Today

- Quiz
- Lists in Prolog

Assignments

- HW-8, R-8 due
- HW-9, R-9 due Thurs
Lists in Prolog

Lists are special structures

- \([1,2,3,4]\)
- \([a,b,c,d]\)
- \(['a',a,5,r(x,y)]\)

- Lists are “heterogeneous” (can contain values of different “types”) in prolog

The \([]\)'s are just syntactic sugar

?- \([1,2] = .(1,(2,[])).\).
true.

- Note that \(=\) means “unifies with” here ...
- The \(.\) notation is similar to \(1:2:[]\) (cons) in Haskell

Head and tail

Special syntax for accessing the head and tail of a list

- \([H \mid T]\)

- \(H\) is the head and \(T\) is the tail

- \([1\mid[2,3]] = [1,2,3]\)
- \([1\mid[2\mid[3]]] = [1,2,3]\)
- \([1,2\mid[3]] = [1,2,3]\)
- \([1,2,3\mid[]] = [1,2,3]\)

- Like Haskell’s \(\(:\)\), the \((\mid)\) operator is used in pattern matching
List predicates: Recursion Strikes Back

How would we write a list member predicate?

In Haskell:

\[
\begin{align*}
    \text{member } & \_ \text{ } [] = \text{False} \\
    \text{member } x \text{ } (y:ys) = \\
        \text{if } x == y \text{ then True else member } x \text{ } ys
\end{align*}
\]

In Prolog:

\[
\begin{align*}
    \text{member}(X,[X|\_]). \\
    \text{member}(X,[\_|T]) :- \text{member}(X,T).
\end{align*}
\]

This example works like pattern matching in Haskell ...

- try first rule ... if it doesn’t succeed, try next candidate
- unfold the rule
- and so on

Note that we can “reuse” X in the first pattern!

Q: What is the proof tree for \text{member}(2,[1,2])?

Q: What is the proof tree for \text{member}(2,[1,3])?

Note unlike Haskell we don't need an explicit False case
How would we write a `last` predicate?

```prolog
last(X, [X]).
last(X, [_|T]) :- last(X, T).
```

Q: What is the proof tree for `last(2, [1, 2])`?

And more examples of list processing ...

- consider a prefix predicate
  ```prolog
  prefix([1,2], [1,2,3,4])
  ```

- we can write this w/out reusing variables:
  ```prolog
  prefix([], _).
prefix([X|T1], [Y|T2]) :- X = Y, prefix(T1, T2).
  ```

- or using shared variables:
  ```prolog
  prefix([], _).
prefix([X|T1], [X|T2]) :- prefix(T1, T2).
  ```
Another example:

- the append predicate
  
  \[
  \text{append}([1,2],[3,4],[1,2,3,4])
  \]

- and one way to write it:
  
  \[
  \begin{align*}
  \text{append}([],L,L). \\
  \text{append}([H|T],L,[H|NewT]) & : \text{append}(T,L,NewT).
  \end{align*}
  \]

- note that we are “constructing” the list in the head!
  
  if \(T\) append \(L\) is \(\text{NewT}\), then \([H|T]\) append \(L\) is \([H|\text{NewT}]\)
We can use append to define a reverse predicate

\[
\begin{align*}
\text{reverse}([], []). \\
\text{reverse}([H | T], L) & \leftarrow \text{reverse}(T, R), \text{append}(R, [H], L).
\end{align*}
\]

Q: What is the proof tree for reverse([1,2], A)?

Q: What happens if the goal is reverse(X, [1,2])?

- Hint: Draw the proof tree ...

- The “signature” for reverse is: reverse(+List1, ?List2)
- That is, List1 must be an input
- List2 can be an input or output
Operators

Prolog supports standard arithmetic operators

- But beware the definition of = (unifies)

    ?- X = 2 + 3.
    X = 2+3

    ?- X = 2 * 3 + 4.
    X = 2*3+4

- These are not being evaluated!
  - really, we are unifying X with relations (structures)
  - e.g., we end up with X = +(2,3) in the first case

- Use “is” to evaluate expressions ...

    ?- X is 2 + 3.
    X = 5

    ?- X is 2 * 3 + 4.
    X = 10

- Real and integer division

    ?- X is 7/2.  % real division
    X = 3.5

    ?- X is 7//2. % integer division
    X = 3

    ?- X is 7 mod 2.
    X = 1
Using Operators

Calculating the length of a list

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

Q: What is the proof tree for \(\text{length}([1,2,3], N)\)?

Additional comparison operators

?- 3 < 4.
   true

?- 3 =< 4.
   true.

?- 3 > 4.
   false.

?- 4 >= 4.
   true.

?- X == 4.  \% value comparison, not unification
   false.

?- X \== Y.  \% same idea, but not-equal
   true.

?- X > 4.
   ERROR: >/2: Args not sufficiently instantiated

All operands must be instantiated to evaluate arithmetic/comparison operators
Watch out for backtracking ...

A relation to zero-out an unwanted value ...

• e.g., zero_out(3, [1, 3, 2, 3, 5], Xs) returns Xs = [1, 0, 2, 0, 5]

zero_out(_, [], []). % base case
zero_out(X, [X|Ys], [0|Zs]) :- % found value
     zero_out(X, Ys, Zs).
zero_out(X, [Y|Ys], [Y|Zs]) :- % didn’t find val
     X \== Y, zero_out(X, Ys, Zs).

Q: What happens if we leave out the X \== Y relation?

... HINT: Create a proof tree to see!