project3_tree

October 7, 2020

[ ]:

```
# Please do not change this cell because some hidden tests might depend on it.
import os

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2020/project3.git .tmp
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[ ]:

```
# Initialize Otter
import otter
grader = otter.Notebook()
```

1 Project 3: Parsing Using the Cocke-Kasami-Younger (CKY) Algorithm

Constituency Parsing is an intermediary task in Natural Language Processing, where the goal is to extract the syntactic parse tree given a sentence.

In this project, you will implement the CKY algorithm for context-free grammars (CFG). You will implement parsing algorithms for both a determinstic grammar (CFG) and a probabilistic grammar (PCFG).

1.1 Goals

1. Finish a CFG for the ATIS dataset.
2. Implement an algorithm for determining whether a parse exists or not given a sentence.
3. Implement the CKY algorithm for parsing using CFGs.
4. Construct a probabilistic context-free grammar (PCFG) based on a CFG.
5. Implement the CKY algorithm for parsing using PCFGs.

1.2 Setup

[ ]:

```
!pip install nltk
```

[ ]:

```
import nltk
nltk.download('large_grammars')

from nltk.grammar import ProbabilisticProduction, PCFG, Nonterminal
from nltk.tree import Tree
from nltk import treetransforms

from collections import defaultdict, Counter
```

[ ]:

```
# Grammar Coverage
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/train.nl

# PCFG Construction
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/train.trees
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/dev.trees
!wget -nv -N -P data https://raw.githubusercontent.com/nlp-course/data/master/
 ↪ATIS/test.trees

# Tree Utils
```

```
!wget -nv -N -P scripts https://raw.githubusercontent.com/nlp-course/data/
↪master/scripts/trees/tree_utils.py
!wget -nv -N -P scripts https://raw.githubusercontent.com/nlp-course/data/
↪master/scripts/trees/evalb.py
!wget -nv -N -P scripts https://raw.githubusercontent.com/nlp-course/data/
↪master/scripts/trees/tree.py
```

1.2.1 Helper Functions

The standard CKY algorithm requires the grammar to be in Chomsky normal form (CNF), i.e., only rules of the form *A -> B C*, *A -> a* are allowed where *A*, *B*, *C* are nonterminals and *a* is a terminal symbol. However, in some downstream applications (such as our next project) we want to use grammar rules of more general forms, such as *A -> B C D*. To satisfy both of these constraints, we will convert the grammar to CNF, parse using CKY, and then convert back to the form of the original grammar. We have provided helper functions to perform those conversions.

To convert a grammar to CNF:

`cnf_grammar, cnf_grammar_wunaries = get_cnf_grammar(grammar)`

To convert a tree output from CKY back to the original form of the grammar in place:

`un_cnf(tree, cnf_grammar_wunaries)`

Note that above we need to pass in `cnf_grammar_wunaries`, an intermediate version of the grammar before removing unary rules. It's the second returned value of `get_cnf_grammar`.

[ ]:
```
from scripts.tree_utils import get_cnf_grammar, un_cnf
```

1.3 A custom ATIS grammar

To parse, we need a grammar. In this project, we will use a hand-crafted grammar for the ATIS dataset. It captures a large fraction of the language in the ATIS dataset while including a minimal number of rules. This tiny grammar will be used again in the next project for a question answering application.

1.3.1 Goal 1: Finish the CFG for the ATIS dataset

Fill in the missing rule for `PREJ`. This is a production rule that captures all the ”junk” before the actual semantically meaningful question. For example, in ”i would like a flight between boston and dallas”, ”i would like” carries no meaningful information hence is considered ”junk”.

Hint: you will make use of the `JUNK` preduction

[ ]:
```
#TODO
miniATIS = nltk.CFG.fromstring("""
S -> PREJ DET ADJS FLIGHT PPS | DET ADJS FLIGHT PPS | PREJ ADJS FLIGHT PPS | ␣
 ↪ADJS FLIGHT PPS | PREJ DET FLIGHT PPS | DET FLIGHT PPS | PREJ FLIGHT PPS |␣
 ↪FLIGHT PPS | PREJ DET ADJS FLIGHT  | DET ADJS FLIGHT | PREJ ADJS FLIGHT |␣
 ↪ADJS FLIGHT | PREJ DET FLIGHT | DET FLIGHT | PREJ FLIGHT | FLIGHT
PPS -> PP | PP PPS
ADJS -> ADJ | ADJ ADJS

FLIGHT -> 'flights' | 'flight' | 'to' 'fly'

PREJ ->␣
 ↪TODO<<add<<here<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
JUNK -> 'me' | 'show' | 'now' | 'only' | 'can' | 'you' | 'the' | 'itinerary' |␣
 ↪'of' | 'also' | 'a' | 'list' | 'could' | 'give' | 'which' | 'what' | 'is' |␣
 ↪"what's" | 'are' | 'my' | 'choices' | 'for' | 'i' | 'would' | 'like' | "i'd"␣
 ↪| 'to' | 'see' | 'have' | 'make' | 'book' | 'find' | 'information' | 'on' |␣
 ↪'know' | 'some' | 'hello' | 'yes' | 'please' | 'repeat' | 'do' | 'have' |␣
 ↪'there' | 'need' | 'hi' | 'get' | 'may' | 'listing' | 'listings' | 'travel' |␣
 ↪'arrangements' | 'okay' | 'want' | 'tell' | 'about' | 'how' | 'would' | 'be'␣
 ↪| 'able' | 'put' | 'requesting' | "i'm" | 'looking' | 'display' | UNK

ADJ -> AIRLINE | SIMPLEWEEKDAY | FLIGHTTYPE | SIMPLETIME | "monday's" |␣
 ↪"tuesday's" | "wednesday's" | "thursday's" | "friday's" | "saturday's" |␣
 ↪"sunday's" | 'available' | 'possible' | 'first' 'class' | 'economy' |␣
 ↪'thrift' 'economy' | 'cheapest' | 'lowest' 'cost' | 'least' 'expensive' |␣
 ↪'most' 'expensive' | 'weekday' | 'daily' | 'last' | 'first' | 'dinner' |␣
 ↪'transcontinental'

DET -> 'all' 'the' | 'all' | A | 'an' | THE | 'any' | 'all' 'of' 'the'
BETWEEN -> 'between'
AND -> 'and'
OR -> 'or'
EITHER -> 'either'
OF -> 'of'
THE -> 'the'
A -> 'a'

PP -> PPLACE PLACE OR PLACE | PPLACE EITHER PLACE OR PLACE | PPLACE PLACE |␣
 ↪EITHER PLACE OR PLACE | PLACE OR PLACE | PLACE | BETWEEN PLACE AND PLACE |␣
 ↪BETWEEN TIME AND TIME | PTIME TIME | TIME | PDAY WEEKDAY | WEEKDAY TIME |␣
 ↪WEEKDAY | PDAY WEEKDAY TIME | PDAY DATE | DATE | PAIRLINE AIRLINE | AIRCRAFT␣
 ↪| FLIGHTTYPE | FARETYPE | PRICE | FOOD | AVAIL | POSTJ
```

```
PPLACE -> 'to' | 'that' 'arrive' 'at' | 'that' 'arrives' 'in' | 'coming' 'back'␣
 ↪'to' | 'that' 'go' 'to' | 'and' 'then' 'to' | 'arriving' 'in' | 'and'␣
 ↪'arriving' 'in' | 'and' 'arrive' 'in' | 'to' 'arrive' 'in' | 'arrive' 'in' |␣
 ↪'going' 'to' | 'into' | 'for' | 'with' 'the' 'destination' 'city' 'of' |␣
 ↪'arriving' | 'goes' 'to' | 'flying' 'into' | 'goes' 'on' 'to' | 'reaching' |␣
 ↪'in' | 'and' 'then' | 'arriving' 'to' | 'from' | 'leaving' | 'return' 'from'␣
 ↪| 'leaving' 'from' | 'departing' 'from' | 'departing' | 'go' 'from' | 'going'␣
 ↪'from' | 'back' 'from' | 'that' 'goes' 'from' | 'that' 'departs' | 'which'␣
 ↪'leaves' 'from' | 'which' 'leave' | 'that' 'leave' | 'originating' 'in' |␣
 ↪'leave' | 'out' 'of' | 'leaves' 'from' | 'to' 'get' 'from' | 'via' | 'with'␣
 ↪'a' 'stopover' 'in' | 'with' 'a' 'layover' 'in' | 'with' 'a' 'stopover' 'at'␣
 ↪| 'and' 'a' 'stopover' 'in' | 'stop' 'in' | 'stopping' 'in' | 'make' 'a'␣
 ↪'stop' 'in' | 'with' 'a' 'stop' 'in' | 'with' 'one' 'stop' 'in' | 'go'␣
 ↪'through' | 'which' 'go' 'through' | 'makes' 'a' 'stopover' 'in' | 'that'␣
 ↪'stops' 'in' | 'that' 'stops' 'over' 'in' | 'by' 'way' 'of' | 'connecting'␣
 ↪'through' | 'that' 'will' 'stop' 'in' | 'which' 'connects' 'in' | 'arriving'␣
 ↪'and' 'departing' 'at'

PLACE -> 'anywhere' | 'atlanta' | 'austin' | 'baltimore' | 'boston' | 'boston'␣
 ↪'logan' | 'burbank' | 'bwi' | 'charlotte' | 'chicago' | 'cincinnati' |␣
 ↪'cleveland' | 'cleveland' 'ohio' | 'columbus' | 'dallas' 'fort' 'worth' |␣
 ↪'dallas' | 'denver' | 'denver' 'colorado' | 'detroit' | 'fort' 'worth' |␣
 ↪'general' 'mitchell' 'international' | 'general' 'mitchell' | 'houston' |␣
 ↪'indianapolis' | 'jfk' | 'kansas' 'city' | 'laguardia' 'airport' | 'las'␣
 ↪'vegas' | 'long' 'beach' | 'los' 'angeles' | 'love' 'field' | 'memphis' |␣
 ↪'miami' | 'milwaukee' | 'minneapolis' | 'montreal' | 'montreal' 'quebec' |␣
 ↪'montreal' 'canada' | 'nashville' | 'new' 'york' 'city' | 'new' 'york' |␣
 ↪'newark' | 'newark' 'new' 'jersey' | 'oakland' | 'oakland' 'california' |␣
 ↪'ontario' | 'orlando' | 'orlando' 'florida' | 'philadelphia' | 'philly' |␣
 ↪'phoenix' | 'pittsburgh' | 'salt' 'lake' 'city' | 'san' 'diego' | 'san'␣
 ↪'diego' 'california' | 'san' 'francisco' | 'san' 'jose' | 'seattle' | 'st.'␣
 ↪'louis' | 'st.' 'paul' | 'st.' 'petersburg' | 'tacoma' | 'tacoma'␣
 ↪'washington' | 'tampa' | 'toronto' | 'various' 'cities' | 'washington' |␣
 ↪'washington' 'dc' | 'dc' | 'westchester' 'county' | UNK

PTIME -> 'that' 'arrive' 'before' | 'that' 'arrives' 'before' | 'arriving'␣
 ↪'before' | 'arrival' 'by' | 'arrives' | 'before' | 'departing' 'before' |␣
 ↪'that' 'leaves' 'before' | 'which' 'arrive' 'before' | 'by' | 'around' |␣
 ↪'that' 'return' 'around' | 'that' 'gets' 'in' 'around' | 'at' | 'arriving'␣
 ↪'around' | 'arriving' 'about' | 'that' 'arrive' 'soon' 'after' | 'leaving'␣
 ↪'at' | 'leaving' | 'which' 'leave' 'after' | 'leaving' 'after' | 'after' |␣
 ↪'departing' 'after' | 'that' 'depart' 'after' | 'departing' 'at' | 'arriving'␣
 ↪'after' | 'in' | 'departing' 'in' | 'on' | 'that' 'leaves' 'in'
```

```
TIME -> SIMPLETIME | 'the' 'afternoon' | 'the' 'late' 'afternoon' | 'the'␣
 ↪'evening' | 'the' 'morning' | 'the' 'early' 'am' | 'the' 'day' | 'mornings' |␣
 ↪'afternoons' | 'evenings' | 'as' 'early' 'as' 'possible' | 'earliest'␣
 ↪'possible' 'time' | 'as' 'soon' 'thereafter' 'as' 'possible' | UNK

SIMPLETIME -> 'one' | 'two' | 'three' | 'four' | 'five' | 'six' | 'seven' |␣
 ↪'eight' | 'nine' | 'ten' | 'eleven' | 'twelve' | '1' | '2' | '3' |'4' | '5' |␣
 ↪'6' | '7' |  '8' | '9' | '10' | '11' | '12' | '1pm' | '2pm' | '3pm' | '4pm' |␣
 ↪'5pm' | '6pm' | '7pm' | '8pm' | '9pm' | '10pm' | '11pm' | '12pm' | '1am' |␣
 ↪'2am' | '3am' | '4am' | '5am' | '6am' | '7am' | '8am' | '9am' | '10am' |␣
 ↪'11am' | '12am' | '230' | '1505' | '630am' | '720am' | '723pm' | '819' | '845'␣
 ↪| '1026' | '1145am' | '1' "o'clock" | '1' "o'clock" 'am' | "1" "o'clock" "pm"␣
 ↪| "2" "o'clock" | "2" "o'clock" "am" | "2" "o'clock" "pm" | "3" "o'clock" |␣
 ↪"3" "o'clock" "am" | "3" "o'clock" "pm" | "4" "o'clock" | "4" "o'clock" "pm" |␣
 ↪"4" "o'clock" "am" | "5" "o'clock" | "5" "o'clock" "am" | "5" "o'clock" "pm"␣
 ↪| "6" "o'clock" | "6" "o'clock" "pm" | "6" "o'clock" "am" | "7" "o'clock" |␣
 ↪"7" "o'clock" "pm" | "7" "o'clock" "am" | "8" "o'clock" | "8" "o'clock" "am"␣
 ↪| "8" "o'clock" "pm" | "9" "o'clock" | "9" "o'clock" "pm" | "9" "o'clock"␣
 ↪"am" | "10" "o'clock" | "10" "o'clock" "am" | "10" "o'clock" "pm" | "11"␣
 ↪"o'clock" | "11" "o'clock" "am" | "11" "o'clock" "pm" | "12" "o'clock" | "12"␣
 ↪"o'clock" "pm" | "12" "o'clock" "am" |  'noon' | '12' 'noon' | "12" "o'clock"␣
 ↪"noon" | 'midnight' | '12' 'midnight' | 'lunch' 'time' | 'dinnertime' | ␣
 ↪'evening' | 'night' | 'morning' | 'afternoon' | 'early' | 'late' | 'tonight'␣
 ↪| 'earliest' 'possible' | 'earliest' | 'latest' 'possible' | 'latest'


PDAY -> 'on' | 'returning' 'on' | 'of' | 'for' | 'next' | 'the' 'next' | 'in'␣
 ↪'the' 'next' | 'of' 'next' | 'leaving' | 'arriving' | 'arriving' 'on' |␣
 ↪'which' 'leave' | 'that' 'arrive' 'on' | 'leaving' 'on' | 'which' 'arrive'␣
 ↪'on' | 'a' 'week' 'from'

SIMPLEWEEKDAY -> 'saturday' | 'sunday' | 'monday' | 'tuesday' | 'wednesday' |␣
 ↪'thursday' | 'friday'
WEEKDAY -> SIMPLEWEEKDAY | A SIMPLEWEEKDAY | 'saturdays' | 'sundays' | 'mondays'␣
 ↪| 'tuesdays' | 'wednesdays' | 'thursdays' | 'fridays' | 'this' 'saturday' |␣
 ↪'this' 'sunday' | 'this' 'monday' | 'this' 'tuesday' | 'this' 'wednesday' |␣
 ↪'this' 'thursday' | 'this' 'friday' | 'this' 'coming' 'saturday' | 'this'␣
 ↪'coming' 'sunday' | 'this' 'coming' 'monday' | 'this' 'coming' 'tuesday' |␣
 ↪'this' 'coming' 'wednesday' | 'this' 'coming' 'thursday' | 'this' 'coming'␣
 ↪'friday' | 'day' | 'week' | 'today' | 'tomorrow' | 'the' 'day' 'after'␣
 ↪'tomorrow' | 'a' 'weekday' | 'weekdays'

DATE -> MONTH DAY YEAR | MONTH DAY | THE DAY | MONTH DAY OR DAY | THE DAY OR DAY␣
 ↪| EITHER MONTH DAY OR DAY | EITHER THE DAY OR THE DAY | THE DAY OF MONTH | DAY␣
 ↪OF MONTH | EITHER THE DAY OR THE DAY OF MONTH | THE DAY OF MONTH OR THE DAY OF␣
 ↪MONTH | EITHER THE DAY OF MONTH OR THE DAY OF MONTH
```

```
MONTH -> 'january' | 'february' | 'march' | 'april' | 'may' | 'june' | 'july' |␣
 ↪'august' | 'september' | 'october' | 'november' | 'december'
DAY -> 'first' | 'second' | 'third' | 'fourth' | 'fifth' | 'sixth' | 'seventh' |␣
 ↪'eighth' | 'ninth' | 'tenth' | 'eleventh' | 'twelfth' | 'thirteenth' |␣
 ↪'fourteenth' | 'fifteenth' | 'sixteenth' | 'seventeenth' | 'eighteenth' |␣
 ↪'nineteenth' | 'twentieth' | 'twenty-first' | 'twenty' 'first' |␣
 ↪'twenty-second' | 'twenty' 'second' | 'twenty-third' | 'twenty' 'third' |␣
 ↪'twenty-fourth' | 'twenty' 'fourth' | 'twenty-fifth' | 'twenty' 'fifth' |␣
 ↪'twenty-sixth' | 'twenty' 'sixth' | 'twenty-seventh' | 'twenty' 'seventh' |␣
 ↪'twenty-eighth' | 'twenty' 'eighth' | 'twenty-ninth' | 'twenty' 'ninth' |␣
 ↪'thirtieth' | 'thirty-first' | 'thirty' 'first' | 'one' | 'two' | 'three' |␣
 ↪'four' | 'five' | 'six' | 'seven' | 'eight' | 'nine' | 'ten' | 'eleven' |␣
 ↪'twelve' | 'thirteen' | 'fourteen' | 'fifteen' | 'sixteen' | 'seventeen' |␣
 ↪'eighteen' | 'nineteen' | 'twenty' | 'twenty' 'one' | 'twenty' 'two' |␣
 ↪'twenty' 'three' | 'twenty' 'four' | 'twenty' 'five' | 'twenty' 'six' |␣
 ↪'twenty' 'seven' | 'twenty' 'eight' | 'twenty' 'nine' | 'thirty' | 'thirty'␣
 ↪'one'
YEAR -> '1991' | '1992'

PAIRLINE -> 'on' | 'using' | 'of' | 'with'

AIRLINE -> 'continental' | 'continental' 'airline' | 'continental' 'airlines' |␣
 ↪'american' 'airlines' | 'american' 'airlines' | 'american' | 'united'␣
 ↪'airlines' | 'united' | 'united' 'airline' | 'northwest' 'and' 'united' |␣
 ↪'northwest' 'airlines' | 'northwest' 'airline' | 'northwest' | 'us' 'air' |␣
 ↪'us' 'airlines' | 'delta' | 'twa' | 'air' 'canada' | 'eastern' 'airlines' |␣
 ↪'midwest' 'express' | 'trans' 'world' 'airline'

AIRCRAFT -> 'using' 'a' 'j31' 'aircraft'

FLIGHTTYPE -> 'round' 'trip' | 'round-trip' | 'nonstop' | 'one' 'way' | 'return'␣
 ↪| 'direct' | 'connecting' | 'direct' 'and' 'connecting' | 'nonstop' 'or'␣
 ↪'connecting'

FARETYPE -> 'with' 'economy' 'fares' | 'qualify' 'for' 'fare' 'code' 'qx' |␣
 ↪'with' 'first' 'class' 'service' | 'that' 'offer' 'first' 'class' | 'on'␣
 ↪'first' 'class' | 'economy' 'class' | 'with' 'the' 'lowest' 'one' 'way'␣
 ↪'fares' | 'with' 'a' 'class' 'of' 'service' 'code' 'f' | 'with' 'q' 'fares' |␣
 ↪'first' 'class' 'fare'

PRICE -> 'less' 'than' '1100' 'dollars' | 'with' 'the' 'highest' 'fare' | 'the'␣
 ↪'cheapest' 'way' 'possible' | 'less' 'than' '466' 'dollars' | 'lowest' 'cost'␣
 ↪| 'least' 'expensive' | 'most' 'expensive' | 'cheapest'

FOOD -> 'dinner' | 'a' 'meal' | 'lunch' | 'breakfast'
```

```
AVAIL -> 'available' | 'i' 'can' 'get'

POSTJ -> 'please' | 'there' | 'are' | 'currently' | 'do' | 'you' | 'have' |␣
 ↪'fares' | 'information' | 'i' | 'want' | 'would' | 'like' | 'the' | 'flight'␣
 ↪| 'be' | 'go' | 'departures' | 'is' | 'such' | 'a' | 'that' | 'serves' |␣
 ↪'both' | 'and' | 'along' | 'with' | 'can' | 'get' | "i'd" | 'traveling' |␣
 ↪'for' | 'me' | UNK | '.' | '?'
""")
```

Solution

[ ]:
```
#Solution
miniATIS = nltk.CFG.fromstring("""
S -> PREJ DET ADJS FLIGHT PPS | DET ADJS FLIGHT PPS | PREJ ADJS FLIGHT PPS | ␣
 ↪ADJS FLIGHT PPS | PREJ DET FLIGHT PPS | DET FLIGHT PPS | PREJ FLIGHT PPS |␣
 ↪FLIGHT PPS | PREJ DET ADJS FLIGHT  | DET ADJS FLIGHT | PREJ ADJS FLIGHT |␣
 ↪ADJS FLIGHT | PREJ DET FLIGHT | DET FLIGHT | PREJ FLIGHT | FLIGHT
PPS -> PP | PP PPS
ADJS -> ADJ | ADJ ADJS

FLIGHT -> 'flights' | 'flight' | 'to' 'fly'

PREJ -> JUNK PREJ | JUNK
JUNK -> 'me' | 'show' | 'now' | 'only' | 'can' | 'you' | 'the' | 'itinerary' |␣
 ↪'of' | 'also' | 'a' | 'list' | 'could' | 'give' | 'which' | 'what' | 'is' |␣
 ↪"what's" | 'are' | 'my' | 'choices' | 'for' | 'i' | 'would' | 'like' | "i'd"␣
 ↪| 'to' | 'see' | 'have' | 'make' | 'book' | 'find' | 'information' | 'on' |␣
 ↪'know' | 'some' | 'hello' | 'yes' | 'please' | 'repeat' | 'do' | 'have' |␣
 ↪'there' | 'need' | 'hi' | 'get' | 'may' | 'listing' | 'listings' | 'travel' |␣
 ↪'arrangements' | 'okay' | 'want' | 'tell' | 'about' | 'how' | 'would' | 'be'␣
 ↪| 'able' | 'put' | 'requesting' | "i'm" | 'looking' | 'display' | UNK

ADJ -> AIRLINE | SIMPLEWEEKDAY | FLIGHTTYPE | SIMPLETIME | "monday's" |␣
 ↪"tuesday's" | "wednesday's" | "thursday's" | "friday's" | "saturday's" |␣
 ↪"sunday's" | 'available' | 'possible' | 'first' 'class' | 'economy' |␣
 ↪'thrift' 'economy' | 'cheapest' | 'lowest' 'cost' | 'least' 'expensive' |␣
 ↪'most' 'expensive' | 'weekday' | 'daily' | 'last' | 'first' | 'dinner' |␣
 ↪'transcontinental'

DET -> 'all' 'the' | 'all' | A | 'an' | THE | 'any' | 'all' 'of' 'the'
BETWEEN -> 'between'
AND -> 'and'
OR -> 'or'
EITHER -> 'either'
OF -> 'of'
THE -> 'the'
A -> 'a'
```

```
PP -> PPLACE PLACE OR PLACE | PPLACE EITHER PLACE OR PLACE | PPLACE PLACE |␣
 ↪EITHER PLACE OR PLACE | PLACE OR PLACE | PLACE | BETWEEN PLACE AND PLACE |␣
 ↪BETWEEN TIME AND TIME | PTIME TIME | TIME | PDAY WEEKDAY | WEEKDAY TIME |␣
 ↪WEEKDAY | PDAY WEEKDAY TIME | PDAY DATE | DATE | PAIRLINE AIRLINE | AIRCRAFT␣
 ↪| FLIGHTTYPE | FARETYPE | PRICE | FOOD | AVAIL | POSTJ

PPLACE -> 'to' | 'that' 'arrive' 'at' | 'that' 'arrives' 'in' | 'coming' 'back'␣
 ↪'to' | 'that' 'go' 'to' | 'and' 'then' 'to' | 'arriving' 'in' | 'and'␣
 ↪'arriving' 'in' | 'and' 'arrive' 'in' | 'to' 'arrive' 'in' | 'arrive' 'in' |␣
 ↪'going' 'to' | 'into' | 'for' | 'with' 'the' 'destination' 'city' 'of' |␣
 ↪'arriving' | 'goes' 'to' | 'flying' 'into' | 'goes' 'on' 'to' | 'reaching' |␣
 ↪'in' | 'and' 'then' | 'arriving' 'to' | 'from' | 'leaving' | 'return' 'from'␣
 ↪| 'leaving' 'from' | 'departing' 'from' | 'departing' | 'go' 'from' | 'going'␣
 ↪'from' | 'back' 'from' | 'that' 'goes' 'from' | 'that' 'departs' | 'which'␣
 ↪'leaves' 'from' | 'which' 'leave' | 'that' 'leave' | 'originating' 'in' |␣
 ↪'leave' | 'out' 'of' | 'leaves' 'from' | 'to' 'get' 'from' | 'via' | 'with'␣
 ↪'a' 'stopover' 'in' | 'with' 'a' 'layover' 'in' | 'with' 'a' 'stopover' 'at'␣
 ↪| 'and' 'a' 'stopover' 'in' | 'stop' 'in' | 'stopping' 'in' | 'make' 'a'␣
 ↪'stop' 'in' | 'with' 'a' 'stop' 'in' | 'with' 'one' 'stop' 'in' | 'go'␣
 ↪'through' | 'which' 'go' 'through' | 'makes' 'a' 'stopover' 'in' | 'that'␣
 ↪'stops' 'in' | 'that' 'stops' 'over' 'in' | 'by' 'way' 'of' | 'connecting'␣
 ↪'through' | 'that' 'will' 'stop' 'in' | 'which' 'connects' 'in' | 'arriving'␣
 ↪'and' 'departing' 'at'

PLACE -> 'anywhere' | 'atlanta' | 'austin' | 'baltimore' | 'boston' | 'boston'␣
 ↪'logan' | 'burbank' | 'bwi' | 'charlotte' | 'chicago' | 'cincinnati' |␣
 ↪'cleveland' | 'cleveland' 'ohio' | 'columbus' | 'dallas' 'fort' 'worth' |␣
 ↪'dallas' | 'denver' | 'denver' 'colorado' | 'detroit' | 'fort' 'worth' |␣
 ↪'general' 'mitchell' 'international' | 'general' 'mitchell' | 'houston' |␣
 ↪'indianapolis' | 'jfk' | 'kansas' 'city' | 'laguardia' 'airport' | 'las'␣
 ↪'vegas' | 'long' 'beach' | 'los' 'angeles' | 'love' 'field' | 'memphis' |␣
 ↪'miami' | 'milwaukee' | 'minneapolis' | 'montreal' | 'montreal' 'quebec' |␣
 ↪'montreal' 'canada' | 'nashville' | 'new' 'york' 'city' | 'new' 'york' |␣
 ↪'newark' | 'newark' 'new' 'jersey' | 'oakland' | 'oakland' 'california' |␣
 ↪'ontario' | 'orlando' | 'orlando' 'florida' | 'philadelphia' | 'philly' |␣
 ↪'phoenix' | 'pittsburgh' | 'salt' 'lake' 'city' | 'san' 'diego' | 'san'␣
 ↪'diego' 'california' | 'san' 'francisco' | 'san' 'jose' | 'seattle' | 'st.'␣
 ↪'louis' | 'st.' 'paul' | 'st.' 'petersburg' | 'tacoma' | 'tacoma'␣
 ↪'washington' | 'tampa' | 'toronto' | 'various' 'cities' | 'washington' |␣
 ↪'washington' 'dc' | 'dc' | 'westchester' 'county' | UNK
```

```
PTIME -> 'that' 'arrive' 'before' | 'that' 'arrives' 'before' | 'arriving'␣
 ↪'before' | 'arrival' 'by' | 'arrives' | 'before' | 'departing' 'before' |␣
 ↪'that' 'leaves' 'before' | 'which' 'arrive' 'before' | 'by' | 'around' |␣
 ↪'that' 'return' 'around' | 'that' 'gets' 'in' 'around' | 'at' | 'arriving'␣
 ↪'around' | 'arriving' 'about' | 'that' 'arrive' 'soon' 'after' | 'leaving'␣
 ↪'at' | 'leaving' | 'which' 'leave' 'after' | 'leaving' 'after' | 'after' |␣
 ↪'departing' 'after' | 'that' 'depart' 'after' | 'departing' 'at' | 'arriving'␣
 ↪'after' | 'in' | 'departing' 'in' | 'on' | 'that' 'leaves' 'in'

TIME -> SIMPLETIME | 'the' 'afternoon' | 'the' 'late' 'afternoon' | 'the'␣
 ↪'evening' | 'the' 'morning' | 'the' 'early' 'am' | 'the' 'day' | 'mornings' |␣
 ↪'afternoons' | 'evenings' | 'as' 'early' 'as' 'possible' | 'earliest'␣
 ↪'possible' 'time' | 'as' 'soon' 'thereafter' 'as' 'possible' | UNK

SIMPLETIME -> 'one' | 'two' | 'three' | 'four' | 'five' | 'six' | 'seven' |␣
 ↪'eight' | 'nine' | 'ten' | 'eleven' | 'twelve' | '1' | '2' | '3' |'4' | '5' |␣
 ↪'6' | '7' |  '8' | '9' | '10' | '11' | '12' | '1pm' | '2pm' | '3pm' | '4pm' |␣
 ↪'5pm' | '6pm' | '7pm' | '8pm' | '9pm' | '10pm' | '11pm' | '12pm' | '1am' |␣
 ↪'2am' | '3am' | '4am' | '5am' | '6am' | '7am' | '8am' | '9am' | '10am' |␣
 ↪'11am' | '12am' | '230' | '1505' | '630am' | '720am' | '723pm' | '819' | '845'␣
 ↪| '1026' | '1145am' | '1' "o'clock" | '1' "o'clock" 'am' | "1" "o'clock" "pm"␣
 ↪| "2" "o'clock" | "2" "o'clock" "am" | "2" "o'clock" "pm" | "3" "o'clock" |␣
 ↪"3" "o'clock" "am" | "3" "o'clock" "pm" | "4" "o'clock" | "4" "o'clock" "pm" |␣
 ↪"4" "o'clock" "am" | "5" "o'clock" | "5" "o'clock" "am" | "5" "o'clock" "pm"␣
 ↪| "6" "o'clock" | "6" "o'clock" "pm" | "6" "o'clock" "am" | "7" "o'clock" |␣
 ↪"7" "o'clock" "pm" | "7" "o'clock" "am" | "8" "o'clock" | "8" "o'clock" "am"␣
 ↪| "8" "o'clock" "pm" | "9" "o'clock" | "9" "o'clock" "pm" | "9" "o'clock"␣
 ↪"am" | "10" "o'clock" | "10" "o'clock" "am" | "10" "o'clock" "pm" | "11"␣
 ↪"o'clock" | "11" "o'clock" "am" | "11" "o'clock" "pm" | "12" "o'clock" | "12"␣
 ↪"o'clock" "pm" | "12" "o'clock" "am" |  'noon' | '12' 'noon' | "12" "o'clock"␣
 ↪"noon" | 'midnight' | '12' 'midnight' | 'lunch' 'time' | 'dinnertime' | ␣
 ↪'evening' | 'night' | 'morning' | 'afternoon' | 'early' | 'late' | 'tonight'␣
 ↪| 'earliest' 'possible' | 'earliest' | 'latest' 'possible' | 'latest'


PDAY -> 'on' | 'returning' 'on' | 'of' | 'for' | 'next' | 'the' 'next' | 'in'␣
 ↪'the' 'next' | 'of' 'next' | 'leaving' | 'arriving' | 'arriving' 'on' |␣
 ↪'which' 'leave' | 'that' 'arrive' 'on' | 'leaving' 'on' | 'which' 'arrive'␣
 ↪'on' | 'a' 'week' 'from'

SIMPLEWEEKDAY -> 'saturday' | 'sunday' | 'monday' | 'tuesday' | 'wednesday' |␣
 ↪'thursday' | 'friday'
```

```
WEEKDAY -> SIMPLEWEEKDAY | A SIMPLEWEEKDAY | 'saturdays' | 'sundays' | 'mondays'␣
 ↪| 'tuesdays' | 'wednesdays' | 'thursdays' | 'fridays' | 'this' 'saturday' |␣
 ↪'this' 'sunday' | 'this' 'monday' | 'this' 'tuesday' | 'this' 'wednesday' |␣
 ↪'this' 'thursday' | 'this' 'friday' | 'this' 'coming' 'saturday' | 'this'␣
 ↪'coming' 'sunday' | 'this' 'coming' 'monday' | 'this' 'coming' 'tuesday' |␣
 ↪'this' 'coming' 'wednesday' | 'this' 'coming' 'thursday' | 'this' 'coming'␣
 ↪'friday' | 'day' | 'week' | 'today' | 'tomorrow' | 'the' 'day' 'after'␣
 ↪'tomorrow' | 'a' 'weekday' | 'weekdays'

DATE -> MONTH DAY YEAR | MONTH DAY | THE DAY | MONTH DAY OR DAY | THE DAY OR DAY␣
 ↪| EITHER MONTH DAY OR DAY | EITHER THE DAY OR THE DAY | THE DAY OF MONTH | DAY␣
 ↪OF MONTH | EITHER THE DAY OR THE DAY OF MONTH | THE DAY OF MONTH OR THE DAY OF␣
 ↪MONTH | EITHER THE DAY OF MONTH OR THE DAY OF MONTH
MONTH -> 'january' | 'february' | 'march' | 'april' | 'may' | 'june' | 'july' |␣
 ↪'august' | 'september' | 'october' | 'november' | 'december'
DAY -> 'first' | 'second' | 'third' | 'fourth' | 'fifth' | 'sixth' | 'seventh' |␣
 ↪'eighth' | 'ninth' | 'tenth' | 'eleventh' | 'twelfth' | 'thirteenth' |␣
 ↪'fourteenth' | 'fifteenth' | 'sixteenth' | 'seventeenth' | 'eighteenth' |␣
 ↪'nineteenth' | 'twentieth' | 'twenty-first' | 'twenty' 'first' |␣
 ↪'twenty-second' | 'twenty' 'second' | 'twenty-third' | 'twenty' 'third' |␣
 ↪'twenty-fourth' | 'twenty' 'fourth' | 'twenty-fifth' | 'twenty' 'fifth' |␣
 ↪'twenty-sixth' | 'twenty' 'sixth' | 'twenty-seventh' | 'twenty' 'seventh' |␣
 ↪'twenty-eighth' | 'twenty' 'eighth' | 'twenty-ninth' | 'twenty' 'ninth' |␣
 ↪'thirtieth' | 'thirty-first' | 'thirty' 'first' | 'one' | 'two' | 'three' |␣
 ↪'four' | 'five' | 'six' | 'seven' | 'eight' | 'nine' | 'ten' | 'eleven' |␣
 ↪'twelve' | 'thirteen' | 'fourteen' | 'fifteen' | 'sixteen' | 'seventeen' |␣
 ↪'eighteen' | 'nineteen' | 'twenty' | 'twenty' 'one' | 'twenty' 'two' |␣
 ↪'twenty' 'three' | 'twenty' 'four' | 'twenty' 'five' | 'twenty' 'six' |␣
 ↪'twenty' 'seven' | 'twenty' 'eight' | 'twenty' 'nine' | 'thirty' | 'thirty'␣
 ↪'one'
YEAR -> '1991' | '1992'

PAIRLINE -> 'on' | 'using' | 'of' | 'with'

AIRLINE -> 'continental' | 'continental' 'airline' | 'continental' 'airlines' |␣
 ↪'american' 'airlines' | 'american' 'airlines' | 'american' | 'united'␣
 ↪'airlines' | 'united' | 'united' 'airline' | 'northwest' 'and' 'united' |␣
 ↪'northwest' 'airlines' | 'northwest' 'airline' | 'northwest' | 'us' 'air' |␣
 ↪'us' 'airlines' | 'delta' | 'twa' | 'air' 'canada' | 'eastern' 'airlines' |␣
 ↪'midwest' 'express' | 'trans' 'world' 'airline'

AIRCRAFT -> 'using' 'a' 'j31' 'aircraft'

FLIGHTTYPE -> 'round' 'trip' | 'round-trip' | 'nonstop' | 'one' 'way' | 'return'␣
 ↪| 'direct' | 'connecting' | 'direct' 'and' 'connecting' | 'nonstop' 'or'␣
 ↪'connecting'
```

```
FARETYPE -> 'with' 'economy' 'fares' | 'qualify' 'for' 'fare' 'code' 'qx' |␣
 ↪'with' 'first' 'class' 'service' | 'that' 'offer' 'first' 'class' | 'on'␣
 ↪'first' 'class' | 'economy' 'class' | 'with' 'the' 'lowest' 'one' 'way'␣
 ↪'fares' | 'with' 'a' 'class' 'of' 'service' 'code' 'f' | 'with' 'q' 'fares' |␣
 ↪'first' 'class' 'fare'

PRICE -> 'less' 'than' '1100' 'dollars' | 'with' 'the' 'highest' 'fare' | 'the'␣
 ↪'cheapest' 'way' 'possible' | 'less' 'than' '466' 'dollars' | 'lowest' 'cost'␣
 ↪| 'least' 'expensive' | 'most' 'expensive' | 'cheapest'

FOOD -> 'dinner' | 'a' 'meal' | 'lunch' | 'breakfast'

AVAIL -> 'available' | 'i' 'can' 'get'

POSTJ -> 'please' | 'there' | 'are' | 'currently' | 'do' | 'you' | 'have' |␣
 ↪'fares' | 'information' | 'i' | 'want' | 'would' | 'like' | 'the' | 'flight'␣
 ↪| 'be' | 'go' | 'departures' | 'is' | 'such' | 'a' | 'that' | 'serves' |␣
 ↪'both' | 'and' | 'along' | 'with' | 'can' | 'get' | "i'd" | 'traveling' |␣
 ↪'for' | 'me' | UNK | '.' | '?'
""")
```

To iterate over the constructed grammar, we can use `grammar.productions` method.

[ ]:
```
for production in miniATIS.productions():
  lhs = production.lhs() # left hand side
  rhs = production.rhs() # right hand side
  print (f'Rule: {lhs} -> {rhs}')
  print (f'{len(rhs)}')
  break
```

To convert the constructed grammar to CNF, we use `get_cnf_grammar`.

[ ]:
```
normal_miniATIS, miniATIS_wunaries = get_cnf_grammar(miniATIS)
print (normal_miniATIS.is_chomsky_normal_form())
```

Now, we can use `normal_miniATIS` for CKY recognition and parsing.

[ ]:
```
for production in normal_miniATIS.productions():
  lhs = production.lhs() # left hand side
  rhs = production.rhs() # right hand side
  print (f'Rule: {lhs} -> {rhs}')
  print (f'{len(rhs)}')
  break
```

1.4 Deterministic parsing

1.4.1 Goal 2: CKY Recognition

Implement a recognizer using the CKY algorithm to determine if a sentence is parsable. See J&M Chapter 13 for notes on CKY and the pseudo-code of the algorithm.

[ ]:

```
#TODO
def cky_recognize(grammar, s):
  """
  CKY algorithm for recognition.
  Arguments:
        grammar: a CFG in CNF
        s: the input string to parse
  Returns whether sentence is parsable
  """
  assert(grammar.is_chomsky_normal_form())

  # TODO: Implement a CKY Recognizer
  return False
```

Solution

[ ]:

```
#Solution
def cky_recognize(grammar, s):
  """
  CKY algorithm for recognition.
  Arguments:
        grammar: a CFG in CNF
        s: the input string to parse
  Returns whether sentence is parsable
  """
  assert(grammar.is_chomsky_normal_form())
  nonterminals = set([])

  # Get the nonterminals
  for production in grammar.productions():
    nonterminals.add(production.lhs())
    assert(type(production.lhs()) == Nonterminal)
  num_nonterminals = len(nonterminals)

  nonterminals = list(nonterminals)

  # Setup lookup table
  dynamic_table = {}

  # Lookup table initialization
  for i in range(len(s)):
    dynamic_table[i] = {}
    for j in range(len(s), i, -1):
      dynamic_table[i][j] = {}
      for k in nonterminals:
        dynamic_table[i][j][k] = False

  # Iterate over each token
  for j in range(1, len(s)+1):

    # Unary productions of the form A -> token[j-1]
    for production in list(filter(lambda p : len(p.rhs()) == 1 and p.rhs()[0] ==␣
 ↪s[j-1], grammar.productions())):
      dynamic_table[j-1][j][production.lhs()] = True

    # Spans from size j-2 to 0
    for i in range(j-2, -1, -1):
      # Iterate over possible midpoints
      for k in range(i+1, j):
        for production in list(filter(lambda p : len(p.rhs()) == 2,grammar.
 ↪productions())):
          if dynamic_table[i][k][production.rhs()[0]] and␣
 ↪dynamic_table[k][j][production.rhs()[1]]:
            dynamic_table[i][j][production.lhs()] = True

  # Is the sentence in the grammar?
  return dynamic_table[0][len(s)][grammar.start()]
```

You can use the following test sentences to test your code:

[ ]:

```
# A grammartical sentence
test_sentence = "show me the flights on united that arrive before 4 between␣
 ↪atlanta and new york".split()
result = cky_recognize(normal_miniATIS, test_sentence)
print (result)

# An ungrammatical sentence
test_sentence = "show me the flights on united that between atlanta and new york␣
 ↪arrive before 4 ".split()
result = cky_recognize(normal_miniATIS, test_sentence)
print (result)
```

1.4.2 Goal 3: CKY Parsing

Implement the CKY algorithm for parsing under CFGs. You only need to add a few lines of code to your CKY recognizer to achieve this, using back-pointers. The expected return value is an NLTK tree, which can be constructed using `Tree.fromstring`.

A standard tree string will be of the following form:

`(S (A B) (C (D E) (F G)))`

which corresponds to the following tree (drawn using tree.pretty_print()):

```
     S
  ___|___
 |       C
 |    ___|___
 A   D       F
 |   |       |
 B   E       G
```

[ ]:

```python
#TODO
def cky_parse(grammar, s):
  """
  CKY algorithm for parsing.
  Arguments:
        grammar: a CFG in CNF
        s: the input string to parse
  Returns an nltk Tree if parsable or None if not parsable
  """
  assert(grammar.is_chomsky_normal_form())

  # TODO: Implement a CKY Parser
  return Tree.fromstring('(S (A B) (C (D E) (F G)))')
```

Solution

[ ]:

```python
#Solution
def cky_parse(grammar, s):
  """
  CKY algorithm for parsing.
  Arguments:
        grammar: a CFG in CNF
        s: the input string to parse
  Returns an nltk Tree if parsable or None if not parsable
  """
  assert(grammar.is_chomsky_normal_form())
  nonterminals = set([])
  # Get the nonterminals
  for production in grammar.productions():
    nonterminals.add(production.lhs())
    assert(type(production.lhs()) == Nonterminal)
  num_nonterminals = len(nonterminals)
```

```
nonterminals = list(nonterminals)

# Setup lookup table
dynamic_table = {}

# bottom up parse tree
tree = {}

# Initialization of lookup table
for i in range(len(s)):
  dynamic_table[i] = {}
  tree[i] = {}
  for j in range(len(s), i, -1):
    dynamic_table[i][j] = {}
    tree[i][j] = {}
    for k in nonterminals:
      dynamic_table[i][j][k] = False
      tree[i][j][k] = ''

# For each token
for j in range(1, len(s)+1):
  # Unit productions of the form A -> token[j - 1]
  for production in list(filter(lambda p : len(p.rhs()) == 1 and p.rhs()[0] == s[j-1], grammar.productions())):
    dynamic_table[j-1][j][production.lhs()] = True
    tree[j-1][j][production.lhs()] = '(' + production.lhs().__str__() + ' ' + s[j-1] + ')'

  # For spans from size j-2 to 0
  for i in range(j-2, -1, -1):

    # Iterate over possible midpoints
    for k in range(i+1, j):
      # Binary productions
      for production in list(filter(lambda p : len(p.rhs()) == 2,grammar.productions())):
        if dynamic_table[i][k][production.rhs()[0]] and dynamic_table[k][j][production.rhs()[1]]:
          dynamic_table[i][j][production.lhs()] = True
          tree[i][j][production.lhs()] = '(' + production.lhs().__str__() + ' ' + tree[i][k][production.rhs()[0]] + ' ' + tree[k][j][production.rhs()[1]] + ')'

# If the sentence is in the grammar return the constructed tree
if dynamic_table[0][len(s)][grammar.start()]:
  print (tree[0][len(s)][grammar.start()])
  return Tree.fromstring(tree[0][len(s)][grammar.start()])
else:
  return None
```

You can use the following test sentence to test your code:

[ ]:

```
test_sentence = "show me the flights on united that arrive before 4 between␣
 ↪atlanta and new york".split()
tree = cky_parse(normal_miniATIS, test_sentence)
un_cnf(tree, miniATIS_wunaries) # convert back to original grammar
tree.pretty_print()
```

You can also compare against a built-in nltk parser:

[ ]:

```
parser = nltk.parse.BottomUpChartParser(miniATIS)
parses = [p for p in parser.parse(test_sentence)]
print('Reference parse:')
for ptree in parses:
  print(ptree)

print ('Predicted parse:')
print (tree)

print("Parses match (at least one):", tree in parses)
```

Grammar Coverage To check your implementation, let's compute the fraction of the original ATIS dataset that parses under the ATIS grammar. **This grammar should cover 2116 out of the 4379 sentences.**

[ ]:

```
# This script might take up to 30min.
# We recommend to make sure the above checks work first.
all_sents_big_ATIS = []
with open('data/train.nl') as f:
  for line in f:
    all_sents_big_ATIS.append(line.split())

parsed = 0
total = 0
for sent in all_sents_big_ATIS:
  total += 1
  sent = [tok.lower() for tok in sent]

  tree = cky_parse(normal_miniATIS, sent)
  if tree:
    parsed += 1

print(parsed, total)
```

1.5 Probabilistic parsing

In practice, we want to work with grammars that cover nearly all the language we expect to come across for a given application. This leads to an explosion of rules and a large number of possible parses for any one sentence. To remove ambiguity between the different parses, it's desirable to move to PCFGs. In this part, you will construct a PCFG from data, parse using a probabalistic version of CKY, and evaluate the quality of the resulting parses against gold trees.

1.5.1 Goal 4: PCFG construction

Compared to CFGs, PCFGs need to assign probabilities to grammar rules. Fill in the missing pieces of the below method to build a PCFG in CNF by counting using the *train* split (`train.trees` that we downloaded in Setup). Note that we want the PCFG to be in CNF format because the probabalistic version of CKY also requires the grammar to be in CNF. However, the gold trees are not in CNF form, so in this case you will need to convert the gold *trees* to CNF before building the PCFG. To accomplish this, we will use the `treetransforms` package from `nltk`, which includes functions for converting to and from CNF.

Finish the below code to construct a PCFG grammar. You will use it later for parsing.

[ ]:
```
#TODO
def construct_pcfg_rules(gold_trees):
  """
  Construct PCFG grammar rules from gold parse trees.
  Arguments:
        gold_trees: a list of gold parse trees
  Returns a set of ProbabilisticProduction objects constructed by
        ProbabilisticProduction(lhs, rhs, prob=prob)
  """
  productions_with_probs = set([])

  pcfg_data = defaultdict(Counter)
  for tree in gold_trees:
    # Convert to CNF
    treetransforms.collapse_unary(tree, collapsePOS=True)
    treetransforms.chomsky_normal_form(tree)
    # TODO: implement here
    # Construct a PCFG by counting rule frequencies from these trees
    # Hint: You can get the list of productions from the tree via tree.
 ↪productions()

  # TODO: implement here
  # Compute probabilities using the counts
  return productions_with_probs
```

Solution

[ ]:

```
#Solution
def construct_pcfg_rules(gold_trees):
  """
  Construct PCFG grammar rules from gold parse trees.
  Arguments:
        gold_trees: a list of gold parse trees
  Returns a set of ProbabilisticProduction objects constructed by
        ProbabilisticProduction(lhs, rhs, prob=prob)
  """
  productions_with_probs = set([])

  pcfg_data = defaultdict(Counter)
  for tree in gold_trees:
    # Convert to CNF
    treetransforms.collapse_unary(tree, collapsePOS=True)
    treetransforms.chomsky_normal_form(tree)
    # Count rule frequencies
    for production in tree.productions():
      pcfg_data[production.lhs()][production.rhs()] += 1
  # Compute probabilities
  for lhs, rhs_counter in pcfg_data.items():
    total_count = sum(rhs_counter.values())
    for rhs, count in rhs_counter.items():
      prob = count/total_count
      productions_with_probs.add(ProbabilisticProduction(lhs, rhs, prob=prob))
  return productions_with_probs
```

[ ]:

```
# Load gold trees
gold_trees = []
with open('data/train.trees') as f:
    for line in f:
      gold_trees.append(nltk.Tree.fromstring(line.strip()))

productions_with_probs = construct_pcfg_rules(gold_trees)
pgrammar = PCFG(Nonterminal('TOP'), productions_with_probs)
print (pgrammar.is_chomsky_normal_form())
```

1.5.2 Goal 5: Probabalistic CKY parsing

Finally, we are ready to implement CKY parsing under PCFGs. Adapt the CKY parser from Goal 3 to return the most likely parse and its probability given a PCFG.

[ ]:

```
#TODO
def cky_parse_probabalistic(grammar, s):
  """
  CKY algorithm for parsing using PCFG.
  Arguments:
        grammar: a PCFG in CNF
        s: the input string to parse
  Returns twov values:
        first: the most likely nltk Tree if parsable or None if not parsable
        second: the highest probability
  """
  assert(grammar.is_chomsky_normal_form())

  # TODO: Implement a probabilisitc CKY parser
  return Tree.fromstring('(S (A B) (C (D E) (F G)))'), 0
```

Solution

[ ]:

```
#Solution
def cky_parse_probabalistic(grammar, s):
  """
  CKY algorithm for parsing using PCFG.
  Arguments:
        grammar: a PCFG in CNF
        s: the input string to parse
  Returns twov values:
        first: the most likely nltk Tree if parsable or None if not parsable
        second: the highest probability
  """
  assert(grammar.is_chomsky_normal_form())
  nonterminals = set([])

  # Get the nonterminals
  for production in grammar.productions():
    nonterminals.add(production.lhs())
    assert(type(production.lhs()) == Nonterminal)
  num_nonterminals = len(nonterminals)

  nonterminals = list(nonterminals)

  # Setup lookup table
  dynamic_table = {}

  # bottom up parse tree
  tree = {}

  # Dynamic table initialization
  for i in range(len(s)):
    dynamic_table[i] = {}
    tree[i] = {}
    for j in range(len(s), i, -1):
      dynamic_table[i][j] = {}
      tree[i][j] = {}
      for k in nonterminals:
        dynamic_table[i][j][k] = None
        tree[i][j][k] = ''

  # Iterate over each token
  for j in range(1, len(s)+1):
    # Unit productions of the form A -> tokens[j-1]
    for production in list(filter(lambda p : len(p.rhs()) == 1 and p.rhs()[0] == s[j-1], grammar.productions())):
      dynamic_table[j-1][j][production.lhs()] = production.logprob()
      tree[j-1][j][production.lhs()] = '(' + production.lhs().__str__() + ' ' + s[j-1] + ')'

    # Span length from j-2 to 0
    for i in range(j-2, -1, -1):

      # Iterate over all midpoints
      for k in range(i+1, j):

        # Iterate over each binary production
        for production in list(filter(lambda p : len(p.rhs()) == 2,grammar.productions())):
          A = dynamic_table[i][j][production.lhs()]
          B = dynamic_table[i][k][production.rhs()[0]]
          C = dynamic_table[k][j][production.rhs()[1]]
          if B is not None and C is not None and (A is None or B + C + production.logprob() > A):
            dynamic_table[i][j][production.lhs()] = B + C + production.logprob()
            tree[i][j][production.lhs()] = '(' + production.lhs().__str__() + ' ' + tree[i][k][production.rhs()[0]] + ' ' + tree[k][j][production.rhs()[1]] + ')'

  # If the sentence is in the grammar return the constructed tree
  if dynamic_table[0][len(s)][grammar.start()] is not None:
    return Tree.fromstring(tree[0][len(s)][grammar.start()]), 2**dynamic_table[0][len(s)][grammar.start()]
  else:
    return None, None
```

Evaluation There are a number of ways to evaluate parsing algorithms. In this assignment we will use the bracket scoring program `evalb` which we have downloaded during setup. **You are expected to achieve precision of 0.67, recall of 0.41 and F1 of 0.51.**

Read in the test data:

[ ]:

```python
all_sents = []

with open('data/test.trees') as f:
  for line in f:
    t = Tree.fromstring(line.strip())
    all_sents.append(t.leaves())
```

Parse the test sentences using your implementation, and write output to a file:

[ ]:

```python
trees_out = []
for sent in all_sents:
  tree, _ = cky_parse_probabalistic(pgrammar, sent)
  if tree is not None:
    trees_out.append(tree.pformat(margin=9999999999))
  else:
    trees_out.append('()')

with open('outp.trees', 'w') as f:
  for line in trees_out:
    f.write(line + '\n')
```

Compare the predicted trees to the ground truth trees.

[ ]:

```
!python scripts/evalb.py outp.trees data/test.trees
```