CS 131
PROGRAMMING LANGUAGES
(WEEK 6)

UCLA WINTER 2019

TA: SHRUTI SHARAN

DISCUSSION SECTION: 1D

ADMINISTRATIVE INTRODUCTION

TA: Shruti Sharan

Email: shruti5596@g.ucla.edu
(Will reply by EOD)

Office Hours: Mondays 1.30PM – 3.30PM
Location: Eng. VI 3rd Floor

Discussion Section: Friday 4.00-5.50PM
Location: 2214 Public Affairs

TODAY'S AGENDA

PROLOG: INTRODUCTION

PROLOG ARITHMETIC AND LIST PROCESSING

FINITE DOMAIN SOLVER

HOMEWORK #4

Prolog

LOGICAL LANGUAGES

- If OOLs were all about Objects
- Functional Languages were all about functions…
- Then what are Logical Languages about?

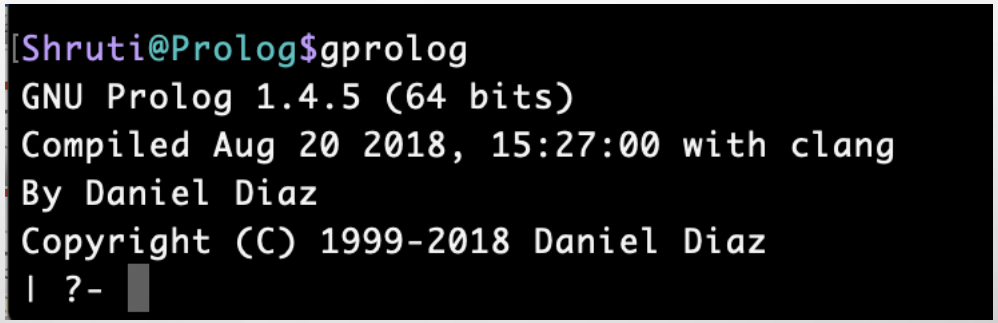
- Yep! Everything is built from **logic statements**

PROLOG: SETUP

- There are various implementations of Prolog.
- We will be using GNU Prolog for grading HW. (Open Source)
  - Windows and Macs:
    - http://www.gprolog.org/#download
    - http://www.gprolog.org/manual/gprolog.html
  - Linux
    - sudo apt install gprolog
  - Seasnet
    - Run using command ***gprolog***
- Another implementation exists called SWI Prolog, we will **not support** it for homework.

PROLOG - HOW TO RUN CODE

- Running the ***gprolog*** command on the terminal, you'll see:



- The last line is the Prolog prompt; it's ready for you to type a command.
- To exit interpreter: Control + D

PROLOG - HOW TO RUN CODE

- Prolog programs (databases) are stored in text files with **.pl file extension.**
- **To load file from the interpreter:**

```
?- consult('filename').

Or

?- ['filename'].
```

- This will load the database from the **file.pl** file into Prolog.

PROLOG - HOW TO RUN CODE

- If Prolog finds that everything is in order, it will say something like this:

```
| ?- [file].
compiling /Users/shrutisharan/Documents/CS131 Programming Languages/Prolog/file.pl
 for byte code...
/Users/shrutisharan/Documents/CS131 Programming Languages/Prolog/file.pl compiled,
 9 lines read - 1053 bytes written, 4 ms

(1 ms) yes
```

- With the database loaded, you can type in queries.

PROLOG: SO… WHAT IS A PROLOG PROGRAM?

- Prolog Programs are a set of **rules** and **queries.**
- In actuality, we create **Prolog databases.**
- In a database you have 2 types of clauses.
  - **Facts**
  - **Rules**
- Declarative: Prolog is all about **"what", not “how”**

PROLOG

- Prolog is a logic programming language
- Basic idea
  - Declare a set of facts
    - Single piece of information
    - Sky is blue
  - Declare a set of rules
    - Ways to generate new information based facts, other rules or even themselves.
    - A is grandparent of Z if A if parent of B and B is parent of Z
  - Run a query to ask if things are true and get solutions based on rules.

PROLOG FACTS

What are facts?

Ans: Defined set of truth statements

- Atoms
  - Uninterpreted constants
  - Always use lowercase
  - Eg. alice is an atom
- Predicate
  - A function that returns boolean.
  - person is a predicate
- Variable
  - Always UPPERCASE

```
person(alice).

person(bob).

blue(sky).

mammal(rabbit).
```

PROLOG RULES

- Rules describe a general relationship between facts.
- Syntax
  - head :- body
  - head and body are both clauses that usually use variables
  - Comma (,) is the AND operator, semi-colon (;) is the OR operator

```
grandparent(X,Z) :- parent(X,Y) , parent(Y,Z).

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

  - We don't explain how to use the rule, we just give prolog a database and it determines how and when to use the rules.

PROLOG QUERY

- Used to ask a question
- Query is like a rule without the body
- Prolog takes the head and tries to determine if true or false
- If we pass variables then prolog tries to determine all the variables.

WHAT HAPPENS AFTER RUNNING A QUERY?

- Three options
  - ; This will compute next solution
  - a This will compute all the remaining solutions
  - <return> This will stop the execution of the query

PROLOG: EXAMPLE

```
father(homer, bart).
father(homer, lisa).
father(homer, maggie).
father(grandpa, homer).
father(grandpa, herb).

mother(marge, bart).
mother(marge, lisa).
mother(marge, maggie).
mother(grandma, homer).
```

- paternal_grandfather(X, Y) :- father(X, Z), father(Z, Y).
- X is Y's paternal grandfather if there exists Z such that X is Z's father, and Z is Y's father
- X, Y, and Z are all variables, and the comma represents "and"
- paternal_grandmother(X, Y) :- mother(X, Z), father(Z, Y).

BACKTRACKING

- To understand performance of Prolog, we need to understand how it solves queries
- Prolog goes through facts/rules one-by-one in order
- If one choice of variables fails, it backtracks and tries the next one
- Prolog Visualizer:
  - http://www.cdglabs.org/prolog/#/
  - https://www.slideshare.net/ZhixuanLai/prolog-visualizer

TRACING THE LOGIC (DEBUGGING)

- Sometimes your programs may return wrong answer because of wrong, incorrect or incomplete rules
- Use trace to figure out why
- Use ***untrace*** command to turn off trace

```
| ?- trace.
The debugger will first creep -- showing everything (trace)

yes
{trace}
| ?- grandparent(hank,X).
      1    1  Call: grandparent(hank,_16) ?
      2    2  Call: parent(hank,_85) ?
      2    2  Exit: parent(hank,ben) ?
      3    2  Call: parent(ben,_16) ?
      3    2  Exit: parent(ben,carl) ?
      1    1  Exit: grandparent(hank,carl) ?

X = carl ? ;
      1    1  Redo: grandparent(hank,carl) ?
      2    2  Redo: parent(hank,ben) ?
      2    2  Exit: parent(hank,denise) ?
      3    2  Call: parent(denise,_16) ?
      3    2  Exit: parent(denise,frank) ?
      1    1  Exit: grandparent(hank,frank) ?

X = frank ? ;
      1    1  Redo: grandparent(hank,frank) ?
      3    2  Redo: parent(denise,frank) ?
      3    2  Exit: parent(denise,gary) ?
      1    1  Exit: grandparent(hank,gary) ?

X = gary

yes
{trace}
| ?-
```

ARITHMETIC IN PROLOG

x < y	X < Y.
x ≤ y	X =< Y.
x > y	X > Y.
x ≥ y	X >= Y.
x == y	X =:= Y
x ≠ y	X =\= Y

PROLOG: ARITHMETIC - 'is'

Posing the following queries yields:

?- 8 is 6+2.

yes

?- 12 is 6*2.

yes

?- -2 is 6-8.

yes

?- 3 is 6/2.

yes

?- 1 is mod(7,2).

yes

And we can also work out answers

?- X is 6+2.

X = 8

?- X is 6*2.

X = 12

?- R is mod(7,2).

R = 1

DIFFERENCE BETWEEN ‘is’ AND ‘=:=’

```
| ?- X=5+2.

X = 5+2

yes
```

```
| ?- X is 5 + 2.

X = 7

yes
```

- `=/2` triggers unification
- `is/2` triggers arithmetic evaluation

DEFINING PREDICATE WITH IS

subtract_5_and_double(X,Y) :- Y is (X – 5) * 2.

Example:

- subtract_5_and_double(6 , 1). -> no
- subtract_5_and_double(6 , X). -> X=2 -> yes
- subtract_5_and_double(X , 1). -> Error… why?

‘is’ USAGE

- Arithmetic statements must be on the right with variables that have an assigned value.
  - X is 6 + 2 allowed
  - 6 + 2 is X not allowed
  - X is Y + 2 will only work if Y has some value assigned to it
- http://www.gprolog.org/manual/gprolog.html#sec98
  - Documentation describe how to use “is”

EQUALITY OPERATOR (==) VS EQUALITY UNIFICATION (=)

- The goal term1==term2 succeeds only if term1 is identical to term2.
  - likes(X,prolog)==likes(john,Y). -> no
  - likes(X,prolog)==likes(X,prolog). -> yes
- The goal term1=term2 succeeds only if term1 and term2 unify, i.e. there is some way of binding variables to values which would make the terms identical.
  - likes(X,prolog)=likes(john,Y). -> X = john Y= prolog -> yes

PROLOG: GENERAL INFO - LISTS

- Lists are written between brackets [and]
  - [ ] is the empty list
  - [b , c] is a list of two symbols b and c
- If H is a symbol and T is a list, then [H | T] is a list with head H and tail T.
  - [a , b , c] = [a | [b , c]]

LIST IN PROLOG

- Pattern Matching
  - ?- [a, b] = [a, X]. -> X = b -> yes
  - ?- [a,b] = X. -> X = [a,b] -> yes
  - ?- [a,b] = [X]. -> no
- Head and Tail
  - ?- [a,b,c,d] = [H|T]. -> H = a T = [b,c,d] yes
  - ?- [a] = [H|T]. -> H = a T = []

LIST PROCESSING

- append(List1, List2, List12)

```
| ?- append([a,b,c],[1,2,3],[a,b,c,1,2,3]).

yes
| ?- append([a,b,c],[1,2,3],[1,2,3,a,b,c]).

no
```

```
| ?- append(X,Y,[a,b]).

X = []
Y = [a,b] ? ;

X = [a]
Y = [b] ? ;

X = [a,b]
Y = []

(1 ms) yes
```

- member(Term, List1)

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

true ? ;

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

true ?

yes
```

- reverse(List1, List2)

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

yes
```

LIST PROCESSING

- prefix(Prefix, List)
- suffix(Prefix, List)
- last(Term,List)
- length(List,Integer)

```
| ?- suffix([0],[9,1,0]).

true ?

yes
| ?- prefix([9],[9,1,0]).

yes
```

```
| ?- last([7,8,9],9).

yes
| ?- length([1,2,3],3).

yes
```

More here

- http://www.gprolog.org/manual/html_node/gprolog044.html

LISTS - PERMUTATION

- From GNU Prolog Manual:
  "permutation(List1, List2) succeeds if List2 is a permutation of the elements of List1."

- Watch out about using variables to give you all possible permutations. Will get a stack overflow very fast.

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

true ?

(1 ms) yes
| ?- permutation([1,2],[3,4]).

no
| ?- permutation([1,2],X).

X = [1,2] ? ;

X = [2,1] ? ;

no
```

LISTS - nth

- nth(N, List, Element) succeeds if the Nth argument of List is Element.

```
| ?- nth( A, [1,2,3,4,3],3).

A = 3 ? ;

A = 5 ?

yes
| ?- nth(3,[4,5,6,5,4],B).

B = 6

yes
| ?- nth(3,L,1).

L = [_,_,1|_]

yes
```

LISTS - maplist

- maplist(Goal, List) succeeds if Goal can successfully be applied on all elements of List

```
| ?- maplist(>(5),[1,2,3,4])
.

yes
| ?- maplist(>(5),[1,2,3,4]).

yes
| ?- maplist(=(1),[1,1,1]).

yes
| ?- maplist(write,[1,2,3]).
123

yes
```

LIST CONSTRUCTION

- Write a function my_append(X,Y,Result) which sets Result as X followed by Y
- Start with the easy case:

```
my_append([],X,X).
```

- Then the more complicated case:

```
my_append([XH|XT],Y,[XH|RT]) :- my_append(XT,Y,RT).
```

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

yes
| ?- append([1,2,3],[4,5],[1,2,3,4,5]).

yes
| ?- append([1,2,3],[4,5],[1,2,3,4,5,6]).

no
```

Succeeds if the concatenation of the list X and the list Y is the list Result.

TRACE OF LIST CONSTRUCTION

```
| ?- my_append([1,2,3],[a,b,c],Result).
[      1    1  Call: my_append([1,2,3],[a,b,c],_291) ?
[      2    2  Call: my_append([2,3],[a,b,c],_324) ?
[      3    3  Call: my_append([3],[a,b,c],_351) ?
[      4    4  Call: my_append([],[a,b,c],_378) ?
[      4    4  Exit: my_append([],[a,b,c],[a,b,c]) ?
[      3    3  Exit: my_append([3],[a,b,c],[3,a,b,c]) ?
[      2    2  Exit: my_append([2,3],[a,b,c],[2,3,a,b,c]) ?
[      1    1  Exit: my_append([1,2,3],[a,b,c],[1,2,3,a,b,c]) ?

Result = [1,2,3,a,b,c]
```

LIST REMOVAL

- Write function **remove(X, List, Result)** that sets Result to be otherwise the same as **List** but removes occurrences of **X**
- E.g. remove(1, [1,2,3,1,2,3], Result) would bind Result to [2,3,2,3]

```
remove(X, [], []).

remove(X, [X|L1t], Result) :- remove(X, L1t, Result).

remove(X, [H|L1t], [H|Result]) :- remove(X, L1t, Result).
~
```

```
| ?- remove(1, [1,2,3,1,2,3], Result).

Result = [2,3,2,3] ?

yes
```

GENERATING A LIST WITH CONSTRAINTS

- Problem: Generate a list of length N where each element is a unique integer between 1 and N
- We can start by outlining what we need:

```
unique_list(List, N) :-
        length(List, N),
        elements_between(List, 1, N),
        all_unique(List).
```

GENERATING A LIST WITH CONSTRAINTS

```
unique_list(List, N) :-
        length(List, N),
        elements_between(List, 1, N),
        all_unique(List).
```

Provided by Prolog

Prolog provides only ***between(Min, Max, X)***

Not provided by prolog

SOLUTION:

```
unique_list(List, N) :-
        length(List, N),
        elements_between(List, 1, N),
        all_unique(List).
```

```
elements_between([], _, _).
elements_between([H|T], Min, Max) :-
        between(Min,Max,H),
        elements_between(T, Min, Max).
```

```
all_unique([]).
all_unique([H|T]) :- exists(H, T), !, fail.
all_unique([H|T]) :- all_unique(T).
```

Query:

```
?- unique_list(List, 6).
List = [1,2,3,4,5,6]
```

Is this efficient??

N^N possible lists to iterate over.

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)
- A domain Dx is associated with each variable X that appears in a constraint.
  - Initially: $D_X = [0, \ldots, \text{fd\_max\_integer}] \subseteq \mathbf{N}^+$
- Less code, optimized solution

FINITE DOMAIN OPERATIONS

Priority	Specifier	Operators
750	xfy	#<=> #\<=>
740	xfy	#==> #\==>
730	xfy	## #\/ #\\/
720	yfx	#/\ #\/\
710	fy	#\
700	xfx	#= #\= #< #=< #> #>= #=# #\=# #<# #=<# #># #>=#
500	yfx	+ -
400	yfx	* / // rem
200	xfy	**
200	fy	+ -

FINITE DOMAIN SOLVER

- Constraints reduce the domain of variables.

```
| ?- X #< 3.

X = _#2(0..2)

yes
```

```
| ?- X + Y #= 5.

X = _#20(0..5)
Y = _#37(0..5)

yes
```

```
| ?- X #< 3 , X + Y #= 6.

X = _#2(0..2)
Y = _#39(4..6)

yes
```

```
| ?- X #< 5, X #= Y.

X = _#2(0..4)
Y = _#2(0..4)

yes
```

FINITE DOMAIN SOLVER

- **The predicates #=, #>, … <u>don't completely solve the constraints.</u>**
- Constraints only eliminate certain possible solutions
- To solve, we need to use **fd_labeling**

FINITE DOMAIN SOLVER

- The predicate fd_labeling solves all constraints that have been posted.
- fd_labeling(Vars, Options) assigns a value to each variable X of the list Vars according to the list of labeling options given by Options.

```
| ?- X #< 3 , X + Y #= 6.

X = _#2(0..2)
Y = _#39(4..6)

yes
```

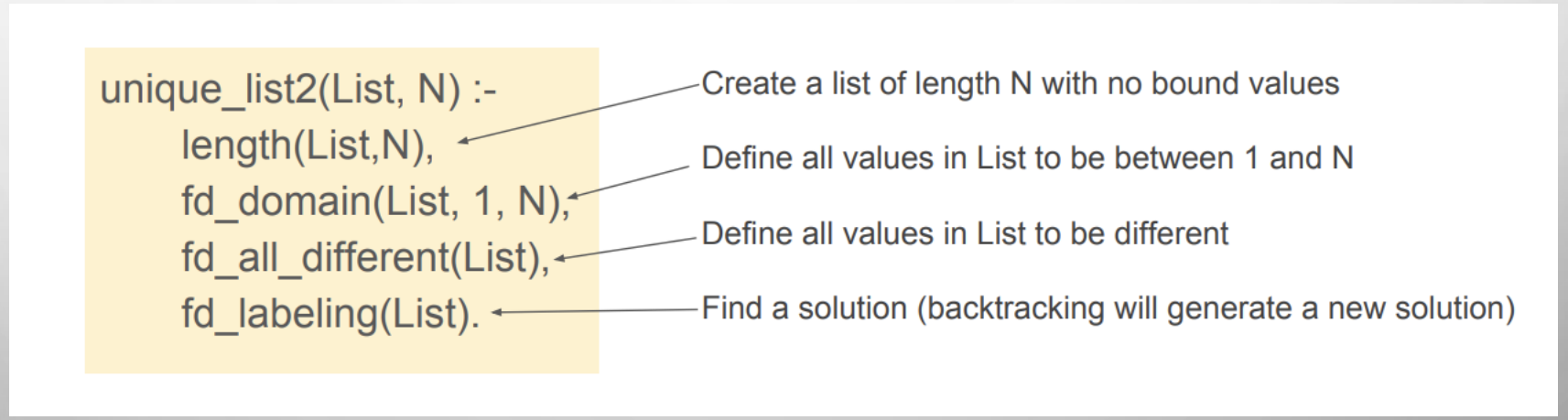
```
| ?- X #< 3 , X + Y #= 6 , fd_labeling([X,Y]).

X = 0
Y = 6 ? ;

X = 1
Y = 5 ? ;

X = 2
Y = 4

yes
```

FINITE DOMAIN SOLVER: BUILT-IN PREDICATES

- There are various Built-in predicates defining various constraints in Prolog:
- Some useful ones:
  - fd_domain(X , L) : Removes from the domain of X values that are not in L.
  - fd_min , fd_max
  - fd_all_different(L) : Constrains all variables in list L to take distinct values.
  - fd_domain_bool(L) : Removes from the domain of each variable in L values that are not in {0, 1}.
- Find detailed list here:
  http://www.gprolog.org/manual/html_node/gprolog054.html

FINITE DOMAIN SOLVER

- Lets solve the earlier problem using the FD solver:



Optimized solution with a fraction of the code!

EFFICIENCY AND ORDER OF EVALUATION

- What order does Prolog evaluate statements?
- Why is this important?
- How can we use this information to write better code?

ORDER OF EVALUATION - EXAMPLE

```
integers_ascending(Is0, Is) :-
        all_distinct(Is0),
        permutation(Is0, Is),
        ascending(Is).
```

```
integers_ascending(Is0, Is) :-
        all_distinct(Is0),
        ascending(Is),
        permutation(Is0, Is).
```

Which solution is better?

Case 1:

```
?- time(integers_ascending([10,9,8,7,6,5,4,3,2,1], Ls)).
% 64,398,640 inferences, 9.928 CPU in 9.994 seconds (99% CPU, 6486524 Lips)
Ls = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] .
```

Case 2:

```
?- time(integers_ascending([10,9,8,7,6,5,4,3,2,1], Ls)).
% 168,432 inferences, 0.025 CPU in 0.026 seconds (98% CPU, 6699761 Lips)
Ls = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] .
```

- Ordering tricks:
  - Put the most restrictive subgoal first.
  - Also, put facts and base cases before recursive cases in a program.

CUTS (!)

- The cut, in Prolog, is a goal, written as ' ! ', which always succeeds, but cannot be backtracked past.
- There are two reasons why cuts (!) can increase the efficiency of a program.
  - **Pruning:** Cutting off useless (possibly infinite) branches of the search tree
  - **Determinacy:** Informing Prolog that no further backtracking can occur for a clause.
- When Prolog knows that no backtracking is possible for a predicate, it can make optimised calls (in particular, no backtracking info has to be kept in memory).

CUTS EXAMPLE

- let's define a (cut-free) predicate max which takes integers as arguments and succeeds if the third argument is the maximum of the first two.

```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X>Y.
```

```
?- max(2,3,Max).

Max = 3
yes

?- max(2,1,Max).

Max = 2
yes
```

- This is a perfectly correct program, and we might be tempted simply to stop here. But we shouldn't: it's not good enough.

```
max(X,Y,Y) :- X =< Y,!.
max(X,Y,X).
```

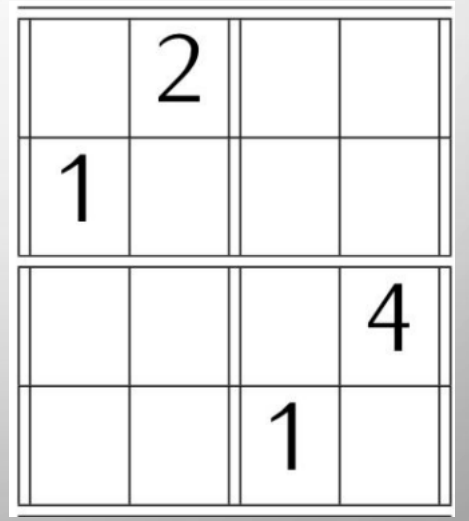
STATISTICS

- "Displays statistics about memory usage and run times."

```
| ?- statistics
.
Memory                limit         in use           free

   trail  stack       16383 Kb            0 Kb       16383 Kb
   cstr   stack       16384 Kb            0 Kb       16384 Kb
   global stack       32767 Kb            4 Kb       32763 Kb
   local  stack       16383 Kb            0 Kb       16383 Kb
   atom   table       32768 atoms      1778 atoms    30990 atoms

Times              since start      since last

   user   time        0.008 sec       0.008 sec
   system time        0.014 sec       0.014 sec
   cpu    time        0.022 sec       0.022 sec
   real   time     4137.140 sec    4137.140 sec

yes
```

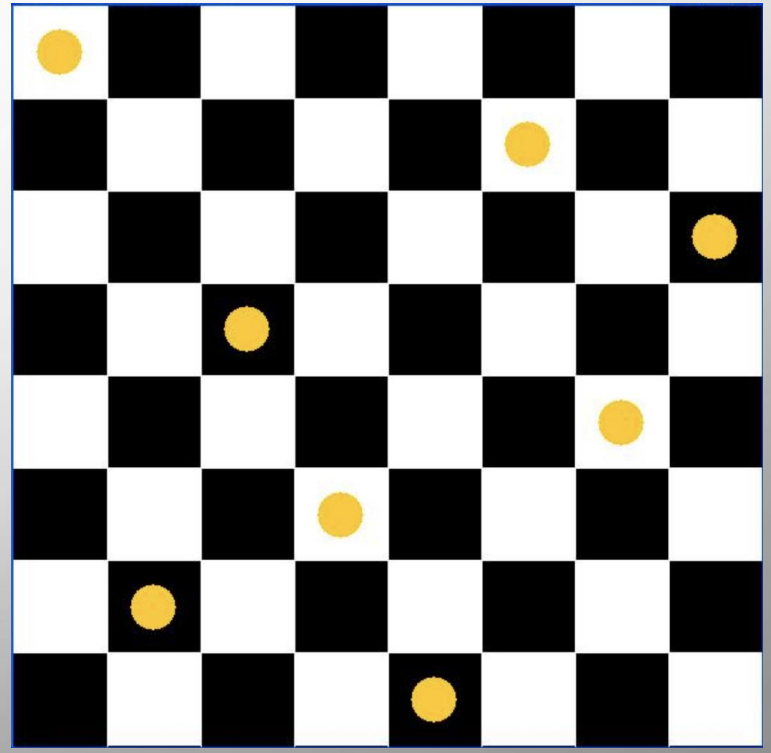
SUDOKU WITH FD

- How can you solve 4x4 Sudoku problem using FD solver?
- Start with definition sudoku4_fd(L):- …
- Use FD constraints:
  - fd_domain(List, Min, Max)
  - fd_all_different(List)



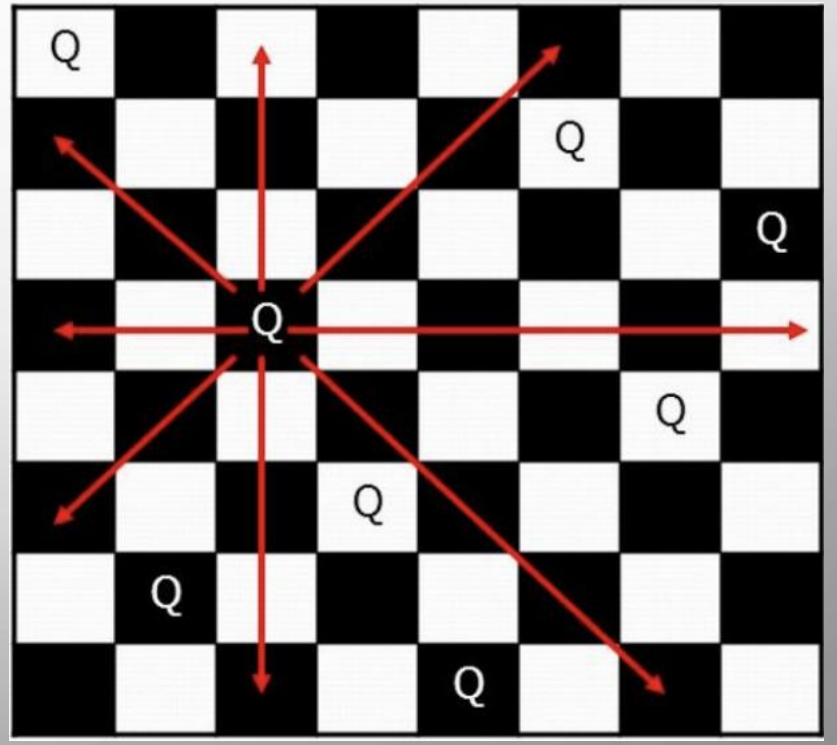
PRACTICE - EIGHT QUEENS PROBLEM

- Problem:
  - We have 8 queen pieces
  - 8 x 8 chess board
  - We want an arrangement of locations such that no queen can attack another queen



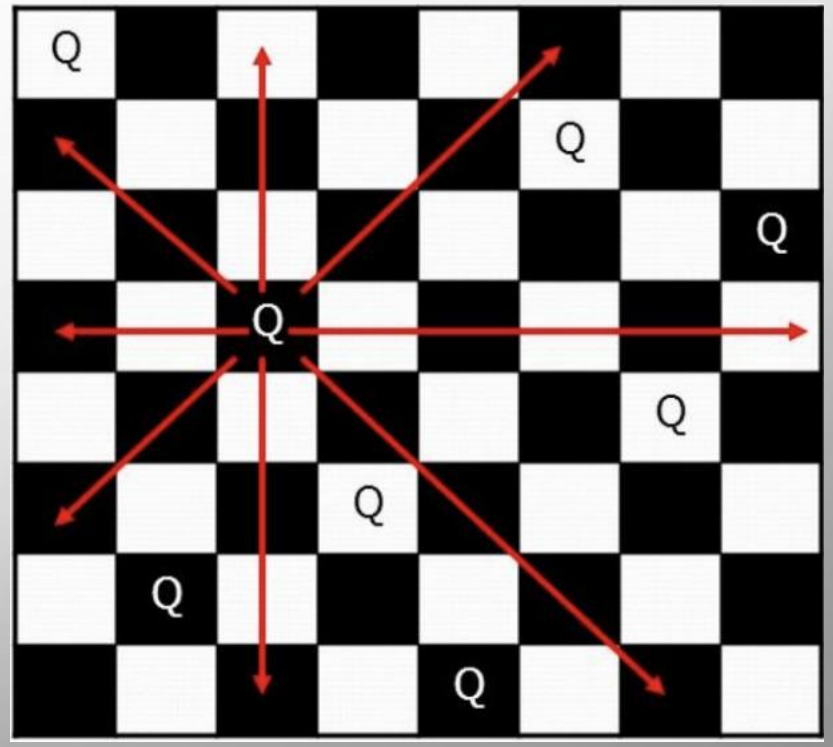
PRACTICE - EIGHT QUEENS PROBLEM

- **Step 1: Defining Constraints**
  - Placement
    - 8 x 8 possible locations
  - Possible Moves
    - Anywhere in same column
    - Anywhere in same row
    - Anywhere on either of its diagonals



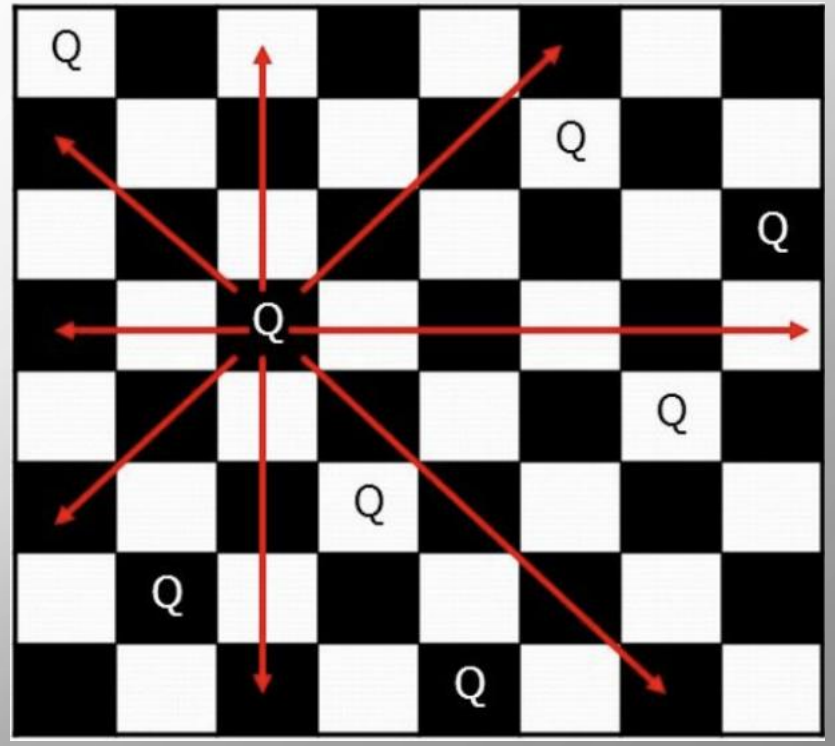
PRACTICE - EIGHT QUEENS PROBLEM

- **Step 2: Implementing Constraints**
  - Placement
    - Let's use a list of length 8
    - Each index represents a row
    - Each value in the list represents the column of the queen in that row
    - Why does this work?



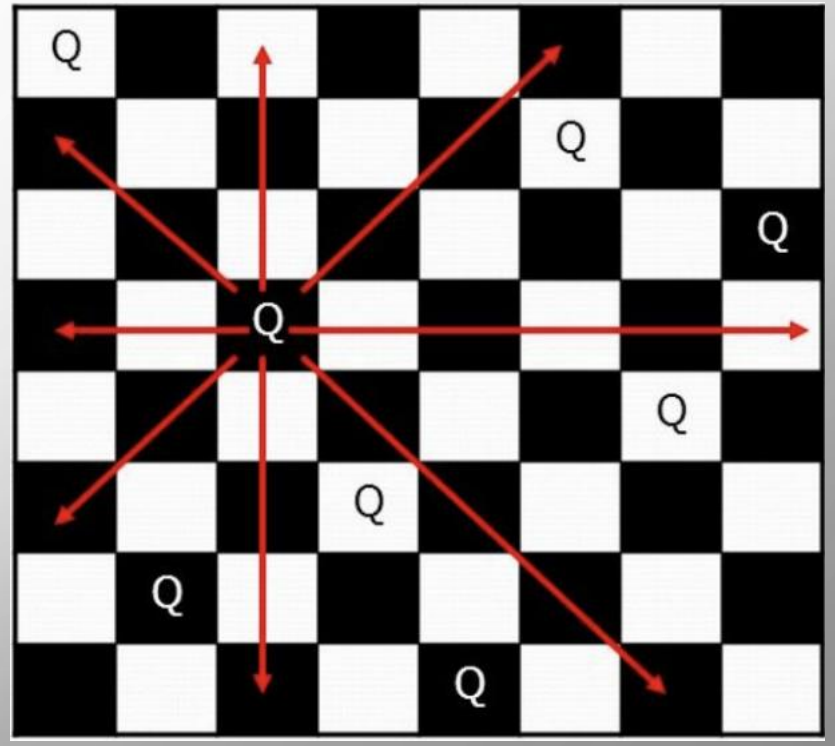
PRACTICE - EIGHT QUEENS PROBLEM

- **Step 2: Implementing Constraints**
  - Moves
    - Diagonals
      - A little trickier
      - Need to check for each item in the list, no other items on its diagonals
      - How can we check this?

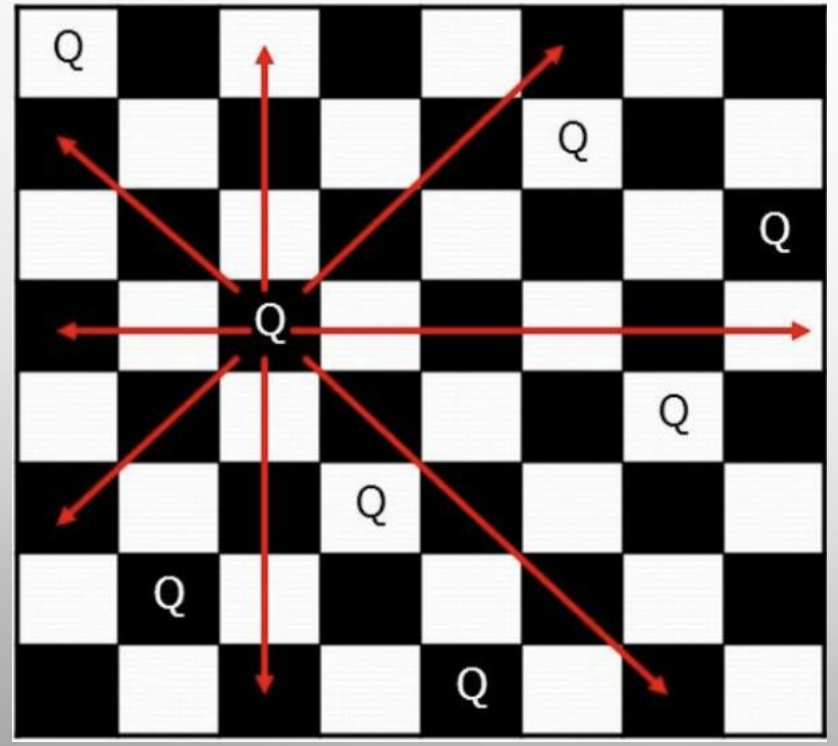


PRACTICE - EIGHT QUEENS PROBLEM

- **Step 3: Prolog Implementation**
  - How could the FD Solver be useful?
  - Which predicates might be useful?
  - What are our base cases?
  - How do we evaluate every position in list?



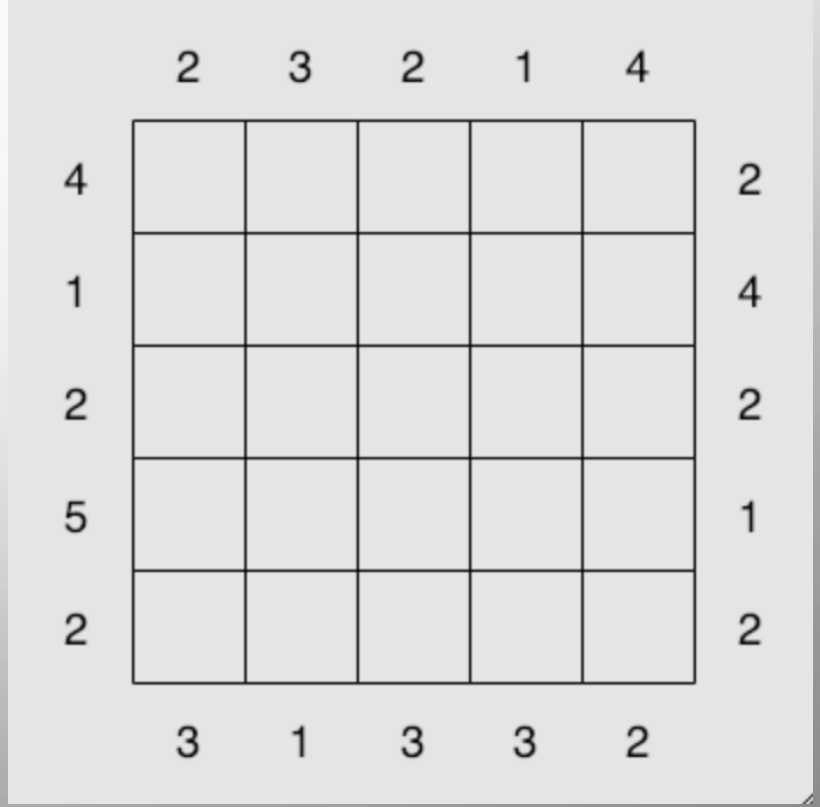
PRACTICE - EIGHT QUEENS SOLUTION




```
noattack(Q,[],_).
noattack(Q1,[Q2|Rest],Diff):-
    Q1#\=#Q2,
    Diff#\=#Q1-Q2,
    Diff#\=#Q2-Q1,
    Diff2 is Diff+1,
    noattack(Q1,Rest,Diff2).

safe([]).
safe([Q|T]):-noattack(Q,T,1),safe(T).

eightqueens(Solution):-
    Solution = [_,_,_,_,_,_,_,_],
    fd_domain(Solution,1,8),
    safe(Solution),
    fd_labeling(Solution).
```

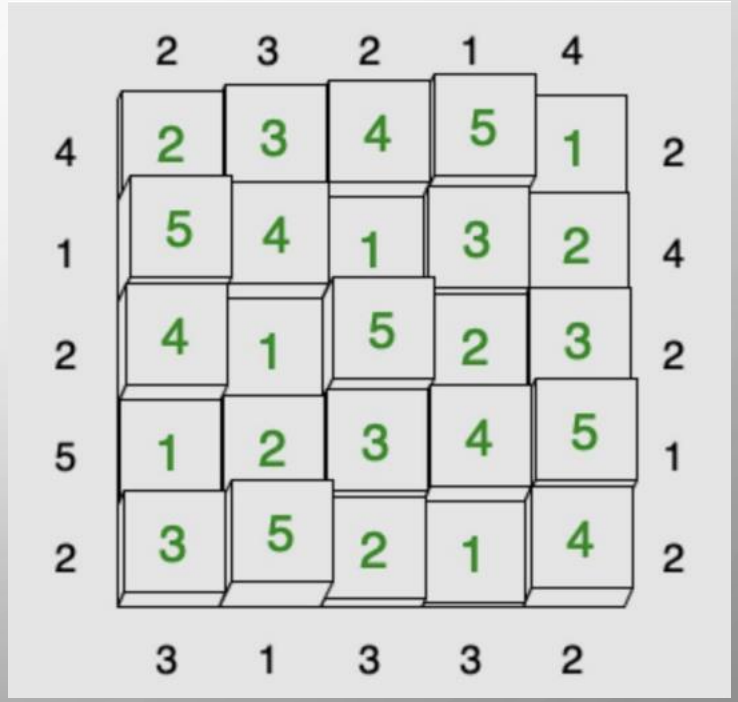
HOMEWORK #4 : TOWERS SOLVER

- N*N square is filled with numbers 1 to 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 online: https://www.chiark.greenend.org.uk/~sgtatham/puzzle s/js/towers.html



HOMEWORK #4 - TOWERS SOLVER

- 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
- Write a solver that finds ambiguous rules - i.e. Numbers on the side can be caused by multiple tower arrangements

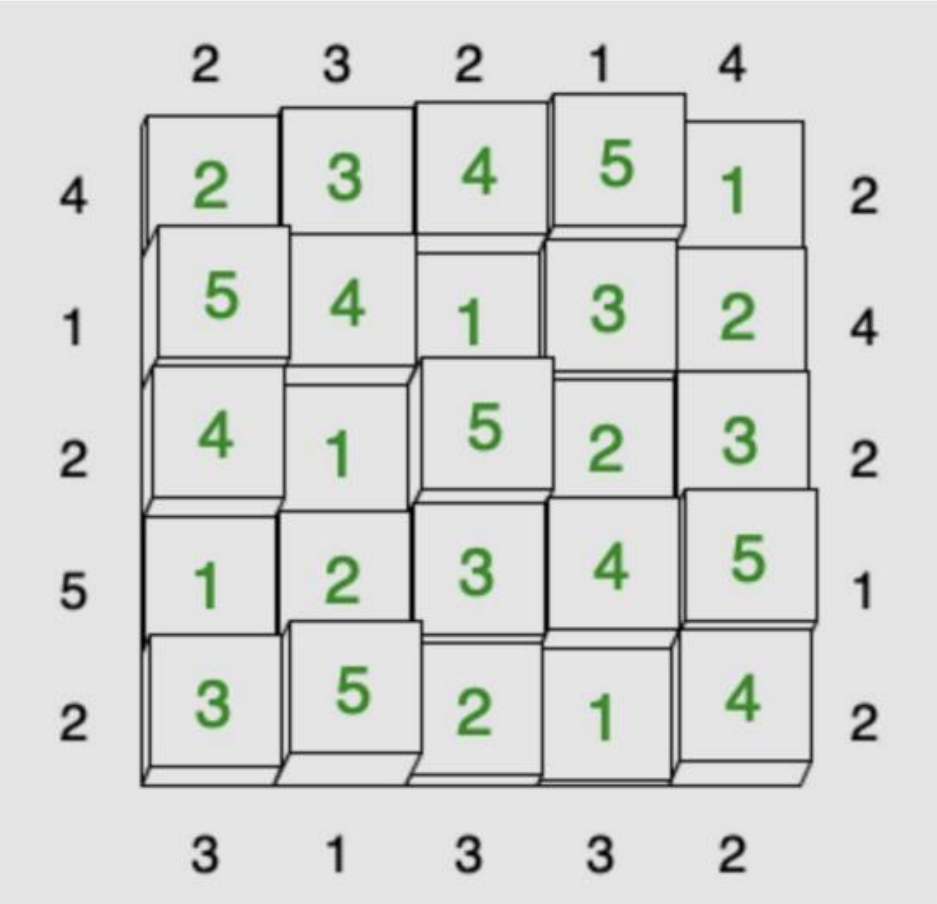


TOWERS:

- **BASIC CONSTRAINTS:**
- N x N board
- Each row and column:
  - All numbers in range 1 to N
  - No repeated values

- **GENERAL CONSTRAINTS:**
- Constraints on each row and column dictate ordering of values
- "How many towers are visible?"

TOWERS



- **GOALS**
- tower(N, T, C)
  - Can use FD Solver
- plain_tower(N, T, C)
  - Can't use FD Solver
- speedup(X)
  - Compares performance
- ambiguous(N, C, T1, T2)
  - Find ambiguous puzzle

HOMEWORK SUBMISSION

- Make sure everything compiles without warnings or errors on SEASnet servers!
- We will use automated tests for grading, so make sure your solution works exactly the same as instructions say.
- Submission is next Friday (02/22/2019).
- All the best!

PROLOG RESOURCES

- GNU Prolog manual: http://www.gprolog.org/manual/gprolog.html
- Prolog Wikibook: https://en.wikibooks.org/wiki/Prolog
- Prolog Visualizer: http://www.cdglabs.org/prolog/#/
- Tuturials :
  - http://www.cs.toronto.edu/~sheila/384/w11/Prolog/prolog-tutorial-part1.pdf
  - http://www.cs.toronto.edu/~sheila/384/w11/Prolog/prolog-tutorial-part2.pdf
- Prolog examples: http://www.cs.toronto.edu/~sheila/384/w11/simple-prolog-examples.html

- When looking for resources, make sure to check they are for GNU Prolog, not SWI-Prolog!