Queries (Asking Questions)

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

variable

```
?-likes(al,eve).
yes
?-likes(al, pie)
no
?-likes(eve,al).
no
?-likes(person,food).
no
```

query

answer

```
?-likes(al,Who).
Who=eve
?-likes(eve,W).
W=pie  ;
W=tom  ;
W=eve  ;
no
```

answer with variable binding

force search for more answers

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).
```

same binding

```
?-likes(eve,W),likes(W,V).
W=eve,V=pie ; W=eve,V=tom ; W=eve,V=eve
?-likes(eve,W),person(W),food(V).
W=tom,V=pie ; W=tom,V=apple
?-likes(eve,V),(person(V);food(V)).
V=pie  ; V=tom ; no
?-likes(eve,W),\+likes(al,W).
W=pie ; W=tom ; no
```

or

not

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:

```
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).       a fact
female(alice).      Facts are put in a file.
male(edward).
female(victoria).
parents(edward,victoria,albert).
parents(alice,victoria,albert).
?- [family].        loads file
yes
?- male(albert).          a query
yes
?- male(alice).
no
?- parents(edward,victoria,albert).
yes
?- parents(bullwinkle,victoria,albert).
no
```

cf Clocksin and Mellish

Queen Victoria Example, cont.

- **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
```

a rule

- **Prolog program consists of facts, rules, and queries**
- **A query is a proposed fact, needing 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 which are reported (called a *substitution*)**

Horn Clauses

- **A Horn Clause is: $c \leftarrow h_1 \wedge h_2 \wedge h_3 \wedge \ldots \wedge h_n$**
  - ***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, . . . , and h_n are all true**

```
sister_of(X,Y):-
female(X),parents(X,M,F),parents(Y,M,F).
"X is the sister of Y, if X is female, X's parents are
 M and F, and Y's parents are M and F."
```

Horn Clauses

- In Prolog, a Horn clause $c \leftarrow h_1 \wedge \ldots \wedge h_n$ 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:
  - $c(X_1,\ldots,X_n)$:- $h(X_1,\ldots,X_m,Y_1,\ldots,Y_k)$.
    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

`father(X,jane) :- male(X),parents(jane,Y,X).`

- Scope of `X` is this rule
- Y is an auxiliary variable, X is a variable
- `jane` is a constant

- **Goal-oriented (declarative) semantics:**
  - `father(X,jane)` is true for those values of X which make subgoals `male(X)` and `parents(jane,Y,X)` true.
  - Recursively apply this reasoning until reach rules that are facts; called *backwards chaining*

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?

```
(1)male(albert).
(2)female(alice).
(3)male(edward).
(4)female(victoria).
(5)parents(edward,victoria,albert).
(6)parent(alice,victoria,albert).
```

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

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

```
sis(X,Y):-female(X),parents(X,M,F),
     parents(Y,M,F),\+(X==Y).
?-sis(X,Y).
```
last subgoal disallows X,Y to have same value

```
X=alice
Y=edward  ;
no
```

= means ***unifies with***

== means ***same in value***

Negation as Failure

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

```
not(X):-X,!,fail.
not(_).
```

- Which means
  - if X succeeds in first rule, then the rule is forced to fail by the last subgoal (`fail`).we cannot backtrack over the cut (!) 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.

Procedural Semantics

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

Semantics:

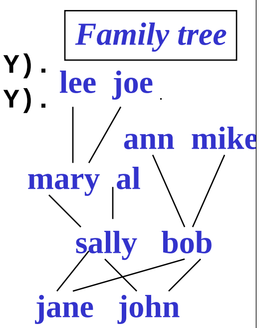
- First *find* an `X` to make `female(X)` true
- Second *find* an `M` and F to make `parents(X,M,F)` true for that `X`.
- Third *find* a `Y` to make `parents(Y,M,F)` true for those `M,F`
- This algorithm is recursive; each *find* works on a new "copy" of the facts+rules. eventually, each find must be resolved by appealing to facts.
- Process is called *backward chaining.*

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**

Minimal Prolog Syntax

<rule> ::= (<head> :- <body> .) | <fact> .

<head> ::= <predicate>

<fact> ::= <predicate>

<body> ::= <predicate> { , <predicate> }

<predicate> ::= <functor> (<term> { ,<term>})

<term> ::= <integer> | <atom> | <variable> |
<predicate>

<query> ::= ?- <predicate>