CS 230
Programming Languages

12 / 06 / 2022

Instructor: Michael Eckmann

Today's Topics

- Questions? / Comments?
- Chapter 16
  - Prolog
  - Example programs in bprolog

Prolog

- List Structures are another basic data structure in Prolog besides atomic propositions.
- A List is a sequence of any number of elements
- Elements can be atoms, atomic propositions, or other terms (including other lists)

```
[apple, prune, grape, kumquat]
[]      (empty list)
[X | Y]     (head X and tail Y)
```

Prolog

- `[X | Y]` (head X and tail Y)
- Head X is like Scheme's car of a list
- Tail Y is like Scheme's cdr of a list
- `|` can be used to dismantle a list into head and tail in query mode
- `|` can be used to construct a list from a head and a tail

Prolog

- **`[X | Y]` (head X and tail Y)**
  - **my_list([apple, prune, grape]).**
  - **my_list([apricot, peach, pear, lime]).**
- **These signify that the lists are new elements of the relation my_list. Just like parent(jane). stated that jane is an element of the relation parent.**
- **my_list is not a variable --- it is a relation.**
- **Example query:**
  - **my_list([My_head | My_tail])**
  - **Results in my_head instantiated with apple, and my_tail instantiated with [prune, grape]**

Prolog

- **`[X | Y]` (head X and tail Y)**
  - **my_list([apple, prune, grape]).**
  - **my_list([apricot, peach, pear, lime]).**
  - **my_list([My_head | My_tail])**
- **If this query was part of a goal and it backtracked, the next value for my_head would be: apricot and the next value for my_tail would be: [peach, pear, lime]**

Prolog

- **`[X | Y]` (head X and tail Y)**
  - **also used for constructing lists**
- **Construction example:**
  - **[apricot, peach | [pear, lime]]**

Prolog

- **Append**
  - **The first two parameters to append are appended together to produce the result which parameter 3 gets instantiated with.**

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3])
    :- append (List_1, List_2, List_3).
```

- **Order matters! The termination step (the headless clause) is first (just like in Scheme.) The recursion step is second. Prolog will try to match the propositions in order because it uses depth-first search.**
- **How do we read these propositions?**

Prolog

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3])
    :- append (List_1, List_2, List_3).
```

- **So it says, if the first parameter is the empty list, the resulting list (parameter 3) is instantiated to the value of the second parameter.**
- **Otherwise, list3 is list1 and list2 appended together if the head of the first parameter is the head of parameter 3 (the result) and the tail of the 3rd parameter is the tail of the 1st parameter appended to the second parameter.**
- **Or in other words: appending `[Head | List_1]` to `List_2` results in `[Head | List_3]` only if, `List_3` is `List_1` appended to `List_2`.**
- **It'll become clearer with an example.**

Prolog

- **Similar to scheme's:**

  (cons (car list1) (append (cdr list1) list2))

```
append([], List, List).
   append([Head | List_1], List_2, [Head | List_3])
       :- append (List_1, List_2, List_3).
```

append([bob, jo], [jake, darcie], Family).

Let's trace through on the board how Prolog will instantiate Family.

Prolog

`Call: append`([bob, jo], [jake, darcie], Family).

Since [bob, jo] is not the empty list it skips to the second proposition

And then `Calls`: **append([jo], [jake, darcie], Family),** because the Head of first is removed to make List_1.

And then `Calls`: **append([], [jake, darcie], Family),** because the Head of first is removed to make List_1.

Which exits with Family becoming [jake, darcie].

The second call can then exit with jo as the Head tacked on to the beginning of Family to become: [jo, jake, darcie]

The first call can then exit with bob as the Head tacked on to the beginning of Family to become: [bob, jo, jake, darcie]

Prolog

- **As stated earlier, Prolog uses depth-first search and left-to-right evaluation. So, resolution problems can occur with recursion that is not tail recursion.**
- **ORDER OF PROPOSITIONS AND SUBGOALS MATTER**
- **Example:**

  ancestor(X, X).

  ancestor(X, Y) :- ancestor(Z, Y), parent(X, Z).

- **Problem because second proposition has 1st subgoal same as proposition (left-recursion). Suppose it is given the query ancestor(mike, jane). It goes to ancestor(Z, jane) and instantiates Z with a value. Then it does the proposition again to determine if it is true ...**
- **Just like the problem a recursive descent parser has (remember?)**
- **How to solve this?**

Prolog

- Note:

  append and member already exist in the prolog language, so we don't need to implement them, except for educational purposes

Prolog

- Cut (!) to prevent backtracking.
- Recall, that prolog backtracks when a goal (or subgoal) fails.

- Let's say we have something like this on the right hand side of a proposition:
  - **member(X, Lis), do_something(X, Lis, Y).**
- If member is satisfied and then do_something is attempted and fails, backtracking will occur and cause member to be reattempted. Because of member's implementation, it will pick up where it left off and possibly not find another match. To make sure it doesn't backtrack, we can use the Cut.

  member(Element, [Element | _]) :- !.

  member(Element, [_ | List]) :- member(Element, List).
- Now, Prolog cannot backtrack to reevaluate member once Element is found in the list.

Prolog

- **Negation problem**

  parent(bill, jake).

  parent(bill, shelly).

  sibling(X, Y) :- parent(M, X), parent(M, Y).

- goal

  sibling(X, Y).

results in:

- X = jake
- Y = jake
- Because it goes through the facts first with parent(M, X) and instantiates them to bill and jake. Then parent(M,Y) starts at the beginning of the facts and tries to find a fact with bill as parent. It finds bill and jake again.

Prolog

- **Negation problem solution**

  parent(bill, jake).

  parent(bill, shelly).

  sibling(X, Y) :- parent(M, X), parent(M, Y), X <> Y.

- **goal**

  sibling(X, Y).

results in:

- X = jake
- Y = shelly
- Because it goes through the facts first with parent(M, X) and instantiates them to bill and jake. Then parent(M,Y) starts at the beginning of the facts and tries to find a fact with bill as parent. It finds bill and jake again. But then X <> Y becomes jake <> jake which fails, so backtracking occurs to parent(M, Y). Prolog remembers where it left off in the list of parent facts, so it gets the next one (bill, shelly) that's good, so then X <> Y becomes jake <> shelly which is satisfied. Done.
- Note: not equals in the syntax of some prologs is: \==

Prolog

- **Fail**
- Can be used (as a subgoal) to force a goal to fail, thus forcing backtracking.
- Only makes sense as the last subgoal of a goal (because any subgoals after it would never get tested.)

Prolog

- Let's look at a couple of programs I have ready to show you.
  - factorial
  - can buy
- Let's write a towers of hanoi solver together

Prolog

- Applications of Logic Programming
  - For relational databases, SQL is non procedural
    - Databases / tables could be described by prolog structures
    - Relationships between tables can be described by prolog rules
    - Retrieval from a database is related to resolution
    - Goals in prolog are like queries in relational databases
  - Expert systems – emulate human expertise in some domain
  - Natural language processing – describe syntax related to CFGs, semantics can possibly be made clear with logic programming

Prolog

- **Example:**

  member(Element, [Element | _]).

  member(Element, [_ | List]) :- member(Element, List).

  /*

  _ is anonymous variable, use when we won't use the variable in the proposition.

  ***/**

  These member propositions state that if first parameter is the head of the second parameter, then member is satisfied. If not, then Element is a member of the List if Element is in the tail of the List.