Week 6 Discussion

CS 131 Section 1B

6 May 2022

Danning Yu

Announcements

- HW4 released, due 5/13
- HW1 grades released
- Midterm grading in progress
- Homeworks should be submitted on BruinLearn, under Assignments
- Before submitting
    - Make sure your code compiles on SEASnet server
    - Make sure your function signatures are correct
    - Follow all instructions and specifications
    - Do not submit files in a .zip unless told to do so
- Help and starter code from past TAs
    - https://github.com/CS131-TA-team

Prolog

Types of Programming

- Imperative programming
  - Using code to change the state of a program
  - Example: counter in a for-loop
  - Java, C, C++, Python, etc.
- Functional programming
  - Output of a function is solely determined by its inputs
  - No side effects
  - OCaml, Scheme, Python, etc.
- Declarative Programming
  - Describes what we want to achieve, not how to do it
  - SQL, regular expressions, Prolog, YAML config files
- Logic programming
  - Based on first order logic
  - Prolog: Programs defined using Facts, Rules and Queries

Intro to Prolog

- This course uses GNU Prolog: http://www.gprolog.org
  - Not SWI-Prolog, which is very different
  - Use `gprolog` to run a Prolog program
- Facts and Rules are written into a file with a `.pl` extension
- In interactive Prolog environment, add rules using either
  - `[myrules].` to load in `myrules.pl`
  - `[user].` to directly input rules in the interactive environment
- After adding in rules, can run queries in the interactive environment

Facts

- Define what is true in the database
  - Database: a collection of facts and rules
  - Must start with a lowercase letter

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

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 underscore
  - X, What, My_variable, ...
- Unification tries to fill in the missing values
  - Bind variables to atoms

Prolog file:

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

Queries:

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

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

Rules

- Establishes relationship consisting of multiple predicates
- Syntax: `conclusion :- premises.`
  - Implicit forall
  - `,` (comma) means AND operator
  - `;` (semicolon) means OR operator
  - `\+` means NOT operator
    - More precisely, not provable
- `%` for comments

Prolog file:

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

human(socrates)
```

Queries:

```
?- mortal(socrates).
yes

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

Rules

- Establishes relationship consisting of multiple predicates
- Syntax: `conclusion :- premises.`
  - Implicit forall
  - `,` (comma) means AND operator
  - `;` (semicolon) means OR operator
  - `\+` means NOT operator
    - More precisely, not provable

Prolog file:

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

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

Equality

- Three equality operators: `=`, `is`, `=:=`
- `=` compares forms
  - Does unification directly without evaluation
- `is` does arithmetic evaluation on right side and unifies
- `=:=` evaluates 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]`
- Unification can be done on lists
  - `[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
- Given `p([H | T], H, T).`
- Output of the following queries?
  - `p([a, b, c], a, [b, c]).`
  - `p([a, b, c], X, Y).`
  - `p([a], X, Y).`
  - `p([], X, Y).`

Searching Lists

- To check if a specific element is in a list:

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

```
Queries:

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

Debugging in Prolog with `trace.`

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

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

```
| ?- 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
```

Lists: `member/2`

- From the gprolog 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
```

Lists: `permutation/2`

- From the manual: “permutation(List1, List2) succeeds if List2 is a permutation of the elements of List1.”
- First argument must have known elements, or else stack overflow occurs

```
?- 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] ? ;
```

Lists: `length/2`

- 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
```

Lists: `nth/2`

- 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
```

Generating a List with Constraints

- 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
- Outline the constraints:

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

- `elements_between` and `all_unique` not provided by Prolog
  - We need to implement them

Generating a List with Constraints

- `elements_between`

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

- `all_unique`

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

Finite Domain Solver

- Prolog's by default enumerates all possible possibilities to find the answer
- Prolog's finite domain solver works differently
  - 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 leads to a more optimized solution with less code
  - Runs faster too

Finite Domain Solver for Unique Numbers

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

Finite Domain Constraints

- FD constraints are written differently
- Arithmetic constraints
  - `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

Finite Domain Constraints

- Constraints alone do not find a solution, they just limit the options
  - Use `fd_labeling/1` to get a solution

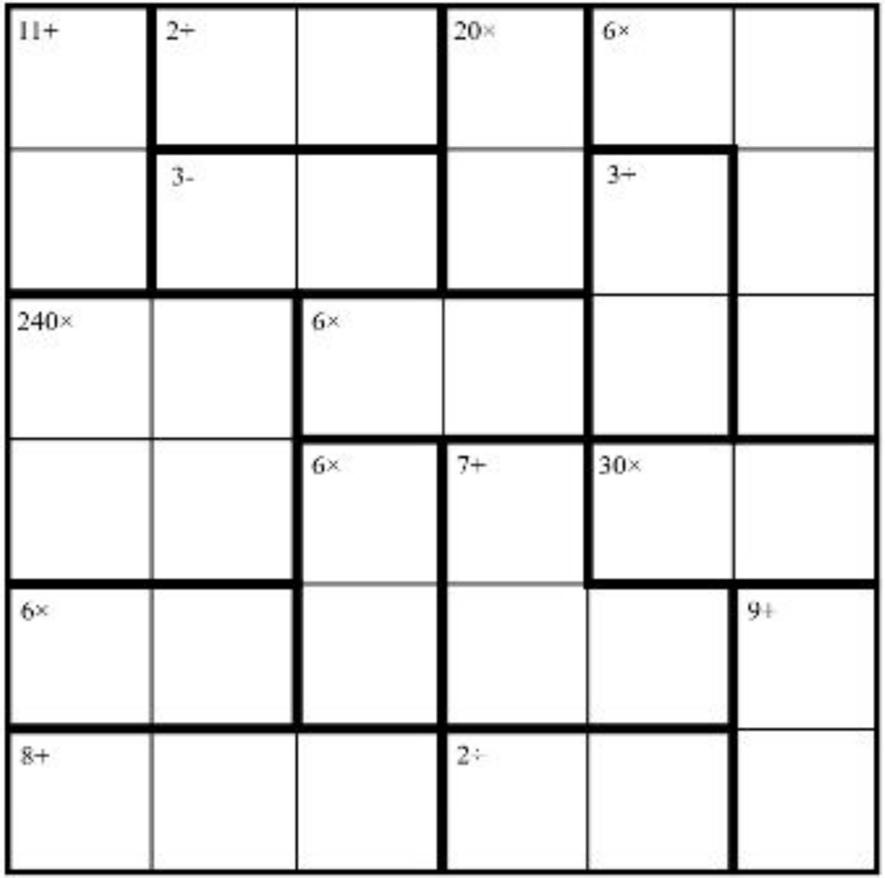
```
?- X #= Y.

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

HW4: KenKen

- N by N square filled with numbers 1..N, values not repeated in any row/column
- Constraints on one or more contiguous cells
  - Sum or product is a certain value
  - Difference or quotient is a certain value
    - Limited to two cells



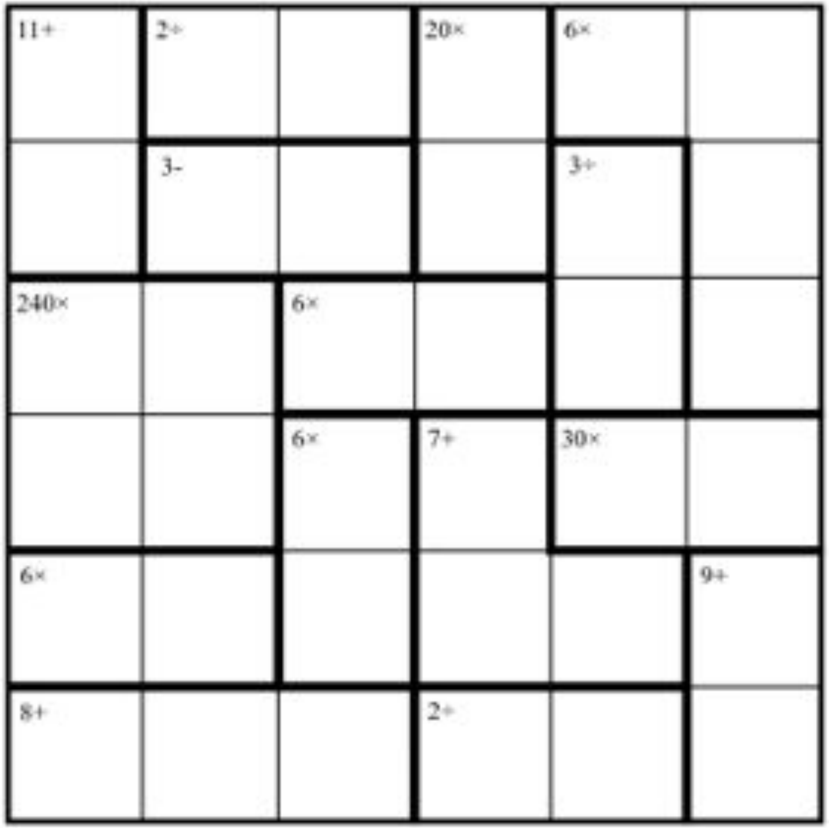
KenKen Prolog Solver

- Two implementations: one with FD solver, the other with only plain Prolog primitives
  - Provide comparison of performance
  - Note: non-FD solver probably won’t work well with larger grids, compare with 4x4 grid
- Design a proper API for no-op KenKen
  - Constraints only come with numbers, with the operators erased
  - Operators also need to be figured out
  - Give a sample invocation (no need to implement).

Constraint Representation

- Example: “11+” in the upper-left corner
  - `+(11, [[1|1],[2|1]])`
- The whole set of constraints

```
[
 +(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])
]
```



How to Run Your Program

- Refer to the examples section of the spec
- type `[kenken].` in the environment to load your code
- Sample call on the right

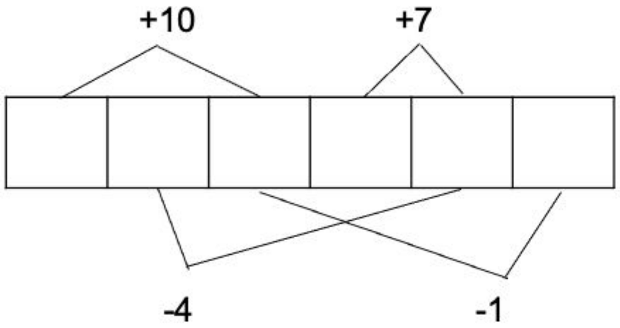
```
| ?- kenken(
  4,
  [
   +(6, [[1|1], [1|2], [2|1]]),
   *(96, [[1|3], [1|4], [2|2], [2|3], [2|4]]),
   -(1, [3|1], [3|2]),
   -(1, [4|1], [4|2]),
   +(8, [[3|3], [4|3], [4|4]]),
   *(2, [[3|4]])
  ],
  T
), write(T), nl, fail.
[[1,2,3,4],[3,4,2,1],[4,3,1,2],[2,1,4,3]]
[[1,2,4,3],[3,4,2,1],[4,3,1,2],[2,1,3,4]]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

HW Hints

- Properly describe the properties of solution
  - Don't code *how* it's achieved, instead write *what constraints* to satisfy
- Solution outline:
  - T is an NxN matrix
    - Same solution for both FD and plain
  - All values are between 1, 2, ... N
    - FD: use FD primitives; plain: implement logic by hand
  - Every row/column is different (or a permutation of [1,2,...,N])
    - FD: use FD primitives; plain: implement logic by hand
  - Satisfies all cell constraints
    - FD and plain should be similar, but with slightly different operators

A Simplified Problem

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




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

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

A 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).
```

Measuring Performance in Prolog

- Use the statistics/2 call
- `SinceStart` is CPU time used since gprolog started
- `SinceLast` is CPU time used since statistics was last called
- Units in ms

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

SinceLast = 1
SinceStart = 42

yes
```

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/#/

Important:

- **When looking for resources, make sure that they are for GNU Prolog, not SWI-Prolog!**

Thank You