Prolog

• Logic programming (declarative)
  – Goals and subgoals
• Prolog Syntax
• Database example
  – rule order, subgoal order, argument invertibility,
    backtracking model of execution, negation by
    failure, variables
• Data structures (lists, trees)
  – Recursive Functions: append, member
  – Lazy evaluation, terms as trees, Prolog search trees
• Models of execution
  – Goal-oriented semantics
  – Procedural view

Intro to Logic Programming

• Specifies relations between objects
  larger (2,1), father (tom, jane)
• Separates control in PL from description
  of desired outcome
  father(X, jane) :- male(X), parent(X, jane).
• Computation engine: theorem proving
  and recursion
  – Higher-level PL than imperative languages
  – More accessible to non-technical people
Horn Clauses

- Conjunct of 0 or more conditions which are atomic formulae in predicate logic (constants, predicates, functions)
  \[ h_1 \land h_2 \land \ldots \land h_n \rightarrow c \]
- Means \( c \) if \( h_1, h_2, \ldots, h_n \) are all true
- Can have variables in the \( h_i \)’s or \( c \)
  \[ c( x_1, x_2, \ldots, x_m ) \text{ if } h(x_1, x_2, \ldots, x_m, y_1, \ldots, y_k) \]
  means for all objects \( x_1, x_2, \ldots, x_m, c \) holds if there are objects \( y_1, \ldots, y_k \) such that \( h \) holds.

  \[
  \text{father}(X, jane) :- \text{male}(X), \text{parents}(X, Y, jane)
  \]

Logic Programming

- Goal-oriented semantics
  - goal is true for those values of variables which make each of the subgoals true
    - father(\( X, jane \)) will be true if \( \text{male}(X) \) and \( \text{parents}(X, Y, jane) \) are true with specific values for \( X \) and \( Y \)
  - recursively apply this reasoning until reach rules that are facts.
  - called \textit{backwards chaining}
Logic Programming

• Nondeterminism
  – Choice of rule to expand subgoal by
  – Choice of subgoal to explore first

\[ \text{father}(X, \text{jane}) \ :- \ \text{male}(X), \text{parents}(X, Y, \text{jane}). \]
\[ \text{father}(X, \text{jane}) \ :- \ \text{father}(X,Y), \text{brother}(Y, \text{jane}). \]

which rule to use first? which subgoal to explore first?

– Prolog tries rules in sequential order and proves subgoals from left to right. - Deterministic!

Victoria Database Program

male(albert).
male(Edward).
female(alice).
female(victoria).
parents(Edward,victoria,albert).
parents(alice,victoria,albert).

?- male(albert).
yes
?- male(alice).
no
?- female(X).
X = alice ;
X = victoria ;
no

By responding <cr> you quit the query; <cr> you continue to find another variable binding that makes the query true.
Victoria Example

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

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

• Variable X has been unified to all possible values that make \texttt{female(X)} true.
  – Performed by pattern match search
• Variables capitalized, predicates and constants are lowercase

?-sister_of(X,Y):-
  female(X),parents(X,M,F),parents(Y,M,F).
?- sister_of(alice,Y).
  \texttt{Y = edward}
?- sister_of(alice, victoria).
\texttt{no}

• Prolog program consists of \texttt{facts}, \texttt{rules}, and \texttt{queries}
• A query is a proposed fact, needing to be proven
  – If query has no variables and is provable, answer is \texttt{yes}
  – If query has variables, the proof process causes some variables to be bound to values which are reported (called a \texttt{substitution})
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(X,Y).
X = alice
Y = edward ;
X = alice
Y = alice ;
no

Subgoal order, argument invertibility, backtracking, rule order

sis(X,Y) :- female(X), parents(X,M,F),
    parents(Y,M,F), \
=\+(X==Y).

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

\+(P) succeeds when P fails; called negation by failure

= means unifies with
== means same in value
Negation by Failure

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

if X succeeds in first rule, then the goal fails because of the last term.
if we type “;” the cut (\!) will prevent us from backtracking over it or trying the second rule so there is no way to undue the fail.
if X fails in the first rule, then the goal fails because subgoal X fails. the system tries the second rule which succeeds, since “\_” unifies with anything.

Procedural Semantics

?-\text{sister\_of}(X,Y):-
\text{female}(X),\text{parents}(X,M,F),\text{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 \textit{backward chaining}. 
**Prolog Syntax in EBNF**

```
<term>  →  <integer> | <atom> | <variable> | 
<functor>  (<term> {, <term>})

head  :-  body

<rule>  →  <predicate> (<term> {, <term>}) :-  
<term> {, <term>}. | <fact>

<fact>  →  <functor> (<term>) {, <term>}.)

<query>  →  ?- <functor>(<term>{,<term>}).
```

a proposed fact that must be proven

---

**Syntax**

- Names come from first order logic
  
  ```
  a(X,Y) :- b(c(Y)), integer(X).
  ```

  - Predicates are evaluated
  - Functors with terms are unified
  - Call this a clause or rule

Lists

\[
\begin{align*}
\text{list} & \quad \text{head} & \quad \text{tail} \\
[a, b, c] & \quad a & \quad [b, c] \\
[X, [\text{cat}], Y] & \quad X & \quad [\text{cat}, Y] \\
[a, [b, c], d] & \quad a & \quad [b, [c], d] \\
[X \mid Y] & \quad X & \quad Y \\
\end{align*}
\]

\[a\text{ list consists of a sequence of terms}\]

Unifying Lists

\[
[X, Y, Z] = [\text{john, likes, fish}]
\]

\[
X = \text{john}, \ Y = \text{likes}, \ Z = \text{fish}
\]

\[
[\text{cat}] = [X \mid Y]
\]

\[
X = \text{cat}, \ Y = []
\]

\[
[\text{the, Y} \mid Z] = [\text{[X, hare]} \mid \text{[is, here]}]
\]

\[
X = \text{the}, \ Y = \text{hare}, \ Z = \text{[is, here]}
\]

\[
[1 \mid 2] \quad \text{versus} \quad [1, 2]
\]
Lists

- Sequence of elements separated by commas, or
- `[ first element | rest_of_list ]`
  - `[ car(list ) | cdr(list)] notation`
- `[ [the | Y] | Z] = [ [X, hare] | [ is, here] ]`

XabcY [ ] XabcY

[X, abc, Y] =? [ X, abc | Y]

There is no value binding for Y, to make these two trees isomorphic.
Lists

\[ [a,b \mid Z] =? [ X \mid Y] \]
\[ X = a, Y = [b \mid Z], Z = \_ \]
look at the trees to see why this works!
\[ [ a, b, c] = [ X \mid Y] \]
\[ X = a, Y = [b,c] ; \]
no

Member_of Function

member(A, [A \mid B] ).
member(A, [B \mid C]) :- member (A, C).

*goal-oriented semantics*: can get value assignment for goal member(A,[B\mid C]) by showing truth of subgoal member(A,C) and retaining value bindings of the variables

*procedural semantics*: think of head of clause as procedure entry, terms as parameters. then body consists of calls within this procedure to do the calculation. variables bindings are like “returned values”.
Example

?- member(a,[a,b]).
yes
?- member(a,[b,c]).
no
?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
no

Invertibility of Prolog arguments
2. member(A, [B | C]) :- member (A, C).

Example

?- member(a,[b,c,X]).
X= a  ;
no
?- member(X,Y).
X = _123
Y = [ X | _124]  ;
X = _123
Y = [ _125, X | _126]  ;
X = _123
Y = [ _127, _128, X | _129]

Lazy evaluation of a priori unbounded list structure. Unbound X variableis first element, then second element, then third element, in a sequence of generated lists of increasing length.
?- member(X, [a,b,c]).
match rule 1.  member(A, [A | B]) so X = A = a, B = [b,c]  
X = a ;
match rule 2.  member(A, [B | C]) so X = A, B = a, C = [b,c]  
then evaluate subgoal member(X, [b,c])
  match rule 1.  member(A",[A" | B"])) so X=b, B" = [c]  
X = b ;
match rule 2.  member(A",[B" | C"]) so X=A", B" = b, C" = [c]  
then evaluate subgoal member(X, [c])
  match rule 1.  member(A"",[A"" | B""]) so X=A""= c, B""=[ ]  
X = c ;
match rule 2.  member(A"",[B"" | C""]) so X=A"", B""=c, C""=[ ], but member(X, [ ]) is unsatisfiable, no
**Another Search Tree**

![Search Tree Diagram]

1. member(A, [A | B]).
2. member(A, [B | C]) :- member(A, C).

**Prolog Search Trees**

- Really have built an evaluation tree for the query  `member(X, [a, b, c])`.
- Search trees provide a formalism to consider all possible computation paths
- Leaves are **success** nodes or **failures** where computation can proceed no further
  - Can have more than 1 of each type of node
- By convention, to model Prolog, leftmost subgoal is tried first
Prolog Search Trees, cont.

- Label edges with variable bindings that occur by *unification*
- There can be infinite branches in the tree, representing non-terminating computations (performed lazily by Prolog);
  - *Lazy evaluation* implies only generate a node when needed.

Another Member_of Function

Equivalent set of rules:

\[
\text{mem}(A, [A|\_] ) . \\
\text{mem}(A, [\_| C]) :- \text{mem}(A,C). \\
\]

Can examine search tree and see the variables which have been excised were auxiliary variables in the clauses.
Append Function

append ([ ], A, A).
append([A|B], C, [A|D]):- append(B, C, D).

• Build a list
  ?- append([a], [b], Y).
  Y = [a, b]

• Break a list into constituent parts
  ?- append(X, [b], [a, b]).
  X = [a]
  ?- append([a], Y, [a, b]).
  Y = [b]

More Append

?- append(X, Y, [a, b]).
X = []
Y = [a, b] ;
X = [a]
Y = [b] ;
X = [a, b]
Y = [] ;
no

append ([ ], A, A).
append([A|B], C, [A|D]):- append(B, C, D).
Still More Append

• Generating an unbounded number of lists
  
  ?- append(X, [b], Y).
  
  \[X = [];\]
  \[Y = [b];\]
  \[X = [_169];\]
  \[Y = [_169, b];\]
  \[X = [_169, _170 ];\]
  \[Y = [_169, _170, b];\]
  \[etc.\]

  \begin{verbatim}
  append ([ ], A, A).
  append([A|B], C, [A|D]):- append(B, C, D).
  \end{verbatim}

Common Beginner’s Errors

• Compile-time
  – Forget ending “.”
  – Misspelled functors
  – Need to override precedences with (..)

• Runtime
  – Infinite loops - check your recursion
  – Variables instantiated with unexpected values
  – Circular definitions
  – Giving wrong numbers of arguments to a clause
Examples Online on Paul

- `/grad/users/ryder/prolog/programs/` contains examples of Prolog programs
- `/grad/users/ryder/prolog/newtraces/` contains traces of runs of these programs
- Remember, to run these you need to copy them to your own working directory so you can write on them