CSCI.4430/6430 Programming Languages Fall 2017
Programming Assignment #3

*This assignment is to be done either **individually** or **in pairs**. **Do not show your code to any other group** and **do not look at any other group's code**. Do not put your code in a public directory or otherwise make it public. However, you may get help from the TAs or the instructor. You are encouraged to use the LMS Discussions page to post problems so that other students can also answer/see the answers.*

Part 1: Rat in a Maze

For this assignment you are going to help a lab rat, Einstein, make its way through a maze in order to find a path to his favorite food, cheese. In test type a, Einstein will simply find a valid path to the cheese that is not blocked by any walls.

However, simple mazes are trivial for Einstein, so his researchers have decided to challenge him by adding a number of buttons to the maze that he needs to press before awarding him with his precious cheese.

The researchers want Einstein to be able to press every single button in the maze before they put the food at the goal. In test type b, Einstein can press buttons in any order. However, in test type c, Einstein has to press them in a particular order (button 1 -> button 2 -> button 3 -> ...). Note that in part 1, if Einstein moves over a coordinate that has a button, he will press the button no matter what.

Also, note that test type c is a more specific path condition, so an answer for c is also a valid answer for b and a. So one strategy would be to complete a solution for test type c first, which would always satisfy test type b and a as well. However, it may be easier to implement type a first, and then build upon your solution to account for buttons.

You will be provided with the following information to help Einstein get through the maze:

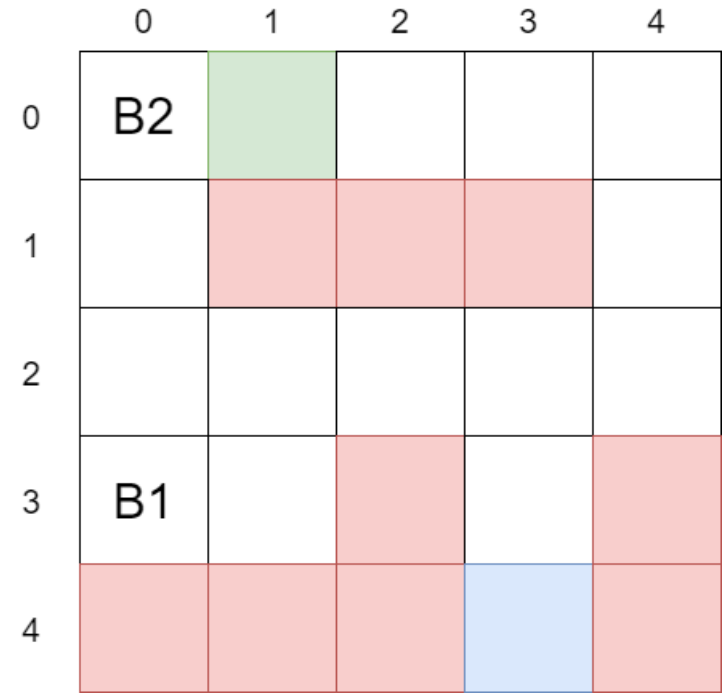
- Size of the maze (width and height)
- Locations of each of the 'Walls' which are areas that Einstein cannot walk through
- Locations and ID of each of the buttons
- The number of buttons
- Start location
- Goal (where the cheese will be after buttons are pressed)

You will keep track of Einstein's path, and once he gets to the goal, you will write his path to a file called `path-solution.txt` if completed in Prolog. Oz solutions will print the path using Browse.

Each line in your solution should be a coordinate in Einstein's path, beginning with the starting point and ending with the goal point. ie:

```
[3,4]
[3,3]
[3,2]
[2,2]
[1,2]
[0,2]
[0,3]
[0,2]
[0,1]
[0,0]
[1,0]
```

This would be a solution for the following maze. Your path doesn't have to be the same, any valid path that solves the puzzle correctly will be a correct solution. You can also assume that the researchers **will never give him an impossible maze** (in other words, the cases we will test you on will always be solvable for all of the tests).

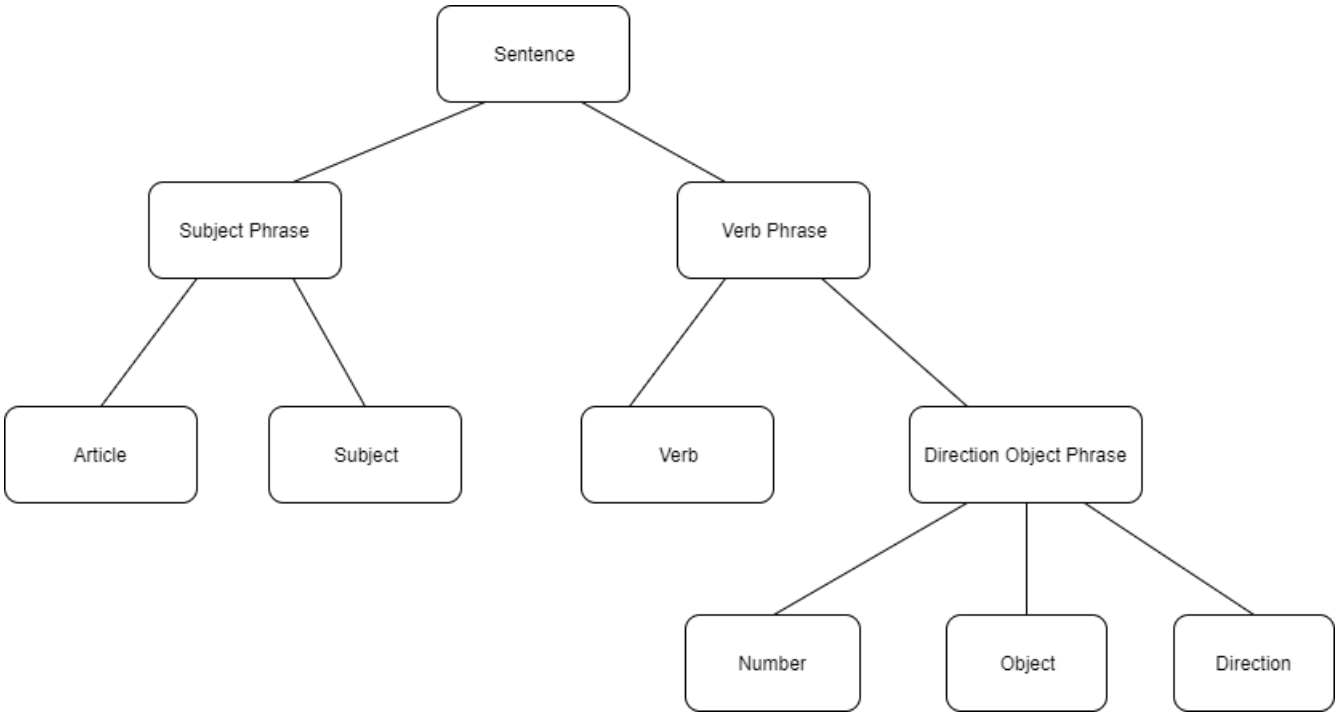


Where blue represents Einstein's starting point, green represents the goal, and B1 and B2 are the buttons with their IDs. This solution would be valid for test type a, b or c.

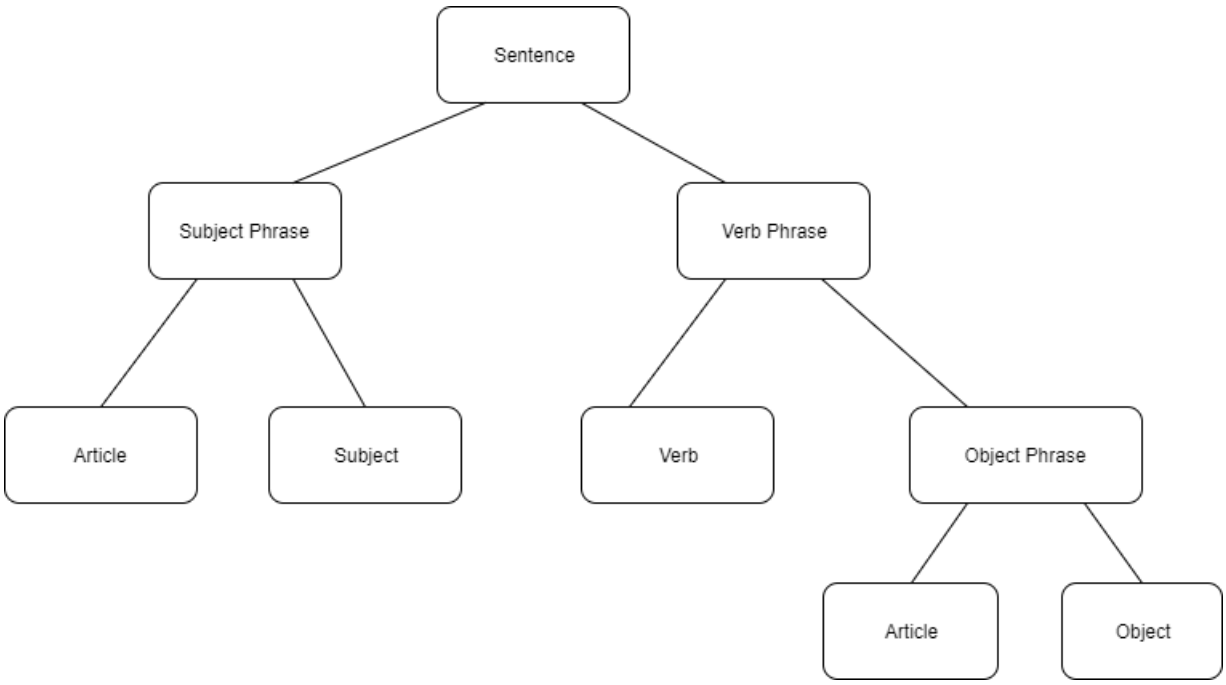
It is recommended that you keep track of the locations that Einstein has been to, so that he doesn't go back over the same point over and over again when trying to get to a button or the goal. Once Einstein reaches a button however, he may have to travel to a location he has been before, so you can clear out the visited list every time Einstein touches a button. Solutions that take too long to run because they do not implement this will be penalized.

Part 2: Natural Language Parsing

For this portion of the assignment, your program should be able to read in an input containing sentences about Einstein's movements in natural language. For Prolog, you will be given a text file containing sentences about what direction Einstein moved, and how many spaces he moved. In Oz you will be provided with a list of sentences as lists of atoms. Sentences will be provided in the following 2 language structures:



OR



You must parse the language in this structure, not simply look for keywords. Solutions that simply use keyword search for directions will be penalized.

Your solution should be able to determine if the sentence is a valid sentence in these language structures, and if so, you will check the maze to determine if it is a valid move or not.

Here are some examples of the kinds of sentences you will need to parse:

- the rat ran 2 squares up
- it moved 3 squares left
- einstein moved 1 cell down
- the rat pushed the button
- a rodent scurried 3 squares up
- down einstein scurried squares 3
- he moved 1 cell left

For each sentence, you will parse it and determine if it is a valid sentence in this structure. If not, you will output "Not a valid sentence" and move to the next sentence. If it is a valid sentence, then you will check if it is a valid move in the maze. If it is a valid move, move Einstein to that location, output "Valid move", and move on to the next sentence. If it is not valid, then output "Not a valid move" and then stop parsing the sentences.

Part 2 is independent of the test type a, b and c paths from part 1. So, you will not need to check if Einstein got to the goal or not, only if the move is valid or not valid in the maze. In contrast to part 1, Einstein may choose to not press a button if he passes over that space. In the case that the sentence says that he pushed a button, you must check to see if there is a button on that location, or else it is not a valid move.

The sentences will be lower case so you don't need to worry about capitalization. Also, note that the words "einstein", "he" and "it" don't have an article so you could consider them a subject phrase directly. The words "square" and "cell" need to be accepted as singular or plural ("squares"/"cells"). However, you do not need to check for the correct usage of plural/singular. For example, the sentence "a rodent scurried 3 square up" will still be considered as valid. You can assume that if the sentence is in the first language structure, then the rat is moving so you will check if his path is valid. If the sentence is in the second language structure, then you can assume that the rat is pressing a button (check for if a button is on the current location).

If we use the example maze from part 1, this would be the correct output:
Valid move
Valid move
Valid move
Valid move
Valid move
Not a valid sentence
Not a valid move

Prolog solutions will write this output to a file called `NL-parse-solution.txt`. Oz solutions will print this output via Browse.

Note for Prolog Programmers:

Your code for part 1 should be placed in a file called `mazeSolver.pl`. You will use the `:- module(mazeInfo, [info/3, wall/2, button/3, num_buttons/1, start/2, goal/2]).` command to read in the configuration of the maze, and use each of the defined predicates to find the solution. The info predicate will have the width as the first parameter, the height as the second parameter and the type of test as the third parameter (a, b or c). The wall predicate will have the x coordinate as the first parameter and the y coordinate as the second parameter. The button predicate will have the first parameter as the x coordinate for the button, the second parameter as the y coordinate for the button, and the third parameter is the ID of the button, which will help you determine in what order to press the buttons in test type c. The num_buttons predicate will simply be the number of buttons in the maze. And the start/goal predicates give the coordinates of the start and goal points in the maze. You can assume that a button and a wall will not be in the same location, and everything will be in the bounds of the maze size. Please take a look at mazeInfo.pl to see the format. The walls and buttons will be written in increasing x and then increasing y order in the mazeInfo.pl file.

The code for part 2 should be placed in a file called `NLParser.pl` and read sentences from a file called `NL-input.txt` The input for the example described in part 2 can be found here

Here is some starter code to read line by line from the file.

For both part 1 and part 2, you must write your code such that it has a `main :-` that upon calling it runs the program and creates the necessary solution text files. For example, your code should produce the solution text files if run with the command `swipl -q -f mazeSolver.pl -t main` and `swipl -q -f NLParser.pl -t main` for part 1 and 2 respectively.

Notes for Oz Programmers:

You should remove any other Oz installations and use Oz 1.4.0 downloadable from http://sourceforge.net/projects/mozart-oz/files/v1/ and Emacs downloadable from http://gnu.mirror.vexxhost.com/emacs/, to enable the Oz programming interface to find the proper emacs running command such as runemacs.exe, you need to create an environment variable OZEMACS and set its value to for example F:\Programs\emacs-21.1\bin\runemacs.exe

Your code for part 1 should be placed in a file called `mazeSolver.oz`. You will be given the configuration of the maze with a set of functions to be placed at the top of your file. See the starter code here which contains the configuration of the maze example mentioned in part 1. You will be provided with an "info" function that returns a 3 element tuple. The first element is the width of the maze, the second element is the height of the maze, and the third element is the type of test (a, b or c) as an atom. The "Wall" function gives a choice block which each line representing a coordinate of a wall. The coordinates provided are tuples with the first element being the x coordinate, and the second element is the y coordinate. The "Button" function provides the locations of the buttons in the same manner as "Wall", except it comes with another element in each tuple which represents the ID of the button, which will help you determine in what order to press the buttons in test type C. The "NumButtons" function simply returns the total number of buttons in the maze. Finally, the "Start" and "Goal" functions return the coordinates of the start/goal points as a 2 element tuple.

The code for part 2 should be placed in a file called `NLParser.oz`. See the starter code for part 2 here. The maze configuration will be provided in the same manner as part 1. You will be given an "Input" variable that will be a list of sentences. Each sentence is a list of atoms with each atom being a word in the sentence. So the "Input" is a list of lists of atoms. You will pass this to a "ValidMoves" function that will ultimately print the solution using Browse.

See CTM Chapter 9 for relational programming techniques. Also see the logic programming section in the OZ documentation to see how to solve logic problems.

Due date and submission guidelines

Due Date: Thursday, 11/30, 7:00PM

Grading: The assignment will be graded mostly on correctness, but code clarity / readability will also be a factor (make sure to comment your code!).

Submission Requirements: Please submit a ZIP file with your code, including a README file. README files must be in plain text; markdown is acceptable. Your ZIP file should be named with your LMS user name(s) and chosen language as the filename, either userid1_oz.zip (or userid1_pl.zip) or userid1_userid2_oz.zip (or userid1_userid2_pl.zip). Only submit one assignment per pair via LMS. In the README file, place the names of each group member (up to two). Your README file should also have a list of specific features / bugs in your solution.

Do not include unnecessary files. Test your archive after uploading it. Name your source files appropriately.