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

1 Introduction, Finite Automata, Regular Expressions 2

1.1 Finite Automata 2

1.2 Formal Definition of Computation 4

1.3 Regular Expressions 5

2 Nondeterminism, Closure Properties, Regular Expressions −> Finite Automata 6

2.1 Nondeterminism 6

2.2 NFA 6

3 The Regular Pumping Lemma, Finite Automata → Regular Expressions, CFGs 11

3.1 GNFA 11

3.2 Non-Regular Languages 12

3.3 Pumping Lemma 13

3.4 Context Free Grammars 14

4 Pushdown Automata, CFG<−>PDA 15

4.1 CFG: Context Free Grammars 15

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**.

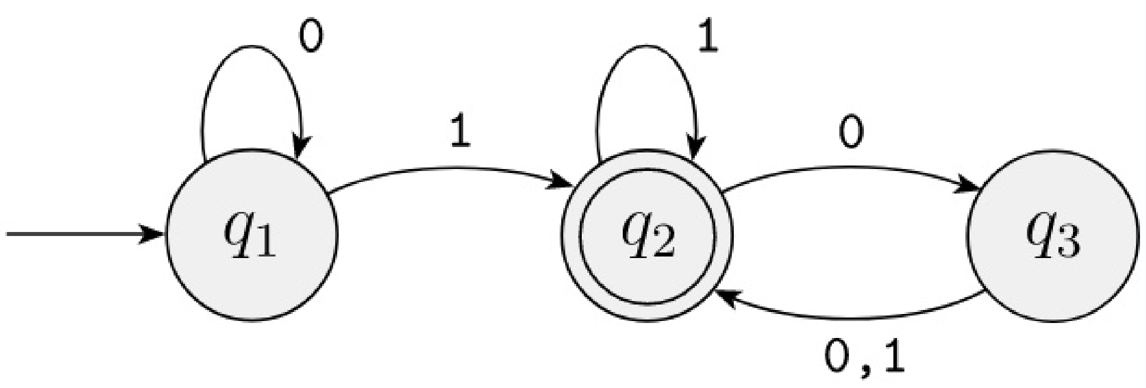
The simplest model among them is **finite state machine** or **finite automaton**.

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**:

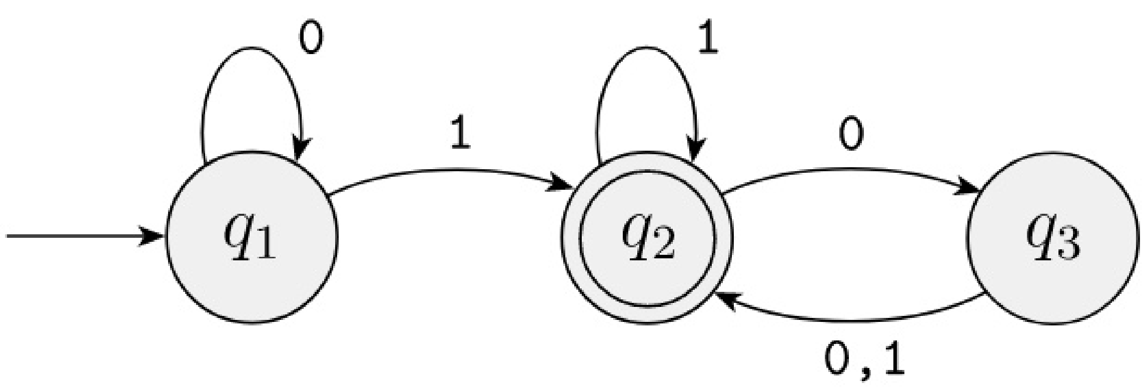
1. Start in state q_1
2. Read `1`, follow transition from q_1 to q_2

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.

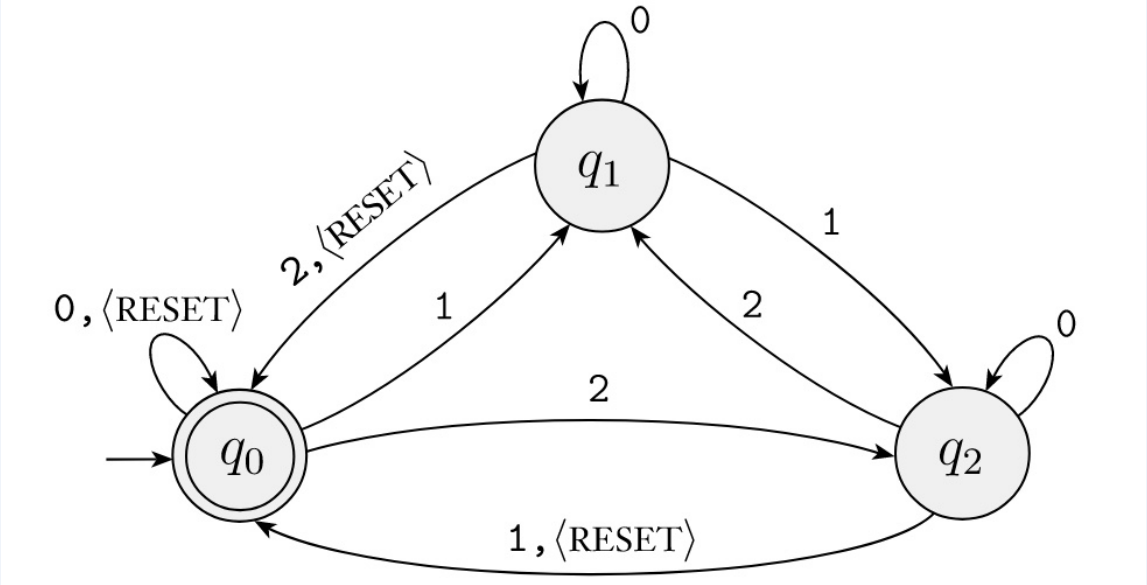
Remark. A machine may accept several strings, but it always recognizes only one language. If the machine accepts no strings, it still recognizes one language – namely, the empty language $\emptyset$.

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 0s 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 $A = \{\omega | M accepts \omega\}$

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. Σ^*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.

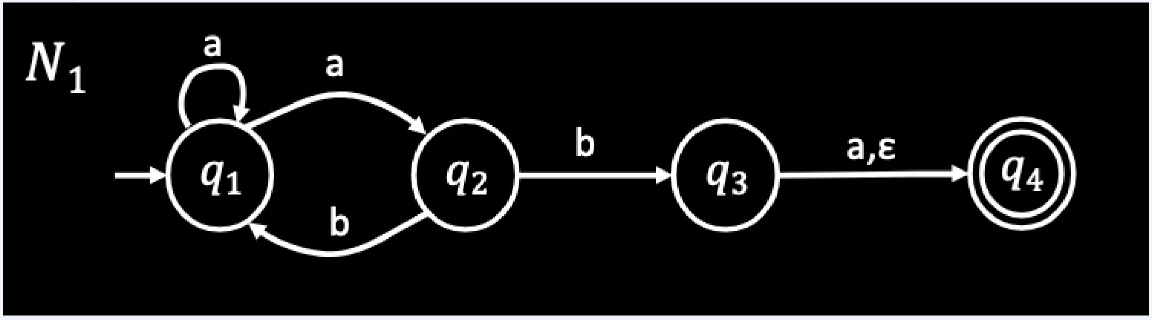
Theorem 1.3.2. The class of regular language is closed under the concatenation operation.
In other words, if A_1 and A_2 are regular languages then so is $A_1 \circ A_2$.

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)
- $\delta'(R, a) = \{q | q \in \delta(r, a) for\ some\ r \in R\}$ $(R \in Q')$
- $q_0' = \{q_0\}$
- $F' = \{R \in Q' | R\ intersects\ F\}$

■

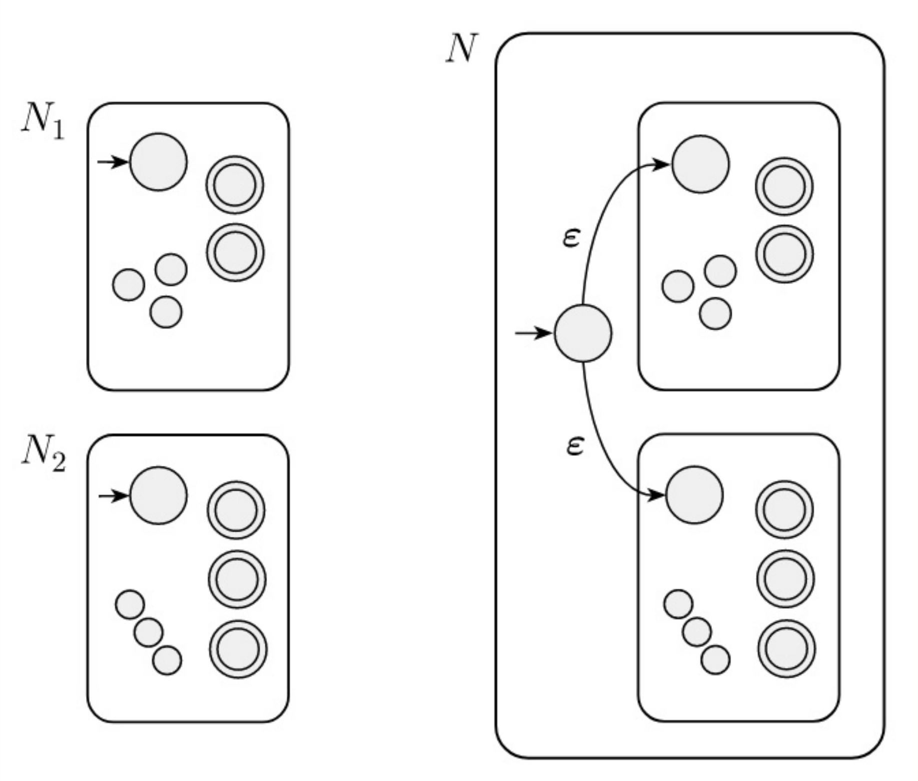
Remark. If M has n states, how many states does M' have by this construction?

The answer is 2^n.

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

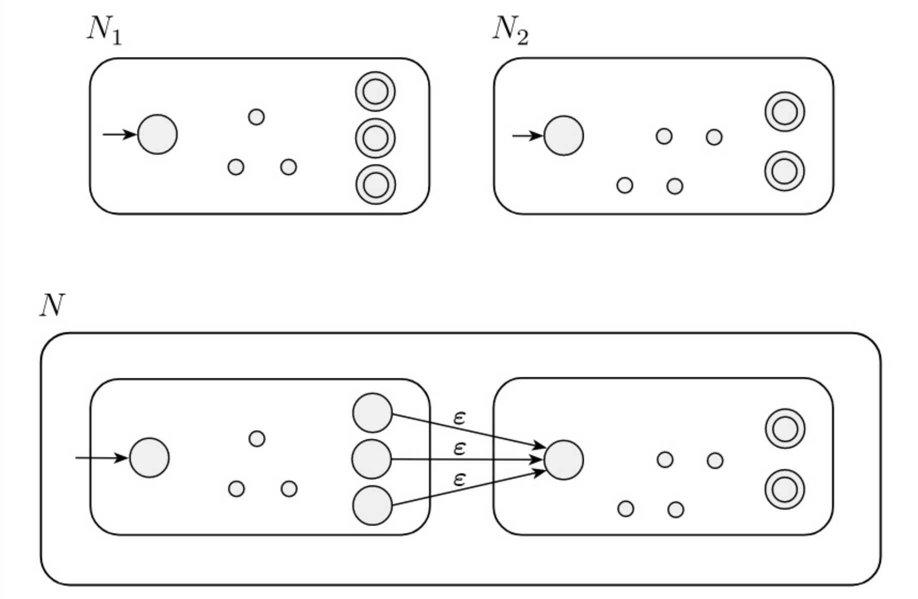
Proof: Closure Properties Union. N1 corresponds to the DFA M1 recognizes A_1, and N2 corresponds to an input of DFA M2 who recognizes A_2.

We can construct an NFA like following:



■

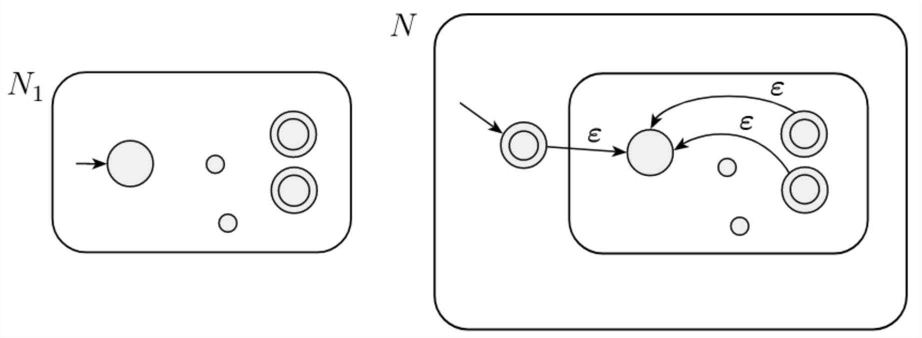
Proof: Closure Properties Concat. Similar with the last one, this one constructs an NFA recognizes A_1A_2:



■

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:

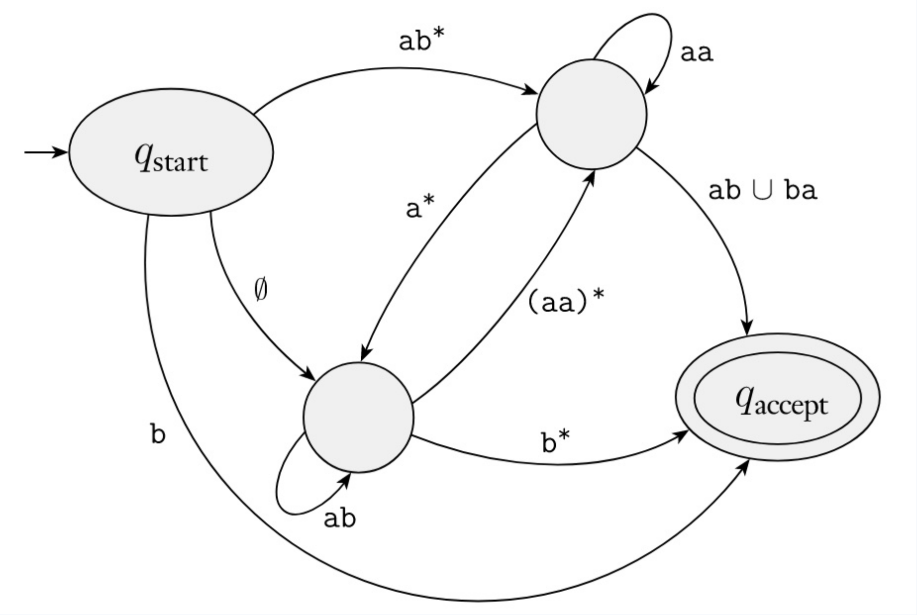
Theorem 3.1.1 (DFA -> Regular Expressions)**.** If a language is regular, then it is described by a regular expression.

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.

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.

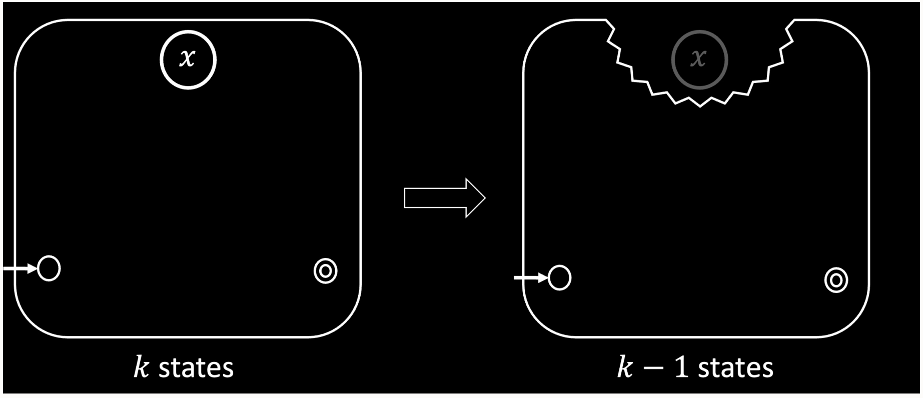
Proof. By induction(recursion) on the number of states k of G.

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

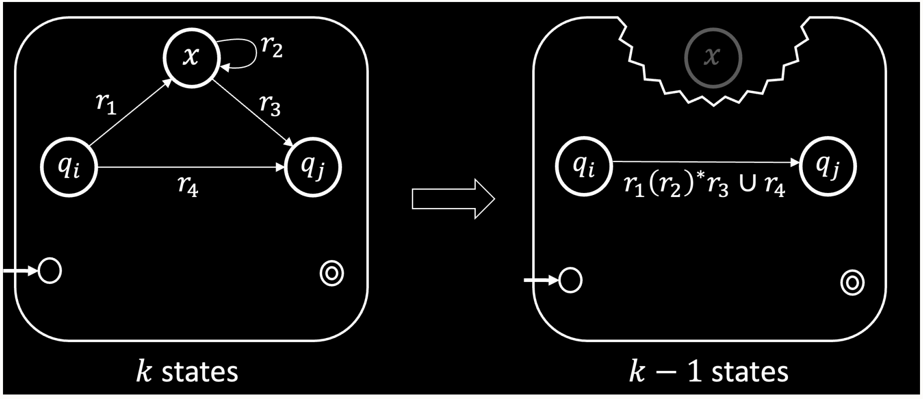
Induction step($k > 2$): Assume Lemma true for $k - 1$ states and prove for k states
IDEA: Convert k-state GNFA to equivalent $(k-1)$-state GNFA

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

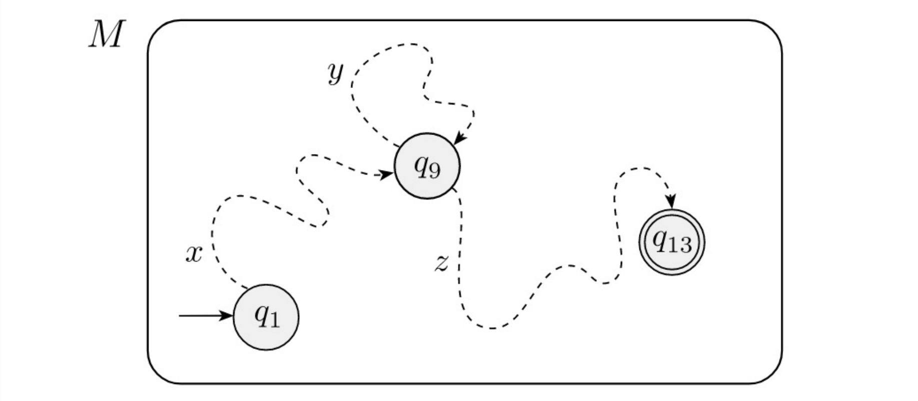
1. $xy^iz \in A$ for all $i \geq 0$ $\quad y^i = yy \cdots y$(i number of y)
2. $y \neq \epsilon$
3. $|xy| \leq p$

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 $s = 0^p 1^p \in D$.

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 $s = 0^p 1 0^p 1 \in F$, 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:

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

Shorthand:

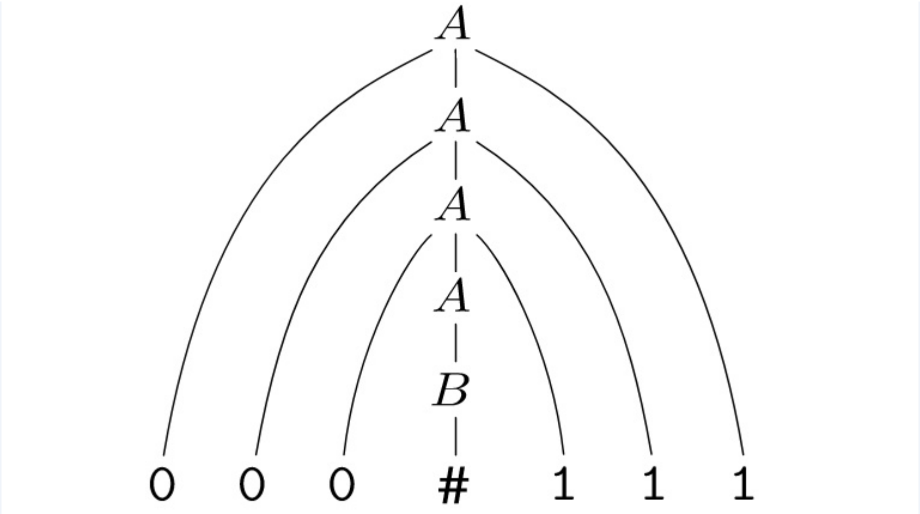
$$\begin{aligned} A &\rightarrow 0A1|B \\ B &\rightarrow \# \end{aligned}$$

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