CMSC 330, summer 2016

Organization of Programming Languages

Project 5 - Prolog Programming

Due 11:59pm Mon, Jul 18, 2016

Errata:

- Updated on 04/13: Project description is modified. Re-download the project files.

Introduction

For this project you will need to implement a number of functions in Prolog that together can be used to find solutions for mazes. This project will provide experience dealing with logic, recursion, lists, and other features in Prolog.

Getting Started

Download the following archive file p5.zip and extract its contents.

Along with files used to make direct submissions to the submit server (submit.jar, .submit, submit.rb), you will find the following project files:

- Your Prolog program - logic.pl
- Public tests
  - publicRecursion1.pl
  - publicRecursion2.pl
  - publicRecursion3.pl
  - publicMaze1.pl
  - publicMaze2.pl
- Expected outputs for public tests
  - publicRecursion1.out
  - publicRecursion2.out
  - publicRecursion3.out
  - publicMaze1.out
  - publicMaze2.out
- Public test driver - goTest.pl

The logic.pl file you downloaded contains a number of utility functions, and comments describing the functions you are required to implement.

Note that you must implement your functions with the exact parameters specified, or else the submit server tests will fail.

Running Public Tests

The public tests are set up as a number of Prolog programs that will call functions from your logic.pl code and test them with different inputs. To execute a public test, you need to load both logic.pl and the public test file into the Prolog interpreter, then call the appropriate public test function for each logic.pl function you were required to implement. The public test functions are:

- public_ackermann
- public_prod
- public_fill
- public_genN
- public_genXY
- public_flat
- public_isprime
- public_inlang

- public_stats
- public_validPath
- public_findDistance
- public_solve

Here's an example of how to run a public test manually:

```
swipl                                     % start prolog
?- working_directory(C,'path to p5'). % go to p5 directory
                             % start here if using swipl-win.exe
?- ['logic.pl'].                          % load your code
?- ['publicMaze1.pl'].                    % load public test
?- maze1_public.                        % run public test
etc...
```

On Windows machines, opening the logic.pl file with swipl-win.exe will bring up a window running the Prolog interpreter in the directory containing logic.pl, so it is not necessary to start swipl or call the function working_directory manually.

Alternatively, you may run all of the public tests at once using the goTest.pl public test driver provided. It will load logic.pl and all the public tests, then run all public tests at once.

```
swipl                                     % start prolog
?- working_directory(C,'path to p5'). % go to p5 directory
?- ['goTest.pl'].                         % load test driver
                             % start here if using swipl-win.exe
?- run.                                   % run all public tests
```

Prolog Library Functions Allowed

For this project you should write most code yourself, and only use Prolog's built-in and library functions where absolutely necessary. You are not allowed to use any library or built-in functions unless they are explicitly listed as permitted functions. The only built-in function that you are allowed to use for this project are:

Type	Built-in Functions
Arithmetic	+, -, *, div, mod, <, =<, >, >=, is, =:=, =\=, floor, float, sqrt
Logic	==, =, \==, \=, \+
Lists	[H|T], [H1,H2|T], [H1,H2,H3|T], etc.
List Utilities	member(X,L), append(X,Y,R), sort(X,R)
Cut	!
Collecting Solutions	findall(X,Y,R), setof(X,Y,R)

Since many functions you need to implement are similar to those from previous projects, you may find it useful to examine your previous solutions when writing your solution in Prolog.

Part 1: Recursion

Write the following recursive functions:

Name	Parameters	Example

ackermann(M,N,R)	M=int
	N=int
	• R=the ackermann function on M and N
	• M and N will always be given
		?- ackermann(0,1,R).
		R=2.
		?- ackermann(2,3,R).
		R=9.
		?- ackermann(3,4,R).
		R=125.

prod(L,R)	L=list of ints
	R=product of elements of L
	• R=1 if L=[]
	• L will always be given
		?- prod([1,2,3],R).
		R=6.
		?- prod([],R).
		R=1.

fill(N,X,R)	N=int	?- fill(4,2,R).

	X=int
	R=list containing N copies of X
	• R=[] if N=0
	• N will always be given
	• Either X or R will be given
		R=[2,2,2,2].
		?- fill(4,X,[2,2,2,2]).
		X=2.

genN(N,R)	N=non-zero positive int
	R=int values between 0 and N-1, inclusive, in ascending order
	• N will always be given
		?- genN(2,R).
		R=0;
		R=1.

genXY(N,R)	N=non-zero positive int
	R=pairs [X,Y], where X & Y are values between 0 and N-1, inclusive,
	generated in ascending lexicographic order.
	• N will always be given
		?- genXY(2,R).
		R=[0,0];
		R=[0,1];
		R=[1,0];
		R=[1,1].

flat(L,R)	L=list
	R=elements of L concatenated together, preserving relative order,
	first placing non-list elements in a list if necessary
	• R=[] if L=[]
	• L will always be given
	• Only removes one level of list, unlike flatten/2
		?- flat([[1],[2,3]],R).
		R=[1,2,3].
		?- flat([1,[2,3]],R).
		R=[1,2,3].
		?- flat([[1,[2]],3],R).
		R=[1,[2],3].

is_prime(N)	N=int Is `true` if the integer *N* is a prime number. • A simple algorithm for primality checking is to start with the axioms that 2 and 3 are prime numbers (1 is not). Then, an arbitrary number N greater than 3 is prime iff N is not divisible by D, for all D from sqrt(N) up to N-1. You will need to implement a helper function to do this iteration. You will find `=\=`, `float`, and `sqrt` functions helpful. Note: your algorithm should aim to reject large non-primes quickly, or you might experience timeouts. • N will always be given	?- is_prime(3). true. ?- is_prime(4). false. ?- is_prime(31). true.

in_lang(L)	L=list of atoms `a` and `b`
	Is `true` if the list *L*, viewed as a string, is contained in the language S defined by the following CFG:
	`S -> T | V`
	`T -> UU`
	`U -> aUb | ab`
	`V -> aVb | aWb`
	`W -> bWa | ba`
	Put another way, the language S is specified as follows:
	`S = {a^n b^n a^m b^m | n,m >= 1} U {a^n b^m a^m b^n | n,m >= 1} .`
	• L will always be given
		?- in_lang([a,a,b,b,a,b]).
		true.
		?- in_lang([a,a,a,b,b,a,a,b,b,b]).
		true.
		?- in_lang([a,a,a,b,b,a,b,b,b]).
		false.

Part 2: Maze solver

Maze descriptions

For this project, the mazes you will compute with will be given as Prolog databases. In particular, you will be given three kinds of *facts*

- **maze**(*N,SX,SY,EX,EY*). This fact indicates that:
  - The height and width of the maze is *N* cells,
  - The default starting position is *SX,SY*, and

  - The ending position is *EX,EY*.

- **cell**(*X,Y,Dirs,Wts*). A fact of this form describes a cell in the maze. In particular, it says that the cell at position *X,Y,* has open walls as described by *Dirs*, the list of directions. More precisely:
  - The list *Dirs* will contain at most one of each of the atoms **u**, **d**, **l**, and **r**, which designate openings going up, down, left, or right, respectively.
  - Recall from project one that the coordinate system places 0,0 in the upper left corner of the maze.
  - The *Wts* component of the fact indicates the weights granted to paths following the respective direction. That is, each element in *Dirs* has a corresponding weight in *Wts*.

  As an example, the fact **cell**(1,0,[**r**,**d**],[16.6, 0.89]) indicates that the cell at 1,0 has two open walls: one leading to the right (to cell 2,0) with weight 16.6, and one leading down (to cell 1,1) with weight 0.89.

- **path**(*N,SX,SY,Dirs*). This fact describes a path named *N* (a string) through the maze starting at position *SX,SY* and following the directions given by *Dirs*. For example, the fact **path**('path1',0,3,[**u**,**r**,**u**,**l**,**u**]) indicates that there is a path 'path1' that starts at 0,3 and follows the given directions to end up at position 0,0.

Based on these maze facts, you need to implement the following functions for solving a maze.

Name	Parameters	Example

stats(U,D,L,R)	U,D,L,R=number of cells with openings up, down, left, and right.	?- stats(U,D,L,R).
		U = D, D = 8,
		L = R, R = 7.

validPath(N,W)	N=name of valid path (only goes through openings)
	W=float value for weight of path (rounded to 4 decimal places)
	• Return valid paths in same order as in database
	• Use round4(X,Y) :- T1 is X*10000, T2 is round(T1), Y is T2/10000.
	• Apply round4() to final float weight, not intermediate sums.
		?- validPath(N,W).
		N = path1,
		W = 99.9958;
		N = path2,
		W = 103.779.

findDistance(L)	L=list of coordinates of cells at distance D from maze start • Elements of L are in form [D, [[X1,Y2],[X2,Y2],...]] • Values of D range from 0 to D, in ascending order • D=distance of cell furthest from start • Cell coordinates [X,Y] are in lexicographic order	?- findDistance(L). L = [[0, [[0, 3]]], [1, [[0, 2]]], [2, [[1, 2]]], [3, [[1, 1], [2,2]], ..., [6, [[3, 2]]].
solve	• True if maze is solvable, fails otherwise.	?- solve. true.

This is obviously a terse description; if you have further questions (once you have completed the rest of the assignment), ask the course staff.

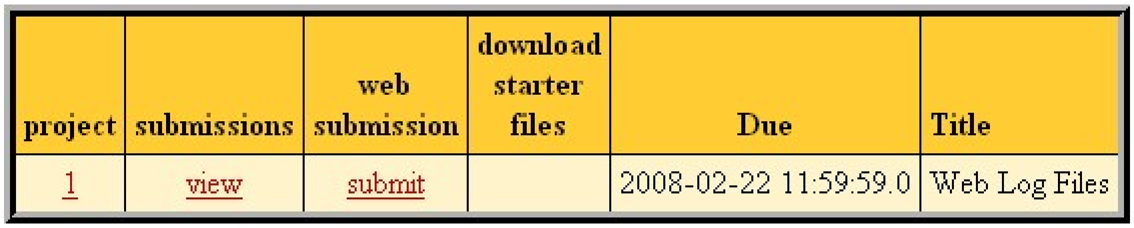
Hints

- Unlike previous projects, you may be able to rely on Prolog's backtracking to find multiple possible solutions through multiple queries.
- The Prolog functions *findall* will collect all results from a function using backtracking. The function *setof* will, in addition, sort the result and remove duplicates. These functions are used in the public tests.

Submission

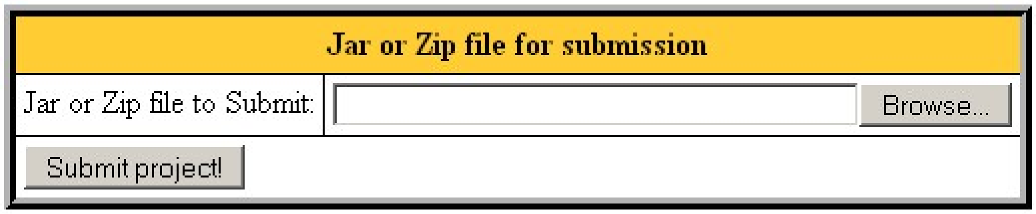
You can submit your project in two ways:

- Submit your file logic.ml directly to the submit server by clicking on the submit link in the column "web submission".



project	submissions	web submission	download starter files	Due	Title
1	view	submit		2008-02-22 11:59:59.0	Web Log Files

Next, use the submit dialog to submit your logic.ml file.



Jar or Zip file for submission	
Jar or Zip file to Submit:	

Select your file using the "Browse" button, then press the "Submit project!" button.

- You may also submit directly by executing a Java program on a computer with Java and network access. Use the submit.jar file from the archive p5.zip, To submit, go to the directory containing your project, then either execute submit.rb (preferred method) by typing:

```
ruby submit.rb
```

  or use the java jar directly using the following command:

```
java -jar submit.jar
```

  You will be asked to enter your class account and password, then all files in the directory (and its subdirectories) will be put in a jar file and submitted to the submit server. If your submission is successful you will see the message:

```
Successful submission # received for project 5
```

Academic Integrity

The Campus Senate has adopted a policy asking students to include the following statement on each assignment in every course: "I pledge on my honor that I have not given or received any unauthorized assistance on this assignment." Consequently your program is requested to contain this pledge in a comment near the top.

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus---please review it at this time.

Web Accessibility