1 Tokenization

Tokenizing a sentence by identifying strings which match

[a-zA-Z]+	tokens consisting of any number of letters
'[a-z]+	tokens consisting of ' proceeding any number of letters
[,.?;:()-]	tokens consisting of single punctuation marks

would be sufficient to account for the most common English uses of punctuation symbols, including recognizing as separate tokens possesive *'s* and contractions such as *'d, 't, 're*. However, there is a number of problematic cases:

- lone apostrophes as possessive suffix on nominals ending in s (such as *flowers' pollen* meaning pollen of many flowers);
- ellipsis of the centuries part of a year (such as *the '70s, in the '98*);
- informal contractions (such as *about* → *'bout, unless* → *'less, because* → *'cause, and* → *'n'*);
- surnames and lexical items with apostrophes (such as *'o clock, O'Brian, M'Gregor*);

The tokenize function changes the lone-apostrophes possessive into *'s* so that a word can be tokenized into its main lexical part and the possessive suffix. It also recognizes contracted dates. The other two problems, being largely lexical, remain unsolved.

```
def tokenise(tokenstring):
'''Split a string into a list of tokens, treating punctuation as
separate tokens, and splitting contractions into their parts.
So for example "I'm leaving." --> ["I","'m","leaving","."]'''
    # normalize possessive ' to possessive 's
    normalized_1 = re.sub(r"s'\b", "s's ", tokenstring)
    # if ' is followed by more that 2 digits, it's not a year
    # split into ' and digit string
    normalized_2 = re.sub(r"'(\d{3})", r"' \1", normalized_1)
    # find all alphanumeric strings, 'digit-digit strings, 's/'d/'m etc. strings
    # and single punctuation marks
    return re.findall(r"[a-zA-Z\d]+|'\d\d\b|'[a-z]*|[,.?;:()-]", normalized_2)
```

2 Parse trees

3 Remarks on the grammar

3.1 Design of the grammar rules

a. **Redundancy of rules in the verbal domain**

   There is a redundancy in the treatment of ditransitive verb phrases. *VPo* an *VPio* are two nodes which expand in the exact same way:

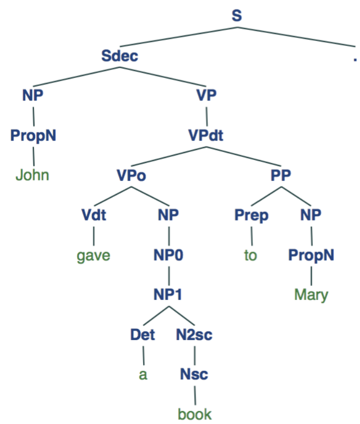

Figure 1: *John gave a book to Mary.*

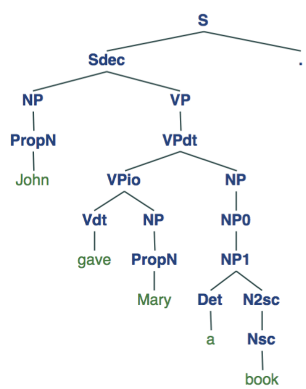

Figure 2: *John gave Mary a book.*

$$VPo \rightarrow Vdt\ NP \text{ and } VPio \rightarrow Vdt\ NP.$$

and therefore there is no difference between them within the grammar. From a linguistic point of view this separation might have a motivation. The ditransitive verb and its direct object are represented by *Po*, while *VPio* is a unit consisting of a ditransitive verb and its indirect object. As valid linguistic distinction as it might be, it does not need to be included in simple grammar like the one discussed here. In particular, no use is made of the distinction between that would limit overgenration. If that was the case, then having both non-terminals would be a valid design choice. For instance, one might consider putting some restrictions on what kinds of NP can be direct or indirect objects. The needless distinction between *VPo* an *VPio* propagates up to the topmost VP expansion rules. As a result, there are more VP-related rules than necessary. The grammar

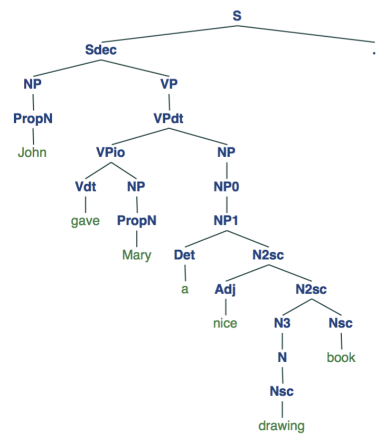

Figure 3: *John gave Mary a nice drawing book.*

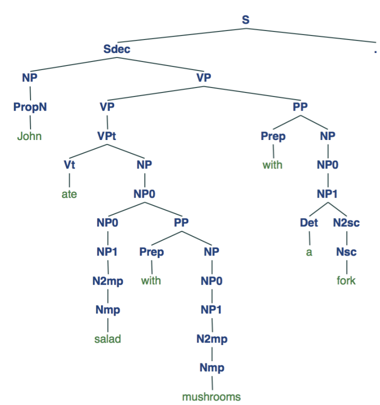

Figure 4: *John ate salad with mushrooms with a fork.*

contains

$$VP \to VPi \mid VPt \mid VPdt \mid Mod\ VP \mid VP\ Adv \mid VP\ PP$$
$$VPdt \to VPo\ PP$$
$$VPdt \to VPio\ NP$$
$$VPo \to Vdt\ NP$$
$$VPio \to Vdt\ NP$$

but it could have contained as little as three rules instead:

$$VP \to VPi \mid VPt \mid VPdt \mid Mod\ VP \mid VP\ Adv \mid VP\ PP$$
$$VPdt \to VPd\ NP \mid VPd\ PP$$

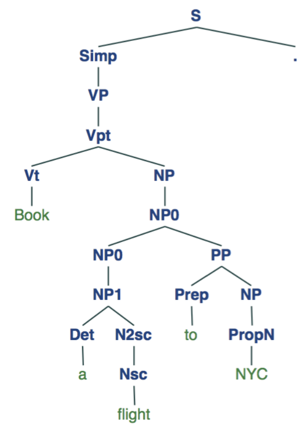

Figure 5: *Book a flight to NYC.*

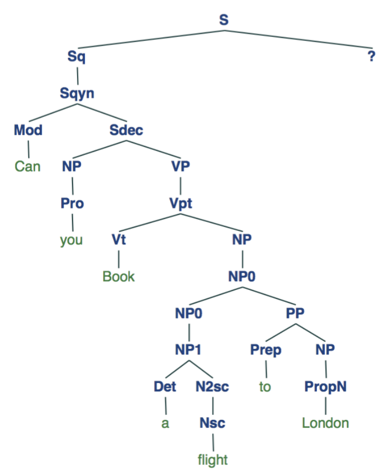

Figure 6: *Can you book a flight to London?*

$$VPd \rightarrow Vdt\ NP$$

b. **Redundancy of rules in the nominal domain**

Redundancy can be also observed among the nominal rules. The NP_0 non-terminal node seems redundant. It's purpose is to allow for recursive generation of prepositional phrases following a noun phrase. This can be achieved by accounting for this kind of recursion within one of the existing NP rules. We could propose two sets of rules to replace the following:

$$\text{NP} \rightarrow \text{PropN} \mid \text{Pro} \mid \text{NP}_0$$
$$\text{NP}_0 \rightarrow \text{NP}_1 \mid \text{NP}_0\ \text{PP}$$

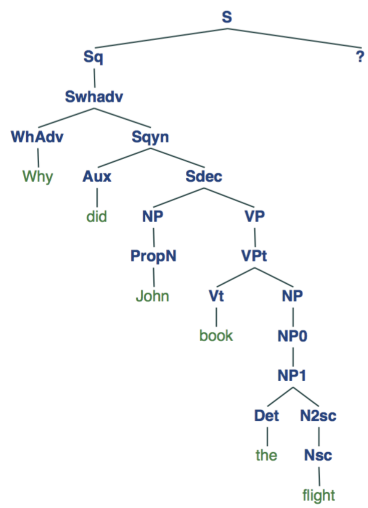

Figure 7: *Why did John book a the flight?*

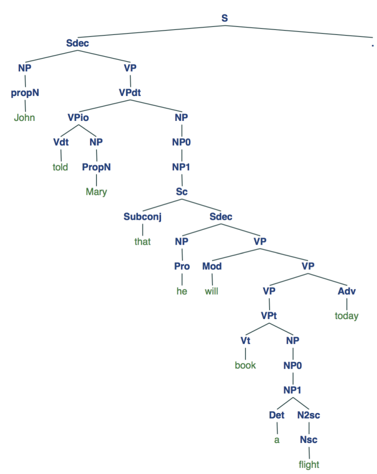

Figure 8: *John told Mary that he will book a flight today.*

$$\mathrm{NP}_1 \rightarrow \mathrm{Det\ N2sc} \mid \mathrm{N2mp} \mid \mathrm{Sc}$$

The first replacement option moves recursive *PP* addition into an *NP* expansion rule:

$$\mathrm{NP} \rightarrow \mathrm{PropN} \mid \mathrm{Pro} \mid \mathrm{NP}_1 \mid \mathrm{NP\ PP}$$
$$\mathrm{NP}_1 \rightarrow \mathrm{Det\ N2sc} \mid \mathrm{N2mp} \mid \mathrm{Sc}$$

Such rearrangement has the advantage of allowing for not only common nouns, but also proper names and pronouns to be modified by *PPs*. This could be desirable in a more comprehensive grammar, but additional restrictions would be required, e.g. to account for pronouns generally not taking *PP* complements. The second option moves the recursion into an NP_1 expansion:

$$NP \rightarrow PropN \mid Pro \mid NP_1$$
$$NP_1 \rightarrow Det\ N2sc \mid N2mp \mid Sc \mid NP_1\ PP$$

It is more conservative in that with these rules in replacing the originals, the grammar would generate the same strings, i.e. the grammars with and without replacement would be weakly equivalent [TODO citation]. The NP_0 node expands to an NP_1 followed by an arbitrary number of *PPs*. The same effect will be achieved by recursively expanding NP_1 into an NP_1 and *PP*, with the benefit of reducing the number of non-terminals.

One might argue that linguistic justification of the existence of the NP_0 can be found in its correspondence with N' node in X-bar syntax. Both are projections above N and below full NP, and allow for adding in principle an unlimited number of modifiers to the head N-complement unit[TODO citation]. However, the recursive expansion of the NP_0 node implies that PPs to the right of head N are adjuncts rather than complements of that N, which is not true. [TODO citation]. The similarity between NP_0 and N' is superficial and can be rejected as motivation for including the former in the grammar.

c. **Dobtfull usefulness of the *Sq* node**

The *Sq* node serves no purpose other than to reflect the conceptual relation between *Sqyn* and *Swhadv*, namely the fact both are non-terminals expanding into questions. This, however, represents only a superficial, if not mistaken, grammatical insight. In fact wh-questions and auxiliary-inversion questions are represented by very different structures (WH-phrases and Aux-phrases respectively[TODO citation]). If the motivation for including the *Sq* node it linguistic, then the reasons seem insufficient. If, one the other hand, the node was included in the grammar to simplify the expansion of *S*, then by analogy we should also include

$$Sd \rightarrow Sdec \mid Simp$$

an change the expansion of *S* to

$$S \rightarrow Sd\ '.' \mid Sq\ '?'$$

d. **Lower case and capitalized versions of terminals.**

It does not seem necessary to include lower case and capitalized versions of the same terminals. In the sample sentences we have seen *Book, Can*, and *Why* being capitalized to account for their sentence-initial positions.

instead of duplicating lexical items, we could simply capitalize the first word of a generated sentence and ignore sentence-initial capitalization during parsing. There would be no ill effects of such a change, since every generated sentence is, by definition, grammatical. Every word which happens to be initial in a sentence produced by the grammar is allowed to be there. The additional confirmation of this fact in form of the word's capitalised version being present in the grammar is unnecessary.

3.2 Overgeneration

a. **Lack of subject-verb agreement**

There's distinction between single/count and plural/mass nouns, but no corresponding distinction in the verbal domain. The problem is somewhat masked by the fact that most of the included verbs are in past tense, where in English there are no subject-verb agreement phenomena. Nevertheless, we can see that the grammar would accept **the fork book a flight.*, a sentence which ignores verb conjugation requirements.

b. **Treating VP as full sentences**

The rules:

$$\text{S} \rightarrow \text{Simp } `.\text{'}$$
$$\text{Simp} \rightarrow \text{VP}$$

imply that any *VP* can be a grammatical sentence on its own. This, however, is true only of *VP* in imperative mood. The grammar, however, allows for treating any *VP*, e.g. **ate today*, as a full sentence. Solving this problem is not possible without substantially expanding the grammar. We would have to distinguish between infinitival and inflected forms of verbs, and extend the set of possible expansions of *VP* accordingly. We could then state that *Simp* subcategories for *VPs* whose head is an infinitival form of a verb. Therefore, we would need the following rules:

$$Vt_inf \rightarrow \text{`Book'} \mid \text{`Tell'} \mid \ldots$$
$$VPi_inf \rightarrow \text{`Eat'} \mid \ldots$$
$$VPt_inf \rightarrow Vt_inf \text{ NP} \mid VPt_inf \text{ Adv} \mid VPt_inf \text{ PP}$$
$$\text{VP} \rightarrow VPt_inf$$
$$\text{Simp} \rightarrow VPt_inf \mid VPi_inf$$

Even then, additional restrictions would be needed to account for characteristics of the imperative mood, e.g. the fact that the second person pronoun needs to be in its reflexive form when used as a direct object of the matrix verb [TODO citation] . Otherwise the grammar would generate sentences such as **Book you a flight to London.* instead of grammatical **Book yourself a flight to London.*

c. **Case agreement**

The grammar does not account for case subcategorisation requirements of verbs. The only corner of the modern English syntax which still exhibits case phenomena is the pronoun system[TODO citation], however the grammar does not include a *ProNom* and a *ProAcc* pre-terminals. Having no distinction between nominative and accusative forms leads to admitting non-grammatical sentences, e.g. **Mary gave **he** a nice salad.* instead of *Mary gave **him** a nice salad.*

d. **Verb and noun subcategorization for pronouns**

Verbs an nouns tend to co-occur with particular pronouns more often than with others. However, the grammar does not account for this preference. For instance, one can *eat with a fork*, *eat with Mary*, or have a *flight to London*, but the grammar also allows for having a **salad to fork* and booking a **flight with NYC.*

e. **Multiple modals**

The grammar allows for an unlimited number of modals to proceed a *VP*. This can lead to generating sentences such as **Can will can can John book a flight?*. However, English limits the acceptable number (up to 4, depending on dialetct), orderings (evidential, epistemic, deontic), and identity (*might* or *may* being the most commonly first in the sequence (Di Paolo 1989)) of verbs in multiple modal constructions. In order to capture these restrictions we would need a whole series of serially connected expansion rules, each introducing one modal of a certain type.

4 Comments for CKY.buildIndices

```
def buildIndices(self,productions):
    """ Creates dictionaries for storing the production rules.
        In each dictionary, the rhs of a rule is the key and and
        a list of all lhs which expand as the rhs is the value."""

    # create dictionaries for unary and binary rules
    self.unary=defaultdict(list)
    self.binary=defaultdict(list)

    for production in productions:
        # separate its right hand-side from its left hand-side
        rhs=production.rhs()
        lhs=production.lhs()
        # the assumption about the rules is that rhs is non-empty
        # and rhs has no more than 2 non-terminals
        assert(len(rhs)>0 and len(rhs)<=2)
        # if the rule is unary,
        # add it's lhs to the unary dictionary under rhs key
```

```
        if len(rhs)==1:
            self.unary[rhs[0]].append(lhs)
        # if the rule is binary,
        # add it's lhs to the binary dictionary under rhs key
        # because of the assertion we know that len(rhs)==2
        else:
            self.binary[rhs].append(lhs)
```

5 Comments for CKY.unary_fill & Cell.unary_update

```
def unary_fill(self):
    """Determine the possible non-terminals that
       each terminal can result from.
       Fill the results in the middle diagonal and print"""

    for r in range(self.n-1):
        # the middle diagonal
        cell=self.matrix[r][r+1]
        # initialize the cell
        word=self.words[r]
        cell.addLabel(word)
        # recursively update the cell
        cell.unary_update(word,self.unary)
        # print out the possible non-terminals for each cell (terminal)
        if self.verbose:
            print "Unary branching rules at node (%s,%s):%s"%(r,r+1,cell.labels())

def unary_update(self,symbol,unaries):
    """Update the cell labels by adding non-terminals
       which expand as the given symbol"""

    # if the symbol is a right hand-side of a unary production rule
    if symbol in unaries:
        # add each of possible corresponding left hand-sides
        # to the cell's labels
        for parent in unaries[symbol]:
            # only add labels that is not in the cell already
            # to avoid the exponential cost of the recognition process
            if parent not in self._labels:
                self.addLabel(parent)
                # a recursive call is needed because in the grammar
                # not all unary productions are terminal productions
                self.unary_update(parent,unaries)
```

6 Comments for CKY.parse & CKY.maybe_build

```
def parse(self,tokens,verbose=False):
    """Initialise a matrix from the sentence,
    then parse it across the middle and upper-right diagonals
    Optional verbose argument controls debugging output, defaults to False"""

    self.verbose=verbose
    self.words = tokens
    # size of the matrix, which equals to number of gaps between words
    self.n = len(self.words)+1
    self.matrix = []
    # We index by row, then column
    #  So Y below is 1,2 and Z is 0,3
    #    1   2   3  ...
    # 0  X   X   Z
    # 1      Y   X
    # 2          X
    # ...
    # create as many rows as there are words
    for r in range(self.n-1):
        # create a row
        row=[]
        for c in range(self.n):
            # populate the row with cells
            if c>r:
                # create only cells corresponding to the upper right half
                # of the matrix
                row.append(Cell())
            else:
                # rest of the matrix is to be filled with None instead of Cell
                row.append(None)
        # add the row to the matrix
        self.matrix.append(row)
    # fill in the middle diagonal
    self.unary_fill()
    # proceed to fill subsequent upper-right diagonals
    # in increasing order of constituent length
    self.binary_scan()
    # if the last cell in row 0 contains the start symbol, return True
    return self.grammar.start() in self.matrix[0][self.n-1].labels()

def maybe_build(self, start, mid, end):
    """Search for the possible combinitions of
       the symbols in two given cells (one from each)
       to match the rhs of binary branching rules"""

    if self.verbose:
        print "Binary branching rules for %s--%s--%s:"%(start,mid,end),
```

```
cell=self.matrix[start][end]

# search from the given cells
for s1 in self.matrix[start][mid].labels():
    for s2 in self.matrix[mid][end].labels():
        # for a binary branching rule match
        if (s1,s2) in self.binary:
            # add all possible non-terminals
            # from the lhs of the rule
            for s in self.binary[(s1,s2)]:
                cell.addLabel(s)
                # add more possible non-terminals
                # because, in the grammar, there are unary rules
                # that can produce non-terminals
                cell.unary_update(s,self.unary)
                if self.verbose:
                    print " %s -> %s %s"%(s, s1,s2),
if self.verbose:
    print
```

*All the comments above are included in *cky.py*

7 Ambiguity Analysis

7.1 "John gave a book to Mary."

Distinct Parse(s): 3

Ambiguity Source: 3 different **VPs** found in (1,6) can pair up with the **NP** in (0,1) to form **Sdecl**. The **VPs** came from the following rules:

$$\boldsymbol{VP \rightarrow VP(1,4)\ PP(4,6)}$$
$$\boldsymbol{VP \rightarrow VPdt \rightarrow VPo(1,4)\ PP(4,6)}$$
$$\boldsymbol{VP \rightarrow VPt \rightarrow Vt(1,2)\ NP(2,6)}$$

Interpretation of ambiguity:
[VP [VP John gave a book] [PP to Mary]]
In this parse, *John gave a book* is a full VP, and *to Mary* is an additional modifier. This an analysis would be correct for a sentence such as *John read a book on the couch*, where *the couch* is not an argument of the verb. In case of *give* both the NP and the PP are arguments.
[VP [VPdt [VPo John gave a book] [PP to Mary]]]
This is the most appropriate parse, in which *give* has a direct object, *a book*, and an indirect one, *Mary*. The only VP in this analysis is the one which correctly encompasses all of the participants of the event.

[VP [VPt [Vt John gave] [NP a book to Mary]]]
This analysis incorrectly treats *to Mary* as forming as a modifier of *a book*. This structure would correctly describe a sentence such as *John ate pasta with tuna*, where the PP really modifies the noun, and the verb really is transitive.

7.2 "John gave Mary a book."

Distinct Parse(s): 1

7.3 "John gave Mary a nice drawing book."

Distinct Parse(s): 2

Ambiguity Source: 2 different **N2sc** found in (4,7) can pair up with the **Det** in (3,4) to form **NP1**. The **N2sc** came from the following rules:

$$\mathit{N2sc} \rightarrow \mathit{Adj(4,5)}\ \mathit{N2sc(5,7)}$$
$$\boldsymbol{N2sc(5,7) \rightarrow N3(5,6)\ Nsc(6,7)}$$
$$\boldsymbol{N2sc(5,7) \rightarrow Adj(5,6)\ N2sc(6,7)}$$

Interpretation of ambiguity:
[N2sc [Adj drawing] [Nsc book]]
The grammar lists *drawing* as an adjective, so the unit *drawing book* can be analysed as a noun modified by an adjective, in the way *fragile* is a modifier in *fragile cup*.
[N2sc [N3 [N drawing]] [Nsc book]]
In this parse both *drawing* and *book* are treated as nouns, and N2sc represents a noun-noun compound, in the way *tea cup* is a compound of two nouns. This seems to be a correct interpretation, given how popular such structures are in English, and that there is no evidence of *drawing* being an adjective (e.g. there are no comparative forms **a drawinger book*).

7.4 "John ate salad with mushrooms with a fork."

Distinct Parse(s): 5

Ambiguity Source: 5 different **VP** found in (1,8) can pair up with the **NP** in (0,1) to form **Sdecl**. The **VP** came from the following rules:

$$\boldsymbol{VP \rightarrow VP(1,3)\ PP(3,8)}$$
$$\mathit{VP} \rightarrow \mathit{VP(1,5)}\ \mathit{PP(5,8)}$$
$$\boldsymbol{VP(1,5) \rightarrow VP(1,3)\ PP(3,5)}$$
$$\boldsymbol{VP(1,5) \rightarrow VPt \rightarrow Vt(1,2)\ NP(2,5)}$$
$$\mathit{VP} \rightarrow \mathit{VPt} \rightarrow \mathit{Vt(1,2)}\ \mathit{NP(2,8)} \rightarrow \mathit{NP0}$$
$$\boldsymbol{NP0 \rightarrow NP0(2,3)\ PP(3,8)}$$
$$\boldsymbol{NP0 \rightarrow NP0(2,5)\ PP(5,8)}$$

Interpretation of ambiguity:

7.5 "Book a flight to NYC."

Distinct Parse(s): 2

Ambiguity Source: 2 different **Simp** found in (0,5) can pair up with the **'.'** in (5,6) to form **S**. The **Simp** came from the following rules:

$$\textit{Simp} \rightarrow \textit{VP}$$
$$\boldsymbol{\textit{VP} \rightarrow \textit{VP(0,3) PP(3,5)}}$$
$$\boldsymbol{\textit{VP} \rightarrow \textit{VPt} \rightarrow \textit{Vt(0,1) NP(1,5)}}$$

Interpretation of ambiguity:

7.6 "Can you book a flight to London?"

Distinct Parse(s): 2

Ambiguity Source: 2 different **Sq** found in (0,7) can pair up with the **'?'** in (7,8) to form **S**. The **Sq** came from the following rules:

$$\textit{Sq} \rightarrow \textit{Sqyn} \rightarrow \textit{Mod(0,1) Sdecl(1,7)}$$
$$\textit{Sdecl(1,7)} \rightarrow \textit{NP(1,2) VP(2,7)}$$
$$\boldsymbol{\textit{VP(2,7)} \rightarrow \textit{VP(2,5) PP(5,7)}}$$
$$\boldsymbol{\textit{VP(2,7)} \rightarrow \textit{VPt} \rightarrow \textit{Vt(2,3) NP(3,7)}}$$

Interpretation of ambiguity:

7.7 "Why did John book the flight?"

Distinct Parse(s): 1

7.8 "John told Mary that he will book a flight today."

Distinct Parse(s): 3

Ambiguity Source: 3 different **Sdecl** found in (0,10) can pair up with the **'.'** in (10,11) to form **S**. The **Sdecl** came from the following rules:

$$\textit{Sdecl} \rightarrow \textit{NP(0,1) VP(1,10)}$$
$$\boldsymbol{\textit{VP(1,10)} \rightarrow \textit{VP(1,9) Adv(9,10)}}$$
$$\textit{VP(1,10)} \rightarrow \textit{VPdt} \rightarrow \textit{VPio(1,3) NP(3,10)}$$
$$\textit{NP(3,10)} \rightarrow \textit{NP0} \rightarrow \textit{NP1} \rightarrow \textit{Sc} \rightarrow \textit{Subconj(3,4) Sdecl(4,10)}$$

$$Sdecl(4,10) \rightarrow NP(4,5)\ VP(5,10)$$
$$\boldsymbol{VP(5,10) \rightarrow Mod(5,6)\ VP(6,10)}$$
$$\boldsymbol{VP(5,10) \rightarrow VP(5,9)\ Adv(9,10)}$$

Interpretation of ambiguity:

8 Generating a parse tree

8.1 CKY parsing

Up to now, the program is a CKY recognizer that can only determine whether a string belongs to the language generated by the given grammar. In order to extend it into a CKY parser that can construct a parse tree, back-pointers, which records the one or more ways in which a constituent spanning (i,j) can be made from constituents spanning (i,k) and (k,j), will be attached to each label in the table cells. Therefore, after successful parsing, we can traceback from the Start Symbols (S) in the top-right-hand corner of the matrix to every terminals from the string. The final result is then a shared-forest of possible parse trees, where common tree parts are factored between the various parses [1].

8.2 Design and implementation

First, a Label class was defined to describe the labels in each cell which was represented by String. Benefited from the flexible data structure in Python, this class can be initialized with at most 4 parameters, as shown below.

List of parameters:

- **symbol** the represented label (required)
- **start** row number of the cell (optional)
- **end** column number of the cell (optional)
- **rhs** list of right-hand-sides of the production rules which generate the target symbol; mid value is also included in the list, which refers to the source cells from which the constituents of right-hand-sides came (optional)

The Label class can take different number of parameters depending on where the label came from, such as the left-hand-sides of rules and words from the input string. For instance, if a symbol was added according to a unary rule, the row number and column number will be the same as its parent label which we already knew. For the labels of words, which will be the leaf nodes of a parse tree, the value of right-hand-side will be omitted (not available).
Moreover, a number of the existing CKY and Cell methods were edited to construct/exploit this richer label structure. Three utility functions for listing,

Parameter \ Source Type	binary rule	unary rule	word
symbol	✓	✓	✓
start	✓		
end	✓		
rhs	✓	✓	

Table 1: Parameter Structure for Different Labels

searching and updating label list are also provided.

After that, the backtrack algorithm was implemented in two functions:

- **update_trees** to update all the possible subtrees of the give label in a cell (In this implementation, the function will be ended after the first subtree was returned)
- **update_children** to construct NLTK tree nodes according to given rules

update_trees and *update_children* will call each other recursively to generate the whole parse tree.

8.3 Summary

To generate the first parse tree, we only need to pass the Start Symbols (S) in the top-right-hand corner of the matrix to *update_trees* method and it will return the result after the first parse tree was found. However, in order to return all the possible parse trees, the Start Symbols (S) in the top-right-hand corner of the matrix should be treated differently from other labels and *update_trees* method should return a list of possible subtrees to construct all the ambiguous result. The detail of generating all parse trees will not be discussed in this report.

References

[1] B. Lang, "Recognition can be harder than parsing," *Computational Intelligence*, vol. 10, no. 4, pp. 486–494, 1994.