CS 2051: Honors Discrete Mathematics
Spring 2023 Homework 9 Supplement

Sarthak Mohanty

Overview

One day when I was a student taking CS 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.

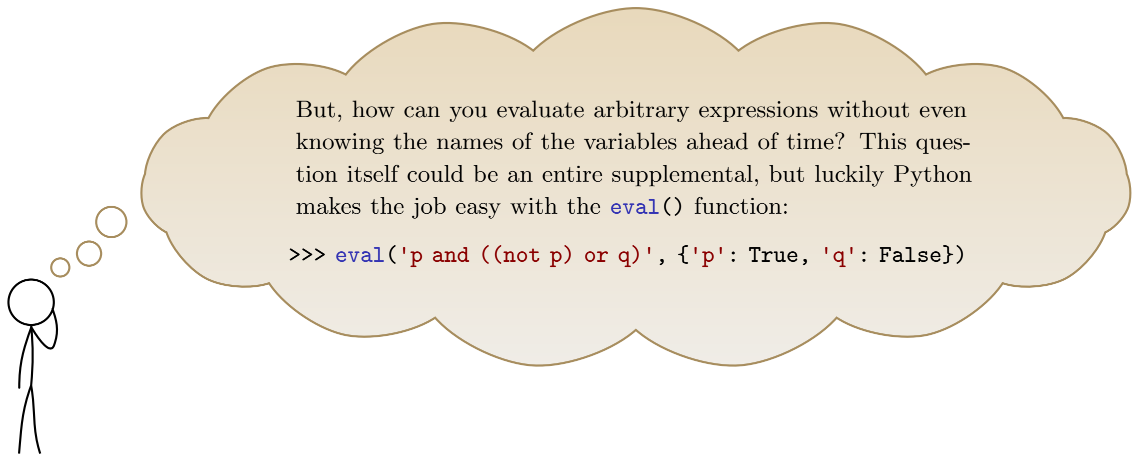
Initially, I was rather confused. Noam Chomsky may be many things, including a world-renowned linguist and psychologist, but it was difficult to see his correlation to discrete mathematics. However, as I looked more into the topic, I realized the true importance of his discoveries. Chomsky's work has not only transformed the field of linguistics but has also had a significant impact on the entire domain of computer science.

In this supplement, you'll quickly get up to speed (if you aren't already) with a powerful tool known as recursion. You'll use this tool to develop an important computational model, known as a context-free grammar. Finally, using this model, you'll try your hand at algorithmically expressing the English language. Let's get started!

(Optional) Part 1: Way Down We Go

> 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*

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})
```

We now have the opportunity to answer the question we originally posited. How would we go about implementing 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 turn to a more elegant solution.

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, such as a tree. Second, we must recursively evaluate this data structure to arrive at the final evaluation of the proposition.

Let's build our own `eval()` function. Implement the following functions:

- `parse(proposition)`: 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(proposition, model)`: This function takes in a string as well as an assignment of variables and returns the evaluation. The output should be identical to that of the analogous built-in function:

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

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

Part 2: Models of Computation (10 points)

In this section, we introduce an abstract representation for computational machines. *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. A computational model takes in a string as input, and returns a boolean value that represents 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*. We first introduce the concept through an example, and then formalize the definition.

Example 1: Binary strings

Let's say we were trying to create a computational model to *decide* the language $L = \{1^n 0^n, n \in \mathbb{N}\}$. What this means is that our computational model should take in a string, and *accept* the string (i.e. return `True`) if the string is in L, and *reject* otherwise.

One such model could be represented as follows: we would start with a temporary variable, call it S. Now at every time step, keep splitting up S into $1S0$; we can denote this as

$$S \to 1S0.$$

Finally, we want to get rid of S. Therefore, at every time step, allow the possibility of S to dissapear and be replaced with the empty string, ϵ. We indicate this as follows:

$$\begin{matrix} S \to 1S0 \\ S \to \epsilon \end{matrix} \quad \overset{\text{def}}{=\!=} \quad S \to 1S0 \mid \epsilon$$

If we ever generate a string identical to the input we take in, we *accept*, otherwise, we *reject*. Let's informally illustrate why this model decides L. Suppose we took in the input string $w = 111000$. Since $w \in L$, our model should accept the string, so our CFG should be able to generate it. To do this, all we have to do is tell our machine to execute the following set of steps:

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

At every time step, we increase the number of 1s and 0s on either side of S. Finally, when we have generated the desired string, we tell S to disappear, by replacing it with the empty string ϵ.

Context-Free Grammars

We now formalize the above concept. A **context-free grammar** is a 4-tuple (V, Σ, R, S). We now explain each of these elements in detail.

1. V is a finite set called the **variables**. The elements in V are commonly represented using a capital letter (i.e. $A, B, \dots$). In the above example, the only variable was S.
2. Σ is a finite set, disjoint from V, called the **terminals**. These can be any symbol; in Example 1, they were the binary numbers $0, 1$.
3. R is a finite set of **rules**. The left hand side of a rule is a single variable, and the right hand side can be any combination of variables and terminals.
4. $S \in V$ is the start variable.

The language of the grammar is then $L_{\mathsf{CFG}} = \{w \in \Sigma^* : S \to u_1 \to u_2 \to \cdots \to w\}$[1]. Put another way, it is the set of all valid strings generated by repeated application of our rules containing no variables.

Example 2: Union of Two Languages

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

$$L = \underbrace{\{0^n 1^n \mid n \geq 0\}}_{L_1} \cup \underbrace{\{1^n 0^n \mid n \geq 0\}}_{L_2}.$$

First, let's construct a grammar for L_1. This is the same as in Example 1:

$$S_1 \to 0S_11 \mid \epsilon.$$

Similarly, we can generate a grammar for L_2:

$$S_2 \to 1S_20 \mid \epsilon.$$

[1]Here the kleene star was used to define w as the concatenation of terminals.

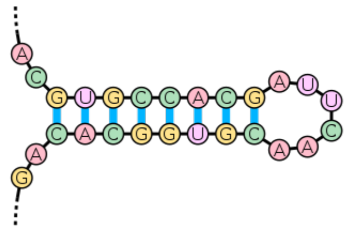
Then we can generate our desired language L with a CFG by 'combining' the two grammars, as follows:

$$S \to S_1 \mid S_2$$
$$S_1 \to 0S_11 \mid \epsilon$$
$$S_2 \to 1S_20 \mid \epsilon$$

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 scenario.

Each RNA string is a string over the alphabet $\{A, C, G, U\}$. The string GUGCCACGAUUCAACGUGGCAC can fold into a lollipop shape like so:



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 (the reverse of z), in the sense that that corresponding pairs of letters fall in the set $\{(A, U), (U, A), (C, G), (G, C)\}$. Can you find a CFG to successfully accept such strings?

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 models**.

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^n0^n can be generated by the CFG.

BASE CASE: $P(0)$ is true, since we can apply the rule $S \to \epsilon$ to generate the string $1^00^0 = \epsilon$.

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^k0^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^n0^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 $\epsilon = 1^00^0$.

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^n0^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 prove the validity of our CFGs simply through intuition and test cases.

Let's practice generating CFGs. Implement the following functions:

- `generate_cfg_binary`: In Example 1 above, we presented a CFG to generate a language of the form 1^n0^n. Now let's weaken our restrictions even more. Create a CFG to decide the language

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

that is to say, 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

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

- `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

$$L_4 = \{1^i0^j : 2i \neq 3j + 1\}$$

Note: Make sure to check whether your implementation correctly accepts or rejects the empty string ϵ, which we denote in code as the empty list `[]`.

Part 3: Syntatical Structures (10 points)

This brings us to Noam Chomsky. He proposed that language is innate to humans and that we are born with a universal grammar that allows us to learn and produce language. He argued that this universal grammar consists of a set of **rules** or principles that are hard-wired into our brains, and that these rules underlie the structure of <u>all</u> human languages. Sound familiar? Indeed, Noam Chomsky's rules were formulated in the exact structure of a context-free grammar.

Syntax vs Semantics

Recall that the entire first part of this course was dedicated to Logic, and how we can translate our human language to logical statements. Thus, Chomsky's hypothesis presented above implies that we can express the language of Logic as a context-free grammar. You are tasked to do so below.

But wait! What about the logical statement

$$T \vee T \vee T$$

Should our context-free grammar generate this sentence? On one hand, it is **syntactically** valid, in the sense that it follows the rules of well-formed formulae. On the other hand, one could argue it is **semantically** pointless: it is equivalent to (and should be replaced with) the simplified expression T.

Syntactically, propositional statements are just strings, devoid of meaning. It is our interpretation of them that gives them semantic meaning. Chomsky's argument only applies to the syntax of languages, not semantics, so our CFG should accept such statements. On the other hands, strings such as $((\Rightarrow))$ are not even syntatically valid, so we should reject them.

English as a CFG

Even English can be (somewhat) modeled as a context-free grammar. In this section, we provide a brief overview of such a blueprint.

Consider the following sentence:

"The jet black cat sat on the mat."

This simple sentence contains a subject ("cat") and a predicate ("sat on the mat"), which are both necessary components for a complete sentence in English. However, English also allows for more complex structures. For example,

"The <u>very</u> jet black cat sat on the mat."

In this case, we have incorporated the adverb "very" while maintaining the sentence's validity. No worries, right? Simply adding a rule to allow an adverb before adjectives should suffice to handle this situation. However, the situation is more nuanced, as it appears that we can insert an indefinite number of adjectives without compromising the sentence's syntactical integrity. The sentence

"The very very very very very jet black cat sat on the mat"

is still completely valid, even if it's impractical. This is a form of "recursion" in English – enabled by our ability to multiply meanings by a kind of mental recycling of structures, and CFGs are a perfect candidate to represent such structures.

Even entire sentences can be expressed using conjunctions, as in the following:

$$\underbrace{\text{The cat sat on the mat}}_{\text{sentence}} \text{ and } \underbrace{\text{the dog sat on the log}}_{\text{sentence}}.$$

While these simple examples provide a cursory introduction, further resources should be consulted to gain a deeper understanding of the complex syntactical structures needed for the English language (and to complete the implementation requirements). The following set of slides and exposition should be sufficient.

Implement the following functions:

- `generate_cfg_logic(atoms)`: Create a CFG for a syntactical representation of the language of Logic. The alphabet of your grammar should be over $\Sigma = \texttt{atoms} \cup \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, (,), T, F\}$. Here `atoms` is a set of lowercase letters representing the propositions over which our language is defined. You may insert more parentheses than necessary.
- `generate_cfg_english(parts_of_speech)`: Create a CFG for a reasonable implementation of the english language. 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 = {
    "singular_noun": {"core", "barbell", "ab roller", "treadmill",...},
    "plural_noun": {"dumbbells", "weights", "ab rollers", "treadmills",...},
    "proper_noun": {"Reese", "Paul", "Sofia", "Saloni", "Ananya", "Ron"...},
    "intransitive verb": {"exercise", "run", "swim", "deadlift", "bench",...},
    "transitive verb": {"lift", "carry", "deadlift", "bench",...},
    "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",...}
    }
```

For full credit, your CFG must be able to generate sentences of the following structures (ignoring tense, spaces, punctuation and capitalization):

- The trainer carried the dumbbells.
- She ran on the treadmill quickly and enthusiastically.
- They exercised with the ab roller and the jump rope.
- The man swims.
- Reese gave her pull-up bar to Paul.
- Ron ate his cold, delicious protein shake.
- The very extremely tall, intelligent woman began to deadlift.
- The motivated fellow lifted the weights, but the unmotivated fellow dropped them.
- ~~With the ab roller, you can strengthen your core.~~
- ~~A fit and healthy lifestyle is a worthy goal to pursue.~~

Note: This CFG also does not have to generate semantically correct sentences. For example, since we did not distinguish between common and proper nouns, it is possible to generate the sentence “She benched Saloni enthusiastically,” which semantically does not make sense.

Submission Instructions (10 pts)

After you fill the appropriate functions, submit the `generate_grammar.py` file to Gradescope. The main structure of the autograder is given to you this time, so you should be able to check your work locally.

If you want, you can also submit `proposition_parser.py` and I will take a look, but it is ungraded.