CS131: Programming Languages

Boyan Ding

DIS 1E Week 6

Winter 2021

About TA

TA: Boyan Ding

Email: dboyan@cs.ucla.edu

Office Hours:

- Tuesday & Thursday 9:30–10:30am
- Zoom Link on CCLE

Discussion Section: 1E, Fridays 2:00 – 3:50pm

Course Announcement

- HW4 due: Next Friday, Feb. 19, 2021 11:55pm
  - Cutoff time one week later
- Homeworks should be submitted on CCLE, under “Assignments”

Agenda

- Prolog
- Homework #4

Prolog

Declarative Programming

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

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
  - Command: gprolog, available on SEASnet servers

How to program in Prolog?

- Facts and Rules are written into a file, e.g. myrules.pl
- In interactive Prolog environment, consult the rule file
  - Command: **[myrules].**
  - Or, use **[user].** to directly input rule in interactive environment.
- 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

- Variables: strings that start with a capital letter (or an underscore)
  - e,g, X, What, My_variable, ...
- Unification tries to find a way to fill the missing values
  - Binding variables to atoms

Prolog file:

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

Queries:

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

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

Rules

- Rules establishes relationship of multiple predicates
- Syntax: *conclusion* :- *premises*.
- Consider the statement: "All men are mortal":

Prolog file:

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

human(socrates)
```

Queries:

```
?- mortal(socrates).
yes

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

Rules

- Using multiple predicates in the premise
  - Comma (,) is the AND operator, semi-colon (;) is the OR operator


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

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

Equality in Prolog

- Three equality operators: **=**, **is**, **=:=**
  - "=" compares forms, does unification directly without evaluation
  - "is" does arithmetic evaluation on the right side and unifies
  - "=:=" evalues both sides


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

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

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

?- 7 is 5 + 2
yes

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

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

?- X =:= 4 + 3.
exception

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

Arithmetic comparisons

Mathematical Representation	Prolog
$x < y$	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

Lists

- Syntax: [val1, val2, val3, ..., valn]
- We can do unification on list
  - [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
  - Similar to pattern matching in OCaml

List: 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)*, with is true when *X* in in *List*


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

Queries:

```
?- exists(a, [a, b, c]).
yes
?- exists(a, [x, y, z]).
no
?- exists(X, [1, 2, 3]).
X = 1 ? a
X = 2
X = 3
```

Tracing in Prolog.

- **trace.** shows all the calls (use **notrace.** to turn off)

```
| ?- 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
```

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

Prolog's List library

- Some "functions" we will cover:
  - member (actually the same as "exists" above)
  - permutation
  - length
  - nth
  - maplist

List: member

- **From the manual**: “member(Element, List) succeeds if Element belongs to the List. This predicate is re-executable on backtracking and can thus be 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 known elements in the first argument:

```
?- 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: length

- **From the manual**: "length(List1, 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).
List = [_,_,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), [1,2,3]).
no
```

Generating a List with Constraints

- Problem: Generate a list of length N where each element is a unique integer between 1...N
- Approach: implement unique_list(List, N), that succeeds when List satisfies the constraint above
- 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 only has between(Min, Max, X)

Not provided by prolog

Generating a List with Constraints

- Implementation

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

!, fail is a combination to cause failure of current attempt and prevents backtracking

Finite Domain Solver

- Previous solution: enumerate all possible possibility to find answer
- ***Finite Domain Solver*** works in another way
  - Variable values are limited to a finite domain (non-negative integers)
  - Symbolic constraints are added to limit solution space
  - Solution is obtained by going through the final constrained space
- Often lead to more optimized solution with less code

Finite Domain Solver

- Let's solve the earlier problem with 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

FD Constraints

- FD constraints are written in different ways than ordinary ones
- Arithmetic constraints example:
  - FdExpr1 #= FdExpr2: equality
  - FdExpr1 #\= FdExpr2: inequality
  - FdExpr1 #< FdExpr2: less than
  - FdExpr1 #=< FdExpr2: less than or equal
  - FdExpr1 #> FdExpr2: greater than
  - FdExpr1 #>= FdExpr2: greater than or equal
- See official documentation for more built-in constraints
  - http://www.gprolog.org/manual/html_node/gprolog054.html

FD Constraints

- Note: constraints do not find a solution, they just limit the options
  - Solution is found with fd_labeling


```
?- X #= Y.

X = _#0(0..268435455)
Y = _#0(0..268435455)

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

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

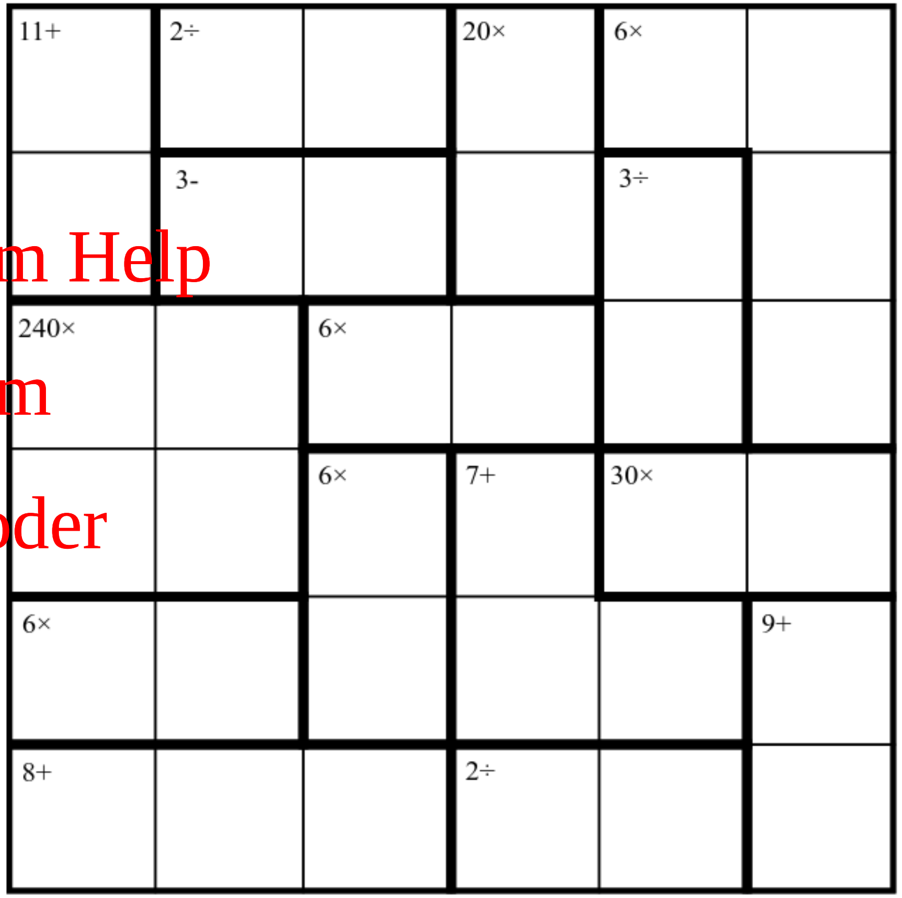
Homework #4

Homework #4: KenKen

N*N square filled with numbers 1..N, values not repeated in any row/column

A set of constraints on one or more contiguous cells

- Sum/Product is a certain value
- (A pair of cells') difference / quotient is a certain value



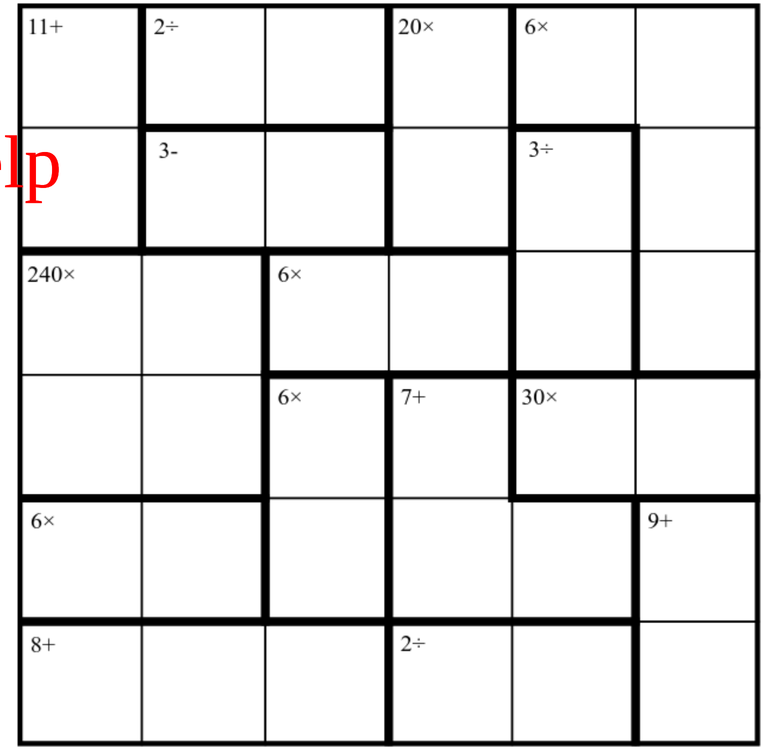
Homework #4

- Write Prolog code to solve KenKen puzzle
- Two implementations: one with FD solver, the other without (only using plain Prolog primitives)
  - Provide comparison of performance
  - Note: non-FD solver probably won't work well with larger grids, might try 4x4
- Additionally, design a proper API for no-op KenKen
  - Constraints only come with numbers, with the operators erased. They needed to be figured out during the solution process
  - Give a sample invocation (no need to implement).

Constraint Representation

- e.g., the “11+” in the upper-left corner
  - +(11, [1|1],[2|1])
- The whole constraint set:


```
[
 +(11, [[1|1], [2|1]]),
 /(2, [1|2], [1|3]),
 *(20, [[1|4], [2|4]]),
 *(6, [[1|5], [1|6], [2|6], [3|6]]),
 -(3, [2|2], [2|3]),
 /(3, [2|5], [3|5]),
 *(240, [[3|1], [3|2], [4|1], [4|2]]),
 *(6, [[3|3], [3|4]]),
 *(6, [[4|3], [5|3]]),
 +(7, [[4|4], [5|4], [5|5]]),
 *(30, [[4|5], [4|6]]),
 *(6, [[5|1], [5|2]]),
 +(9, [[5|6], [6|6]]),
 +(8, [[6|1], [6|2], [6|3]]),
 /(2, [6|4], [6|5])
]
```



Invoking your solution

- Refer to “Example” sections of the course website
- A sample call


```
| ?- kenken(
   4,
   [
    +(6, [[1|1], [1|2], [2|1]]),
    *(96, [[1|3], [1|4], [2|2], [2|3], [2|4]]),
    -(1, [3|1], [3|2]),
    -(1, [4|1], [4|2]),
    +(8, [[3|3], [4|3], [4|4]]),
    *(2, [[3|4]])
   ],
   T
), write(T), nl, fail.
[[1,2,3,4],[3,4,2,1],[4,3,1,2],[2,1,4,3]]
[[1,2,4,3],[3,4,2,1],[4,3,1,2],[2,1,3,4]]
[[3,2,4,1],[1,4,2,3],[4,3,1,2],[2,1,3,4]]
[[2,1,3,4],[3,4,2,1],[4,3,1,2],[1,2,4,3]]
[[2,1,4,3],[3,4,2,1],[4,3,1,2],[1,2,3,4]]
[[3,1,2,4],[2,4,3,1],[4,3,1,2],[1,2,4,3]]

no
```

Hints

- Properly describe the properties of solution
- The solution outline should probably look like
  - T is an NxN matrix
  - All values are between 1, 2, ..., N
  - Every row/column is different (or a permutation of [1,2,...,N])
  - Satisfies all constraints

Common for FD and plain

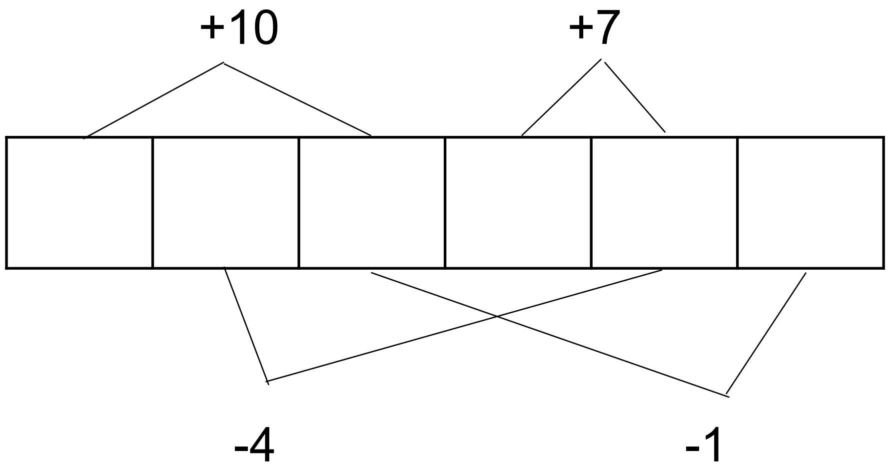
FD: directly leverage primitives, simple

plain: implement logic by hand

FD and plain should be similar, but with slightly different operators

Simplified problem

- Consider a 1-D "line" problem
  - A line of 6 cells, their values are all within 1, 2, ... 6, and each pair of cells cells contain different value.
  - A set of constraints
    - +(S, A, B): Cell A + Cell B equals to S
    - -(D, A, B): abs(Cell A - Cell B) equals to S




```
| ?- line([+(10,1,3),-(4,2,5),+(7,4,5),-(1,3,6)],L).

L = [6,1,4,2,5,3] ?
```

Simplified problem

- Solution

```
line_constraint(L, +(S, A, B)) :-
  nth(A, L, X),
  nth(B, L, Y),
  S is X + Y.

line_constraint(L, -(D, A, B)) :-
  nth(A, L, X),
  nth(B, L, Y),
  (D is X - Y; D is Y - X).

line(C, L) :-
  permutation([1,2,3,4,5,6], L),
  maplist(line_constraint(L), C).
```

```
fd_line_constraint(L, +(S, A, B)) :-
  nth(A, L, X),
  nth(B, L, Y),
  S #= X + Y.

fd_line_constraint(L, -(D, A, B)) :-
  nth(A, L, X),
  nth(B, L, Y),
  (D #= X - Y; D #= Y - X).

fd_line(C, L) :-
  length(L, 6),
  fd_domain(L, 1, 6),
  fd_all_different(L),
  maplist(fd_line_constraint(L), C),
  fd_labeling(L).
```

Homework #4: Statistics

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

SinceLast = 1
SinceStart = 42

yes
```


SinceStart = cpu time used since gprolog started

SinceLast = cpu time used since statistics was last called

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, first make sure that they are for GNU Prolog, not SWI-Prolog.

Questions?