CS131 - Week 6

UCLA Spring 2019
TA: Kimmo Karkkainen

Today

- Prolog
- Homework #4

Declarative programming

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

Prolog

- Logic programming language
- Programs defined using *Facts*, *Rules*, and *Queries*
- This course uses GNU Prolog: http://www.gprolog.org
  - Make sure you are not using SWI-Prolog, they have lots of differences
  - You can run it on SEASnet servers with command ***gprolog***

Prolog - How to run code

- Facts and Rules written into a file, e.g. **myrules.pl**
- In interactive Prolog environment, you *consult* the rule file: **[myrules].**
- After that, you can run *queries* in the interactive environment

Facts

- Facts define what is true in our *database*
- Always start with a lowercase letter
- For example:

Prolog file:

```
raining.
john_is_cold.
john_forgot_his_raincoat.
```

Queries:

```
?- raining.
yes

?- john_is_cold.
yes

?- john_is_tired.
exception
```

Relations

- Facts consisting of one or more terms
- Closed-world assumption
- For example:

Prolog file:

```
student(fred).
eats(fred, oranges).
eats(fred, bananas).
eats(tony, apples).
```

Queries:

```
?- eats(fred, oranges).
yes.

?- eats(fred, apples).
no.

?- student(fred).
yes.
```

Variables and Unification

- Unification tries to find a way to fill the missing values
  - No return values in Prolog!
- Variable is any string that starts with a capital letter
  - E.g. My_variable, What, Who
- Unification binds *variables* to *atoms*

Prolog file:

```
eats(fred, oranges).
eats(fred, bananas).
eats(tony, apples).
```

Queries:

```
?- eats(fred, What).
What = oranges ? a
What = bananas
yes

?- eats(Who, apples).
Who = tony
yes
```

Rules

- Rules allow us to make conditional statement
- Syntax: *conclusion :- premises*.
  - Conclusion is true if the premises are true
- Consider the statement “All men are mortal”:

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

human(socrates).
```

Queries:

```
?- mortal(socrates).
yes

?- mortal(Who)
Who = socrates
yes
```

Rules

- Rules can contain multiple statements
    - Comma (,) is the AND operator, semi-colon (;) is the OR operator

```
red_car(X) :-
  red(X),
  car(X).

red(ford_focus).
car(ford_focus).
```

Queries:

```
?- red_car(ford_focus).
yes

?- red_car(What)
What = ford_focus
yes
```

Rules

- Rules can contain multiple statements
  - Comma (,) is the AND operator, semi-colon (;) is the OR operator

```
red_or_blue_car(X) :-
  (red(X);blue(X)),
  car(X).

red(ford_focus).
car(ford_focus).
```

Queries:

?- red_or_blue_car(ford_focus).
yes

?- red_or_blue_car(What)
What = ford_escort
yes

Recap

- Facts: Start with a lowercase letter
  - e.g. raining. john_hungry.
- Variables: Start with a uppercase letter:
  - e.g. What, Who, XYZ123
- Relations: Like facts, but including multiple *atoms* inside parentheses
  - e.g. likes(steve,dancing). hungry(john).
- Rules: conclusion :- premises.
  - e.g. parent(X,Y) :- father(X,Y); mother(X,Y).

Equality

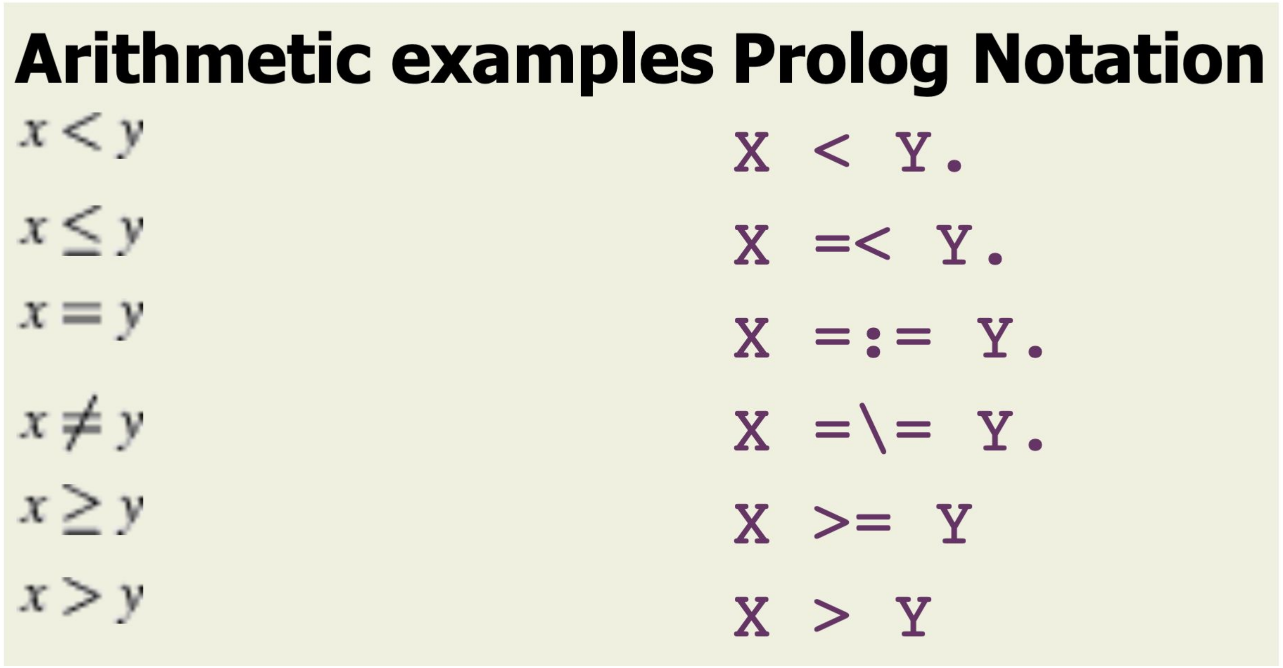
- Three equality operators: **=** , **is** , **=:=**
    - = tries unification directly, **is** evaluates the right side and unifies, **=:=** evaluates both sides

```
?- 7 = 5 + 2.
no

?- X = 5 + 2.
X = 5+2
yes

?- A + B = 5 + 2.
A = 5
B = 2
```

```
?- X is 5 + 2.
X = 7
yes

?- 7 is 5 + 2.
yes

?- 5 + 2 is 7.
no

?- X is 5 + Y.
uncaught exception:
error(instantiation_error,(is)/2)
```

```
?- 5 + 2 =:= 4 + 3.
yes

?- X =:= 4 + 3.
uncaught exception:
error(instantiation_error,(=:=)/2)

?- X = 5, Y = 5, X =:= Y.
X = 5
Y = 5
yes
```

Arithmetic comparisons



Arithmetic examples	Prolog Notation
$x < y$	`X < Y.`
$x \leq y$	`X =< Y.`
$x = y$	`X =:= Y.`
$x \neq y$	`X =\= Y.`
$x \geq y$	`X >= Y`
$x > y$	`X > Y`

Backtracking

```
hold_party(X) :-
    birthday(X),
    happy(X).

birthday(tom).
birthday(fred).
birthday(helen).

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

- To understand the 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
- E.g. consider query *hold_party(Who).*

Prolog visualizer: http://www.cdglabs.org/prolog/#/

Backtracking

```
hold_party(X) :-
    birthday(X),
    happy(X).

birthday(tom).
birthday(fred).
birthday(helen).

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

1. Ask query *hold_party(Who)*
2. Our database does not have relations *hold_party(name)*, so try applying the rule
3. Choose first person who matches *birthday(X)* (tom)
4. Try relation *happy(tom)*
5. Relation is not found, so backtrack to next *birthday*
6. Try again with fred
7. Relation *happy(fred)* fails too
8. Backtrack and try with *helen*
9. This time relation is found, so X=helen

Recursion

```
flight(lax,atl).
flight(atl,jfk).
flight(jfk,lhr).

can_travel(X,Y) :- flight(X,Y).
can_travel(X,Y) :- flight(X,Z),
                   can_travel(Z,Y).
```

Queries:

?- can_travel(lax,lhr).
yes

?- can_travel(lhr,lax)
no

Recursion

- How can we define ancestor(X,Y) to test whether Y is X’s parent, grandparent, grandgrandparent, etc?

```
father(john,paul).
father(paul,henry).
mother(paul,mary).
mother(mary,susan).
```

Recursion

- How can we define ancestor(X,Y) to test whether Y is X’s parent, grandparent, grandgrandparent, etc?

```
father(john,paul).
father(paul,henry).
mother(paul,mary).
mother(mary,susan).
```

```
ancestor(X,Y) :- father(X,Y); mother(X,Y).
ancestor(X,Y) :- (father(X,Z), ancestor(Z,Y));
                 (mother(X,Z), ancestor(Z,Y)).
```

```
?- ancestor(john,Who).
Who = paul
Who = henry
Who = mary
Who = susan
```

Lists

- Important data structure in Prolog
    - Especially in your homework
- Syntax: ***[val1, val2, val3, …, valn]***
- Similar to OCaml, we can do pattern matching to head and tail:
    - ***[1,2,3,4] = [A | B]*** -> A is bound to 1, B is bound to [2, 3, 4]
    - ***[1,2,3,4] = [A, B | C]*** -> A = 1, B = 2, C = [3, 4]
    - ***[1,2,3,4] = [A, B, C, D]*** -> A = 1, B = 2, C = 3, D = 4

Lists - Examples

- Consider the following relation:

```
p([H | T], H, T).
```

- What is the result of the following queries?

```
1)  p([a, b, c], a, [b, c]).
2)  p([a, b, c], X, Y).
3)  p([a], X, Y).
4)  p([], X, Y).
```

List searching

- How can we check if a specific element is in a list?
- Write a rule ***exists(X, List)*** which is true if ***X*** is in ***List***

List searching

- How can we check if a specific element is in a list?
- Write a rule ***exists(X, List)*** which is true if ***X*** is in ***List***

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

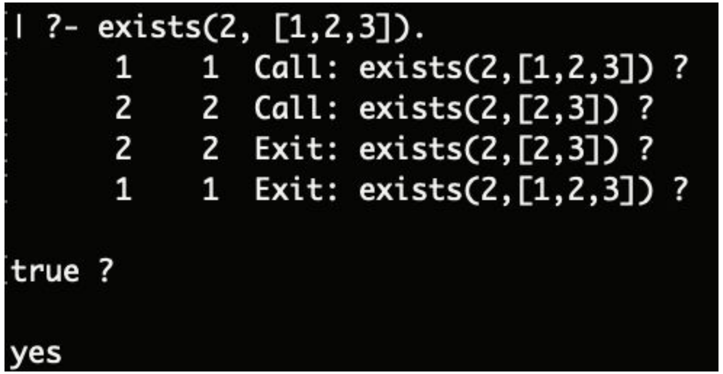
Queries:

?- exists(a, [a,b,c]).
yes.

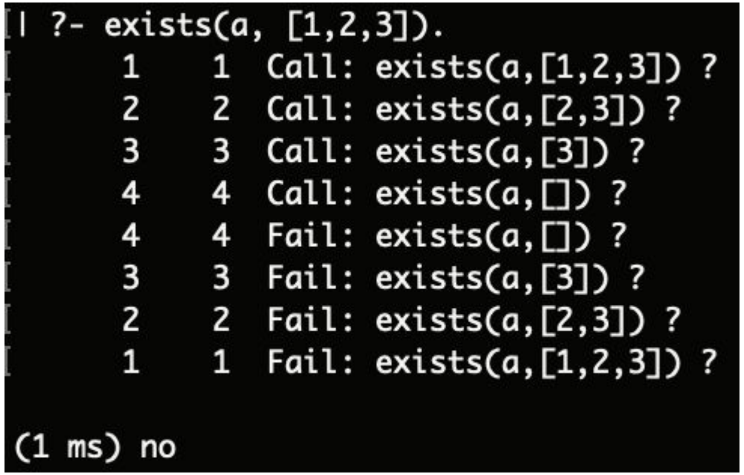
?- exists(a, [x,y,z]).
no.

How can I debug my program?

- ***trace.*** shows all the evaluations (turn off using ***notrace.***)



```
| ?- exists(2, [1,2,3]).
      1    1  Call: exists(2,[1,2,3]) ?
      2    2  Call: exists(2,[2,3]) ?
      2    2  Exit: exists(2,[2,3]) ?
      1    1  Exit: exists(2,[1,2,3]) ?

true ?

yes
```



```
| ?- exists(a, [1,2,3]).
      1    1  Call: exists(a,[1,2,3]) ?
      2    2  Call: exists(a,[2,3]) ?
      3    3  Call: exists(a,[3]) ?
      4    4  Call: exists(a,[]) ?
      4    4  Fail: exists(a,[]) ?
      3    3  Fail: exists(a,[3]) ?
      2    2  Fail: exists(a,[2,3]) ?
      1    1  Fail: exists(a,[1,2,3]) ?

(1 ms) no
```

List construction

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

```
append([], Y, Y).
```

- Then the more complicated case:

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

- Consider e.g. ***append([1,2,3], [a,b,c], [1,2,3,a,b,c])***

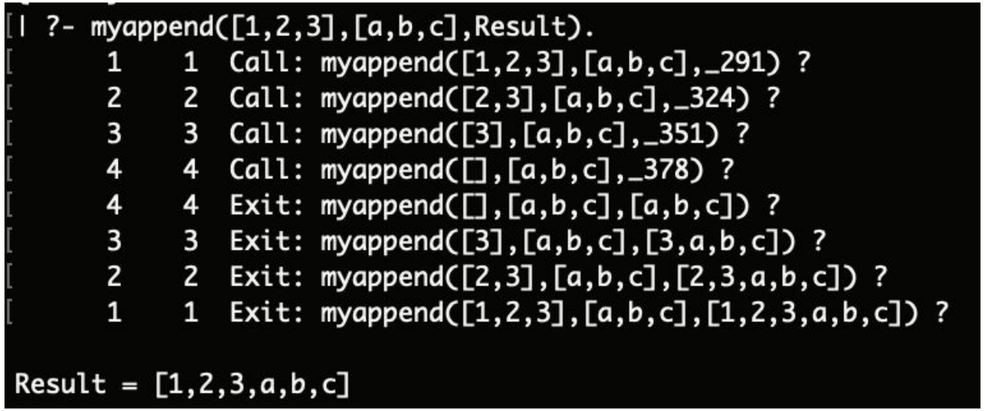
List construction

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

```
append([], Y, Y).
```

- Then the more complicated case:

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



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

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

- Consider e.g. ***append([1,2,3], [a,b,c], [1,2,3,a,b,c])***

List removal

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

List removal

- How to write function ***remove(X, List, Result)*** that sets ***Result*** to be otherwise the same as ***List*** but removing 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).
```

List - append

- **From the manual:** “append(List1, List2, List12) succeeds if the concatenation of the list List1 and the list List2 is the list List12.”

```
?- append([1,2], [3,4], Result).
Result = [1,2,3,4]

?- append(A, [3,4], [1,2,3,4]).
A = [1,2]
```

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

A = []
B = [1,2,3] ? a

A = [1]
B = [2,3]

A = [1,2]
B = [3]

A = [1,2,3]
B = []
```

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

?- member(X, [1,2,3]).
X = 1 ? a
X = 2
X = 3
```

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
```

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

List - permutation

- Note: You should have the known elements in the first argument:

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

```
?- permutation(X, [1,2,3]).
X = [1,2,3] ? a
Fatal Error: global stack overflow (size: 32768 Kb, reached:
32765 Kb, environment variable used: GLOBALSZ)
```

List - prefix / suffix

- **From the manual:** “prefix(Prefix, List) succeeds if Prefix is a prefix of List. suffix(Suffix, List) succeeds if Suffix is a suffix of List.”

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

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

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

?- length(List, 5).
List = [_,_,_,_,_]
yes
```

List - nth

- **From the manual:** “nth(N, List, Element) succeeds if the Nth argument of List is Element.

```
?- nth(5, [1,2,3,4,5,6], Element).
Element = 5
yes

?- nth(N, [1,2,3,4,5,6], 3).
N = 3
yes

?- nth(3, L, 5).
L = [_,_,5|_]
yes
```

List - maplist

- **From the manual:** “maplist(Goal, List) succeeds if Goal can successfully be applied on all elements of List.”

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

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

Generating a List with Constraints

- Problem: Generate a list of length N where each element is a unique integer between 1..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

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

```
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

Generating a List with Constraints

```
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).
```

Is there an easier way to write this?

Generating a List with Constraints

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

```
elements_between(List, Min, Max) :-
        maplist(between(Min,Max), List).
```

Generating a List with Constraints

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

```
elements_between(List, Min, Max) :-
        maplist(between(Min,Max), List).
```

```
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]
```

Generating a List with Constraints

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

```
elements_between(List, Min, Max) :-
        maplist(between(Min,Max), List).
```

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

Is this efficient?

Generating a List with Constraints

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

```
elements_between(List, Min, Max) :-
        maplist(between(Min,Max), List).
```

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

Is this efficient?

N^N possible lists to try

Finite Domain Solver

Finite Domain Solver

- Finds assignments to variables that fulfill constraints
- Variable values are limited to a finite domain (non-negative integers)
- Less code, optimized solution

Finite Domain Solver

- Lets solve the earlier problem using the FD solver:

```
unique_list2(List, N) :-
        length(List,N),
        fd_domain(List, 1, N),
        fd_all_different(List),
        fd_labeling(List).
```

Create a list of length N with no bound values

Define all values in List to be between 1 and N

Define all values in List to be different

Find a solution (backtracking will generate a new solution)

- Optimized solution with a fraction of the code!

FD Constraints

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.
- FdExpr1 #=< FdExpr2 constrains FdExpr1 to be less than or equal to FdExpr2.
- FdExpr1 #> FdExpr2 constrains FdExpr1 to be greater than FdExpr2.
- FdExpr1 #>= FdExpr2 constrains FdExpr1 to be greater than or equal to FdExpr2.

See documentation for more built-in constraints

FD Constraints

- Note that constraints do not find a solution - they just limit the options:

```
?- X #= Y.
X = _#0(0..268435455)
Y = _#0(0..268435455)

?- X #< 5.
X = _#2(0..4)

?- X #< 5, fd_labeling(X).
X = 0 ? ;
X = 1 ? ;
X = 2 ? ;
X = 3 ? ;
X = 4
```

```
?- X #< 3, X*Y #= 6.
X = _#2(1..2)
Y = _#22(3..6)
```

Sudoku with FD

How can you solve 4x4 Sudoku problem using FD solver?

Use FD constraints:

- fd_domain(List, Min, Max)
- fd_all_different(List)

	2		
1			
			4
		1	

Sudoku with FD

```
sudoku4_fd(L):-
        L = [X11,X12,X13,X14,X21,X22,X23,X24,X31,X32,X33,X34,X41,X42,X43,X44],
        fd_domain(L, 1, 4),
        fd_all_different([X11,X12,X13,X14]) , fd_all_different([X21,X22,X23,X24]),
        fd_all_different([X31,X32,X33,X34]) , fd_all_different([X41,X42,X43,X44]),
        fd_all_different([X11,X21,X31,X41]) , fd_all_different([X14,X24,X34,X44]),
        fd_all_different([X12,X22,X32,X42]) , fd_all_different([X13,X23,X33,X43]),
        fd_all_different([X11,X12,X21,X22]) , fd_all_different([X13,X14,X23,X24]),
        fd_all_different([X31,X32,X41,X42]) , fd_all_different([X33,X34,X43,X44]),
        fd_labeling(L).
```

```
?- sudoku4_fd([1,2,3,4,X21,X22,X23,X24,X31,X32,X33,X34,X41,X42,X43,X44]).

X21 = 3, X22 = 4, X23 = 1, X24 = 2, X31 = 2, X32 = 1, X33 = 4, X34 = 3, X41 = 4, X42 = 3,
X43 = 2, X44 = 1
```

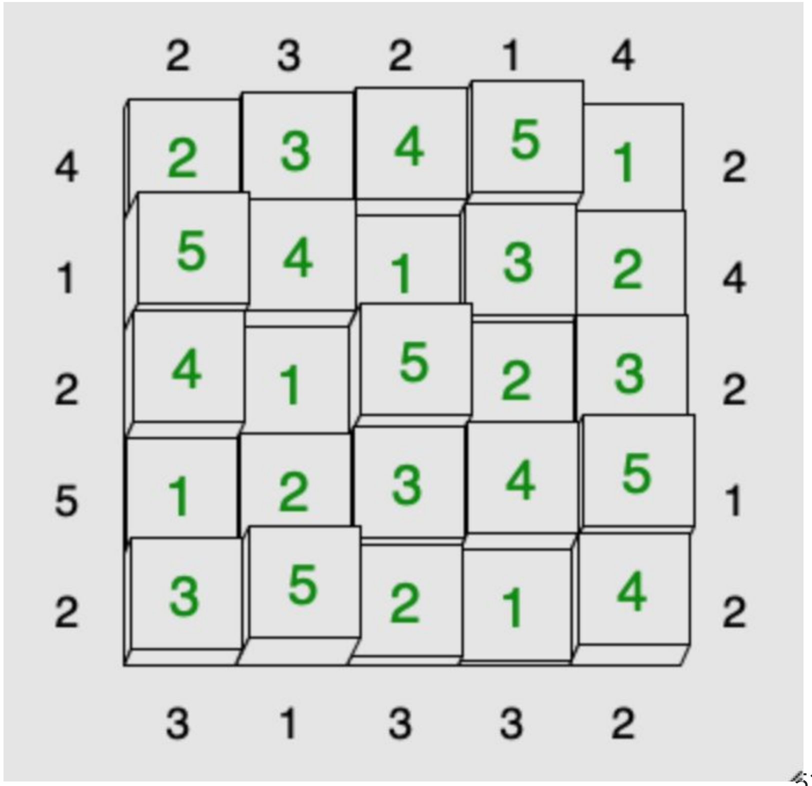
Homework #4

Homework #4 - Towers Solver (DL Friday Feb 22)

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 online:
https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/towers.html

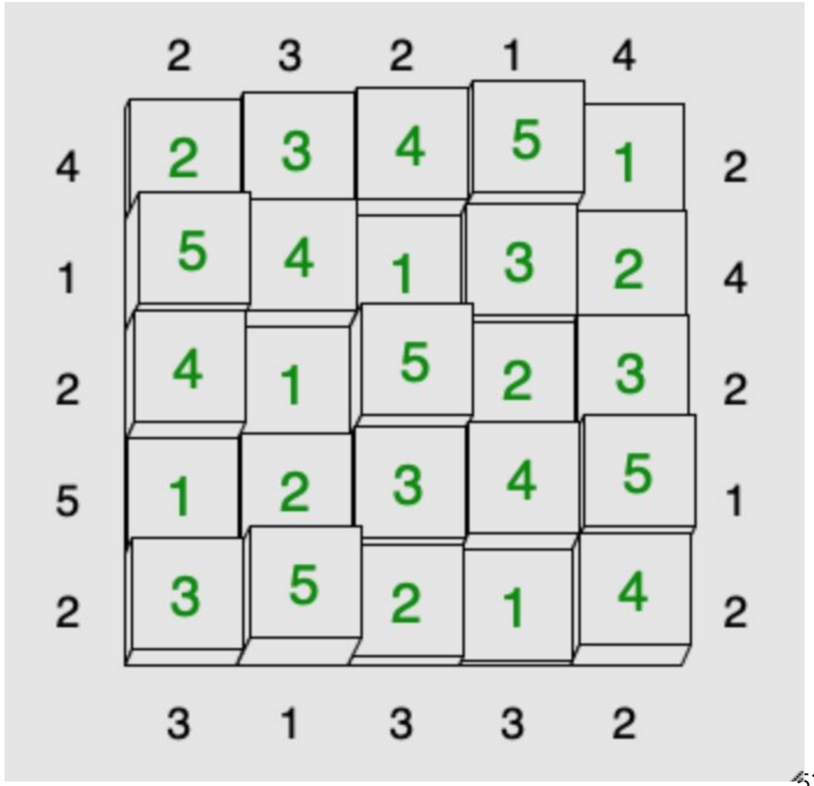


Homework #4 - Towers Solver

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 well with larger grids. Testing with 5x5 is enough

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



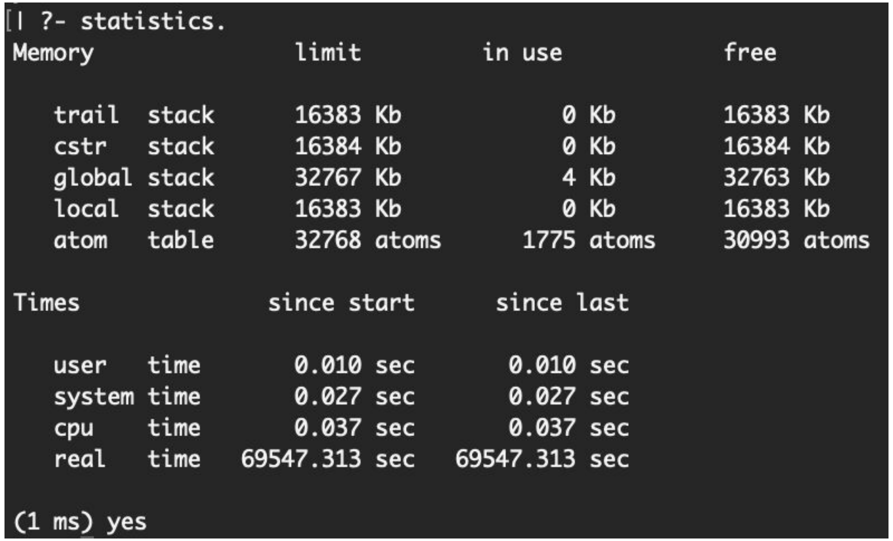
Homework #4 - Towers Solver

Examples:

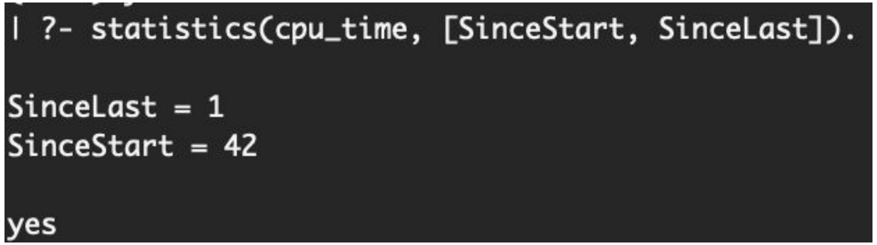
```
?- tower(5,
        [[2,3,4,5,1],
         [5,4,1,3,2],
         [4,1,5,2,3],
         [1,2,3,4,5],
         [3,5,2,1,4]],
        C).

C = counts([2,3,2,1,4],
          [3,1,3,3,2],
          [4,1,2,5,2],
          [2,4,2,1,2])
```

```
?- tower(5, T,
        counts([2,3,2,1,4],
               [3,1,3,3,2],
               [4,1,2,5,2],
               [2,4,2,1,2])).

T = [[2,3,4,5,1],
     [5,4,1,3,2],
     [4,1,5,2,3],
     [1,2,3,4,5],
     [3,5,2,1,4]]
```

Homework #4 - Statistics



```
[| ?- 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    1775 atoms     30993 atoms

Times              since start      since last

   user   time       0.010 sec       0.010 sec
   system time       0.027 sec       0.027 sec
   cpu    time       0.037 sec       0.037 sec
   real   time   69547.313 sec   69547.313 sec

(1 ms) yes
```



```
| ?- statistics(cpu_time, [SinceStart, SinceLast]).

SinceLast = 1
SinceStart = 42

yes
```

Homework #4

- Hint for plain version: First generate some heights and then test if it is correct
    - You can optimize it later
- Make sure everything compiles without warnings or errors on SEASnet servers
    - Before submitting, restart gprolog and try your solution once more - you might be calling rules that you renamed/deleted!
- Try to make your solution reasonably efficient
    - No need to put a lot of effort on this, but if your solution takes >10mins, it can be optimized...
    - Consider how to fail as early as possible
- Don’t use FD solver in your plain solution
    - E.g. comparisons with **#<** are not allowed
- **Do not use SWI-Prolog!**

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/#/

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

Questions?