CKY Parser for Japanese

ICS661 – Fall 2012

Final Project

Ryan Bungard

1. Introduction

The Cocke-Kasami-Younger (CKY) algorithm is an entirely generative approach for syntactically parsing individual sentences in a natural language. It requires a user-defined context-free grammar (CFG), which is a set of rules that express the way a symbol of a language can be grouped or ordered together along with a lexicon of words and symbols (Jurafsky and Martin 2009: 387). The symbols consist of two types: terminals and non-terminals. Terminals represent the words in a language while non-terminals symbolize abstract units that organize the terminals into full sentence structures (388). CFG non-terminals tend to correspond with the grammatical categories or constituents that form the hierarchies of phrases, clauses, and sentences in the analysis of natural grammar. An example set of rules forming a simple English sentence with a CFG is below.

Terminals: {I, ran, quickly}
Non-terminals: {S, NP, Pronoun, VP, Verb, Adv}

S → NP VP Adv
NP → Pronoun
Pronoun → I
VP → Verb
Verb → ran
Adv → quickly

In order for the CKY algorithm to function, the CFG is converted into Chomsky normal form (CNF), where rules are only allowed a binary set of symbols on the right hand side (RHS). A conversion of the example above is shown with the following:

S → X1 Adv
NP → Pronoun
Pronoun → I
VP → Verb
Verb → ran
Adv → quickly
X1 → NP VP

This conversion is achieved by changing any mixed rules containing both terminals and non-terminals, removing unit productions containing only one non-terminal on the right hand side, and then making every rule binary. The CKY algorithm then utilizes a matrix to search non-terminal binary combinations at each word of the sentence. It proceeds to do so in a bottom-up fashion while filling in cells of the matrix with every phrase it finds until finally recognizes whether the whole string is a sentence or not. Pseudo-code for the algorithm is provided by Jurafsky and Martin (440).

function CKY-PARSE(*words, grammar*) **returns** *table*

for $j \leftarrow$ **from** 1 **to** LENGTH(*words*) **do**
  $table[j-1, j] \leftarrow \{A \mid A \rightarrow words[j] \in grammar\}$
  for $i \leftarrow$ **from** $j-2$ **downto** 0 **do**
    for $k \leftarrow i+1$ **to** $j-1$ **do**
      $table[i,j] \leftarrow table[i,j] \cup$
        $\{A \mid A \rightarrow BC \in grammar,$
          $B \in table[i,k],$
          $C \in table[k,j]\}$

Figure 1. Pseudo-code for the CKY algorithm.

This project sets out to implement and test the CKY algorithm against a small grammar created for natural Japanese sentences. First, the program will accept a user-defined grammar file that will serve as the language model. Then, the program accepts input strings through the command line and determines if the string is a sentence as defined by the language model. Finally, if the program recognizes a sentence, it will display in bracket notation all parses it encountered from the algorithm. This bracket notation will be as linguistic-friendly as possible.

2. Description

2.1 Resources

This program was developed in Python 2.7 and must be run from the Python interpreter on the command line, especially one that supports Unicode input. I used Ubuntu 10.04 since I had it readily available. A previous implementation for English sentences used Python, so ultimately this project is a modification of that existing code. Naturally, some of Python's native collection data structures were used, such as dictionaries (associative arrays), sets (an array storing unique, non-duplicated entries), and lists (typical numerically indexed arrays). UTF-8 is the primary encoding for the scripts and regular expressions.

Additionally, the MeCab morphological analyzer for Japanese was used. In this case, the binding library for Python and at least one appropriately formatted corpus file is required for the library to operate. MeCab was developed in C, thus this program also requires the Simplified Wrapper

and Interface Generator (SWIG), a tool for wrapping C/C++ functionality into other programming languages (i.e. Python). The IPA dictionary contains parameter estimations modeled with the IPA corpus and Conditional Random Fields (CRF). It is intended to be used as a general morphological analyzer without focus on any specific domain of language.

2.2 Usage

To run the program in the present working directory, the following command is used:

python cky_driver.py jp_nocnf_input.txt

The script accepts an additional command line argument for the grammar file.

Enter a “1” if the input file is in plain CFG format or “2” if the grammar is already in CNF.

You are then prompted to enter a Japanese sentence. Please keep in mind that this program was designed to work for Japanese sentences in UTF-8.

To end the program, enter the EOF character. In Unix, this is Ctrl-D.

2.3 Implementation

The main program is divided into three files.
Driver: cky_driver.py
Grammar-related classes: CFG.py
CKY algorithm class: CKY.py

A general overview of the classes is described below:

- CFG
  - Parent class of a grammar.
  - Stores a list of rules in dictionaries as well as non-terminals and terminals their respective sets.
- CNF
  - Subclass of CFG.
  - Public and pseudo-private methods for CNF conversion including mixed rule manipulation, unit production removal, and binary rule conversion.
  - Stores a collection of new nodes introduced from any mixed rule or binary conversions, which is later used to create a bracket notation accurate with the original input grammar.
  - Contains a CNF_search_tree for searching RHS symbols and returning the corresponding left hand side (LHS) symbol. This is used when comparing non-terminal nodes in the CKY algorithm.
- Rule
  - Class for representing of individual grammar rules.

    - Maintains unit production history for newly converted rules. Used for bracket notation output in the CKY parser.
- CNF_search_tree
    - Tree of dictionaries used to search the LHS associated with a given rule or terminal node. LHS symbols are stored in a set at the end of a traversal.
- CKY
    - Parser containing the matrix (2D list) and logic for running the CKY algorithm.
    - Segments input string into words for parsing via MeCab library.
    - Contains accessor that prints out a rough from of the CKY table and, more importantly, the bracket notation of all possible parses.
- CKY_indexer
    - Tracks symbols stored in each cell of the CKY parse table and the RHS used to create it. Used as a reference for bracket notation formation.

The driver takes in the grammar's text file name as a command line argument and stores its representation as a CNF object. The constructor builds a dictionary of Rule objects and then sets of non-terminals and terminals through the constructor of the inherited parent class, CFG. Beyond identifying and storing the types of symbols during instantiation, the CNF object mutates the data into CNF form if so indicated by the user. Methods proceed through conversion by changing the current grammar rather than copying. The modified steps are below (Jurafsy and Martin, 437):

1. ~~Copy all conforming rules to the new grammar unchanged.~~
2. Convert terminals within rules to dummy non-terminals.
3. Convert unit-productions.
4. Make all rules binary ~~and add them to new grammar~~.

Mixed rules introduce terminals as a new dummy non-terminal, or in another sense, linguistic category of the form "CAT#." The unit production historical data is stored as a list within the individual Rule object for easy retrieval and rebuilding when displaying bracket notation true to the original grammar. The LHS of new rules introduced by CNF binary conversion are stored in a list in the form "X#." When this is finished, the CNF object creates a search tree from all rules so that the CKY algorithm can effectively compare non-terminals and produce the correct LHS. The program then prints out the CNF grammar, non-terminals, and terminals.

Afterwards, the CKY object is instantiated with the CNF object as an argument for construction. An indexer class is also instantiated within the CKY object, which essentially contains backpointers to all of the nodes necessary for filling the cells of the CKY parse table. The driver asks the user for an input string which is in turn segmented by a method that calls the morphological analyzer, MeCab. MeCab returns a string of words separated by spaces, leading to easy segmentation by a regular expression. Once these words are segmented, the CKY algorithm

can proceed to parse normally. Upon finishing, the object prints out a copy of the parse table, a verdict of “S” or “Not S,” and if it was an S, the recursively constructed bracket notation of all parses that resulted in an S.

鉛筆	で	手紙	を	書く
[0,0] Nominal, Noun, CommonNoun	[0,1] PP	[0,2]	[0,3]	[0,4] VP, S, S
	[1,1] Postposition	[1,2]	[1,3]	[1,4]
		[2,2] Nominal, Noun, CommonNoun	[2,3] AccNP, NP	[2,4] VP, S
			[3,3] AccPart	[3,4]
				[4,4] VP, S, Verb, NatVerb

Figure X. Visualization of the CKY parse table for the example above. Indices are altered from Figure X to accommodate actual implementation.

2.4 Implementation Challenges and Decisions

Word segmentation in Japanese was undoubtedly a challenge that could not be resolved with regular expressions alone. Segmented words are a crucial precondition for the CKY algorithm to function. Unlike English, written Japanese sentences combine words into a string of characters without division by spaces or any overt markers. At the phrase level, it is possible to separate with a comma or other form of punctuation, but this happens relatively infrequently. Another approach could be to identify words by the three sets of characters used in Japanese orthography. Kanji, adopted Chinese characters, are used for most native Japanese words with semantic value, hiragana is used for both native words and grammatical markers such as case and inflections, and katakana covers an assortment of types like foreign loan words, onomatopoeia, and emphasis. However, producing segmentation with only this information proves fruiteless (Example 1).

[私] - [は] - [飛行機] - [で] - [東京] - [に] - [行 - く] - [こ - と] - [を] - [好 - む]。

Example 1. Characters are separated with dashes by kanji/hiragana distinction. The actual words are bracketed.

Consequently, I was coerced into adapting a more sophisticated method to arrive at the desired segmentations. The MeCab morphological analyzer suited this role. MeCab accepts an argument that formats the output with space-separation between morphemes. Since Japanese is an agglutinative language with easily segmented morphemes, most words with any type of conjugation would contain space-separation between the root and additional inflections. My program was designed to handle constituents at the word-level, so I did not include words with morphological inflections where possible. In practical use, the part-of-speech (POS) information provided by the morphological analyzer should be used to either piece the separated morphemes back into such words or assign categories to form sound syntactic structures. The limitation of the parser to recognize a string of hiragana characters from a semantic word to a grammatical constituent, i.e. case, tense/aspect markers, etc., is reason enough to use the tools here while avoiding a standard string search algorithm that considers only pre-defined terminals. Furthermore, morphemes, not just words, may be considered as equal with other parts-of-speech which could introduce more abstract phrase types (AP for aspectual phrase, TP for tense phrase, etc.) in some methods of syntactic analyses (Haegeman 1994: 598). In such a case, it would have to take advantage of the POS tagging information by the analyzer.

Other options were available for morphological analysis. The NLTK library contains a “reader” package with a module “chasen,” which makes used of the ChaSen morphological analyzer built on Hidden Markov Models (HMM) (http://nltk.org/api/nltk.corpus.reader.html#module-nltk.corpus.reader.chasen). The authors of MeCab state that CRF-modeled analyzers possess two strong advantages over Markov model implementations (HMMs and Maximum Entropy Markov Models [MEMM]). First, it avoids the label bias. Even if the right path is trained on a Markov model, another path may be chosen by default due to fewer outgoing transitions from the current state (thus less probability to divide with other possibilities). It also minimizes length bias, which the MEMM falls victim too. Short paths, or parses with a small number of tokens, are preferred to longer paths creating lower entropy. CRFs avert both biases because they assign “a single exponential model for the joint probability of the entire path.” For instance, an MEMM contains “a sequential combination of exponential models, each of which estimates a conditional probability of next tokens given the current state.” (Kudo, et al. 2004)

With these claimed advantages, and the fact that I have not encountered a counter argument as of yet, I chose to stick with the MeCab library. In addition, the test grammar is kept small with basic words to ensure the core functionality of the CKY parser is sound. The difference in segmentation would be negligible for this project, but it would seem feasible to test ChaSen in comparison with MeCab if one endeavors to settle on a final parser implementation.

MeCab also required Python version 2.x. Ultimately, I was forced to convert my code from Python 3.3 to 2.7 in the end due to the required version for the MeCab binding library, which is no different than NLTK. Python 3.3 was convenient in the sense that the script and input strings are encoded in UTF-8 by default. In Python 2.7, it is mandatory to use explicit functions for

encoding and decoding strings as well as flagging regular expression functions to expect Unicode.

Framing the bracketed notation true to the original grammar proved tedious. Displaying the notation in converted CNF could be accomplished by a simple recursive algorithm implemented in the CKY class using only the information provided by the CKY Indexer. However, it seemed inconvenient to regenerate the hierarchy of rules dictated by the original grammar without storing some history of changes in the CNF class. Particularly, storing the chain of unit productions required some planning. At first, I considered changing the CNF to keep unit productions and handle them appropriately in the CKY parse. However, I would need a data structure in the cell of the CKY table to distinguish the resolved LHS node from a list of unit production nodes that fall under it. That list could then be retrieved when forming the bracket notation. Then again, my CNF Search Tree implementation would have to change to accommodate these rules, and for a data structure central to an already working implementation of the CKY algorithm, this posed risky. This is why I decided to maintain a list of each node in a unit production chain under a Rule object for the new rule that replaced them.

3. Analysis

As far as the scope of the project, the parser recognizes all sentences it should given a user-defined grammar with all input in UTF-8. After all, the parser is only as good as the language model applied. Examples for recognized input sentences and non-sentences are presented in Appendix A-3.

Again, the grammar contained words in their basic forms (ex. verbs are all in their imperfective form) for easy segmentation by the morphological analyzer. Future work would include a means to combine conjugated morphemes back into a single word.

The test CFG, as small as it is, contained many similar and recursive rules to address just a few variations of slightly different syntactic constructions. This phenomena is not specific to the CKY algorithm, but it is a concern for any algorithm relying on a CFG. For example, I have three types of verbals that resolve to VP.

VP > GaVerb
VP > Verb
Verb > NatVerb
Verb > VerbalNoun する

The GaVerb exists for a special class of verbs that use が as an accusative case marker rather than the typical を. The NatVerb is used for all native Japanese verbs. The VerbalNoun する rule

is another type of verbal that functions just as the NatVerb categories with case marking and conjugations, but instead they use Sino-Japanese nouns that precede the verb する. This is just a sample of the verbal intricacies that have to be captured by the grammar for this algorithm to work successfully. What was presented here is purely syntactic, but you could include other linguistic dependencies such as animacy, tense/aspect representation, etc. to factor into other possible syntactic variations. The categorical and consequent rule counts rise rather quickly in effort to cover all combinations, and as a result, it makes non-context-free alternatives look more promising.

The method for developing the bracket notation is also successful. The historical information saved throughout every step of the CNF conversion and CKY parse is effectively used to build an accurate notation based on the original CFG. Please refer to Appendix A-1 for the grammar and Appendix A-2 for example output in bracket notation.

4. Conclusion

Overall, the program worked as intended with the strict scope of a syntactic parser. I learned potential ways that a morphological analyzer's output could contribute to the next step, syntactic parsing, in the natural language processing paradigm. It began more as a solution for segmenting words in Japanese, but it brought to light how POS data should interact with a grammar and how that grammar should be designed to fit that data. It was also an introduction to a new statistical mode (CRFs) beyond the Markov models presented in class.

Handling UTF-8 helped me develop an appreciation for programmers working with languages other than English, especially for less commonly used encodings. It made me understand how much I have taken for granted the built-in string functions and regular expressions under a high-level language's default API. At least for Python, things seem to be progressing for UTF-8 where others enjoy the benefits of not having to know exactly when and how to explicitly define an encoding or decoding, for better or for worse. It is then a matter of waiting for NLP developers to upgrade their libraries to interface with the newest versions.

There is still room to explore how this parser would size up against more popular options, such as any parser provided by the NLTK library. It is difficult to make a fair comparison at this stage because both this parser and anything from NLTK's "parse" package would have to depend on the same format of grammar under the same algorithm. I have not found a simple implementation of CKY under this package yet. However, it may be interesting to test the simple CKY algorithm against others like the Shift/Reduce and Earley algorithms, as well as the probability based CFG algorithms, which are all available under NLTK (http://nltk.org/api/nltk.parse.html).

Bibliography

Haegeman, Liliane. 1994. *Government & Binding Theory*. Malden: Blackwell Publishing.

Jurafsky, Daniel, and James H. Martin. 2009. *Speech and Language Processing*. New Jersey: Pearson Education, Inc.

Kudo, Taku, Kaoru Yamamoto, and Yuji Matsumoto. 2004. Applying Conditional Random Fields to Japanese Morphological Analysis. *Joho Shori Gakkai Kenkyu Hokoku [Data Processing Society Research Report]* 47: 89-96.

MeCab: Yet Another Part-of-Speech and Morphological Analyzer. http://mecab.googlecode.com/svn/trunk/mecab/doc/index.html. (accessed December 03, 2012).

NLTK 2.0 Documentation. http://nltk.org/. (accessed December 03, 2012).

Python v2.7.3. Documentation. http://docs.python.org/2/. (last accessed December 03, 2012).

SWIG. http://www.swig.org/exec.html. (accessed December 01, 2012).

Appendix

A-1: Test Context-Free Grammar

S_inform > S よ
S_ques > S か
S > VP
S > NomNP VP
S > AddNP VP
S > TopNP S
S > PP S
NP > NomNP
NP > AccNP
NP > DatNP
NP > AddNP
NomNP > Nominal NomPart
AccNP > Nominal AccPart
GaAccNP > Nominal GaAccPart
DatNP > Nominal DatPart
TopNP > Nominal TopPart
AddNP > AddNominal
PP > Nominal Postposition
Nominal > Noun
Nominal > Nominal GenPart Noun
Nominal > Det Noun
Nominal > Nominal Conjunct Nominal
Nominal > InterrogNoun
Nominal > S FormalNoun
AddNominal > Nominal Additive
AddNominal > AddNominal Nominal Additive
Noun > Pronoun
Noun > CommonNoun
Noun > ProperNoun
VP > AccNP Verb
VP > GaAccNP GaVerb
VP > Verb
VP > GaVerb
VP > PP VP
VP > Nominal Copula
Copula > だ
Verb > NatVerb
Verb > VerbalNoun する
NatVerb > 含む | 好む | 行く | 読む | 送る | 書く | 買う | する
GaVerb > 分かる | 出来る | できる
VerbalNoun > 予約 | 旅行
Det > この | その | あの
Pronoun > 私 | 彼女 | 彼
CommonNoun > 本 | 飛行機 | ご飯 | お金 | 鉛筆 | 手紙
ProperNoun > 東京 | 英語 | 日本語
ProperNoun > NameNoun さん

ProperNoun > NameNoun
NameNoun > 村上 | ライアン
FormalNoun > の | こと | ところ
InterrogNoun > 何 | 誰 | どこ
PP > Nominal Postposition
Postposition > から | に | へ | まで | までに | で
NomPart > が
AccPart > を
GaAccPart > が
GenPart > の
DatPart > に
TopPart > は
Conjunct > と | や
Additive > も

A-2: Full Output Example

Input file is set to: jp_input_nocnf.txt
Is this input formatted in: (1) CFG (2) CNF ? 1
Grammar:
AccNP > Nominal AccPart
AccPart > を
AddNP > Nominal Additive
AddNP > X3 Additive
AddNominal > Nominal Additive
AddNominal > X3 Additive
Additive > も
CAT1 > か
CAT2 > さん
CAT3 > する
CAT4 > よ
CommonNoun > 本 | 飛行機 | ご飯 | お金 | 鉛筆 | 手紙
Conjunct > と | や
Copula > だ
DatNP > Nominal DatPart
DatPart > に
Det > この | その | あの
FormalNoun > の | こと | ところ
GaAccNP > Nominal GaAccPart
GaAccPart > が
GaVerb > 分かる | 出来る | できる
GenPart > の
InterrogNoun > 何 | 誰 | どこ
NP > Nominal NomPart
NP > Nominal AccPart
NP > Nominal DatPart
NP > Nominal Additive
NP > X3 Additive
NameNoun > 村上 | ライアン

NatVerb > 含む | 好む | 行く | 読む | 送る | 書く | 買う | する
NomNP > Nominal NomPart
NomPart > が
Nominal > X1 Noun
Nominal > Det Noun
Nominal > X2 Nominal
Nominal > S FormalNoun
Nominal > 私 | 彼女 | 彼
Nominal > 本 | 飛行機 | ご飯 | お金 | 鉛筆 | 手紙
Nominal > 東京 | 英語 | 日本語
Nominal > NameNoun CAT2
Nominal > 村上 | ライアン
Nominal > 何 | 誰 | どこ
Noun > 私 | 彼女 | 彼
Noun > 本 | 飛行機 | ご飯 | お金 | 鉛筆 | 手紙
Noun > 東京 | 英語 | 日本語
Noun > NameNoun CAT2
Noun > 村上 | ライアン
PP > Nominal Postposition
PP > Nominal Postposition
Postposition > から | に | へ | まで | までに | で
Pronoun > 私 | 彼女 | 彼
ProperNoun > 東京 | 英語 | 日本語
ProperNoun > NameNoun CAT2
ProperNoun > 村上 | ライアン
S > NomNP VP
S > AddNP VP
S > TopNP S
S > PP S
S > AccNP Verb
S > GaAccNP GaVerb
S > 含む | 好む | 行く | 読む | 送る | 書く | 買う | する
S > VerbalNoun CAT3
S > 分かる | 出来る | できる
S > PP VP
S > Nominal Copula
S_inform > S CAT4
S_ques > S CAT1
TopNP > Nominal TopPart
TopPart > は
VP > AccNP Verb
VP > GaAccNP GaVerb
VP > PP VP
VP > Nominal Copula
VP > 含む | 好む | 行く | 読む | 送る | 書く | 買う | する
VP > VerbalNoun CAT3
VP > 分かる | 出来る | できる
Verb > VerbalNoun CAT3
Verb > 含む | 好む | 行く | 読む | 送る | 書く | 買う | する
VerbalNoun > 予約 | 旅行
X1 > Nominal GenPart

X2 > Nominal Conjunct
X3 > AddNominal Nominal

Non-terminals:
['AccNP', 'AccPart', 'AddNP', 'AddNominal', 'Additive', 'CAT1', 'CAT2', 'CAT3', 'CAT4', 'CommonNoun', 'Conjunct', 'Copula', 'DatNP', 'DatPart', 'Det', 'FormalNoun', 'GaAccNP', 'GaAccPart', 'GaVerb', 'GenPart', 'InterrogNoun', 'NP', 'NameNoun', 'NatVerb', 'NomNP', 'NomPart', 'Nominal', 'Noun', 'PP', 'Postposition', 'Pronoun', 'ProperNoun', 'S', 'S_inform', 'S_ques', 'TopNP', 'TopPart', 'VP', 'Verb', 'VerbalNoun']

Terminals:
あの--お金--か--から--が--こと--この--ご飯--さん--する--その--だ--で--できる--と--ところ--どこ--に--の--は--へ--まで--までに--も--や--よ--を--ライアン--予約--何--出来る--分かる--含む--好む--彼--彼女--手紙--旅行--日本語--書く--本--村上--東京--私--英語--行く--読む--誰--買う--送る--鉛筆--飛行機

Please enter a sentence:
彼が鉛筆で手紙を書く。
彼--が--鉛筆--で--手紙--を--書く
[0,0] ['Nominal', 'Noun', 'Pronoun']
[0,1] ['NP', 'NomNP', 'GaAccNP']
[1,1] ['NomPart', 'GaAccPart']
[0,2] []
[1,2] []
[2,2] ['Nominal', 'Noun', 'CommonNoun']
[0,3] []
[1,3] []
[2,3] ['PP']
[3,3] ['Postposition']
[0,4] []
[1,4] []
[2,4] []
[3,4] []
[4,4] ['Nominal', 'Noun', 'CommonNoun']
[0,5] []
[1,5] []
[2,5] []
[3,5] []
[4,5] ['AccNP', 'NP']
[5,5] ['AccPart']
[0,6] ['S']
[1,6] []
[2,6] ['VP', 'S', 'S']
[3,6] []
[4,6] ['VP', 'S']
[5,6] []
[6,6] ['VP', 'S', 'Verb', 'NatVerb']

S found!
Possible parses: 1
[S [NomNP [Nominal [Noun [Pronoun 彼]]][NomPart が]][VP [PP [Nominal [Noun [CommonNoun 鉛筆]]][Postposition で]][VP [AccNP [Nominal [Noun [CommonNoun 手紙]]][AccPart を]][Verb [NatVerb 書く]]]]

Please enter a sentence:

EOF hit. Ending program.

A-3: Sentences and Non-Sentences

```
--------------------------------
*************************
SENTENCES recognized correctly.
*************************
--------------------------------

行くよ！
行く--よ
Possible parses: 1
[S_inform [S [VP [Verb [NatVerb 行く]]]][CAT4 よ]]

------------------------------
彼がライアンだ。
彼--が--ライアン--だ
Possible parses: 1
[S [NomNP [Nominal [Noun [Pronoun 彼]]][NomPart が]][VP [Nominal [Noun [ProperNoun [NameNoun ライア
ン]]]][Copula だ]]]

------------------------------
村上さんの本を読む。
村上--さん--の--本--を--読む
Possible parses: 1
[S [VP [AccNP [Nominal [Nominal [Noun [ProperNoun [NameNoun 村上][CAT2 さん]]]][GenPart の][Noun
[CommonNoun 本]]][AccPart を]][Verb [NatVerb 読む]]]]

------------------------------

飛行機がご飯を含む。
飛行機--が--ご飯--を--含む
Possible parses: 1
[S [NomNP [Nominal [Noun [CommonNoun 飛行機]]][NomPart が]][VP [AccNP [Nominal [Noun [CommonNoun ご
飯]]][AccPart を]][Verb [NatVerb 含む]]]]

------------------------------

飛行機で旅行する。
飛行機--で--旅行--する
Possible parses: 2
[S [VP [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][VP [Verb [VerbalNoun 旅行][CAT3 す
る]]]]]
```

[S [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][S [VP [Verb [VerbalNoun 旅行][CAT3 する]]]]]

村上さんが英語が出来る。
村上--さん--が--英語--が--出来る
Possible parses: 1
[S [NomNP [Nominal [Noun [ProperNoun [NameNoun 村上][CAT2 さん]]]][NomPart が]][VP [GaAccNP [Nominal [Noun [ProperNoun 英語]]][GaAccPart が]][GaVerb 出来る]]]

ご飯は、何を買う？
ご飯--は--何--を--買う
Possible parses: 1
[S [TopNP [Nominal [Noun [CommonNoun ご飯]]][TopPart は]][S [VP [AccNP [Nominal [InterrogNoun 何]][AccPart を]][Verb [NatVerb 買う]]]]]

日本語で本や手紙を読むことができる。
日本語--で--本--や--手紙--を--読む--こと--が--できる
Possible parses: 10
[S [PP [Nominal [Noun [ProperNoun 日本語]]][Postposition で]][S [NomNP [Nominal [Nominal [Noun [CommonNoun 本]]][Conjunct や][Nominal [S [VP [AccNP [Nominal [Noun [CommonNoun 手紙]]][AccPart を]][Verb [NatVerb 読む]]]][FormalNoun こと]]][NomPart が]][VP [GaVerb できる]]]]

[S [VP [PP [Nominal [Noun [ProperNoun 日本語]]][Postposition で]][VP [GaAccNP [Nominal [Nominal [Noun [CommonNoun 本]]][Conjunct や][Nominal [S [VP [AccNP [Nominal [Noun [CommonNoun 手紙]]][AccPart を]][Verb [NatVerb 読む]]]][FormalNoun こと]]][GaAccPart が]][GaVerb できる]]]]

[S [PP [Nominal [Noun [ProperNoun 日本語]]][Postposition で]][S [VP [GaAccNP [Nominal [Nominal [Noun [CommonNoun 本]]][Conjunct や][Nominal [S [VP [AccNP [Nominal [Noun [CommonNoun 手紙]]][AccPart を]][Verb [NatVerb 読む]]]][FormalNoun こと]]][GaAccPart が]][GaVerb できる]]]]

[S [PP [Nominal [Noun [ProperNoun 日本語]]][Postposition で]][S [NomNP [Nominal [S [VP [AccNP [Nominal [Nominal [Noun [CommonNoun 本]]][Conjunct や][Nominal [Noun [CommonNoun 手紙]]]][AccPart を]][Verb [NatVerb 読む]]]][FormalNoun こと]][NomPart が]][VP [GaVerb できる]]]]

[S [VP [PP [Nominal [Noun [ProperNoun 日本語]]][Postposition で]][VP [GaAccNP [Nominal [S [VP [AccNP [Nominal [Nominal [Noun [CommonNoun 本]]][Conjunct や][Nominal [Noun [CommonNoun 手紙]]]][AccPart を]][Verb [NatVerb 読む]]]][FormalNoun こと]][GaAccPart が]][GaVerb できる]]]]

[S [PP [Nominal [Noun [ProperNoun 日本語]]][Postposition で]][S [VP [GaAccNP [Nominal [S [VP [AccNP [Nominal [Nominal [Noun [CommonNoun 本]]][Conjunct や][Nominal [Noun [CommonNoun 手紙]]]][AccPart を]][Verb [NatVerb 読む]]]][FormalNoun こと]][GaAccPart が]][GaVerb できる]]]]

[S [NomNP [Nominal [S [VP [PP [Nominal [Noun [ProperNoun 日本語]]][Postposition で]][VP [AccNP [Nominal [Nominal [Noun [CommonNoun 本]]][Conjunct や][Nominal [Noun [CommonNoun 手紙]]]][AccPart を]][Verb [NatVerb 読む]]]]][FormalNoun こと]][NomPart が]][VP [GaVerb できる]]]

[S [VP [GaAccNP [Nominal [S [VP [PP [Nominal [Noun [ProperNoun 日本語]]][Postposition で]][VP [AccNP [Nominal [Nominal [Noun [CommonNoun 本]]][Conjunct や][Nominal [Noun [CommonNoun 手紙]]]][AccPart を]][Verb [NatVerb 読む]]]]][FormalNoun こと]][GaAccPart が]][GaVerb できる]]]

[S [NomNP [Nominal [S [PP [Nominal [Noun [ProperNoun 日本語]]][Postposition で]][S [VP [AccNP [Nominal [Nominal [Noun [CommonNoun 本]]][Conjunct や][Nominal [Noun [CommonNoun 手紙]]]][AccPart を]][Verb [NatVerb 読む]]]]][FormalNoun こと]][NomPart が]][VP [GaVerb できる]]]

[S [VP [GaAccNP [Nominal [S [PP [Nominal [Noun [ProperNoun 日本語]]][Postposition で]][S [VP [AccNP [Nominal [Nominal [Noun [CommonNoun 本]]][Conjunct や][Nominal [Noun [CommonNoun 手紙]]]][AccPart を]][Verb [NatVerb 読む]]]]][FormalNoun こと]][GaAccPart が]][GaVerb できる]]]

私は、飛行機で東京に行くことを好む。
私--は--飛行機--で--東京--に--行く--こと--を--好む
Possible parses: 13
[S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [VP [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][VP [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][VP [AccNP [Nominal [S [VP [Verb [NatVerb 行く]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]]]]

[S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][S [VP [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][VP [AccNP [Nominal [S [VP [Verb [NatVerb 行く]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]]]]

[S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][S [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][S [VP [AccNP [Nominal [S [VP [Verb [NatVerb 行く]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]]]]

[S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [VP [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][VP [AccNP [Nominal [S [VP [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][VP [Verb [NatVerb 行く]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]]]

[S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][S [VP [AccNP [Nominal [S [VP [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][VP [Verb [NatVerb 行く]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]]]

[S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [VP [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][VP [AccNP [Nominal [S [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][S [VP [Verb [NatVerb 行く]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]]]

[S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][S [VP [AccNP [Nominal [S [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][S [VP [Verb [NatVerb 行く]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]]]

[S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [VP [AccNP [Nominal [S [VP [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][VP [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][VP [Verb [NatVerb 行く]]]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]]

[S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [VP [AccNP [Nominal [S [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][S [VP [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][VP [Verb [NatVerb 行く]]]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]]

[S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [VP [AccNP [Nominal [S [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][S [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][S [VP [Verb [NatVerb 行く]]]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]]

[S [VP [AccNP [Nominal [S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [VP [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][VP [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][VP [Verb [NatVerb 行く]]]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]

[S [VP [AccNP [Nominal [S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][S [VP [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][VP [Verb [NatVerb 行く]]]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]

[S [VP [AccNP [Nominal [S [TopNP [Nominal [Noun [Pronoun 私]]][TopPart は]][S [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][S [PP [Nominal [Noun [ProperNoun 東京]]][Postposition に]][S [VP [Verb [NatVerb 行く]]]]]][FormalNoun こと]][AccPart を]][Verb [NatVerb 好む]]]]

彼も彼女も東京から飛行機で行く。
彼--も--彼女--も--東京--から--飛行機--で--行く
Possible parses: 1
[S [AddNP [AddNominal [AddNominal [Nominal [Noun [Pronoun 彼]]][Additive も]][Nominal [Noun [Pronoun 彼女]]][Additive も]]][VP [PP [Nominal [Noun [ProperNoun 東京]]][Postposition から]][VP [PP [Nominal [Noun [CommonNoun 飛行機]]][Postposition で]][VP [Verb [NatVerb 行く]]]]]

****
NON-SENTENCES recognized correctly.
****

ライアン (Name in isolation)
英語を分かる。 (Ga-verb with normal accusative marker)
行くだ。(verb + copula)

****
BAD SENTENCES semantically parsed as S syntactically according to the grammar.
****

私が日本語を送る。
私--が--日本語--を--送る

```
Possible parses: 1
[S [NomNP [Nominal [Noun [Pronoun 私]]][NomPart が]][VP [AccNP [Nominal [Noun [ProperNoun 日本
語]]][AccPart を]][Verb [NatVerb 送る]]]]

ライアンが本を書くのを予約する。
ライアン--が--本--を--書く--の--を--予約--する
Possible parses: 2
[S [NomNP [Nominal [Noun [ProperNoun [NameNoun ライアン]]]][NomPart が]][VP [AccNP [Nominal [S [VP
[AccNP [Nominal [Noun [CommonNoun 本]]][AccPart を]][Verb [NatVerb 書く]]]][FormalNoun の]][AccPart
を]][Verb [VerbalNoun 予約][CAT3 する]]]]

[S [VP [AccNP [Nominal [S [NomNP [Nominal [Noun [ProperNoun [NameNoun ライアン]]]][NomPart が]][VP
[AccNP [Nominal [Noun [CommonNoun 本]]][AccPart を]][Verb [NatVerb 書く]]]][FormalNoun の]][AccPart
を]][Verb [VerbalNoun 予約][CAT3 する]]]]



--------------------------------
**************************
Example SENTENCES not parsed with the current test grammar.
**************************
--------------------------------
日本語で。(casual reply)
どこで? (casual question)
本を村上さんが書く。(OSV scrambling)
東京に行く飛行機だ。(noun modified by a relative clause)
東京に旅行するのに飛行機で行く。(formal noun + postposition [dative particale] + S)
```

B: Source Code

cky_driver.py

```
---------------------------------
#!/usr/bin/python2
# encoding: utf-8
import os,sys, string, re
from CFG import *
from CKY import *
from os.path import *

"""
This is the driver program. It takes in the input grammar text file, prompts the user
if they want to convert grammar into CNF, and utilizes all of the related classes stored
in CFG.py and CKY.py to parse input sentences entered in STDIN.
"""

#Get the command line argument -- the grammar file.
try:
     inputPath = sys.argv[1]

     inputFileName = basename(inputPath)

     if isfile(inputPath):
          print("Input file is set to: " + inputFileName)
     else:
          wd = dirname(abspath(__file__))
```

```
          inputPath = join(wd, inputFileName)
          if isfile(inputPath):
               print("Path changed to: " + inputPath)
          else:
               print("Input file could not be found.")
               sys.exit()

     convert_CNF = raw_input("Is this input formatted in: (1) CFG (2) CNF ? ") == '1'

except IndexError:
     sys.exit("Please check that you have an argument for the input file.")
except:
     print(sys.exc_info())
     sys.exit("There was an error running the program.")

txt_file = open(inputPath, "r")
txtStr = txt_file.read()

txt_file.close()

#Create the CNF grammar to work as the language model for CKY parsing.
grammar = CNF(txtStr, convert_CNF)
grammar.printGrammar()
grammar.printNonTerminals()
grammar.printTerminals()

#Create the parser with the new grammar.
parser = CKY(grammar)

#Continue reading sentences until EOF is received.
while True:
     try:
          sentence = raw_input("Please enter a sentence:\n").decode("utf-8")
          parser.parseSentence(sentence)

     except EOFError:
          print("\nEOF hit. Ending program.\n")
          break
```

CFG.py

```
# encoding: utf-8
import re,string,sys
#from collections import *

"""
Classes used to represent all facets of the context-free grammar needed for the CKY
algorithm.

Update for Assig 4:
CNF contains another class, CNF_search_tree, used to take in arguments from the
RHS of a proposed rule and find any corresponding LHS symbols (used for CKY)
"""
#Parent class
class CFG:

     #Dict with unique left hand symbols as keys and lists of rules assigned as the value.
     rule_list = {}
     #Unique set of symbols (actually view) that represent non-terminals.
     non_terminals = set()
     #Unique set of symbols that represent terminals.
     terminals = set()
```

```
    def __init__(self, grammar):
        lines = grammar.splitlines()

        #Create Rule objects and store in the rule list.
        for line in lines:
            temp = Rule(line)
            ind = temp.getLHS()

            #Create an empty list if this unique LHS doesn't exist yet
            if ind not in self.rule_list:
                self.rule_list[ind] = []
            self.rule_list[ind].append(temp)

        #Naturally, the unique LHS symbols are the non-terminals.
        #Notice that this is a view of dict keys. There is no need to manually
        #add entries as the rule_list's keys become updated.
        self.non_terminals = self.rule_list.keys()
        self.__storeTerminals()

    #Search the rule list for terminal symbols.
    def __storeTerminals(self):
        rules = self.rule_list
        for r_key in rules:
            for rule in rules[r_key]:
                rhs_symbols = rule.getRHS()
                i=0
                while i < len(rhs_symbols):
                    if rhs_symbols[i] not in self.non_terminals:
                        self.terminals.add(rhs_symbols[i].decode("utf-8"))
                    i+=1

    #Append a Rule object to the rule list dict.
    def appendRule(self, rule):
        ind=rule.getLHS()
        if ind not in self.rule_list:
                self.rule_list[ind] = []
        self.rule_list[ind].append(rule)

    #Convert grammar to a string.
    def toString(self):
        grammarStr = ''
        for i in sorted(self.rule_list):
            for j in self.rule_list[i]:
                grammarStr += j.toString() + "\n"
        return grammarStr

    #Print out the grammar in STDOUT.
    def printGrammar(self):
        print("Grammar:")
        for i in sorted(self.rule_list):
            for j in self.rule_list[i]:
                j.printRule()
        print("\n")

    #Print all non-terminals.
    def printNonTerminals(self):
        print("Non-terminals:")
        print(sorted(self.non_terminals))
        print("\n")

    #Print all terminals.
    def printTerminals(self):
        print("Terminals:")
        print('--'.join(sorted(self.terminals)).encode("utf-8"))
        print("\n")

#-----------------------------------------------------------------------
"""
Subclass CNF that inherits CFG.
```

```
It additionally contains instance data and methods for converting a grammar to CNF,
searching the CNF for a constructed rule, and historical information about any new
nodes created.
"""
class CNF(CFG):
     search_tree = None
     #Counter for labeling new variables used in rules.
     repl_counter = 1
     #Counter for new categories returned by fixing mixed rules.
     cat_counter = 1
     #Keeps history of new nodes created converting relevant rules into binary form.
     new_nodes = []

     #The constructor only calls to convert the grammar unless the user supplies an
     #argument saying that the grammar is already in CNF.
     def __init__(self, grammar, convertCNF=True):
          CFG.__init__(self,grammar)
          if convertCNF:
               self.convertToCNF()
          self.__createSearchTree()

     #Create a CNF search tree for the grammar in final CNF form.
     #See class CNF_search_tree for details.
     def __createSearchTree(self):
          self.search_tree = CNF_search_tree(self.terminals)
          rules = self.rule_list
          for r_key in rules:
               for rule in rules[r_key]:
                    self.search_tree.addRule(rule)

     #Call the CNF_search_tree object to find the rule based on the RHS parameter.
     def searchRule(self, RHS):
          if self.search_tree is not None:
               return self.search_tree.search(RHS)
          else:
               return None

     #Method that calls the steps of CNF conversion in order.
     def convertToCNF(self):
          self.__removeMixedRules()
          self.__removeUnitProductions()
          self.__makeRulesBinary()

     #This has not since been updated with the new addition of the terminal set. It
     #could be handled differently to make use of it, but it works as is.
     def __removeMixedRules(self):
          rules = self.rule_list
          nonTerm = self.non_terminals
          new_rules = []

          #Get the rule for the unique LHS.
          #Use a double loop since I'm using a dict (one for keys, the other for containing lists).
          for r_key in rules:
               for rule in rules[r_key]:
                    containsNT = False
                    termIndices = set()
                    arg_list = rule.getRHS()

                    #If it's not > 1, it can't be mixed.
                    if len(arg_list) > 1:
                         i = 0
                         #first, check to see if this rule is mixed
                         while i < len(arg_list):
                              #we have a terminal node
                              if arg_list[i] not in nonTerm:
                                   termIndices.add(arg_list[i])
                              else:
                                   containsNT=True
                              i+=1
```

```
                    #we have  a mixed rule
                    if len(termIndices) > 0 and containsNT:
                         for term in termIndices:
                              new_nt = "CAT" + str(self.cat_counter)
                              self.non_terminals.append(new_nt)
                              """iterate through the arg_list and use the dummy
                              NT as a replacement for terminals (as many times
                              as one may occur) to make the rule all NT's
                              """
                              arg_list = [new_nt if arg==term else arg for arg in arg_list]
                              self.cat_counter += 1
                              new_rules.append(Rule(new_nt + " > " + term))
                              #self.appendRule(new_rule)
                         #change the rule's RHS
                         rule.modifyRHS(arg_list)

          for new_rule in new_rules:
               self.appendRule(new_rule)
          #end removeMixedRules

     #Find all of the unit productions and create new rules to replace them.
     def __removeUnitProductions(self):
          rules = self.rule_list
          nonTerm = self.non_terminals
          new_rules = []

          for r_key in rules:
               for rule in rules[r_key]:
                    temp_new_rules = []
                    arg_list=rule.getRHS()

                    #Only one non-terminal on the RHS.
                    #Use a recursive algorithm to get the appropriate terminal replacement.
                    if len(arg_list) == 1 and arg_list[0] in nonTerm:
                         temp_new_rules = self.__getNonUP(arg_list[0], rule.getLHS(), [])

                    #Mark unit productions for removal.
                    if len(temp_new_rules) > 0 :
                         rule.markRemoval(True)
                    new_rules = new_rules + temp_new_rules

          #Delete everything marked for removal and readjust rule count
          #for each unique LHS as necessary.
          for r_key in rules:
               rule_ind = 0
               num_rules = len(rules[r_key])
               while(rule_ind < num_rules and num_rules > 0):
                    rule = rules[r_key][rule_ind]
                    if rule.isMarkedRemoval():
                         del rules[r_key][rule_ind]
                         rule_ind -= 1
                         num_rules -= 1
                    rule_ind +=1

          #Add all of the new rules.
          for n_rule in new_rules:
               self.appendRule(n_rule)
          #end removeUnitProductions

     #Make all rules binary, particularly ones with a RHS symbol count > 2.
     def __makeRulesBinary(self):
          rules = self.rule_list
          new_rules = []
          nonTerm = self.non_terminals
          for r_key in rules:
               for rule in rules[r_key]:
                    rhs=rule.getRHS()
```

```
                #We have no mixed rules at this point, so just test the first node on the RHS
                #whether it is a non-terminal or not.
                if len(rhs) > 2 and rhs[0] in nonTerm:
                     num_args = len(rhs)
                     new_rhs = rhs
                     while num_args > 2:
                          #Reuse dummy variables X# here if the pairings already exist.
                          new_nt = None
                          for nrule in new_rules:
                               nrule_args = nrule.getRHS()

                               #new_rules are only allocated in this method, so they should all be
                               #binary rules. No need for checking.
                               #If we find a match of pairs with a new rule and a proposed RHS,
                               #reuse the new rule.
                               if nrule_args[0] == new_rhs[0] and nrule_args[1] == new_rhs[1]:
                                    new_nt = nrule.getLHS()
                                    break

                          #No new rules exist already for the proposed RHS pair of symbols so
                          #create one.
                          if new_nt is None:
                               new_nt = "X" + str(self.repl_counter)
                               self.repl_counter += 1
                               new_rules.append(Rule(new_nt + " > " + new_rhs[0] + " " + new_rhs[1]))
                               self.new_nodes.append(new_nt)

                          #Shorten the RHS with the introduction of the new rule.
                          new_rhs = [new_nt] + new_rhs[2:len(new_rhs)]
                          num_args -= 1
                     rule.modifyRHS(new_rhs)
          for n_rule in new_rules:
               self.appendRule(n_rule)
          #end makeRulesBinary

     #Recursive method to find the first non-unit production if there is a chain.
     def __getNonUP(self, lhs, old_lhs, up_hist):
          new_rules = []
          for rule in self.rule_list[lhs]:
               rhs = rule.getRHS()
               temp_uph = list(up_hist)
               temp_uph.append(lhs)

               #If this rule is a unit production, go to the next rule and check again.
               if len(rhs) == 1 and rhs[0] in self.non_terminals:
                    new_rules = new_rules + self.__getNonUP(rhs[0], old_lhs, temp_uph)
               else:
                    new_rules.append(Rule(old_lhs + ' > ' + rule.getRHS_string(), temp_uph))

          return new_rules

     #This was a hack to get the rule with the new unit production history.
     def getRuleUPHist(self, lhs, rhs1, rhs2=None):
          rules = self.rule_list

          #Iterate through rules dict to find one the first that matches our terminal symbol.
          #Not ideal way of search, but this is a hack afterall.
          for rule in rules[lhs]:
               rhs_list = rule.getRHS()
               rhs_list = [x.decode('utf-8') for x in rhs_list]

               #Get rule UP history searched by a terminal word (no rhs2) or a binary
               #rule combo.
               if rhs2 is None and rule.hasUPHist() and rhs1 in rhs_list:
                    return rule.getUPHist()
               if len(rhs_list) == 2 and rhs1 == rhs_list[0] and rhs2 == rhs_list[1]:
                    return rule.getUPHist()
          return []
```

```
     #Tests whether this node is in the original grammar and not introduced by
     #CNF conversion.
     def isOrigNode(self, lhs):
          return lhs not in self.new_nodes

#-----------------------------------------------------------------------
"""
Class that represents individual rules. It stores the LHS as a string and the RHS as
a list of strings.
"""
class Rule:
     lhs = ''
     rhs = []
     #Stores unit production history so that the bracket notation output can be
     #accurately formed based on the original grammar.
     up_hist = []
     #Determines whether the rules output should separate RHS nodes with a pipe character.
     pipe = False
     #Mark for removal - used during unit production conversion.
     _removal = False

     def __init__(self, input_str, unit_prod_hist = None):
          (self.lhs, rhs_str) = input_str.split(' > ')
          self.rhs = re.split(r'[ |]+', rhs_str, re.UNICODE)

          if re.search(r'[|]', rhs_str, re.UNICODE):
               self.pipe=True
          if type(unit_prod_hist) is list:
               self.up_hist = unit_prod_hist

     def modifyRHS(self, in_rhs):
          self.rhs = in_rhs
     def markRemoval(self, val):
          self._removal = val
     def isMarkedRemoval(self):
          return self._removal
     def getLHS(self):
          return self.lhs
     def getRHS(self):
          return self.rhs
     def getRHS_string(self):
          if self.pipe:
               return ' | '.join(self.rhs)
          else:
               return ' '.join(self.rhs)

     #Return the chain of nodes that were removed during unit production conversion.
     def getUPHist(self):
          return self.up_hist
     def hasUPHist(self):
          return len(self.up_hist) > 0
     def toString(self):
          return self.lhs + " > " + self.getRHS_string()
     def printRule(self):
          print(self.lhs + " > " + self.getRHS_string())

#-----------------------------------------------------------------------
"""
Class that allows search of a LHS based on RHS symbols. The corresponding rules
must be in CNF, thus this class is only instantiated under such objects.
This is actually a tree of dicts where child data contains the LHS data for the
current traversal, and the tree contains a subtree yet to be traversed.

Terminals are reused here for addition purposes.
"""
class CNF_search_tree:

     def __init__(self, terms=[]):
          self.tree= {}
```

```
        self.child_data= set([])
        self.level = 0
        self.terminals = [x.lower() for x in terms]

    #Iterative version of addition to the tree.
    #All resulting LHS data is stored in a subtree without any trees of it's own
    #i.e. The child subtree at position x where x = length of rule + 1.
    def addRule(self, rule):
        terminals = self.terminals
        RHS = rule.getRHS()
        LHS = rule.getLHS()
        CST = self
        level_count = 1

        #Only need terminals for top level of the tree. Anything that extends below
        #this level must be a non-terminal.
        for rhs_sym in RHS:
            rhs_sym = rhs_sym.decode("utf-8")

            #Create a new subtree if an entry does not exist for this symbol     .
            if rhs_sym not in CST.tree:
                CST.tree[rhs_sym] = CNF_search_tree()

            CST = CST.tree[rhs_sym]
            CST.level = level_count
            level_count += 1

            #If we have a terminal symbol, add to child subtree and restart
            #at the top level.
            if rhs_sym in terminals:
                CST.child_data.add(LHS)
                CST = self

        CST.child_data.add(LHS)

    #Iterative version of a search.
    def search(self, RHS):
        terminals = self.terminals
        CST = self

        #Check if a single string was mistakenly input. If so, insert in a list.
        if type(RHS) == str or type(RHS) == unicode:
            temp = RHS
            RHS = [temp]

        if type(RHS) != list:
            print("List parameter needed for CKY_search_tree.search()")
            return None

        rhs_len = len(RHS)
        rhs_ind = 1

        for rhs_sym in RHS:
            isCST = rhs_sym in CST.tree and isinstance(CST.tree[rhs_sym], CNF_search_tree)

            #Check if we are still in bounds of the tree.
            if(isCST and rhs_ind <= rhs_len):
                CST = CST.tree[rhs_sym]
                rhs_ind +=1

                #We hit the child subtree containing data for our rule.
                if(rhs_ind > rhs_len and len(CST.child_data) > 0):
                    return list(CST.child_data)

            else:
                return None
```

CKY.py

```
# encoding: utf-8
import re,string
from collections import *
from CFG import *
import MeCab

"""
Class used to implement CKY algorithm. It only takes in a CNF grammar during construction.
The grammar joins an parse history tracker as the only instance data.
"""

class CKY:

     def __init__(self, in_grammar):
          self.grammar = in_grammar
          self.indexer = CKY_indexer()

     #String conversion of parse table.
     def tableToString(self, parseTable):
          row_ind = 0
          col_ind = 0
          cky_string = ''
          for i in parseTable:
               for j in i:
                    sym_list = []
                    if j is not None:
                         for ind in j:
                              sym_list.append(self.indexer.getSym(ind))
                    cky_string = cky_string + '['+str(row_ind)+','+str(col_ind)+']\t'
                    cky_string += str(sym_list) + "\n"
                    row_ind += 1
               col_ind += 1
               row_ind = 0
          return cky_string

     #Method that calls a recursive algorithm to generate the bracket notation. It begins
     #the brackets for the top level S node.
     def parseToNotationString(self, s_indices, words):
          notation = ''
          for s in s_indices:
               notation = notation + "[" + self.indexer.getSym(s) +" " + self.__generateNotation(s, words)
               notation = notation + "]\n\n"
          return notation

     #Recursive method that generates the rest of the bracket notation within S.
     def __generateNotation(self, parent, words):
          notation = ''
          rhs1 = self.indexer.getRHS1(parent)
          rhs2 = self.indexer.getRHS2(parent)

          #If both rhs1 and rhs2 are not null, this must be a full binary rule.
          #Get the symbols from the indexer, and determine if it was from the original
          #grammar. Get the unit prod. history for this rule as well.
          if rhs1 is not None and rhs2 is not None:
               rhs1_sym = self.indexer.getSym(rhs1)
               rhs2_sym = self.indexer.getSym(rhs2)
               rhs1_orig = self.grammar.isOrigNode(rhs1_sym)
               rhs2_orig = self.grammar.isOrigNode(rhs2_sym)
               unit_prod_history = self.grammar.getRuleUPHist(self.indexer.getSym(parent), rhs1_sym,
rhs2_sym)
               close1 = ""
               close2= ""

               #Keep generating new categories if we have the unit prod. history to do so.
```

```
            for uph_elem in unit_prod_history:
                notation = notation + "[" + uph_elem + " "

            if rhs1_orig:
                notation = notation + "[" + rhs1_sym + " "
                close1 = "]"
            notation = notation + self.__generateNotation(rhs1, words) + close1
            if rhs2_orig:
                notation = notation + "[" + rhs2_sym + " "
                close2 = "]"
            notation = notation + self.__generateNotation(rhs2, words) + close2

            end_bracket = "]" * len(unit_prod_history)
            notation = notation + end_bracket

        #This rule is not binary, and it must be a terminal word.
        else:
            word = words[self.indexer.getCol(parent)]
            unit_prod_history = self.grammar.getRuleUPHist(self.indexer.getSym(parent), word)
            for uph_elem in unit_prod_history:
                notation = notation + "[" + uph_elem + " "

            notation = notation + word
            end_bracket = "]" * len(unit_prod_history)
            notation = notation + end_bracket

        return notation

    #Method that determines if a string of words constitutes a sentence
    #based on the instance grammar.
    def parseSentence(self, sentence):
        grammar = self.grammar

        #Utilize the MeCab morphological analyzer to separate words by spaces.
        tagger = MeCab.Tagger("-Owakati")
        words = tagger.parse(sentence.encode("utf-8"))
        #Remove (replace) punctuation characters with empty string.
        word_pattern = re.compile(u'\s+[\u3002\u0021\u3001\uFF1F\uFF01]|\s+$')
        words = word_pattern.sub(u'', words.decode('utf-8'))
        word_list = words.split(' ')
        #Display segmented words to STDOUT.
        print '--'.join(word_list).encode('utf-8')
        words_len = len(word_list)
        indexer = self.indexer

        #We cannot have a sentence if we do not have at least one word.
        if words_len < 1:
            print("Not S\n")
            return

        """
        The implementation of the CKY parse from the book uses indices
        that do not match up with indexing of lists, so this was modified
        to reflect this.

        The list comprehension belows gives a staggered matrix of lists.

        We start by filling in the categories of the words first.
        """
        parse_table = [[None]*(words_len+1-x) for x in range(words_len, 0, -1)]
        for j in range(words_len):

            #searchRule returns a list of symbols that give the current word
            first_sym = grammar.searchRule(word_list[j])
            if first_sym is not None:
                for lhs in first_sym:
                    index = indexer.insert(lhs, j, j)
                    if parse_table[j][j] is None:
                        parse_table[j][j] = []
```

```
                    parse_table[j][j].append(index)

               #Note the lower boundary of range.
               for i in range(j-1, -1, -1):
                    for k in range(i, j):
                         if parse_table[k][i] is not None and parse_table[j][k+1] is not None:

                              #Search for all combinations of symbols in the relevant
                              #indices of the parse table.
                              for B_ind in parse_table[k][i]:
                                   for C_ind in parse_table[j][k+1]:
                                        B_sym = indexer.getSym(B_ind)
                                        C_sym = indexer.getSym(C_ind)
                                        temp_rhs = [B_sym, C_sym]

                                        #Returns a list of LHS sybmols meeting the rule.
                                        A_symbols = grammar.searchRule(temp_rhs)
                                        if A_symbols is not None:
                                             if parse_table[j][i] is None:
                                                  parse_table[j][i] = []

                                             #Add new symbol to the parse table appropriately.
                                             for lhs in A_symbols:
                                                  index = indexer.insert(lhs, j, i, B_ind, C_ind)
                                                  parse_table[j][i].append(index)

          #Print CKY parse table contents.
          print(self.tableToString(parse_table))

          #Check the top right corner of our parse table for a sentence.
          if parse_table[words_len-1][0] is None:
               print("Not S")
          else:
               s_ind = []
               for pt_ind in parse_table[words_len-1][0]:
                    #pt_symbols.append(indexer.getSym(pt_ind))
                    s_nodes = ['S', 'S_ques', 'S_inform']
                    if indexer.getSym(pt_ind) in s_nodes:
                         s_ind.append(pt_ind)

               if len(s_ind) > 0:
                    print("S found!\nPossible parses: " + str(len(s_ind)))
                    #indexer.printIndexer()
                    print(self.parseToNotationString(s_ind, word_list))
               else:
                    print("Not S")

"""
The indexer is a history tracker of all symbols used in each cell of the CKY parse table.
It assigns a unique index to the resulting node, stores the indices of the nodes from
other cells that created it, and returns relevant data to the method for creating
bracket notation.
"""

class CKY_indexer:
     def __init__(self):
          self.ind_table = {}
          self.counter = 0

     def insert(self, d, r, c, r1_ind=None, r2_ind=None):
          self.ind_table[self.counter] = {"data":d, "row":r, "col":c, "r1_ind":r1_ind, "r2_ind":r2_ind}
          current_ind = self.counter
          self.counter = self.counter + 1
          return current_ind

     def getSym(self, i): return self.ind_table[i]['data']
     def getRHS1(self, i): return self.ind_table[i]['r1_ind']
     def getRHS2(self, i): return self.ind_table[i]['r2_ind']
     def getCol(self, i): return self.ind_table[i]['col']
```