INFO-Y004: Natural Language Processing
Assignment 2a: Chart Parser

Brian Delhaisse

April 29, 2015

1 Introduction

The goal of this assignment is to build a chart parser which is appropriate for ambiguous grammars such as grammars of natural languages. It uses the dynamic programming approach to solve the problem. From a practical point of view, the goal is to compute the parse tables and from those to generate the final parse trees.

For this assignment, I used Python (2.7.5) as the programming language, and the *Natural Language Toolkit* (*nltk*) library.

2 Sentences

Here is the list of sentences from which we need to produce the parse tables and parse trees.

1. *I gave an apple to the teacher.*
2. *I gave the teacher a very heavy book.*
3. *I gave the policeman a flower.*
4. *Mary thinks that I gave a policeman a flower.*
5. *I persuaded Harry to give the policeman a flower.*
6. *John is eager to please.*
7. *John is easy to please.*
8. *This is the dog that chased the cat.*
9. *This is the cat that the dog chased.*
10. *This is the cat that Mary said that the dog chased.*

From those sentences, we can identify the lexical categories for each word, and regroup those lexical categories into phrasal categories, in order to create a grammar.

3 Grammar

A grammar is composed of nonterminal (written in capitals) and terminal (written in miniscule) symbols. The nonterminal symbols correspond to the lexical and phrasal categories, while terminal symbols correspond to the words present in the different sentences.

Here is the list of lexical and phrasal categories[1] identified in the different sentences:

[1] For more information about some of the different categories, see http://en.wikipedia.org/wiki/Syntactic_category

- Lexical Categories: determiner (D), noun (N), proper-noun (PN), adjective (Adj), adverb (Adv), pronoun (Pr), particle (Par), verb (V), adposition (preposition, postposition, circumposition) (P), subordinate conjunction (Sub), relative pronoun (RPr).
- Phrasal categories: sentence (S), noun phrase (NP), verb phrase (VP), adjective phrase (AP), adposition phrase (PP), Direct object Phrase (DOP), adjective-noun phrase (ANP), Sub phrase (SP), relative pronoun phrase (RP), noun-relative pronoun phrase (NRP), particle-verb phrase (ParVP), particle phrase (ParP).

In order to apply the CKY (Cocke-Kasami-Younger) algorithm, which use a bottom-up dynamic programming approach to create the parse trees, we need to the grammar to be in Chomsky Normal Form (CNF). A CNF grammar is a grammar in which all the production rules are in the form: $\begin{array}{ccc} \text{A} & \rightarrow & \text{B C} \\ \text{B} & \rightarrow & \text{b} \end{array}$ where A, B and C are nonterminal symbols and b is a terminal symbol. Combining this fact and the lexical/phrasal categories defined above, we obtain the following CNF grammar for the given sentences:

S	→	NP VP
NP	→	D N | D ANP
ANP	→	AP N | Adj N
AP	→	Adv Adj
VP	→	V NP | V PP | V DOP | V SP | V NRP | V ParP | V VP | V AP | V Adj
PP	→	P NP
DOP	→	NP PP | NP NP
SP	→	Sub S
NRP	→	NP RP
RP	→	RPr VP | RPr S
ParP	→	NP ParVP | Adj ParVP | AP ParVP
ParVP	→	Par VP
NP	→	'I' | 'This'
RPr	→	'that'
Sub	→	'that'
NP	→	'Mary' | 'John' | 'Harry'
D	→	'an' | 'a' | 'the'
N	→	'apple' | 'teacher' | 'book' | 'policeman' | 'cat' | 'dog' | 'flower'
V	→	'gave' | 'give' | 'please' | 'is' | 'chased' | 'thinks' | 'said' | 'persuaded'
VP	→	'gave' | 'give' | 'please' | 'is' | 'chased' | 'thinks' | 'said' | 'persuaded'
P	→	'to'
Par	→	'to'
Adj	→	'heavy' | 'easy' | 'eager'
Adv	→	'very'

Table 1: My CNF grammar

One can notice that my grammar is little bit more general than the grammar which would be produced only by the given sentences. My grammar can for example accepts sentence which contains more than one verb like "John is persuaded that the dog chased the cat" which contains two verbs ('is' and 'persuaded'). It can also accept sentences such as "This is (very) easy", etc.

4 Code

To launch the code, just type the command *python assignment2.py* into the console.

In my code, I have 6 functions:

1. the *displayGrammar(grammar)* function which displays the grammar in the standard output. The grammar is produced by using the *nltk.CFG.fromstring("""<grammar>""")* method.

2. the *createParseTableAndTree(tokens, grammar, trace=False)* function which creates the parse table and the parse tree. It accepts as arguments the list of tokens in the sentence, the grammar, and an optional argument to indicate if we want to print in the standard output the different decisions made. This function returns the parse table and the parse tree. The implementation of the CKY algorithm in this function is described in the subsection 4.1 below.

3. the *displayParseTreeNLTK(tokens, grammar)* function which takes as argument the list of tokens and the grammar. This function uses the *nltk.ChartParser(grammar)* which produces a chart *parser* (using the Earley algorithm[2]) which can parse the list of tokens by calling *parser.parse(tokens)* which produce the parser tree. This tree is printed in parentheses notation (LISP syntax) in the standard output. This function was primarly there to test if I obtain the same result with the CKY algorithm.

4. the *displayParseTree(tree, draw = False, filename=")* function which takes as argument the tree, and 2 optional arguments to indicate if we want to display graphically the parse tree and/or to save it into a PostScript file (*<filename>.ps*). If the filename argument is an empty string (value by default), it will not save the tree into a file. By default, this function prints the parse trees in the standard output using the parentheses notation (LISP syntax).

5. the *displayParseTable(table)* function which displays the parse table in the standard output. This method has been taken from the book *"Natural Language Processing with Python" - chap8*[3], and has been slightly modified.

6. the *printParseTableLatex(table)* function which prints in the standard output the parse table in the latex format.

I also wrote quickly a bash script "*makeAllGraph.sh*" to produce the PNG files from the PostScript files.

4.1 Implementation of the CKY algorithm

The CKY algorithm has been implemented in the *createParseTableAndTree* function. It's a recognizer thus it only determines if a sentence is in the language described by the grammar. In order to transform it into a parser, we need to keep track of the decisions it has made. In my function, I have 2 dictionnaries: one which contains as keys the right hand side of the production rules of my grammar, and as values the left hand side of those rules. The other dictionnary contains the matching rules found while building the parse table. The CKY algorithm used the upper right triangular portion of the table. It will then look for every cell in this portion going to each column, each row as described in the course, that is it will scan the table from left to right, and from bottom to top.

To illustrate how the CKY algorithm works, I will use as example the 1st sentence "*I gave an apple to the teacher*". Suppose that we are at the cell (3,4) in the parse table (see table 2), and the left part of the table have been already filled. This cell (3,4) corresponds to the word "*apple*" because it's between the node 3 and 4 (see the initial chart data structure, fig 1). This word will be looked in the 1st dictionnary, to get the nonterminal

[2]The Earley algorithm, compared to the CKY algorithm, uses a top-down dynamic approach, and allows arbitrary CFGs.

[3]http://www.nltk.org/book/ch08.html

symbol which produces this word (in this case, $N \rightarrow apple$). If they are several possible nonterminal symbols, they will be all added in the corresponding cell. Then, we scan from bottom to top (the CKY is a bottom-up dynamic programming approach). Going in the upper row, we are in the cell (2,4), meaning that we are considering the words that are between the node 2 and 4 in the chart data structure (see figure 1), that is "*an apple*". From the cell (2,4), we will look at the left side if the nonterminal symbol at cell (2,3) D, and the nonterminal symbol at cell (3,4) N can be produced by a nonterminal symbol $X \rightarrow DN$. We looked in the 1st dictionnary to get this symbol if it exists, in this case, it's NP. Because we found a nonterminal symbol, this matching rule and the corresponding indices are added into the 2nd dictionnary which at the end will contain all the matching rules found while building the parse table. The nonterminal found is also added in the cell. From that point, we continue to the upper row, and continue this approach.

In general, to find all the possible nonterminal symbol at a particular cell (i,j), we look in the 1st dictionnary if the nonterminal at the cell (i,k), and the nonterminal at the cell (k,j) can be produced by a nonterminal symbol (k goes from $i + 1 \rightarrow j - 1$). If there is at least one nonterminal, this one is added in the cell (i,j) and the matching rule is added in the 2nd dictionnary along with the indices of those nonterminals. Once done, we consider subsequent part of the sentence by going to the upper row (the part will then consider an extra word compared to the row that we came from), or going to the next column (once there is no upper row in the current column we are) which in turn will consider subsequent increasing part of the sentence by going to upper rows.

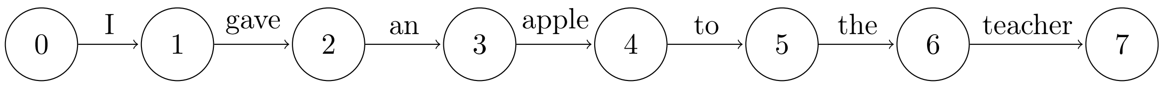

Figure 1: The initial chart data structure of the 1st sentence

	1	2	3	4	5	6	7
0	[NP]	[S]		[S]			[S]
1		[VP, V]		[VP]			[VP]
2			[D]	[NP]			[DOP]
3				[N]			
4					[P, Par]		[PP]
5						[D]	[NP]
6							[N]

Table 2: The parse table for the 1st sentence

At the end, the parse tree is created if the parse table contains at the upper right corner of the table the initial symbol S, which means that the sentence has been recognized. From that point, my function creates the parse tree using the 2nd dictionnary which contains the matching rules found when constructing the parse table (see figure 2).

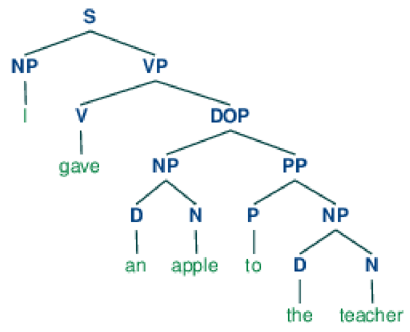

Figure 2: The parse tree for the 1st sentence

The result of filling the parse table can also be visualized in the chart data structure (see figure 3).

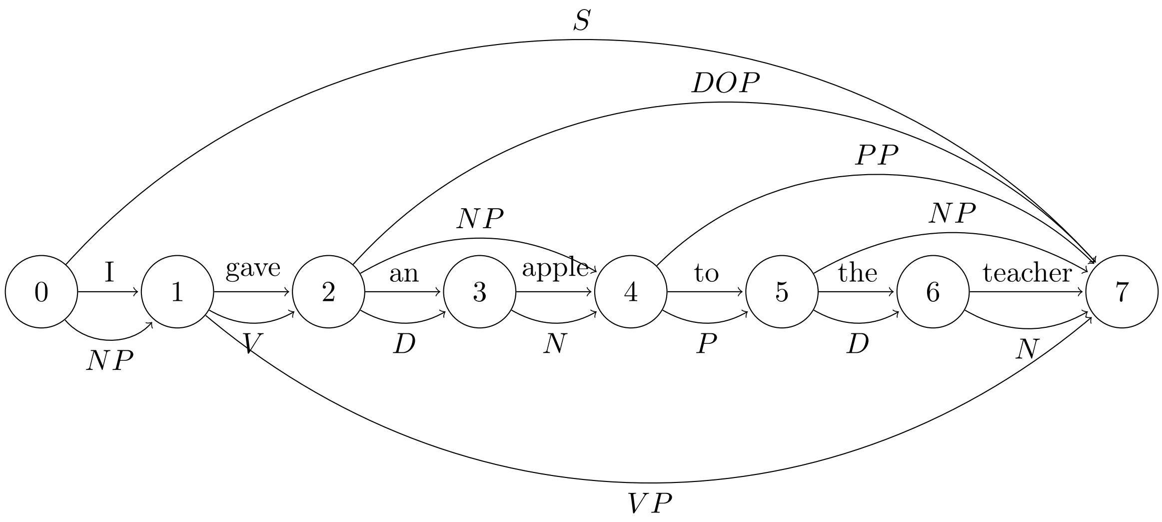

Figure 3: The final chart data structure of the 1st sentence

More information on the algorithm or other technical details can be found in the source code at the appendix.

5 Result

In the following subsections 5.1 and 5.2, the parse table and parse tree for each sentence is presented.

5.1 Parse Tables

Here are the parse tables for each sentence.

1. *I gave an apple to the teacher.*

	1	2	3	4	5	6	7
0	[NP]	[S]		[S]			[S]
1		[VP, V]		[VP]			[VP]
2			[D]	[NP]			[DOP]
3				[N]			
4					[P, Par]		[PP]
5						[D]	[NP]
6							[N]

2. *I gave the teacher a very heavy book.*

	1	2	3	4	5	6	7	8
0	[NP]	[S]		[S]				[S]
1		[VP, V]		[VP]				[VP]
2			[D]	[NP]				[DOP]
3				[N]				
4					[D]			[NP]
5						[Adv]	[AP]	[ANP]
6							[Adj]	[ANP]
7								[N]

3. *I gave the policeman a flower.*

	1	2	3	4	5	6
0	[NP]	[S]		[S]		[S]
1		[VP, V]		[VP]		[VP]
2			[D]	[NP]		[DOP]
3				[N]		
4					[D]	[NP]
5						[N]

4. *Mary thinks that I gave a policeman a flower.*

	1	2	3	4	5	6	7	8	9
0	[NP]	[S]			[S]		[S]		[S]
1		[VP, V]			[VP]		[VP]		[VP]
2			[RPr, Sub]		[RP, SP]		[RP, SP]		[RP, SP]
3				[NP]	[S]		[S]		[S]
4					[VP, V]		[VP]		[VP]
5						[D]	[NP]		[DOP]
6							[N]		
7								[D]	[NP]
8									[N]

5. *I persuaded Harry to give the policeman a flower.*

	1	2	3	4	5	6	7	8	9
0	[NP]	[S]	[S]		[S]		[S]		[S]
1		[VP, V]	[VP]		[VP]		[VP]		[VP]
2			[NP]		[ParP]		[ParP]		[ParP]
3				[P, Par]	[ParVP]		[ParVP]		[ParVP]
4					[VP, V]		[VP]		[VP]
5						[D]	[NP]		[DOP]
6							[N]		
7								[D]	[NP]
8									[N]

6. *John is eager to please.*

	1	2	3	4	5
0	[NP]	[S]	[S]		[S]
1		[VP, V]	[VP]		[VP]
2			[Adj]		[ParP]
3				[P, Par]	[ParVP]
4					[VP, V]

7. *John is easy to please.*

	1	2	3	4	5
0	[NP]	[S]	[S]		[S]
1		[VP, V]	[VP]		[VP]
2			[Adj]		[ParP]
3				[P, Par]	[ParVP]
4					[VP, V]

8. *This is the dog that chased the cat.*

	1	2	3	4	5	6	7	8
0	[NP]	[S]		[S]		[S]		[S]
1		[VP, V]		[VP]		[VP]		[VP]
2			[D]	[NP]		[NRP]		[NRP]
3				[N]				
4					[RPr, Sub]	[RP]		[RP]
5						[VP, V]		[VP]
6							[D]	[NP]
7								[N]

9. *This is the cat that the dog chased.*

	1	2	3	4	5	6	7	8
0	[NP]	[S]		[S]				[S]
1		[VP, V]		[VP]				[VP]
2			[D]	[NP]				[NRP]
3				[N]				
4					[RPr, Sub]			[RP, SP]
5						[D]	[NP]	[S]
6							[N]	
7								[VP, V]

10. *This is the cat that Mary said that the dog chased.*

	1	2	3	4	5	6	7	8	9	10	11
0	[NP]	[S]		[S]			[S]				[S]
1		[VP, V]		[VP]			[VP]				[VP]
2			[D]	[NP]			[NRP]				[NRP]
3				[N]							
4					[RPr, Sub]		[RP, SP]				[RP, SP]
5						[NP]	[S]				[S]
6							[VP, V]				[VP]
7								[RPr, Sub]			[RP, SP]
8									[D]	[NP]	[S]
9										[N]	
10											[VP, V]

5.2 Parse Trees

Here are the parse trees produced by the chart parser.

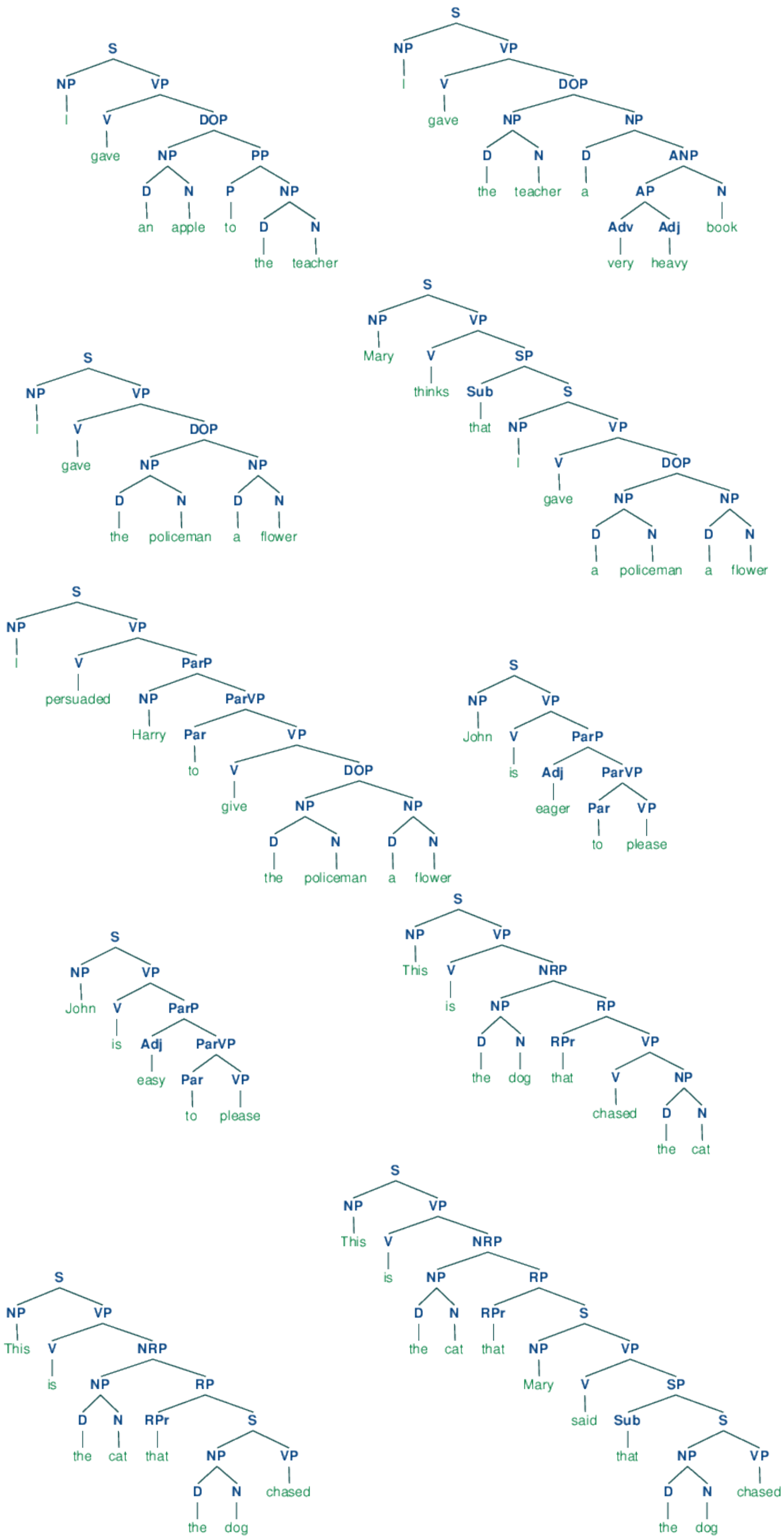

Figure 4: Drawing of the parse trees for the different sentences

6 Summary

This assignment allowed me to revise some lexical and phrasal categories that I forgot long ago, and to construct a context-free grammar (to be more specific a CNF grammar) which is the basis for describing the (syntactic) structure of natural language sentences.

It allowed me also to understand better the various problems that can arise with ambiguous grammar. The CKY algorithm which uses a bottom up dynamic programming approach is quite effective but require to have a CNF grammar. The Earley algorithm which is top-down approach is a little bit more general in the sense that it accepts arbitrary CFGs. This is the one used by the nltk library. Using this last one, it becomes quite easy to do the assignment, indeed it suffices to do:

```
grammar = nltk.CFG.fromstring("""<grammar>""")
parser = nltk.ChartParser(grammar)
trees = parser.parse(tokens)
```

where <grammar> is our context-free grammar which contains all the production rules.

References

- Wikipedia: http://en.wikipedia.org/wiki/Syntactic_category.
- Natural Language Processing with Python - chapter 8: http://www.nltk.org/book/ch08.html.
- Natural Language Processing course - Lecture 7.

Appendix - Code

```
# Assignment 2a: Practical Orientation - chart parser
# author: Brian Delhaisse
# Python 2.7.5
# references: http://www.nltk.org/book/ch08.html + course + wikipedia

import nltk

# List of sentences
listSent = ["I gave an apple to the teacher", "I gave the teacher a very heavy book", "I gave the policeman a flower",\
  "Mary thinks that I gave a policeman a flower", "I persuaded Harry to give the policeman a flower", \
  "John is eager to please", "John is easy to please", "This is the dog that chased the cat", \
  "This is the cat that the dog chased", "This is the cat that Mary said that the dog chased"]

# Grammar in CNF (Chomsky Normal Form)
grammar = nltk.CFG.fromstring("""
  S -> NP VP
  NP -> D N | D ANP
  ANP -> AP N | Adj N
  AP -> Adv Adj
  VP -> V NP | V PP | V DOP | V SP | V NRP | V ParP | V VP | V AP | V Adj
  PP -> P NP
  DOP -> NP PP | NP NP
  SP -> Sub S
  NRP -> NP RP
  RP -> RPr VP | RPr S
  ParP -> NP ParVP | Adj ParVP | AP ParVP
  ParVP -> Par VP
  NP -> 'I' | 'This'
  RPr -> 'that'
  Sub -> 'that'
  NP -> 'Mary' | 'John' | 'Harry'
  D -> 'an' | 'a' | 'the'
  N -> 'apple' | 'teacher' | 'book' | 'policeman' | 'cat' | 'dog' | 'flower'
  VP -> 'gave' | 'give' | 'please' | 'is' | 'chased' | 'thinks' | 'said' | 'persuaded'
  V -> 'gave' | 'give' | 'please' | 'is' | 'chased' | 'thinks' | 'said' | 'persuaded'
  P -> 'to'
  Par -> 'to'
  Adj -> 'heavy' | 'easy' | 'eager'
  Adv -> 'very'
  """)

# Create a chart parser based on the CKY algorithm
# @return: table - the parse table
# @return: tree - the parse tree
def createParseTableAndTree(tokens, grammar, trace=False):
  # Useful variables
  N = len(tokens)
  table = [[None for i in xrange(N+1)] for j in xrange(N+1)] # This table will contain the lexical/phrasal categories
  dic = {} # this dictionary will contain as keys the right hand side (rhs), and as value the left hand side (lhs) of the grammar.
```

```
  for p in grammar.productions():
    rhs = p.rhs() if len(p.rhs())>1 else (p.rhs()[0], None)
    if rhs in dic: # Ex: A word could belong to multiple lexical categories
      dic[rhs].append(p.lhs())
    else:
      dic[rhs] = [p.lhs()]
  tree = {} # this dictionnary will contain the decisions we made.

  # fill the table
  for j in xrange(1,N+1): #[1,N], position going left to right.
    for i in xrange(j-1,-1,-1): #[j-1,0], position going down to top.
      if i==(j-1): # diagonal elements
        prod = dic[(tokens[i],None)]
        for p in prod:
          if ((p,None) in dic) and all(item not in prod for item in dic[(p,None)]):
            prod = prod + dic[(p,None)]
        table[i][j] = prod
      else: # non diagonal elements
        var = range(i+1,j) if (j-2) >= (i+1) else range(i+1,j-2,-1)
        for k in var: #[i+1,j-1], span.
          if (table[i][k] != None) and (table[k][j] != None):
            for elem1 in table[i][k]:
              for elem2 in table[k][j]:
                if (elem1, elem2) in dic: #add a category in a cell of the table
                  if table[i][j] == None:
                    table[i][j] = dic[(elem1, elem2)] #replace the value
                  else:
                    table[i][j] = table[i][j] + dic[(elem1, elem2)] #concatenate
                  if trace:
                    print "(table[%s][%s] table[%s][%s] table[%s][%s]) = (%s %3s %3s)" % \
                    (i, j, i, k, k, j, dic[(elem1, elem2)][0], elem1, elem2)
                  tree[(dic[(elem1, elem2)][0],i,j)] = [(elem1,i,k), (elem2,k,j)]

  # Construct the tree string, that is the tree in parentheses notation
  def constructTreeString(elem):
    if elem[1]==(elem[2]-1):
      if all(item != elem[0] for item in dic[(tokens[elem[1]],None)]):
        return "(%s (%s %s))" % (elem[0], dic[(tokens[elem[1]],None)][0], tokens[elem[1]])
      return "(%s %s)" % (elem[0], tokens[elem[1]])
    else:
      l = tree[elem]
      return "(%s %s %s)" % (elem[0], constructTreeString(l[0]), constructTreeString(l[1]))

  # Construct the tree
  if (nltk.grammar.Nonterminal("S"),0,N) in tree:
    treeString = constructTreeString((nltk.grammar.Nonterminal("S"),0,N))
    tree = nltk.Tree.fromstring(treeString)
  else:
    print 'Error: the sentence cannot be recognized by the grammar'

  return table, tree

# Display the grammar
def displayGrammar(grammar):
  print '\nGrammar:'
  print grammar
```

```python

# Display the parse tree using the NLTK chart parser
def displayParseTreeNLTK(tokens, grammar):
  parser = nltk.ChartParser(grammar)
  trees = parser.parse(tokens)
  for elem in trees:
    print elem

# display the parse tree
def displayParseTree(tree, draw = False, filename=''):
  print '\nParse tree: '
  print tree
  if draw:
    tree.draw()
  if filename!='':
    cf = nltk.draw.util.CanvasFrame()
    tc = nltk.draw.TreeWidget(cf.canvas(),tree)
    cf.add_widget(tc,10,10) # (10,10) offsets
    cf.print_to_file(filename+'.ps')
    cf.destroy()

# display the parse table (taken from http://www.nltk.org/book/ch08.html and slightly modified)
def displayParseTable(table):
  print '\nChart Table: '
  print '    ' + ' '.join([("%-10d" % i) for i in range(1, len(table))])
  for i in range(len(table)-1):
    print "%d " % i,
    for j in range(1, len(table)):
      print "%-10s" % (table[i][j] or '.'),
    print

def printParseTableLatex(table):
  print '\\begin{table}[H]'
  print '\centering'
  print '\\begin{tabular}{' + '|c'*len(table) + '|}'
  print '  \hline & ' + ' & '.join([("%d" % i) for i in range(1, len(table))]) + ' \\\\'
  for i in range(len(table)-1):
    print "  \hline %d" % i,
    for j in range(1, len(table)):
      print " & %s" % (table[i][j] or ''),
    print '\\\\'
  print '  \hline'
  print '\\end{tabular}'
  print '\\end{table}'

# Main code
displayGrammar(grammar)
for i in xrange(len(listSent)):
  print '\n\n\n%s) Sentence: %s' % (i+1,listSent[i])
  sent = listSent[i].split()
  table, tree = createParseTableAndTree(sent, grammar, trace=False)
  displayParseTable(table)
# printParseTableLatex(table)
# displayParseTreeNLTK(sent, grammar) # Using nltk, which use the Earley algo
  displayParseTree(tree)
#  displayParseTree(tree, draw=True, filename='tree'+str(i+1))
```