In [7]:

```
!gdown --id 1hTgKXDk_cJGewSdSboERmssec7DHAu5d
```

```
/usr/local/lib/python3.10/dist-packages/gdown/cli.py:121: FutureWarning: Option `--id` wa
s deprecated in version 4.3.1 and will be removed in 5.0. You don't need to pass it anymo
re to use a file ID.
  warnings.warn(
Downloading...
From: https://drive.google.com/uc?id=1hTgKXDk_cJGewSdSboERmssec7DHAu5d
To: /content/requirements.txt
100% 20.0/20.0 [00:00<00:00, 102kB/s]
```

In [8]:

```
!pip install  -r /content/requirements.txt
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publi
c/simple/
Requirement already satisfied: nltk==3.5 in /usr/local/lib/python3.10/dist-packages (from
-r /content/requirements.txt (line 1)) (3.5)
Requirement already satisfied: wget in /usr/local/lib/python3.10/dist-packages (from -r /
content/requirements.txt (line 2)) (3.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from -r /
content/requirements.txt (line 3)) (4.65.0)
Requirement already satisfied: click in /usr/local/lib/python3.10/dist-packages (from nlt
k==3.5->-r /content/requirements.txt (line 1)) (8.1.3)
Requirement already satisfied: joblib in /usr/local/lib/python3.10/dist-packages (from nl
tk==3.5->-r /content/requirements.txt (line 1)) (1.2.0)
Requirement already satisfied: regex in /usr/local/lib/python3.10/dist-packages (from nlt
k==3.5->-r /content/requirements.txt (line 1)) (2022.10.31)
```

In [9]:

```
# 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/cs236299-2023-spring/project3.git .tmp
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

%%latex \newcommand{\vect}[1]{\mathbf{#1}} \newcommand{\cnt}[1]{\sharp(#1)} \newcommand{\argmax}[1]{\underset{#1}{\operatorname{argmax}}} \newcommand{\softmax}{\operatorname{softmax}} \newcommand{\Prob}{\Pr} \newcommand{\given}{\,|\,}

236299 - Introduction to Natural Language Processing

Project 3: Parsing – The CKY Algorithm

Constituency parsing is the recovery of a labeled hierarchical structure, a *parse tree* for a sentence of a natural language. It is a core intermediary task in natural-language processing, as the meanings of sentences are related to their structure.

In this project, you will implement the CKY algorithm for parsing strings relative to context-free grammars (CFG). You will implement versions for both non-probabilistic context-free grammars (CFG) and probabilistic grammars (PCFG) and apply them to the parsing of ATIS queries.

The project is structured into five parts:

1. **Finish a CFG for the ATIS dataset.**
2. **Implement the CKY algorithm for *recognizing* grammatical sentences, that is, determining whether a parse exists for a given sentence.**
3. **Extend the CKY algorithm for *parsing* sentences, that is, constructing the parse trees for a given sentence.**
4. **Construct a probabilistic context-free grammar (PCFG) based on a CFG.**
5. **Extend the CKY algorithm to PCFGs, allowing the construction of the most probable parse tree for a sentence according to a PCFG.**

Setup

In [10]:

```
# Download needed files and scripts
import wget
os.makedirs('data', exist_ok=True)
os.makedirs('scripts', exist_ok=True)
# ATIS queries
wget.download("https://raw.githubusercontent.com/nlp-236299/data/master/ATIS/train.nl", o
ut="data/")
# Corresponding parse trees
wget.download("https://raw.githubusercontent.com/nlp-236299/data/master/ATIS/train.trees"
, out="data/")
wget.download("https://raw.githubusercontent.com/nlp-236299/data/master/ATIS/test.trees",
out="data/")

# Code for comparing and evaluating parse trees
wget.download("https://raw.githubusercontent.com/nlp-236299/data/master/scripts/trees/eva
lb.py", out="scripts/")
wget.download("https://raw.githubusercontent.com/nlp-236299/data/master/scripts/trees/tra
nsform.py", out="scripts/")
wget.download("https://raw.githubusercontent.com/nlp-236299/data/master/scripts/trees/tre
e.py", out="scripts/")
```

Out[10]:

```
'scripts//tree.py'
```

In [11]:

```
import shutil

import nltk

import sys

from collections import defaultdict, Counter

from nltk import treetransforms
from nltk.grammar import ProbabilisticProduction, PCFG, Nonterminal
```

```
from nltk.tree import Tree

from tqdm import tqdm

# Import functions for transforming augmented grammars
sys.path.insert(1, './scripts')
import transform as xform
```

In [12]:

```
## Debug flag used below for turning on and off some useful tracing
DEBUG = False
```

A custom ATIS grammar

To parse, we need a grammar. In this project, you will use a hand-crafted grammar for a fragment of the ATIS dataset. The grammar is written in a "semantic grammar" style, in which the nonterminals tend to correspond to semantic classes of phrases, rather than syntactic classes. By using this style, we can more closely tune the grammar to the application, though we lose generality and transferability to other applications. The grammar will be used again in the next project segment for a question-answering application.

We download the grammar to make it available.

In [13]:

```
if not os.path.exists('./data/grammar_distrib3'):
  wget.download("https://raw.githubusercontent.com/nlp-236299/data/master/ATIS/grammar_distrib3", out="data/")
if os.path.exists('./data/grammar_distrib3') and (not os.path.exists('./data/grammar')):
  shutil.copy('./data/grammar_distrib3', './data/grammar')
```

Take a look at the file `data/grammar_distrib3` that you've just downloaded. The grammar is written in a format that extends the NLTK format expected by `CFG.fromstring`. We've provided functions to make use of this format in the file `scripts/transform.py`. You should familiarize yourself with this format by checking out the documentation in that file.

> **We made a copy of this grammar for you as `data/grammar`. This is the file you'll be modifying in the next section. You can leave it alone for now.**

As described there, we can read the grammar in and convert it into NLTK's grammar format using the provided `xform.read_augmented_grammar` function.

In [14]:

```
atis_grammar_distrib, _ = xform.read_augmented_grammar("grammar_distrib3", path="data")
```

To verify that the ATIS grammar that we distributed is working, we can parse a sentence using a built-in NLTK parser. We'll use a tokenizer built with NLTK's tokenizing apparatus.

In [15]:

```
## Tokenizer
tokenizer = nltk.tokenize.RegexpTokenizer('\d+|[\w-]+|\$[\d\.]+|\S+')
def tokenize(string):
  return tokenizer.tokenize(string.lower())

## Demonstrating the tokenizer
## Note especially the handling of `"11pm"` and hyphenated words.
print(tokenize("Are there any first-class flights at 11pm for less than $3.50?"))
```

```
['are', 'there', 'any', 'first-class', 'flights', 'at', '11', 'pm', 'for', 'less', 'than'
, '$3.50', '?']
```

In [16]:

In [16]:

```
## Test sentence
test_sentence_1 = tokenize("show me the flights before noon")

## Construct parser from distribution grammar
atis_parser_distrib = nltk.parse.BottomUpChartParser(atis_grammar_distrib)

## Parse and print the parses
parses = atis_parser_distrib.parse(test_sentence_1)
for parse in parses:
  parse.pretty_print()
```

```
                                                 S
                        _________________________|________________________
                       |                                              NP_FLIGHT
                       |                                                  |
                       |                                              NOM_FLIGHT
                       |                                                  |
                       |                                               N_FLIGHT
                       |                                       ___________|_________
                   PREIGNORE                                   |                    PP
        _______________|____________                           |                    |
       |                        PREIGNORE                      |                 PP_TIM
E
       |                ____________|____________              |           _________|___
_____
       |               |                     PREIGNORE      N_FLIGHT      |
NP_TIME
       |               |                         |             |          |
|
PREIGNORESYMBOL PREIGNORESYMBOL           PREIGNORESYMBOL TERM_FLIGHT   P_TIME
TERM_TIME
       |               |                         |             |          |
|
      show             me                       the         flights     before
noon
```

```
                                           S
                     ______________________|__________________
                    |                                     NP_FLIGHT
                    |                       __________________|_________
                    |                      |                        NOM_FLIGHT
                    |                      |                            |
                    |                      |                         N_FLIGHT
                    |                      |        ____________________|_________
                    |                      |       |                              PP
                    |                      |       |                              |
                PREIGNORE                  |       |                           PP_TIME
        ____________|_____________         |       |                     _________|_____
___
       |                     PREIGNORE     |    N_FLIGHT                |
```

```
NP_TIME
       |                          |          |       |                    |
|
PREIGNORESYMBOL            PREIGNORESYMBOL DET TERM_FLIGHT            P_TIME           T
ERM_TIME
       |                          |          |       |                    |
|
      show                       me         the    flights               before
noon
```

Testing the coverage of the grammar

We can get a sense of how well the grammar covers the ATIS query language by measuring the proportion of queries in the training set that are parsable by the grammar. We define a `coverage` function to carry out this evaluation.

> **Warning: It may take a long time to parse all of the sentence in the training corpus, on the order of 30 minutes. You may want to start with just the first few sentences in the corpus. The `coverage` function below makes it easy to do so, and in the code below we just test coverage on the first 50 sentences.**

In [17]:

```python
## Read in the training corpus
with open('data/train.nl') as file:
  training_corpus = [tokenize(line) for line in file]
```

In [18]:

```python
def coverage(recognizer, corpus, n=0):
  """Returns the proportion of the first `n` sentences in the `corpus`
  that are recognized by the `recognizer`, which should return a boolean.
  `n` is taken to be the whole corpus if n is not provided or is
  non-positive.
  """
  n = len(corpus) if n <= 0 else n
  parsed = 0
  total = 0
  for sent in tqdm(corpus[:n]):
    total += 1
    try:
      parses = recognizer(sent)
    except:
      parses = None
    if parses:
      parsed += 1
    elif DEBUG:
      print(f"failed: {sent}")
  if DEBUG: print(f"{parsed} of {total}")
  return parsed/total
```

In [19]:

```python
coverage(lambda sent: 0 < len(list(atis_parser_distrib.parse(sent))),  # trick for turnin
g parser into recognizer
         training_corpus, n=50)
```

```
100%|██████████| 50/50 [00:00<00:00, 86.02it/s]
```

Out[19]:

```
0.0
```

Sadly, you'll find that the coverage of the grammar is extraordinarily poor. That's because it is missing crucial parts of the grammar, especially phrases about *places*, which play a role in essentially every ATIS query. You'll

need to complete the grammar before it can be useful.

Part 1: Finish the CFG for the ATIS dataset

Consider the following query:

In [20]:

```python
test_sentence_2 = tokenize("show me the united flights from boston")
```

You'll notice that the grammar we distributed doesn't handle this query because it doesn't have a subgrammar for airline information (`"united"`) or for places (`"from boston"`).

In [21]:

```python
len(list(atis_parser_distrib.parse(test_sentence_2)))
```

Out[21]:

```
0
```

Follow the instructions in the grammar file `data/grammar` to add further coverage to the grammar. (You can and should leave the `data/grammar_distrib3` copy alone and use it for reference.)

We'll define a parser based on your modified grammar, so we can compare it against the distributed grammar. Once you've modified the grammar, this test sentence should have at least one parse.

> **You can search for "TODO" in `data/grammar` to find the two places to add grammar rules.**

In [25]:

```python
atis_grammar_expanded, _ = xform.read_augmented_grammar("grammar", path="data")
atis_parser_expanded = nltk.parse.BottomUpChartParser(atis_grammar_expanded)

parses = [p for p in atis_parser_expanded.parse(test_sentence_2)]
for parse in parses:
  parse.pretty_print()
```

```
                                                               S
                        _______________________________________|______________________
                       |                                                          NP_FL
IGHT
                       |                                                              |
                       |                                                          NOM_F
LIGHT
                       |                                        ______________________|
__________
                       |                                       |
NOM_FLIGHT
                       |                                       |
|
                   PREIGNORE                                   |
N_FLIGHT
        _______________|____________                           |            ____________
_________|_________
       |                        PREIGNORE                      |           |
PP
       |                ____________|____________              |           |
|
       |               |                     PREIGNORE        ADJ       N_FLIGHT
PP_PLACE
       |               |                         |             |           |
_________|_________
PREIGNORESYMBOL PREIGNORESYMBOL           PREIGNORESYMBOL ADJ_AIRLINE TERM_FLIGHT
```

```
PREIGNORESYMBOL PREIGNORESYMBOL          PREIGNORESYMBOL ADJ_AIRLINE TERM_FLIGHT
P_PLACE            TERM_PLACE
       |                |                          |              |            |
|                  |
      show             me                         the          united      flights
from               boston

                                                    S

                          ______________________________|___________

                         |                                     NP_FLIGHT

                         |                       ___________________|__________

                         |                      |                          NOM_FLIGHT

                         |                      |        ______________________|__________

                         |                      |        |                                   NOM_FL
IGHT
                         |                      |        |                                       |

                         |                      |        |                                    N_FLI
GHT
                         |                      |        |            _____________________|_
________
                     PREIGNORE                  |        |            |
PP
        ____________|____________               |        |            |
|
       |                    PREIGNORE           |       ADJ        N_FLIGHT
PP_PLACE
       |                         |              |        |            |                           _
________|_________
PREIGNORESYMBOL            PREIGNORESYMBOL DET ADJ_AIRLINE TERM_FLIGHT                     P_PLAC
E            TERM_PLACE
       |                         |              |        |            |                           |
|
      show                      me             the     united      flights                      from
boston
```

In [29]:

```python
test_sentence_3 = tokenize("what is the most expensive one way flight from boston to atla
nta on american airlines")
parsess = [p for p in atis_parser_expanded.parse(test_sentence_3)]
for parse in parsess:
  parse.pretty_print()
```

Once you're done adding to the grammar, to check your grammar, we'll compute the grammar's coverage of the ATIS training corpus as before. This grammar should be expected to cover about half of the sentences in the first 50 sentences, and a third of the entire training corpus.

In [27]:

```python
coverage(lambda sent: 0 < len(list(atis_parser_expanded.parse(sent))),  # trick for turni
ng parser into recognizer
         training_corpus, n=50)
```

```
100%|██████████| 50/50 [00:00<00:00, 271.25it/s]
```

Out[27]:

```
0.44
```

CFG recognition via the CKY algorithm

Now we turn to implementing recognizers and parsers using the CKY algorithm. We start with a recognizer,

which should return `True` or `False` if a grammar does or does not admit a sentence as grammatical.

Converting the grammar to CNF for use by the CKY algorithm

The CKY algorithm requires the grammar to be in Chomsky normal form (CNF). That is, only rules of the forms

$$A \rightarrow B\,C$$
$$A \rightarrow a$$

are allowed, where A, B, C are nonterminals and a is a terminal symbol.

However, in some downstream applications (such as the next project segment) we want to use grammar rules of more general forms, such as $A \rightarrow B\,C\,D$. Indeed, the ATIS grammar you've been working on makes use of the additional expressivity beyond CNF.

To satisfy both of these constraints, we will convert the grammar to CNF, parse using CKY, and then convert the returned parse trees back to the form of the original grammar. We provide some useful functions for performing these transformations in the file `scripts/transform.py`, already loaded above and imported as `xform`.

To convert a grammar to CNF:

```
cnf_grammar, cnf_grammar_wunaries = xform.get_cnf_grammar(grammar)
```

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

```
xform.un_cnf(tree, cnf_grammar_wunaries)
```

> We pass into `un_cnf` a version of the grammar before removing unary nonterminal productions, `cnf_grammar_wunaries`. The `cnf_grammar_wunaries` is returened as the second part of the returned value of `get_cnf_grammar` for just this purpose.

In [30]:

```
atis_grammar_cnf, atis_grammar_wunaries = xform.get_cnf_grammar(atis_grammar_expanded)
assert(atis_grammar_cnf.is_chomsky_normal_form())
```

In the next sections, you'll write your own recognizers and parsers based on the CKY algorithm that can operate on this grammar.

Part 2: Implement a CKY recognizer

Implement a *recognizer* using the CKY algorithm to determine if a sentence `tokens` is parsable. The labs and J&M Chapter 13, both of which provide appropriaste pseudo-code for CKY, should be useful references here.

> **Hint:** Recall that you can get the production rules of a grammar using `grammar.productions()`.
>
> Throughtout this project segment, you should use `grammar.start()` to get the special start symbol from the grammar instead of using `S`, since some grammar uses a different start symbol, such as `TOP`.

In [78]:

```
def cky_recognize(grammar, tokens):
    """
    Returns True if and only if the list of tokens `tokens` is admitted
    by the `grammar`.
    Implements the CKY algorithm, and therefore assumes `grammar` is in
    Chomsky normal form.
    """
    # Create an empty chart for storing intermediate results
    chart = [[set() for _ in range(len(tokens) + 1)] for _ in range(len(tokens) + 1)]
```

```
    # Fill in the diagonal of the chart with nonterminals that produce the terminals
    for j in range(1, len(tokens) + 1):
        for prod in grammar.productions(rhs=tokens[j-1]):
            chart[j-1][j].add(prod.lhs())

    # Apply the CKY algorithm
    for length in range(2, len(tokens) + 1):
        for start in range(len(tokens) - length + 1):
            end = start + length
            for mid in range(start + 1, end):
                for prod in grammar.productions():
                    if len(prod.rhs()) == 2:
                        B, C = prod.rhs()
                        for B_item in chart[start][mid]:
                            for C_item in chart[mid][end]:
                                if B == B_item and C == C_item:
                                    chart[start][end].add(prod.lhs())

    # Check if the start symbol is in the final cell of the chart
    return grammar.start() in chart[0][len(tokens)] and len(chart[0][len(tokens)]) > 0
```

You can test your recognizer on a few examples, both grammatical and ungrammatical, as below.

In [79]:

```
test_sentences = ["show me flights from boston",
                  "show me united flights before noon",
                  "are there any twa flights available tomorrow",
                  "show me flights united are there any"]

for sentence in test_sentences:
  recognized = "+" if cky_recognize(atis_grammar_cnf, tokenize(sentence)) else "-"
  print(f"{recognized:5}{sentence}")
```

```
+    show me flights from boston
+    show me united flights before noon
-    are there any twa flights available tomorrow
-    show me flights united are there any
```

You can also verify that the CKY recognizer verifies the same coverage as the NLTK parser.

In [33]:

```
coverage(lambda sent: cky_recognize(atis_grammar_cnf, sent),
         training_corpus, n=50)
```

```
100%|██████████| 50/50 [00:16<00:00,  3.02it/s]
```

Out[33]:

```
0.44
```

Part 3: Implement a CKY parser

In part 2, you implemented a context-free grammar recognizer. Next, you'll implement a *parser*.

Implement the CKY algorithm for parsing with CFGs as a function `cky_parse`, which takes a grammar and a list of tokens and returns a single parse of the tokens as specified by the grammar, or `None` if there are no parses. You should only need to add a few lines of code to your CKY recognizer to achieve this, to implement the necessary back-pointers. The function should return an NLTK tree, which can be constructed using `Tree.fromstring`.

A tree string will be like this example:

```
"(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
```

> **Hint: You may want to extract from a `Nonterminal` its corresponding string. The `Nonterminal.__str__` method or f-string `f'{Nonterminal}'` accomplishes this.**

In [157]:

```
from nltk import CFG, Tree

def cky_parse(grammar, tokens):
    # Create an empty chart for storing intermediate results
    chart = [[set() for _ in range(len(tokens) + 1)] for _ in range(len(tokens) + 1)]
    back = [[{} for _ in range(len(tokens) + 1)] for _ in range(len(tokens) + 1)]

    # Fill in the diagonal of the chart with nonterminals that produce the terminals
    for j in range(1, len(tokens) + 1):
        for prod in grammar.productions(rhs=tokens[j-1]):
            chart[j-1][j].add(prod.lhs())
            back[j-1][j][prod.lhs()] = (tokens[j-1])

    # Apply the CKY algorithm
    for length in range(2, len(tokens) + 1):
        for start in range(len(tokens) - length + 1):
            end = start + length
            for mid in range(start + 1, end):
                for prod in grammar.productions():
                    if len(prod.rhs()) == 2:
                        B, C = prod.rhs()
                        for B_item in chart[start][mid]:
                            for C_item in chart[mid][end]:
                                if B == B_item and C == C_item:
                                    chart[start][end].add(prod.lhs())
                                    back[start][end][prod.lhs()] = (mid, B, C)

    # If parse is successful, build and return parse tree
    if grammar.start() in chart[0][len(tokens)]:
        return build_tree(back, 0, len(tokens), grammar.start())
    else:
        return None

def build_tree(back, start, end, nonterminal):
    if isinstance(back[start][end][nonterminal], str):
        return Tree(nonterminal.symbol(), [back[start][end][nonterminal]])
    else:
        mid, B, C = back[start][end][nonterminal]
        left_tree = build_tree(back, start, mid, B)
        right_tree = build_tree(back, mid, end, C)
        return Tree(nonterminal.symbol(), [left_tree, right_tree])
```

You can test your code on the test sentences provided above:

In [158]:

```
for sentence in test_sentences:
  tree = cky_parse(atis_grammar_cnf, tokenize(sentence))

  if not tree:
    print(f"failed to parse: {sentence}")
  else:
```

```
    xform.un_cnf(tree, atis_grammar_wunaries)
    tree.pretty_print()
```

```
                                               S
                     __________________________|__________
                    |                                      NP_FLIGHT
                    |                                          |
                    |                                      NOM_FLIGHT
                    |                                          |
                    |                                       N_FLIGHT
                    |                           _______________|__________
                PREIGNORE                      |                          PP
        ____________|___________               |                          |
       |                    PREIGNORE       N_FLIGHT                  PP_PLACE
       |                        |              |               _________|__________
PREIGNORESYMBOL          PREIGNORESYMBOL TERM_FLIGHT   P_PLACE             TERM_PLACE
       |                        |              |           |                    |
      show                      me          flights      from                 boston
                                               S

                     __________________________|______________________

                    |                                                  NP_FLIGHT

                    |                                                      |

                    |                                                  NOM_FLIGHT

                    |                           ___________________________|___________

                    |                          |                                      NOM_FLIGHT

                    |                          |                                          |

                    |                          |                                       N_FLIGHT

                    |                          |            ______________________________|______
___
                    |                          |           |
PP
                    |                          |           |
|
                PREIGNORE                      |           |
PP_TIME
        ____________|___________               |           |                          ______
___|_________
       |                    PREIGNORE         ADJ       N_FLIGHT                      |
NP_TIME
       |                        |              |           |                          |
|
PREIGNORESYMBOL          PREIGNORESYMBOL ADJ_AIRLINE TERM_FLIGHT                   P_TIME
TERM_TIME
       |                        |              |           |                          |
|
      show                      me           united     flights                     before
noon

failed to parse: are there any twa flights available tomorrow
failed to parse: show me flights united are there any
```

You can also compare against the built-in NLTK parser that we constructed above:

In [159]:

```
for sentence in test_sentences:
  refparses = [p for p in atis_parser_expanded.parse(tokenize(sentence))]
  predparse = cky_parse(atis_grammar_cnf, tokenize(sentence))
  if predparse:
    xform.un_cnf(predparse, atis_grammar_wunaries)
  """
```

```
  print('Reference parses:')
  for reftree in refparses:
    print(reftree)

  print('\nPredicted parse:')
  print(predparse)
  """

  if (not predparse and len(refparses) == 0) or predparse in refparses:
    print("\nSUCCESS!")
  else:
    print("\nOops. No match.")
```

```
SUCCESS!

SUCCESS!

SUCCESS!

SUCCESS!
```

Again, we test the coverage as a way of verifying that your parser works consistently with the recognizer and the NLTK parser.

In [160]:

```
coverage(lambda sent: cky_parse(atis_grammar_cnf, sent),
         training_corpus, n=50)
```

```
100%|██████████| 50/50 [00:15<00:00,  3.21it/s]
```

Out[160]:

```
0.44
```

Probabilistic CFG parsing via the CKY algorithm

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 probabilistic context-free grammars (PCFG). In this part of the assignment, you will construct a PCFG from training data, parse using a probabilistic version of CKY, and evaluate the quality of the resulting parses against gold trees.

Part 4: PCFG construction

Compared to CFGs, PCFGs need to assign probabilities to grammar rules. For this goal, you'll write a function `pcfg_from_trees` that takes a list of strings describing a corpus of trees and returns an NLTK PCFG trained on that set of trees.

> **We expect you to implement `pcfg_from_trees` directly. You should not use the [`induce_pcfg`] function in implementing your solution.**

We want the PCFG to be in CNF format because the probabilistic version of CKY that you'll implement next 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 from them. To accomplish this, you should use the `treetransforms` package from `nltk`, which includes functions for converting to and from CNF. In particular, you'll want to make use of `treetransforms.collapse_unary` followed by `treetransforms.chomsky_normal_form` to convert a tree to its binarized version. You can then get the counts for all of the productions used in the trees, and then normalize them to probabilities so that the probabilities of all rules with the same left-hand side sum to 1.

We'll use the `pcfg_from_trees` function that you define later for parsing.

> **To convert an `nltk.Tree` object `t` to CNF, you can use the below code. Note that it's different**

> from the `xform` functions we used before as we are converting *trees*, not *grammars*.
>
> ```
>     treetransforms.collapse_unary(t, collapsePOS=True)
>     treetransforms.chomsky_normal_form(t) # After this the tree will be
>  in CNF
> ```
>
> **To construct a PCFG with a given start state and set of productions, see `nltk.grammar.PCFG`.**

In [166]:

```python
import nltk
from nltk import Nonterminal, ProbabilisticProduction, PCFG
from nltk import treetransforms

def rule_counter(sentence_list):
    out = {}
    for sentence in sentence_list:
        for production in nltk.Tree.fromstring(sentence).productions():
            key = nltk.grammar.Production(production.lhs(), production.rhs())
            if key in out:
                out[key] += 1
            else:
                out[key] = 1
    return out

def lhs_counter(sentence_list):
    out = {}
    for sentence in sentence_list:
        for production in nltk.Tree.fromstring(sentence).productions():
            key = production.lhs()
            if key in out:
                out[key] += 1
            else:
                out[key] = 1
    return out

def rule_probs(sentence_list):
    rule_dict = rule_counter(sentence_list)
    lhs_dict = lhs_counter(sentence_list)
    out = rule_dict
    for key, value in out.items():
        out[key] /= lhs_dict[key.lhs()]
    return out

def pcfg_from_trees(trees, start=Nonterminal('TOP')):
    # Convert trees to Chomsky Normal Form
    cnf_trees = []
    for tree_str in trees:
        tree = nltk.Tree.fromstring(tree_str)
        treetransforms.collapse_unary(tree, collapsePOS=True)
        treetransforms.chomsky_normal_form(tree)
        cnf_trees.append(tree.pformat(margin=float('inf')))

    # Calculate rule probabilities
    rule_probabilities = rule_probs(cnf_trees)

    # Create probabilistic productions
    prob_productions = [
        ProbabilisticProduction(p.lhs(), p.rhs(), prob=rule_probabilities[p])
        for p in rule_probabilities
    ]

    # Create and return PCFG
    return PCFG(start, prob_productions)
```

We can now train a PCFG on the *train* split `train.trees` that we downloaded in the setup at the start of the notebook.

In [167]:

```
with open('data/train.trees') as file:
  ## Convert the probabilistic productions to an NLTK probabilistic CFG.
  pgrammar = pcfg_from_trees(file.readlines())

## Verify that the grammar is in CNF
assert(pgrammar.is_chomsky_normal_form())
```

Part 5: Probabilistic CKY parsing

Finally, we are ready to implement probabilistic CKY parsing under PCFGs. Adapt the CKY parser from Part 3 to return the most likely parse and its log probability (base 2) given a PCFG. Note that to avoid underflows we want to work in the log space.

> **Hint:** `production.logprob()` **will return the log probability of a production rule** `production`**.**

In [233]:

```
import numpy as np

def build_tree(i, j, nonterminal, back):
    if (j - i) == 1:
        return nltk.Tree(str(nonterminal), [back[i][j][nonterminal]])
    else:
        split, B, C = back[i][j][nonterminal]
        return nltk.Tree(str(nonterminal), [build_tree(i, split, B, back), build_tree(sp
lit, j, C, back)])

def cky_parse_probabilistic( grammar,tokens):
    """ Returns the most probable parse tree of `tokens` and its log probability
    given a probabilistic CFG `grammar`.

    Implements the CKY algorithm, and therefore assumes `grammar` is in
    Chomsky normal form.
    """

    assert(grammar.is_chomsky_normal_form())
    num_tokens = len(tokens)

    # Initialize data structures
    score = [[{} for j in range(num_tokens + 1)] for i in range(num_tokens)]
    back = [[{} for j in range(num_tokens + 1)] for i in range(num_tokens)]

    # Fill in base case probabilities
    for j in range(1, num_tokens + 1):
        for production in grammar.productions(rhs=tokens[j-1]):
            score[j-1][j][production.lhs()] = np.log2(production.prob())
            back[j-1][j][production.lhs()] = tokens[j-1]

    # Fill in rest of table
    for length in range(2, num_tokens + 1):
        for i in range(num_tokens - length + 1):
            j = i + length
            for split in range(i + 1, j):
                for production in grammar.productions():
                    if len(production.rhs()) == 2:
                        B, C = production.rhs()
                        if B in score[i][split] and C in score[split][j]:
                            new_prob = np.log2(production.prob()) + score[i][split][B] +
score[split][j][C]
                            if production.lhs() not in score[i][j] or new_prob > score[i
][j][production.lhs()]:
                                score[i][j][production.lhs()] = new_prob
                                back[i][j][production.lhs()] = (split, B, C)

    # Return most probable parse and its score
```

```
    if grammar.start() not in back[0][num_tokens]:
        return None, -np.inf

    return build_tree(0, num_tokens, grammar.start(), back), score[0][num_tokens][gramma
r.start()]
```

As an aid in debugging, you may want to start by testing your implementation of probabilistic CKY on a much smaller grammar than the one you trained from the ATIS corpus. Here's a little grammar that you can play with.

> **Hint: By "play with", we mean that you can change the gramamr to try out the behavior of your parser on different test grammars, including ambiguous cases.**

In [222]:

```
grammar = PCFG.fromstring("""
  S -> NP VP [1.0]
  VP -> V NP [1.0]
  PP -> P NP [1.0]
  NP -> 'sam' [.3]
  NP -> 'ham' [.7]
  V -> 'likes' [1.0]
  """)
print(grammar.is_chomsky_normal_form())
```

```
True
```

In [234]:

```
tree, logprob = cky_parse_probabilistic(grammar, tokenize('sam likes ham'))
print(tree)
tree.pretty_print()
print(f"logprob: {logprob:4.3g} | probability: {2**logprob:4.3g}")
```

```
(S (NP sam) (VP (V likes) (NP ham)))
      S
  ____|____
 |         VP
 |     ____|___
 NP   V        NP
 |    |        |
sam likes     ham

logprob: -2.25 | probability: 0.21
```

In [235]:

```
# We don't use our tokenizer because the gold trees do not lowercase tokens
sent = "Flights from Cleveland to Kansas City .".split()
tree, logprob = cky_parse_probabilistic(pgrammar, sent)
tree.un_chomsky_normal_form()
tree.pretty_print()
print(f"logprob: {logprob:4.3g} | probability: {2**logprob:4.3g}")
```

```
                           TOP
                      ______|___________________
                    FRAG                        |
                     |                          |
                     NP                         |
    _________________|___________               |
   |          PP                 PP             |
   |      ____|______        ____|_____         |
   NP    |           NP     |          NP       |
   |     |           |      |     _____|___     |
  NNS    IN         NNP     TO  NNP       NNP  PUNC
   |     |           |      |    |         |    |
Flights from     Cleveland  to Kansas     City  .

logprob:  -27 | probability: 7.42e-09
```

Evaluation of the grammar

There are a number of ways to evaluate parsing algorithms. In this project segment, you will use the ["industry-standard" `evalb` implementation](#) for computing constituent precision, recall, and F1 scores. We downloaded `evalb` during setup.

We read in the test data...

In [236]:

```
with open('data/test.trees') as file:
  test_trees = [Tree.fromstring(line.strip()) for line in file.readlines()]

test_sents = [tree.leaves() for tree in test_trees]
```

...and parse the test sentences using your probabilistic CKY implementation, writing the output trees to a file.

In [237]:

```
trees_out = []
for sent in tqdm(test_sents):
  tree, prob = cky_parse_probabilistic(pgrammar, sent)
  if tree is not None:
    tree.un_chomsky_normal_form()
    trees_out.append(tree.pformat(margin=9999999999))
  else:
    trees_out.append('()')

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

```
100%|██████████| 58/58 [00:02<00:00, 21.98it/s]
```

Now we can compare the predicted trees to the ground truth trees, using `evalb`. You should expect to achieve F1 of about 0.83.

In [238]:

```
shell("python scripts/evalb.py data/outp.trees data/test.trees")
```

```
data/outp.trees 345 brackets
data/test.trees 471 brackets
matching 339 brackets
precision 0.9826086956521739
recall 0.7197452229299363
F1 0.8308823529411764
```

Debrief

Question: **We're interested in any thoughts you have about this project segment so that we can improve it for later years, and to inform later segments for this year. Please list any issues that arose or comments you have to improve the project segment. Useful things to comment on might include the following:**

- **Was the project segment clear or unclear? Which portions?**
- **Were the readings appropriate background for the project segment?**
- **Are there additions or changes you think would make the project segment better?**

```
BEGIN QUESTION
name: open_response_debrief
manual: true
```

but you should comment on whatever aspects you found especially positive or negative.

Type your answer here, replacing this text

Type your answer here, replacing this text.

Instructions for submission of the project segment

This project segment should be submitted to Gradescope at https://rebrand.ly/project3-submit-code and https://rebrand.ly/project3-submit-pdf, which will be made available some time before the due date.

Project segment notebooks are manually graded, not autograded using otter as labs are. (Otter is used within project segment notebooks to synchronize distribution and solution code however.) **We will not run your notebook before grading it.** Instead, we ask that you submit the already freshly run notebook. The best method is to "restart kernel and run all cells", allowing time for all cells to be run to completion. You should submit your code to Gradescope at the code submission assignment at https://rebrand.ly/project3-submit-code. Make sure that you are also submitting your `data/grammar` file as part of your solution code as well.

We also request that you **submit a PDF of the freshly run notebook**. The simplest method is to use "Export notebook to PDF", which will render the notebook to PDF via LaTeX. If that doesn't work, the method that seems to be most reliable is to export the notebook as HTML (if you are using Jupyter Notebook, you can do so using `File -> Print Preview`), open the HTML in a browser, and print it to a file. Then make sure to add the file to your git commit. Please name the file the same name as this notebook, but with a `.pdf` extension. (Conveniently, the methods just described will use that name by default.) You can then perform a git commit and push and submit the commit to Gradescope at https://rebrand.ly/project3-submit-pdf.

End of project segment 3