THE BASICS OF THE THEORY OF COMPUTATION

SHAMINI BIJU AND ZZ ZHANG

Mentored by Katherine Taylor

1. Introduction

This paper introduces topics about the different machines used to explore the theory of computation: Finite Automata, Context-Free Grammar, Pushdown Automata, and Turing Machines. Each of these machines is used as a tool to understand the capabilities and computations of computers by recognizing different classes of languages.

To understand the importance of languages, let us first define them. A language is a set of strings over some alphabet. The alphabet is a nonempty finite set of symbols. For example, a language could be $L = \{0^n | n \geq 0\}$, the alphabet would be $\{0\}$, and the symbol would be 0. Several strings in the language could be 0, 00, 000 (keep in mind that n doesn't signify exponents, instead, it is the number of times 0 repeats).

These languages are used in the theory of computation to represent problems by defining the input strings, allowing precision in describing problems. Then, a corresponding machine recognizes the language by accepting the strings in the language (denoted as L(name of the machine)), relating to the input and the solution of a problem. To put it more simply, the language of a machine is a set of strings that a machine "accepts." Finally, the languages that will be introduced in this paper are regular languages, context-free languages, Turing-recognizable languages, and Turing-decidable languages.

2. Finite Automata

To understand how computers process information, we first need to explore finite automata - simple yet powerful models of computation. These abstract machines take in strings of input and determine whether they follow specific rules, giving us insight into how computers handle tasks like language processing and search algorithms. Finite automata are foundational in fields such as programming languages, compilers, and text recognition. There are two main types: Deterministic Finite Automata (DFA) and Nondeterministic Finite Automata (NFA), each with a unique approach to reading and interpreting input, but both play a key role in how we model computation. In addition, we will be talking about regular expressions, which are used to describe languages.

2.1. Deterministic Finite Automata (DFA).

A Deterministic Finite Automaton (DFA) is a machine used to recognize patterns in strings of input. It consists of a finite set of states, transitions between those states, and rules that dictate how it moves based on input symbols. For each state

and input symbol, a DFA has exactly one transition to a new state, which makes its behavior entirely deterministic. A DFA that accepts binary strings ending in "01" has five states and moves deterministically based on the input symbols. For example, it accepts the string "1001" because the last two digits match the pattern "01". Let us take a look at an example.

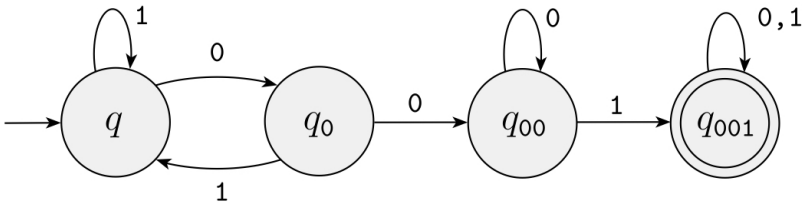


FIGURE 1

In Figure 1, we have the following components:

States: These represent the different positions the machine can be in. The DFA moves between these states based on the input it reads. For example,q, q_0, q_{00}, q_{001} are all states in Figure 1.

Transitions: The arrows show how the machine changes from one state to another in response to an input symbol. For each state and input symbol, there is exactly one possible transition. For example, the only way to transition from q to q_0 is by processing the input 0.

Start State: This is the state where the DFA begins its operation, typically marked with an arrow pointing toward it. This is q in Figure 1.

Accepting States: If the DFA ends in one of these states after reading the entire input, the input is accepted. If it ends in a non-accepting state, the input is rejected. These are denoted with two circles, as seen in q_{001} in Figure 1.

This diagram provides a visual way to understand how a DFA processes input and moves between states according to specific rules. Now that we've looked at the diagram and how the DFA works, let's dive into its formal definition.

Definition 2.1. A finite automata is a 5-tuple, which means it is defined by five components:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

(1) Q is a finite set of states,
(2) Σ is the input alphabet (the symbols the DFA can read),
(3) $\delta : Q \times \Sigma \to Q$ is the transition function, which tells the DFA how to move from one state to another,
(4) $q_0 \in Q$ is the start state, where the DFA begins, and
(5) $F \subseteq Q$ is the set of accepting states.

2.2. **Nondeterministic Finite Automata (NFA).** Following our discussion of DFAs, let us now explore nondeterministic finite automata (NFA), another type of finite automaton that introduces a bit more flexibility.

An NFA is similar to a DFA in that it processes an input string and moves through different states. However, unlike a DFA, an NFA can have multiple possible transitions from a state for a given input symbol, or it might not transition at all. This means that the machine can have multiple possible paths to follow, which introduces nondeterminism into its behavior. An NFA that accepts strings where the third symbol from the end is an "a" can use multiple paths to "guess" where that symbol occurs. For instance, it accepts "baaba" because one path leads to acceptance by checking the third-last character is "a".

Let's take a closer look at how an NFA works.

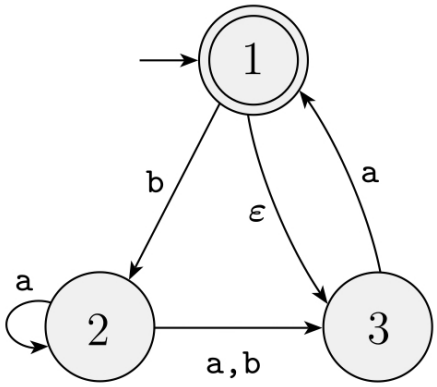


FIGURE 2. This is an example of a NFA.

In Figure 2, we have the following components:

States: These represent the different positions the machine can be in. The NFA moves between these states based on the input it reads, but unlike a DFA, there can be multiple possible transitions from a state for the same input symbol.

Transitions: The arrows show how the machine changes from one state to another in response to an input symbol. In an NFA, for each state and input symbol, there may be multiple possible transitions, or no transition at all. This introduces nondeterminism, meaning the machine can follow different paths at the same time.

Start State: This is the state where the NFA begins its operation, typically marked with an arrow pointing toward it.

Accepting States: If the NFA ends in one of these states after reading the entire input, the input is accepted. If it ends in a non-accepting state, the input is rejected. However, the NFA is

allowed to "choose" different possible paths, so even if one path is rejected, another path might still lead to acceptance.

This diagram provides a visual way to understand how an NFA processes input and moves between states with multiple possible transitions. Now that we've looked at the diagram and how the NFA works, let's dive into its formal definition.

Definition 2.2. A nondeterministic finite automaton (NFA) is also a 5-tuple, defined by five components:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

(1) Q is a finite set of states,
(2) Σ is the input alphabet (the symbols the NFA can read),
(3) $\delta : Q \times \Sigma\epsilon \to P(Q)$ is the transition function, which tells the NFA how to move from one state to another, allowing for multiple possible transitions (or none) for a given input symbol,
(4) $q_0 \in Q$ is the start state, where the NFA begins, and
(5) $F \subseteq Q$ is the set of accepting states.

2.3. **Regular Expressions.** Regular expressions are an important aspect of computer science applications because they give users of text applications the ability to search for strings with certain patterns, and regular expressions do a great job at describing these patterns. But before we dive into regular expressions, let's talk about regular operations, which are used to construct regular expressions. There are three regular operations that are used in the theory of computation as tools to study properties of the regular languages.

Definition 2.3. Let A and B be languages. We define the regular operations **union**, **concatenation**, and **star**:

(1) **Union:** $A \cup B = \{x | x \in A \text{ or } x \in B\}$
(2) **Concatenation:** $A \circ B = \{xy | x \in A \text{ and } y \in B\ \}$
(3) **Star:** $A^* = \{x_1 x_2 ... x_k | k \geq 0 \text{ and each } x_i \in A\}$

The union operation takes strings in both A and B and combines them to make one big language. The concatenation operation connects a string in language A with a string in language B following behind it. Lastly, the star operation focuses on one language and takes strings from that language to get a new language. It is important to note that the empty string ϵ is always a string in language A^*, no matter what A is.

Now let's move on to regular expressions. Similar to operations in arithmetic, we can use regular operations to build expressions describing languages: regular expressions. An example is:

$$(0 \cup 1)0^*$$

In this case, the parentheses act like concatenation between $(0 \cup 1)$ and 0^*. Additionally, regular operations have an order we have to follow, just like PEMDAS in arithmetic. In regular expressions, parentheses are first, then the star operation, then concatenation, and finally union.

Definition 2.4. Say that R is a **regular expression** if R is us

(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.

2.4. **NFAs and DFAs: Different, but Equivalent.** While DFAs and NFAs might look different on the surface, they're both types of finite automata—and surprisingly, they're equally powerful. Let's break down their main differences and how, in the end, they can do the exact same things.

States: Both DFAs and NFAs are made up of states that represent different steps in processing an input string. The machine moves through these states as it reads each symbol. In both cases, one of the states is marked as the *start*, and some are marked as *accepting*.

Pathways: Here's where things begin to split. In a **DFA**, for every state and input symbol, there is exactly one path to follow. It's like being on a road with one clear direction at every intersection. In contrast, **NFAs** allow multiple (or even zero) transitions for a given symbol and state. You can think of them as a book about choosing your own adventure: multiple possible paths can unfold, and the machine can "guess" which path might lead to acceptance.

Acceptance: A DFA accepts a string only if there is one unique path that leads to an accepting state. An NFA, on the other hand, accepts a string *if at least one* of its possible paths ends in an accepting state. So, as long as there's some way to reach an accepting state, the NFA says yes.

So what does this all mean in practice? Despite their structural differences, DFAs and NFAs are ultimately capable of recognizing the same types of languages. This brings us to one of the most fundamental ideas in automata theory:

Equivalence: Even though NFAs might seem more flexible or powerful because they can explore multiple paths at once, they're actually not more powerful than DFAs. In fact, for every NFA, there is a DFA that accepts exactly the same language. This important fact is what we mean when we say DFAs and NFAs are *equivalent*.

3. Context-Free Grammar and Pushdown Automata

Compared to finite automata, context-free grammars and pushdown automata are much more powerful in describing languages. Similar to regular expressions, context-free grammars are used to describe language syntax like Java and Python. Additionally, compilers use context-free grammars to parse code to check whether it's written correctly. While context-free grammars are used to define a language, pushdown automata are machines that recognize or accept the language defined by the context-free grammar.

3.1. **Context-Free Grammar (CFG).** Context-free grammars are used to describe context-free languages, which include regular languages and many more.

The following grammar is named G_1, consisting of substitution rules or productions.

$$\begin{aligned} A &\to 0A1 \\ A &\to B \\ B &\to \# \end{aligned}$$

FIGURE 3. Example of a context-free grammar.

Each line of a grammar represents a rule, it's made of a symbol and a string separated by an array. The symbol is called a **variable**, and the string often consists of variables and **terminals**, which are often represented as lowercase letters, numbers, or special symbols, while variables are capitalized. Additionally, the top left variable is assigned as the **start variable**. In this example, the start variable is A, A and B are the variables, and the terminals are 0, 1, and #. To describe a language, you generate strings of the language using a grammar using a process called a **derivation**. The process includes three steps:

(1) Start with the start variable.
(2) Find a variable that is written down and a rule that starts with that variable. Replace the written-down variable with the right-hand side of that rule.
(3) Repeat step 2 until no variables remain.

To depict a derivation visually, we can use a parse tree...

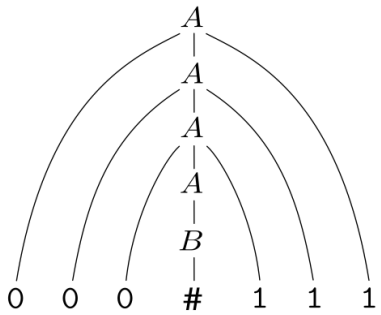


FIGURE 4. Parse tree for the derivation of string 000#111 in grammar G_1

while the derivation process of string 000#111 is

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

The language of a context-free grammar, denoted $L(G_1)$ for the grammar G_1, is constructed using all the strings generated through the process of derivation and is called a **context-free language**. Specifically, the language recognized by G_1 is represented by this: $L(G_1) = \{0^n\#1^n | n > 0\}$.

Definition 3.1. The formal definition of a context-free grammar is a 4-tuple:

$$(V, \Sigma, R, S),$$

where:

(1) V is a 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.

3.2. **Pushdown Automata (PDA).** Parallel to context-free grammars, pushdown automata also recognize context-free languages. However, pushdown automata are machines that read input and have states like a nondeterministic finite automata with an additional feature called the stack, making pushdown automata more powerful as it provides additional memory.

Here's a schematic drawing of a pushdown automata, it has features including state control (states and transition function), the tape (input string), and the arrow (pointing at the input symbol being read).

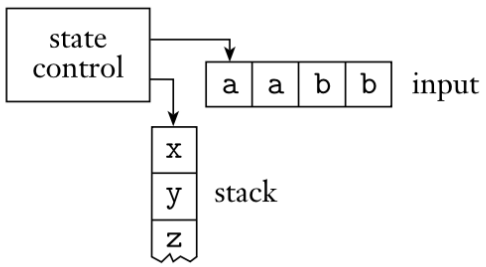


FIGURE 5. Schematic of a pushdown automaton

Now let's take a look at the state diagram of a PDA M_1 that recognizes the language $\{0^n1^n | n \geq 0\}$:

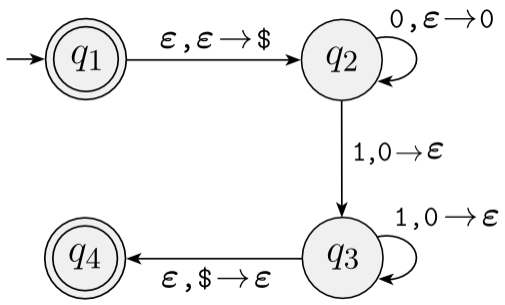


FIGURE 6. State diagram of PDA M_1

Each arrow in the state diagram has a corresponding rule that tells the PDA whether to push or pop a symbol in the stack (It's important to note that PDAs always push a special symbol $ before reading anything input). In this example, the PDA can recognize this language by:

Pushing a symbol onto the stack for every 0 read.

Popping a symbol from the stack for every 1 read.

Accepting if the stack reads the special symbol $ as it would mean that the stack is empty. Therefore, the number of 1s seen is the same as the number of 0s seen. Otherwise, **rejecting**.

Furthermore, it's important to understand that this language would not be recognized by a deterministic or nondeterministic finite automata as they have no memory to process whether the amount of 1s would be the same as the amount of 0s.

Definition 3.2. The formal definition of a PDA is similar to that of a finite automaton because they are essentially the same thing with the addition of a stack. Therefore, the definition of 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) $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \ P(Q \times \Gamma_\epsilon)$
(5) $q_0 \in Q$ is the start state, and
(6) $F \subseteq Q$ is the set of accept state.

3.3. **CFGs and PDAs: Different, but Equivalent.** Despite having very different formats, context-free grammars and pushdown automata are equivalent in power, meaning that they recognize the same class of languages: regular languages. Let's take a look at the proof of this statement.

Theorem 3.1. *A language L is context-free if and only if there exists a pushdown automaton M such that $L = L(M)$.*

When we use the statement "if and only if," it means that the theorem needs to be proved both ways.

First, let's construct a pushdown automaton from a context-free grammar. Given a CFG $G = (V, \Sigma, R, S)$, we construct a PDA $P = (\{q\}, \Sigma, V \cup \Sigma \cup \$, \delta, q, \$, \emptyset)$ that accepts empty stack.

- **States Q:** q as the PDA doesn't need to remember complex state information, the stack does most of the work.
- **Input alphabet** Σ is the same as the terminal alphabet of G.
- **Stack alphabet** Γ includes all non-terminals (V), all terminals (Σ), and the special bottom of the stack symbol ($).
- **Initial state** is q.
- **Transition function δ:** when the stack is empty, push the grammar's start symbol S onto the stack. If the top of the stack is a non-terminal A, the PDA can choose any production rule $A \to \beta$ from the grammar. It pops A and pushes β onto the stack in reverse order. If the top of the stack is a terminal symbol a, the PDA consumes the input a and pops a from the stack.
- **Accept state** is the empty set $\emptyset$.

The PDA M accepts a string ω if, after processing ω, the stack becomes empty. This means all non-terminals have been expanded and all terminals have been matched against the correct input.

Now we'll construct a context-free grammar from a pushdown automaton. Given a PDA M that accepts by empty stack, construct a CFG G such that $L(G) = L(M)$.

We assume M accepts by empty stack. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \emptyset)$, we construct a CFG $G = (V, \Sigma, R, S)$.

Let's identity the production rules: If the PDA, while in state q, reads input a (or ϵ), pops A, and moves to state p without pushing anything onto the stack: Then the non-termianl can directly generate the input a. If the PDA, while in state q, reads input a (or ϵ), pops A, pushes a sequence of symbols onto the stack, and moves to state p: Then the non-terminal can generate a followed by a sequence of other non-terminals.

Finally, if M accepts ω by empty stack, there is a sequence of transitions that starts in q_0, processes ω, and empties the stack. This entire computation can be broken down into segments, each corresponding to a non-terminal in the grammar. By combining these segments according to the grammar rules, a derivation for ω can be constructed.

If ω is generated by G, there is a derivation $S \Rightarrow \omega$. Each step in this derivation corresponds to a valid sequence of operations in M. Specifically, any non-terminal generating a substring x implies that M can go from state q to state p while popping A and reading x. Following the derivation from S will lead to a complete computation in M that accepts ω.

These constructions demonstrate the equivalence between context-free grammars and pushdown automata, meaning that any language that can be described by a CFG can also be recognized by a PDA, and vice versa.

4. Turing Machines

Turing Machines, designed by Alan Turing in the 1930s, are one of the most important models in the theory of computation, designed to capture the full range of what it means to compute something. The standard Turing Machine, which is deterministic, operates by reading and writing symbols on an infinite tape, one step at a time, following a fixed set of rules. Think of it as a super-powered automaton with memory.

But that's just the beginning. Over time, many variants have been introduced—some with multiple tapes working at once (multitape Turing Machines), others that allow the machine to explore many computational paths at the same time (nondeterministic Turing Machines), among many more. These variations may look different, but interestingly, they all turn out to be equivalent in what they can compute. We'll explore how and why as we go.

4.1. Standard Turing Machines.

A standard, or deterministic, Turing Machine has a tape that extends infinitely in both directions and serves as both input and memory. At any moment, it can read a symbol from the tape, write a new one, and move its read-write head left or right. This simple setup is powerful enough to simulate any algorithm, making it the foundation of modern computational theory. A Turing Machine can recognize the language $\{a^n b^n \mid n \geq 1\}$ by repeatedly crossing off one 'a' and one 'b'. For example, on input "aaabbb", it alternates between scanning for unmatched 'a's and 'b's until the tape is fully processed and accepts. Here is an example.

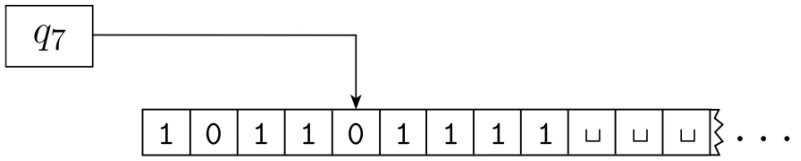


Figure 7. This is a standard Turing Machine.

In Figure 4, we can see a Turing machine in the middle of computation. The machine is currently in state $q7$, as indicated by the labeled box. Its tape—an infinite sequence of cells—holds a string of binary digits followed by blank symbols representing empty space. The arrow points to the second '0', which means the tape head is currently reading that cell. Based on the current state $q7$ and the symbol 0 under the head, the machine will consult its transition function to determine the next action: it could write a new symbol, move the tape head left or right, and switch to a new state. This visual captures the core mechanics of how a Turing machine processes input step by step.

Like DFAs and NFAs, Turing Machines have a formal definition.

Definition 4.1 (Turing Machine)**.** A *Turing machine* is a 7-tuple, which contains 7 components:

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

where Q, Σ, Γ are all finite sets and:

(1) Q is the set of states,
(2) Σ is the input alphabet not containing the *blank symbol* $\sqcup$,
(3) Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
(4) $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
(5) $q_0 \in Q$ is the start state,
(6) $q_{\text{accept}} \in Q$ is the accept state, and
(7) $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

4.2. **Variant Turing Machines.** Variant Turing Machines are modified versions of the standard Turing Machine, designed to explore different approaches to computation. These variants tweak the structure of the original machine to model different computational processes or make certain types of computation easier to do. In this section, we will focus on two key variants: **multitape** and **nondeterministic Turing Machines**. While both offer unique features, they do not fundamentally change the limits of what can be computed. Rather, they simply offer different ways to approach problem solving.

Multitape Turing Machines. A multitape Turing Machine is just like a regular Turing Machine, but it has more than one tape. Each tape has its own head that can read, write, and move independently. This makes it easier for the machine to work with different pieces of information at the same time.

For example, one tape could have the input, while another tape could hold the results of the machine's calculations. Even though it has multiple tapes, a multitape Turing Machine isn't any more powerful than a regular one. Instead, it just makes some problems easier to solve by speeding up the process. The figure below is an example of such a machine in comparison to a more standard Turing Machine.

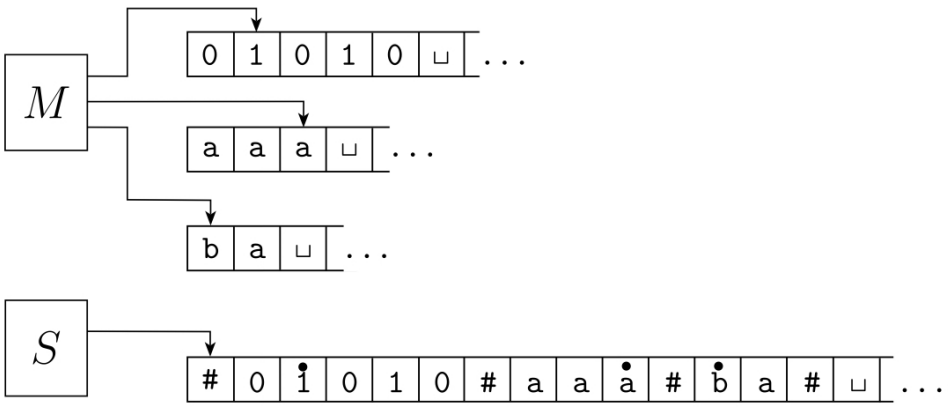


FIGURE 8. A Multitape Turing Machine in comparison to a Standard Turing Machine.

The top part of Figure 5 shows a multi-tape Turing machine, named M, with three separate tapes, each with its own read/write head.

Tape 1 contains the symbols 0 1 0 1 0 ⊔, with the head pointing at the second symbol, 1.
Tape 2 has the symbols a a a ⊔, with the head on the last a.
Tape 3 has b a ⊔, with the head on the b.
Each of these tapes works independently, allowing the machine to read from and write to all three at the same time. This setup makes the machine more efficient because it can perform multiple tasks in a single step. For example, you can read input from one tape, use data from another, and write to a third, all in parallel.

The figure clearly shows how the heads are positioned at different spots across the tapes, highlighting the flexibility and speed advantage of a multi-tape machine compared to the standard Turing Machine, named S, that only has a single tape and head.

Nondeterministic Turing Machines. While multitape machines give us more working space, nondeterministic Turing Machines change the way choices are made altogether.

Instead of following just one path like a regular Turing Machine, a nondeterministic Turing Machine can explore many possibilities at once. At any point during its computation, it can *branch out* into multiple different paths based on the input and transition rules. If **any** of those paths lead to an accepting state, the machine accepts the input. This idea doesn't make the machine more powerful in terms of what it can compute, but it opens up a new way of thinking about how efficiently we might solve certain problems.

Let's take a look at what this looks like in action.

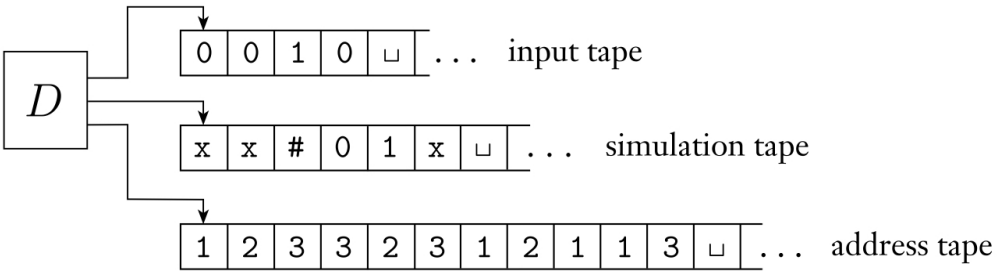
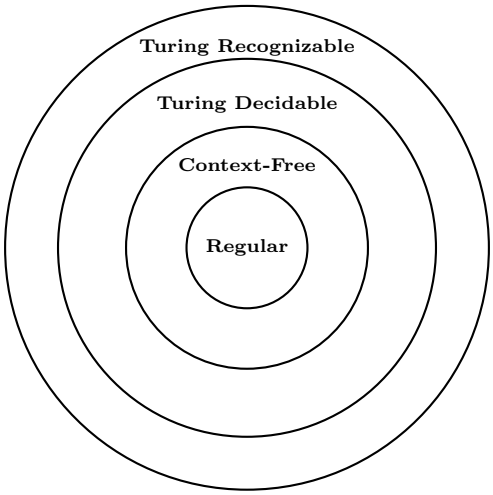


FIGURE 9. This is a Nondeterministic Turing Machine.

Figure 3.17 shows a nondeterministic Turing machine starting from one configuration at the top and branching into different paths below. Each line going down shows a possible move the machine can make at that step. As you go further down the tree, the machine keeps making choices, and each branch shows a different path it could follow. Some of the paths stop without accepting, but one of them reaches an accepting state, which is marked at the bottom right. Since the machine only needs one path to accept the input, this input would be accepted. The figure shows how a nondeterministic machine explores many paths at once, and acceptance happens if any one of them ends in success.

5. Relationship Between All Languages

Throughout this paper, we see how computational models vary in their ability. As we move from simpler to more powerful machines, each gains additional capabilities—such as memory and, eventually, infinite memory—which expand the class of languages they can recognize. The following nested diagram illustrates the hierarchical relationship among these language classes. Languages that fall outside this diagram are not recognized by any known computational model.



L is a **regular language** if and only if it is recognized by a DFA, NFA, or regular expression.
L is a **context-free language** if and only if it is recognized by a CFG, PDA
L is a **Turing-decidable language** if and only if it is recognized by a Turing machine that halts on every input
L is a **Turing-recognizable language** if and only if it is recognized by a Turing machine or its variants

References

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