CSE550: Homework 2

Question 1: Grammars

You will build an interface to a student database. You should be able to enroll yourself in any course, and withdraw from any course that you have enrolled in. You should also be able to have the system list the courses that you are enrolled in, and find out some basic information about each course.

You could build a finite state dialogue system in which it takes you several steps to specify your command. Instead, build a system where you can specify the entire command in a single utterance. For instance, you should be able to say the following types of sentences.

“enroll in CSE550”
“withdraw from CSE503”
“list available courses”
“who is teaching ECE543”
“what is CSE581 about”
“list my courses”

Build a dialogue model with just a start state, end state and two intermediate states. The first intermediate state should issue a prompt such as “how can I help you?” For this state, specify a regular grammar that will allow you to issue the above commands. Make the grammar concise by using a subrule that specifies available courses.

Have the second state just repeat back what was recognized.

Prepare a list of 20 different sentences that are in your grammar and that reasonably covers everything that it allows. Check how well the system did in recognizing each one.

Writeup 1.1  Hand in your grammar. What percentage of sentences was it recognizing accurately? Explain whether you did anything to improve recognition of the course names and to improve the speech synthesis of them.

Question 2: Adding in Semantic Interpretation

For the above system, make up a semantic representation language. Make sure that the representation language is simple enough that the terminal substitution formalism can be used to construct it. Do not make the semantics any more complicated than you need for the above sample of sentences.

Writeup 2.2  Give the semantics for each of the six sentences above.

Writeup 2.3  Augment your regular grammar grammar with the terminal substitution formalism so that the recognizer outputs your semantic representation.

Question 3: A Complete System

Rather than have the second state of your system simple repeat back what was said, turn this into a complete working system. To keep this simple, just implement the ‘enroll’, ‘withdraw’ and ‘list my courses’ commands. You should be able to have the following dialogue.
The system should issue an error message if you try to enroll in a course you are already enrolled in, or try to withdraw from a course you are not in.

This exercise will force you to learn some Tcl programming, which you will need for later assignments.

You might need to be aware of local versus global variables in Tcl. You might also need to be aware of associative arrays in Tcl. There is lots of free documentation about Tcl on the web, including tutorials.

To get you started, you might want the following line of Tcl to declare all available courses in your start state:

```
set courses {CSE550 ECE543 CSE581 ...}
```

Note that this will be redundant with the list of courses in your speech recognition grammar.

You could use the associative array name `taking` to be true of a course if the user has enrolled in it. In your start state, To begin with, you can set taking to be 0 for each course, with the following.

```
foreach course $courses {
    set taking($course) 0
}
```

To check if a person is taking CSE550, you can just do the following:

```
if {$taking($course) == 1} {
    ...
}
```

**Writeup 3.4** Hand in the Tcl code for each state.

**Question 4: Breath First Search**

Before doing a breadth-first search bottom-up parser, you will first make a breadth-first search algorithm for finding a path through a maze. First, you will make a Tcl program independently of the CSLU toolkit. Here is a simple ‘hello world’ program. Put this code into a file called “hello.tcl,” and double click on it in Windows.

```
wm withdraw .
console show
puts "Hello world"
```

The window that is displayed is a Tcl console window, and you can quit this Tcl console by typing in ‘exit’. Note that because this is a console, you can also type in an Tcl command that you would like.

Now, consider the maze below.
The first step is to represent the maze internally. For each cell, set up a list of the cells that it is connected to. This could be done using code like the following (but there are better ways of doing it).

```
set connected(1) {2 5}
set connected(2) {1 3}
```

The pseudo code for the breadth first search is as follows.

Create a path which just has the start state
Make this path the only member of the list of alternatives to be explore
While list of alternatives is not empty and not done
  Set firstpath to be the first path from the list of alternatives
  Update alternatives so it doesn’t include the first path
  Set last to be the last member of firstpath
  For each cell connected to the last member
    Create newpath with cell as at the end of firstpath
    If cell is 16
      Display path
      exit
    else
      Add newpath to end of list of alternatives
```

For debugging your code, feel free to throw in ‘puts’ statements. Note that Tcl normally does not refresh the screen until all activity is done. You can force it to update the screen with the ‘update’ command. You can also add in a delay by using ‘after nnn’ where nnn is the number of milliseconds you want it to pause.

For debugging, rather than using puts statements, there is a debugger that you can use. The debugger is very good at pin-pointing syntax errors in your tcl code (which the normal tcl interpreter does not do), as well as lets you step through your code and shows you the values of all variables. More information about this is in the tcl web page that is listed next to the link to this homework from the class web page.

To speed things along, I have filled given you the Tcl code that corresponds to the above pseudo code.

```
set path [list 1]
set alternatives [list $path]
while {([llength $alternatives] > 0)} {
  set firstpath [lindex $alternatives 0]
  set alternatives [lrange $alternatives 1 end]
  set last [lindex $firstpath end]
  foreach cell $connected($last) {
    set newpath [linsert $firstpath end $cell]
    if {$cell == 16} {
      puts "Answer is $newpath"
      update
      after 10000
      exit
    } else {
      puts "Path is $newpath"
      lappend alternatives $newpath
    }
  }
}
```
Note that \texttt{index} and \texttt{range} are all functions that do not modify their parameters. This differs from \texttt{append}, which modifies its first parameter.

One thing to note about Tcl is that it likes to treat everything like a list. The following all set $l$ to the same value.

\begin{verbatim}
set l [list a b c]
set l {a b c}
set l "a b c"
\end{verbatim}

But the following do not.

\begin{verbatim}
set a 1
set b 2
set c 3
set l [list $a $b $c]
set l {$a $b $c}
set l "$a $b $c"
\end{verbatim}

Try them out.

Tcl likes to treat everything as a list, even if it is not. Try out the following.

\begin{verbatim}
set l a
set m {a}
if {$l == $m} {
  puts 1
}
\end{verbatim}

Tcl coerces non-lists into a singleton list when needed. Hence, it is impossible to distinguish a list with one item in it, from just the item itself.

Okay, now back to the breadth-first search code. The code above exits when it finds a path from the start cell to the end cell. Now we want to change the code so that it returns all paths from the start to the end cell. If you just remove the exit, you should notice that the algorithm never stops.

Look at the paths as the algorithm is exploring them. Which ones do not make any sense for it to explore? (See answer at the end of assignment.)

Change the code so that these ones are not explored. This should prevent your algorithm from finding an infinite number of paths, and so it should halt.

In the code, the newpath is added to the end of the list of alternatives. If you add the newpath to the beginning of the list of alternatives, you would be doing a depth-first search.

**Writeup 4.5** Hand in your code, highlighting what code you added.

**Writeup 4.6** Explain how depth-first and breadth-first search differ in terms of how they explore the maze.