Automata theory

What is automata theory

- Automata theory is the study of abstract computational devices
- Abstract devices are (simplified) models of real computations
- Computations happen everywhere: On your laptop, on your cell phone, in nature, …
- Why do we need abstract models?

What does Automata mean?

- It is the plural of automaton, and it means "something that works automatically".
- Automata theory is the study of abstract computational devices and the computational problems that can be solved using them.
- Abstract devices are (simplified) models of real computations.

- Helps in design and construction of different software's and what we can expect from our software's.
- Automata play a major role in theory of computation, compiler design, artificial intelligence.

Introduction to languages

Kinds of languages:

– Talking language
– Programming language
– Formal Languages (Syntactic languages)

EMPTY STRING or NULL STRING

- Sometimes a string with no symbol at all is used, denoted by (Small Greek letter Lambda) λ or (Capital Greek letter Lambda) Λ, is called an empty string or null string.

- The capital lambda will mostly be used to denote the empty string, in further discussion.

Words

- Definition:

  Words are strings belonging to some language.

- Example:

  If Σ= {a} then a language L can be defined as
  L={a^n : n=1,2,3,…..} or L={a,aa,aaa,….}

Here a,aa,… are the words of L

All words are strings, but not all strings are words

These devices can model many things

- They can describe the operation of any "small computer", like the control component of an alarm clock or a microwave
- They are also used in lexical analyzers to recognize well formed expressions in programming languages:

`ab1` is a legal name of a variable in C
`5u=` is not

Some devices we will see

finite automata	Devices with a finite amount of memory. Used to model "small" computers.
push-down automata	Devices with infinite memory that can be accessed in a restricted way. Used to model parsers, etc.
Turing Machines	Devices with infinite memory. Used to model any computer.
time-bounded Turing Machines	Infinite memory, but bounded running time. Used to model any computer program that runs in a "reasonable" amount of time.

Alphabets and strings

- A common way to talk about words, number, pairs of words, etc. is by representing them as strings
- To define strings, we start with an alphabet

An alphabet is a finite set of symbols.

- Examples

$\Sigma_1 = \{a, b, c, d, \ldots, z\}$: the set of letters in English

$\Sigma_2 = \{0, 1, \ldots, 9\}$: the set of (base 10) digits

$\Sigma_3 = \{a, b, \ldots, z, \#\}$: the set of letters plus the special symbol #

$\Sigma_4 = \{\, (, \,) \,\}$: the set of open and closed brackets

Strings

A string over alphabet Σ is a finite sequence of symbols in Σ.

- The empty string will be denoted by ε
- Examples

abfbz is a string over $\Sigma_1 = \{a, b, c, d, \ldots, z\}$

9021 is a string over $\Sigma_2 = \{0, 1, \ldots, 9\}$

$\mathrm{ab\#bc}$ is a string over $\Sigma_3 = \{a, b, \ldots, z, \#\}$

$))()(()$ is a string over $\Sigma_4 = \{\ (,\ )\ \}$

Languages

A language is a set of strings over an alphabet.

- Languages can be used to describe problems with "yes/no" answers, for example:

$L_1 =$ The set of all strings over Σ_1 that contain the substring "fool"

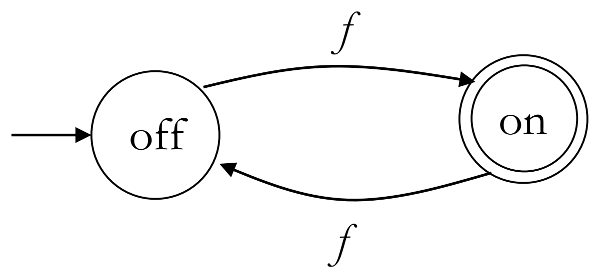
$L_2 =$ The set of all strings over Σ_2 that are divisible by 7 $= \{7, 14, 21, \ldots\}$

$L_3 =$ The set of all strings of the form $\mathrm{s\#s}$ where s is any string over $\{\mathrm{a, b, \ldots, z}\}$

$L_4 =$ The set of all strings over Σ_4 where every (can be matched with a subsequent)

Finite Automata

Example of a finite automaton

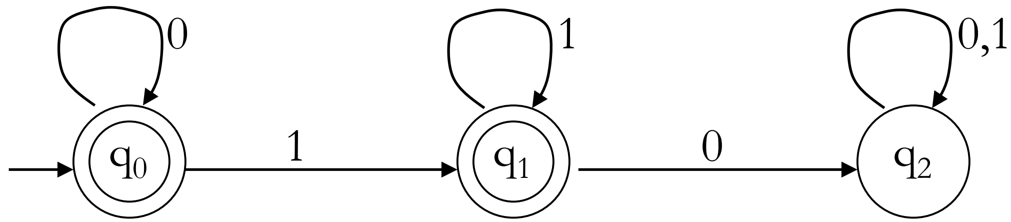


- There are states off and on, the automaton starts in off and tries to reach the "good state" on
- What sequences of fs lead to the good state?
- Answer: $\{f, fff, fffff, \ldots\} = \{f^{\,n} : n \text{ is odd}\}$
- This is an example of a deterministic finite automaton over alphabet $\{f\}$

Deterministic finite automata

- A deterministic finite automaton (DFA) is a 5-tuple $(\mathcal{Q}, \Sigma, \delta, q_0, F)$ where
  - $\mathcal{Q}$ is a finite set of states
  - Σ is an alphabet
  - $\delta: \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is a transition function
  - $q_0 \in \mathcal{Q}$ is the initial state
  - $F \subseteq \mathcal{Q}$ is a set of accepting states (or final states).
- In diagrams, the accepting states will be denoted by double loops

Example



alphabet $\Sigma = \{0, 1\}$
start state $\mathcal{Q} = \{q_0, q_1, q_2\}$
initial state q_0
accepting states $F = \{q_0, q_1\}$

transition function δ:

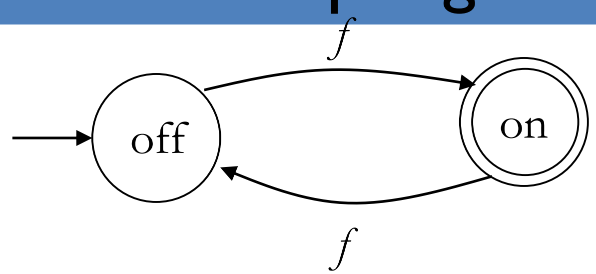
inputs

states	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_2	q_2

Language of a DFA

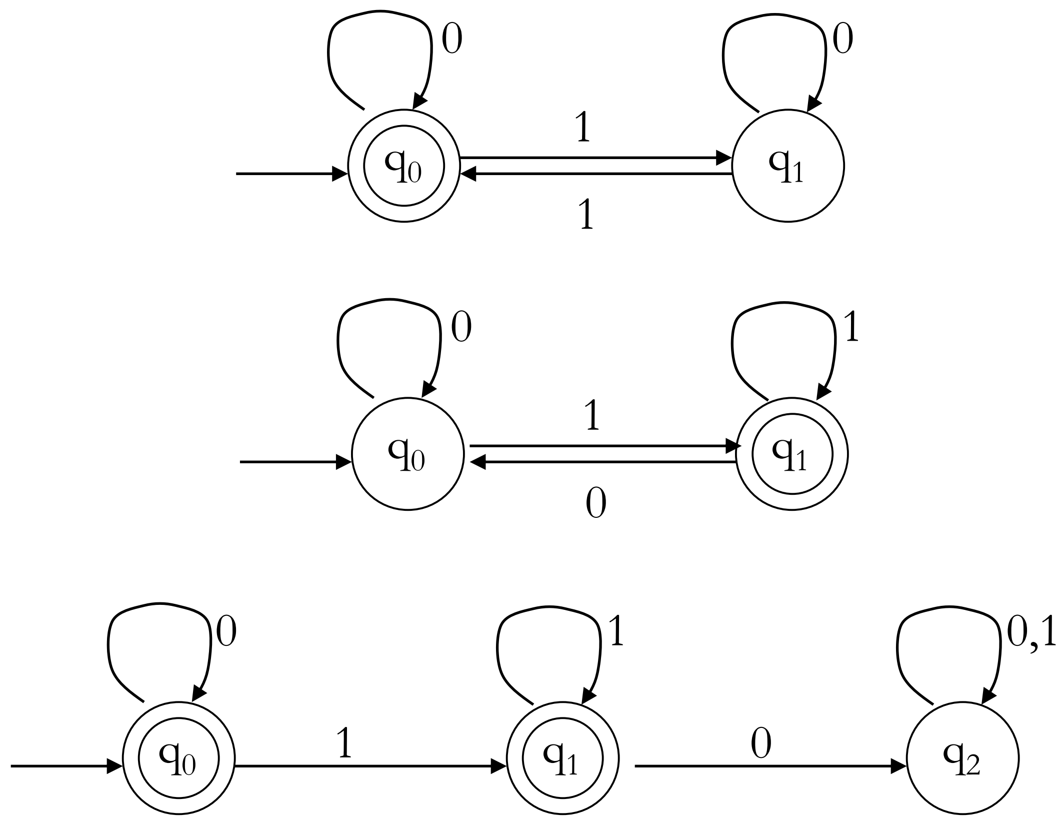
The language of a DFA $(Q, \Sigma, \delta, q_0, F)$ is the set of all strings over Σ that, starting from q_0 and following the transitions as the string is read left to right, will reach some accepting state.

M:



- Language of M is $\{f, fff, fffff, \ldots\} = \{f^n : n \text{ is odd}\}$

Examples



What are the languages of these DFAs?

Examples

- Construct a DFA that accepts the language

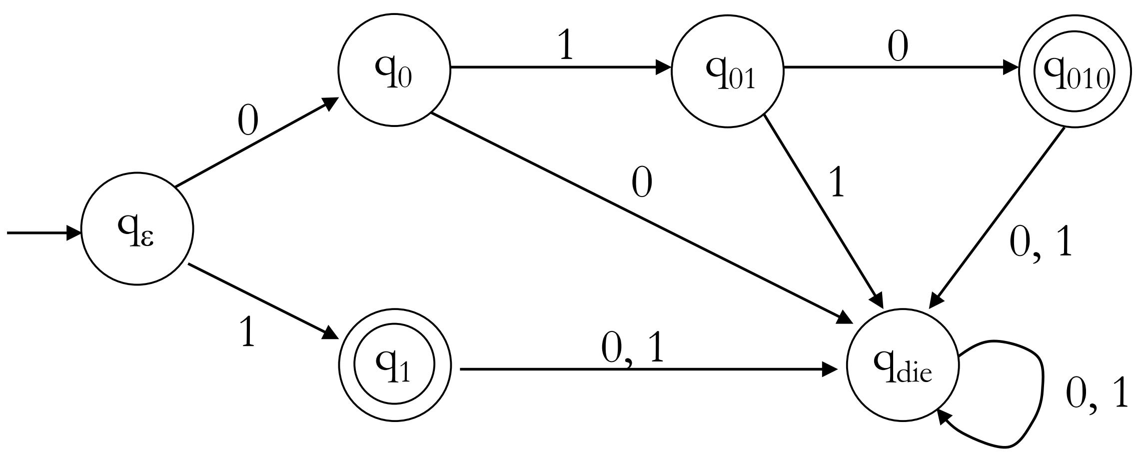
$$L = \{010, 1\} \qquad (\, \Sigma = \{0, 1\} \,)$$

Examples

- Construct a DFA that accepts the language

$$L = \{010, 1\} \qquad (\, \Sigma = \{0, 1\}\,)$$

- Answer



Examples

- Construct a DFA over alphabet $\{0, 1\}$ that accepts all strings that end in 101

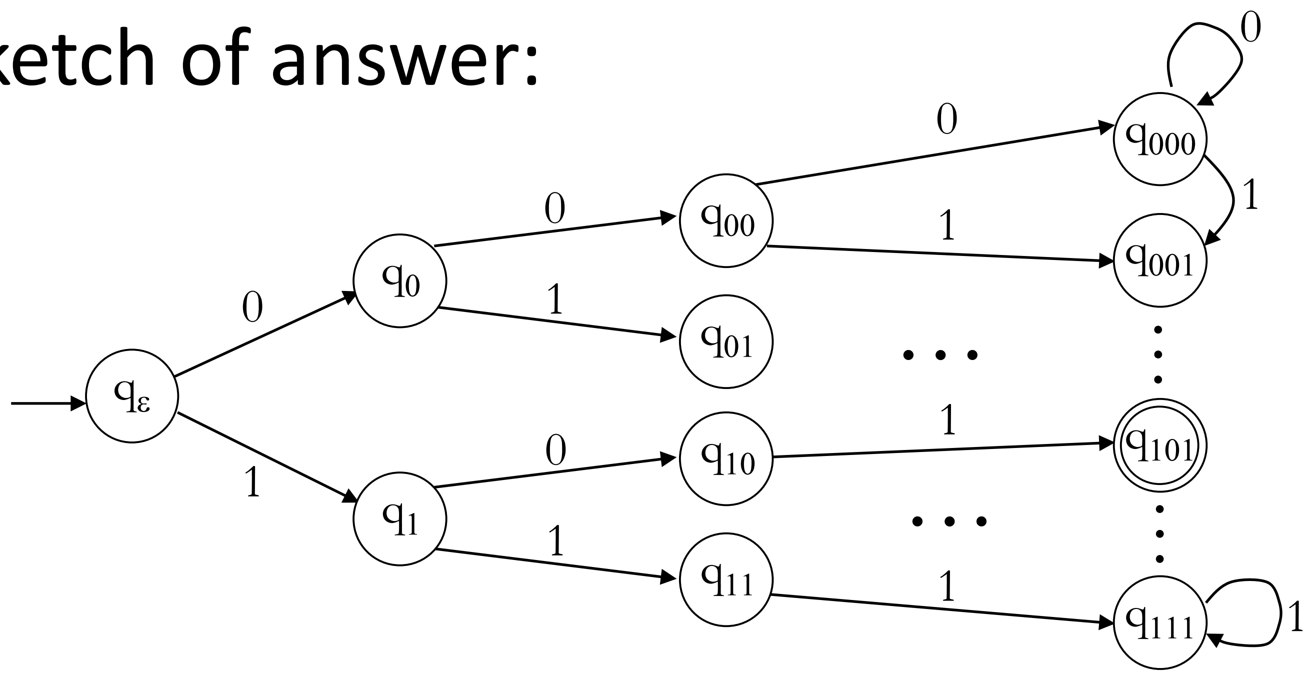
Examples

- Construct a DFA over alphabet $\{0, 1\}$ that accepts all strings that end in 101

- Hint: The DFA must "remember" the last 3 bits of the string it is reading

Examples

- Construct a DFA over alphabet $\{0, 1\}$ that accepts all strings that end in 101
- Sketch of answer:



Grammar ?

- Describes underlying rules (syntax) of programming languages

  Compilers (parsers) are based on such descriptions

- More expressive than regular expressions/finite automata
- Context-free grammar (CFG) or just *grammar*

Grammar and its Chomsky Classification

- We'll cover three types of structures used in modeling computation:
- Grammars
  - Used to generate sentences of a language and to determine if a given sentence is in a language
  - Formal languages, generated by grammars, provide models for programming languages (Java, C, etc) as well as natural language --- important for constructing compilers
- Finite-state machines (FSM)
  - FSM are characterized by a set of states, an input alphabet, and transitions that assigns a next state to a pair of state and an input. We'll study FSM with and without output. They are used in language recognition (equivalent to certain grammar)but also for other tasks such as controlling vending machines
- Turing Machine – they are an abstraction of a computer; used to compute number theoretic functions

Intro to Languages

- English grammar tells us if a given combination of words is a valid sentence.
- The syntax of a sentence concerns its form while the semantics concerns
- its meaning.
- e.g. the mouse wrote a poem
- From a syntax point of view this is a valid sentence.
- From a semantics point of view not so fast…perhaps in Disney land
- Natural languages (English, French, Portguese, etc) have very complex rules of syntax and not necessarily well-defined.

Formal Language

- Formal language – is specified by well-defined set of rules of syntax

- We describe the sentences of a formal language using a grammar.

- Two key questions:
- 1 - Is a combination of words a valid sentence in a formal language?
- 2 – How can we generate the valid sentences of a formal language?

- Formal languages provide models for both natural languages and programming languages.

Grammars

- A formal *grammar G* is any compact, precise mathematical definition of a language *L*.
  - As opposed to just a raw listing of all of the language's legal sentences, or just examples of them.
- A grammar implies an algorithm that would generate all legal sentences of the language.
  - Often, it takes the form of a set of recursive definitions.
- A popular way to specify a grammar recursively is to specify it as a *phrase-structure grammar.*

Grammars (Semi-formal)

- Example: A grammar that generates a subset of the English language

$$\langle sentence \rangle \rightarrow \langle noun\_phrase \rangle \ \langle predicate \rangle$$

$$\langle noun\_phrase \rangle \rightarrow \langle article \rangle \ \langle noun \rangle$$

$$\langle predicate \rangle \rightarrow \langle verb \rangle$$

$$\langle article \rangle \rightarrow a$$

$$\langle article \rangle \rightarrow the$$

•

$$\langle noun \rangle \rightarrow boy$$

$$\langle noun \rangle \rightarrow dog$$

$$\langle verb \rangle \rightarrow runs$$

$$\langle verb \rangle \rightarrow sleeps$$

- A derivation of "the boy sleeps":

$$
\begin{aligned}
\langle sentence \rangle &\Rightarrow \langle noun\_phrase \rangle \ \langle predicate \rangle \\
&\Rightarrow \langle noun\_phrase \rangle \ \langle verb \rangle \\
&\Rightarrow \langle article \rangle \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow the \ \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow the \ \ boy \ \ \langle verb \rangle \\
&\Rightarrow the \ \ boy \ \ \ sleeps
\end{aligned}
$$

- Language of the grammar:

L = { "a boy runs",
"a boy sleeps",
"the boy runs",
"the boy sleeps",
"a dog runs",
"a dog sleeps",
"the dog runs",
"the dog sleeps" }

Notation

- $$\langle noun \rangle \to boy$$
  $$\langle noun \rangle \to dog$$

Variable
or
Non-terminal

Symbols of
the vocabulary

Production
rule

Terminal
Symbols of
the vocabulary

Basic Terminology

- A ***vocabulary***/***alphabet***, V is a finite nonempty set of elements called symbols.
  - Example: V = {a, b, c, A, B, C, S}
- A ***word***/***sentence*** over V is a string of finite length of elements of V.
  - Example: *Aba*
- The ***empty***/***null string***, λ is the string with no symbols.
- V^* is the set of all words over V.
  - Example: V^* = {*Aba*, *BBa*, *bAA*, *cab* …}
- A ***language*** over V is a subset of V^*.
  - We can give some criteria for a word to be in a language.

Analytical Definition of grammar

A grammar is a 4-tuple G = (V,T,P,S)

- V: set of variables or nonterminals
- T: set of terminal symbols (terminals)
- P: set of productions
  - Each production: **head** $\rightarrow$ **body**, where **head** is a variable, and **body** is a string of zero or more terminals and variables
- S: a start symbol from V

Example 1:

Assignment statements

- V = { S, E }, T = { i, =, +, *, n }
- Productions:

  $S \rightarrow i = E$

  $E \rightarrow n$

  $E \rightarrow i$

  $E \rightarrow E + E$

  $E \rightarrow E * E$

Example 3: 0^n1^n

- V = { S }, T = { 0, 1 }
- Productions:
  $S \rightarrow \varepsilon$
  $S \rightarrow 0S1$

Derivation

- Definition
- Let G=(V,T,S,P) be a phrase-structure grammar.
- Let $w_0=lz_0r$ (the concatenation of l, z_0, and r) $w_1=lz_1r$ be strings over V.
- If $z_0 \rightarrow z_1$ is a production of G we say that w1 is directly derivable from w0 and we write $w_o \Rightarrow w_1$.
- If $w_0, w_1, \ldots, w_n$ are strings over V such that $w_0 \Rightarrow w_1, w_1 \Rightarrow w_2, \ldots, w_{n-1} \Rightarrow w_n$, then we say that w_n is derivable from w_0, and write $w_0 \Rightarrow^* w_n$.
- The sequence of steps used to obtain w_n from w_o is called a derivation.

L(G): Language of a grammar

- Definition: Given a grammar G, and a string w over the alphabet T, $S \Rightarrow^{*}_{G}$ w if there is a sequence of productions that derive w
- L(G) = { w in T* | $S \Rightarrow^{*}_{G}$ w },
  the language of the grammar G

Leftmost vs rightmost derivations

- Leftmost derivation: the leftmost variable is always the one replaced when applying a production
  - Example: $S \Rightarrow i = E \Rightarrow i = E + E$
    $\Rightarrow i = n + E \Rightarrow i = n + n$
- Rightmost derivation: rightmost variable is replaced
  - Example: $S \Rightarrow i = E \Rightarrow i = E + E$
    $\Rightarrow i = E + n \Rightarrow i = n + n$

Sentential forms

- In a derivation, assuming it begins with S, all intermediate strings are called sentential forms of the grammar G
- Example: i = E and i = E + n are sentential forms of the assignment statement grammar
- The sentential forms are called leftmost (rightmost) sentential forms if they are a result of leftmost (rightmost) derivations

Parse trees

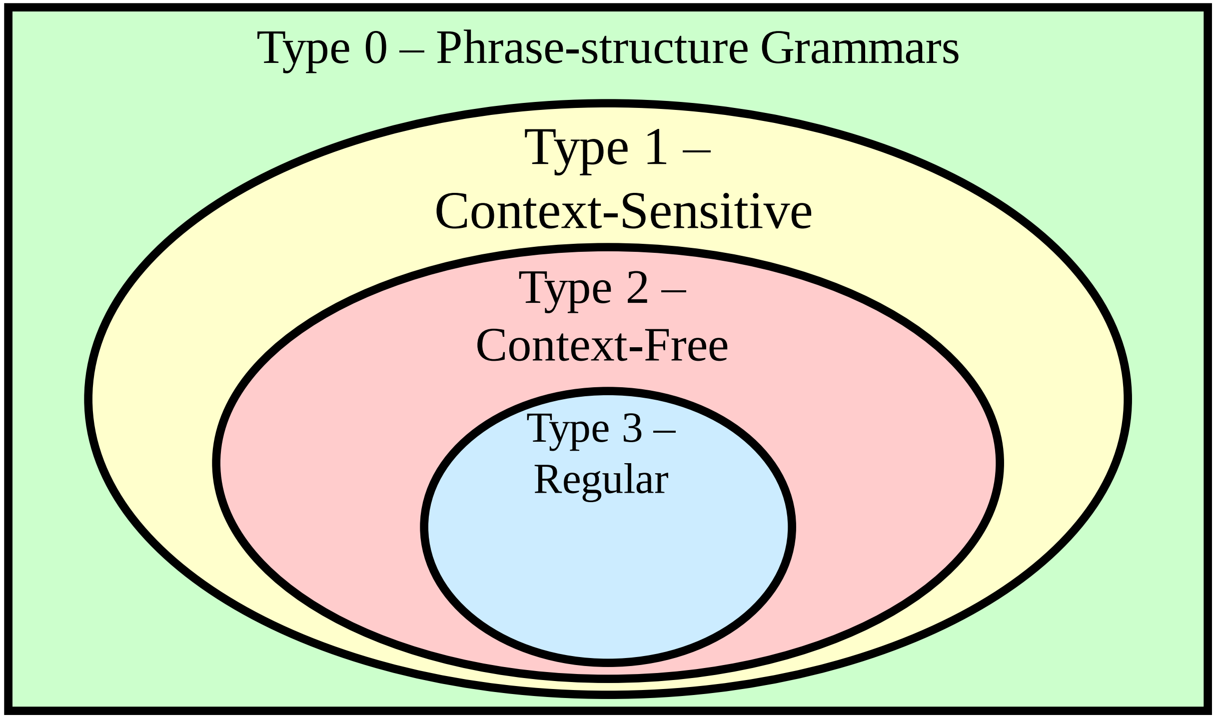
- Recall that a tree in graph theory is a set of nodes such that
  - There is a special node called the root
  - Nodes can have zero or more child nodes
  - Nodes without children are called leaves
  - Interior nodes: nodes that are not leaves
- A parse tree for a grammar G is a tree such that the interior nodes are non-terminals in G and children of a non-terminal correspond to the body of a production in G

Yield of a parse tree

- Yield: concatenation of leaves from left to right
- If the root of the tree is the start symbol, and all leaves are terminal symbols, then the yield is a string in L(G)
- A derivation always corresponds to some parse tree

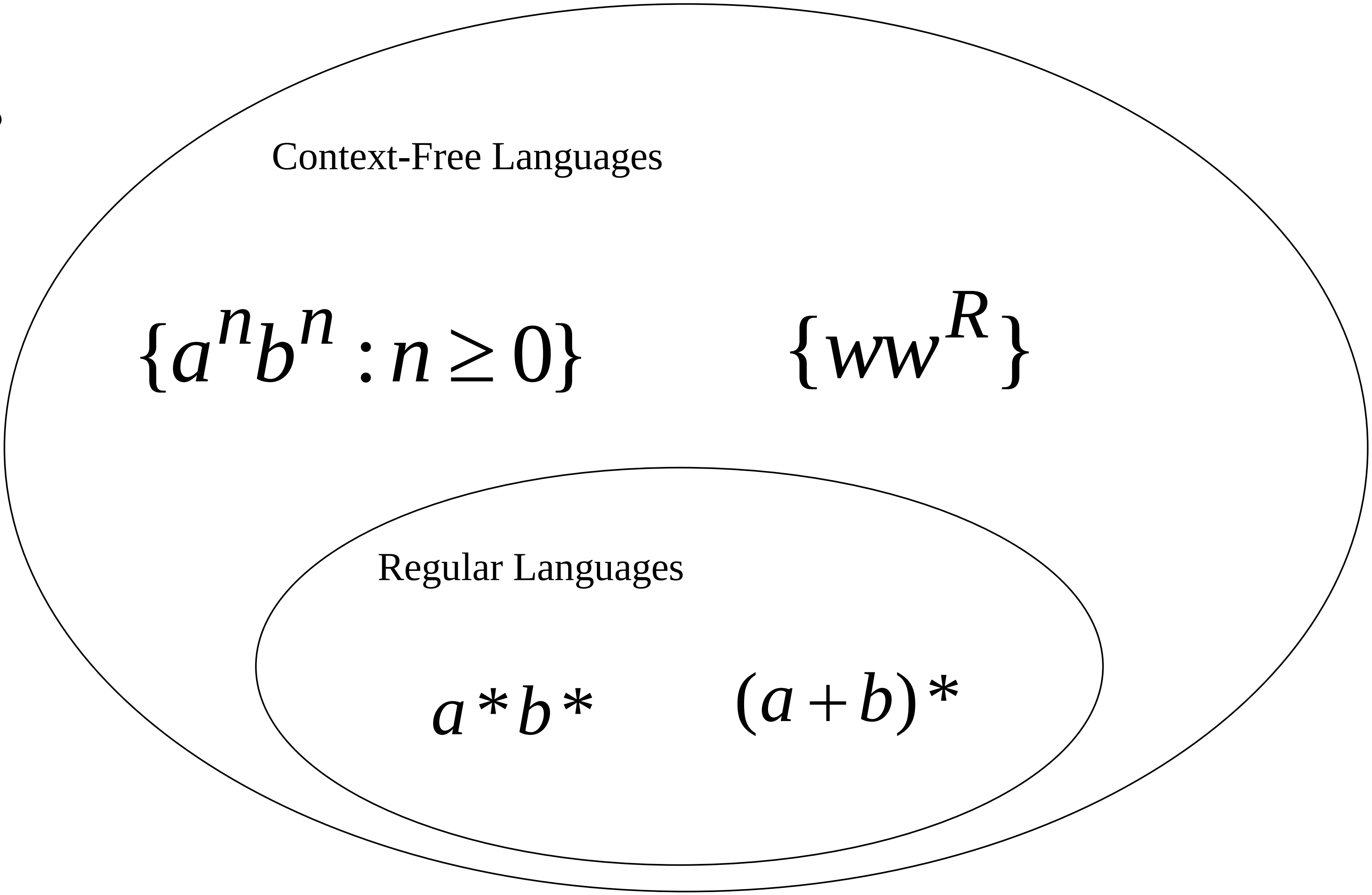
Types of Grammars - Chomsky hierarchy of languages

- Venn Diagram of Grammar Types:



Classifying grammars

- Given a grammar, we need to be able to find the smallest class in which it belongs. This can be determined by answering three questions:
- Are the left hand sides of all of the productions single non-terminals?
- If yes, does each of the productions create at most one non-terminal and is it on the right?
- **Yes – regular** **No – context-free**
- If not, can any of the rules reduce the length of a string of terminals and non-terminals?
- **Yes – unrestricted** **No – context-sensitive**


Context-Free Languages
$\{a^n b^n : n \geq 0\}$
$\{ww^R\}$
Regular Languages
a^*b^*
$(a+b)^*$

Definition: Context-Free Grammars

Grammar $G = (V, T, S, P)$

Vocabulary

Terminal symbols

Start variable

Productions of the form:

$$A \rightarrow x$$

Non-Terminal

String of variables and terminals

Derivation Tree of A Context-free Grammar

- Represents the language using an ordered rooted tree.

- Root represents the starting symbol.
- Internal vertices represent the nonterminal symbol that arise in the production.
- Leaves represent the terminal symbols.

- If the production $A \longrightarrow w$ arise in the derivation, where w is a word, the vertex that represents A has as children vertices that represent each symbol in w, in order from left to right.

Ambiguity

Grammar for mathematical expressions

$$\boxed{E \to E + E \;\mid\; E * E \;\mid\; (E) \;\mid\; a}$$

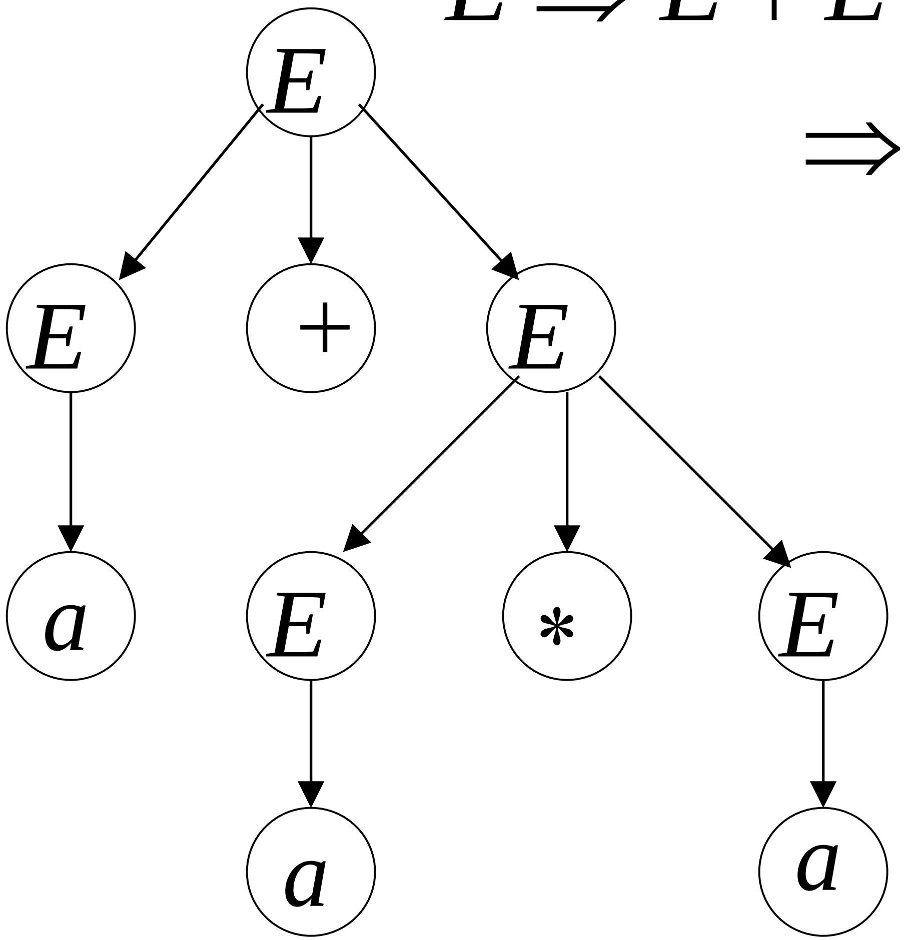
Example strings:

$$(a + a) * a + (a + a * (a + a))$$

Denotes any number

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E$$
$$\Rightarrow a + a * E \Rightarrow a + a * a$$

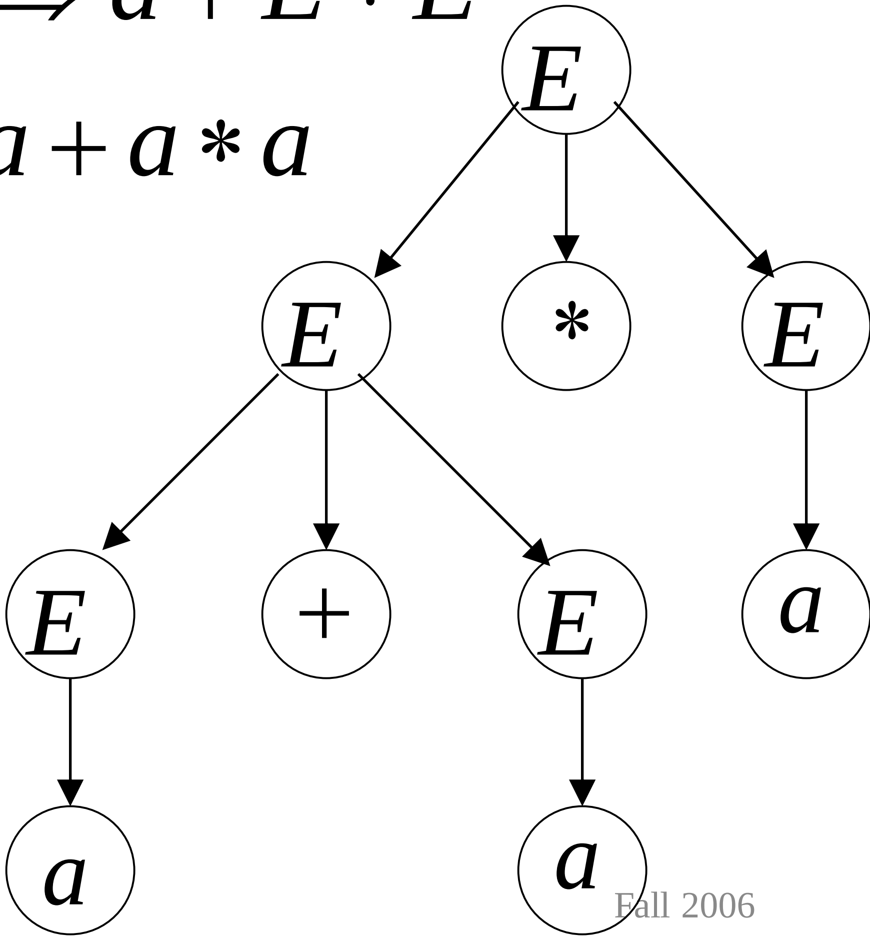


A leftmost derivation
for

$$a + a * a$$

$$E \to E + E \mid E * E \mid (E) \mid a$$

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E$$
$$\Rightarrow a + a * E \Rightarrow a + a * a$$



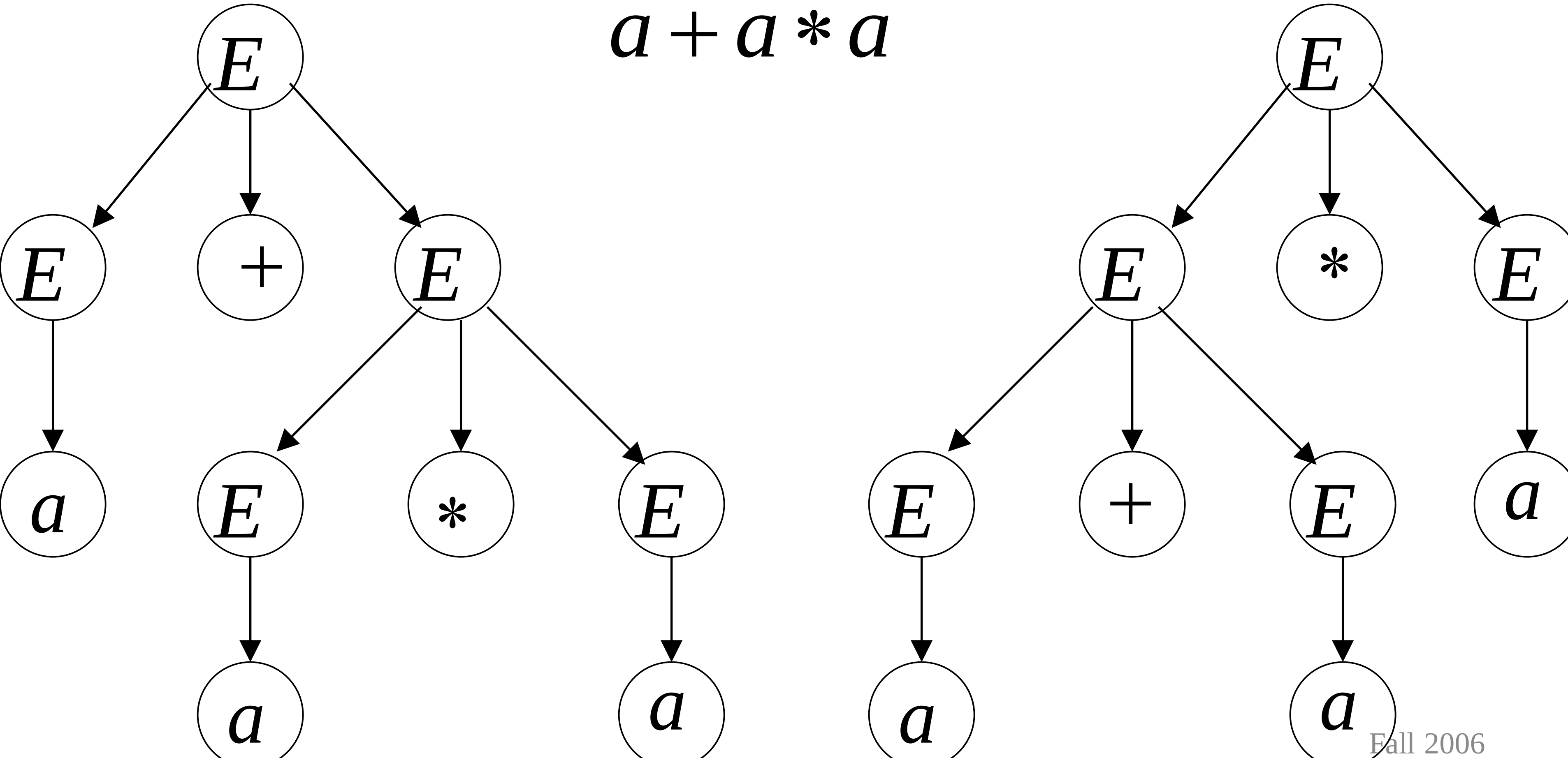
Another leftmost derivation for

$$a + a * a$$

$$E \to E + E \mid E * E \mid (E) \mid a$$

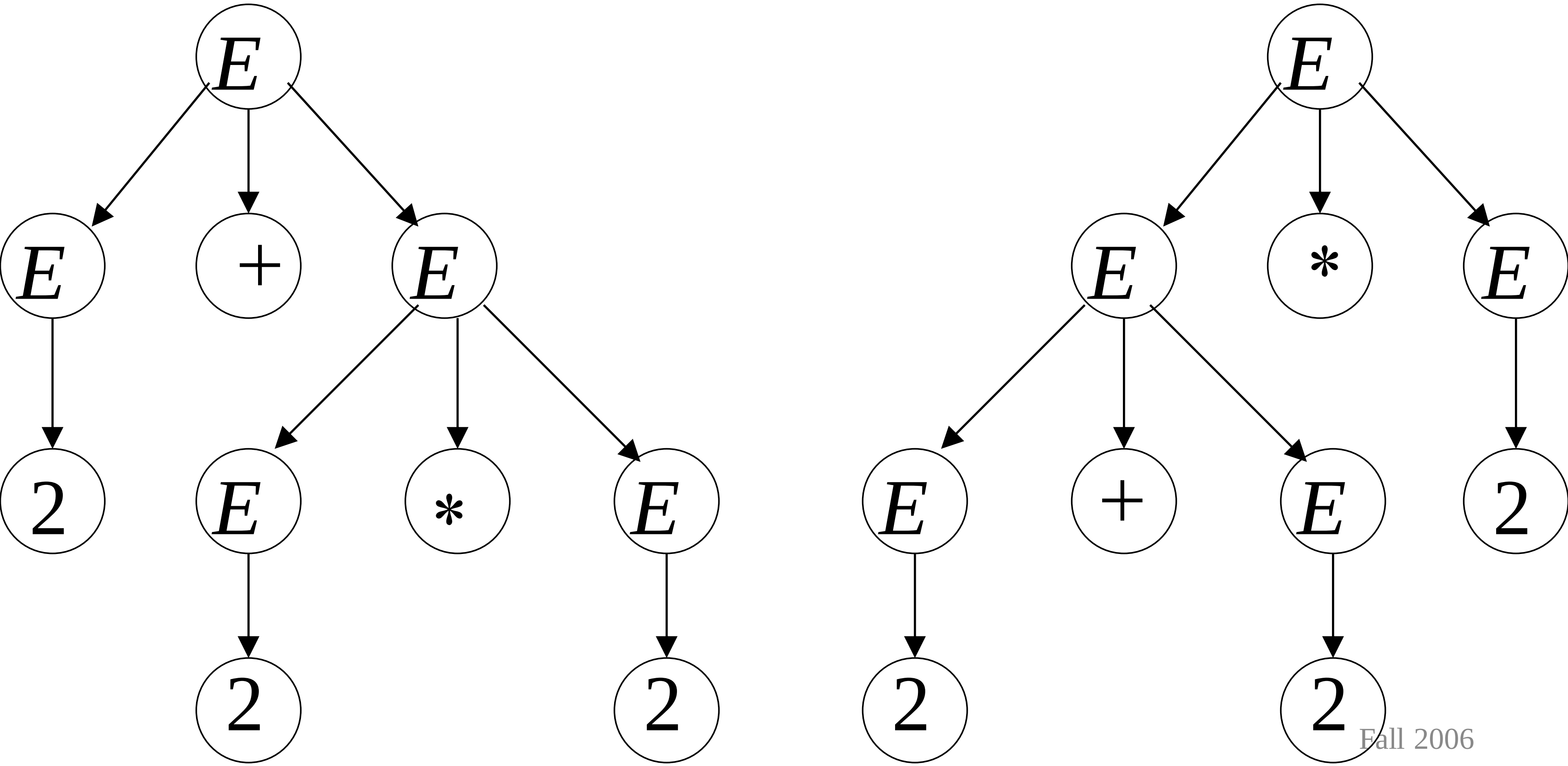
Two derivation trees
for

$$a + a * a$$



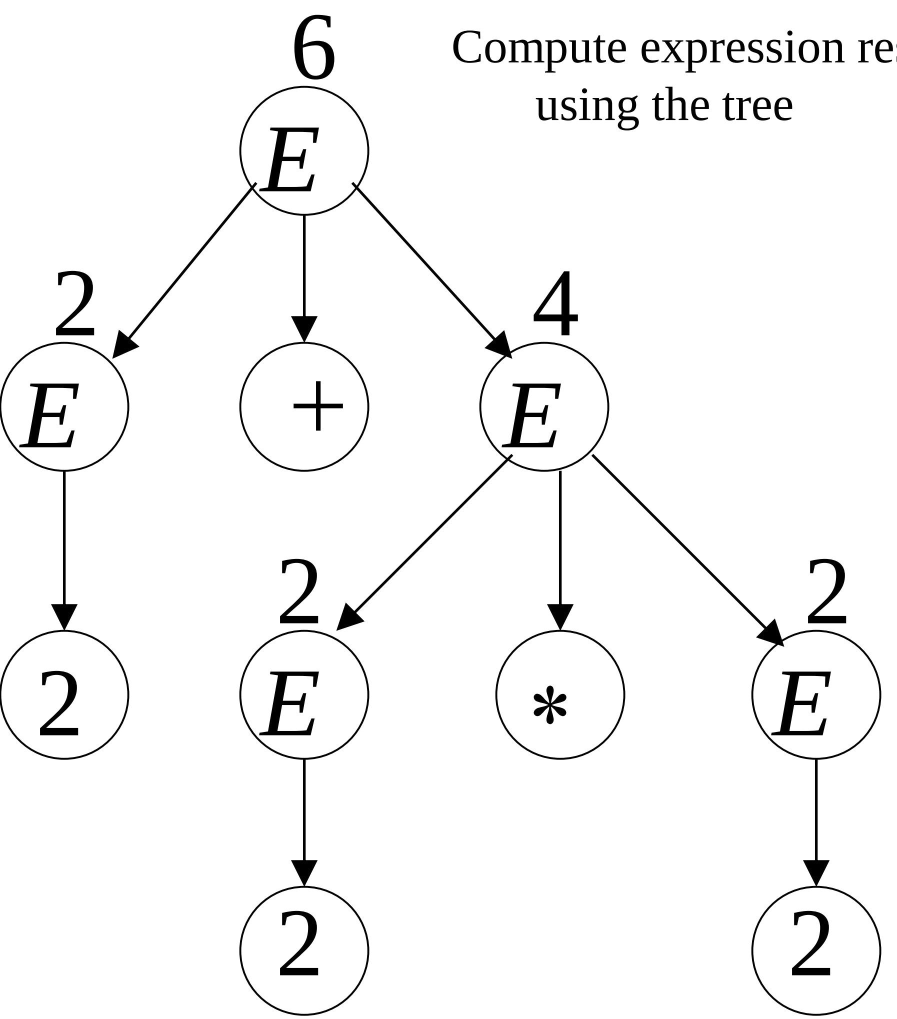
take $a = 2$

$$a + a * a = 2 + 2 * 2$$



Good Tree

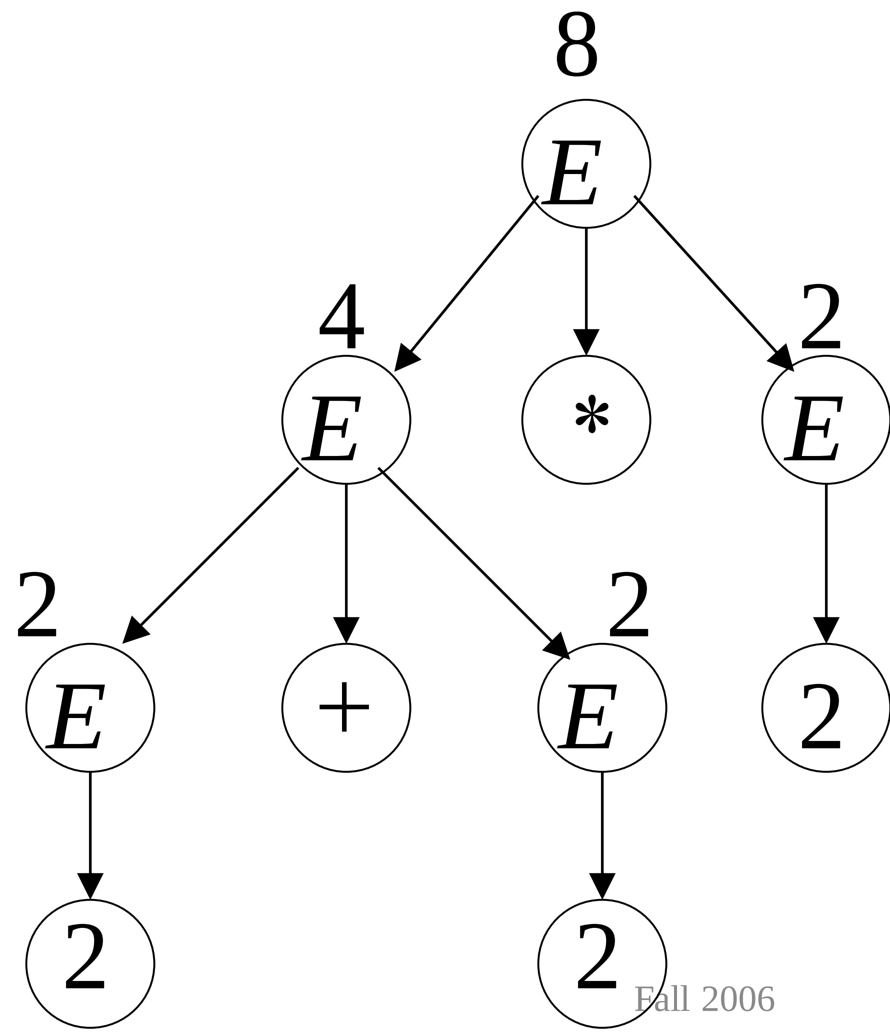
$$2+2*2=6$$



Compute expression result using the tree

Bad Tree

$$2+2*2=8$$



Ambiguous Grammar:

A context-free grammar G is ambiguous
if there is a string $w \in L(G)$ which has:

two different derivation trees
or
two leftmost derivations

(Two different derivation trees give two different leftmost derivations and vice-versa)

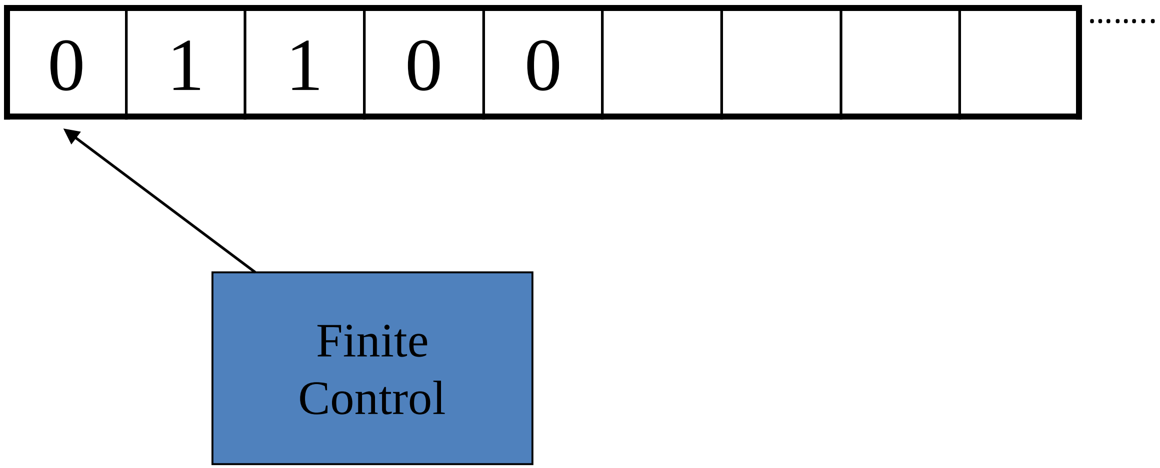
• Context-Sensitive Languages

- The language $\{ a^n b^n c^n \mid n \geq 1\}$ is context-sensitive but not context free.
- A grammar for this language is given by:
- $S \rightarrow aSBC \mid aBC$
- $CB \rightarrow BC$
- $aB \rightarrow ab$
- $bB \rightarrow bb$
- $bC \rightarrow bc$
- $cC \rightarrow cc$

Terminal and non-terminal

- A derivation from this grammar is:-
- $S \Rightarrow aSBC$
- $\Rightarrow aaBCBC$ (using $S \rightarrow aBC$)
- $\Rightarrow aabCBC$ (using $aB \rightarrow ab$)
- $\Rightarrow aabBCC$ (using $CB \rightarrow BC$)
- $\Rightarrow aabbCC$ (using $bB \rightarrow bb$)
- $\Rightarrow aabbcC$ (using $bC \rightarrow bc$)
- $\Rightarrow aabbcc$ (using $cC \rightarrow cc$)
- which derives $a^2b^2c^2$.

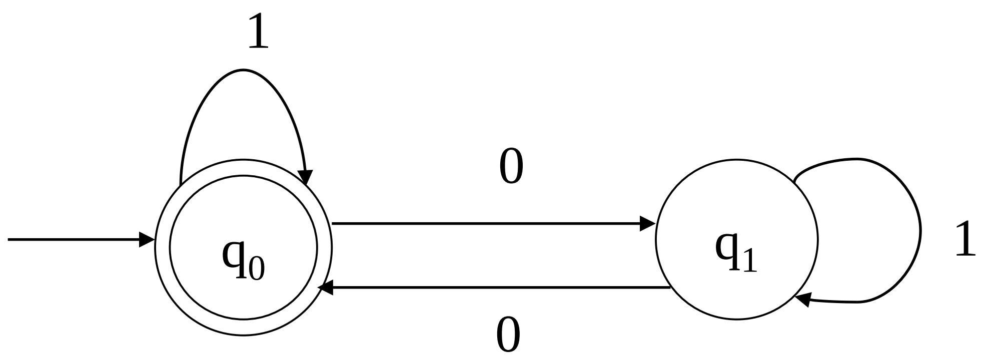
Deterministic Finite State Automata (DFA)



- One-way, infinite tape, broken into cells
- One-way, read-only tape head.
- Finite control, i.e.,
  - finite number of states, and
  - transition rules between them, i.e.,
  - a program, containing the position of the read head, current symbol being scanned, and the current "state."
- A string is placed on the tape, read head is positioned at the left end, and the DFA will read the string one symbol at a time until all symbols have been read. The DFA will then either *accept* or *reject* the string.

- The finite control can be described by a transition diagram or table:

- Example #1:



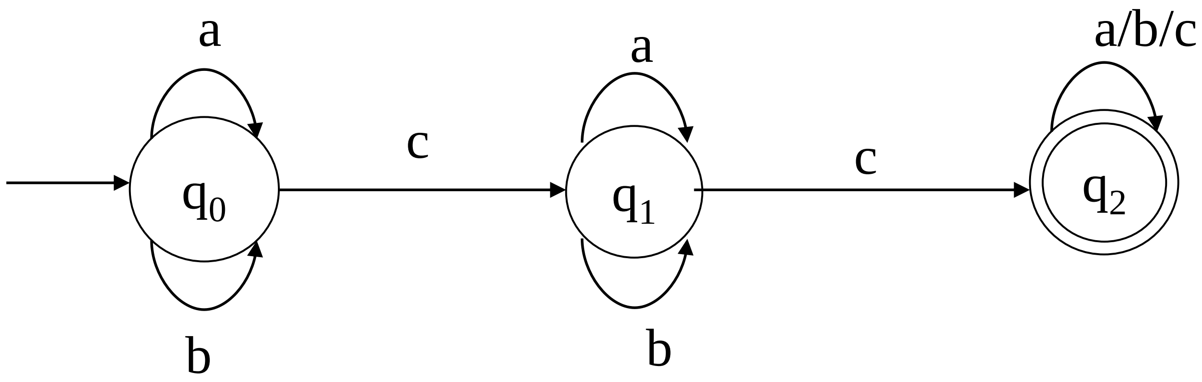
$$\begin{array}{cccccccccccc} & 1 & & 0 & & 0 & & 1 & & 1 & \\ q_0 & & q_0 & & q_1 & & q_0 & & q_0 & & q_0 \end{array}$$

- One state is final/accepting, all others are rejecting.
- The above DFA accepts those strings that contain an even number of 0's, including the *null* string, over *Sigma* = {0,1}

  L = {all strings with zero or more 0's}

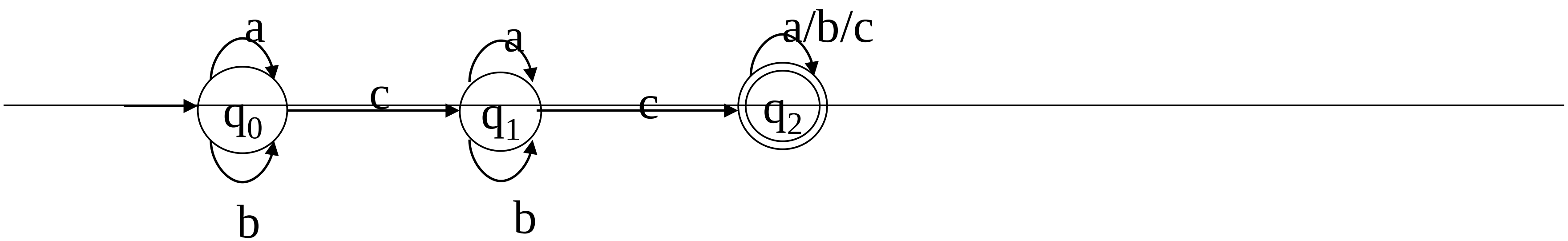
- Note, the DFA must reject all other strings

- Example #2:



	a	c	c	c	b	<u>accepted</u>
q_0	q_0	q_1	q_2	q_2	q_2	
	a	a	c			<u>rejected</u>
q_0	q_0	q_0	q_1			

- Accepts those strings that contain <u>at least</u> two *c*'s



Inductive Proof (sketch): that the machine correctly accepts strings with at least two *c*'s
Proof goes over the length of the string.

Base: *x* a string with |x|=0. state will be q0 => rejected.
Inductive hypothesis: |x|= integer *k*, & string *x* is *rejected* - in state q0 (x must have *zero* c),
OR, *rejected* – in state q1 (x must have *one* c),
OR, *accepted* – in state q2 (x has already with *two* c's)

Inductive steps: Each case for symbol *p*, for string *xp* (|*xp*| = *k*+1), the last symbol *p* = *a*, *b* or *c*

	xa	**xb**	**xc**
x ends in q0	q0 =>reject *(still zero c => should reject)*	q0 =>reject *(still zero c => should reject)*	q1 =>reject *(still zero c => should reject)*
x ends in q1	q1 =>reject *(still one c => should reject)*	q1 =>reject *(still one c => should reject)*	q2 =>accept *(two c now=> should accept)*
x ends in q2	q2 =>accept (two c already => should accept)	q2 =>accept (two c already => should accept)	q2 =>accept (two c already => should accept)

Formal Definition of a DFA

- A DFA is a five-tuple:

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

Q	A finite set of states
Σ	A finite input alphabet
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, which is a total function from Q x Σ to Q

$\delta: (Q \times \Sigma) \to Q$	δ is defined for any q in Q and s in Σ, and
$\delta(q,s) = q'$	is equal to some state q' in Q, could be q'=q

  Intuitively, $\delta(q,s)$ is the state entered by M after reading symbol s while in state q.

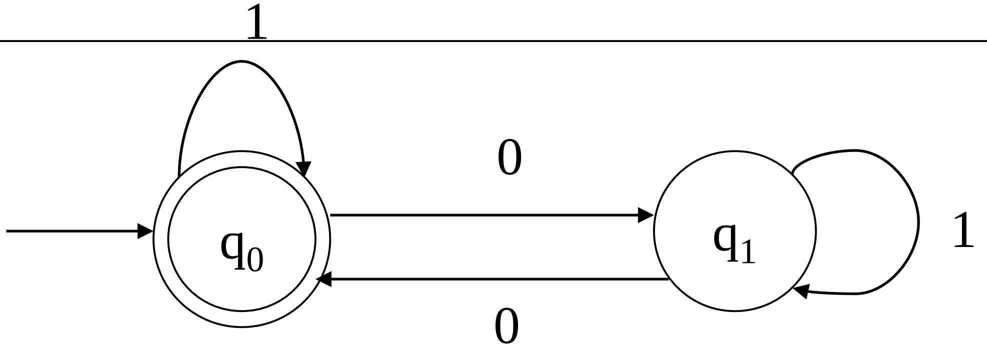
- Revisit example #1:

$Q = \{q_0, q_1\}$

$\Sigma = \{0, 1\}$

Start state is q_0

$F = \{q_0\}$



δ:

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

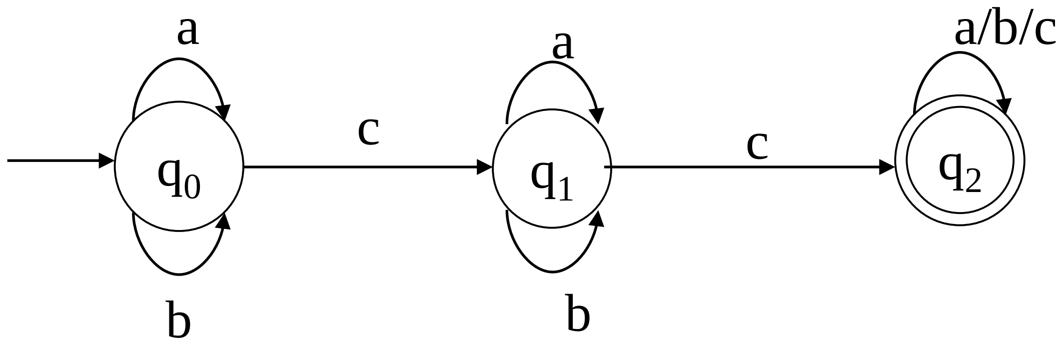
- Revisit example #2:

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{a, b, c\}$

Start state is q_0

$F = \{q_2\}$



δ:

	a	b	c
q_0	q_0	q_0	q_1
q_1	q_1	q_1	q_2
q_2	q_2	q_2	q_2

- Since δ is a function, at each step M has exactly one option.
- It follows that for a given string, there is exactly one computation.

Nondeterministic Finite State Automata (NFA)

- An NFA is a five-tuple:

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

Q	A finite set of states
Σ	A finite input alphabet
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, which is a total function from Q x Σ to 2^Q

$\delta: (Q \times \Sigma) \to$ **2^Q** :2^Q is the power set of Q, the set of *all subsets* of Q

$\delta(q,s)$:The set of all states p such that there is a transition labeled s from q to p

$\delta(q,s)$ is a function from Q x S to 2^Q (but not only to Q)

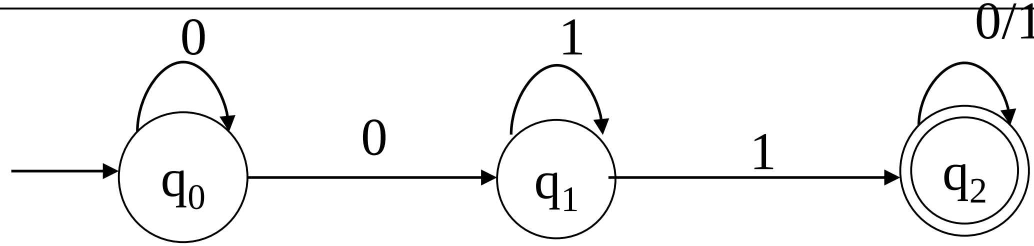
- Example #1: one or more 0's followed by one or more 1's

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1\}$

Start state is q_0

$F = \{q_2\}$



δ:

	0	1
q_0	$\{q_0, q_1\}$	$\{\}$
q_1	$\{\}$	$\{q_1, q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$

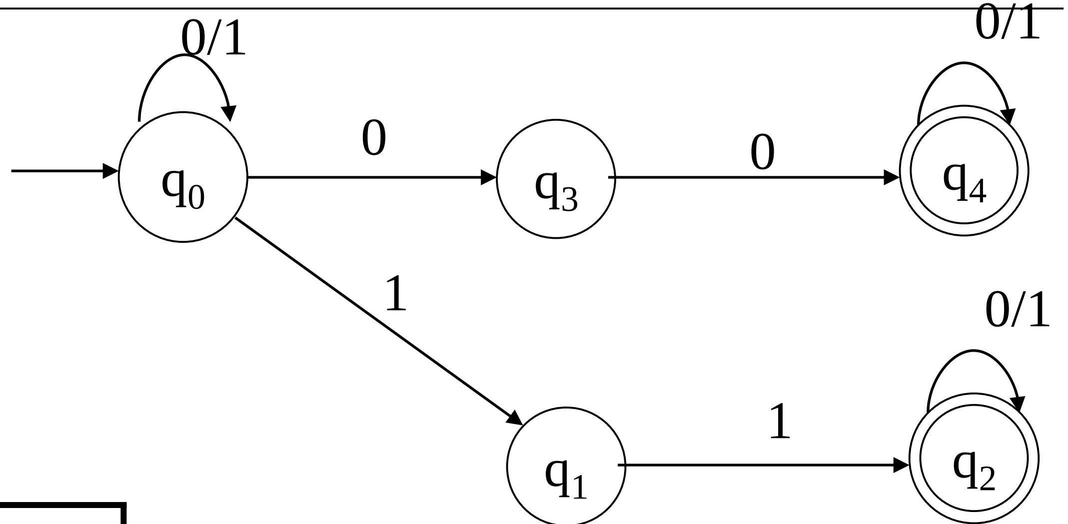
- Example #2: pair of 0's *or* pair of 1's as substring

$Q = \{q_0, q_1, q_2, q_3, q_4\}$

$\Sigma = \{0, 1\}$

Start state is q_0

$F = \{q_2, q_4\}$



δ:

	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\{\}$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\{\}$
q_4	$\{q_4\}$	$\{q_4\}$

- Question: Why non-determinism is useful?
  - Non-determinism = Backtracking
  - Compressed information
  - Non-determinism hides backtracking
  - Programming languages, e.g., Prolog, hides backtracking => Easy to program at a higher level: *what we want to do, rather than how to do it*
  - Useful in algorithm complexity study

  - Is NDA more "powerful" than DFA, i.e., accepts type of languages that any DFA cannot?

Equivalence of DFAs and NFAs

- Do DFAs and NFAs accept the same *class* of languages?
  - Is there a language L that is accepted by a DFA, but not by any NFA?
  - Is there a language L that is accepted by an NFA, but not by any DFA?

- Observation: Every DFA is an NFA, DFA is only restricted NFA.

- Therefore, if L is a regular language then there exists an NFA M such that L = L(M).

- It follows that NFAs accept all regular languages.

- But do NFAs accept more?

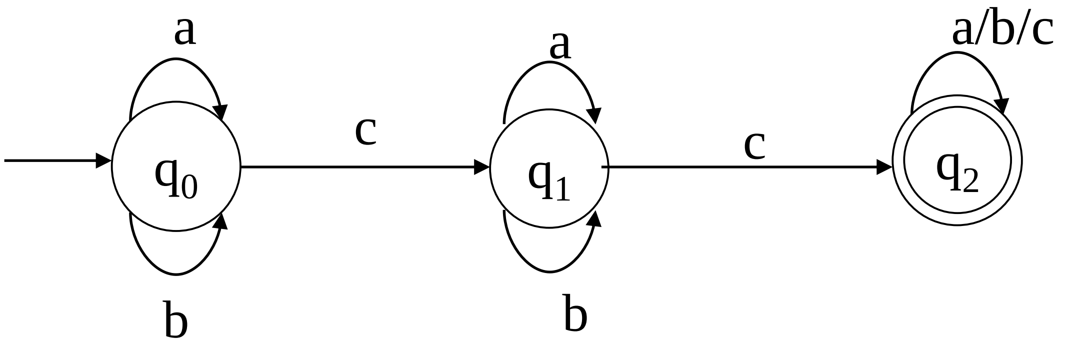
- Consider the following DFA: 2 or more c's

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{a, b, c\}$

Start state is q_0

$F = \{q_2\}$



δ:

	a	b	c
q_0	q_0	q_0	q_1
q_1	q_1	q_1	q_2
q_2	q_2	q_2	q_2

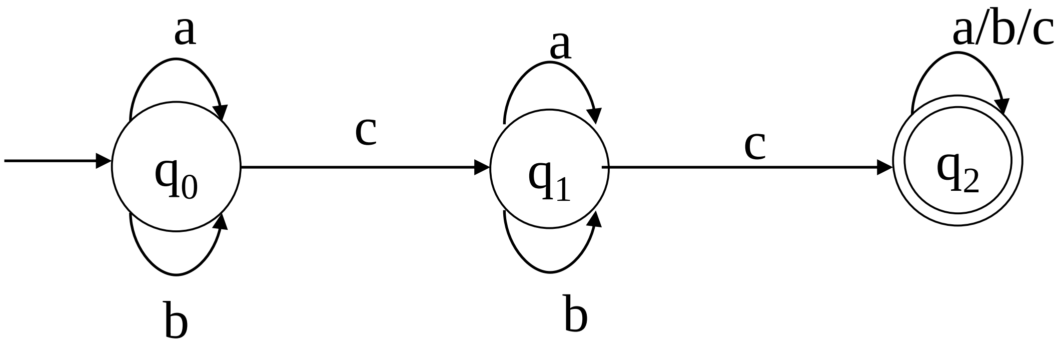
- An Equivalent NFA:

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{a, b, c\}$

Start state is q_0

$F = \{q_2\}$



δ:

	a	b	c
q_0	$\{q_0\}$	$\{q_0\}$	$\{q_1\}$
q_1	$\{q_1\}$	$\{q_1\}$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$

Real-life Uses of DFAs

Grep

Coke Machines

Thermostats (fridge)

Elevators

Train Track Switches

Lexical Analyzers for Parsers

Chomsky & Greibach Normal Forms

Presentation Outline

- Introduction
- Chomsky normal form
  - Preliminary simplifications
  - Final steps
- Greibach Normal Form
  - Algorithm (Example)
- Summary

Introduction

Grammar: $G = (V, T, P, S)$

Terminals	$T = \{ a, b \}$
Variables	$V = A, B, C$
Start Symbol	S
Production	$P = S \to A$

Grammar example

$S \rightarrow aBSc$
$S \rightarrow abc$
$Ba \rightarrow aB$
$Bb \rightarrow bb$

$$L = \{\, a^n b^n c^n \mid n \geq 1 \,\}$$

$$S \Longrightarrow aBSc \Longrightarrow aBabcc \Longrightarrow aaBbcc \Longrightarrow aabbcc$$

Context free grammar

The head of any production contains only one non-terminal symbol

$S \rightarrow P$
$P \rightarrow aPb$
$P \rightarrow \varepsilon$

$$L = \{ a^n b^n \mid n \geq 0 \}$$

Chomsky Normal Form

A context free grammar is said to be in **Chomsky Normal Form** if all productions are in the following form:

$$A \rightarrow BC$$
$$A \rightarrow \alpha$$

- A, B and C are non terminal symbols
- α is a terminal symbol

Preliminary Simplifications

There are three preliminary simplifications

1 Eliminate Useless Symbols

2 Eliminate ε productions

3 Eliminate unit productions

Preliminary Simplifications

Eliminate Useless Symbols

We need to determine if the symbol is useful by identifying if a symbol is **generating** and is **reachable**

- X is **generating** if $X \overset{*}{\Longrightarrow} \omega$ for some terminal string ω.
- X is **reachable** if there is a derivation $X \overset{*}{\Longrightarrow} \alpha X \beta$ for some α and β

Preliminary Simplifications

Example: Removing **non-generating** symbols

$S \to AB \mid a$
$A \to b$

Initial CFL grammar

$S \to AB \mid a$
$A \to b$

Identify generating symbols

$S \to a$
$A \to b$

Remove non-generating

Preliminary Simplifications

Example: Removing **non-reachable** symbols

$S \rightarrow a$
$A \rightarrow b$

Identify reachable symbols

$S \rightarrow a$

Eliminate non-reachable

Preliminary Simplifications

The order is important.

Looking first for non-reachable symbols and then for non-generating symbols can still leave some useless symbols.

$S \to AB \mid a$
$A \to b$

$\Rightarrow$

$S \to a$
$A \to b$

Preliminary Simplifications

Finding **generating** symbols

If there is a production $A \rightarrow \alpha$, and every symbol of α is already known to be generating. Then A is generating

$S \rightarrow AB \mid a$
$A \rightarrow b$

We cannot use $S \rightarrow AB$ because B has not been established to be generating

Finding **reachable** symbols

S is surely reachable. All symbols in the body of a production with S in the head are reachable.

$S \rightarrow AB \mid a$
$A \rightarrow b$

In this example the symbols {S, A, B, a, b} are reachable.

Preliminary Simplifications

Eliminate ε Productions

- In a grammar ε productions are convenient but not essential
- If L has a CFG, then L – {ε} has a CFG

$$A \stackrel{*}{\Rightarrow} \varepsilon$$

Nullable variable

Preliminary Simplifications

If A is a nullable variable

- Whenever A appears on the body of a production
  A might or might not derive ε

$$
\begin{aligned}
S &\rightarrow ASA \mid aB \\
A &\rightarrow B \mid S \\
B &\rightarrow b \mid \varepsilon
\end{aligned}
$$

Nullable: {A, B}

Preliminary Simplifications

Eliminate ε Productions

- Create two version of the production, one with the nullable variable and one without it
- Eliminate productions with ε bodies

S → ASA | aB
A → B | S
B → b | ε

⇒

S → ASA | aB | AS | SA | S | a
A → B | S
B → b

Preliminary Simplifications

Eliminate ε Productions

- Create two version of the production, one with the nullable variable and one without it
- Eliminate productions with ε bodies

$S \rightarrow ASA \mid aB$
$A \rightarrow B \mid S$
$B \rightarrow b \mid \varepsilon$

$\Rightarrow$

$S \rightarrow ASA \mid aB \mid AS \mid SA \mid S \mid a$
$A \rightarrow B \mid S$
$B \rightarrow b$

Preliminary Simplifications

Eliminate ε Productions

- Create two version of the production, one with the nullable variable and one without it
- Eliminate productions with ε bodies

$S \rightarrow ASA \mid aB$
$A \rightarrow B \mid S$
$B \rightarrow b \mid \varepsilon$

$\Rightarrow$

$S \rightarrow ASA \mid aB \mid AS \mid SA \mid S \mid a$
$A \rightarrow B \mid S$
$B \rightarrow b$

Preliminary Simplifications

Eliminate unit productions

A unit production is one of the form $A \rightarrow B$ where both A and B are variables

Identify **unit pairs**

$$A \overset{*}{\Longrightarrow} B$$

$A \rightarrow B$, $B \rightarrow \omega$, then $A \rightarrow \omega$

Preliminary Simplifications

Example:

$T = \{*, +, (,), a, b, 0, 1\}$

$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
$F \rightarrow I \mid (E)$
$T \rightarrow F \mid T * F$
$E \rightarrow T \mid E + T$

Basis: (A, A) is a unit pair of any variable A, if $A \overset{*}{\Rightarrow} A$ by 0 steps.

Pairs	Productions
(E, E)	$E \rightarrow E + T$
(E, T)	$E \rightarrow T * F$
(E, F)	$E \rightarrow (E)$
(E, I)	$E \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T, T)	$T \rightarrow T * F$
(T, F)	$T \rightarrow (E)$
(T, I)	$T \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F, F)	$F \rightarrow (E)$
(F, I)	$F \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I, I)	$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Example:

Pairs	Productions
…	…
(T, T)	**T → T * F**
(T, F)	**T → (E)**
(T, I)	**T → a | b | Ia |Ib | I0 | I1**
…	…

I → a | b | Ia | Ib | I0 | I1

E → E + T | T * F | (E) | a | b | Ia | Ib | I0 | I1

T → T * F | (E) | a | b | Ia | Ib | I0 | I1

F → (E) | a | b | Ia | Ib | I0 | I1

Final Simplification

Chomsky Normal Form (CNF)

Starting with a CFL grammar with the preliminary simplifications performed

1. Arrange that all bodies of length 2 or more to consists only of variables.
2. Break bodies of length 3 or more into a cascade of productions, each with a body consisting of two variables.

Final Simplification

Step 1: For every terminal α that appears in a body of length 2 or more create a new variable that has only one production.

E → E + T | T * F | (E) | a | b | Ia | Ib | I0 | I1
T → T * F | (E) | a | b | Ia | Ib | I0 | I1
F → (E) | a | b | Ia | Ib | I0 | I1
I → a | b | Ia | Ib | I0 | I1

E → EPT | TMF | LER | a | b | IA | IB | IZ | IO
T → TMF | LER | a | b | IA | IB | IZ | IO
F → LER | a | b | IA | IB | IZ | IO
I → a | b | IA | IB | IZ | IO
A → a   B → b   Z → 0   O → 1
P → +   M → *   L → (R →)

Final Simplification

Step 2: Break bodies of length 3 or more adding more variables

E → E**PT** | T**MF** | L**ER** | a | b | IA | IB | IZ | IO
T → T**MF** | L**ER** | a | b | IA | IB | IZ | IO
F → L**ER** | a | b | IA | IB | IZ | IO
I → a | b | IA | IB | IZ | IO
A → a B → b Z → 0 O → 1
P → + M → * L → (R →)

$C_1 \rightarrow PT$
$C_2 \rightarrow MF$
$C_3 \rightarrow ER$

Greibach Normal Form

A context free grammar is said to be in **Greibach Normal Form** if all productions are in the following form:

$$A \rightarrow \alpha X$$

- A is a non terminal symbols
- α is a terminal symbol
- X is a sequence of non terminal symbols.
  It may be empty.

Greibach Normal Form

Example:

$S \rightarrow XA \mid BB$
$B \rightarrow b \mid SB$
$X \rightarrow b$
$A \rightarrow a$

CNF

$S = A_1$
$X = A_2$
$A = A_3$
$B = A_4$

New Labels

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow b \mid A_1A_4$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

Updated CNF

Greibach Normal Form

Example:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow b \mid A_1A_4$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

First Step $\boxed{A_i \rightarrow A_jX_k \quad j > i}$

X_k is a string of zero or more variables

✕ $A_4 \rightarrow A_1A_4$

Greibach Normal Form

Example:

First Step $\boxed{A_i \rightarrow A_j X_k \quad j > i}$

$A_4 \rightarrow A_1 A_4$

$A_4 \rightarrow A_2 A_3 A_4 \mid A_4 A_4 A_4 \mid b$

$A_4 \rightarrow b A_3 A_4 \mid A_4 A_4 A_4 \mid b$

$A_1 \rightarrow A_2 A_3 \mid A_4 A_4$

$A_4 \rightarrow b \mid A_1 A_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Example:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow bA_3A_4 \mid A_4A_4A_4 \mid b$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

Second Step

Eliminate Left Recursions

✕ $A_4 \rightarrow A_4A_4A_4$

Greibach Normal Form

Example:

Second Step

Eliminate Left Recursions

$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$

$Z \rightarrow A_4A_4 \mid A_4A_4Z$

$A_1 \rightarrow A_2A_3 \mid A_4A_4$

$A_4 \rightarrow bA_3A_4 \mid A_4A_4A_4 \mid b$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Greibach Normal Form

Example:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$
$Z \rightarrow A_4A_4 \mid A_4A_4Z$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

$$\boxed{A \rightarrow \alpha X}$$

GNF

Greibach Normal Form

Example:

$$A_1 \rightarrow A_2A_3 \mid A_4A_4$$
$$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$$
$$Z \rightarrow A_4A_4 \mid A_4A_4Z$$
$$A_2 \rightarrow b$$
$$A_3 \rightarrow a$$

$$A_1 \rightarrow bA_3 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$$

$$Z \rightarrow bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$$

Greibach Normal Form

Example:

$A_1 \rightarrow bA_3 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$

$A_4 \rightarrow bA_3A_4 \mid b \mid bA_3A_4Z \mid bZ$

$Z \rightarrow bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4 \mid bA_3A_4A_4 \mid bA_4 \mid bA_3A_4ZA_4 \mid bZA_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Grammar in Greibach Normal Form

Regular Expressions

Definition of a Regular Expression

- R is a regular expression if it is:
  1. **a** for some *a* in the alphabet $\sum$, standing for the language {a}
  2. ε, standing for the language {ε}
  3. Ø, standing for the empty language
  4. R_1+R_2 where R_1 and R_2 are regular expressions, and + signifies union (sometimes | is used)
  5. R_1R_2 where R_1 and R_2 are regular expressions and this signifies concatenation
  6. R* where R is a regular expression and signifies closure
  7. (R) where R is a regular expression, then a parenthesized R is also a regular expression

This definition may seem circular, but 1-3 form the basis
Precedence: Parentheses have the highest precedence, followed by *(iteration), concatenation, and then union(ICU)

RE Examples

- L(**001**) = {001}
- L(**0+10***) = { 0, 1, 10, 100, 1000, 10000, … }
- L(**0*10***) = {1, 01, 10, 010, 0010, …}   i.e. {w | w has exactly a single 1}
- L(∑∑)* = {w | w is a string of even length}
- L((**0(0+1**))*) = { ε, 00, 01, 0000, 0001, 0100, 0101, …}
- L((**0+ε)(1+ ε**)) = {ε, 0, 1, 01}
- L(1∅) = ∅   ; concatenating the empty set to any set yields the empty set.
- Rε = R
- R+∅ = R

- Note that R+ε may or may not equal R (we are adding ε to the language)
- Note that R∅ will only equal R if R itself is the empty set.

Regular Expressions

- Regular expressions
- describe regular languages

$$(a+b\cdot c)^*$$

- Example:

$$\{a,bc\}^* = \{\lambda, a, bc, aa, abc, bca, \ldots\}$$

- describes the language

Languages of Regular Expressions

$L(r)$: language of regular expression r

- Example

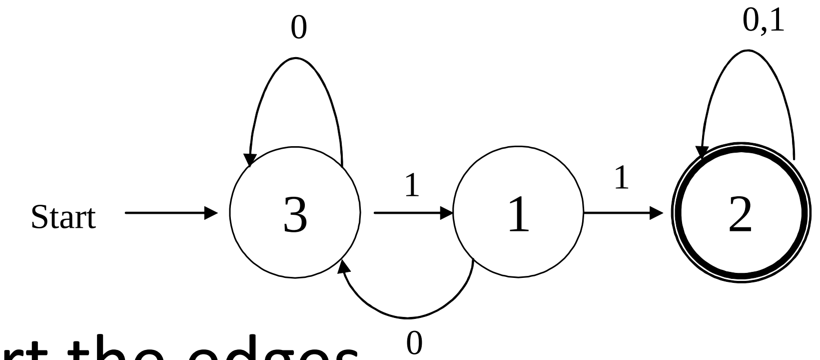
$$L((a+b\cdot c)^*)=\{\lambda, a, bc, aa, abc, bca, \ldots\}$$

Equivalence of FA and RE

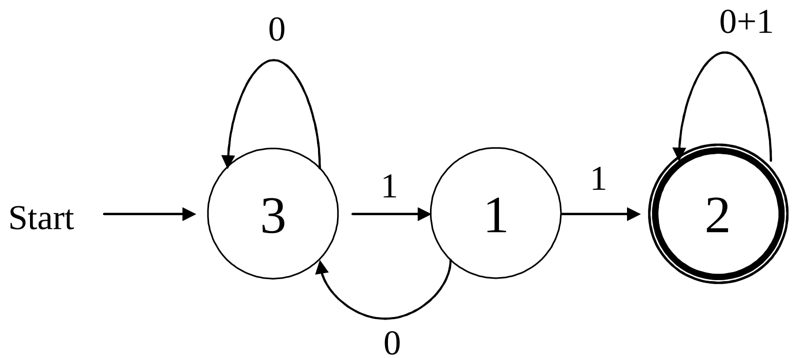
- Finite Automata and Regular Expressions are equivalent. To show this:
  - Show we can express a DFA as an equivalent RE
  - Show we can express a RE as an ε-NFA. Since the ε-NFA can be converted to a DFA and the DFA to an NFA, then RE will be equivalent to all the automata we have described.

DFA→RE Example

- Convert the following to a RE



- First convert the edges to RE's:



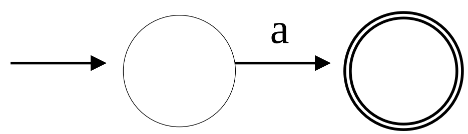
Converting a RE to an Automata

- We have shown we can convert an automata to a RE. To show equivalence we must also go the other direction, convert a RE to an automaton.
- We can do this easiest by converting a RE to an ε-NFA
  - Inductive construction
  - Start with a simple basis, use that to build more complex parts of the NFA

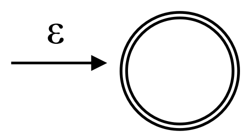
RE to ε-NFA

- Basis:

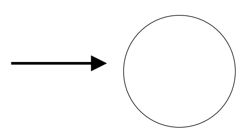
R=a



R=ε

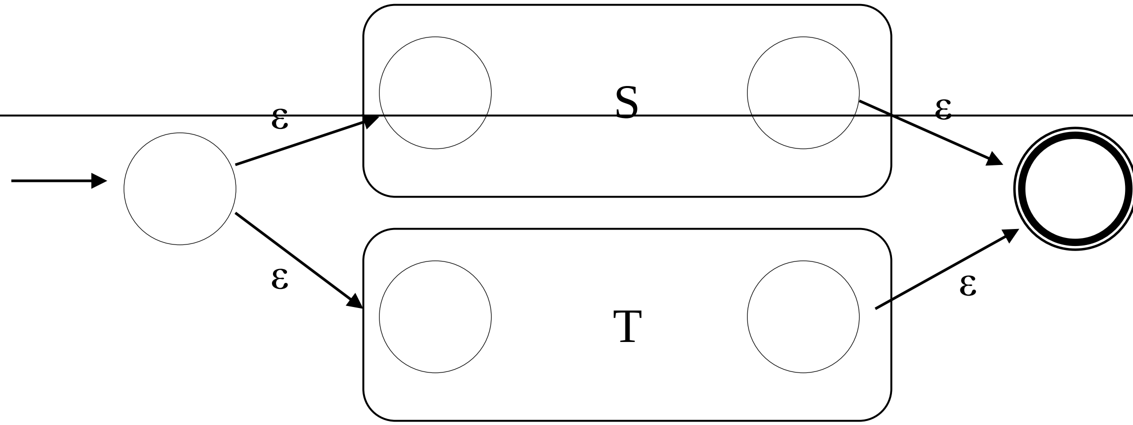


R=Ø

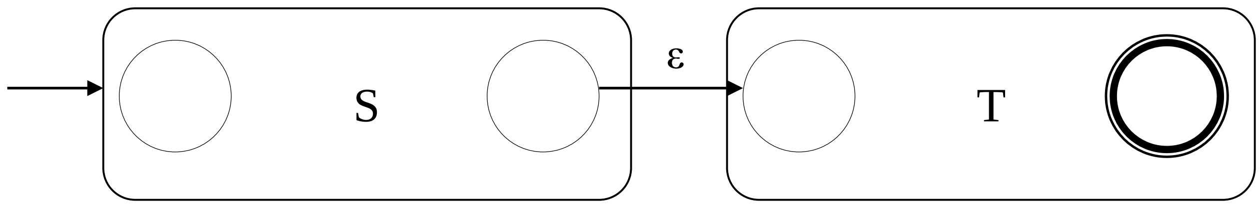


Next slide: More complex RE’s

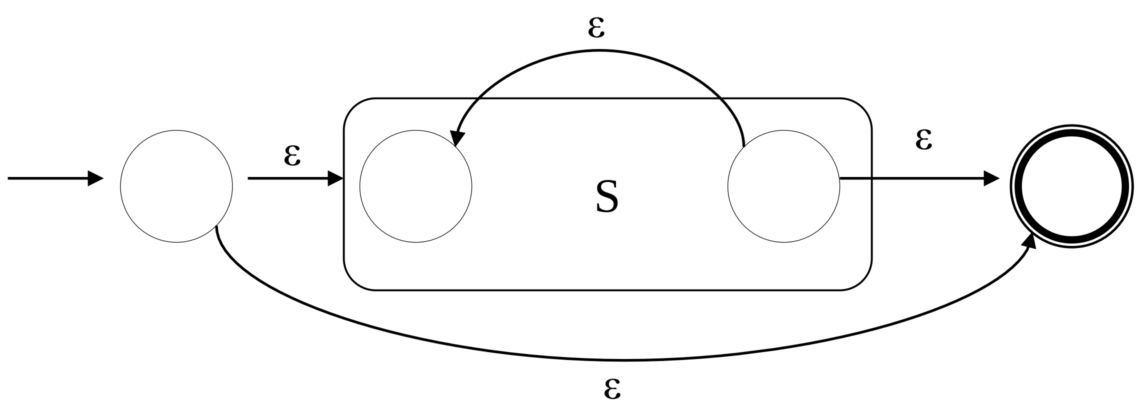
R=S+T



R=ST



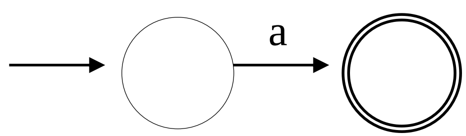
$R=S^*$



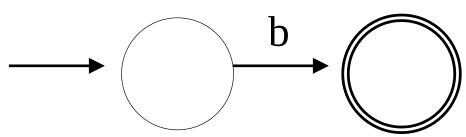
RE to ε-NFA Example

- Convert R= (ab+a)* to an NFA
  - We proceed in stages, starting from simple elements and working our way up

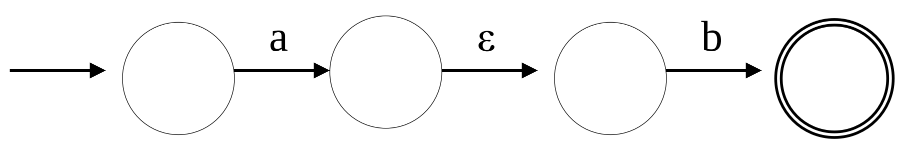
a



b

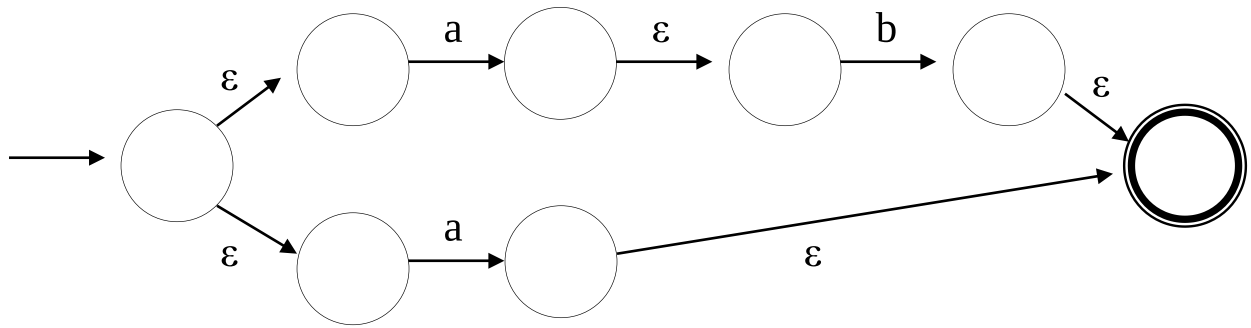


ab

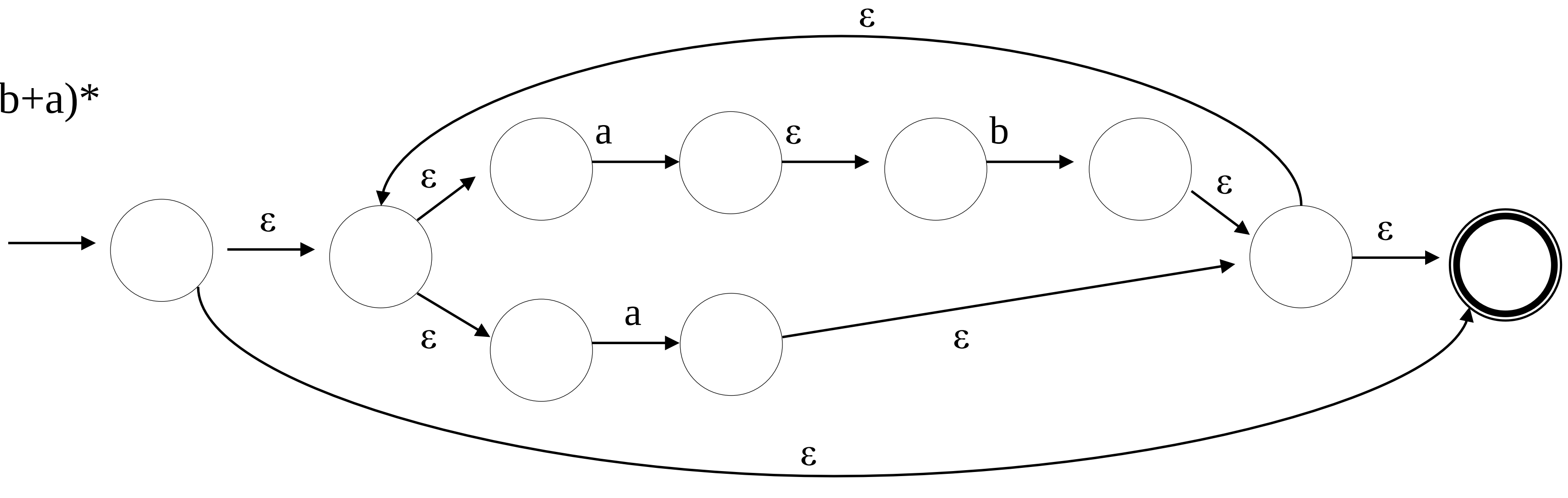


RE to ε-NFA Example (2)

ab+a



(ab+a)*



Pushdown Automata

Formal Definition of a PDA

- A pushdown automaton (PDA) is a seven-tuple:

  $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

Q	A finite set of states
Σ	A finite input alphabet
Γ	A finite stack alphabet
q_0	The initial/starting state, q_0 is in Q
z_0	A starting stack symbol, is in Γ // need not always remain at the bottom of stack
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, where

$$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$$

Pushdown Automaton

- A pushdown automaton (PDA) is an abstract model machine similar to the FSA

- It has a finite set of states. However, in addition, it has a pushdown stack. Moves of the PDA are as follows:
- 1. An input symbol is read and the top symbol on the stack is read.
- 2. Based on both inputs, the machine enters a new state and writes zero or more symbols onto the pushdown stack.
- 3. Acceptance of a string occurs if the stack is ever empty. (Alternatively, acceptance can be if the PDA is in a final state. Both models can be shown to be equivalent.)

Power of PDAs

- PDAs are more powerful than FSAs.
- $a^n b^n$, which cannot be recognized by an FSA, can easily be recognized by the PDA.
- Stack all a symbols and, for each b, pop an a off the stack.
- If the end of input is reached at the same time that the stack becomes empty, the string is accepted.

- It is less clear that the languages accepted by
- PDAs are equivalent to the context-free languages.

NDPDAs are different from DPDAs

- What is the relationship between deterministic
- PDAs and nondeterministic PDAs? They are different.

- Consider the set of palindromes, strings reading the same forward and backward, generated by the grammar
- $S \rightarrow 0S0 \mid 1S1 \mid 2$
- We can recognize such strings by a deterministic PDA:
  - 1. Stack all 0s and 1s as read.
  - 2. Enter a new state upon reading a 2.
  - 3. Compare each new input to the top of stack, and pop stack.
- However, consider the following set of palindromes:
- $S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1$
- In this case, we never know where the middle of the string is. To recognize these palindromes, the automaton must guess where the middle of the string is (i.e., is nondeterministic).

- The PDA can be represented by

$$M = (Q, \Sigma, \Gamma, \delta, s, F)$$

where Σ is the alphabet of input symbols and Γ is the alphabet of stack symbols.

- The set of all strings accepted by a PDA M is denoted by $L(M)$. We also say that the language $L(M)$ is accepted by M.

- The transition diagram of a PDA is an alternative way to represent the PDA.
- For $M = (Q, \Sigma, \Gamma, \delta, s, F)$, the transition diagram of M is an edge-labeled digraph $G=(V, E)$ satisfying the following:

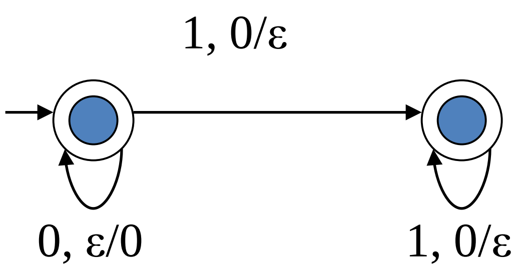
$V = Q$ ($s = \rightarrow$, $f =$ for $f \in F$)

$E = \{ q \xrightarrow{a,\ v/u} p \mid (p,u) \in \delta(q, a, v) \}$.

Example 1. Construct PDA to accept

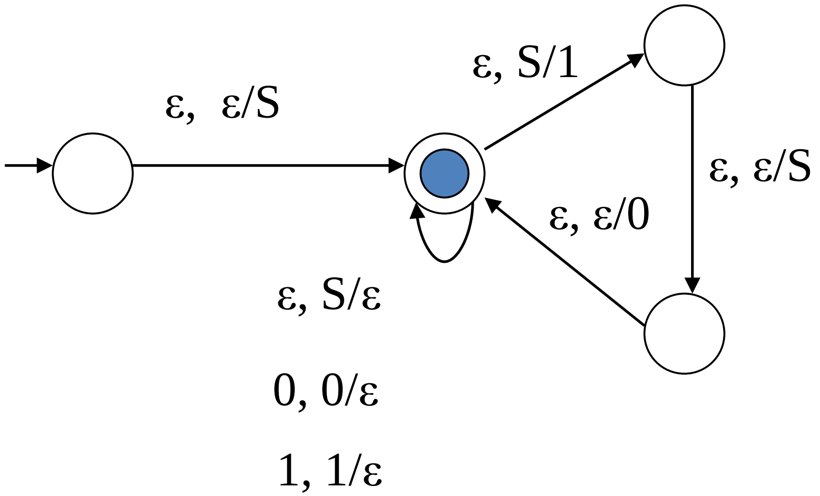
$$L = \{0^n 1^n \mid n \geq 0\}$$

Solution 1.



Solution 2.

Consider a CFG
$G = (\{S\}, \{0,1\}, \{S \rightarrow \varepsilon \mid 0S1\}, S)$.



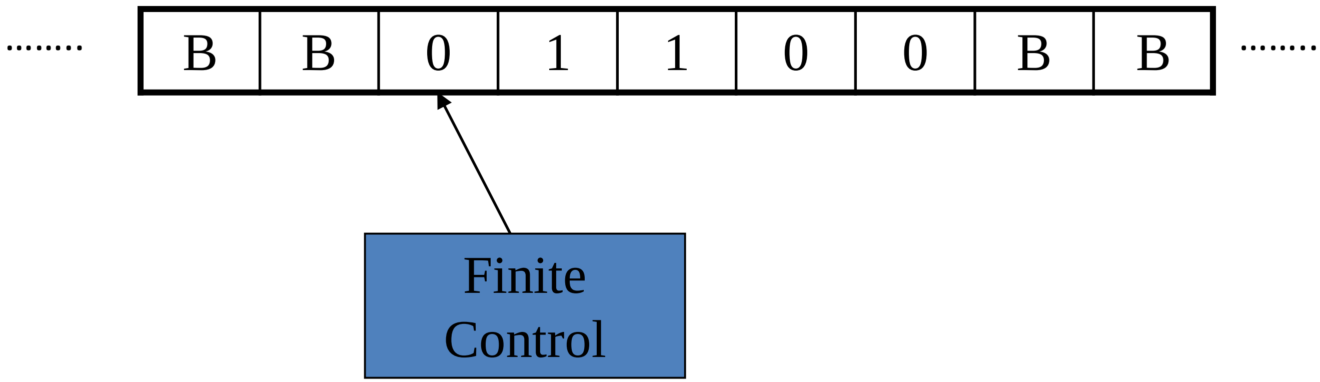
- TMs model the computing capability of a general purpose computer, which informally can be described as:
  - Effective procedure
    - Finitely describable
    - Well defined, discrete, “mechanical” steps
    - Always terminates
  - Computable function
    - A function computable by an effective procedure

- TMs formalize the above notion.

- **Church-Turing Thesis:** There is an effective procedure for solving a problem if and only if there is a TM that halts for all inputs and solves the problem.
  - There are many other computing models, but all are equivalent to or subsumed by TMs. *There is no more powerful machine* (Technically cannot be proved).

- DFAs and PDAs do not model all effective procedures or computable functions, but only a subset.

Deterministic Turing Machine (DTM)



- Two-way, infinite tape, broken into cells, each containing one symbol.
- Two-way, read/write tape head.
- An input string is placed on the tape, padded to the left and right infinitely with blanks, read/write head is positioned at the left end of input string.
- Finite control, i.e., a program, containing the position of the read head, current symbol being scanned, and the current state.
- In one move, depending on the current state and the current symbol being scanned, the TM 1) changes state, 2) prints a symbol over the cell being scanned, and 3) moves its' tape head one cell left or right.
- Many modifications possible, but Church-Turing declares equivalence of all.

Formal Definition of a DTM

- A DTM is a seven-tuple:

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

Q	A finite set of states
Σ	A finite input alphabet, which is a subset of $\Gamma - \{B\}$
Γ	A finite tape alphabet, which is a strict superset of Σ
B	A distinguished blank symbol, which is in Γ
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A next-move function, which is a *mapping* (i.e., may be undefined) from $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$

Intuitively, $\delta(q,s)$ specifies the next state, symbol to be written, and the direction of tape head movement by M after reading symbol s while in state q.

- **Example #1:** {w | w is in {0,1}* and w ends with a 0}

0

00

10

10110

Not ε

$Q = \{q_0, q_1, q_2\}$

$\Gamma = \{0, 1, B\}$

$\Sigma = \{0, 1\}$

$F = \{q_2\}$

δ:

	0	1	B
$\to q_0$	$(q_0, 0, R)$	$(q_0, 1, R)$	(q_1, B, L)
q_1	$(q_2, 0, R)$	-	-
q_2^*	-	-	-

– q_0 is the start state and the "scan right" state, until hits B
– q_1 is the verify 0 state
– q_2 is the final state

- **Example #2:** $\{0^n1^n \mid n \geq \mathbf{1}\}$

	0	1	X	Y	B
->q_0	(q_1, X, R)	-	-	(q_3, Y, R)***0's finished***	-
q_1	(q_1, 0, R)***ignore1***	(q_2, Y, L)	-	(q_1, Y, R) ***ignore2***	- (more 0's)
q_2	(q_2, 0, L) ***ignore2***	-	(q_0, X, R)	(q_2, Y, L) ***ignore1***	-
q_3	-	- (more 1's)	-	(q_3, Y, R) ***ignore***	(q_4, B, R)
q_4*	-	-	-	-	-

- **Sample Computation:** (on 0011), *presume state q looks rightward*

$q_0$0011BB.. |— X$q_1$011
|— X0$q_1$11
|— X$q_2$0Y1
|— q_2X0Y1
|— X$q_0$0Y1
|— XXq_1Y1
|— XXY$q_1$1
|— XXq_2YY
|— Xq_2XYY
|— XXq_0YY
|— XXYq_3Y B…
|— XXYYq_3 BB…
|— XXYYBq_4

- **Same Example #2:** $\{0^n1^n \mid n \geq 1\}$

	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

Logic: cross 0's with X's, scan right to look for corresponding 1, on finding it cross it with Y, and scan left to find next leftmost 0, keep iterating until no more 0's, then scan right looking for B.

- The TM matches up 0's and 1's
- q_1 is the "scan right" state, looking for 1
- q_2 is the "scan left" state, looking for X
- q_3 is "scan right", looking for B
- q_4 is the final state

Can you extend the machine to include n=0?
How does the input-tape look like for string epsilon?

- **Other Examples:**

000111	00
11	001
011	

Formal Definitions for DTMs

- Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM.

- **Definition:** An *instantaneous description* (ID) is a triple $\alpha_1 q \alpha_2$, where:

  – q, the current state, is in Q
  – $\alpha_1\alpha_2$, is in Γ^*, and is the current tape contents up to the rightmost non-blank symbol, or the symbol to the left of the tape head, whichever is rightmost
  – The tape head is currently scanning the first symbol of α_2
  – At the start of a computation $\alpha_1 = \boldsymbol{\varepsilon}$
  – If $\alpha_2 = \boldsymbol{\varepsilon}$ then a blank is being scanned

- **Example:** (for TM #1)

$q_0 0011$ $\quad$ $Xq_1 011$ $\quad$ $X0q_1 11$ $\quad$ $Xq_2 0Y1$ $\quad$ $q_2 X0Y1$

$Xq_0 0Y1$ $\quad$ $XXq_1 Y1$ $\quad$ $XXYq_1 1$ $\quad$ $XXq_2 YY$ $\quad$ $Xq_2 XYY$

$XXq_0 YY$ $\quad$ $XXYq_3 Y$ $\quad$ $XXYYq_3$ $\quad$ $XXYYBq_4$

- Suppose the following is the current ID of a DTM

$$x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n$$

Case 1) $\delta(q, x_i) = (p, y, L)$

(a) if $i = 1$ then $qx_1x_2\ldots x_{i-1}x_ix_{i+1}\ldots x_n \mid\!\!— pByx_2\ldots x_{i-1}x_ix_{i+1}\ldots x_n$

(b) else $x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n \mid\!\!— x_1x_2\ldots x_{i-2}px_{i-1}yx_{i+1}\ldots x_n$

- If any suffix of $x_{i-1}yx_{i+1}\ldots x_n$ is blank then it is deleted.

Case 2) $\delta(q, x_i) = (p, y, R)$

$x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n \mid\!\!— x_1x_2\ldots x_{i-1}ypx_{i+1}\ldots x_n$

- If $i>n$ then the ID increases in length by 1 symbol

$x_1x_2\ldots x_nq \mid\!\!— x_1x_2\ldots x_nyp$

L is Recursively enumerable:

TM exist: M_0, M_1, *...*

They accept string in L, and do not accept any string outside L

L is Recursive:

at least one TM halts on L and on $\sum^*-L$*, others may or may not*

L is Recursively enumerable but not Recursive:

TM exist: M_0, M_1, *...*

but <u>none</u> halts on <u>all</u> x in $\sum^*-L$

M_0 *goes on infinite loop on a string p in* $\sum^*-L$*, while* M_1 *on q in* $\sum^*-L$

However, each correct TM accepts each string in L, and none in $\sum^*-L$

L is not R.E*:*

no TM exists

Modifications of the Basic TM Model

- **Other (Extended) TM Models:**
  - One-way infinite tapes
  - Multiple tapes and tape heads
  - Non-Deterministic TMs
  - Multi-Dimensional TMs (n-dimensional tape)
  - Multi-Heads
  - Multiple tracks

All of these extensions are equivalent to the basic DTM model

The Halting Problem - Background

- **Definition:** A <u>decision problem</u> is a problem having a yes/no answer (that one presumably wants to solve with a computer). Typically, there is a list of parameters on which the problem is based.
  - Given a list of numbers, is that list sorted?
  - Given a number x, is x even?
  - Given a C program, does that C program contain any syntax errors?
  - Given a TM (or C program), does that TM contain an infinite loop?

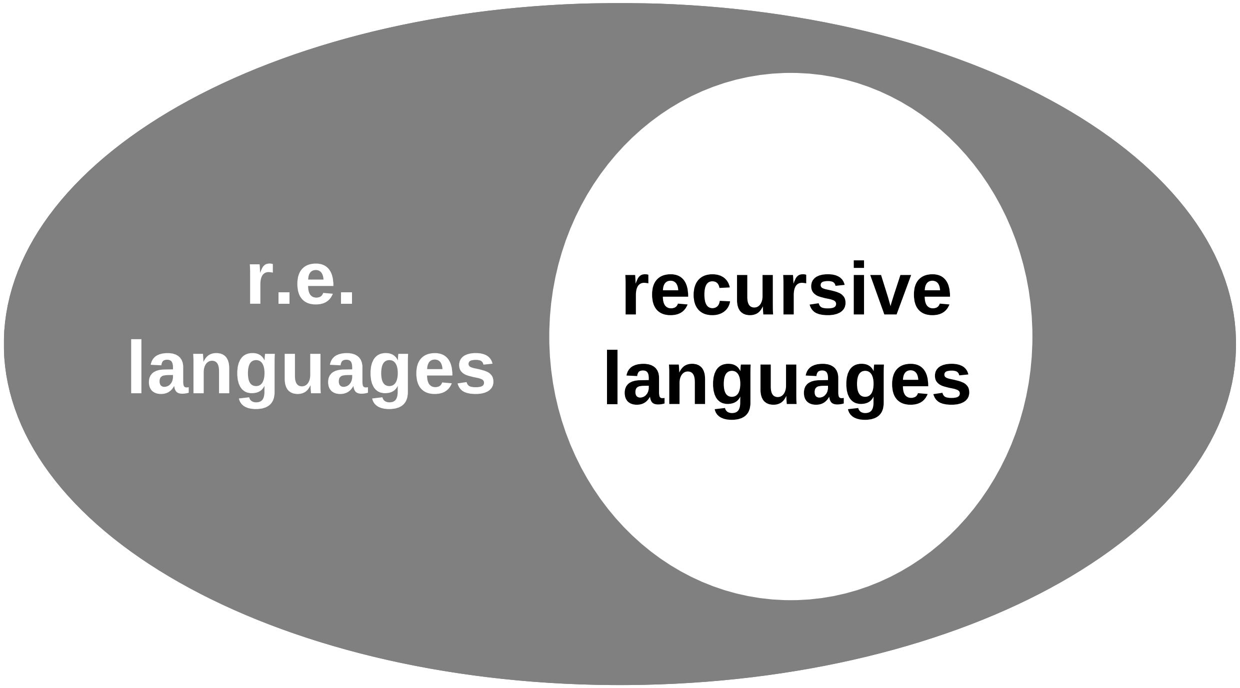
  From a practical perspective, many decision problems do not seem all that interesting. However, from a theoretical perspective they are for the following two reasons:
  - Decision problems are more convenient/easier to work with when proving complexity results.
  - Non-decision *counter-parts* can always be created & are typically at least as difficult to solve.

- **Notes:**
  - The following terms and phrases are analogous:

Algorithm	-	A halting TM program
Decision Problem	-	A language *(will show shortly)*
(un)Decidable	**-**	**(non)Recursive**

A language is called Turing-recognizable or recursively enumerable (r.e.) if some TM recognizes it.

A language is called decidable or recursive if some TM decides it.



THE HALTING PROBLEM

$HALT_{TM}$ = { (M,w) | M is a TM that halts on string w }

Theorem: $HALT_{TM}$ is undecidable

Proof: Assume, for a contradiction, that TM H decides $HALT_{TM}$

We use H to construct a TM D that decides A_{TM}

On input (M,w), D runs H on (M,w)

If H rejects then reject

If H accepts, run M on w until it halts:

Accept if M accepts and
Reject if M rejects

References

- http://www.cse.cuhk.edu.hk/~andrejb/csc3130
- cs.www.duke.edu › courses › fall07 › cps102 › lectures › lecture02
- ie.technion.ac.il › courses › verification › C4.1_NFA+Buchi.ppt
- curry.ateneo.net › ~jpv › cs130-L6-Grammars
- montefiore.www.ulg.ac.be › cours › psfiles › calc-chap3
- cit.courses.cornell.edu › PPT › INFO-2950-Languages-and-Grammars
- cs.www.rpi.edu › ~moorthy › Courses › modcomp › slides › PDA
- cs.fit.edu › ~dmitra › FormaLang › Lectures › PushdownAutomata
- cs.www.cmu.edu › ~emc › flac09 › lectures
- cse.www.iitd.ernet.in › ~naveen › courses › COL352 › slides
- cs.www.rpi.edu › ~moorthy › Courses › modcomp › slides › NFA
- cs.fit.edu › ~dmitra › FormaLang › Lectures › FiniteAutomata