CS 2051: Honors Discrete Mathematics
Spring 2023 Homework 9 Supplement

Sarthak Mohanty

The control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers.

Sun Tzu, *The Art of War*

Overview

One day when I was a student taking 2051, I was perusing an article with all the important results in discrete math. However, I noticed an interesting statement:

> Noam Chomsky invents context-free grammars in his book "Syntactic Structures". At the time, he called them phrase-structure grammars and explored them in the context of modeling the syntax of natural languages.

Initially, I was confused. Noam Chomsky may be many things, including a world-renowned linguist and psychologist, but it was difficult to see his correlation to discrete math. However, as I learned more, I realized the importance of that statement. His discovery took an algorithmic approach to language development whose effects are still being seen today.

In this supplement, you'll learn about recursion, computational models, syntactical structures, and more!

Part 1: Way Down We Go (9 points)

Let's motivate this section by recalling some explication from a prior supplement:

> But, how can you evaluate arbitrary expressions without even knowing the names of the variables ahead of time? This question itself could be an entire supplemental, but luckily Python makes the job easy with the `eval()` function.

```
>>> eval('p and ((not p) or q)', {'p': True, 'q': False})
```

However, suppose we were now creating our own `eval()` function. This problem is not a trivial one, and requires careful consideration of various algorithmic techniques. Initially, one may assume that an iterative approach would suffice to traverse the string representation of the proposition, evaluating sub-expressions and gradually building up to the final evaluation. However, such an approach would quickly lead to convoluted code that would be difficult to maintain and debug. Therefore, we must turn to a more elegant solution: recursion.

Recursion is a powerful tool used for solving algorithms, and is loosely given as follows:

- If the problem is small enough to solve directly, then do so.
- Otherwise, reduce it to one or more simpler instances of the same problem.

There are many advantages to this approach, the main one lying in the fact that the same code can be used to solve both the original problem and its sub-problems, leading to simpler, more modular code.

To apply recursion to the task of creating our `eval()` function, we can break the problem down into two parts. First, we must parse the string representation of the proposition into a data structure that can be easily evaluated. Second, we must recursively evaluate this data structure to arrive at the final evaluation of the proposition.

1. We first recurse on the string itself. The idea is to first read the first token in the string, where a token is a basic "word" of our language: either one of the single-letter tokens `'T'`, `'F'`, `'('`, `')'`, `'~'`, `'&'`, `'|'`, or the two/three-letter "implies" and "iff" tokens `'<->'`, `'->'`, or a variable name like `'p'` or `'q76'`. This first token will tell you in a unique way how to continue reading the rest of the string, where this reading can be done recursively. For example, if the first token is an open parenthesis, `'('`, then we know that a formula ϕ must follow, which can be read by a recursive call. Once ϕ was recursively read, we know that the following token must be one of `'&'`, `'|'`, `or` `'->'`, and once this token is read then a formula ψ must follow, and then a closing parenthesis, `')'`. We can use this technique to encode our proposition into a data structure known as a tree, where we start with a root, and then branch off into different subtrees, each with their own roots. This will become concrete as you implement the below tasks.
2. Once we have constructed the tree data structure, we can move on to the second part of the problem: evaluating the proposition. This involves recursively evaluating the subtrees of the tree data structure, starting from the leaves and working our way up to the root. Each subtree represents a sub-expression of the original proposition, and its evaluation can be determined based on the logical structure of the subtree. Once we have evaluated all the subtrees, we can recombine them in such a way that the entire proposition is evaluated.

Note that there are two different types of recursion involved in this process. The first type of recursion involves parsing the string representation of the proposition and constructing the tree data structure, while the second type of recursion involves recursively evaluating the subtrees of the tree data structure.

These two types of recursion correspond to two different types of data structures: *lists* (in the form of strings) and *trees* (in the form of our constructed data structure). This will be explored in future coursework such as CS 351X.

In the very first coding supplement, you were tasked with evaluating a propositional formula given a supposed model. Fortunately, the built-in function `eval()` completed this task for us. *Unfortunately*, now you must do so without the use of that function.

In this part, you'll implement the following functions:

- `parse`: The function takes in a string, represented as a propositional formula, and parses it to create an appropriate Formula object. You may assume that the input string is valid.
- `evaluate`: this function takes in a Formula object, equipped with a model (remember that term from hw2-supp!?) and returns the evaluation.

Note: All functions must be implemented recursively, or call other recursive functions.

Part 2: Context-Free Grammars (10 points)

In this section, we get a taste of *models of computation*. Models of computation are abstract representations of how computation can be performed. They are commonly used to understand the limits and capabilities of computers and other computational devices.

It is not difficult to make the assumption that models of computation take in *input*, and return *output*. For the purposes, let's assume the input is a string, and the output is a boolean function. In other words, a computational model takes in a string as input, and decides whether or not to *accept* that string.

There are several models of computation, such as Turing machines, lambda calculus, and finite automata. In this section we present one such model, known as a *context-free-grammar*.

First, some definitions.

- A **terminal** is a symbol from our alphabet Σ^*. For example, if our alphabet was the set of binary stirngs, then 0 and 1 would be our terminals.
- A **nonterminal**, on the other hand, is any capital letter. For example, A, B, C are all examples of nonterminals.
- The rules of a context-free grammar consist of a set of **production rules** that describe how to generate strings from a starting symbol. Each production rule has a left-hand side, which is a nonterminal symbol, and a right-hand side, which is a sequence of terminal and nonterminal symbols.

To-do: finish the explanation (but hopefully the examples will make it all clear.)

Example 1: Binary strings

Let's say we were trying to create a context-free grammar for the language $L = \{1^n 0^n, n \in \mathbb{N}\}$. One such grammar is presented below

$$S \to 1S0 \mid \epsilon$$

Let's informally illustrate why this CFG is correct. Let's say we were trying to generate the string 111000. To do this, we would have the following set of steps:

$$S \to 1S0 \to 11S00 \to 111S000 \to 111000$$

As you can see, anytime we move one step to the right, we increase both the number of ones to the left and the number of zeros to the right by 1.

Example 2: Union of Two Languages

Let's say we were trying to create a CFG for the language

$$L = \{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}.$$

Using the first example, let's construct a grammar for the first language, given as

$$S_1 \to 0S_1 1 \mid \epsilon$$

and the grammar

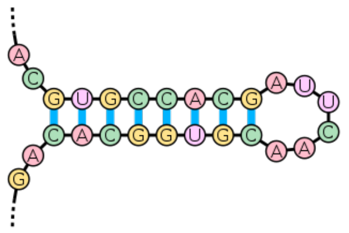
$$S_2 \to 1S_2 0 \mid \epsilon.$$

Then we can generate our desired language L with a CFG as follows:

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to 0S_1 1 \mid \epsilon \\ S_2 &\to 1S_2 0 \mid \epsilon \end{aligned}$$

Example 3: RNA Secondary Structure

CFGs were applied successfully to RNA secondary structure prediction in the early 90s. We can get a taste of this application through the following task.
Each RNA string is a string over the alphabet $\{A, C, G, U\}$. The string GUGCCACGAUUCAACGUGGCAC can fold into a lollipop shape like this:



The prefix GUGCCACG and suffix CGUGGCAC can together form the handle of the lollipop because they line up in such a manner that As are across from Us (or vice versa) and Cs are across from Gs (or vice versa). The candy part of the lollipop consists of the string AUUCAA, which has length 6.

In general, an RNA string can fold into a lollipop shape if it can be written as xyz, where x and z contain at least two characters each, y contains at least four characters, and the characters of x match up with those of z^R, in the sense that that corresponding pairs of letters fall in the set $\{(A, U), (U, A), (C, G), (G, C)\}$.

Induction vs Recursion

As a side note, you might be wondering why we introduced recursion in this supplement, when the current focus of the course this week is induction. This is because the two are actually deeply intertwined. **Induction is often used to prove the validity of recursive algorithms**[1].

For example, let's try to prove Example 1 using induction. We wish to prove that

$$L = L(G),$$

where $L(G)$ is the language of G. We do so in two parts.
PART 1. We first prove $L \subseteq L(G)$; in other words we prove $P(n)$, the statement

Every string of the form $1^n 0^n$ can be generated by the CFG.

[1] You'll definitely use this fact if you're taking a class such as CS 3511

Base Case: $P(0)$ is true, since we can apply the rule $S \to \epsilon$ to generate the string $1^0 0^0 = \epsilon$.

Inductive Step: Let $n \in \mathbb{N}$ such that $P(n)$ is true. To generate a string of the form $1^{n+1}0^{n+1}$, apply the first production rule $S \to 1S0$ once, obtaining the string $1S0$. Now we can replace S with $1^k 0^k$, which we know can be generated by the CFG by the inductive hypothesis. Thus, we obtain the string $1^{k+1}0^{k+1} \in L$, so $P(k+1)$ is true.

Conclusion: Therefore by induction, $P(n)$ is true for all $n \in \mathbb{W}$.

Part 2. Next, we prove $L \supseteq L(G)$; in other words we prove $P(n)$, the statement

Every string generated by the CFG in n productions is of the form $1^n 0^n$.

Base Case: $P(0)$ is true, since if a string in the CFG is generated in only one production it can only be the string ϵ, and $\epsilon = 1^0 0^0$.

Inductive Step: Let $n \in \mathbb{N}^+$ such that $P(n)$ is true. For G to generate a nonempty word, the first rule it must apply is $S \to 1S0$. By inductive hypothesis, the S on the right produces a word of the form $1^n 0^n$ in n productions. Thus the string produced in $n+1$ productions is of the form $1^{n+1}0^{n+1}$, so $P(n+1)$ is true as well.

Conclusion: Therefore by induction, $P(n)$ is true for all $n \in \mathbb{N}^+$.

This was an extremely tedious proof. The worst part? This grammar only had one production! Imagine if there were multiple production rules and multiple variable. A formal proof would take forever! For this reason, we try to generate CFGs simply through our intuition and examples.

In this part, you'll practice generating CFGs to create certain languages.
Implement the following functions:

- `generate_cfg_binary`: In Example 1 above, we presented a CFG to generate a language of the form $1^n 0^n$. Now let's weaken our restrictions even more. Create a CFG to decide the language

$$L_1 = \{w : w \in \{0,1\}^*, \#1(w) = \#0(w)\};$$

  in other words, L_1 represents all binary strings with the same number of 1's and 0's.

- `generate_cfg_union`: In Example 2, you learnt how to create CFGs for the union of multiple languages. Let's put that to the test. Implement a CFG to decide

$$L_2 = \{a^i b^j c^k \mid i, j, k \geq 0, \text{ and } i = j \text{ or } i = k\}$$

- `generate_cfg_rna`: Create a CFG to decide the set of all RNA secondary structures of the form of a "stem loop" as described above. Since terminals are typically lowercase characters, use 'u, g, c, u' instead of 'U, G, C, U'.

- `generate_cfg_tricky`: This one is tricky. Generate a CFG to decide the language

$$L_4 = \{1^i 0^j : 2i \neq 3j + 1\}$$

Part 3: Syntatical Structures (9 points)

Even English can be (at least somewhat) modeled as a context-free grammar. In this section, we give a brief overview of how this modeling works. However, you will need to go out and consult outsitde resources to fully complete the implementation. I recommend, in order

They also speak in "complex" sentences—sentences which become more capable of expressing our "complex" thoughts by including other sentences inside them. This is a form of "recursion" in English—enabled by our ability to multiply meanings by a kind of mental recycling of structures. Consider the following examples:

- 1
- 3

Implement the following functions:

- `generate_cfg_logic(atoms)` In Part 1, we created a parser for propositional strings. Since parsers are typically created for context free grammars, it is natural to assume there exists a CFG for the language of propositions, which there is.
- `generate_cfg_english(parts_of_speech)`: Create a CFG for a reasonable implementation of the english language. **For full credit, your CFG must be able to generate the sentences given in the test cases.** Your implementation should cover the main parts of speech: nouns, verbs, adjectives, adverbs, prepositions, articles, pronouns, conjunctions. An example of `parts_of_speech` is as follows:

```
parts_of_speech = {
    "noun": {"dumbbell", "barbell", "ab roller", "treadmill", "Prahlad", "Dave" ...}
    "adjective": {"fit", "athletic", "healthy", "motivated", "resilient"...}
    "adverb": {"quickly", "slowly", "eagerly", "steadily", "loudly", ...},
    "preposition": {"in", "on", "with", "at", "from", "over", "under", ...},
    "article": {"the", "a", "an", ...},
    "pronoun": {"he", "she", "they", "it", ...},
    "conjunction": {"and", "or", "but", ...},
    "intransitive verb": {"exercise", "run", "swim", "deadlift", "bench", ...}
    "transitive verb": {"lift", "carry", "deadlift", "bench", ...}
    }
```

Your CFG should be able to generate sentences like the following (ignoring punctuation and capitalization):

- The fit athlete lifted the dumbbell slowly.
- She ran on the treadmill quickly and enthusiastically.
- They exercised with the ab roller and the jump rope.
- He pulled the barbell steadily and with great effort.

Note that your CFG does not need to handle complex sentence structures such as relative clauses or embedded clauses, but it should be able to generate a range of simple and compound sentences. Your CFG also does not have to only generate semantically correct sentences, for example since we did not distinguish between common and nouns, it is possible to generate the sentence "She benched Dave enthusiastically", which doesn't semantically make sense.

To do: talk about ambiguity, bench can be verb or noun.

Also note that some verbs can be both intransitive and transitive.

(Optional) Part 4: Building a Parser

In Part 1 we built a parser for propositions. However, this can be made more general. A real parser is any component that extracts the meaning (or *semantics*) of a computational model. You may have heard about it in your other computer science classes, since most compilers and interpreters use parsers to generating the compiled code or performing the interpreted execution.

In the context of CFGs, this means a parser can take in a candidate string and check if the string is in the language or not. Indeed, the autograder contains such a parser to do just this.
The recursive approach is too inefficient for our purposes, so a good parser uses something called *dynamic programming*. The idea is found here.

Conclusion

Submission Instructions (10 pts)

After you fill the appropriate functions, submit the following files to Gradescope.

- `proposition_parser.py`
- `generate_grammar.py`

Most of the autograder is given to you this time, so you should be able to check your work locally.

References

[1] Gonczarowski, Y. A., & Nisan, N. (2022). *Mathematical Logic through Python*. Cambridge University Press.

[2] Sag, I. A., Bender, E. M., & Wasow, T. (2006). Syntactic theory: A formal introduction. CSLI.