Theory Of Computation

Muyao Xiao

Nov. 2024

Abstract

The note is taken by studying 18.404J/6.5400 *Theory of Computation* course by professor Michael Sipser of MIT. The course material can be downloaded in MIT OpenCourseWare. Meanwhile, most contents in this note will also be derived from his book *Introduction to the Theory of Computation, third edition.*

The course is divided into 2 parts, computational theory and complexity theory. Computational theory is developed during 1930s - 1950s. It concerns about what is computable. This note will be focused on the first part.

Example. Program verification, mathematical truth

Example (Models of Computation)**.** Finite automata, Turing machines, $\cdots$

Contents

Chapter 1

Introduction, Finite Automata, Regular Expressions

The theory of computation begins with a question: What is a computer. The real computer is too complicated to understand, to start with, we use an idealized computer called **computational model**.

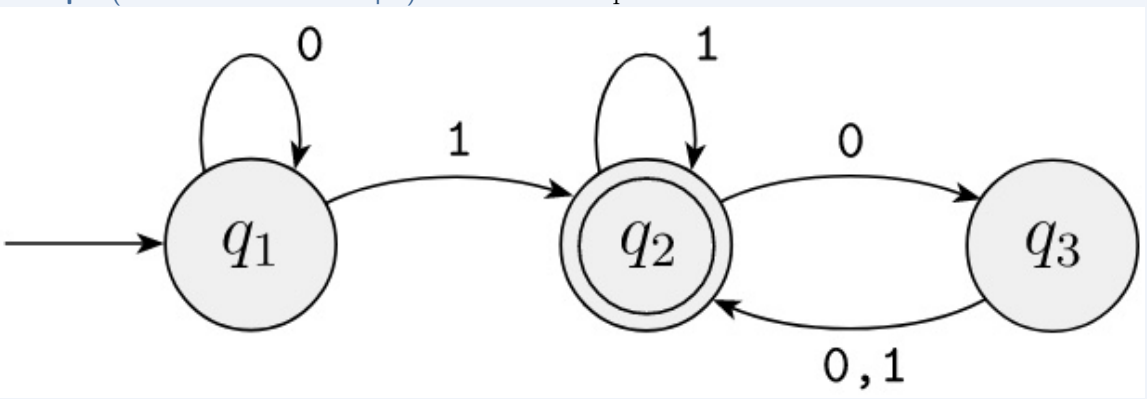
The simplest model among them is **finite state machine** or **finite automaton**.

1.1 Finite Automata

Finite automata are good models for computers with an extremely limited amount of memory.

Finite automata and their probabilistic counterpart **Markov chains** are useful tools when we're attempting to recognize patterns in data. Markov chains have even been used to model and predict price changes in financial markets.

Example (Finite Automata Example)**.** Here's an example of finite automata:



- The figure is called **state diagram** of M_1.
- Three **states**: q_1, q_2 and q_3.
- **Start state**: q_1.
- **Accept state**: q_2.
- The arrows going from one state to another are called **transitions**.

When the automaton receives an input string such as `1101`, it processes that string and produces an output. The output is either **accept** or **reject**:

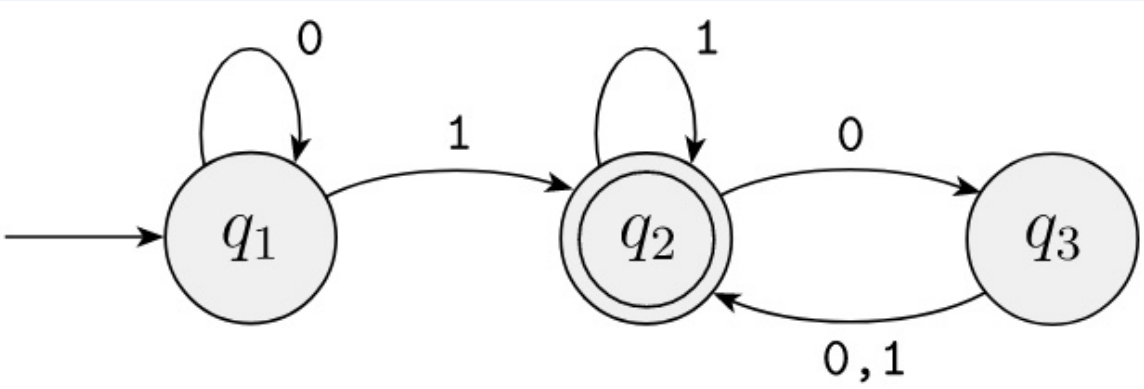
1. Start in state q_1
2. Read `1`, follow transition from q_1 to q_2

3. Read `1`, follow transition from q_2 to q_2
4. Read `0`, follow transition from q_2 to q_3
5. Read `1`, follow transition from q_3 to q_2
6. *Accept* because M_1 is in an accept state q_2 at the end of the input

Definition 1.1.1 (Formal Definition of A Finite Automaton)**.** A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called **state**
2. Σ is a finite set called the **alphabet**
3. $\delta : Q \times \Sigma \Rightarrow Q$ is the **transition function**
4. $q_0 \in Q$ is the **start state**
5. $F \subseteq Q$ is the **set of accept state**

Example (Revisit Finite Automata Example)**.** Let's revisit the finite automata example M_1 and see from the formal definition perspective:



We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$
2. $\Sigma = \{0, 1\}$
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state
5. $F = \{q_2\}$

If A is the set of all strings that machine M accepts, we say that A is the **language of machine M** and write $L(M) = A$. We say that **M recognizes A** or that **M accepts A**. Here because *accept* has different meaning, we use *recognize* for the language.

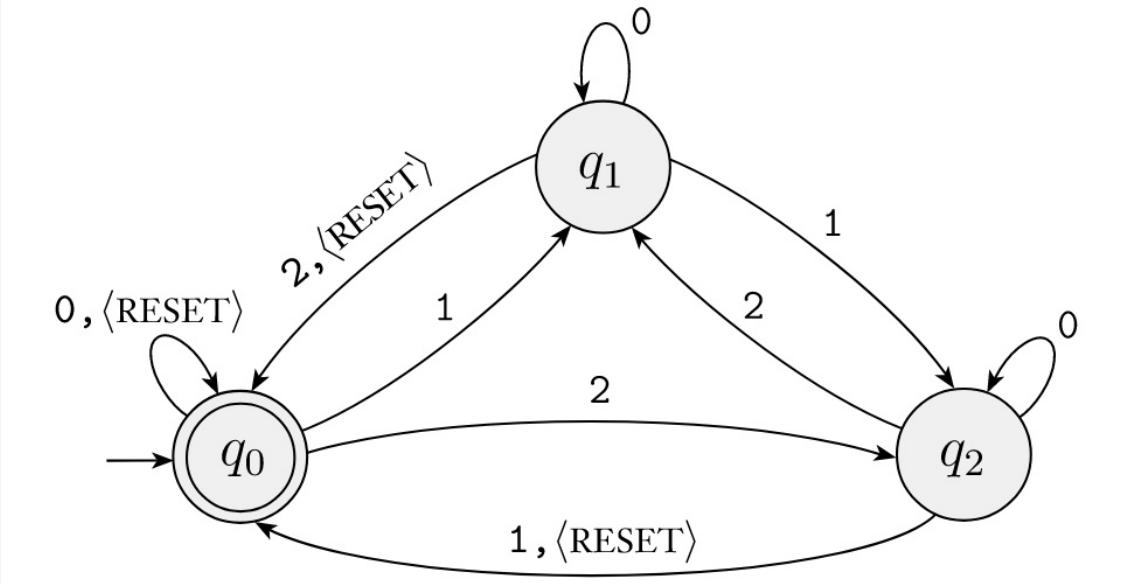
Remark. A machine may accept several strings, but it always recognizes only one language. If the machine accepts no strings, it still recognizes one language – namely, the empty language $\emptyset$.

Example (Revisit Finite Automata Example: Language)**.** In our example, the language set A can be represented as:

A = $\{\omega | \omega$ contains at least one `1` and an even number of `0`s follow the last `1`$\}$.

Then $L(M_1) = A$, or equivalently, M_1 recognizes A.

Example. When describing such a machine:



The alphabet $\Sigma = \{1, 2, 3, < RESET >\}$, we treat $< RESET >$ as a single symbol.

The machine keeps a running count of the sum of the numerical input symbols it reads, modulo 3. Every time it receives $< RESET >$ symbol, it resets the count to 0. It accepts if the sum is 0 modulo 3.

1.2 Formal Definition of Computation

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a FA and let $\omega = \omega_1\omega_2 \cdots \omega_n$ be a string where each ω_i is a member of alphabet Σ. Then M **accepts** ω if a sequence of state $r_0, r_1, \cdots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$ (machine starts at initial state)
2. $\delta(r_i, \omega_{i+1}) = r_{i+1}$ (machine goes form state to state following the transition function)
3. $r_n \in F$ (machine accepts its input if it ends up in an accept state)

We say that M recognizes language A if $A = \{\omega | Maccepts\omega\}$

Note. A is the language, ω is the accepted string. A is the set of all instances of ω.

We say a machine "accepts" a string, and a machine "recognizes" a language.

Definition 1.2.1 (Regular Language)**.** A language is called a **regular language** if some finite automaton recognizes it.

Example. Let B = { ω | ω has even number of 1s }

B is a regular language.

Example. Let C = { ω | ω has equal numbers of 0s and 1s }

C is <u>not</u> a regular language.

1.3 Regular Expressions

1.3.1 Regular Operations

Definition 1.3.1. Let A and B be languages, we define the regular operations **union** , **concatenation** , and **start** as follows:

- Union: $A \cup B = \{x | x \in A || x \in B\}$
- Concatenation: $A \circ B = \{xy | x \in A \& y \in B\}$
- Star: $A^* = \{x_1 x_2 \cdots x_k | k \geq 0 \& x_i \in A\}$

Notice that ϵ(empty language) always belongs to $A*$.

Example. $\Sigma^* 1$ is the language end with 1

Remark. Show finite automata equivalent to regular expressions.

1.3.2 Closure Properties

Theorem 1.3.1. The class of regular language is closed under the union operation.
In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

Proof. Let $M_1 = \{Q_1, \Sigma, \delta_1, q_1, F_1\}$ recognize A_1.
Let $M_2 = \{Q_2, \Sigma, \delta_2, q_2, F_2\}$ recognize A_2. (assuming in the same alphabet to make the proof simple)
Construct $M = (Q, \Sigma, \delta, q_0, F)$ recognizing $A_1 \cup A_2$.
M should accept input w if either M_1 or M_2 accepts w.
Component of M:

- $Q = Q_1 \times Q_2$
- $q_0 = (q_1, q_2)$
- $\delta((q, r), a) = (\delta_1(q, a), \delta(r, a))$
- $F = (F_1 \times Q_2) \cup (Q_1 \times F2)$
- not $F = \cancel{F_1 \times F2}$ (this gives intersection!)

■

Example (What is close?). Positive integers close under addition but not close under subtraction.

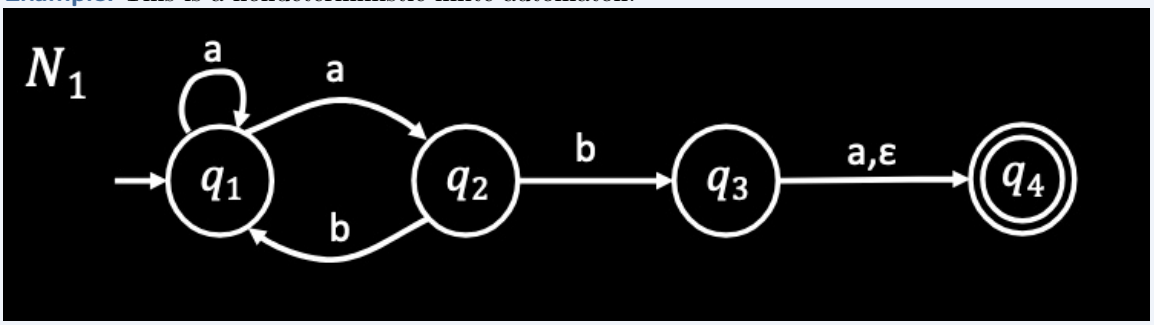
Theorem 1.3.2. The class of regular language is closed under the concatenation operation.
In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

Chapter 2

Nondeterminism, Closure Properties, Regular Expressions –> Finite Automata

2.1 Nondeterminism

Example. This is a nondeterministic finite automaton:



What's the difference with what we saw in the last lecture:

- in q_1, when accepting an a, you can either stay in q_1 or go to q_2
- in q_1, if getting b then there's nowhere to go
- ...

Example inputs

- `ab` (**accept**)
- `aa` (**reject**)

New features of nondeterminism:

- multiple paths possible (0, 1 or many at each step)
- ϵ-transition is a "free" move without reading input
- Accept input if some path leads to accept state (acceptance overrules rejection) (if one of possible ways to go accepts, then accepts)

Nondeterminism doesn't correspond to a physical machine we can build, however it is useful mathematically.

2.2 NFA

Definition 2.2.1. A **nondeterministic finite automaton** is a 5-tuple $Q, \Sigma, \delta, q_0, F$, where

1. Q is a finite set of states
2. Σ is a finite alphabet
3. $\delta : Q \times \Sigma_\epsilon \Rightarrow \mathsf{P}(Q)$ is the transition function
4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the set of accept states

In which, Σ_ϵ is a shorthand of $\Sigma \cup \{\epsilon\}$. P(Q) means the power set of Q which can be represented as $P(Q) = R | R \subseteq Q$, which is the set which contains all the subset of Q.

Example. Check the NFA example, we can write transition function such as:

- $\delta(q_1, a) = \{q_1, q_2\}$
- $\delta(q_1, b) = \emptyset$

Ways to think about nondeterminism:
Computational view: Fork new parallel thread and accept if any thread leads to an accept state.
Mathematical view: Tree with branches, accept if any branch leads to an accept state.
Magical: Guess at each nondeterministic step which way to go. Machine always makes the right guess that leads to accepting, if possible.

Theorem 2.2.1. If an NFA recognizes A then A is regular.

Proof. By showing how to convert an NFA to an equivalent DFA.
Let NFA $M = (Q, \Sigma, \delta, q_0, F)$ recognize A, we're going to construct DFA $M' = (Q', \Sigma, \delta', q_0', F')$ recognizing A.
IDEA: DFA M' keeps track of the subset of possible states in NFA M.
Construct of M':

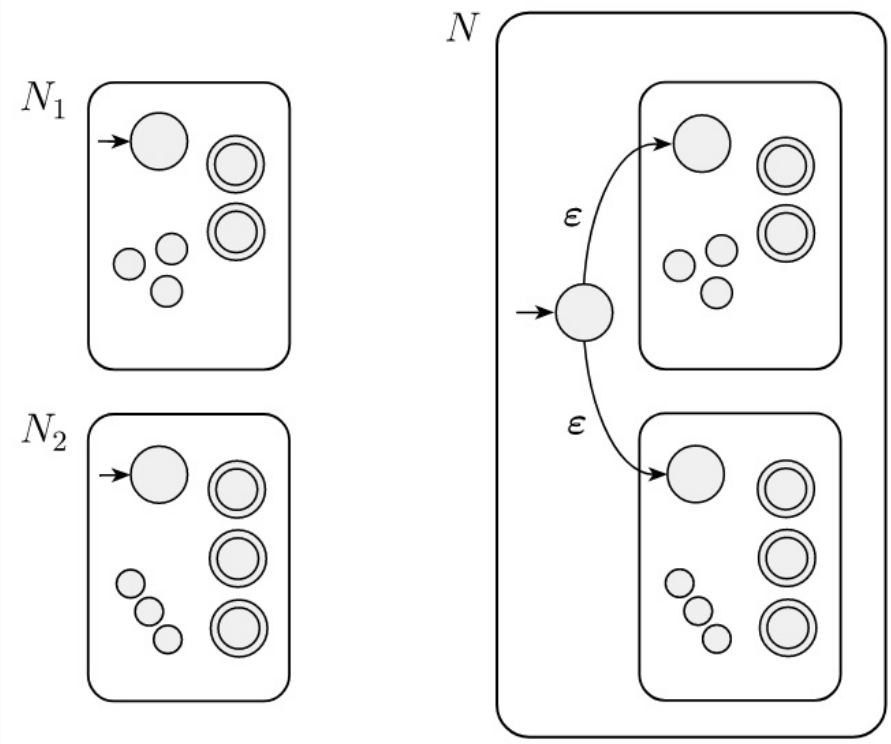
- $Q' = P(Q)$ (States of M' is the power set of Q)
- $\delta'(R, a) = \{q | q \in \delta(r, a) for\ some\ r \in R\}$ $(R \in Q')$
- $q_0' = \{q_0\}$
- $F' = \{R \in Q' | R\ intersects\ F\}$

■

Remark. If M has n states, how many states does M' have by this construction?
The answer is 2^n.

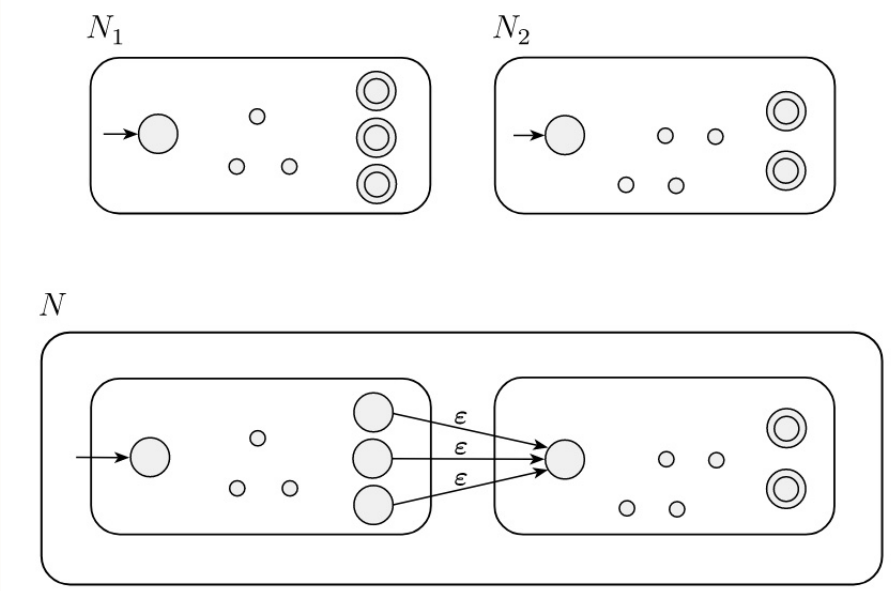
Return to Union Closure Property and Concatenation Closure Property by constructing an NFA to prove.

Proof: Closure Properties Union. N1 corresponds to the DFA M1 recognizes A_1, and N2 corresponds to an input of DFA M2 who recognizes A_2.
We can construct an NFA like following:



■

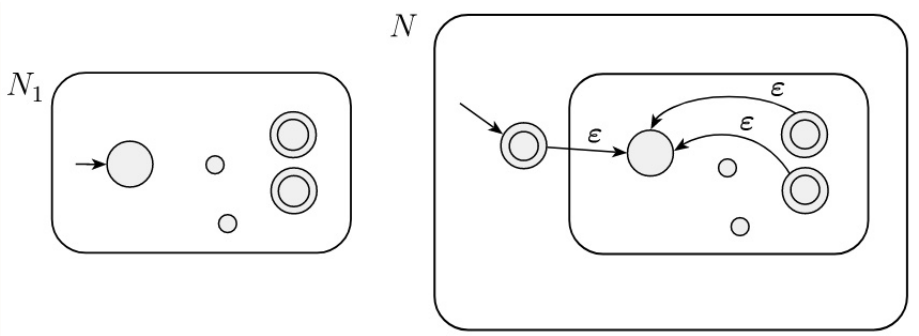
Proof: Closure Properties Concat. Similar with the last one, this one constructs an NFA recognizes A_1A_2:



■

Theorem 2.2.2 (Closure Under Star)**.** If A is a regular language, so is $A*$.

Proof. Given DFA M recognizing A, construct NFA M' recognizing A^*.



■

Remark. M' have n + 1 states in this construction.

Theorem 2.2.3. If R is regular expression and $A = L(R)$ then A is regular.
That is to say: A language is regular and only if some regular expression describes it.
This theorem has two directions:

Lemma 2.2.1. If a language is described by a regular expression, then it is regular

Proof. IDEA: Say that we have a regular expression R describing some language A. We show how to convert R into an NFA recognizing A.
When R is atomic, we can easily construct NFA for:

- R = a for some $a \in \Sigma$
- $R = \epsilon$
- $R = \emptyset$

When R is composite, as we have already constructed for them (closure properties).

Note. This proof works in a recursive way.

■

Lemma 2.2.2. If a language is regular, then it is described by a regular expression.

Proof. This will be proved in next lecture. ■

Note. Before going to prove this theorem, I need to make some clarifications here.

- **Regular language**: if some finite automaton recognizes a language, then it is called regular language.
- We say a machine recognizes a language if the language is the set contains all the string that can run the machine into an accept state.

Also we need the definition of **regular expression** here:

Definition 2.2.2. Say that 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
6. R_1^* where R_1 is a regular expression

Chapter 3

The Regular Pumping Lemma, Finite Automata → Regular Expressions, CFGs

3.1 GNFA

Let's go back to the Lemma in the previous lecture:

Theorem 3.1.1 (DFA -> Regular Expressions). If a language is regular, then it is described by a regular expression.

In another word, if A is regular then $A = L(R)$ for some regular exp R.

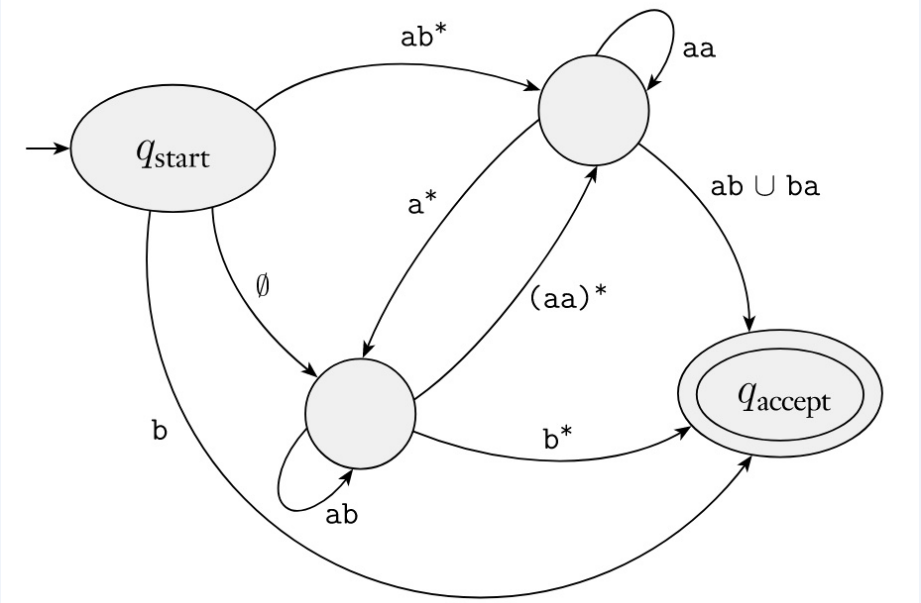
Proof. IDEA: Give conversion DFA $M \to R$ We need new tool: Generalized NFA ■

Definition 3.1.1 (Generalized NFA). A Generalized nondeterministic Finite Automaton(GNFA) is similar to an NFA, but allows regular expressions as transition labels.

The formal definition is that a **generalized nondeterministic finite automaton** is a 5-tuple, $(Q, \Sigma, \delta, q_{start}, q_{accept})$, where

1. Q is the finite set of states
2. Σ is the input alphabet
3. $\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \to \mathsf{R}$ is the transition function
4. q_{start} is the start state
5. q_{accept} is the accept state

Example (Example of GNFA). Similar with NFA but transitions allow regular expressions



Now we're going to covert GNFA to regular expressions to proof the Lemma.
For convenience, we will assume (we can easily modify the machine to achieve):

- One accept state, separate from the start state.
- One arrow from each state to each state, except
  - only exiting the start state
  - only entering the accept state

Lemma 3.1.1 (GNFA -> Regular Expressions)**.** Every GNFA G has an equivalent regular expression R.

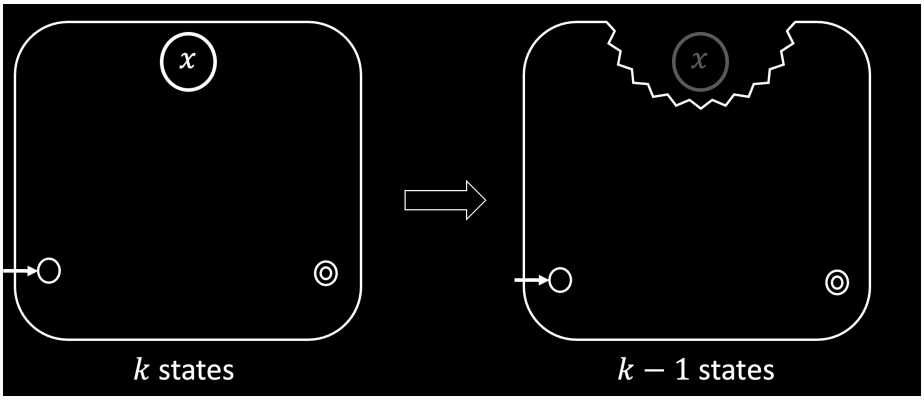
Proof. By induction(recursion) on the number of states k of G.

Basis($k = 2$):G can only looks like:
G = ->start –r–>accept
In this case, R = r.

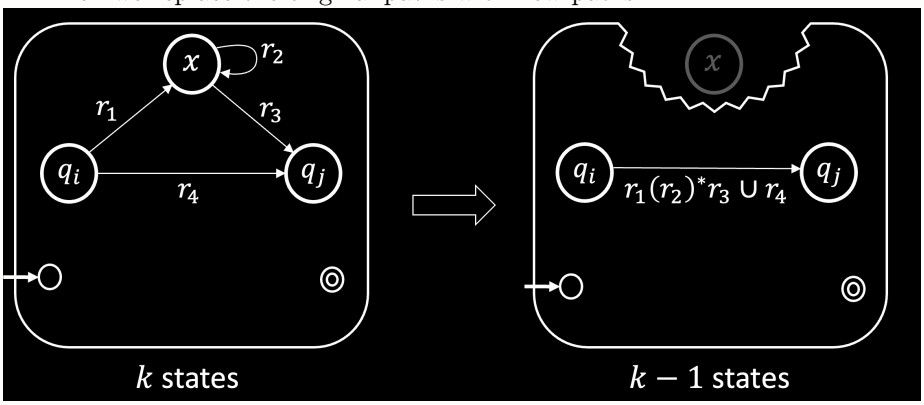
Induction step($k > 2$): Assume Lemma true for $k - 1$ states and prove for k states
IDEA: Convert k-state GNFA to equivalent $(k - 1)$-state GNFA

1. First we pick any state x except the start and accept states.
2. Remove x.
3. Repair the damage by recovering all paths that went through x.
4. Make the indicated change for each pair of states q_i, q_j.

In the following example, this is how we remove a state x:



Then we replace the original paths with new paths:



We're already done, because DFAs are a type of GNFAs. ■

3.2 Non-Regular Languages

How do we show a language is not regular?

- To show a language is regular we give a DFA.
- To show a language is not regular, we must give a proof (It is not enough to say that you couldn't find a DFA for it, therefore the language is not regular).

Example. Here $\Sigma = \{0, 1\}$

1. B has equal numbers of 0 and 1
   Intuition: B is not regular because DFAs cannot count unboundedly.
2. C has equal numbers of `01` and `10` substrings.
   `0101` $\notin$ C
   `0110` $\in$ C
   Intuition: C is not regular because DFAs cannot count unboundedly. (Wrong! Actually, C is regular!)

Moral: You need a proof.

3.3 Pumping Lemma

Pumping lemma is the method for proving non-regularity.

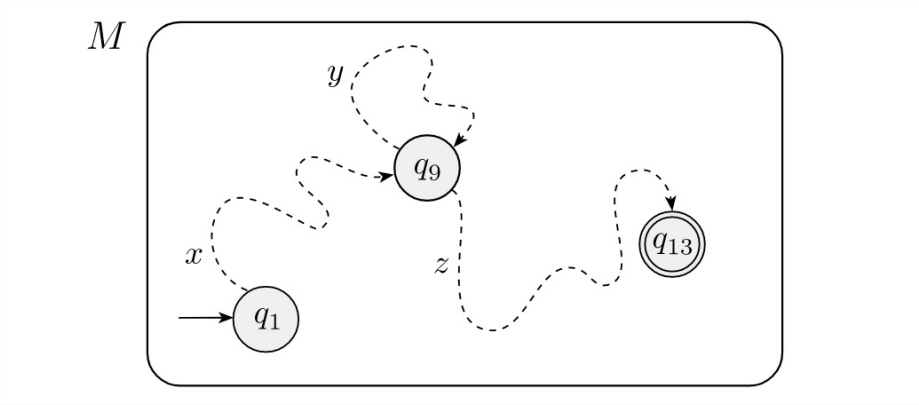
Lemma 3.3.1 (Pumping Lemma). For every regular language A, there's a number p (the "pumping length") such that if $s \in A$ and $|s| \geq p$ then $s = xyz$ where

1. $xy^iz \in A$ for all $i \geq 0$ $\quad y^i = yy\cdots y$(i number of y)
2. $y \neq \epsilon$
3. $|xy| \leq p$

Remark. $|s|$ means the length of the string s.

Informally: A is regular -> every long string in A can be pumped and the result stays in A.

Proof. Let DFA M recognize A. Let p be the number of states in M. Pick $s \in A$ where $|s| \geq p$.
When you have a too long string, you inevitably have to revisit the same state again and again, forming the struct as following:



This is also known as The Pigeonhole Principle. ■

Example (Prove Non-regularity). Let $D = \{0^k1^k | k \geq 0\}$

Proof. Show: D is not regular
Proof by Contradiction:
Assume that D is regular, applying pumping lemma, let $s = 0^p1^p \in D$.
Pumping lemma says that can divide $s = xyz$ satisfying the 3 conditions.
But $xyyz$ has excess 0s and thus $xyyz \notin D$ contradicting the pumping lemma. ■

Example. Let $F = \{ww | w \in \Sigma^*\}$, Say $\Sigma^* = \{0, 1\}$

Proof. Assume F is regular.
Try $s = 0^p10^p1 \in F$, show cannot be pumped $s = xyz$ satisfying the 3 conditions. ■

Variant: Combine closure properties with the pumping lemma.

Example. Let $B = \{w | w$ has equal number of 0s and 1s$\}$.

Proof. Assume that B is regular.
We know that 0^*1^* is regular so $B \cap 0^*1^*$ is regular (closure under intersection).
But $D = B \cap 0^*1^*$ and we already showed D is not regular. Contradiction! ■

3.4 Context Free Grammars

It is a stronger computation model.

Chapter 4

Pushdown Automata, CFG<–>PDA

4.1 CFG: Context Free Grammars

Using FA and regular expressions, some language such as $\{0^n1^n|n \geq 0\}$ can not be described.

Context-free grammars is a more powerful method of describing languages. Such grammar can describe certain features that have a recursive structure, which makes them useful in a variety of applications.

An important application of CFG occurs in the specification and compilation of programming languages. A number of methodologies facilitate the construction of a **parser** once a CFG is available. Some tools even automatically generate the parser from the grammar.

Example (CFG example)**.** The following CFG we call G_1:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

Shorthand:

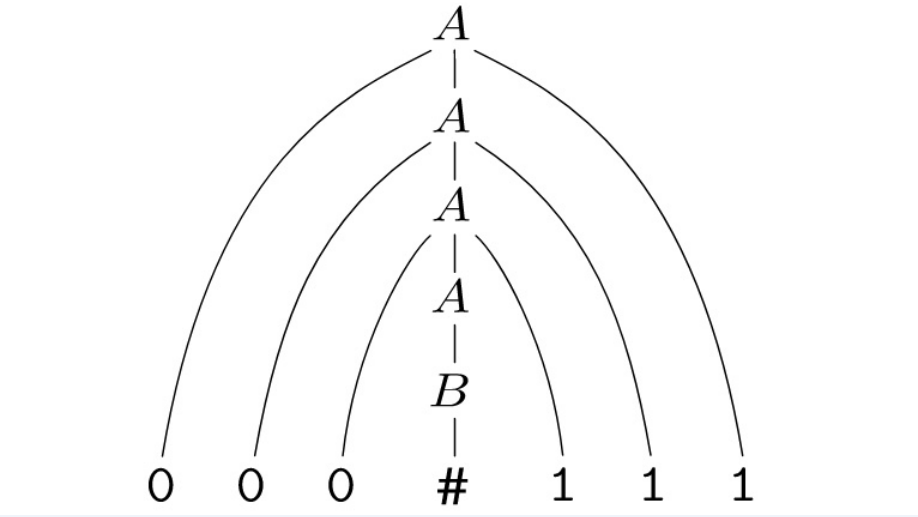
$$A \to 0A1|B$$
$$B \to \#$$

- It contains a collection of **substitution rules**, also called **productions**.
- Left side: **variable**, capital letters
- Right side: **terminals**, analogous to the input alphabet and often represented by lowercase letters, numbers or special symbols
- In this example, G_1's variables are A and B, its terminals are 0, 1 and #.

Example of using G_1 to generate string `000#111`:

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

The above information can be shown in a **parse tree**:



Definition 4.1.1 (CFG). A **context-free grammar** is a 4-tuple (V, Σ, 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 (rule form: $V \to (V \cup \Sigma)^*$)
4. $S \in V$ is the start variable

For $u, v \in (V \cup \Sigma)^*$ write

1. $u \Rightarrow v$ if can go from u to v with one substitution step in G.
2. $u \overset{*}{\Rightarrow} v$ if can go from u to v with some number of substitution steps in G.
   $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k = v$ is called a derivation of v from u.

Definition 4.1.2 (Context Free Language). Given $L(G) = \{w | w \in \Sigma^* \text{ and } S \overset{*}{\Rightarrow} w\}$
A is a Context Free Language(CFL) if $A = L(G)$ for some CFG G.

Example. We have 2 set of rules here:
C_1:

$$B \to 0B1|\epsilon$$
$$B1 \to 1B$$
$$0B \to B0$$

C_2:

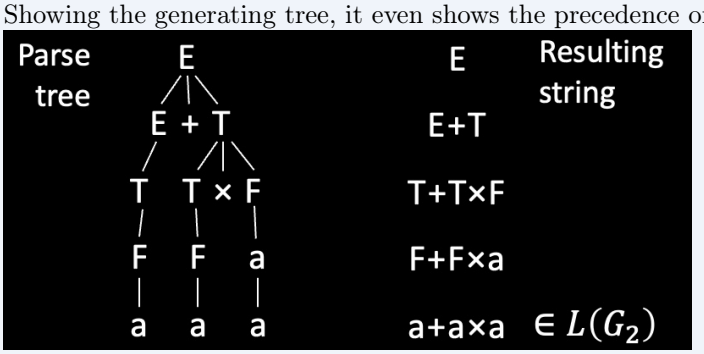
$$S \to 0S|S1$$
$$R \to RR$$

C_1 is not a CFG, because the left side is not pure variable, it has "context".
C_2, on the other hand, even though we can not derive a string with all terminals, but it does not violate the rule, so C_2 is a CFG.

Example (CFG). G_2:

$$E \to E + T | T$$
$$T \to T \times F | F$$
$$F \to (E) | a$$

$V = \{E, T, F\}$
$\Sigma = \{+, \times, (,), a\}$
$R =$ the 6 rules above
$S = E$
Showing the generating tree, it even shows the precedence of the operator $\times$ is higher than $+$:



If a string has 2 different parse trees then it is derived **ambiguously** and we say that the grammar is ambiguous.

Example (Ambiguity). G_2:

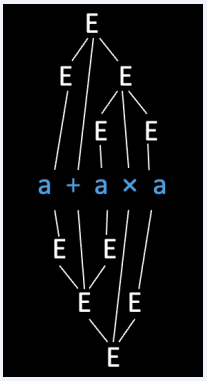
$$E \to E + T | T$$
$$T \to T \times F | F$$
$$F \to (E) | a$$

G_3

$$E \to E + E | E \times E | (E) | a$$

Both of them recognize the same language, $L(G_2) = L(G_3)$, however G_2 is an unambiguous CFG and G_3 is ambiguous.

This tree shows the ambiguity of G_3:



4.2 Pushdown Automata (PDA)

The **Pushdown Automaton** operates like an NFA except can write-add(push) or read-remove(pop) symbols from the top of stack.

Example (PDA). PDA for $D = \{0^k 1^k | k \geq 0\}$

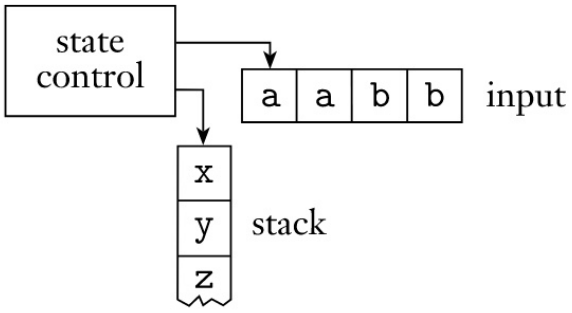


Figure 4.1: Schematic of a pushdown automaton

1. Read 0s from input, push onto stack until read 1.
2. Read 1s from input, while popping 0s from stack
3. Enter accept state if stack is empty (acceptance only at end of input)

Definition 4.2.1 (PDA). A **pushdown automaton** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, F are all finite sets:

1. Σ input alphabet
2. Γ stack alphabet
3. δ: $Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathsf{P}(Q \times \Gamma_\epsilon)$ ($\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$, we allow nondeterminism here)
   Left side is the domain(the set of possible inputs to a function) of the transition function; The current state, and the reading(popping) from the top of the stack and reading from the input (can read ϵ) decide the input of the transition function
   Right side is the writing, the combination of the state and the writing to the top of the stack (can write ϵ) decide the output of the transition
4. q_0: the start state
5. F: the accept states

For a transition function like $\delta(q, a, c) = \{(r_1, d), (r_2, e)\}$, the current state is q, and reading an input a and popping from the stack c, then we might end up going to:

- state r_1 and writing d to the top of stack
- state r_2 and writing e to the top of the stack

One interesting thing about this definition is that we have no primitive to decide whether the stack is empty at the end.

The reason for that is **we don't need to**! Because we can write special symbol to the bottom of the stack at the beginning when the machine starts.

Remark. Should do some example here to understand deeper into the PDA.

Theorem 4.2.1 (Equal of CFL and PDA). A language is context free if and only if some pushdown automaton recognize it.

Proof. We need to prove it in 2 directions:

Lemma 4.2.1 (CFL to PDA). If a language is context free, then some pushdown automaton recognize it.

and:

Lemma 4.2.2 (PDA to CFL). If a pushdown automaton recognize some language, then it is context free.

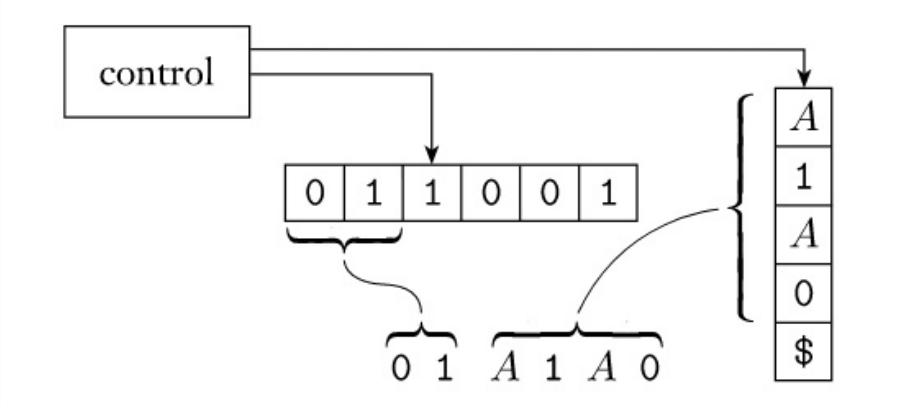
■

CFL to PDA. The idea is to use the stack to do partial substitution, and if the top of the stack matches the input string, we pop it out.

The reason that why we can always do the correct substitution is because of the nondeterminism nature of PDA, so actually we fork a lot of branches and there'll be the correct substitution in it.

Here's the informal description of the PDA P:

1. Place the marker symbol $ and the start variable on the stack.
2. Repeat the following steps forever:
   (a) If the top of the stack is a variable A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.
   (b) If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If match, repeat; if not, reject on this branch of the nondeterminism.
   (c) If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

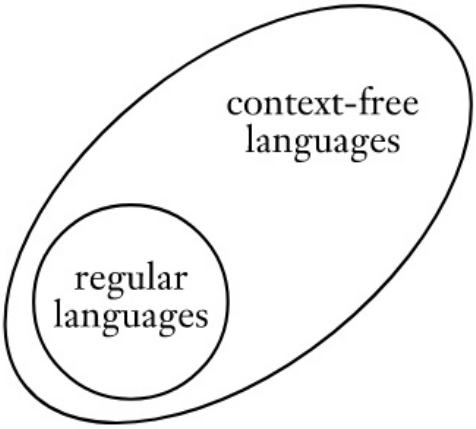


■

PDA to CFL. I will omit this proof as it's too complicated and not included in the course. But there's a proof for that in the book for LEMMA 2.27. ■

Because every finite automaton is automatically a pushdown automaton that simply ignores its stack, we have:

Corollary 4.2.1. Every regular language is context free.



4.2.1 Recap

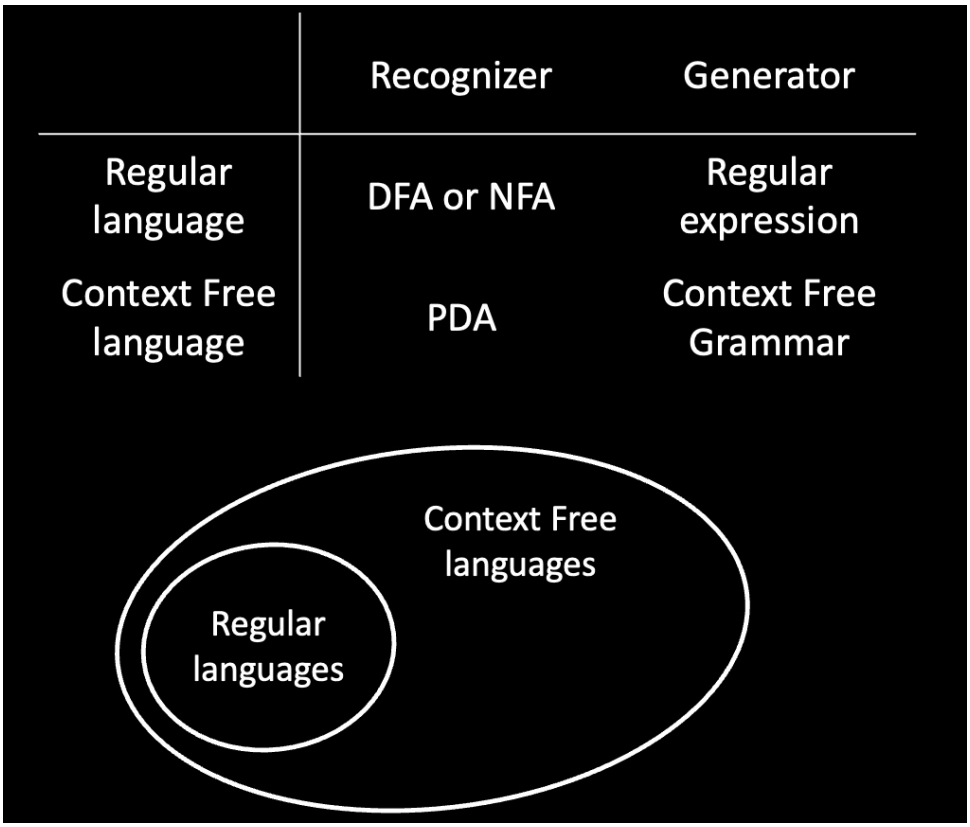


Figure 4.2: Recap until now

Chapter 5

The CF Pumping Lemma, Turing Machines

5.1 Non-context-free languages

Based on the equivalence of CFGs and PDAs, there are some more corollaries we can see or easy to prove:

Corollary 5.1.1. Every regular language is a CFL

Corollary 5.1.2. If A is a CFL and B is regular then $A \cap B$ is a CFL.

Example. A: $A = \{a^n b^n | n \geq 0\}$ B: $B = \{(ab)^*\}$

Proof sketch: while reading the input, the finite control of the PDA for A simulates the DFA for B.

Need to notice that the class of CFLs is not closed under $\cap$ which is different from regular languages.

Example. I will show an example where the intersection of 2 CFLs is not a CFL.
$L_1 = \{a^n b^n c^m | n, m \geq 0\}$
$L_2 = \{a^m b^n c^n | n, m \geq 0\}$
$L_1 \cap \mathsf{L2} = \{a^n b^n c^n | n \geq 0\}$
Intuitively, to recognize the intersection of L_1 and L_2 we need to compare the count of a, b, c. One stack can only handle the comparison of a and b, but we have no extra stack for comparing c.

But the class of CFLs is closed under $\cup, \circ, *$.

Note. Should try to prove its closed under union, concatenation and star!

5.1.1 CF Pumping Lemma

To prove a language is **not context free**, we need another tool:

Example. Let $B = \{0^k 1^k 2^k | k \geq 0\}$, we will show that B is not a CFL.

The tool we'll use is **Pumping Lemma for CFLs**:

Lemma 5.1.1 (Pumping Lemma for CFLs)**.** If A is a CFL, then there is a number p(the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$
2. $|vy| > 0$
3. $|vxy| \leq p$

Intuition. Proof IDEA: let s be a *very long* string in A, then the parsing tree for s must be very tall. Then the parse tree must contain some long path from the start variable to the terminal symbol at a leaf. On this long path, because of the pigeonhole principle, some variable R must repeat.

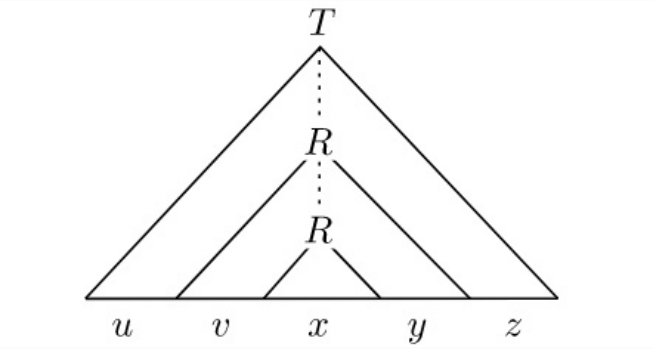


Figure 5.1: when $s = uvxyz$

This repetition allows us to replace the subtree under the second occurrence of R with the subtree under the first occurrence of R and still get a legal parse tree.

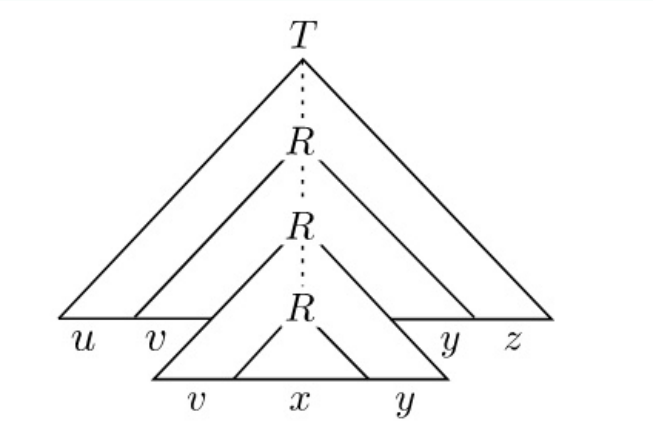


Figure 5.2: when $s = uv^ixy^iz$

Also v and y can repeat 0 time:

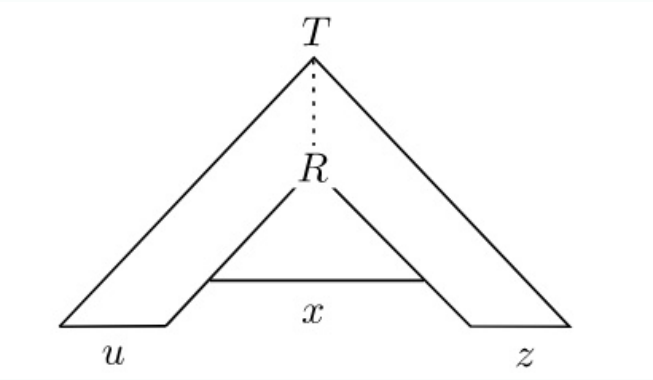


Figure 5.3: when $s = uxz$

■

Let's go back to the example:

Example. $B = \{0^k 1^k 2^k | k \geq 0\}$
Proof by Contradiction: Assume that B is a CFL.
Based on CFL pumping lemma gives p as above, Let $s = 0^p 1^p 2^p \in B$.
Pumping lemma says that can divide $s = uwxyz$ satisfying the 3 conditions.
Condition 3 ($|vxy| \leq p$) implies that vxy can not contain both 0s and 2s.
So uv^2xy^2z has unequal numbers of 0s, 1s and 2s.

Another example:

Example. Let $F = \{ww | w \in \Sigma^*\}$. $\Sigma = \{0, 1\}$.
Show: F is not a CFL.
Suppose F is a CFL, we try to construct a string which violate the CFL pumping lemma.

- Try $s_1 = 0^p 10^p 1$, we can choose $x = 1$ of the middle, and v and y the left and right 0, and we can pump to this string. → This is a bad choice to try to find a contradiction
- Try $s_2 = 0^p 1^p 0^p 1^p$, now we can find the contradiction!

5.2 Turing Machines(TMs)

Finite automata are good models for devices that have a small amount of memory. Pushdown automata are good models for devices that have an unlimited memory that is usable only in the last in, first out manner of stack.

We also have shown that some very simple tasks are beyond the capabilities of the models.

Alan Turing propose Turing Machine in 1936, it's similar to a finite machine but with an **unlimited and unstricted memory**. It is a much more accurate model of a general purpose computer. *A Turing machine can do anything that a real computer can do, but even a Turing machine cannot solve certain problems.*

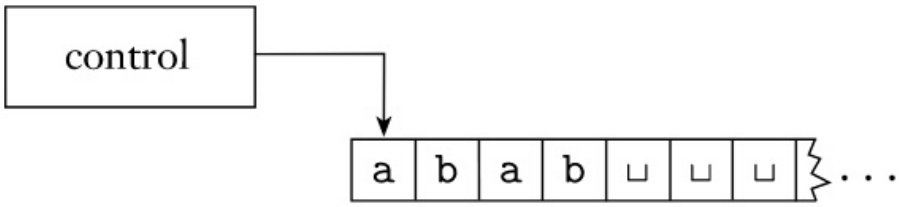


Figure 5.4: Schematic of a Turing machine

Important characteristics of it:

1. Head can read and write
2. Head is two way (can move left or right)
3. Tape is infinite (to the right)
4. Infinitely many blank follow input
5. Can accept or reject any time (not only at end of input)

Example. TM recognizing $B = a^k b^k c^k | k \geq 0$:
Example: aaabbbccc␣␣

1. Scan right until ␣ while checking if input is in $a^*b^*c^*$, *reject* if not

2. Return head to left
3. Scan right, crossing off single a, b and c: $\not{a}a\not{b}b\not{c}c$
4. If the last one of each symbol, *accept*
5. If the last one of some symbol but not others, *reject*
6. If all symbols remain, return to left and repeat from step 3.

After step 6, it will go back to step 3, and the string will become $\not{a}\not{a}a\not{b}\not{b}b\not{c}\not{c}c$. Then go to step 4, they are all last one of each symbol, so we *accept*.

The effect of "crossing off" is brought by use a tape alphabet like $\Gamma = \{a, b, c, \not{a}, \not{b}, \not{c}, \sqcup\}$.

Definition 5.2.1 (Turing Machine)**.** A Turing Machine (TM) is a 7-tuple $(Q, \Sigma, \Gamma, \sigma, q_0, q_{acc}, q_{rej})$.

- Σ input alphabet
- Γ tape alphabet ($\Sigma \subseteq \Gamma$)
- $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ (L = left, R = right)

The transition function means that the head in a certain state q and the head is over a tape square containing a symbol a. And if $\delta(q, a) = (r, b, L)$, the machine writes the symbol b replacing a, and goes to state r. The third L means the head moves to left after writing.

On input w a TM M may halt (enter q_{acc} or q_{rej}) or M may run forever ("loop").
So M has 3 possible outcomes for each input w:

1. Accept w (enter q_{acc})
2. Reject w by halting(enter q_{rej})
3. Reject w by looping (running forever)

The above definition of TM is **deterministic**, but we can change it to be **nondeterministic** by change δ to $\delta : Q \times \Gamma \to \mathsf{P}(Q \times \Gamma \times \{L, R\})$

5.2.1 TM recognizers and deciders

Definition 5.2.2. A is Turing-recognizable if $A = L(M)$ for some TM M.

Definition 5.2.3. TM M is a decider if M halts on all inputs. (no looping, meaning eventually the machine will halt to q_{acc} or q_{rej})

Definition 5.2.4. A is Turing-decidable if $A = L(M)$ for some TM decider M.

A Turing recognizable language can cause the machine reject by looping, but a Turing decidable language can always call the TM halt, thus we have the relationship:

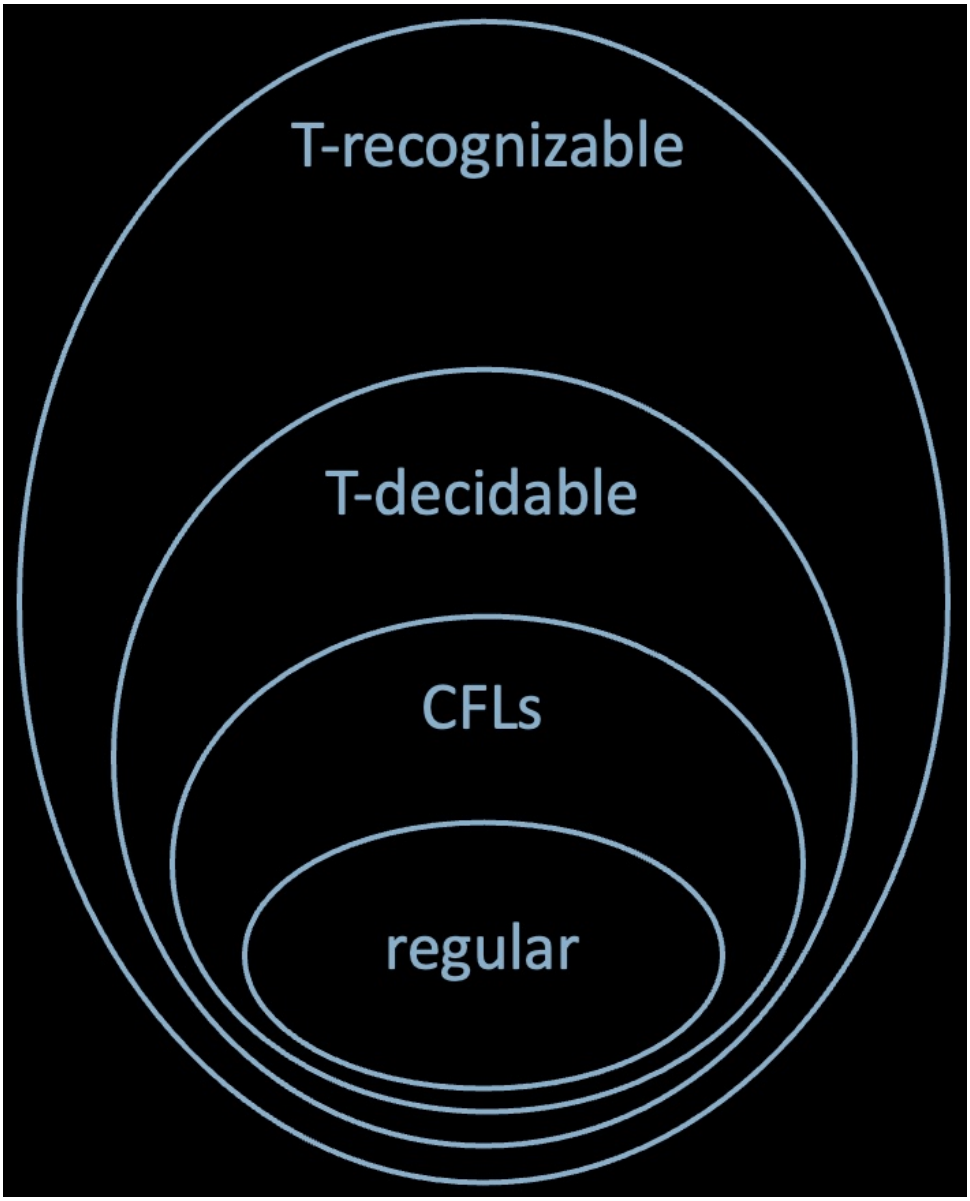


Figure 5.5: Summary of languages

Chapter 6

TM Variants, the Church-Turing Thesis

Note. What kind of models are suitable for general purpose computation?

6.1 Variants of the Turing Machine Model

Turing machine is picked as for the general purpose computation, but choice doesn't matter, all reasonable models are equivalent in power.

6.1.1 Multi-tape TMs

Simple Turing machine has one tape, but we can also have multiple tapes.

One tape is for input, other tapes are work tapes, initially blank. All tapes can be read and write.

Theorem 6.1.1. A is T-recognizable iff some multi-tape TM recognizes A.

Lemma 6.1.1. A is T-recognizable then some multi-tape TM recognizes A.

Proof. Proof. This is trivial. Just use one tape. ■

Lemma 6.1.2. Multi-tape TM recognized A is also single tape recognizable.

Proof. We can simulate multi-tape TM in single-tape Turing Machine:

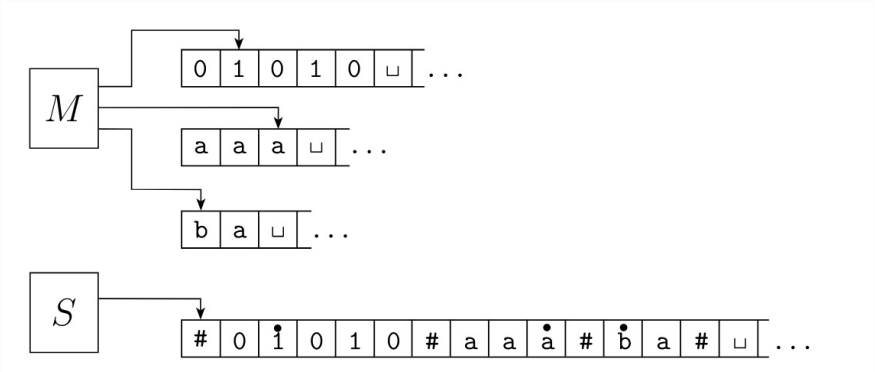


Figure 6.1: S simulates M

S simulates M by storing the contents of multiple tapes on a single tape in "block".
Record head positions with dotted symbols, something like: $b \to \dot{b}$
If M writes to originally blank space, S has to shift room as needed. ■

■

6.1.2 Nondeterministic TMs

A $NondeterministicTM$(NTM) is similar to a Deterministic TM except for its transition function $\delta : Q \times \Gamma \to \mathsf{P}(Q \times \Gamma\{L, R\})$.

Theorem 6.1.2. A is T-recognizable iff some NTM recognizes A.

Proof. (omit the trivial direction)

Lemma 6.1.3. convert NTM to deterministic TM.

The lecture and the textbook use 2 different ways to simulate, I write down the proof of the lecture as it is easier to understand.

Lecture Proof. The lecture simulates this using a single tape, just like the proof:

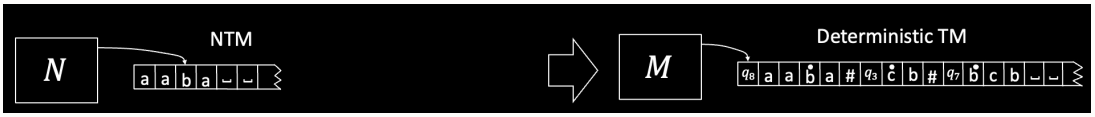


Figure 6.2: M simulates N

- M simulates N by storing each thread in separate "block"
- Also need to store the head location
- Need to store status of each thread
- If a thread forks, then M copies the block
- If a thread accepts then M accepts.

■

■

6.1.3 Enumerators

Definition 6.1.1 (informal). A Turing Enumerator is a deterministic TM with a printer. The Turing machine can use that printer as an output device to print strings. Every time the Turing machine wants to add a string to the list, it sends the string to the printer.

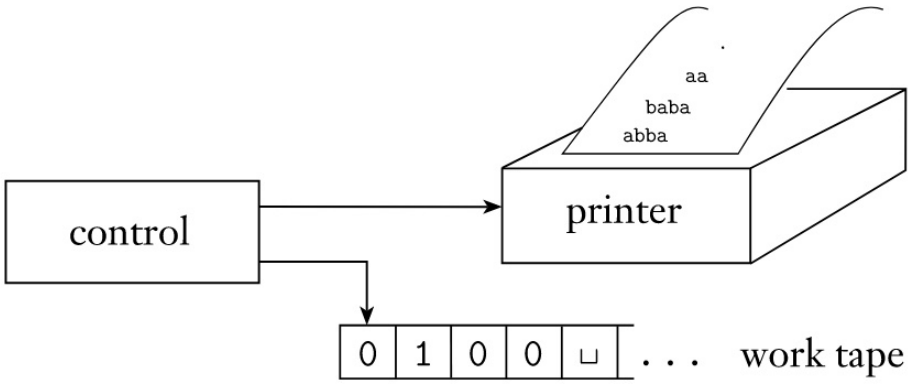


Figure 6.3: Schematic of an enumerator

The enumerator E starts with a blank input on its work tape. The language enumerated by E is the collection of all the strings that it eventually prints out.

Theorem 6.1.3. A is T-recognizable iff $A = L(E)$ for some T-enumerator E.

Lemma 6.1.4. Convert E to equivalent TM M: If we have an enumerator E that enumerates a language A, a TM M recognizes A.

Proof. Proof. The TM M works in the following way:
$M =$" On input w:

1. Run E. Every time that E outputs a string, compare it with w.
2. If w ever appears in the output of E, *accept*."

Clearly, M accepts those strings that appear on E's list. ■

Lemma 6.1.5. If TM M recognizes a language A, we can construct the following enumerator E for A.

Proof. Say that $s_1, s_2, s_3, \cdots$ is a list of all possible strings in Σ^*.
$E =$ "Ignore the input.

1. Repeat the following for $i = 1, 2, 3, \cdots$
2. Run M for i steps on each input, $s_1, s_2, \cdots, s_i$
3. If any computations accept, print out the corresponding s_j"

Intuition. This is basically the E uses the TM it attaches to recognize the language, and print if the attached TM recognizes it.
It can be an infinite loop because all possible strings in Σ^* is infinite if it does not only contain ϵ.
But when really run the TM, we have to run in parallel, as the TM might be looping.

■

■

The essential feature of TM is *unrestricted access to unlimited memory*, it distinguishes TMs from other weak models. Some other models also have the same power as TMs, and they also share the same feature.

6.2 Church-Turing Thesis

Remark. *Intuitive notion of algorithms* equals *Turing machine algorithms*

Church used a notational system called the $\lambda-$calculus to define algorithms. Turing did it with his "machines". The two definitions where shown to be equivalent. The connection between the informal notion of algorithm and the precise definition has come to be called ***Church-Turing thesis***.
It has been used by Yuri Matijasevi to solve Hilbert's 10th problem:

> to provide a general algorithm that, for any given Diophantine equation (a polynomial equation with integer coefficients and a finite number of unknowns), can decide whether the equation has a solution with all unknowns taking integer values.(Wikipeadia: Hilbert's tenth problem)

Note. The idea is to prove the language of Diophantine equations is TM recognizable but not decidable.

6.3 Notation for encodings and TMs

We're going to accept more types (even automaton) as input to Turing machine. But TM only accepts strings, so we need to encode objects into strings.

- If O is some object, we write $\langle O \rangle$ to be an encoding of that object into a string.

- If $O_1, O_2, \cdots, O_k$ is a list of objects then we write $\langle O_1, O_2, \cdots, O_k \rangle$ to be an encoding of them together into a single string.

Chapter 7

Decision Problems for Automata and Grammars

Chapter 8

Undecidability

Chapter 9

Reducibility

Chapter 10

The Computation History Method

Chapter 11

The Recursion Theorem and Logic