CSE550: Homework 3

Question 1: 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 want 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 .
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 1.1 Hand in your parse routine. Make sure that it is formatted so that it is readable.

Question 2: 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.

wm withdraw .
console show
update

# 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 ""
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 1 0
    set ::rulebody($::numrules) {}
    set ::rulevars($::numrules) {}

    set newSem $Semantics
    while {$l < [llength $Body]} {
        set name [lindex $Body $l]
        set var $name
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 implementation 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.

Writeup 2.2 Hand in the code for parse.

Writeup 2.3 Hand in the answers that it returns for the semantic interpretation of the word sequence ‘time flies like an arrow’ using the full grammar. You should have 3 answers.

Writeup 2.4 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.)