Student ID numbers: S1580425, S1680791

TASK 1

def buildIndices(self,productions):

Postcondition: The "defaultdict"s (subclasses of dictionaries that call a factory function attribute) 'unary' and 'binary' are created and populated with unary and binary productions from NLTK. The defaultdict subclass is an optimal data structure because it can create a dictionary, as in this case, where the values are stored as lists. When populating the CKY matrix, the left-hand-side rules need to be looked up quickly to see what right-hand-side rule(s) they map to. So the keys are RHSs because looking up a dictionary key is extremely fast.

How: Create two specialised container datatypes i.e defaultdicts to store the production rules, one for unary rules and one for binary rules. Ensure the "rhs" has a length of one or two. If the length of "rhs" for a production is equal to one, append it as the key with it's associated "lhs" symbol as the value in the dictionary self.unary. If the length of "rhs" for a production is equal to two, append it as the key with its "lhs" symbol as the value in the dictionary self.binary.

:type productions: nltk.grammar.Production
:param productions: A binary or unary CFG rule
:return: none

def unaryFill(self):

Postcondition: The middle cells of the matrix are filled moving along the diagonal from the top left to the bottom right with words and corresponding unary non-terminals.

How: Iterate over the length of the input string. Add word corresponding to cell. Add the non-terminal symbol associated with the word to the cell by looking up the word in the self.unary dictionary.

def unaryUpdate(self,symbol,depth=0,recursive=False):

Postcondition: Cells in the matrix containing symbols (either words or non-terminals) that are RHSs of a unary rule are expanded and the corresponding LHS are added to the cell.

How: Add word (terminal symbol) to cell. If the word is in the dictionary self.unary, add the corresponding LHS (parent symbol) to the cell. If the parent symbol is in the dictionary self.unary, add its respective parent symbol to the cell, recursively.

:type symbol: str OR nltk.grammar.Nonterminal
:param symbol: the symbol that will be passed through to find the related unary rule(s)
:return: none

**def recognise(self,tokens,verbose=False):**

Postcondition: A matrix has been initialized and filled using the CKY algorithm.

How: Define "words" as the list of tokens in the input string. Define "n" as the number of tokens plus 1. Initialize an empty matrix and iteratively add n-1 rows and n columns. Call unaryFill() to fill the cells with all possible unary productions. Call binaryScan() to fill the cells with all possible binary productions. Return whether or not the starting symbol (the first listed rule in the grammar) is in the matrix, indicating that the input string can be parsed according to the grammar.

:type tokens: list(str)
:param tokens: The list of tokens (as strings) used to build the matrix.
:type verbose: bool
:param verbose: show debugging output if True, defaults to False
:rtype: bool
:return: True if the CKY recognizer recognizes some valid parse for the input string, otherwise False

**def maybeBuild(self, start, mid, end):**

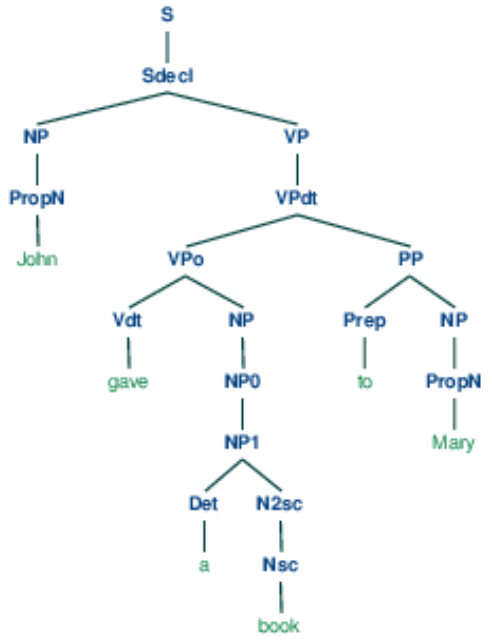
Postcondition: The matrix has been populated with all possible binary productions

How: Set s1 to the nonterminal in cell (start, mid). Set s2 to the nonterminal in cell (mid, end). If there is a RHS (s1, s2) in the binary dictionary, add the corresponding LHS nonterminal to cell (start, end). If s is the RHS of a unary rule, call the unaryUpdate() function to add the corresponding LHS to cell (start, end).

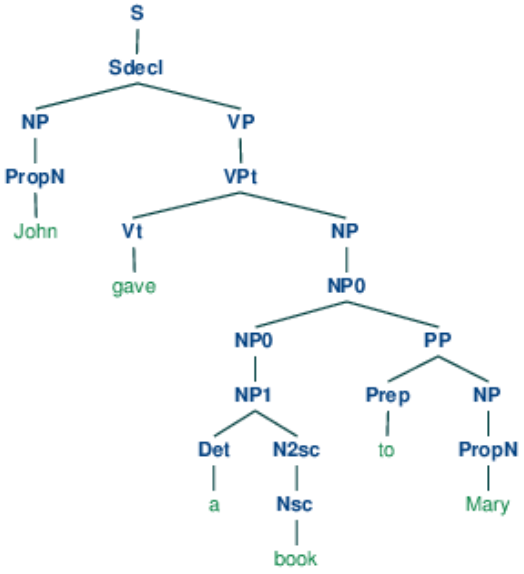
:type start: int
:param start: the beginning position of the token span in question
:type mid: int
:param mid: some position in the token span in question between start and end
:type end: int
:param end: the final position of the token span in question
:return: none

TASK 2

**Tree A:** (S(Sdecl(NP(PropN John))(VP(VPdt(VPo(Vdt gave)(NP(NP0(NP1(Det a)(N2sc(Nsc book))))))(PP(Prep to)(NP(PropN Mary)))))))




**Tree B:** (S(Sdecl(NP(PropN John))(VP(VPt(Vt gave)(NP(NP0(NP0(NP1(Det a)(N2sc(Nsc book))))(PP(Prep to)(NP(PropN Mary)))))))



Tree B is an extremely unlikely parse for this sentence.

Tree A could be paraphrased as "Mary was given a book by John". Tree B is extremely difficult to paraphrase since it is hard to conceive of its meaning, but it would be something like "A book, which was *to* Mary, was given by John".

Tree B would be a correct parse for the English sentence "John likes the book with pictures", whereas Tree A would be a possible parse, but much less likely to a native speaker. In this case, a paraphrase for the sentence parsed as Tree B would be "John enjoys the book that has pictures in it." A paraphrase for the sentence parsed as tree A would be "John and several pictures enjoy the book together." So the second reading is plausible, but very unlikely.

## TASK 5

A brief background in CKY parsing:

CKY (Cocke, Kasami, Younger) is a dynamic programming algorithm that computes a structure for an input string (of span i to j) given a grammar. Dynamic programming here means that the structure can be built using sub-solutions spanning i to j (that is, i to k and k to j).

The CKY parser is an extension of the CKY *recognizer*. It assumes that any sub-string is independent of the rest of the parse, and so avoids having to re-analyse the sub-strings. This algorithm also builds sub-trees over words, building upwards and adding traces leading up to the start Non-Terminal (S). These two qualities make CKY a bottom-up, breadth-first parsing algorithm and in its simplest form, it assumes a grammar in Chomsky Normal Form (CNF). That is, the recognizer component recognizes languages defined by unary or binary rules.

The parser uses the traces generated by the recognizer to trace through the history of building up to the start symbol in order to return a possible parse.

Because CKY builds *all* parses, it reflects global ambiguity, although in this particular case we are only interested in the first generated parse.

Design and implementation:

We wanted to allow the recognizer to run as usual, but at each rule expansion, include a trace in the form of a subtree.

The subtree was attached as the second element of a tuple as defined by Label.tracetup. Label.tracetup constructs tuples by taking two arguments, the first of which will always be a string (when at the first stage of populating the matrix with terminal words) or an nltk.grammar.nonTerminal, and the second of which is a subtree in string form. At each rule expansion, the subtree was expanded in such a way that it could be read by the NLTK tool nltk.tree.Tree.fromstring(). This way, once the recognizer has filled the last cell in the matrix with a start symbol (S), the trace in the Label.tracetup would contain 'S' as its first element and the complete parse of the sentence in an NLTK-readable format as its second element.

In implementing this approach, it was very difficult to restrict the second element of the tuple Label.tracetup to *only* the format readable by nltk.tree.Tree.fromstring(). For this reason, we needed to add extra string processing to the CKY.firstTree method.

The building up of the trace was done within the CKY.maybeBuild method and the Cell.unaryUpdate method. Below is the full documentation for these methods (more detailed explanation of each line can be found in the programme itself):

**def maybeBuild(self, start, mid, end):**

Postcondition: The matrix has been populated with all possible
binary productions and a sub-tree and a Label.tracetup instance have been generated

How: Set s1 to the Label.tracetup in cell (start, mid). Set s2 to the
Label.tracetup in cell (mid, end). If there is a RHS (s1[0], s2[0]) in the
binary dictionary, add the corresponding LHS nonterminal to cell
(start, end). If s is the RHS of a unary rule, call the unaryUpdate()
function to add the corresponding LHS to cell (start, end).
Generate a new sub-tree 'parse_string_bin' and combine it with the RHS non-terminal
's' to create a new Label.tracetup instance 'newLabel'.

:type start: int
:param start: the beginning position of the token span in question
:type mid: int
:param mid: some position in the token span in question between start
and end
:type end: int
:param end: the final position of the token span in question
:return: none

**def unaryUpdate(self,symbol,depth=0,recursive=False):**

Postcondition: Cells in the matrix containing either words
or Label.tracetup instances that are RHSs of a unary rule are expanded and the
corresponding LHS are added to the cell.

How: Add word (terminal symbol) to cell. If the word is in the
dictionary self.unary, add the corresponding LHS (parent symbol) as a Label.tracetup
instance to the cell. If the parent symbol is in the dictionary self.unary, add
its respective parent symbol and Label.tracetup instance to the cell, recursively.

:type symbol: str OR Label.tracetup
:param symbol: the word or Label.tracetup that will be passed through to find the
related unary rule(s)
:return: none

Also, below is the documentation for the CKY.firstTree method and the Label.tracetup method:

**def firstTree(self):**

Postcondition: A complete parse tree is printed based on the traces
derived from the CKY parser

How: Convert the last cell (containing a start symbol 'S') in the matrix to a string.
Process and clean the string to extract only the string in bracket form that
nltk.tree.Tree.fromstring() will accept to build a parse tree. Pass the string to the
nltk.tree.Tree.fromstring() tool and then, optionally, use the tree.draw() tool from NLTK to
draw the parse tree.

**def tracetup(symbol, trace):**

Postcondition: A tuple object has been created with prespecified elements.

How: Take two arguments, store as a tuple.

:type symbol: str OR nltk.grammar.nonTerminal
:param symbol: The highest node in the subtree at some point in parsing
:type trace: str
:param trace: a string that represents the bracket form of a sub-tree
:rtype: tuple(str OR nltk.grammar.nonTerminal, str)
:return: a tuple that logs the highest node at some stage in parsing and the most developed
sub-tree

Ultimately we were able to successfully return a single parse for each well-formed sentence (according to the given grammar) with the option of drawing its tree automatically. Every parse was correct according to the grammar (although, as discussed previously, some parses were inappropriate despite being correct based on the grammar).

At some point in the building up of the trace string, there was a slight error in composing the parse such that there are some duplicate nonterminals. This led to some redundant levels in the tree branches as illustrated below (duplicates of PropN, VPdt, NP0 and NP1), but it did not affect the structure of the tree, which was correct according to the parse in each case (including below). It seems that the duplicates happen only where there has been a unary rule expansion, which led us to believe that there was some redundant code in Cell.unaryUpdate(). However, we were unable to resolve the redundancy without generating other, more significant errors, so we left the method as is.

Parse tree for “John gave Mary a book.”

((S ((Sdecl ((NP ((PropN (PropN (John ))))))((VP ((VPdt (VPdt ((VPio ((VtVdt (gave )))((NP ( (PropN (PropN (Mary )))))))))((NP ((NP0 (NP0 ((NP1 (NP1 ((Det (a )))((N2sc ((Nsc (NscVt (book ))))))))))))))))(. )))