Automata Theory: How Machines Hear and Recognize Speech

Emily Chen, Arya Nayak, Hazel Thrasher
Mentor: Alicia Lin

May 2026

Abstract

The theory of computation provides the framework for understanding how computers and machines process information and solve problems. This paper explores the fundamental concept of automata theory and its connection to real-world speech detection systems. To establish our comprehension of finite automata and regular languages, we present formal definitions and state diagrams. We further investigate automata theory with context-free grammars and pushdown automata, utilizing computational power to illustrate recursive patterns of languages. The relationship between pushdown automata and context-free grammars is significant in understanding how different computational models relate to each other. The link between automata theory and practical applications such as speech recognition demonstrates how simple computational models form the foundation for valuable real-world innovations.

1 What is a computer?

When the word *computer* comes to mind, our first mental image is probably a laptop with a keyboard and screen. Our visualized computer is a device capable of storing information and performing calculations with remarkable speed.

However, in this paper, we will discuss automata theory, which is the study of simpler, idealized computers known as computational models. By breaking down the extreme complexity of real-world computers into their most fundamental components, we can better understand the actual processes that computers use to solve problems, store information, and carry out algorithms.

2 Finite Automata

One of the simplest computation models is the **finite automaton**. Finite automata have limited memory, but despite their small memory are able to carry out many useful tasks. To better understand the process a finite automata uses

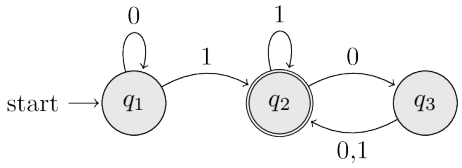

Figure 1: State diagram of a finite automaton A

to process an input, we will introduce the state diagram, visually depicting the finite automaton.

Figure 1 depicts a finite automaton that we will name A. It has 3 **states**, which are the circles labeled q_1, q_2, and q_3. The state q_1 is called the **start state** because it has an arrow pointing at it, indicating that this state is where the input will first enter. The **accept state**, which is q_2, is shown with a double circle around it, and the arrows traveling between states are known as **transitions**.

When an input is received by A, the string will be processed and the automaton will either **accept** the input or **reject** it. An automaton processes an input by starting at the start state and receiving and reading the symbols one by one from left to right. After reading each symbol, A moves between the states as indicated by the transition that has the corresponding symbol as its label. After reading the last symbol, if A is in an accept state, then the output is accept, and if it ends up in a reject state, the output is reject.

Now that we have a rough understanding of how finite automata work, we will introduce the formal definition of a finite automata.

Definition 2.1: A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q represents the finite set of states,
2. Σ represents the input alphabet,
3. $\delta : Q \times \Sigma \rightarrow Q$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

The set of all strings that a finite automata accepts is called the **language** of the finite automata. We can also more broadly define the types of languages that can be accepted by a finite automaton.

3 Regular Languages

Definition 3.1: A language L is a **regular language** if there exists a finite automaton that accepts the strings in L. It is important to note that while a sin-

gle finite automaton machine may accept many, many strings, it only recognizes *one* language.

There are also many operations that allow for the manipulation of regular languages. In basic arithmetic, we use plus, minus, multiplicaiton or division symbols to manipulate expressions of numbers. Similarly, in regular languages, we can use a toolbox of three **regular operations**, which give us understanding about the properties of regular languages. These operations are called **union**, **concatenation**, and **star**.

Definition 3.2: Let A and B be regular languages. The regular expressions union, concatenation, and star are defined as follows:

1. **Union:** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
2. **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
2. **Star:** $A* = \{x_1x_2...x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

We say that the class of regular languages is **closed** under certain operations if applying those operations to a regular language always produces another regular language.

Theorem 3.1: The class of regular languages is closed under the operations of union, concatenation, and star.

We can easily prove theorem 3.1 by using a concept called nondeterminism. We will not include the proofs in our paper, but are available in Sipser's *Introduction to the Theory of Computation*, Edition 3.

4 Nondeterminism

So far in the paper, the finite automata we have been referring to are **deterministic**; every next step in the computation follows in a known, specifically determined way. However, in a **nondeterministic** machine, several choices may exist for the next state in a computation.

Although every deterministic finite automaton (**DFA**) is defined to also be a nondeterministic finite automaton (**NFA**), there are numerous key differences between the two types of machines.

To better visualize these differences, we utilize the state diagram of an NFA, shown in Figure 2.

Many differences immediately become clear when looking at DFAs vs NFAs. Every state of a DFA always has exactly one exiting transition arrow for each symbol of the input alphabet, but this is not the case for NFA N. In Figure 2, State q_1 does not have an exiting arrow labeled a, for example. Also, NFAs often use the label ε, as shown in Figure 2 from state q_1 to q_2, which means a

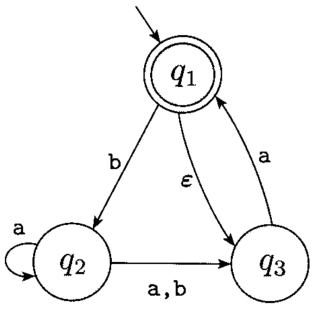

Figure 2: State diagram of a nondeterministic finite automaton N

machine can nondeterministically decide to move from q_1 to q_2 regardless of the input.

An NFA computes by reading a symbol, splitting into many copies of itself, and following all the possibilities of a next step in parallel, repeating this process for every symbol of the input. If any one of these copies reaches an accept state by the end of the input, the NFA accepts the string. In this way, the processing of NFAs is like a tree of possibilities, with the root of the tree at the first symbol of the computation.

5 Regular Expressions

In simple arithmetic we use the $+$, $-$, $\times$, and $\div$ symbols, among countless others, to manipulate numbers into expressions. Similarly we can do this in regular languages, using the regular operations to form **regular expressions**.

While an arithmetic expression like $(5+2)\times 3+7-4$ would yield 24, a regular expression like $(0 \cup 1)1*$ would yield a regular language. In this example, this language would consist of all strings that start with either a 0 or a 1, and is followed by any number of 1s. This expression contains all three regular expressions. The concatenation symbol is after the parenthesis and before the 0*, and for shorthand purposes is omitted from the notation.

The regular expressions also have an order of operations! The star operation is done first, followed by concatenation, and finally by union, with the exceptions being parentheses altering the order. For example, the regular expression $(0 \cup 1)1*$ has parentheses markings around the $0 \cup 1$ means that we consider the $0 \cup 1$ expression first, so we know the language consists of strings starting with a 0 or 1. Next we consider the star operation, so we know the language consists of strings that end in any number of 1s. Finally, we consider concatenation, so we know the language consists of strings that start with 0 or 1 attached in front of any number of 1s.

Now we will look at the formal definition of a regular expression.

Definition 5.1: We say that an expression R is a regular expression if R is

1. a for some a in the alphabet Σ

2. ε,

3. $\emptyset$,

4. $R_1 \cup R_2$ where R_1 and R_2 are regular expressions

5. $R_1 \circ R_2$ where R_1 and R_2 are regular expressions, or

6. $R_1{}^*$ where R_1 is a regular expression

In the first 2 conditions, the regular expressions a and ε represent the languages a, denoting the language containing only the character a, and ε, denoting the language containing only the empty string. In 3, $\emptyset$ represents the empty language, which is the language that has no strings.

Regular expressions and finite automaton, the two computational models discussed so far, may superficially seem like they could not be more different. However, finite automata and regular expressions are equivalent in their descriptive power. We state a relation between regular languages and regular expressions:

Theorem 5.1: A language is regular if and only if some regular expression describes it.

While we do not include a rigorous proof of this theorem, a brief explanation is that since a regular language is a language recognized by a finite automaton, and regular expressions build up regular languages, all languages accepted by finite automaton can in fact be equivalently described with a regular expression.

6 Context Free Grammars

While finite automata and regular languages form the foundation for understanding computation theory, they have important limitations, such as being unable to express certain simple languages. To address these constraints, we introduce **context-free grammars** (see definition 3.1) (CFGs), which provide a more powerful way of describing languages. These grammars also have the ability to describe features with a recursive structure, which makes them useful in real-world applications like the compilation and specification of programming languages.

To better understand how context-free grammars work, we begin with an example. One variable, usually the variable on the left-hand side of the first rule, is designated as the **start variable**. In our example above, G_1's variables are A and B, with A as the start variable. The terminals are 0, 1, and #. Consider a CFG called G_1 defined by the following rules:

- $A \to 0A1$

- $A \to B$
- $B \to \#$

A grammar is a set of **substitution rules**, also known as **productions**. Each rule consists of a variable on the left-hand side and a string on the right-hand side, separated by an arrow. The symbol on the left is called a **variable**, while the string on the right may include both variables and other symbols called **terminals**. The variables are typically written as capital letters, whereas the terminals represent the input alphabet and are often expressed using lowercase letters, numbers, or special symbols.

A grammar generates each string of the language through the following process:

1. Begin with the start variable.

2. Select a variable that is currently in the string and apply a substitution rule that replaces it with the rule's right-hand side.

3. Continue this process until no variable remain and the string contains only terminals.

For example, grammar G_1 generates the string 000#111. The sequence of steps used to produce a string is called a **derivation**. The following derivation demonstrates how grammar G_1 generates the string 000#111:

$$A \implies 0A1 \implies 00A11 \implies 000A111 \implies 000B111 \implies 000\#111.$$

The same information can also be represented visually using a **parse tree**, which shows how a string is generated step by step from the grammar. An example of a parse tree for the string 000#111 in grammar G_1 is shown in Figure 3.

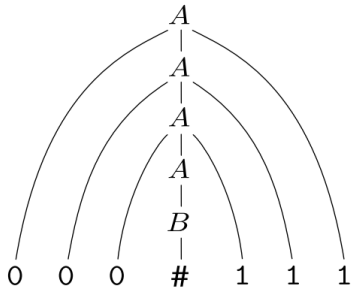

Figure 3: Parse tree for 000#111 in grammar G_1

All strings that can be produced through such derivations make up the **language of the grammar**. We denote the language generated by G_1 as $L(G_1)$. By experimenting with this grammar, we find that:

$$L(G_1) = \{0^n \# 1^n | n \geq 0\}$$

This means that the language consists of any number of 0s followed by a #, and then the same number of 1s.

Any language that can be generated by some context-free grammar is called a **context-free language** (CFL).

We can now define a context-free grammar (CFG) more precisely.

Definition 6.1: A **context-free grammar** is a 4-tuple (V, Σ, R, S), where

1. V is finite set called the **variables**,

2. Σ is a finite set, disjoint from V, called the **terminals**,

3. R is a finite set of **rules**, with each rule being a variable and a string of variables and terminals, and

4. $S \in V$ is the start variable.

If u, v, and w are strings made up of variable and terminals, and $A \to w$ is a substitution rule, then we say that uAv **yields** uwv, which is also written as $uAv \implies uwv$. We say that u **derives** v, written as $u \implies {}^*v$, if $u = v$ or if a sequence $u_1, u_2, ..., u_k$ exists for $k \geq 0$ and $u \implies u_1 \implies u_2 \implies ... \implies u_k \implies v$.

The language of the grammar is the set of all terminal strings that can be derived from the start variable. In this case, it would be

$$\{w \in \Sigma^* | S \implies {}^*w\}.$$

For example, in grammar G_1, $V = \{A, B\}$ is the set of variables, $\Sigma = \{0, 1, \#\}$ is the set of terminals, $S = A$ is the start variable, and R is the set of its three substitution rules.

When using grammars, we describe them by listing only their rules. From these, we can identify the variables as the symbols that appear on the left-hand side of the rules, the terminals as all the remaining symbols, and the start variable as the variable on the left-hand side of the first rule.

Now that we have formally defined a context-free grammar, we can look at an example to better understand how these rules generate strings in a language.

Consider grammar $G_2 = (\{S\}, \{a, b\}, R, S)$, where the set of rules R is $S \to aSb|SS|\epsilon$.

This grammar generates strings such as *abab*, *aaabbb*, and *aababb*. To better understand the structure of this language, we can think of a as a left parenthesis $''('' $ and b as a right parenthesis $'')''$. When viewed in this way, $L(G_2)$ consists of all strings of properly nested parentheses. Additionally, this example shows that a substitution rule may contain the empty string ϵ on the right-hand side.

7 Pushdown Automata

Although context-free grammars give us a way to generate languages, they do not demonstrate how a machine would recognize them. To bridge this gap, we present **pushdown automata** (PDAs), which are computational models similar to nondeterministic finite automata but have an additional component called a **stack**. This extra memory allows PDAs to recognize certain nonregular languages. PDAs are equivalent in power to context-free grammars, giving us two options to prove that a language is context free.

Figure 4 provides a schematic representation of a finite automaton. In this model, the control represents the states and the transition function, the tape contains the input string, and the arrow represents the input head, indicating the next symbol to be read.

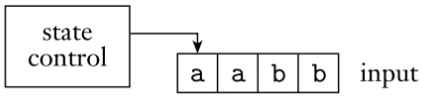

Figure 4: Schematic of a finite automaton

By adding a stack to the model, we obtain a schematic representation of a pushdown automaton, as shown in Figure 5.

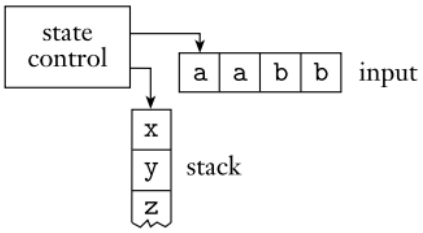

Figure 5: Schematic of a pushdown automaton

A pushdown automaton has the ability to store information using its stack. It can write symbols onto the stack and read later. When a symbol is added, it is placed on top of the stack, which pushes the existing symbols downward. At any time, only the top symbol of the stack can be read or removed. When it is removed, the remaining symbols below move back up to take its place. Adding a symbol to the stack is called a **push**, while removing a symbol is called a **pop**. Because all access to the stack is done at the top, the stack operates as a **last in,**

first out (LIFO) structure. This means that the most recently added symbol is the first one that can be accessed or removed. As a result, earlier information remains inaccessible until all symbols added after it have been removed.

A stack is useful as it can store an unlimited amount of information. Recall that a finite automaton cannot recognize the language $\{0^n1^n|n \geq 0\}$ since its memory is limited and it cannot keep track of large amounts. However, a PDA can recognize this language by using its stack to record the number of 0s it has read. The PDA reads symbols one at a time and pushes a 0 onto the stack each time it is read. Once it begins reading 1s, it pops a 0 off for each 1 read. If the input ends exactly when the stack becomes empty of 0s, then the string is accepted.

Pushdown automata can be either deterministic or nondeterministic, and unlike finite automata, these two types are not equivalent in power. Nondeterministic PDAs can recognize certain languages that deterministic PDAs cannot. If the stack becomes empty while 1s still remain, if the stack still contains 0s after the 1s are finished, or if any 0s appear after a 1, then reject the input.

The formal definition of a PDA is similar to a finite automaton, but with the addition of a stack. The stack may use different alphabets for its input and its stack, so we define both an input alphabet Σ and a stack alphabet Γ.

The key component of a PDA is its transition function, which describes the machine's behavior. Recall that $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$. The transition function depends on three things: the current state, the next input symbol read, and the top symbol of the stack. Either symbol may be ϵ which means that the machine will move without reading a symbol from the input of the stack. Therefore, the domain of the transition function is $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$.

Based on this information, the PDA may: move to a new state or optionally write a symbol on top of the stack. The function δ represents this by returning a member of Q together with a member of Γ_ϵ, which is a member of $Q \times \Gamma_\epsilon$. Because PDAs can be nondeterministic, there may be multiple possible next moves. Due to this, the transition function returns a set of members of $Q \times \Gamma_\epsilon$, which is a member of $\mathcal{P}(Q \times \Gamma_\epsilon)$. Putting it all together, our transition function is δ: $Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$.

Definition 7.1: A **pushdown automaton** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,

2. Σ is the input alphabet,

3. Γ is the stack alphabet,

4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function,

5. $q_0 \in Q$ is the start state, and

6. $F \subseteq Q$ is the set of accept states.

A PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts an input string w if the string can processed through a sequence of valid transitions that ends in an accept state. Let $w = w_1 w_2 ... w_m$, where each $w_i \in \Sigma_\epsilon$. Then, there must exist a sequence of states $r_0, r_1, ..., r_m \in Q$ and strings $s_0, s_1, ..., s_m \in \Gamma^*$ such that

1. $r_0 = q_0$ and $s_0 = \epsilon$. This condition means that M starts in the start state with an empty stack.

2. For $i = 0, ..., m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$. This condition shows that M moves legally according to the state, stack, and next input symbol.

3. $r_m \in F$. This condition states that the machine ends in an accept state after processing the entire input.

Now that we have formally defined a pushdown automaton, we can look at an example that explores how nondeterministic PDAs work in computation theory.

Example 1. *This example describes a pushdown automaton that recognizes the language*

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

.

The PDA begins by reading the input and pushing each a onto the stack. After all the a's have been processed, the stack contains a record of how many were read. The machine must then match the a's with either the b's or the c's, but it cannot determine in advance which condition ($i = j$ or $i = k$) will be satisfied.

To handle this, the PDA uses nondeterminism. It guesses whether to match the a's with the b's or with the c's. You could think of this as the machine having two possible branches, one for each choice. In each branch, the PDA matches symbols by popping from the stack accordingly If at least one branch accepts, then the input is accepted.

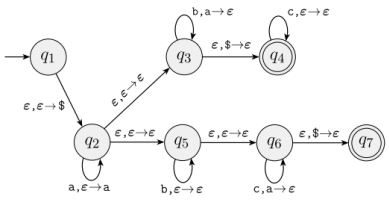

Figure 6: State diagram for PDA M_1 that recognizes

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

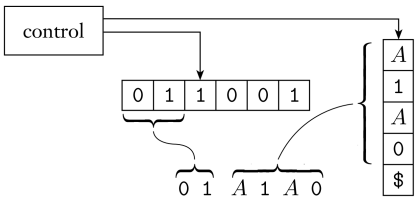

Figure 7: Diagram of P representing an intermediate string $01A1A0$

8 Relationships between Finite Automata and Context Free Grammars

There are many connections that can be made between different types of computational models. Specifically, we can prove that pushdown automata and context free grammars are equivalent in power.

Theorem 8.1: A language is context free if and only if some pushdown automata recognizes it.

We will prove the connection between CFLs and PDAs by proving both directions of this theorem.

Lemma 8.1.1: If a language is context free, then some pushdown automata recognizes it.

Proof of Lemma 8.1.1:

Let A be a CFL. We know A has a CFG G that generates it, and we will show how to convert G into an equivalent PDA called P. We will design this PDA to work by accepting its input w, if G generates that input, by determining whether or not there is a sequence of substitutions using the rules of CFG G that leads from the start variable to w.

The PDA will begin by writing the start variable on its stack, going through a series of intermediate strings and substitutions until it reaches a string that contains only terminal symbols, and there are no more variables to substitute for. This means the PDA accepts exactly when the input string can be generated by the CFG.

A rough description of P is as follows, and a diagram of an intermediate configuration is shown in figure 7.

1. Place an end-marker symbol \$ and the start variable on the stack.

2. Repeat the following steps:

a. If the top of the stack is a variable symbol A, nondeterministically select

one of the rules for A and substitute the variable by the string on the right side of the rule.

b. If the top of the stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat this step. If they do not, reject this branch of nondeterminism.

c. If the top of the stack is the symbol \$, enter the accept state, accepting the input if it all has been read.

Now we will give more formal details on how to construct the PDA $P = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$ that is equivalent to the CFG G. We will use a simplified notation of the transition function δ that will allow us to simulate writing an entire string on the stack in one step, by introducing additional states to write the string that we want on the stack one symbol at a time.

First, we let the symbols q and r represent two states of the PDA P, let the symbol a be in the input alphabet Σ_ε, and let the symbol s be in Γ_ε.

(For clarification, the epsilon subscript of Σ means the PDA transition can either read a normal input symbol from Σ, or make an ε transition, which changes state without consuming any input. Similarly, the epsilon subscript of Γ means that Γ_ε allows a stack symbol from Γ, or no symbol.)

We want the PDA to go from state q to state r when it reads an a in the input and pops s. At the same time we want the PDA to push the string $u = u_1u_2...u_l$ onto the stack. For this step, we will introduce new states $q_1, q_2, ..., q_{l-1}$ and represent the transition function as:

$(r, u) \in \delta(q, a, s)$ where q is the current state of the PDA, a is the next input symbol, and s is the symbol currently on top of the stack.

The PDA reads the symbol a and pops s. Before it moves on to state r, it will push the string u onto the stack with the following steps:

$\delta(q, a, s)$ to contain (q_1, u_l),

$\delta(q_1, \varepsilon, \varepsilon) = (q_2, u_{l-1})$,

$\delta(q_2, \varepsilon, \varepsilon) = (q_3, u_{l-2})$,

...

$\delta(q_{l-1}, \varepsilon, \varepsilon) = (r, u_1)$.

These steps mean that we:

- Start in state q
- Read input symbol a
- Pop stack symbol s
- Then push the string $u_1u_2 \cdots u_l$ onto the stack by a sequence of ε-transitions.

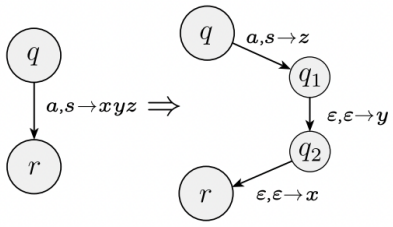

Figure 8: This figure depicts an example of a state diagram implementing the transition function

The symbols are pushed in reverse order:

- first u_l
- then u_{l-1}
- ...
- finally u_1

This reversed order is used because the stack is last in, first out. After all the pushes, the top of the stack ends up being u_1.

Figure 8 shows a state diagram of the shorthand $(r, xyz) \in \delta(q, a, s)$, where we want to push the string xyz onto the stack.

Now we will describe the cases of states and transitions of the PDA P. The PDA P has 3 main states: q_{start}, q_{loop}, and q_{accept}, as well as a set of helper states that help us implement the shorthand transition functions.

The machine begins in the state q_{start} and begins the stack with the symbols $S\$$, where S represents the start variable of the CFG and \$ represents the bottom of the stack. Next, the PDA moves to the looping state, q_{loop}.

In state q_{loop}, the PDA P will repeatedly perform one of three actions:

- If the top of stack is a variable, say A, then the PDA will nondeterministically replace A using one of the grammar rules of the CFG G, $A \to w$.
- If the top of the stack is a terminal symbol that matches the current input symbol, the PDA reads the input symbol and removes - "pops" - the matching terminal from the stack.
- If the bottom of stack marker \$ is reached, the PDA moves to the accept state q_{accept}.

This process, shown visually in Figure 9, shows how the PDA P simulates derivations of the context-free grammar G by replacing variables with the grammar's rules, and matching terminal symbols with the input string. The PDA only

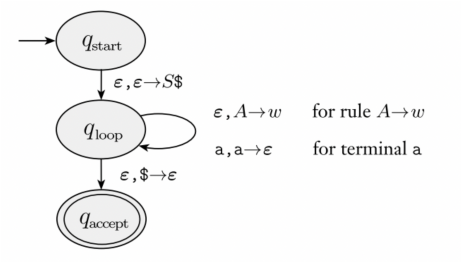

Figure 9: State diagram of PDA P

continues if the input exactly matches the terminals produced by the CFG derivation, meaning the PDA and the CFG are language equivalent and define the same language.

This completes the proof of Lemma 8.1.1, showing that if a language is context free, some pushdown automaton can recognize it.

Lemma: 8.1.2 If a pushdown automaton recognizes some language, then it is context-free.

Proof of Lemma 8.1.2:

To prove the lemma, we construct a context-free grammar G such that $L(G) = L(P)$ for a given pushdown automaton P.

We assume, without loss of generality, that P is modified to adhere to three constraints that make the conversion easier: it has only one accept state; it must empty its stack before acceptance; and each transition performs exactly one stack operation (either a push or a pop).

$P = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept})$, and from that we construct G. The variables of G are $A_{p,q} \mid p, q \in Q$, representing each pair of states in P. The start variable is $A_{q_0, q_{accept}}$.

Two cases must be taken into account when completing this proof:

Case 1: The stack empties before the PDA is finished reading the input, in which case the variable $A_{p,q}$ is separated into $A_{p,r}A_{r,q}$

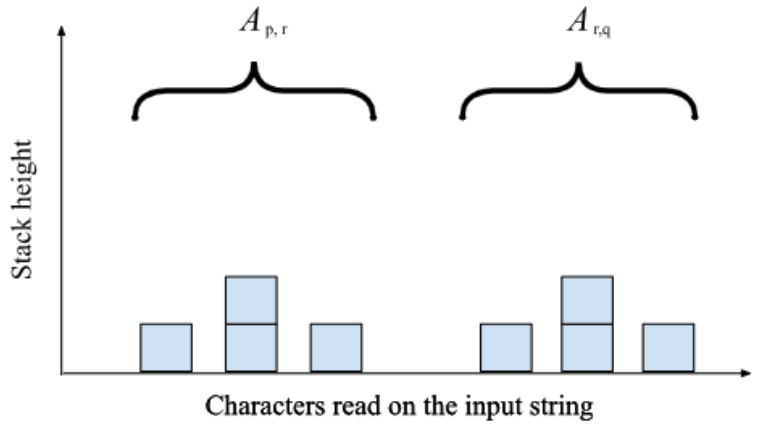

Figure 10: Example of case 1

Case 2: Case 1 doesn't happen

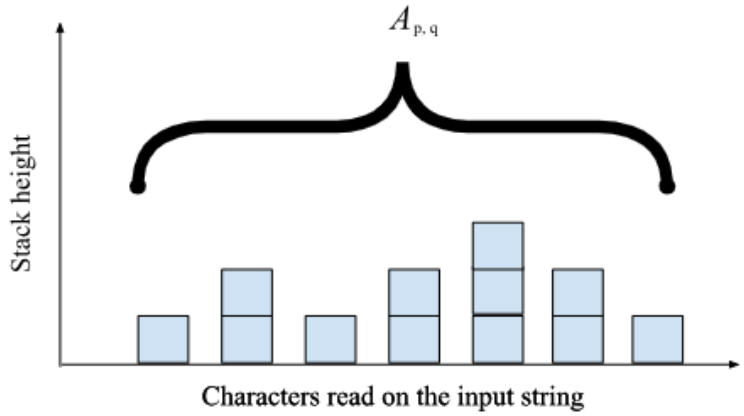

Figure 11: Example of case 2

First, we must show that if $A_{p,q}$ in grammar G generates the string x, then x can bring PDA P from q to p, starting with an empty stack and finishing with an empty stack.

This is a proof by induction on the number of steps it takes to generate the string x when the start variable is $A_{p,q}$.

We begin with a base case, where the number of steps to derive the string is just one. This happens when we begin with variable $A_{p,p} \to \epsilon$. In PDA P, the empty string takes p to p with an empty stack.

Then, we move on to proving that a derivation with $k+1$ steps also brings p to q, and ends in an empty stack, where k is the supposed maximum length where this could be true (and $k \geq 1$). This is shown to be true in Case 1 and Case 2 (see Figures 10 and 11).

Next, we must show that if x can bring PDA P from q to p, starting with an empty stack and finishing with an empty stack, $A_{p,q}$ generates x.

This is very similar. It is also a proof by induction on the number of steps it takes for PDA P to go from state p to q with an empty stack on a given input x.

The basis case is where the computation has zero steps, beginning and ending in a given state p. In this case, $x = \epsilon$, because P can't read any characters, staying in the same state with an empty stack. This shows that $A_{p,p} \stackrel{*}{\Longrightarrow} \epsilon$ which aligns with G's rule that $A_{p,p} \to \epsilon$.

Much like the induction step for the previous claim, this next step shows that this is the case for when the computation has k steps (where k is bigger than the basis case) and $k+1$ steps. This is shown to be true for cases 1 and 2, which proves our claim.

This completes the description of the proof for Lemma 8.1.2, which, along with Lemma 8.1.1, proves **Theorem 8.1**: that for every CFL, there is a PDA that recognizes it.

9 Applications in Natural Language Processing

There are many real-world applications for the contents of this paper. One of them is Natural Language Processing, in which Context-free grammars, which generate Context-free languages, are fundamental. Natural Language Processing, or NLP, is a technique that allows machines to understand, interpret, and generate human language.

The variables (or non-terminal symbols) of the context-free grammar can represent the different parts of speech, such as nouns and verbs. Meanwhile, terminal symbols represent the actual words — or, more specifically, TOKENS —which are the fundamental units—such as words, characters, or subwords—that a computer can process after breaking down text.

Here's an example:

[SENTENCE] → [NOUN-PHRASE][VERB-PHRASE]
[NOUN-PHRASE] → [CMPLX-NOUN] | [CMPLX-NOUN][PREP-PHRASE]
[VERB-PHRASE] → [CMPLX-VERB] | [CMPLX-VERB][PREP-PHRASE]
[PREP-PHRASE] → [PREP][CMPLX-NOUN]
[CMPLX-NOUN] → [ARTICLE][NOUN]
[CMPLX-VERB] → [VERB] | [VERB][NOUN-PHRASE]
[ARTICLE] → a | the
[NOUN] → child | parent | dog
[VERB] → touches | helps | pets
[PREP] → with

Figure 12: Example of how a CFG might generate a sentence

Tokenization in NLP is the foundational process of breaking down raw, unstructured text into smaller, meaningful units called tokens. This can be done using CFGs.

Example sentence: *"The fox and the cat are running a marathon."*, shown in Figure 13. As shown, the sentence is deconstructed into its primary components: the subject, verb, and object. This is further factored into noun and verb phrases, which are ultimately reduced to their base roots and associated suffixes.

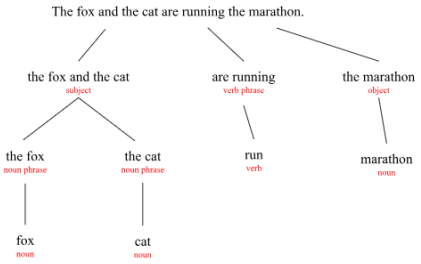

Figure 13: Simplified example of how a computer might tokenize a sentence

CFGs can be used for grammar checking, such as spellcheck, translation (e.g., Google Translate), speech recognition, and language generation and comprehension with AI chatbots. Today, most of these tasks are performed by AI, so we know little about the code and algorithms that solve them. But CFGs are still much more efficient than AI for tasks like input parsing, which is done through tokenization.

9.1 General Applications of Automata Theory

Automata theory has many real-life applications outside of natural language processing. For example, Finite Automata are used to design vending machines, elevators, and traffic light controllers, or other machines that can be described as sequential circuits where the output depends on the current input and previous states. Context-free languages and grammars are the backbone of the syntax of modern programming languages, providing the formal framework used to define their structure and facilitate the design of efficient compilers and parsers.

Speaking more generally, machines such as Context Free Languages are fundamental for understanding the capabilities of real-world computers. It's important to understand how the algorithms and mechanisms behind computers work because they are the backbone of modern computers, which govern all aspects of human life.

10 References

[1] Sipser, M. (2013). Introduction to the Theory of Computation (3rd ed.).

11 Acknowledgements

We would like to express our sincerest gratitude to our amazing mentor, Alicia Lin, for her guidance, patience, and encouragement throughout the semester. With her help, we were able to open our eyes to the fascinating world of computation theory. Alicia greatly helped us deepen our understanding of automata theory and numerous other topics through countless engaging learning experiences and insightful discussions, and we endlessly appreciate her support.

We are also extremely grateful to the PRIMES Circle coordinators, Paige Bright and Mary Stelow, for leading the program and organizing chalk talks and meetings for us to share our knowledge. We are so thankful for their efforts in cultivating a supportive and engaging environment for us to embrace our interests in mathematics.

Emily: I extend my heartfelt gratitude to my math teacher, Dr. Elizabeth Often, for encouraging me to apply to PRIMES Circle and for supporting me in my application. Thank you so much for guiding my interests in math this year in both school and extracurricular endeavors.

Arya: I would like to sincerely thank my math teacher, Hallie Starkey, for encouraging my passion for math and always pushing me to think more creatively. Thank you for creating such an engaging and supportive learning environment and for always motivating me to challenge myself.

Hazel: I would like to express my gratitude to Dad, who always encouraged me to try new things and to stretch my capabilities in math and science. Without

him, it's doubtful that I would have made it through this semester. Thank you so much for always helping me when I needed it.

Finally, we would all like to thank our parents for their unwavering support throughout the PRIMES Circle semester. We are so incredibly grateful for the time and energy they dedicated to bringing us to and from meetings, and for their feedback and assistance along the way. Thank you for all the effort you have put in motivating and supporting us in pursuing our interests.