Prolog

• proof process
  - generate subgoals
  - replace variables by values
    • unification: find suitable set of values so that goal can be matched to fact or head of rule
• ordering and its effect on backtracking:
  - subgoals that are hardest to satisfy should be placed as early as possible in the program
• recursion:
  - recursive rule
  - stopping condition
Prolog – Recursion

child(X, Y) :- mother(X, Y).
child(X, Y) :- father(X, Y).

descendant(X, Y) :- child(X, Y).
descendant(X, Y) :- child(X, Z),
                descendant(Z, Y).
Prolog - Negation

• negation
  - closed-world assumption: all facts about the world are included in the model
  • negation of anything not derivable is considered true

?- not typhoid(bill)

The not goal succeeds in two situations:
1. When bill has definitely not got typhoid.
2. When it cannot prove bill has typhoid.
Prolog - Negation

female(anne)
male(X) :- not female(X).

?- male(anne)
   No
   because anne is definitely not male

?- male(andrea)
   Yes
   because the system cannot prove that andrea is female
   (in Italy andrea is masculine)
Data Objects

- atoms: numbers and character strings
- structured objects
  - functor: the name
  - components

\[
\text{student}(\text{fullname}(\text{Forename}, \text{Surname}), \text{Age})
\]

\[
\text{attends}(\text{student}(\text{fullname}(\text{mary, smith}), 20), \text{stirling})).
\]

\[
\text{attends}(\text{student}(\text{fullname}(\text{joe, brown}), 25), \text{stirling})).
\]

\[
\text{maturestudent}(\text{Surname}) :-
\text{attends}(\text{student}(\text{fullname}(\text{Forename,Surname}), \text{Age}), \text{University}), \text{Age} > 23.
\]

\[
?- \text{maturestudent}(X).
\]

\[
X = \text{brown}
\]
Data Objects

• lists: . and []
  - head: the first element
  - tail: the rest of the list
  - short form for list with head $X$, tail $Y$: $[X|Y]$
  - lists are represented as binary trees
• selector relations: to access components of a structure
  - hide representation of structure

forename(student(fullname(Forename, Surname), Age), Forename).

forename(student(fullname(Forename, _), _), Forename).
Efficiency

• backtracking:
  - depth-first search
  - subgoal evaluation from left to right
  - ordering of subgoals is crucial
• cut: !
  - stops backtracking
  - useful for mutually exclusive clauses

\[ R \leftarrow G_1, G_2, !, G_3 \]
\[ R \leftarrow G_4 \]
Efficiency - Example

\[ f(X, Y) \leftarrow X < 0, Y < 0. \]
\[ f(X, Y) \leftarrow X = 0, Y = 0. \]
\[ f(X, Y) \leftarrow X > 0, Y > 0. \]

?- f(-1, 1).

?- -1 < 0, 1 < 0.
?- -1 = 0, 1 = 0.
?- -1 > 0, 1 > 0.

?- 1 < 0.
Efficiency – Example (cont.)

\[ f(X, Y) :- X < 0, \neg, Y < 0. \]
\[ f(X, Y) :- X = 0, \neg, Y = 0. \]
\[ f(X, Y) :- X > 0, Y > 0. \]

?- \ f(-1, 1).

?- \ -1 < 0, \neg, 1 < 0.

?- \ 1 < 0.
Efficiency

• cut: !
  - programs without cuts have same declarative and procedural meaning
    • slow but correct
  - programs with cuts may not have same declarative and procedural meaning
    • green cuts, red cuts

previous example
Efficiency – Red Cuts

male(george).
mother(jane, anne).
mother(jane, george).
father(john, anne).
father(john, george).

\[
\text{son}(X, Y) \leftarrow \text{father}(Y, X), !, \text{male}(X) \\
\text{son}(X, Y) \leftarrow \text{mother}(Y, X), !, \text{male}(X)
\]

\[
\begin{align*}
\text{son}(\text{george}, \text{jane}) & \Rightarrow \text{yes} \\
\text{son}(\text{anne}, \text{john}) & \Rightarrow \text{no} \\
\text{but}\quad \text{son}(X, \text{jane}) & \Rightarrow \text{no instead of } X = \text{george}.
\end{align*}
\]
Prolog - Example

Symbolic differentiation:

\[ 4 \times \text{power}(x, 3) \quad \text{power}(x, 3) \]

\[ 12 \times \text{power}(x, 2) \quad 3 \times \text{power}(x, 2) \]

To differentiate an expression \( Y \) with respect to \( X \) giving the result \( E \), the heads of the rules will have the form \( \text{derive}(Y, X, E) \).
Prolog - Example

First attempt:

derive(A * F, X, A * C) :- integer(A), derive(F, X, C).
derive(power(X, N), X, N * power(X, N - 1)) :- integer(N).

?- derive(4 * power(x, 3), x, E).
(A ← 4, F ← power(x, 3), X ← x, E ← A * C)
⇒ integer(4), derive(power(x, 3), x, C).
⇒ derive(power(x, 3), C).
(X ← x, N ← 3, C ← N * power(X, N - 1))
⇒ integer(3).

result: E = 4 * (3 * power(x, 3 – 1))
Example (cont.)

derive(A * F, X, G * B) :-
    integer(A), derive(F, X, D * B), G is A * D.
derive(power(X, N), X, N * power(X, P)) :-
    integer(N), P is N - 1.

?- derive(4 * power(x, 3), x, E).
   (A ← 4, F ← power(x, 3), X ← x, E ← G * B)
⇒ integer(4), derive(power(x, 3), x, D * B), G is 4 * D.
⇒ derive(power(x, 3), x, D * B), G is 4 * D.
       (X ← x, N ← 3, D ← 3, B ← power(x, P))
⇒ integer(3), P is 3 - 1, G is 4 * 3.
⇒ P is 2, G is 12.

result: E = 12 * power(x, 2)
ELIZA

ELIZA simulates a therapist by generating suitable questions according to the input.


Example:

I am unhappy.
>> How long have you been unhappy?
   Six month. Can you help me?
>> What make you think I help you?
   You remind me of my father and brother.
>> Can you tell me more about father.
   I like teasing father.
>> Does anyone else in your family like teasing father.
   No, only me.
>> Please go on.
ELIZA (cont.)

We will implement the basic algorithm without considering I/O.


eliza is a predicate on pairs of lists of atoms representing the input resp. the output.

eliza([bye], [goodbye,i,hope,i,have,helped,you]).
eliza(Input, Output) :-
    pattern(Stimulus, Answer),
    match(Stimulus, Table, Input),
    match(Answer, Table, Output), !
ELIZA (cont.)

pattern([i, am, 1], [how, long, have, you, been, 1, ?]).
pattern([1, you, 2, me], [what, makes, you, think, i, 2, you, ?]).
pattern([i, like, 1], [does, anyone, else, in, your family, like, 1, ?]).
pattern([i, feel, 1], [do, you, often, feel, this, way, ?]).
pattern([1, X, 2], [can, you, tell, me, more, about, X, ?])
    :- important(X).
pattern([1], [please, go, on]).

important(father).
important(mother).
important(son).
important(daughter).
important(sister).
important(brother).
ELIZA (cont.)

search returns a value which is stored for a key in a dictionary. The dictionary is represented by a list of pairs (Key, Value).

search(Key, [(Key, Value)|Dictionary], Value).
search(Key1, [(Key2, Value1)|Dictionary], Value2) :-
    search(Key1, Dictionary, Value2).
ELIZA (cont.)

match([N|Pattern], Table, Target) :-
    integer(N),
    search(N, Table, LeftTarget),
    append(LeftTarget, RightTarget, Target),
    match(Pattern, Table, RightTarget).

match([Word|Pattern], Table, [Word|Target]) :-
    atom(Word),
    match(Pattern, Table, Target).

match([], Table, []).