A Taste of Prolog

Aja Hammerly
Basics

• I Like Prolog
• But, I'm not an expert
• This is just an introduction
What Is Prolog

- A logic programing language
- A declarative programming language
- A weird programming language
Uses

- Natural Language Processing
- Grammars
- Theorem Proving
- Expert Systems and other AI
Why Learn Prolog

- Expand your toolbox
- New perspective
- Become a polyglot
Prolog - Weirdness

• “What”, not “How”.

• Programs are expressed as:
  • Facts
  • Rules
“A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have the desired meaning.”

- The Art of Prolog
Seattle.rb Pairing
Facts

editor(zenspider, emacs).
editor(drbrain, vim).
editor(phiggins, vim).
editor(tenderlove, vim).
Questions

editor(zenspider, emacs).
editor(drbrain, vim).

?- editor(zenspider, emacs).
yes

?- editor(zenspider, vim).
no
Questions

editor(zenspider, emacs).
editor(drbrain, vim).

?- editor(drbrain, Editor).
Editor = vim
Questions

editor(zenspider, emacs).
editor(drbrain, vim).

?- editor(Person, Editor).
Person = zenspider
Editor = emacs
Questions

editor(zenspider, emacs).
editor(drbrain, vim).
editor(tenderlove, vim).

?- editor(Person1, vim),
editor(Person2, vim),
   Person1 \== Person2.
Person1 = drbrain
Person2 = tenderlove
Questions

editor(zenspider, emacs).
editor(drbrain, vim).
editor(tenderlove, vim).

?- editor(Person1, Editor),
   editor(Person2, Editor),
   Person1 \== Person2.
   Editor = vim
   Person1 = drbrain
   Person2 = tenderlove
Rules

\[
pair(Person1, Person2) :-
  editor(Person1, Editor),
  editor(Person2, Editor),
  Person1 \leq Person2.
\]

?- pair(Person1, Person2).
Person1 = drbrain
Person2 = tenderlove
Questions & Rules

editor(zenspider, emacs).
editor(drbrain, vim).
editor(tenderlove, vim).

?- pair(drbrain, Person2).
Person1 = tenderlove
Questions

?- pair(Person1, Person2).

Person1 = drbrain
Person2 = tenderlove ;

Person1 = drbrain
Person2 = phiggins ;

Person1 = tenderlove
Person2 = drbrain ?
Rules

pair(Person1, Person2) :-
    editor(Person1, Editor),
    editor(Person2, Editor),
    Person1 @> Person2.

?- pair(Person1, Person2).
Person1 = tenderlove
Person2 = drbrain
Questions

?- pair(Person1, Person2).

Person1 = tenderlove
Person2 = drbrain ? ;

Person1 = tenderlove
Person2 = phiggins ? ;

Person1 = phiggins
Person2 = drbrain ?
Facts

keyboard(zenspider, dvorak).
keyboard(drbrain, dvorak).
keyboard(tenderlove, qwerty).
keyboard(phiggins, qwerty).
Questions

keyboard(zenspider, dvorak).
keyboard(drbrain, dvorak).

?- keyboard(drbrain, Keyboard).
Keyboard = dvorak
Rules

\[
pair(\text{Person1}, \text{Person2}) :-
  \text{keyboard}(\text{Person1}, \text{Keyboard}),
  \text{keyboard}(\text{Person2}, \text{Keyboard}),
  \text{Person1} @> \text{Person2}.
\]

?- pair(\text{Person1}, \text{Person2}).
\text{Person1} = \text{zenspider}
\text{Person2} = \text{drbrain}
Two Rules

pair(P1, P2) :-
    editor(  P1, Editor),
    editor(  P2, Editor),
    P1 @>  P2.

pair(P1, P2) :-
    keyboard(P1, Keyboard),
    keyboard(P2, Keyboard),
    P1 @>  P2.

Thursday, August 2, 2, 12
Questions

?- pair(X, Y).

X = tenderlove, Y = drbrain
X = tenderlove, Y = phiggins
X = phiggins, Y = drbrain
X = zenspider, Y = drbrain
X = tenderlove, Y = phiggins
super_pair(Person1, Person2) :-
    editor(Person1, Editor),
    editor(Person2, Editor),
    keyboard(Person1, Keyboard),
    keyboard(Person2, Keyboard),
    Person1 @> Person2.
Questions

define (editor, keyboard, super_pair).

editor(phiggins, vim).
editor(tenderlove, vim).
keyboard(tenderlove, qwerty).
keyboard(phiggins, qwerty).

?- super_pair(Person1, Person2).

Person1 = tenderlove
Person2 = phiggins
Pattern Matching

• In prolog pattern matching is used to pass arguments.
• For example:
  • human(X) will match human(bill)
• Pattern matching with variables is called unification
List Basics
Examples

- []
- [1, 2, 3]
- [apples, bananas]
- [1, lemon]
- [[1, lemon], [1, lime], [2, coconuts]]
Heads and Tails

• \([1, 2, 3]\)
  • 1 is the head
  • \([2, 3]\) is the tail
• \([H \mid T]\) (read: "H bar T")
• \([H \mid T]\) matches with \([1, 2, 3]\) as \([1\mid[2,3]]\)
Don't Care

• ‘_’ means I don't care
• [1, _, 3] could be
  • [1, 2, 3] or
  • [1, pi, 3] or
  • [1, [apple, pie], 3]
• 2 don't cares can refer to different values
def member(x, ary)
    return false if ary == []
    return true  if ary[0] == x
    member(x, ary[1..-1])
end
def member(x, ary)
    return false if ary == []
    return true  if ary[0] == x
    member(x, ary[1..-1])
end

member(H, [H | _]).
member(X, [_ | T]):-
    member(X, T).
?- member(2, [1, 2, 3]).
true

?- member(6, [1, 2, 3]).
no
?- member(X, [1, 2, 3]).

X = 1 ? a

X = 2

X = 3
Variables Anywhere

?- member(6, X).

X = [6|_] ? ;
X = [_,6|_] ? ;
X = [_,_,6|_] ?
def length(ary)
    return 0 if ary == []
    return length(ary[1..-1]) + 1
end
def length(ary)
    return 0 if ary == []
    return length(ary[1..-1]) + 1
end

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

Length
?- length([a, b, c, d], 4).
yes
?- length([1, 2, 3], X).
X = 3
?- length(X, 2).

X = [_,_]
Circuits
\text{In} & \text{Out} \\
\text{inv}(0, 1) & \text{.} \\
\text{inv}(1, 0) & \text{.} \\

\text{In} & \text{Out} \\
\text{In} & - & \text{Out}
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Out</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
xor(0, 0, 0).
xor(0, 1, 1).
xor(1, 0, 1).
xor(1, 1, 0).
\begin{align*}
\text{A} & \quad \text{B} & \quad \text{Out} \\
nand(0, 0, 1). \\
nand(0, 1, 1). \\
nand(1, 0, 1). \\
nand(1, 1, 0).
\end{align*}
half_adder(A, B, C, S) :-
    xor(A, B, S),
    and(A, B, C).
?- half_adder(A, B, C, S).

A = 0, B = 0, C = 0, S = 0
A = 0, B = 1, C = 0, S = 1
A = 1, B = 0, C = 0, S = 1
A = 1, B = 1, C = 1, S = 0
full_adder(A, B, Cin, Cout, S) :-
    half_adder(A, B, C1, S1),
    half_adder(Cin, S1, C2, S),
    or(C1, C2, Cout).
?- full_adder(A, B, 1, Cout, 1).

A = 0, B = 0, Cout = 0
A = 1, B = 1, Cout = 1

?- full_adder(A, B, Cin, 1, S).

A = 0, B = 1, Cin = 1, S = 0
A = 1, B = 0, Cin = 1, S = 0
A = 1, B = 1, Cin = 0, S = 0
A = 1, B = 1, Cin = 1, S = 1
mystery(A, B, D) :-
nand(A, B, T1),
nand(A, T1, T2),
nand(B, T1, T3),
nand(T2, T3, D).

\[\text{Diagram: NAND gate connections}\]
?- mystery(A, B, D).

A = 0, B = 0, D = 0

A = 0, B = 1, D = 1

A = 1, B = 0, D = 1

A = 1, B = 1, D = 0
Logic Puzzles
1. Adam does not live on the top floor.
2. Bill does not live on the bottom floor.
3. Cora does not live on either the top or the bottom floor.
4. Dale lives on a higher floor than does Bill.
5. Erin does not live on a floor adjacent to Cora's.
6. Cora does not live on a floor adjacent to Bill's.
‘Data Structure’

- A list of the people, ordered by floor
- [Top, Floor4, Floor3, Floor2, Bottom]
- [adam, bill, cora, dale, erin]
Adam does not live on the top floor.

adam \(\equiv\) Top,
Bill does not live on the bottom floor.

\[ \text{bill} \equiv \text{Bottom}, \]
Cora does not live on either the top or the bottom floor.

cora \!=\! Top,
cora \!=\! Bottom,
Dale lives on a higher floor than does Bill.

higher(dale, bill, L),
higher(X, Y, [X | T]) :- member(Y, T).

higher(X, Y, [_ | T]) :- higher(X, Y, T).
Erin does not live on a floor adjacent to Cora's.

not_adjacent(erin, cora, L),
not_adjacent
not_adjacent(X, Y, [X, Z | T]) :-
    Z \== Y,
    member(Y, T).
not_adjacent

not_adjacent(X, Y, [X, Z | T]) :-
    Z \== Y,
    member(Y, T).

not_adjacent(X, Y, [Y, Z | T]) :-
    Z \== X,
    member(X, T).
not_adjacent

not_adjacent(X, Y, [X, Z | T]) :-
   Z \== Y,
   member(Y, T).

not_adjacent(X, Y, [Y, Z | T]) :-
   Z \== X,
   member(X, T).

not_adjacent(X, Y, [_ | T]) :-
   not_adjacent(X, Y, T).
Cora does not live on a floor adjacent to Bill's.

not_adjacent(cora, bill, L),
permutation(L, [adam, bill, cora, dale, erin]).
Puzzle

puzzle(L) :-
    L = [Top, F4, F3, F2, Bottom],
puzzle(L) :-
    permutation(L, [adam, bill, cora, dale, erin]),
    L = [Top, Floor4, Floor3, Floor2, Bottom],
    adam \== Top,
    bill \== Bottom,
    cora \== Top,
    cora \== Bottom,
    higher(dale, bill, L),
    not_adjacent(erin, cora, L),
    not_adjacent(cora, bill, L).
Running

?- puzzle([A, B, C, D, E]).

A = dale
B = cora
C = adam
D = bill
E = erin ? ;

no
Learn More
Books

• Sterling, Leon & Shapiro, Ehud. *The Art of Prolog*

• Clocksin, William F. *Clause and Effect: Prolog Programming for the Working Programmer*

• Bratko, Ivan. *Prolog Programming for Artificial Intelligence*

• Tate, Bruce A. *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*
Thank You