Announcements

- HW1 is graded
  - I’ll release your grades in HW Server after class
  - Papers available for pickup in front of my office, Lally 314

- Quiz 1&2 graded too
  - Grades will be available soon
  - We’ll go over quiz problems before tests

---

Announcements

- HW3 (Prolog) is due February 29th
  - I’ll have HW3 set in the HW Server today

- Get started with SWI Prolog
- Try the simple examples from class
- Read Chapter 11 in Scott’s book
- Ask questions!

---

Last Class

- Logic Programming
- Logic Programming Concepts

- Prolog
  - Language constructs: facts, rules and queries
  - Prolog concepts: search tree, rule ordering, unification, backtracking, backward chaining

---

Today’s Lecture Outline

- Prolog
  - Lists
  - Arithmetic
  - Imperative Control Flow

---

Logic Programming and Prolog

Keep reading: Scott, Chapter 11.2.1-6

---

Lists

<table>
<thead>
<tr>
<th>list</th>
<th>head</th>
<th>tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a, b, c]</td>
<td>a</td>
<td>[b, c]</td>
</tr>
<tr>
<td>[X, [cat], Y]</td>
<td>X</td>
<td>[[cat], Y]</td>
</tr>
<tr>
<td>[a, [b, c], d]</td>
<td>a</td>
<td>[[b, c], d]</td>
</tr>
<tr>
<td>[X</td>
<td>Y]</td>
<td>X</td>
</tr>
</tbody>
</table>

---
Lists: Unification

- \([ H_1 \mid T_1 ] = [ H_2 \mid T_2 ]\)
  - Head \(H_1\) unifies with \(H_2\), possibly recursively
  - Tail \(T_1\) unifies with \(T_2\), possibly recursively

- E.g., \([ \text{a} \mid [\text{b}, \text{c}] ] = [ \text{X} \mid \text{Y} ]\)
  - \(X = \text{a}\),
  - \(Y = [\text{b}, \text{c}]\).

- NOTE: In Prolog, = denotes unification, not assignment!

Improper and Proper Lists

\[[1 \mid 2]\] versus \[[1, 2]\]


Question. Can we unify these lists?

\([\text{abc}, \text{Y}] =? [\text{abc} \mid \text{Y}]\)

What happens here? Can we unify these lists?

Member_of “Procedure”

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

logical semantics: For every value assignment of \(A, B\) and \(C\), we have member(A, [B | C]) if member(A, C):

procedural semantics: Head of clause is procedure entry. Procedure body consists of calls within this procedure.

Example

?- member(a, [b, c, X]).
1. member(A, [A | B]).
2. member(A, [B | C]) :- member(A, C).

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

Lazy evaluation of unbounded list structure. \(a\) as first element, then \(a\) as second element, third element, etc.
Prolog Search Tree (simplified)

Question

Give all answers to the following query:
?- member(a, [b,a,X]).

Another Search Tree

“Procedural” Interpretation

Append “Procedure”
More Append

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

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

Spring 16 CSCI 4430, A Milanova

More Append

Generating an unbounded number of lists
?- append(X, [b], Y).

Be careful when using append with 2 unbounded arguments!!!

Spring 16 CSCI 4430, A Milanova

More Append

Question

- What does this “procedure” do:
  p([],[]).
p([A|B],[[A]|Rest]) :- p(B,Rest).

- Puts brackets around each element in the list
  ?- p([a,b,c],Y).
  Y = [ [a],[b],[c] ]
- It can also “flatten” a list:
  ?- p(X,[[a],[b],[c]]).
  X = [ a,b,c ]

Spring 16 CSCI 4430, A Milanova

Common Structure

- “Processing” a list:
  proc([],[]).
  proc([H|T],[H1|T1]) :- f(H,H1), proc(T,T1).

- Base case: we have reached the end of list.
  In our case, the result for [ ] is [ ].
- Recursive case: result is [H1|T1]. H1 was obtained by calling f(H,H1) --- processes element H into result H1. T1 is the result of recursive call of proc on T.

Spring 16 CSCI 4430, A Milanova

Lecture Outline

- Prolog
  - Lists
  - Arithmetic
  - Imperative Control Flow

Spring 16 CSCI 4430, A Milanova
Arithmetic

- Prolog has all arithmetic operators
- Built-in predicate `is`
  - `is(X, 1+3)` or more commonly we write
  - `X is 1+3`
  - `is` forces evaluation of `1+3`:
    - `- X is 1+3
      X = 4`
- `=` is unification not assignment!
  - `- X = 4-1
    X = 4-1 % unifies X with 4-1!!`

Exercise

- Write `sum`, which takes a list of integers and computes the sum of the integers. E.g.,
  - `sum([1,2,3],R).`
  - `?- R = 6.`
- How about if the integers are arbitrarily nested? E.g.,
  - `sum([[1],[[2]],3]],R).`
  - `?- R = 6.`

Exercise

- Write `plus10`, which takes a list of integers and computes another list, where all integers are shifted +10. E.g.,
  - `plus10([[1,2,3],R).`
  - `?- R = [11,12,13].`
- Write `len`, which takes a list and computes the length of the list. E.g.,
  - `len([1,[2],[3]],R).`
  - `?- R = 3.`

Exercise

- Write `atoms`, which takes a list and computes the number of atoms in the list. E.g.,
  - `atoms([a,[b,[c]]]],R).`
  - `?- R = 3.`
- Hint: built-in predicate `atom(X)` yields true if `X` is an atom (i.e., symbolic constant such as `x, abc, tom`).

Arithmetic: Common Pitfalls

- `is` is not invertible! That is, arguments on the right cannot be unbound!
  - `3 is 3 - X.
    ERROR: is/2: Arguments are not sufficiently instantiated`
- This doesn’t work either:
  - `- X is 4, X = X+1.
    false.
    Why? What is going on here?

Lecture Outline

- Prolog
- Lists
- Arithmetic
- Imperative Control Flow
Imperative Control Flow

- Programmer has **explicit control** on backtracking process

**cut (!)**

- As a goal it succeeds, but with a **side effect**:
  - Commits interpreter to all bindings made since unifying parent goal with left-hand side of current rule

Cut (!) Example

- `rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).`

?- `snowy(C).`

Cut (!) Example 2

- `rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
snowy(troy).

?- `snowy(C).`

Cut (!) Example 3

- `rainy(seattle) :- !.
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
snowy(troy).`

?- `snowy(C).`
Cut (!) Example 3

\[
\begin{align*}
\text{rainy}(\text{seattle}) & : - !. \\
\text{rainy}(\text{rochester}) & . \\
\text{cold}(\text{rochester}) & . \\
\text{snowy}(X) & : - !, \text{rainy}(X), \text{cold}(X). \\
\text{snowy}(\text{troy}) & .
\end{align*}
\]

How about goal \( \neg \text{snowy}(\text{rochester}) \)?

Cut (!) Example 4

\[
\begin{align*}
\text{rainy}(\text{seattle}). \\
\text{rainy}(\text{rochester}). \\
\text{cold}(\text{rochester}). \\
\text{snowy}(X) & : - !, \text{rainy}(X), \text{cold}(X). \\
\text{snowy}(\text{troy}) & .
\end{align*}
\]

\( \neg \text{snowy}(C) \).

Cut (!) Example 5

\[
\begin{align*}
\text{rainy}(\text{seattle}). \\
\text{rainy}(\text{rochester}). \\
\text{cold}(\text{rochester}). \\
\text{snowy}(X) & : \neg \text{rainy}(X), \text{cold}(X), !. \\
\text{snowy}(C) & .
\end{align*}
\]

\( \neg \text{snowy}(C) \).

Negation by Failure

\[
\begin{align*}
\text{takes}(\text{jane}, \text{his}). \\
\text{takes}(\text{jane}, \text{cs}). \\
\text{takes}(\text{ajit}, \text{art}). \\
\text{takes}(\text{ajit}, \text{cs}). \\
\text{classmates}(X, Y) & : \neg \text{takes}(X, Z), \text{takes}(Y, Z). \\
\neg \text{classmates}(\text{jane}, \text{C}). \\
\text{classmates}(X, Y) & : \neg \text{takes}(X, Z), \text{takes}(Y, Z), \neg(X=\text{Y}).
\end{align*}
\]
Negation by Failure: \texttt{not}(X), \texttt{\textbackslash +}(X)

- \texttt{not}(X) succeeds when X fails
- Called negation by failure, defined:
  \texttt{not}(X):- X, !, fail.
  \texttt{not}(\_).
- \texttt{classmates}(X,Y):- takes(X,Z),
  takes(Y,Z), \texttt{\textbackslash +}(X=Y).

- Not the same as logical negation $\neg X$!

Example

- \texttt{p(X) :- q(X), not(r(X)).}
- \texttt{r(X) :- w(X), not(s(X)).}
- \texttt{q(a). q(b). q(c).}
- \texttt{s(a). s(c).}
- \texttt{w(a). w(b).}

Evaluate:
- ?- p(a).
- ?- p(b).
- ?- p(c).

A Harder Exercise

- Remember the grammar...
  1. \texttt{S \rightarrow a S b S}
  2. \texttt{S \rightarrow b S a S}
  3. \texttt{S \rightarrow \varepsilon}
- Write a parser in Prolog which given a string, computes all leftmost derivations:
  \texttt{?- parse([a,b,a,b],R).}
  \texttt{R = [1, 3, 1, 3, 3] ; // seq. of productions}
  \texttt{R = [1, 2, 3, 3, 3] ; // different seq}
  \texttt{false. // no more derivations}
- Hint: use \texttt{append} to break list into constituent parts