CS187 Lab 3-2: Context-free parsing

September 7, 2024

```
[ ]: # 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/cs187-2021/lab3-2.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

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

In this lab, you'll carry out a simple context-free recognition and parsing algorithm, the CKY algorithm independently discovered by John Cocke, Tadao Kasami, and Daniel Younger.

By carrying out and understanding this lab, you should be able to

- Distinguish recognition from parsing,
- Understand the string position representation of constituents, and
- Follow recognition algorithms for context-free grammars.

New bits of Python used for the first time in the *solution set* for this lab, and which you may therefore find useful:

- pandas.DataFrame.iloc
- set.add

Preparation – Loading packages

[ ]:
```
from collections import defaultdict
import functools
import numpy as np
import nltk
import pandas as pd
```

1 Introduction

In the previous lab, you worked with context-free grammars (CFGs), and practiced defining grammars and parse trees.

This lab will focus on the Cocke-Kasami-Younger (CKY) parsing algorithm. The input to the algorithm is a grammar in Chomsky Normal Form (CNF) and a sentence; the output of the algorithm is the set of parse trees that yield the given sentence according to the grammar (the empty set if there are no such parse trees).

We will first focus on the CKY *recognition* algorithm, that only outputs whether the given sentence can be parsed or not. Then, we will see how to slightly modify this algorithm (by adding back-pointers) to also generate the parse trees themselves.

1.1 The sample grammar

Recall the simple arithmetic grammar that you wrote in the previous lab. In this lab, you'll use a simplified version, containing fewer operators and numbers.

[ ]:
```
arithmetic_grammar = nltk.CFG.fromstring(
    """
    S -> NUM | S OP S
    OP -> ADD | SUB | MULT

    NUM -> 'zero' | 'one' | 'two' | 'three'
    ADD -> 'plus'
    SUB -> 'minus'
    MULT -> 'times'
    """
)
```

The CKY algorithm requires grammars in Chomsky Normal Form (CNF). Here is a CNF-converted version of the grammar, which makes use of an addirtional nonterminal `SOP`, which derives prefixes of `S` missing an `S` at the right edge, strings like, for instance, `one plus` or `one plus two times`.

[ ]:
```
cnf_arithmetic_grammar = nltk.CFG.fromstring(
    """
    S -> 'zero' | 'one' | 'two' | 'three'
    S -> SOP S

    SOP -> S ADD | S SUB | S MULT

    ADD -> 'plus'
    SUB -> 'minus'
    MULT -> 'times'
    """
)

print(cnf_arithmetic_grammar)
```

We can verify that the grammar is in CNF, as required by the CKY algorithm.

[ ]:
```
cnf_arithmetic_grammar.is_chomsky_normal_form()
```

Given the sentence "*one plus two times three*", what would be its parse tree according to the grammar?

Without any specific handling of arithmetic operator precedence, this "sentence" can be either parsed as:

- `(one plus two) times three`

or:

- `one plus (two times three)`

1.2 Parsing a sentence

Let's see how a parser parses this sentence:

[ ]:
```
sentence = "one plus two times three"
words = sentence.split()

parser = nltk.parse.BottomUpChartParser(cnf_arithmetic_grammar)
for tree in parser.parse(words):
    tree.pretty_print()
```

As you can see, `nltk`'s parser finds the two possible parses. In this lab, you will manually parse this sentence using the CKY algorithm.

2 The CKY recognition algorithm

The CKY algorithm is a simple *dynamic programming* algorithm. Dynamic programming algorithms work by solving every subproblem of a given problem, in an order such that larger subproblems can merely recombine solutions of smaller subproblems already solved. In the case of CNF parsing, the subproblems are the parsing of *every substring* as *every nonterminal.* In the end, of course, all we care about is whether the *entire string* can be parsed as *the start nonterminal.* But by calculating all of the subproblems in order from the shortest to the longest substrings, we can acquire the information we need in the end.

> You may recall that Viterbi's algorithm is also a dynamic programming algorithm, solving all subproblems of finding, for each prefix of a sequence and each state, the maximum probability of the prefix where the last word was emitted by the state.

We start by representing the string to be parsed in such a way that we can represent each of its substrings compactly. Consider the string `one plus two times three`. We can think of the positions between the words as being numbered from zero to the length of the string (5), as depicted in the figure at right. Then any substring can be characterized by the string position to its left and the string position to its right, which we'll call a *span.* For instance, the substring `one plus` corresponds to the span 0–2 and `two times three` to the span 2–5. Adjacent spans (like these two) share a string position, in this case the string position 2, which corresponds to the "split point" that divides the spans. Two adjacent spans can be combined to form a longer one; the full span is between the left string position of the left span and the right string position of the right span, 0–5.

Some substrings are generated by the grammar from particular nonterminals. For instance, in the example at right, the substring in the span 0–2 is generated by the grammar as an `SOP`, and 2–5 as an `S`. We'll call a span generable by a particular nonterminal a *potential constituent.*

The CKY algorithm proceeds by generating every potential subconstituent of the string to be parsed. It does so by filling in a table `T` indexed by two string positions (a span, specifying a substring) whose entries are sets of nonterminals that can generate the substring between the string positions. Thus, the algorithm in parsing the string `one plus two times three` will place `SOP` in the table at entry `T[0, 2]` and `S` at entry `T[2, 5]` (among other entries). It follows from these two entries, plus the existence of a rule `S -> SOP S` in the grammar, that there should also be an entry `S` in `T[0, 5]`. By filling in these table entries in a particular order, making use of entries previously filled out, the full set of potential subconstituents can be determined, and hence, the particular question of whether `S` covers the entire string can be answered simply by inspecting `T[0, N]`.

Here is the CKY algorithm in pseudo-code:

```
 1.  define cky-parse(string = w_1, ..., w_N, grammar):
 2.      for j in [1..N]:                     # each end string position

             # handle unary rules, of the form A -> w
 3.         for all A where A --> w_j in grammar:
 4.             add A to T[j-1, j]

            # handle binary rules, of the form A -> B C
 5.         for length in [2..j]:            # each subconstituent length
 6.             i := j - length              # start string position
 7.             for split in [i+1..j-1]      # each split point
 8.                 for all A where
 9.                         A -> B C is a rule in the grammar
10.                         and B in T[i, split]
11.                         and C in T[split, j]:
12.                     add A to T[i, j]
13.     if S in T[0, N] then parsed else failed
```

The idea is that we parse the sentence bottom-up, where in every step we check whether a given subsequence of the input can be derived from any single nonterminal. If we perform this in the right order (as above), at every step we already know whether each of the possible splits of the subsequence creates valid subparses.

In this lab, you'll carry out the CKY algorithm manually.

> Note that in the pseudo-code above, the CKY algorithm runs column by column, and in each column from bottom to top row, while in the lecture you saw that it calculates the diagonals from bottom to top. This difference doesn't matter at all since the content of each cell depends only on the cells to its left and the ones beneath.
>
> A detail concerning implementing the algorithm in Python, with its zero-based indexing: Zero-based indexing works quite naturally for the table `T`, since the string positions start with zero. However, the input sentence itself uses one-based indexing, that is, the first word in the input sentence is denoted as `w1` in the pseudo-code. To make it easier to follow the algorithm, we can match our indices to the algorithm's by adding a dummy symbol at the start of the sentence.

[ ]:
```python
words = [''] + sentence.split()
words
```

For the table, we'll use a `pandas` dataframe, primarily for its readable output. (We don't recommend using this data structure when implementing the CKY algorithm variants in project segment 3.)

- Entries that contain `'---'` do not need to be modified.
- To denote entries that contain no non-terminals, we will use `set()` as their content (an empty set).
- To denote entries that contain one or more nonterminals `A`, `B`, and `C`, we will use, e.g., `set(['A'])` or `set(['A','B','C'])`

[ ]:
```python
data = [
        ['---',  set(),  set(),  set(),  set(),  set()],
        ['---',  '---',  set(),  set(),  set(),  set()],
        ['---',  '---',  '---',  set(),  set(),  set()],
        ['---',  '---',  '---',  '---',  set(),  set()],
        ['---',  '---',  '---',  '---',  '---',  set()],
        ['---',  '---',  '---',  '---',  '---',  '---'],
]

table = pd.DataFrame(data, columns=words, index=list('012345'))
table.columns = pd.MultiIndex.from_arrays([table.columns] + [list('012345')])
table
```

We will now manually track the execution of the algorithm. The first step in the algorithm is a loop for `j` starting at `1`. Then,

```
 3.           for all A where A --> w_j in grammar:
 4.               add A to T[j-1, j]
```

In the case at hand `w_j` (that is `w_1`, that is `words[1]`) is `one`. And sure enough there is a rule in the grammar of the required form, namely, `S -> 'one'`. So we must update the table accordingly.

In the next cell, update the relevant entry in the table.

> You can use the `pandas` method `table.iloc[row, col]` for indexing and the `set` method `add` to add a nonterminal to a set.

[ ]:
```
#TODO -- update the table for the first step of j=1
# 3.           for all A where A --> wj in grammar:
# 4.               add A to T[j-1, j]
...
```

[ ]:
```
grader.check("table_addition_1")
```

Let's examine the table to make sure it worked.

[ ]:
```
table
```

Now the next line in the algorithm specifies a loop for all lengths from 2 to `j`. But `j` is `1`, so the loop is vacuous.

> Why lengths starting at 2? Because for shorter strings, there's no way to split the string into two non-empty parts, and in CNF grammars, there are no empty constituents.

We're back to lines 3-4 with `j` now having the value 2. Update the table accordingly in the next cell.

[ ]:
```
#TODO -- update the table
...
```

[ ]:
```
grader.check("table_addition_2")
```

[ ]:
```
table
```

Again, we're at line 5, selecting a length from the range [2..2]. There's only one such length, 2, so the start string position `i` is `j-2`, that is `0`. For strings between string positions 0 and 2, there is only one possible split point, namely 1. At line 7, then, there will be only a single iteration.

Lines 8-12 have us find all rules in the grammar of the form $A \to B\,C$ such that `B` is in `T[i, split]` and `C` is in `T[split, j]`, and add such `A` values to the table.

Update the table accordingly in the next cell.

```
[ ]: #TODO -- update the table
     ...
```

```
[ ]: grader.check("table_addition_3")
```

```
[ ]: table
```

Both loops that iterate over i and over *split* had a single iteration, so we can continue with $j = 3$.

Complete the table according to the algorithm, starting from $j = 3$. Remember, it is okay that a given entry will not have any nonterminals. In that case, the entry value will remain the empty set.

```
[ ]: #TODO - update the table
     ...
```

```
[ ]: grader.check("table_addition_rest")
```

```
[ ]: table
```

The entry that represents the entire input resides at position `[0,5]`. Since our sentence should be parsable by the grammar as the start symbol `S`, the following expression should hold if and only if the string is generated by the grammar.

```
[ ]: 'S' in table.iloc[0,5]
```

3 The CKY parsing algorithm

The algorithm that we executed so far is a *recognizer*, not a parser – if the algorithm ends with the start symbol in `table[0,n]` – it means that the input is parsable, but it doesn't directly provide information about the input's parse trees.

However, we want to know not only *whether* the string is parsable, but *how*. We'd like the parse trees themselves. The necessary modification to the algorithm is to keep in the table entries not only a nonterminal that covers the substring, but the `split` value and production that was used as well.

We will define another table called `back` to keep that information. A `back` table value will be a mapping from nonterminals into a set of values for `split` and the right-hand side of the rule that was used. We use a set because there may be several such split and production values. We initialize the entries to a dictionary mapping to empty sets.

```
[ ]: def empty():
         return defaultdict(set)
```

```
back_data = [
        ['---',  empty(),  empty(),  empty(),  empty(),  empty()],
        ['---',  '---',    empty(),  empty(),  empty(),  empty()],
        ['---',  '---',    '---',    empty(),  empty(),  empty()],
        ['---',  '---',    '---',    '---',    empty(),  empty()],
        ['---',  '---',    '---',    '---',    '---',    empty()],
        ['---',  '---',    '---',    '---',    '---',    '---'],
]

back = pd.DataFrame(back_data, columns=words, index=list('012345'))
back.columns = pd.MultiIndex.from_arrays([back.columns] + [list('012345')])
back
```

Recall (lines 8-12) that we update the `table` every time we find a rule $A \longrightarrow BC$ and `split` such that

```
 9.                            A -> B C in grammar
10.                            and B in T[i, split]
11.                            and C in T[split, j]:
```

In addition to updating `table`, we can also then add to `back` the information about `split` and the rule `A -> B C`. We augment the algorithm with a new line 12.5 like this:

```
12.                        add A to T[i, j]
12.5.                      add <split, B, C> to back[i, j, A]
```

For example, for `i = 0`, `split = 1`, and `j = 2` and the found rule `SOP -> S ADD`:

```
[ ]: back.iloc[0,2]['SOP'].add((1,'S','ADD'))
     back
```

Copy the statements in which you assigned a non-empty value into `table[i,j]` into the code cell below, starting from `j=3`. Then, instead of only assigning a non-terminal into `table[i,j]`, also fill the corresponding value of `back[i,j]`.

(You can either re-run the assignments into `table[i,j]`, or comment them and run only the assignments into `back[i,j]`.)

The dictionary in the final entry, `back[0,5]`, should map `S` onto a set with two elements because there are two possible ways to parse the input.

```
[ ]: #TODO -- update the back table
     ...
```

```
[ ]: grader.check("table_addition_rest_back")
```

```
[ ]: back
```

Now, instead of only telling whether the input sentence is parsable or not, we can also construct the parse trees!

The idea is as follows: we start from `back[0, 5]` with the entry for `S`. For each value in the set in that entry, we construct a parse tree whose root is `S`.

Suppose that `back[0, 5]` contains: `{'S': {(k1, A1, B1), (k2, A2, B2)}}`. We'll reconstruct parse trees for each of the two options. Start with `(k1, A1, B1)`. We recursively construct all possible parse trees based on the `A1` nonterminal covering `[0, k1]` and all possible parse trees based on `B1` covering `[k1, 5]`. Then for each of the `A1` and `B1` trees, we construct a tree (using parentheses notation) of the form `(S (A1 ...) (B1 ...))`. We do the same for the other option `(k2, A2, B2)`. In this way, we'll generate all parse trees for the string.

As it turns out, for the particular string we've been using, there is only one parse tree generable by each of the two options, so there are only two parse trees for the sentence.

Start from `back[0, 5]`, and construct the two possible parse trees by using this method. Enter the two parses as `tree1` and `tree2` in the cell below.

> Note that as a convention we don't put quotation marks around terminals when we use `nltk.Tree.fromstring`. For example, if the tree has root `S` and two terminals `one` and `two`, then we construct the parse using `tree = nltk.Tree.fromstring("(S one two)")` instead of `tree = nltk.Tree.fromstring("(S 'one' 'two')")`.

[ ]:
```
#TODO -- complete the strings using bracket-notation
tree1 = nltk.Tree.fromstring(
    ...
)
tree2 = nltk.Tree.fromstring(
    ...
)
```

[ ]:
```
grader.check("parse_trees_construct")
```

Now let's check your parse trees. Are they the same as the parse trees returned by the NLTK parser?

[ ]:
```
tree1.pretty_print()
tree2.pretty_print()
```

Did you find the same trees as the NLTK parser did at the start of the lab?

These trees were parsed on the basis of the rules in `cnf_arithmetic_grammar`, a CNF version of `arithmetic_grammar` above, which replaced the rule `S -> S OP S` with rules

- `S -> SOP S`
- `SOP -> S ADD`
- `SOP -> S SUB`
- `SOP -> S MULT`

A configuration like

`(S (S ...x...) (OP (ADD ...y...)) (S ...z...))`

from the original grammar thus corresponds to a configuration like

```
(S (SOP (S ...x...) (ADD ...y...)) (S ...z...))
```

from the CNF grammar.

You should thus be able to reconstruct parses for the original grammar by "undoing" this change, replacing the latter configuration with the former. Reconstruct the two parse trees that the original grammar would have generated by undoing this change on `tree1` and `tree2`, and call them `tree1_reconstructed` and `tree2_reconstructed`.

```
[ ]: #TODO -- Reconstruct the trees by providing strings using bracket-notation
     tree1_reconstructed = nltk.Tree.fromstring(
         ...
     )
     tree2_reconstructed = nltk.Tree.fromstring(
         ...
     )
```

```
[ ]: grader.check("parse_trees_reconstruct")
```

```
[ ]: tree1_reconstructed.pretty_print()
     tree2_reconstructed.pretty_print()
```

There. That looks much better.

4 Lab debrief

Question: We're interested in any thoughts you have about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment might include the following, but you're not restricted to these:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there general additions or changes you think would make the lab better?

Type your answer here, replacing this text.

Question: What *specific single* change to the lab would have made your learning more efficient? This might be an addition of a concept that was not explained, or an example that would clarify a concept, or a problem that would have captured a concept in a better way, or anything else you can think of that would have made this a better lab.

Type your answer here, replacing this text.

5 End of lab 3-2

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```