Discussion 1C for CS 131 Programming Languages

Chi Zhang

Week 6, Winter 2021

Today

- Prolog
- Homework 4

Declarative Programing

- OCaml is for functional programing
- Java is for object-oriented programing
- Prolog is for …

Declarative Programing

- Describe *what* we want to achieve not *how*
- Examples: SQL, Prolog

Prolog

- Logic programing
- Programs defined by *Facts*, *Rules*, *Queries*
- Use GNU Prolog: http://www.gprolog.org/
    - **Not** SWI-Prolog
    - SEASnet servers: command *gprolog*

Prolog

- Facts and rules written into a file **filename.pl**
- In an interactive session, *consult* the file by **[filename]**
- Run queries in the interactive session
- Use . to end a statement
- Control + D to exit
- Sounds like a database

Facts

- Facts define what is true in the database
- Start with lowercase letters

Relations

- Facts consisting of one or more terms
- Closed-world assumption

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
    - My_variable, What, Who
- Unification binds *variables* to *atoms*

Rules

- Rules allow us to make conditional statements
- Syntax: conclusion :- premises
  - Conclusion is true if the premises are true

Rules

- Rules can contain multiple statements
  - , -> AND
  - ; -> OR

Equality

- Equality operator: =, is, =:=
  - = tries unification directly, is evaluates the right-hand side and unifies, =:= evaluates both sides

Arithmetic

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

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

Recursion

Lists

- Syntax: ***[val1, val2, val3, …, valn]***
- We can do unification with head and tail:
  - [1, 2, 3, 4] = [A | B]
  - [1, 2, 3, 4] = [A, B | C]
  - [1, 2, 3, 4] = [A, B, C, D]

List Searching

- How to check if a specific element is in a list?

Trace

- Used for debugging the code
- ***trace*** to turn on, ***notrace*** to turn off

List Functions

- Construction
- Removal

Built-in

- Append: append(list1, list2, list12) concatenates two lists
- Member: member(elem, list) if elem is a member of list
- Permutation: permutation(list1, list2) if list2 is a permutation of list 1
- Length: length(list, len) if length of list is len
- Nth: nth(n, list, elem) if the nth element of list is elem
- Maplist: maplist(cond, list) if every elem satisfies the condition

Cuts

- !, always succeeds but cannot be backtracked
- Why?
  - Pruning: Cutting off useless branches of the search tree

Fail

- fail is a special symbol that will immediately fail when Prolog encounters it as a goal
- That may not sound too useful, but remember: when Prolog fails, it tries to backtrack
- Thus fail can be viewed as an instruction to force backtracking
- When combined with cut …

Generate a List with Constraints

- Let’s say we want to find a list of length N where each element is a unique integer between 1 to N

Questions?

Finite Domain Solver

- Finds variable values that fulfill given constraints
- Variable values are limited to a finite domain (non-negative int)
- Less code, optimized solution

Finite Domain Solver

- Let's say we want to find a list of length N where each element is a unique integer between 1 to N

Finite Domain Constraints

- Arithmetic constraints:
  - FdExp1 #= FdExp2: equal
  - FdExp1 #\= FdExp2: different
  - FdExp1 #< FdExp2: less than
  - FdExp1 #=< FdExp2: less than or equal to
  - FdExp1 #> FdExp2: greater than
  - FdExp1 #>= FdExp2: greater than or equal to
- Manual: http://www.gprolog.org/manual/html_node/gprolog054.html

Sudoku with Finite Domain Solver

- 4x4 Sudoku
- fd_domain(List, Min, Max)
- fd_all_different(List)

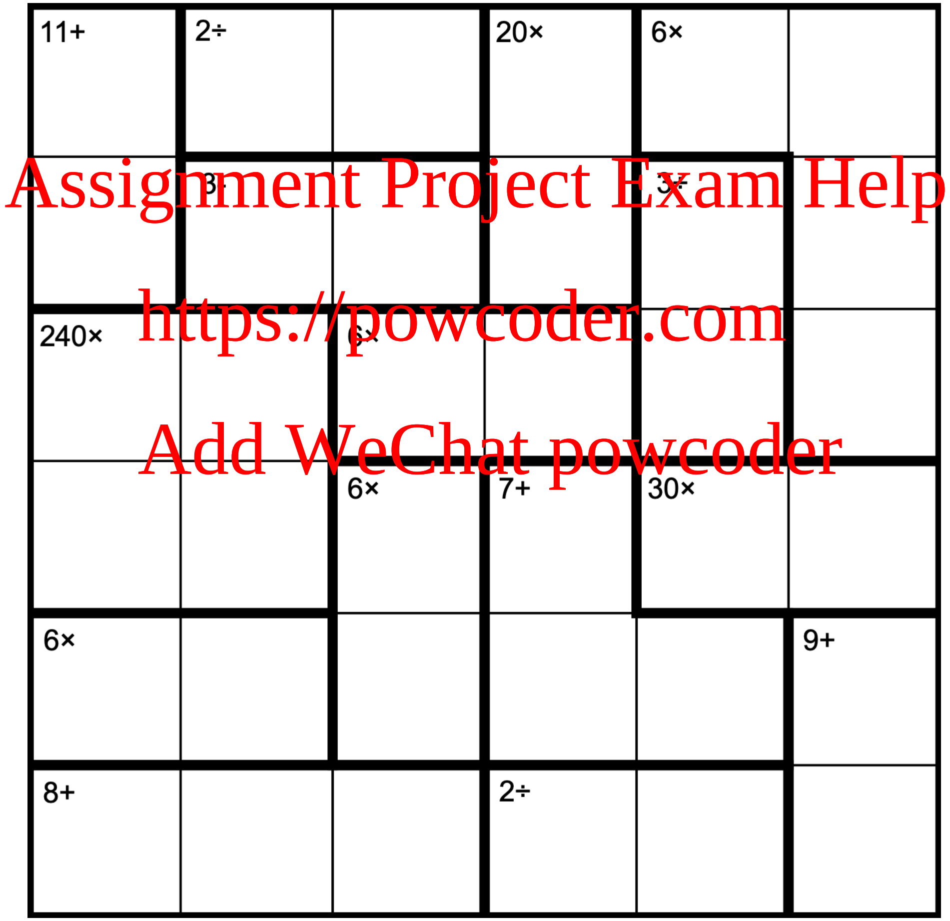
	2		
1			
			4
		1	

Questions?

Homework 4

- KenKen Solver
- NxN square
- Each row / column is filled with 1 to N, (no repeat)
- Additional constraints

Homework 4



Homework 4

- Two implementations: one with FD and one without
  - Compare performance
  - Non-FD solver won't work will with large grids. Testing with 5x5 is enough
- Design a good application programming interface for no-op KenKen

Homework 4

- Statistics
- SinceStart = cpu time since gprolog was started
- SinceLast = cpu time since statistics was called


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

SinceLast = 1
SinceStart = 42

yes
```

Homework 4

- plain_kenken and kenken work in “opposite” directions
  - plain_kenken sets some values to positions and checks if they work
  - kenken first sets all constraints and finds values
- Try to make the code reasonably efficient
  - Try not to run for more than 10min
  - Consider how to fail early
- Do not use FD for your plain solution
- Do not use SWI-Prolog

Resources

- GNU Prolog manual: http://www.gprolog.org/manual/gprolog.html
- Prolog wikibook: https://en.wikibooks.org/wiki/Prolog

Questions?