Question 1: 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.

```
 1 2 3 4
 5 6 7 8
 9 10 11 12
13 14 15 16
```

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 you 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.

```tcl
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 `lindex` and `lrange` are all functions that do not modify their parameters. This differs from `lappend`, 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.

```tcl
set l [list a b c]
set l {a b c}
set l "a b c"
```

But the following do not.

```tcl
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"
```

Try them out.

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

```tcl
set l a
set m {a}
if {$l == $m} {
    puts 1
}
```

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. We want the program to return all paths from the start cell to the end cell. When you run, 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. (See answer at the end of assignment.) 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.

Explain how depth-first and breadth-first search differ in terms of how they explore the maze. (See answer at the end of the assignment.)

Question 2: Top-Down Parsing

Now you are going to build a top-down parser. A top-down parser is very inefficient, as it searches for a parse by trying all possible derivations from the start symbol to a list of terminals. However, it is good to practice your Tcl programming.

Below is the pseudo-code for the parser. Note that this is slightly different from in the class notes.

Create a list of alternatives consisting of just the start symbol
While list of alternatives is not empty
   Pull out first alternative from list
   Find first non-terminal, call it token
   If there is no non-terminal, continue with the next alternative
   For each rule
      If the left-hand-side (head) of the rule is the same as token
         Create new derivation with token replaced by right-hand-side (body) of the rule
         If new derivation is the sequence of words to be parsed, halt with yes
         Add new derivation to end of the list of alternatives

Let’s first focus on finding the first non-terminal. Actually, we not only what to determine what the token is, but also what position it is in the list (position 0, 1, 2, etc). So, rather than using a foreach, you should use the for construct, and check whether the token at each index is a non-terminal. To determine whether a token non-terminal or not, you can simply do the following:
   if {$token >= "A" && $token <= "Z"} ...

Write code that sets the variable which to ””, and then has a for loop that iterates i through the length of the list. For each position, it sets the variable token to ith element in the list, and if the token is a non-terminal, it sets which to the index, and stops. Test out your code by making sure it gives the right answer if the first alternative is {a b C d e}. (See answer at the end of the assignment.)

The next tricky line of code is to create new derivation with token replaced by the right-hand-side of the rule. To do this, you can use lrange to get everything on the left side of the token, and lrange to get everything on the right side of the token, and concat to concatenate all three parts (including the body) together. One nice thing about lrange is that you do not have to worry about the special case where there is nothing on the right (or the left) of the token. [lrange $list 0 -1] gives the empty list. Also, concat does proper thing when concatenating empty lists. Write the code that creates the new derivation where the previous derivation is called $first, token is at position $which, and the right-hand-side of the rule is $rulebody($r). (See answer at the end of the assignment.)

You are now ready to make the top-down parsing routine.

wm withdraw .
```tcl
console show

proc parse {words} {
    set alternatives [list S]
    while {$alternatives != {}} {
        ### YOUR CODE HERE
    }
    return no
}

proc addrule {Head Body} {
    set ::rulehead($::numrules) $Head
    set ::rulebody($::numrules) $Body
    incr ::numrules
}

set numrules 0
addrule S {NP VP}
addrule S {VP}
addrule VP {V AdverbPhrase}
addrule VP {V NP}
addrule VP {V NP AdverbPhrase}
addrule NP {N}
addrule NP {Det N}
addrule NP {N N}
addrule AdverbPhrase {Adverb NP}
addrule N {time}
addrule V {time}
addrule N {flies}
addrule V {flies}
addrule V {like}
addrule Adverb {like}
addrule Det {an}
addrule N {arrow}

parse {time flies like an arrow}
```

Also, to help you out with tcl programming, you can look at the next question, which gives you the code for bottom-up parsing.

Writeup 2.1 Hand in your parse routine. Make sure that it is formatted so that it is readable.

**Question 3: Bottom-Up Parsing & Semantic Interpretation**

In this question, you will be adding semantic interpretation to the bottom-up parser given below. You will be adding support for parallel syntax and semantic rules.
# this version is also keeping track of the ‘parse tree’

proc parse {words} {
    set parselist [list $words]
    set seen {}
    set cnt 0
    while {$parselist != {}} {
        set p [lindex $parselist 0]
        set parselist [lrange $parselist 1 end]
        puts """Considering $p"
        update
        for {set r 0} {$r < $::numrules} {incr r} {
            set l [llength $::rulebody($r)]
            set n 0
            for {set n 0} {$n < [llength $p]} {incr n} {
                set e [expr $n + $l - 1]
                set pr [lrange $p $n $e]
                if {$pr != $::rulebody($r)} continue
                set pnew [concat [lrange $p 0 [expr $n - 1]] \ 
                    $::rulehead($r) \ 
                    [lrange $p [expr $n + $l] end]]
                incr cnt
                if {[lsearch $seen $pnew] > -1} {
                    puts "$cnt Duplicate $pnew"
                } else {
                    lappend parselist $pnew
                    lappend seen $pnew
                    puts "$cnt $pnew"
                }
            }
        }
        if {[expr $cnt % 100]} update
        if {$pnew == "S"} {
            puts "Found S"
            return yes
        }
    }
    puts "No parse found"
    return no
}

proc addrule {Head Body Semantics} {
    set ::rulehead($::numrules) $Head
    #do some pre-processing to make variables easier to deal with
    #call variables <1> depending on position of variable in token

set l 0
set ::rulebody($::numrules) {}
set ::rulevars($::numrules) {}

set newSem $Semantics
while {$l < [llength $Body]} {
    set name [lindex $Body $l]
    set var $name
    set parts [split $name ":"]
    if {[lindex $parts 1] != ""} {
        set name [lindex $parts 1]
        set var [lindex $parts 0]
    }
    lappend ::rulebody($::numrules) $var
    set name [format "<%s>" $name]
    regsub $name $newSem [format "<%d>" $l] newerSem
    if {$newerSem != $newSem} {
        lappend ::rulevars($::numrules) $l
        set newSem $newerSem
    }
    incr l
}
set ::rulesem($::numrules) $newSem
puts "$::rulebody($::numrules) $::rulesem($::numrules)"
incr ::numrules
}

set numrules 0
addrule S {NP VP} S(<NP>,<VP>)
addrule S {VP} S(<VP>)
addrule VP {V AdverbPhrase} VP(<V>,<AdverbPhrase>)
addrule VP {V NP} VP(<V>,<NP>)
addrule VP {V NP AdverbPhrase} VP(<V>,<NP>,<AdverbPhrase>)
addrule NP {N} NP(<N>)
addrule NP {Det N} NP(<Det>,<N>)
addrule NP {N:N1 N:N2} NP(<N1>,<N2>)
addrule AdverbPhrase {Adverb NP} AdverbPhrase(<Adverb>,<NP>)
addrule N {time} N(time)
addrule V {time} V(time)
addrule N {flies} N(flies)
addrule V {flies} V(flies)
addrule V {like} V(like)
addrule Adverb {like} A(like)
addrule Det {an} Det(an)
addrule N {arrow} N(arrow)

parse {time flies like an arrow}

Included above is a new version of addrule. It allows the designer of the grammar to refer to the semantics of a constituent by either its name, or by a renaming. It rewrites the semantic rules to refer to the semantics of the constituents by their index in the syntactic rule. For example, the semantic rule for compound nouns of "<N1><N2>" becomes <0><1>. It also simplifies the syntactic rules to remove the renaming. For
example, it simplifies the rule for compound nouns to just “NP ← N N”. This is done so that your code that
uses the semantic rules to build the semantics can be simpler. Note that the variable rulevars has a list of
the variables in the semantic rule. For the example above, this will be the list \{0, 1\}.

Also included in the grammar above is the corresponding semantics for each grammar rule. The semantic
processing rules are not what you would typically think of as semantic processing rules, but your implement-
ation of the semantic building formalism should not care.

As you are adding the semantic processing to the bottom-up parser, you should modify its check for duplicates
so that a derivation is only removed if it has the same set of tokens and the same semantics as one already
seen.

You should also modify the code so that rather than just stopping after it gets the first answer, it stores it
in a list of answers, and keeps looking for more answers. At the end, it should print out the set of answers
(their semantics). You want to do this as there might be more than one semantic interpretation for the input
sequence of words.

For debugging your code, you might want to exclude some of the grammar rules so that there is just a single
parse. Or consider using one of the simpler grammars that we used in class for debugging. Below, is the
grammar for the banking application that we did in class.

Writeup 3.2  Hand in the code for parse.

Writeup 3.3  Hand in the answers that it returns for the semantic interpretation of the word sequence
‘time flies like an arrow’ using the full grammar.

Writeup 3.4  The semantic rules in the grammar above are obviously not building frame semantics, or
first order predicate calculus. What are they building? (This is a trick question.)
Answers to Question 1:

Which paths do not make any sense for it to explore? You should see that it is creating paths that have loops in them, such as going from cell 1 to cell 2 and then back to cell 1!

How do you eliminate these paths? Immediately inside of the foreach loop that iterates through all of the connect cells, check if the cell is already in the path and if it is, you should continue to the next value in the foreach loop. Here is the code you should add in:

```
if {[lsearch $firstpath $cell]} continue
```

How does depth-first and breadth-first differ? Breadth-first explores all paths equally, exploring all paths of length 1, then exploring each path can be extended to a path of length 2, then exploring how each path can be extended to a path of length 3, etc. Depth-first explores a path fully, and if that does not work, it checks the most recent alternative, for a path to the end.

Answers to Question 2:

Here is the code to find the first non-terminal.

```
set which ""
for {set i 0} {$i < [llength $firstalt]} {incr i} {
    set token [lindex $firstalt $i]
    if {$token >= "A" && $token <= "Z"} {
        set which $i
        break
    }
}
```

Here is the code to construct the new body. Note the use of the \ to show that a line is continued on the next line. This is useful for formatting long lines of code.

```
set new [concat [lrange $first 0 [expr $which - 1]] \  
    $::rulebody($r) \  
    [lrange $first [expr $which + 1] end]]
```