CS131, Spring19 – Discussion 1B

Week 6 (05/10/19)

Administrations

1. TA: Wenhao Zhang (wenhaoz@cs.ucla.edu)
2. Office hour:

- Time: 9:30am ~ 11:30am, Monday
- Location: 3rd floor, common area in Eng VI

3. Class Announcements

- All TAs slides are under **resources tab** on Piazza
- HW3 dued on **05/06**
- HW4 dues on **05/14** (This coming Tuesday)
- Midterm is still being graded
- Please dont copy & paste code from stackoverflow or other sources.

4. Today's agenda

- Prolog basics
- Finite domain solver
- Cut!
- HW4 walkthrough

Prolog basics

Declarative programming

- Describing what we want to achieve, not how to do it.
- Examples: SQL, Prolog, Regular expressions

Prolog overview

- Logic programming language
- Particularly suited to programs that involve symbolic or non-numeric computation. For this reason it is a frequently used language in Artificial Intelligence where manipulation of symbols and inference about them is a common task.
- Programs defined using Facts, Rules, and Queries

Install Prolog

We'll be using **GNU Prolog**, or GProlog. Make sure you're using this and **not** SWI-Prolog. The download link is here You can also run it in SEASnet servers with command `gprolog`.

How to run prolog code

Prolog consists of a series of **facts** and **rules**. These facts and rules are in a seperate file, e.g. `myrules.pl`. A program is run by presenting some query and seeing if this can be proved against these known rules and facts. In interactive Prolog environment, you consult the rule file: `[myrules].` or `consult('myrules').`. After that, you can run queries in the interactive environment.

Simple facts

In Prolog we can make some statements by using facts. Facts either consist of a particular item or a relation between items. **Facts defines what is true in our knowledge base**. The inference process is based on **closed world assumption**. It states that a statement that is true in the knowledge base is also known to be true. Conversely, what is not currently known to be true in our knowledge base is false.

Syntax:

Facts should always begin with a lowercase letter and end with a period.

For example,

```
/*facts in our knowledge base*/
sunny.
raining.

/*query the knowledge base*/
?- sunny.

yes
```

`?-` is the Prolog prompt. To this query, Prolog will answer `yes`. sunny is true because (from above) Prolog matches it in its database of facts.

Question: what's the output of this following query?

```
?- foggy

/*answer is*/
no
```

Facts with arguments

More complicated facts consist of a relation and the items that this refers to. These items are called arguments. Facts can have arbitrary number of arguments from zero upwards. A general model is shown below:

```
relation(<argument1>,<argument2>,....,<argumentN> ).
```

Relation names must begin with a lowercase letter, e.g.

```
likes(john,mary).
```

The above fact says that a relationship likes links john and mary. This fact may be read as either john likes mary or mary likes john.

Look at a more complicated example,

```
/*facts in our knowledge base*/
eats(fred,oranges).                         /* "Fred eats oranges" */

eats(tony,apples).                          /* "Tony eats apples" */

eats(john,grapefruit).                      /* "John eats grapefruit" */
/*query the knowledge base*/
?- eats(fred,oranges).         /* does this match anything in the database? */

yes

?- eats(mike,apples).          /* how about this query, does mike eat apples */

no
```

Variables and unification

Let's say we have this fact in our knowledge base,

```
eats(fred,oranges).
```

We would like to ask the following question, "what does fred eat?". Something like, `?- eats(fred, what)`. But this does not work. This is because `what` is a fact/constant and it doesn't match with `oranges`. Hence, we need the help of variables here.

Variables are distinguished by starting with a capital letter.

```
X.                /* a capital letter */
VaRiAbLe.          /* a word - it be made up or either case of letters */
```

Back to our `eats(fred, oranges)` knowledge base, let's query it in this way,

```
?- eats(fred, What).

What = orange /*this is unification*/

yes
```

The process of matching items with variables is known as unification.

If we have more than one facts,

```
/*facts*/
eats(fred, oranges).
eats(mike, apple).
eats(lisa, banana).


/*query*/
?- eats(X,Y).

X = fred
Y = oranges ? .
/* Action (; for next solution, a for all solutions, RET to stop) ? */
```

Rules

Rules allow us to make conditional statements about our world. Each rule can have several variations, called clauses. These clauses give us different choices about how to perform inference about our world. Let's take an example to make things clearer. Consider the following statement,

> "All men are mortal"

We can express this as the following Prolog rule,

```
mortal(X) :-
        human(X).
```

the clause can be read in two ways (called either a declarative or a procedural interpretation). The declarative interpretation is "For a given X, X is mortal if X is human." The procedural interpretation is "To prove the main goal that X is mortal, prove the subgoal that X is human."

```
mortal(X) :-
          human(X).
human(socrates).

/*query the rule*/
?- mortal(socrates).
```

```
yes
```

Why was this? Well in order to solve the query ?- mortal(socrates)., we used the rule we saw previously. This said that in order to prove someone mortal, we had to prove them to be human. Thus from the goal Prolog generates the subgoal of showing human(socrates). When our subgoal succeeds, our overgoal succeeds automatically.

We can also use variables within queries. For example, we might wish to see if there is somebody who is mortal. This is done by the following line.

```
?- mortal(P).

P = socrates

yes
```

This means that Prolog was able to prove the goal by binding the variable P to socrates. This was done by again proving someone was mortal by proving the subgoal that they were human. Prolog thus asked if there was any P that was human. This matches against the clause human(socrates) thereby binding P to socrates. This binding is then passed back to the parent goal, and the results in the printout we saw above.

We can create more complicated logic using AND (`,`) and OR (`;`) logic connectors.

```
/* facts and rules*/
fun(X) :-
       red(X),
       car(X).

fun(X) :-
       blue(X),
       bike(X).

car(vw_beatle).
car(ford_escort).
bike(harley_davidson).
red(vw_beatle).
red(ford_escort).
blue(harley_davidson).

/*query the knowledge base*/
?- fun(harley_davidson).

yes

?- fun(What).
```

```
What=vw_beatle

yes
```

Let's see how Prolog deals with this query. Firstly we will try the first clause of fun. This results in us trying the goal `red(What)`. This succeeds matching the first clause of red with the binding `What=vw_beatle`. Now we attempt the goal `car(vw_beatle)`. This matches the first clause of car, and, as a result, the fun goal succeeds.

Backtracking

Let's now talk about backtracking in Prolog. Say we have the following knowledge base,

```
/*facts*/
eats(fred,pears).
eats(fred,t_bone_steak).
eats(fred,apples).

/*query*/
?- eats(fred,FoodItem).

FoodItem = pears

FoodItem = t_bone_steak

FoodItem = apples
```

After the first solution, prolog will ask us if we want other possible solutions. The mechanism for finding multiple solution is called backtracking.

```
/*facts and rules*/
hold_party(X):-
             birthday(X),
             happy(X).

birthday(tom).
birthday(jane).
birthday(helen).

happy(mary).
happy(jane).
happy(helen).

/*query*/
```

```
?- hold_party(Who).

X = jane ? ;

X = helen

yes
```

In order to solve the above, Prolog first attempts to find a clause of birthday, it being the first subgoal of birthday. This binds X to tom. We then attempt the goal happy(tom). This will fail, since it doesn't match the above database. As a result, Prolog backtracks. This means that Prolog goes back to its last choice point and sees if there is an alternative solution. In this case, this means going back and attempting to find another clause of birthday. This time we can use clause two, binding X to jane. This then causes us to try the goal happy(jane). This time we find clause two of birthday, and bind X to jane, and attempt the goal happy(jane). This goal matches against clause 2 of our happy database. As a result, hold_party will succeed with X=jane. Similarly, we found X=helen. **Notice that when prolog succeed the first time, it doesn't stop but continue search for other possible solutions**.

Prune search space – cut !

Cut, in Prolog, is a goal, written as !, which always succeeds, but cannot be backtracked past. Further implication: All subgoals to the left of the cut get pruned. Any other clauses below the cut clause also get pruned.

One canonical cut example, **not logic**

```
not(X) :-
       call(X), !, fail.

not(_).
```

Recursion

```
/*facts and rules*/
parent(john,paul).             /* paul is john's parent */

parent(paul,tom).              /* tom is paul's parent */

parent(tom,mary).              /* mary is tom's parent */

ancestor(X,Y):- parent(X,Y).   /* someone is your ancestor if there are your parent */

ancestor(X,Y):- parent(X,Z),   /* or somebody is your ancestor if they are the parent */
```

```
                    ancestor(Z,Y). /* of someone who is your ancestor */

/*query*/
?- ancestor(john,tom).

yes
```

The first clause of ancestor looks to see if there exists a clause that could match the goal parent(john,tom). This fails to match, as a result we try the second clause of ancestor. We now pose the query parent(john,Z). This results in us choosing clause one of parent and binding Z=paul. As a result, we now pose the recursive query ancestor(paul,tom). Applying the ancestor rule again, we first try the first clause. This means we check for parent(paul,tom). which successfully matches the second clause of parent. As a result the goal ancestor(paul,tom) succeeds. This in turn leads to the goal ancestor(john,tom) succeeding and Prolog responding yes

List

List basics

syntax: [var1, var2, . . . , varn].

We can do pattern matching as we did in Ocaml,

```
[1,2,3,4] = [A | B] -> A is bound to 1, B is [2, 3, 4]
[1,2,3,4] = [A, B | C] -> A = 1, B = 2, C = [3, 4]
```

Ok, let's see what this fact does.

```
/*fact*/
p([H|T], H, T).


/*query*/
?- p([a,b,c], X, Y).

X=a


Y=[b,c]


yes

?- p([a], X, Y).
```

```
X=a


Y=[]


yes

?- p([], X, Y).


no
```

List search

```
/*facts and rules*/
on(Item,[Item|Rest]).

on(Item,[DisregardHead|Tail]):-
                              on(Item,Tail).

/*query*/
?- on(apples,  [pears, tomatoes, apples, grapes]).
```

List construction

```
append([],List,List).
append([Head|Tail],List2,[Head|Result]):-
                                        append(Tail,List2,Result).

?- append([a,b,c],[one,two,three],Result)

Result = [a,b,c,one,two,three]
```

List member

From the manual: "member(Element, List) succeeds if Element belongs to the List. This predicate is re-executable on backtracking and can be thus used to enumerate the elements of List.

```
?- member(3, [1,2,3,4,5]).

true
```

List permutation

From the manual: “permutation(List1, List2) succeeds if List2 is a permutation of the elements of List1.”

```
?- permutation([3,2,1], [1,2,3]).

true
```

List length

From the manual: “length(List, Length) succeeds if Length is the length of List.”

```
?- length([1,2,3,4], 4).

yes

?- length([1,2,3,4], Len).
Len = 4
yes
```

Finite domain solver

- Finds assignments to variables that fulfill constraints
- Variable values are limited to a finite domain (e.g. integers between 0 and 10)

An example of generating a list of length N where each element is a unique integer between 1..N.

```
unique_list2(List, N) :-
    length(List,N),             /*Create a list of length N with no bound values*/
    fd_domain(List, 1, N),      /*Define all values in List to be between 1 and N*/
    fd_all_different(List),     /*Define all values in List to be different*/
    fd_labeling(List).          /*Generate a solution using backtracking*/
```

FD Arithmetic Constraints

- FdExpr1 #= FdExpr2 constrains FdExpr1 to be equal to FdExpr2.
- FdExpr1 #= FdExpr2 constrains FdExpr1 to be different from FdExpr2.
- FdExpr1 #< FdExpr2 constrains FdExpr1 to be less than FdExpr2.
- . . .

e.g.

```
?- X #< 5, fd_labeling(X).
X = 0 ?;
```

```
X = 1 ?;
X = 2 ?;
X = 3 ?;
X = 4
```

HW4 – Tower solver

- N*N square is filled with numbers 1..N so that values are not repeated in any row/column.
- Towers have different heights, can you determine the heights if you know how many can be seen from each position?
- Try it here, https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/towers.html

In your homework, write Prolog code for solving the heights based on how many can be seen and vice versa

Write two different implementations: One using FD solver and one without.

- Provide comparison of their performance
- Note: Non-FD solver probably won't work with larger grids. Testing with 5x5 is enough

Write a solver that finds ambiguous rules

- Multiple tower configurations can generated the same edge numbers.

Tips on HW4

- Try to make your solution efficient.
- Don't use FD solver in your plain solution.
- Use `?- statistics` to evaluate performance