What do computers do?

– Take an *input*, process, and produce correct *output*.

- What is input?

  A finite *string* on a finite *alphabet (*a set of characters).

- What is Output?

  "*Input processed successfully*," Or "*not*".

- In other words: *True / False*.

Alphabet and Strings

- Symbol – An atomic unit, such as a digit, character, lower-case letter, etc. Sometimes a word. *[Formal language does not deal with the "meaning" of the symbols.]*

- Alphabet – A <u>finite</u> set of symbols, usually denoted by Σ.

  $\Sigma = \{0, 1\}$ $\Sigma = \{0, a, \ , 4\}$ $\Sigma = \{a, b, c, d\}$

- String – A <u>finite</u> length sequence of symbols, presumably from some alphabet.

  $u=\varepsilon$ $w = 0110$ $y = 0aa$ $x = aabcaa$ $z = 111$

  special string: ε (also denoted by λ)

  concatenation: $wz = 0110111$
  length: $|w| = 4$ $|x| = 6$ but $|u| = 0$
  reversal: $y^R = aa0$

- Some special sets of strings:

  Σ^* All strings of symbols from Σ
  Σ^+ $\Sigma^* - \{\varepsilon\}$

- Example:

  $\Sigma = \{0, 1\}$
  $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001,\ldots\}$
  $\Sigma^+ = \{0, 1, 00, 01, 10, 11, 000, 001,\ldots\}$

- A (formal) language is:

  1) A set of strings from some alphabet (finite or infinite), in other words…
  2) any subset L of Σ^*

- Some special languages:

  $\{\}$ The empty set/language, containing no strings
  $\{\varepsilon\}$ A language containing one string, the empty string.

Formal Language

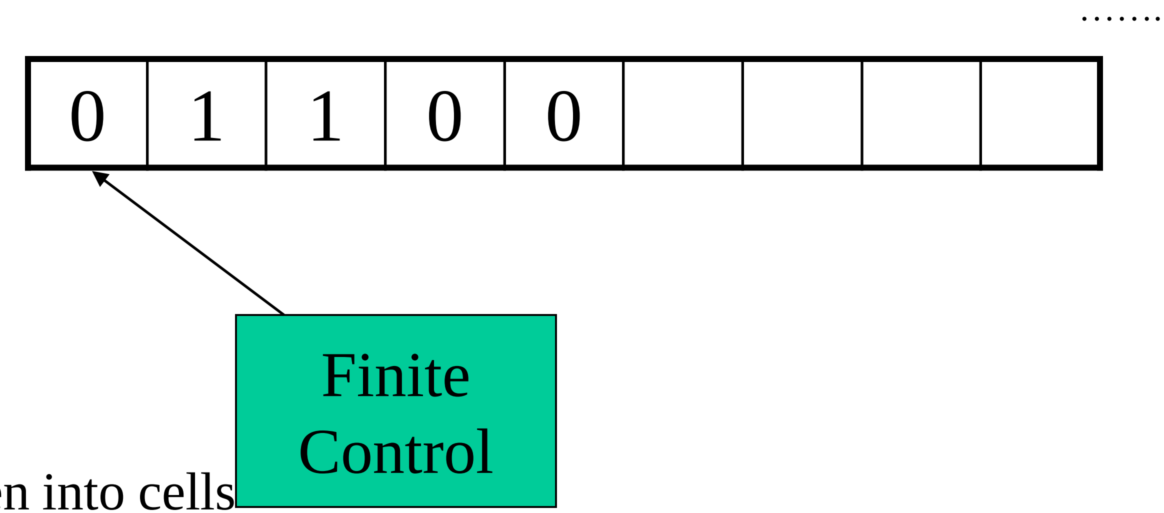
- Finite set of alphabet Σ:

  e.g., {0, 1}, {a, b, c}, { ‘{‘, ‘}’ }

- Language L is a subset of strings on Σ,
  - e.g., {00, 110, 01} a finite language,
  - or, {strings starting with 0} an infinite language

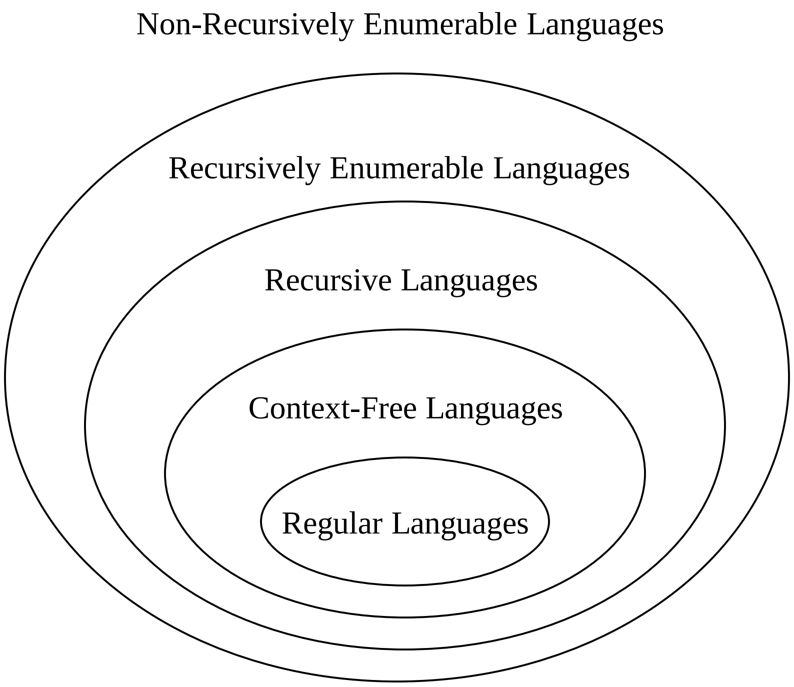
- Σ^* is a special language with all possible strings on Σ

Finite State Machine



- Tape, broken into cells
- Tape head.
- Finite control, i.e., a program, containing the position of the read head, current symbol being scanned, and the current "state", etc.
- A string is placed on the tape, read head is positioned at the left end, and the *machine* reads the string one symbol at a time until all symbols have been read,
- and then either *accepts* or *rejects* the input (to be in the language or not)

Hierarchy of languages



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

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

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

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

- A derivation of "a dog runs":

$$
\begin{aligned}
\langle sentence \rangle &\Rightarrow \langle noun\_phrase \rangle \ \langle predicate \rangle \\
&\Rightarrow \langle noun\_phrase \rangle \ \langle verb \rangle \\
&\Rightarrow \langle article \rangle \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow a \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow a \ dog \ \langle verb \rangle \\
&\Rightarrow a \ dog \ runs
\end{aligned}
$$

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: $V^* = \{Aba, BBa, bAA, cab \ldots\}$
- 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

Grammar OR Phrase-Structure Grammars

- A *phrase-structure grammar* (abbr. PSG) $G = (V,T,S,P)$ is a 4-tuple, in which:
  - V is a vocabulary (set of symbols)
    - The "template vocabulary" of the language.
  - $T \subseteq V$ is a set of symbols called *terminals*
    - Actual symbols of the language.
    - Also, $N :\equiv V - T$ is a set of special "symbols" called *nonterminals*. (Representing concepts like "noun")
  - $S \in N$ is a special nonterminal, the *start symbol*.
    - in our example the start symbol was "sentence".
  - P is a set of *productions* (to be defined).
    - Rules for substituting one sentence fragment for another
    - Every production rule must contain at least one nonterminal on its left side.

Example 1: Assignment statements

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

  $S \rightarrow i = E$

  $E \rightarrow n$

  $E \rightarrow i$

  $E \rightarrow E + E$

  $E \rightarrow E * E$

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
- $L(G) = \{ w \text{ in } T^* \mid S \Rightarrow^{*}_{G} w \}$,
  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

Language

- Let G(V,T,S,P) be a phrase-structure grammar. The language generated by G (or the language of G) denoted by L(G) , is the set of all strings of terminals that are derivable from the starting state S.

$$L(G)= \{w \in T^* \mid S \Rightarrow^* w\}$$

Language L(G)

- **EXAMPLE:**

- Let $G = (V, T, S, P)$, where $V = \{a, b, A, S\}$, $T = \{a, b\}$, S is a start symbol and $P = \{S \longrightarrow aA, S \longrightarrow b, A \longrightarrow aa\}$.

- The language of this grammar is given by L (G) = $\{b, aaa\}$;

1. we can derive aA from using $S \longrightarrow aA$, and then derive aaa using $A \longrightarrow aa$.

2. We can also derive b using $S \longrightarrow b$.

Another example

- Grammar: $G=(V,T,S,P)$ $T=\{a,b\}$ $P =$ $S \to aSb$, $S \to \lambda$

  $V=\{a,b,S\}$

- Derivation of sentence ab:

$$S \Rightarrow aSb \Rightarrow ab$$

$$S \to aSb \qquad S \to \lambda$$

- Grammar: $$S \to aSb$$ $$S \to \lambda$$

- Derivation of sentence $aabb$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

$$S \to aSb \qquad\qquad S \to \lambda$$

- Other derivations:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$$

$$\Rightarrow aaaaSbbbb \Rightarrow aaaabbbb$$

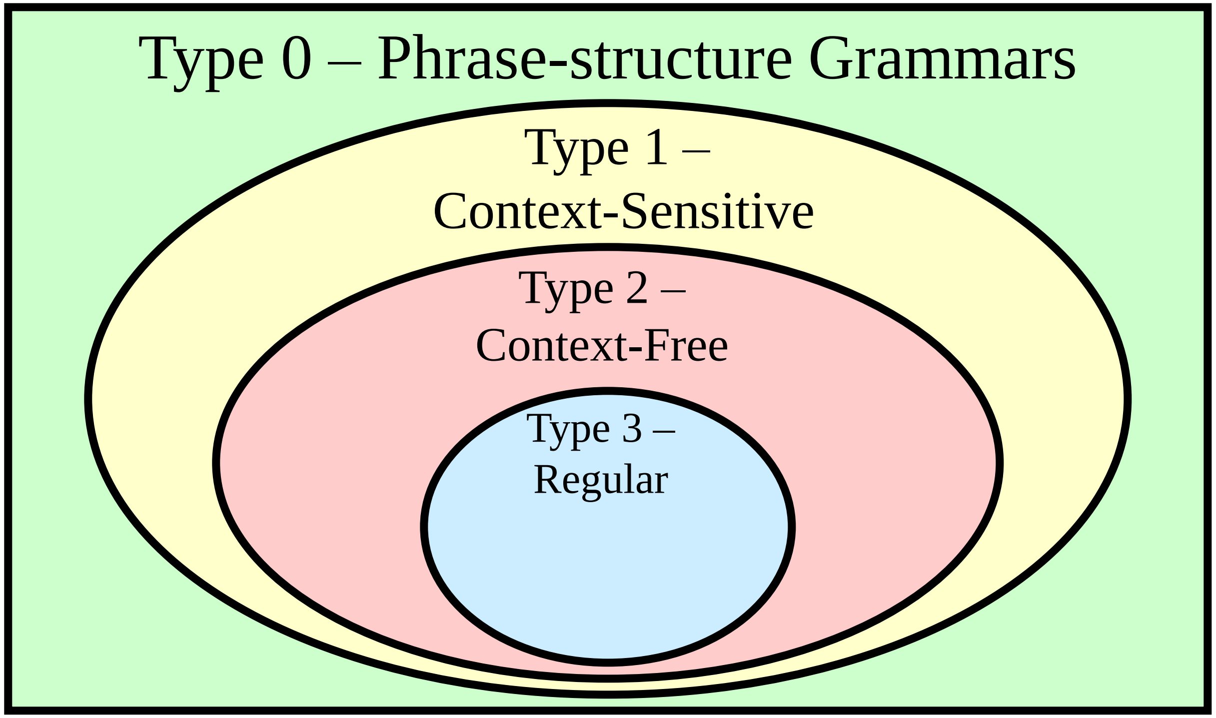
So, what's the language of the grammar with the productions?

$$S \to aSb$$

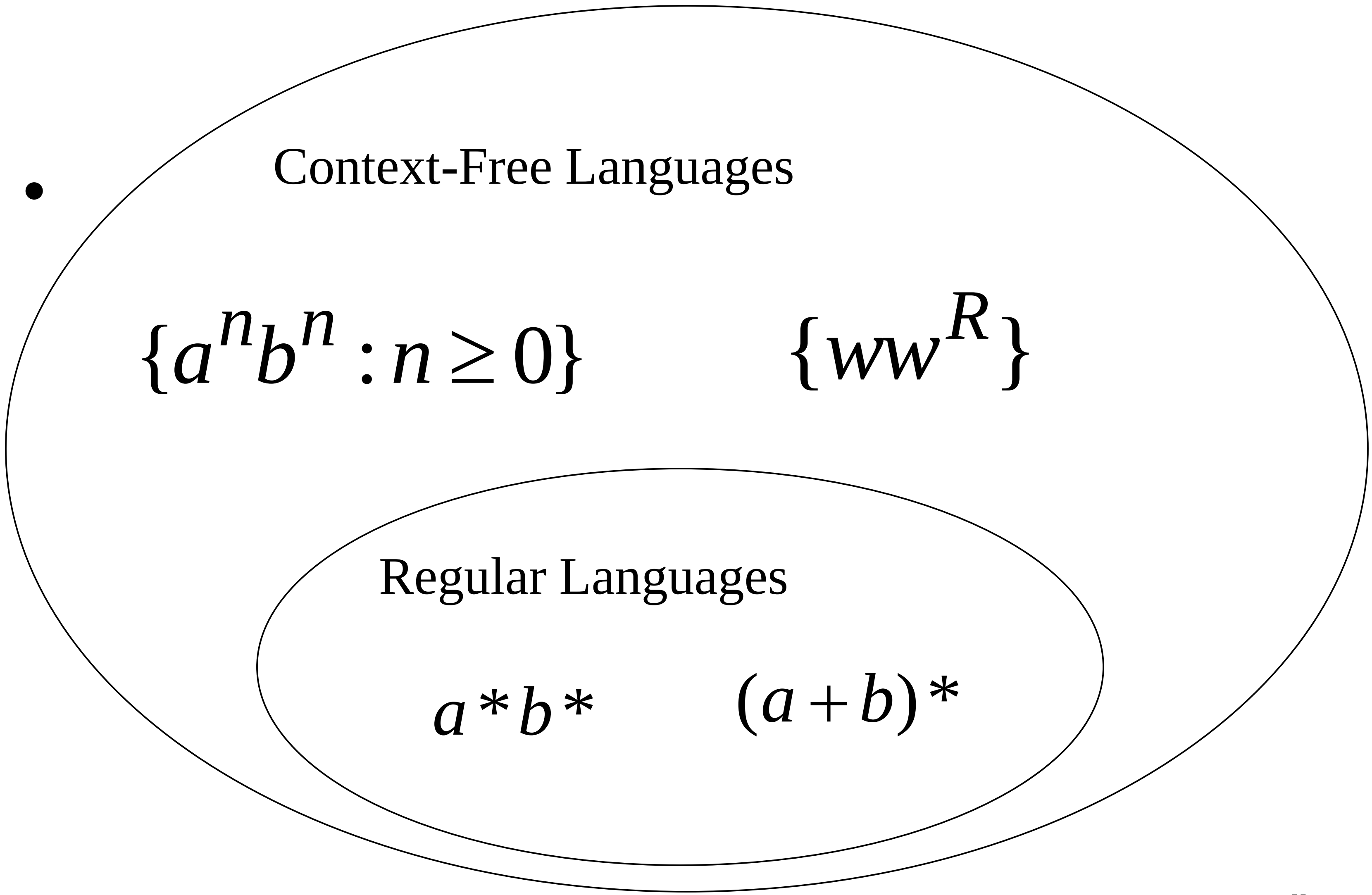
$$S \to \lambda$$

Types of Grammars - Chomsky hierarchy of languages

- Venn Diagram of Grammar Types:



•

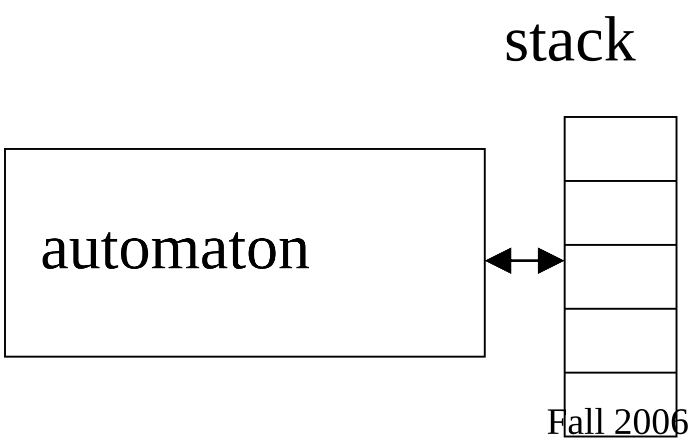


Context-Free Languages
$\{a^n b^n : n \geq 0\}$
$\{ww^R\}$
Regular Languages
a^*b^*
$(a+b)^*$

Context-Free Languages

Context-Free
Grammars

Pushdown
Automata



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 $A \rightarrow w$ 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.

Context-Free Language:

- A language L is context-free
- if there is a context-free grammar G
- with $L = L(G)$

Another Example

Context-free grammar : G

$$S \to aSa \mid bSb \mid \lambda$$

Example derivations:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaba$$

$$L(G) = \{ww^R : \quad w \in \{a,b\}^*\}$$

Palindromes of even length

Derivation Order
and
Derivation Trees

Derivation Order

Consider the following example grammar
with 5 productions:

$$\boxed{\begin{array}{lll} 1.\ S \to AB & 2.\ A \to aaA & 4.\ B \to Bb \\ & 3.\ A \to \lambda & 5.\ B \to \lambda \end{array}}$$

Ambiguity

Grammar for mathematical expressions

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

Example strings:

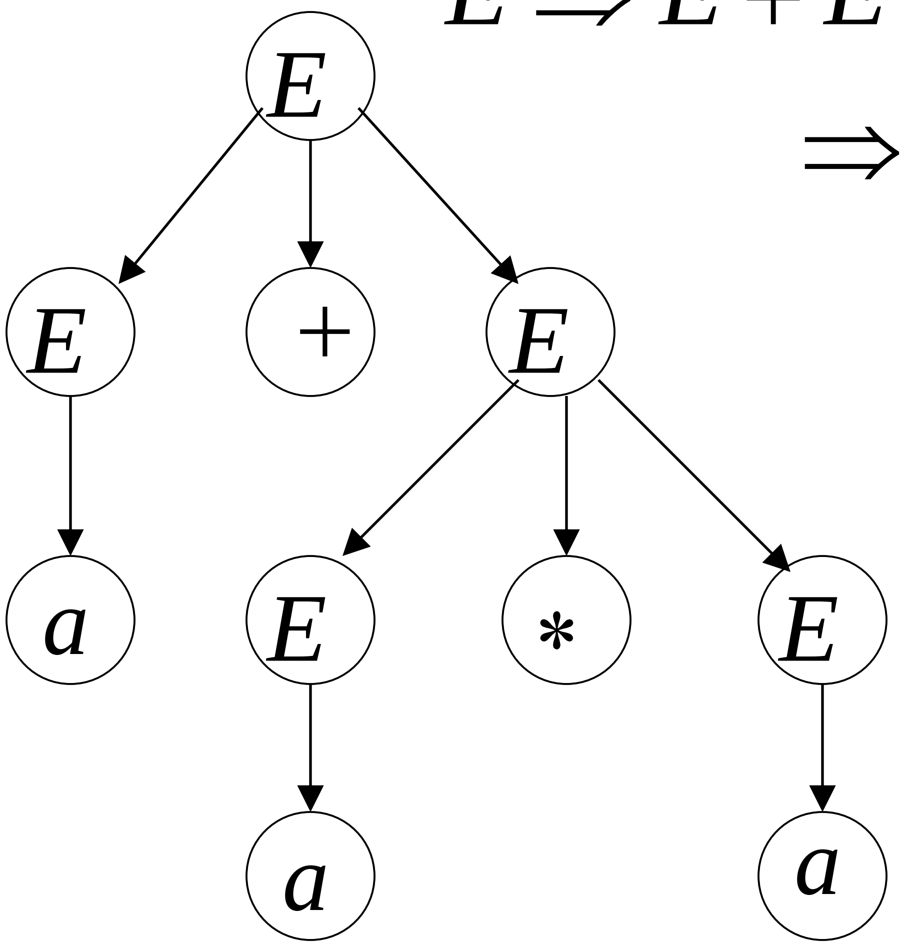
$$(a+a)*a+(a+a*(a+a))$$

Denotes any number

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

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

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



A leftmost derivation for

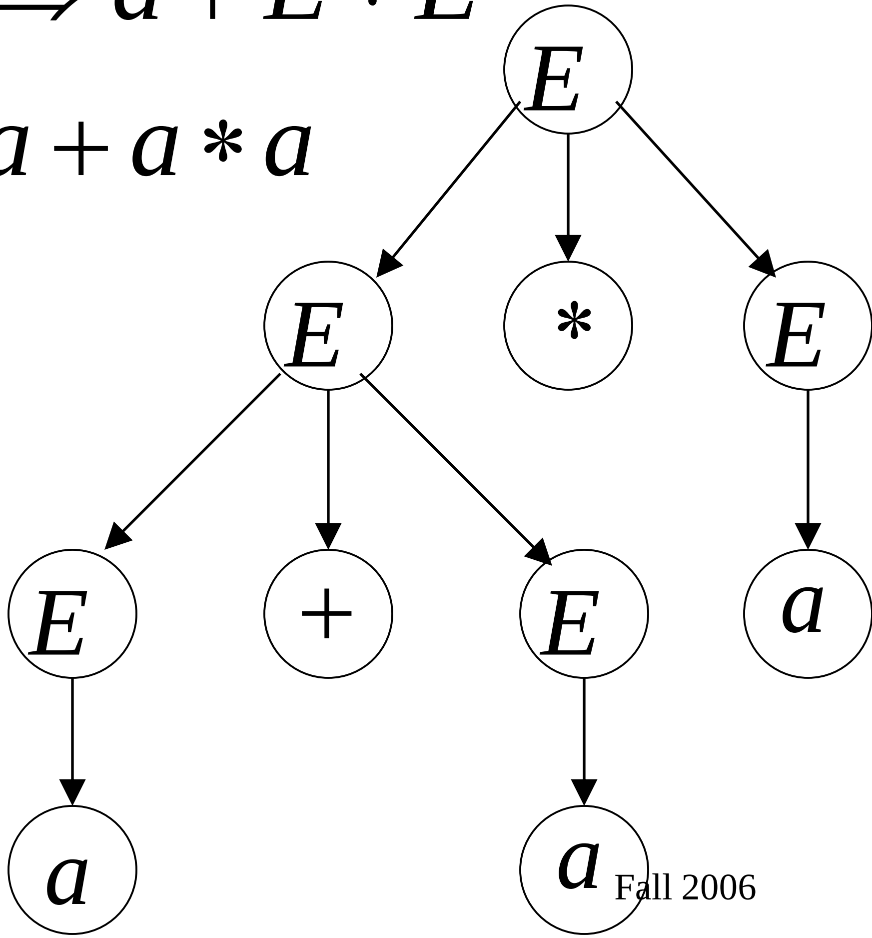
$$a + a * a$$

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

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

Another
leftmost derivation
for

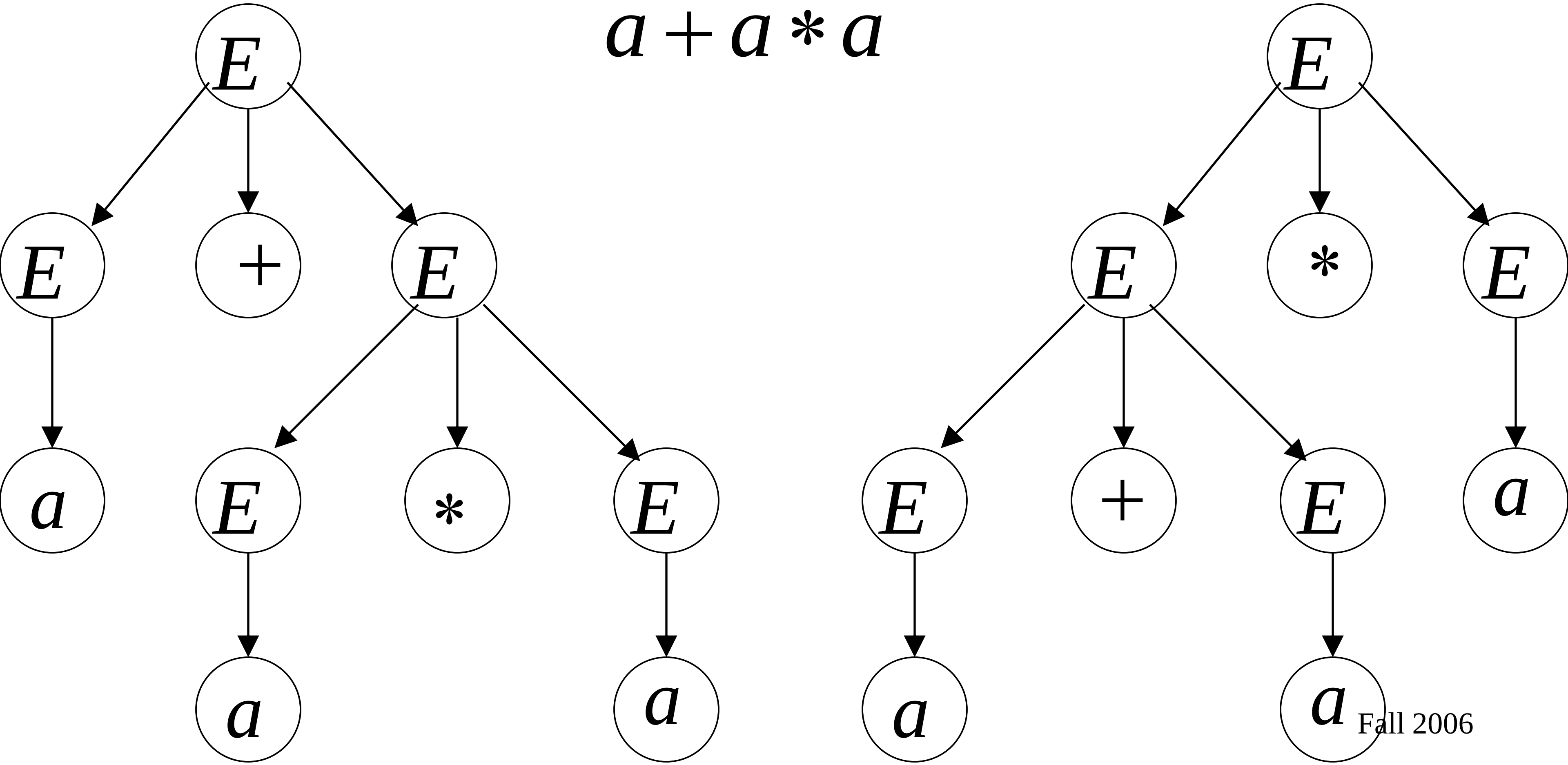
$$a + a * a$$



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

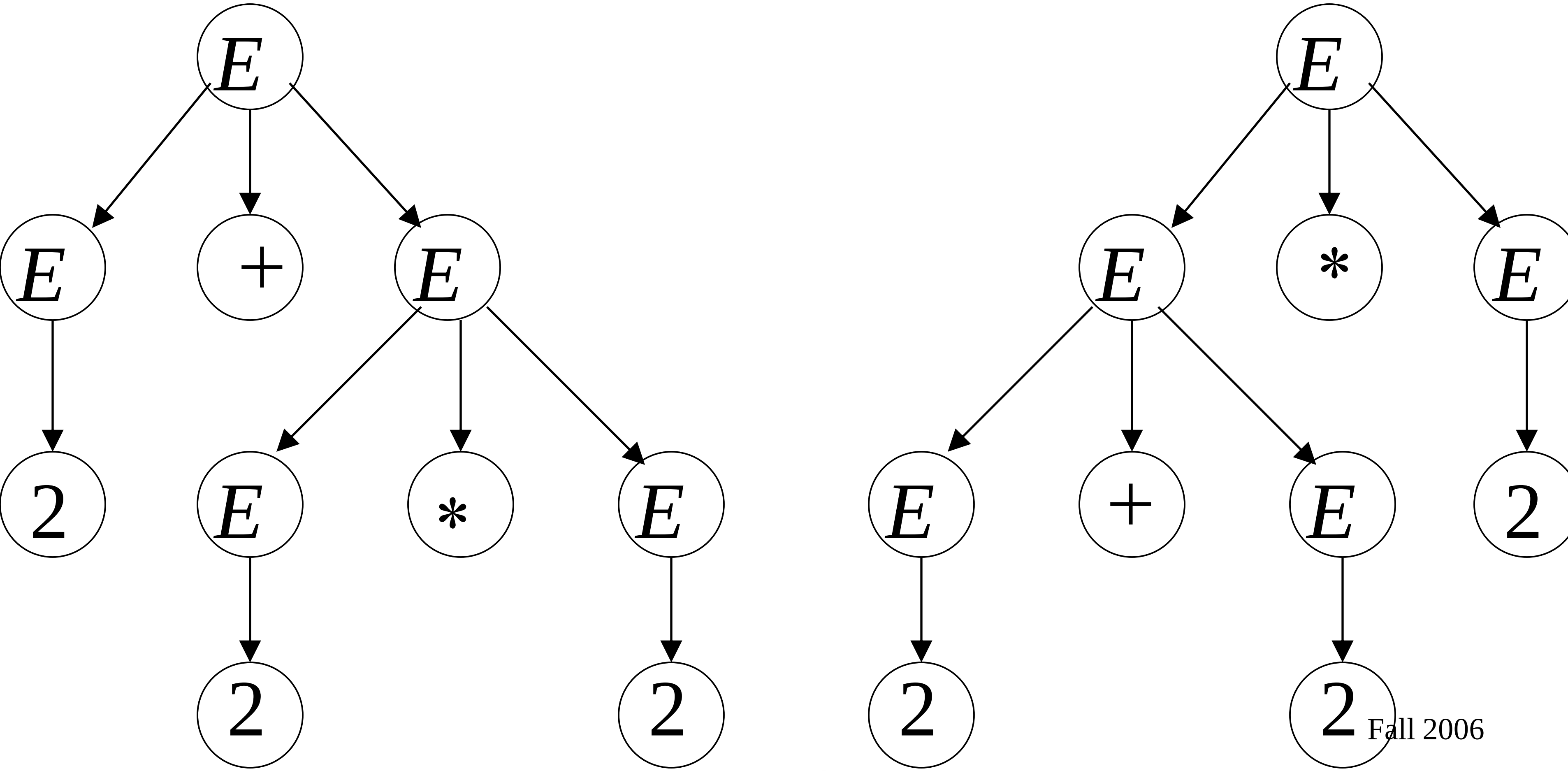
Two derivation trees
for

$$a + a * a$$



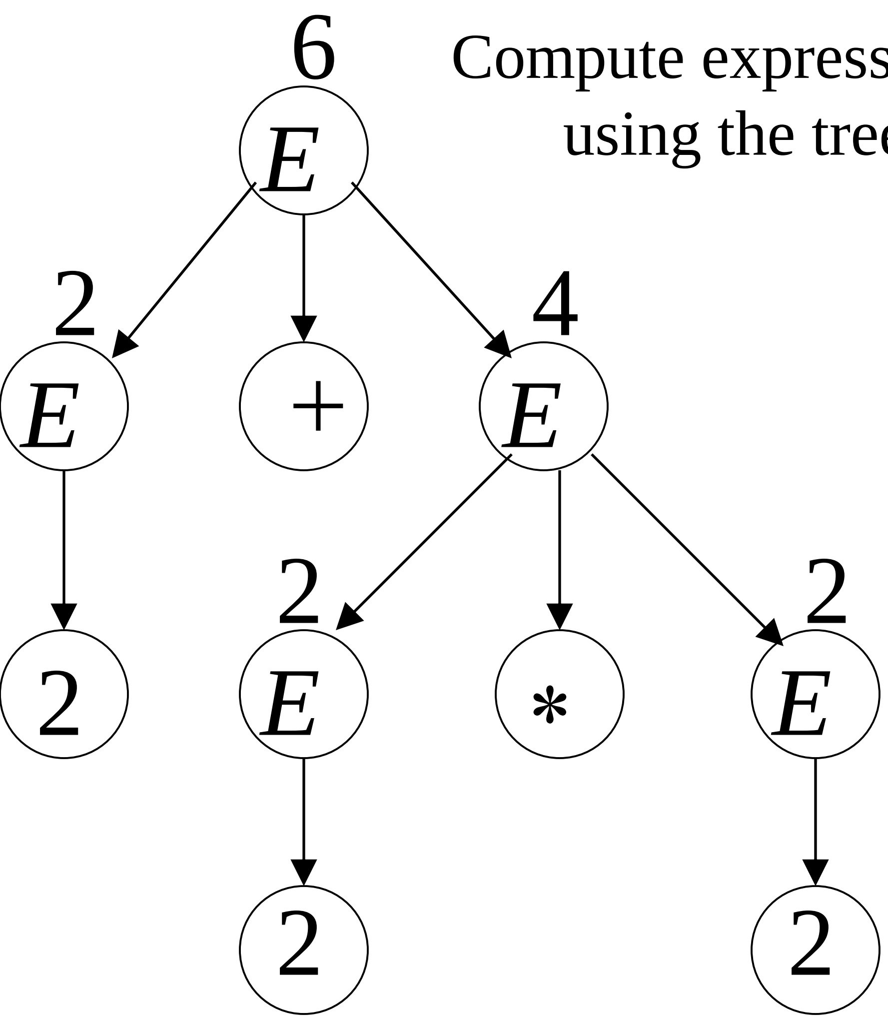
take $a = 2$

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



Good Tree

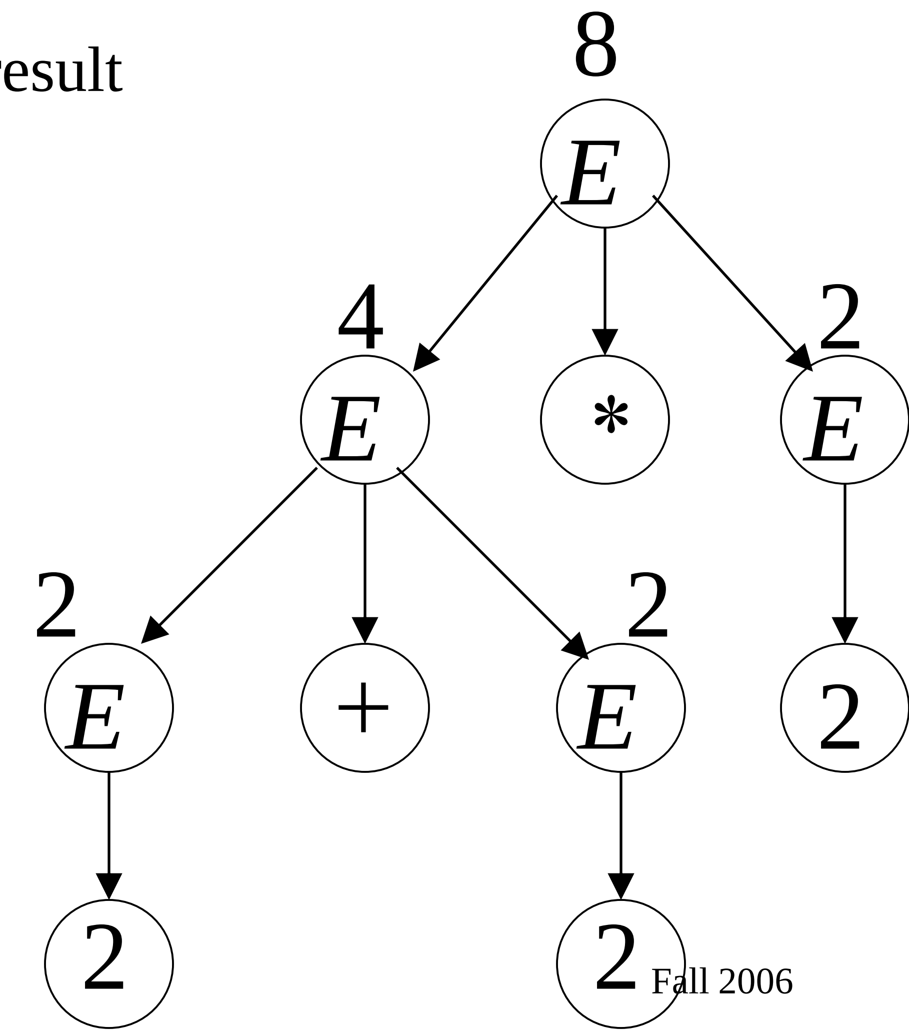
$$2+2*2=6$$



Compute expression result
using the tree

Bad Tree

$$2+2*2=8$$



Two different derivation trees
may cause problems in applications which
use the derivation trees:

- Evaluating expressions

- In general, in compilers
  for programming languages

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

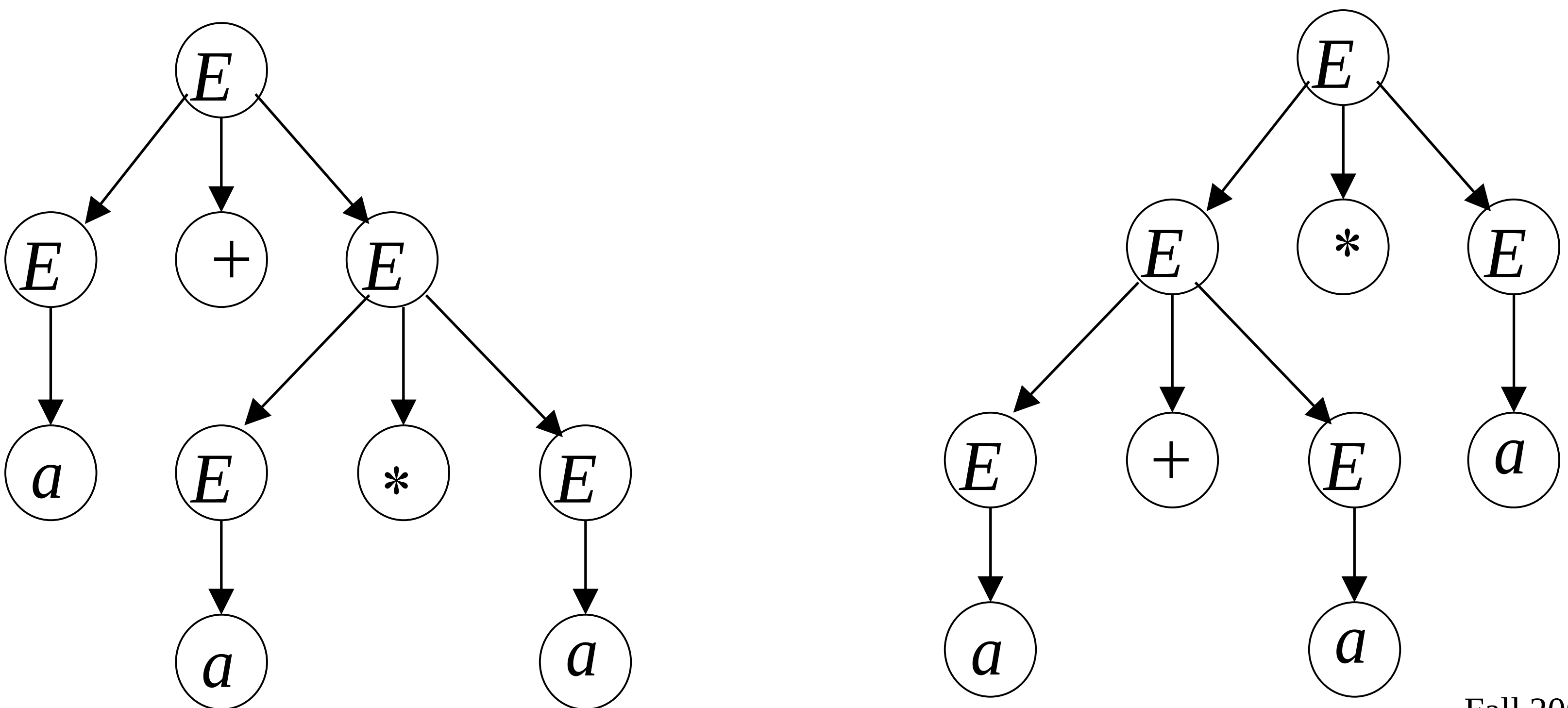
(Two different derivation trees give two different leftmost derivations and vice-versa)

Example:

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

this grammar is ambiguous since

string $a + a * a$ has two derivation trees



Fall 2006

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

this grammar is ambiguous also because

string $a + a * a$ has two leftmost derivations

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

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

Another ambiguous grammar:

IF_STMT $\rightarrow$ if EXPR then STMT

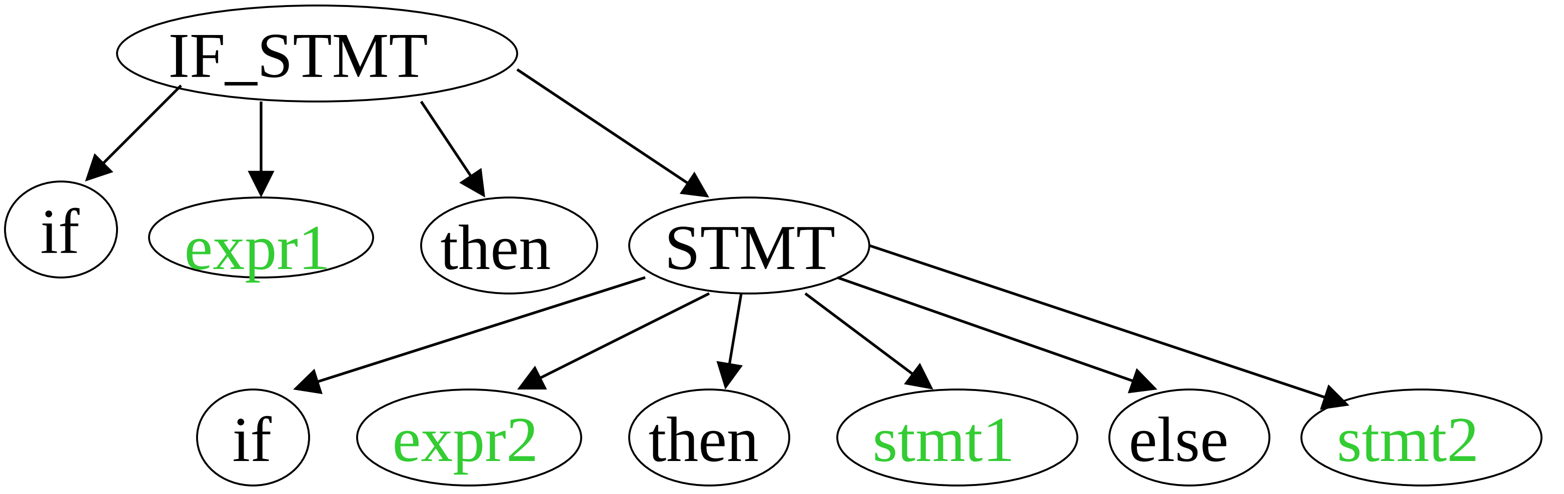
$\quad\quad\quad\quad\;|$ if EXPR then STMT else STMT

Variables

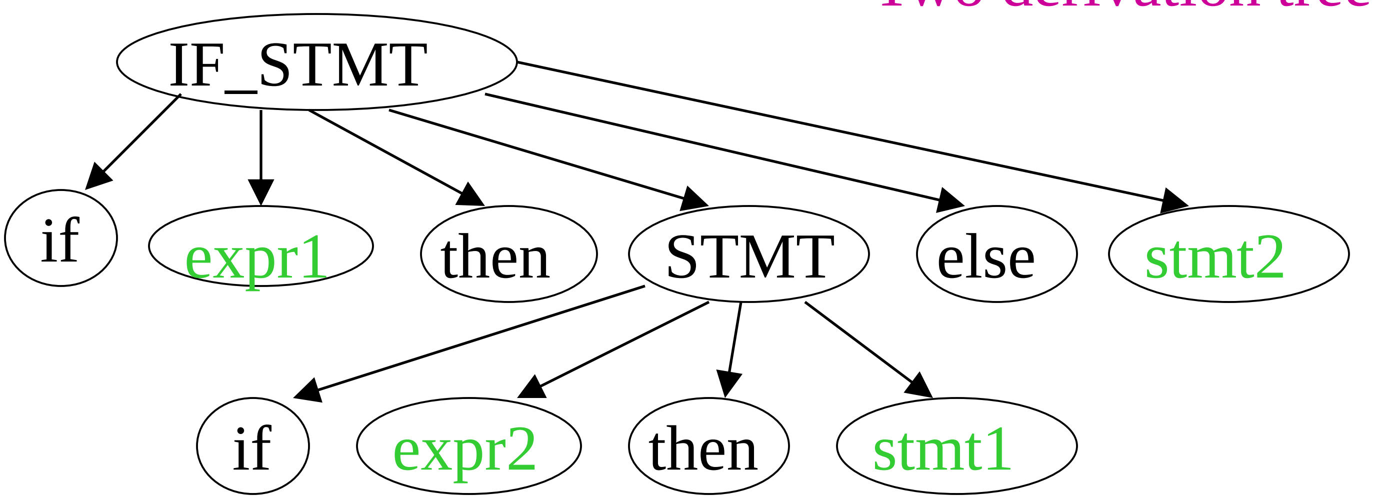
Terminals

Very common piece of grammar
in programming languages

If expr1 then if expr2 then stmt1 else stmt2



Two derivation trees



In general, ambiguity is bad
and we want to remove it

Sometimes it is possible to find
a non-ambiguous grammar for a language

But, in general we cannot do so

A successful example:

Equivalent

Ambiguous
Grammar

Non-Ambiguous
Grammar

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

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

generates the same
language

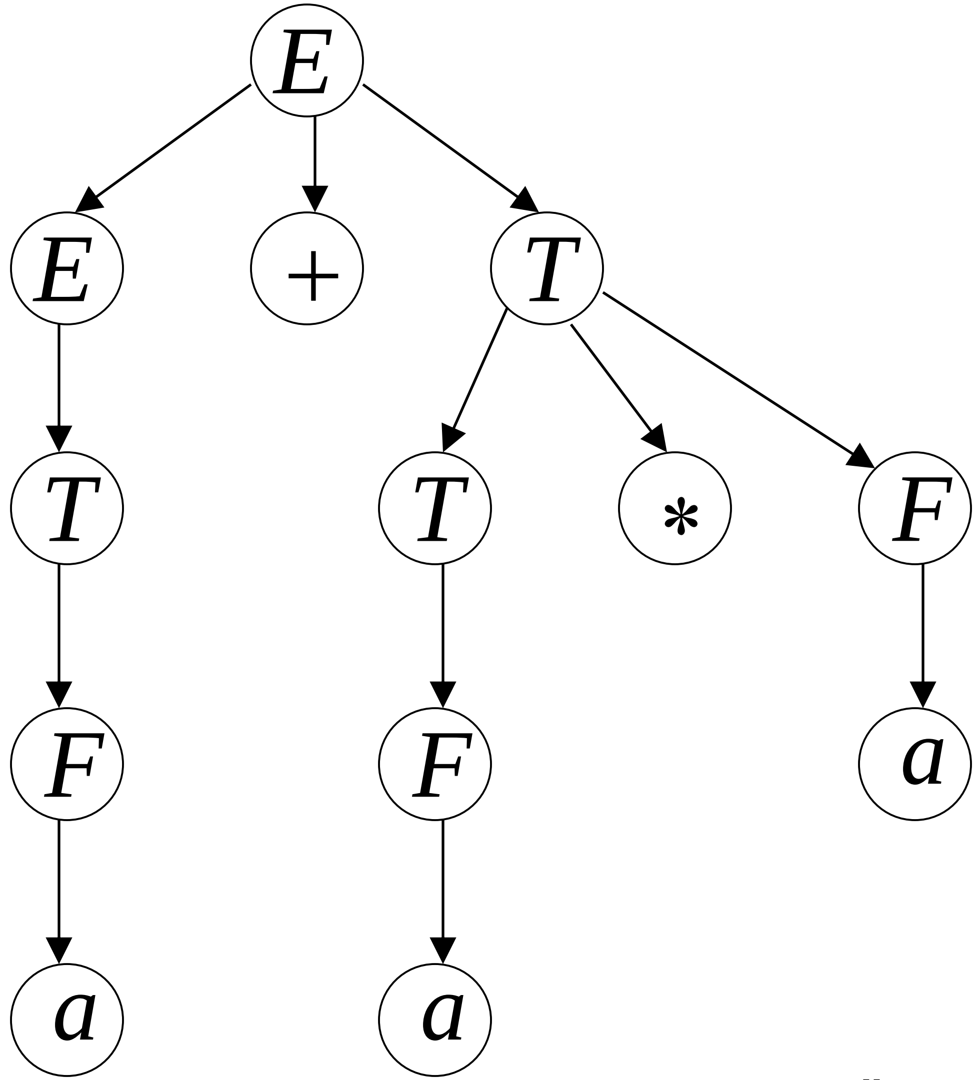
$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F$$

$$\Rightarrow a+F*F \Rightarrow a+a*F \Rightarrow a+a*a$$

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow (E) \mid a$$



Unique
derivation tree
for

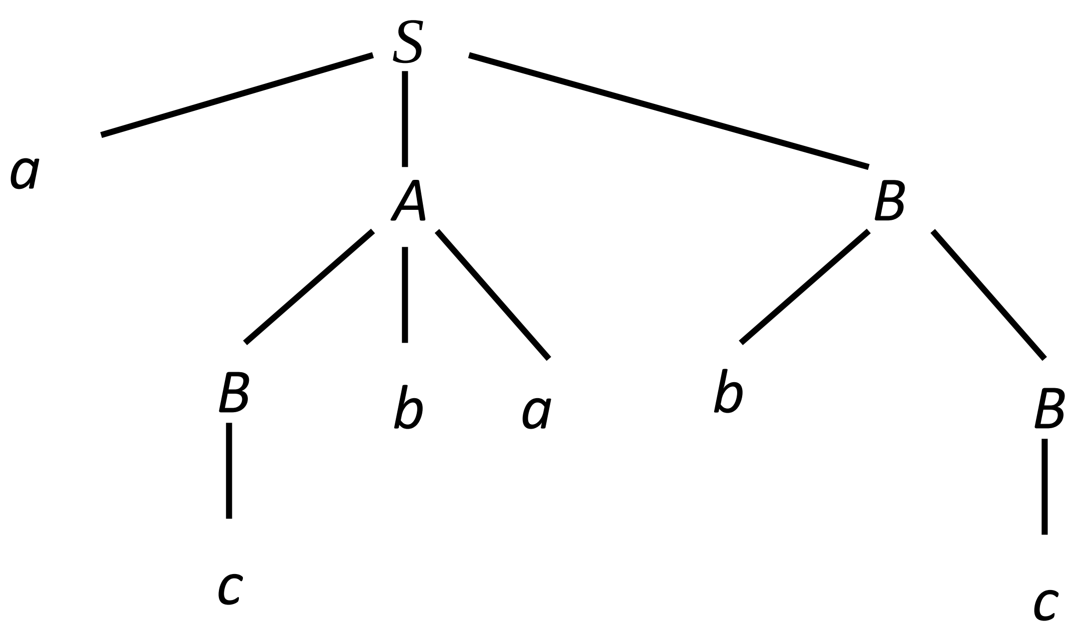
$$a+a*a$$

Example: Derivation Tree

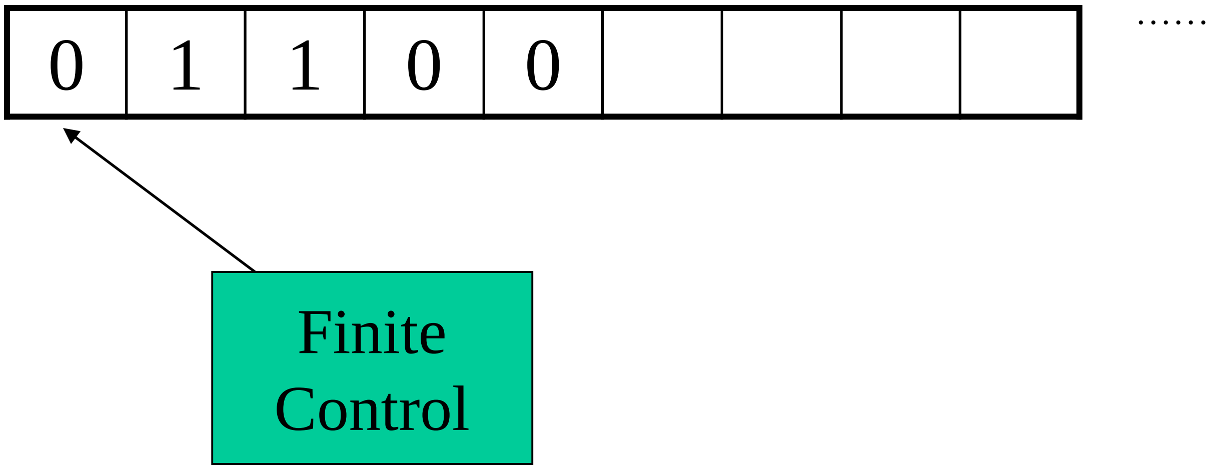
- Let G be a context-free grammar with the productions $P = \{S \rightarrow aAB,\ A \rightarrow Bba,\ B \rightarrow bB,\ B \rightarrow c\}$. The word w = acbabc can be derived from S as follows:

  $$S \Rightarrow aAB \rightarrow a(Bba)B \Rightarrow acbaB \Rightarrow acba(bB) \Rightarrow acbabc$$

  Thus, the derivation tree is given as follows:



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.

Formal Definition of a DFA

- A DFA is a five-tuple:

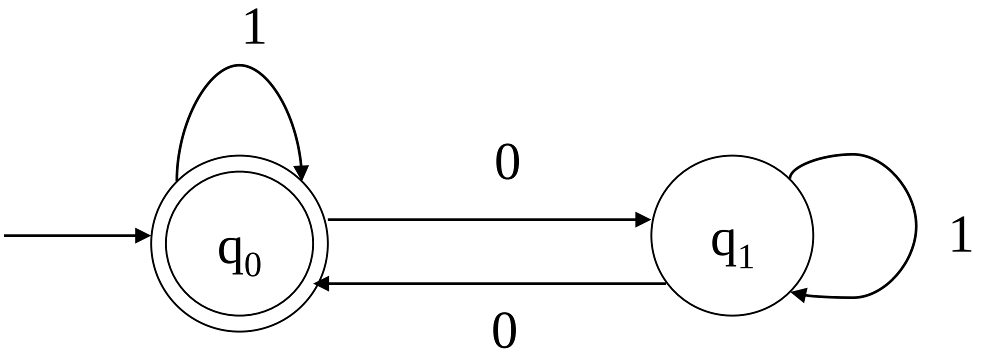
  $M = (Q, \Sigma, \delta, q_0, F)$

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 $Q \times \Sigma$ to Q

$\delta: (Q \times \Sigma) \rightarrow Q$	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:



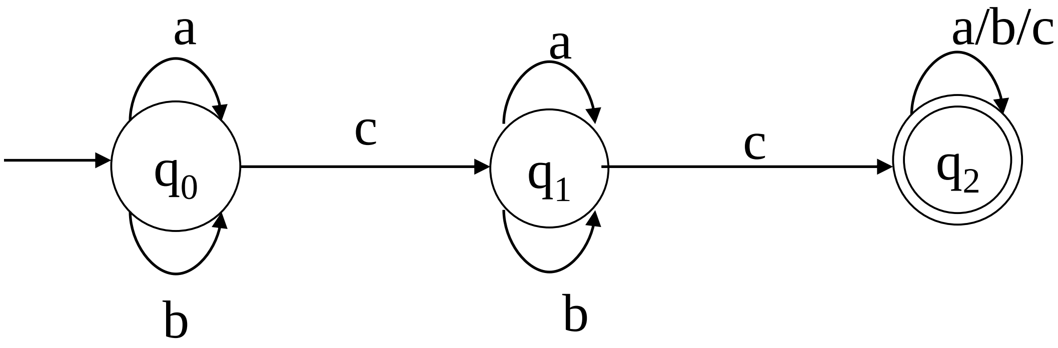
$Q = \{q_0, q_1\}$
$\Sigma = \{0, 1\}$
Start state is q_0
$F = \{q_0\}$

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.

Extension of δ to Strings

$\delta^{\wedge} : (Q \times \Sigma^*) \rightarrow Q$

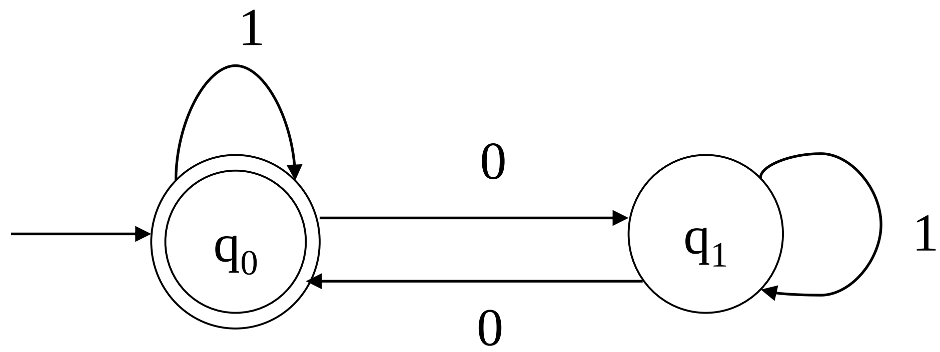
$\delta^{\wedge}(q,w)$ – The state entered after reading string *w* having started in state *q*.

Formally:

1) $\delta^{\wedge}(q, \varepsilon) = q$, and
2) For all w in Σ^* and a in Σ

$$\delta^{\wedge}(q,wa) = \delta(\delta^{\wedge}(q,w), a)$$

- Recall Example #1:



- What is $\delta^\wedge(q_0, 011)$? Informally, it is the state entered by M after processing 011 having started in state q_0.
- Formally:

$\delta^\wedge(q_0, 011)$	$= \delta(\delta^\wedge(q_0, 01), 1)$	by rule #2
	$= \delta(\delta(\delta^\wedge(q_0, 0), 1), 1)$	by rule #2
	$= \delta(\delta(\delta(\delta^\wedge(q_0, \lambda), 0), 1), 1)$	by rule #2
	$= \delta(\delta(\delta(q_0, 0), 1), 1)$	by rule #1
	$= \delta(\delta(q_1, 1), 1)$	by definition of δ
	$= \delta(q_1, 1)$	by definition of δ
	$= q_1$	by definition of δ

- Is 011 accepted? No, since $\delta^\wedge(q_0, 011) = q_1$ is not a final state.

- Notes:
  - A DFA $M = (Q, \Sigma, \delta, q_0, F)$ partitions the set Σ^* into two sets: L(M) and Σ^* - L(M).
  - If L = L(M) then L is a subset of L(M) and L(M) is a subset of L (def. of set equality).
  - Similarly, if $L(M_1) = L(M_2)$ then $L(M_1)$ is a subset of $L(M_2)$ and $L(M_2)$ is a subset of $L(M_1)$.
  - Some languages are regular, others are not. For example, if

    Regular: L_1 = {x | x is a string of 0's and 1's containing an even number of 1's} and

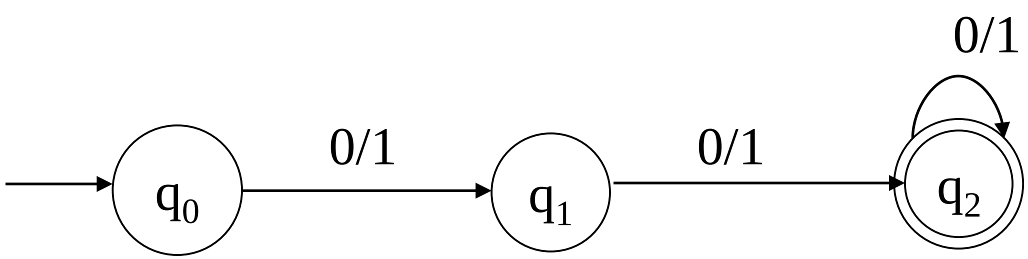
    Not-regular: $L_2 = \{x \mid x = 0^n1^n \text{ for some } n >= 0\}$

- *Can you write a program to "simulate" a given DFA, or any arbitrary input DFA?*

- Question we will address later:
  - How do we determine whether or not a given language is regular?

- Give a DFA M such that:

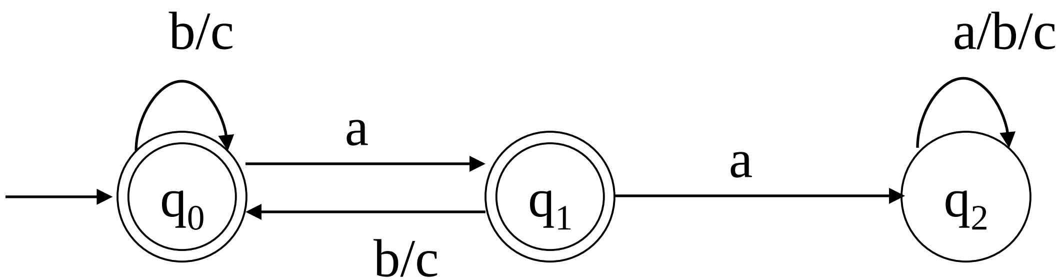
$$L(M) = \{x \mid x \text{ is a string of 0's and 1's and } |x| >= 2\}$$



Prove this by induction

- Give a DFA M such that:

L(M) = {x | x is a string of (zero or more) a's, b's and c's such that x does *not* contain the substring *aa*}



Logic:

In Start state (q0): b's and c's: ignore – stay in same state

q0 is also "accept" state

First 'a' appears: get ready (q1) to reject

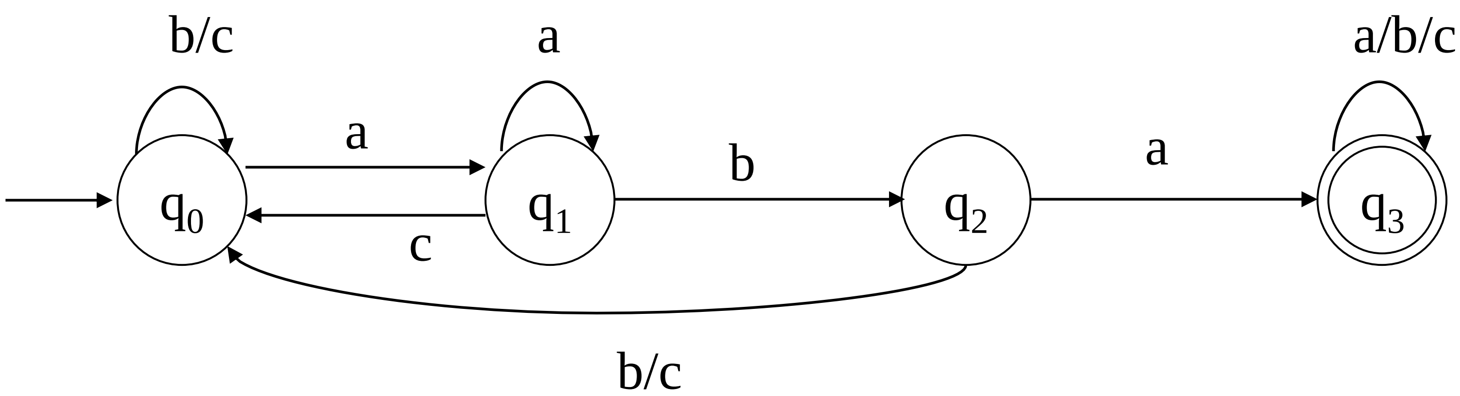
But followed by a 'b' or 'c': go back to start state q0

When second 'a' appears after the "ready" state: go to reject state q2

Ignore everything after getting to the "reject" state q2

- Give a DFA M such that:

L(M) = {x | x is a string of a's, b's and c's such that x contains the substring *aba*}



Logic: acceptance is straight forward, progressing on each expected symbol

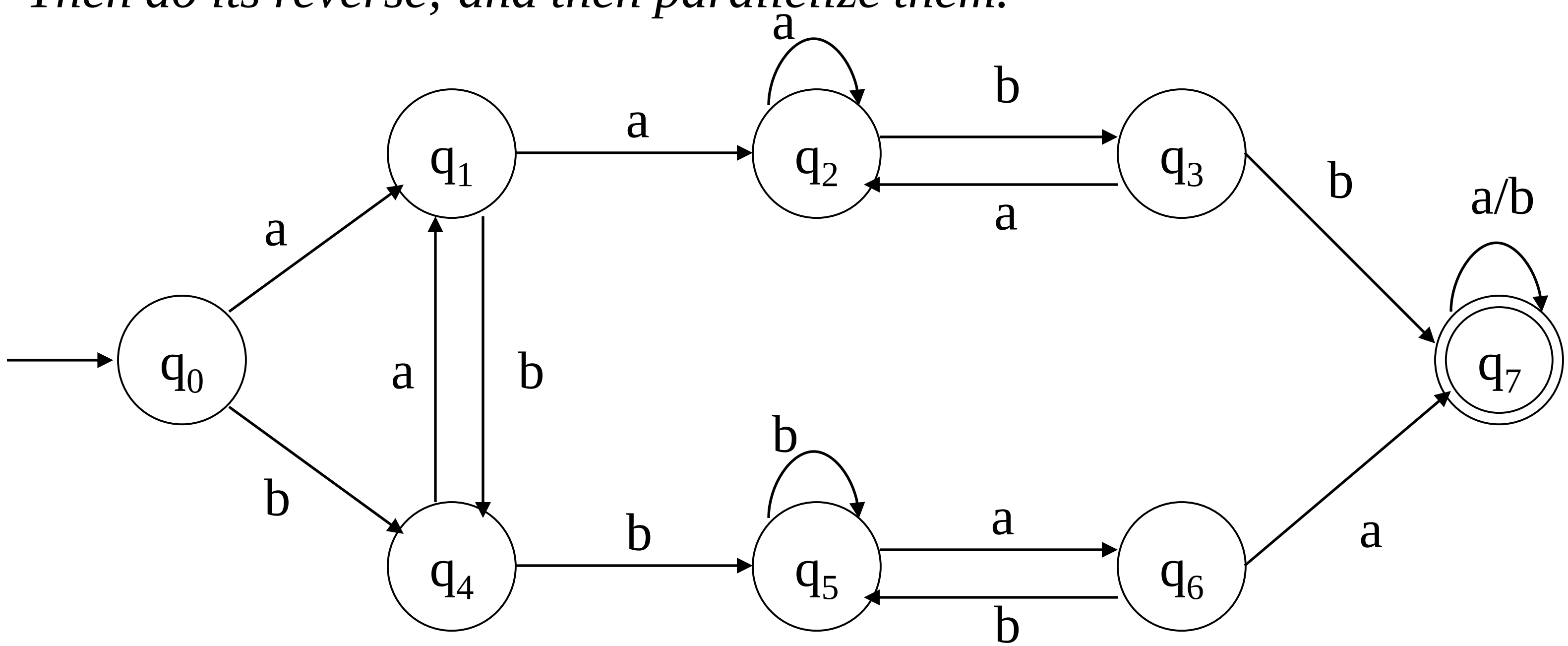
However, rejection needs special care, in each state (for DFA, we will see this becomes easier in NFA, non-deterministic machine)

- Give a DFA M such that:

L(M) = {x | x is a string of a's and b's such that x contains both *aa* and *bb*}

First do, for a language where 'aa' comes before 'bb'

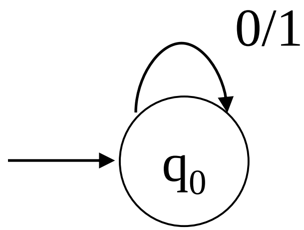
Then do its reverse; and then parallelize them.



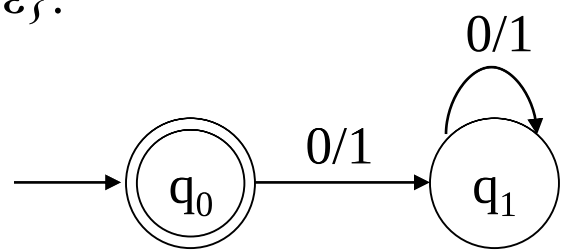
Remember, you may have multiple "final" states, but only one "start" state

- Let $\Sigma = \{0, 1\}$. Give DFAs for $\{\}$, $\{\varepsilon\}$, Σ^*, and Σ^+.

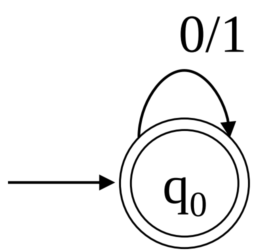
For $\{\}$:



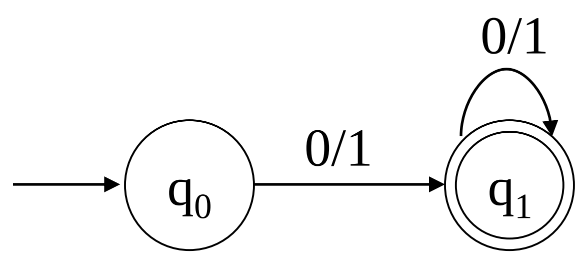
For $\{\varepsilon\}$:



For Σ^*:



For Σ^+:



Nondeterministic Finite State Automata (NFA)

- An NFA is a five-tuple:

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

Q	A <u>finite</u> set of states
Σ	A <u>finite</u> 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

  $\delta: (Q \times \Sigma) \rightarrow$ **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)

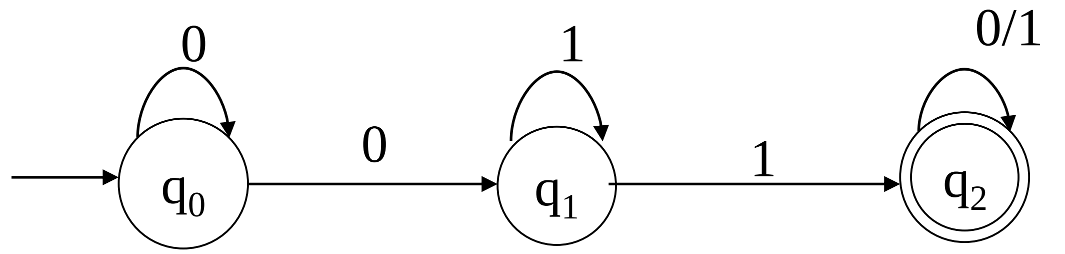
- Example #1: one or more 0's followed by one or more 1's

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

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

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	$\{q_2\}$	$\{q_2\}$

Definitions for NFAs

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA and let w be in Σ^*. Then w is *accepted* by M iff $\delta(\{q_0\}, w)$ contains at least one state in F.

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Then the *language accepted* by M is the set:

  $L(M) = \{w \mid w \text{ is in } \Sigma^* \text{ and } \delta(\{q_0\},w) \text{ contains at least one state in } F\}$

- Another equivalent definition:

  $L(M) = \{w \mid w \text{ is in } \Sigma^* \text{ and } w \text{ is accepted by } M\}$

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.

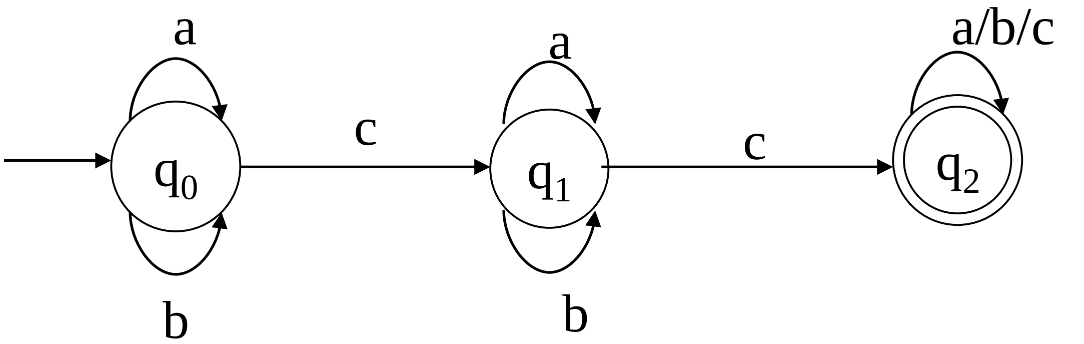
- Therefore, if L is a regular language then there exists an NFA M such that L = L(M).

- 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
$F = \{q_2\}$

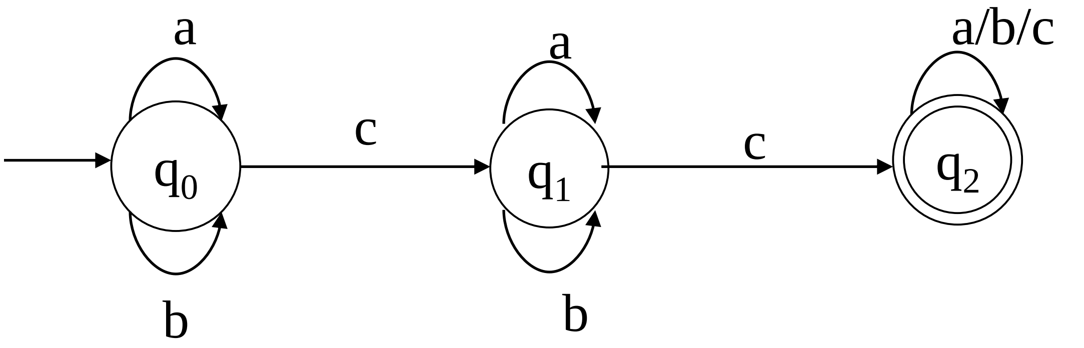


δ:

	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	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$

- **Lemma 1:** Let M be an DFA. Then there exists a NFA M' such that L(M) = L(M').

- **Proof:** Every DFA is an NFA. Hence, if we let M' = M, then it follows that L(M') = L(M).

  The above is just a formal statement of the observation from the previous slide.

- **Lemma 2:** Let M be an NFA. Then there exists a DFA M' such that L(M) = L(M').

- **Proof:** (sketch)

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

Define a DFA $M' = (Q', \Sigma, \delta', q'_0, F')$ as:

$Q' = 2^Q$
$= \{Q_0, Q_1, \ldots,\}$

Each state in M' corresponds to a subset of states from M

where $Q_u = [q_{i0}, q_{i1}, \ldots q_{ij}]$

$F' = \{Q_u \mid Q_u$ contains at least one state in $F\}$

$q'_0 = [q_0]$

$\delta'(Q_u, a) = Q_v$ iff $\delta(Q_u, a) = Q_v$

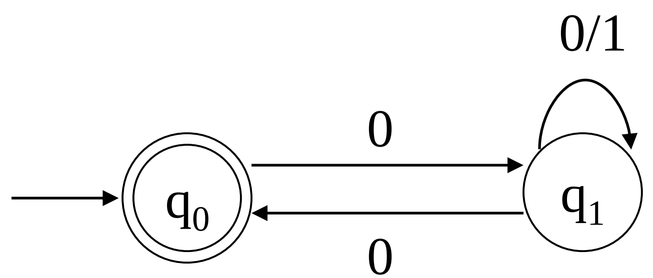
- Example: empty string or start and end with 0

$Q = \{q_0, q_1\}$

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

Start state is q_0

$F = \{q_0\}$



δ:

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

NFAs with ε Moves

- An NFA-ε is a five-tuple:

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

Q	A <u>finite</u> set of states
Σ	A <u>finite</u> 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 U $\{\varepsilon\}$ to 2^Q

  $\delta: (Q \times (\mathbf{\Sigma \cup \{\varepsilon\}})) \rightarrow 2^Q$

  $\delta(q,s)$ -The set of all states p such that there is a transition labeled a from q to p, where a is in Σ U $\{\varepsilon\}$

- Sometimes referred to as an NFA-ε other times, simply as an NFA.

ε-closure

- Define ε-closure(q) to denote the set of all states reachable from q by zero or more ε transitions.

- Examples: (for the previous NFA)

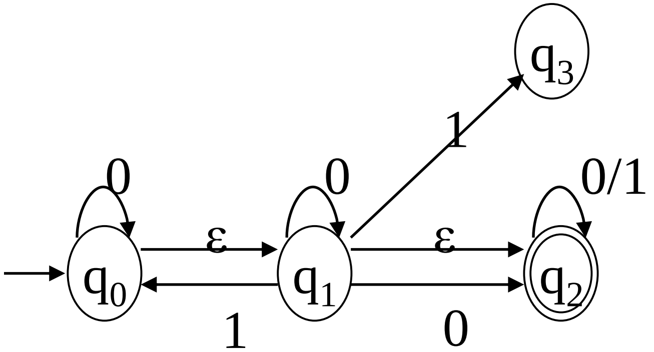
ε-closure(q_0) = {q_0, q_1, q_2} ε-closure(q_2) = {q_2}
ε-closure(q_1) = {q_1, q_2} ε-closure(q_3) = {q_3}

- ε-closure(q) can be extended to sets of states by defining:

$$\varepsilon\text{-closure}(P) = \bigcup_{q \in P} \varepsilon\text{-closure}(q)$$

- Examples:

ε-closure({q_1, q_2}) = {q_1, q_2}
ε-closure({q_0, q_3}) = {q_0, q_1, q_2, q_3}



Chomsky & Greibach Normal Forms

Hector Miguel Chavez
Western Michigan University

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

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

Example: Removing **non-generating** symbols

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

Initial CFL grammar

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

Identify generating symbols

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

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

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.

Eliminate ε Productions

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

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

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

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

Nullable: {A, B}

Eliminate ε Productions

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

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

Eliminate ε Productions

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

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

Eliminate ε Productions

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

$$\begin{aligned} S &\rightarrow ASA \mid aB \\ A &\rightarrow B \mid S \\ B &\rightarrow b \mid \varepsilon \end{aligned} \quad \Longrightarrow \quad \begin{aligned} S &\rightarrow ASA \mid aB \mid AS \mid SA \mid S \mid a \\ A &\rightarrow B \mid S \\ B &\rightarrow b \end{aligned}$$

Eliminate unit productions

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

Identify **unit pairs**

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

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

Preliminary Simplifications

Example:

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

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

Preliminary Simplifications

Example:

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

$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

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

$\mathbf{T \rightarrow T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1}$

$F \rightarrow (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

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

$E \rightarrow EPT \mid TMF \mid LER \mid a \mid b \mid lA \mid lB \mid lZ \mid lO$
$T \rightarrow TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
$F \rightarrow LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
$I \rightarrow a \mid b \mid IA \mid IB \mid IZ \mid IO$
$A \rightarrow a \quad B \rightarrow b \quad Z \rightarrow 0 \quad O \rightarrow 1$

Final Simplification

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

$E \rightarrow E\mathbf{PT} \mid T\mathbf{MF} \mid L\mathbf{ER} \mid a \mid b \mid lA \mid lB \mid lZ \mid lO$

$T \rightarrow T\mathbf{MF} \mid L\mathbf{ER} \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$F \rightarrow L\mathbf{ER} \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$I \rightarrow a \mid b \mid IA \mid IB \mid IZ \mid IO$

$A \rightarrow a \quad B \rightarrow b \quad Z \rightarrow 0 \quad O \rightarrow 1$

$P \rightarrow + \quad M \rightarrow * \quad L \rightarrow (\quad R \rightarrow)$

$C_1 \rightarrow PT$

$C_2 \rightarrow MF$

$C_3 \rightarrow ER$

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:

$S \rightarrow XA \mid BB$	$S = A_1$	$A_1 \rightarrow A_2A_3 \mid A_4A_4$
$B \rightarrow b \mid SB$	$X = A_2$	$A_4 \rightarrow b \mid A_1A_4$
$X \rightarrow b$	$A = A_3$	$A_2 \rightarrow b$
$A \rightarrow a$	$B = A_4$	$A_3 \rightarrow a$
CNF	New Labels	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 $\boxed{A_i \longrightarrow A_jX_k \quad j > i}$

X_k is a string of zero or more variables

✕ $A_4 \rightarrow A_1A_4$

Greibach Normal Form

Example:

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

$A_4 \rightarrow A_1 A_4$

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

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

$A_1 \rightarrow A_2 A_3 \mid A_4 A_4$

$A_4 \rightarrow b \mid A_1 A_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Greibach Normal Form

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

$\times\ A_4 \rightarrow A_4A_4A_4$

Example:

Second Step

Eliminate Left Recursions

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

$Z \to A_4A_4 \mid A_4A_4Z$

$A_1 \to A_2A_3 \mid A_4A_4$

$A_4 \to bA_3A_4 \mid A_4A_4A_4 \mid b$

$A_2 \to b$

$A_3 \to a$

Greibach Normal Form

Example:

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

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

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

- Notation to specify a language
  - Declarative
  - Sort of like a programming language.
    - Fundamental in some languages like perl and applications like grep or lex
  - Capable of describing the same thing as a NFA
    - The two are actually equivalent, so RE = NFA = DFA
  - We can define an algebra for regular expressions

Algebra for Languages

- We use following operators in regular expressions:
  - Union
  - Concatenation
  - Kleene Star

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(1Ø) = Ø   ; concatenating the empty set to any set yields the empty set.
- Rε = R
- R+Ø = R

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

Regular Expressions

- Regular expressions
- describe regular languages

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

- Example:

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

- describes the language

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.

Turning a DFA into a RE

- Theorem: If L=L(A) for some DFA A, then there is a regular expression R such that L=L(R).
- Proof
  - Construct GNFA, Generalized NFA
    - We'll skip this in class, but see the textbook for details
  - State Elimination
    - We'll see how to do this next, easier than inductive construction, there is no exponential number of expressions

DFA to RE: State Elimination

- Eliminates states of the automaton and replaces the edges with regular expressions that includes the behavior of the eliminated states.
- Eventually we get down to the situation with just a start and final node, and this is easy to express as a RE

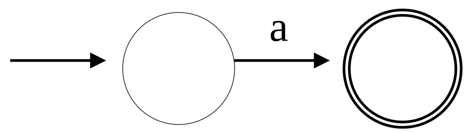
Converting a RE to an Automata

- 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

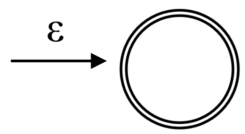
RE to ε-NFA

- Basis:

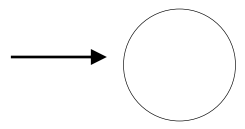
R=a



R=ε

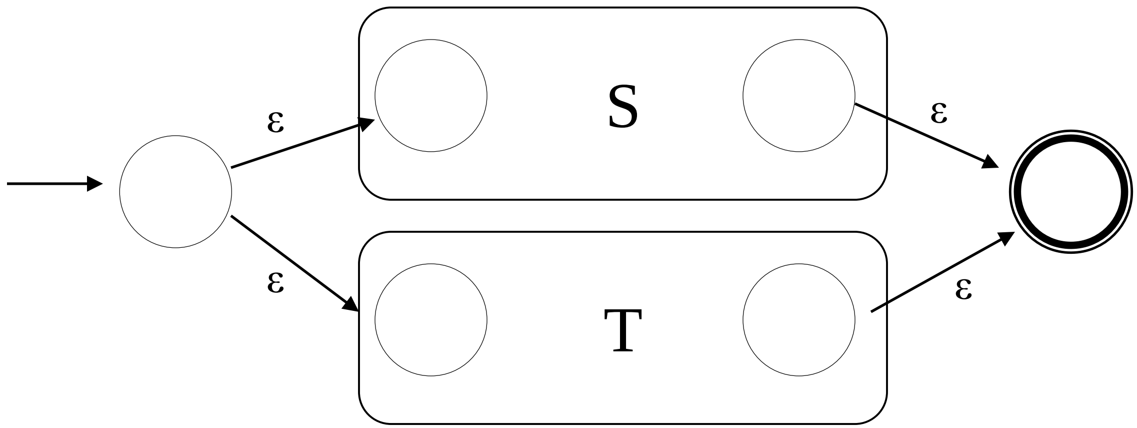


R=Ø

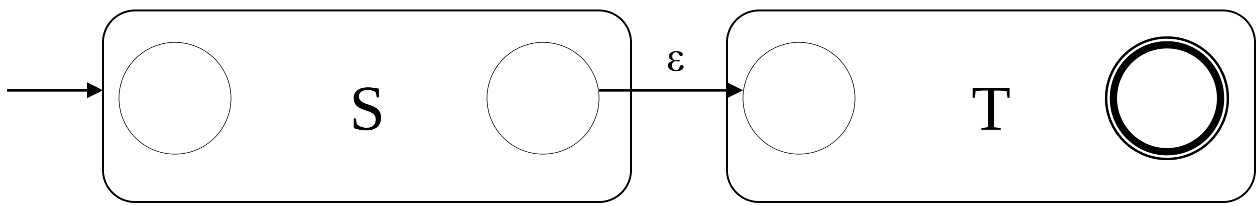


Next slide: More complex RE’s

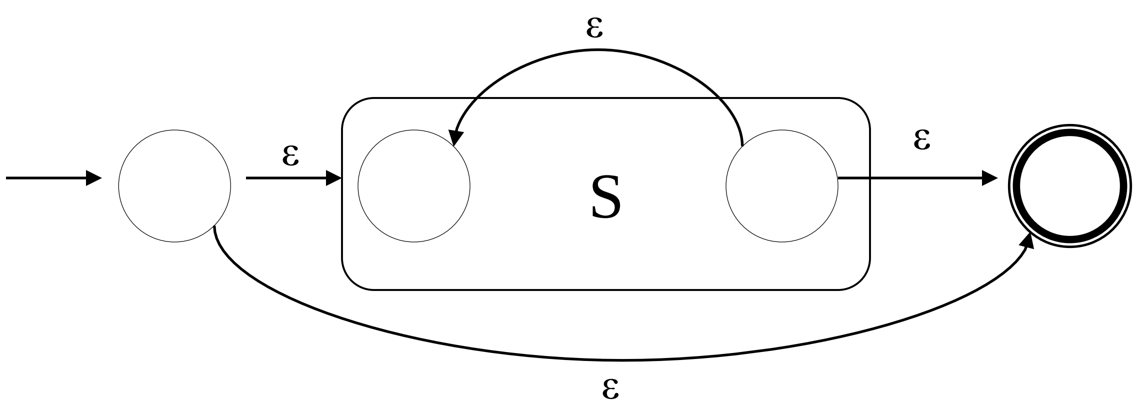
R=S+T



R=ST



$R=S^*$



Let M be a DFA that recognizes L

Let P be the number of states in M

Assume $w \in L$ is such that $|w| \geq P$

We show $w = xyz$

1. $|y| > 0$
2. $|xy| \leq P$
3. $xy^iz \in L$ for any $i \geq 0$

x

…

q_0 q_i q_j $q_{|w|}$

There must be $j > i$ such that $q_i = q_j$

USING THE **PUMPING LEMMA**

Use the pumping lemma to prove that
B = $\{0^n1^n \mid n \geq 0\}$ is not regular

Hint: Assume B is regular

Let B = L(M), for DFA M,
and let P be larger than the
number of states in M

Try pumping s = 0^P1^P

Use the pumping lemma to prove that
C = { w | w has an equal number of 0s and 1s}
is not regular

Hint: Try pumping s = 0^P1^P

If C is regular, s can be split into s = xyz, where for any i ≥ 0, xy^iz is also in C
and $|xy| \leq P$

Formal Definition of a PDA

- A <u>pushdown automaton (PDA)</u> is a seven-tuple:

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

Q	A <u>finite</u> set of states
Σ	A <u>finite</u> input alphabet
Γ	A <u>finite</u> 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.

PDAs to produce derivation strings

- Given some BNF (context free grammar). Produce the leftmost derivation of a string using a PDA:

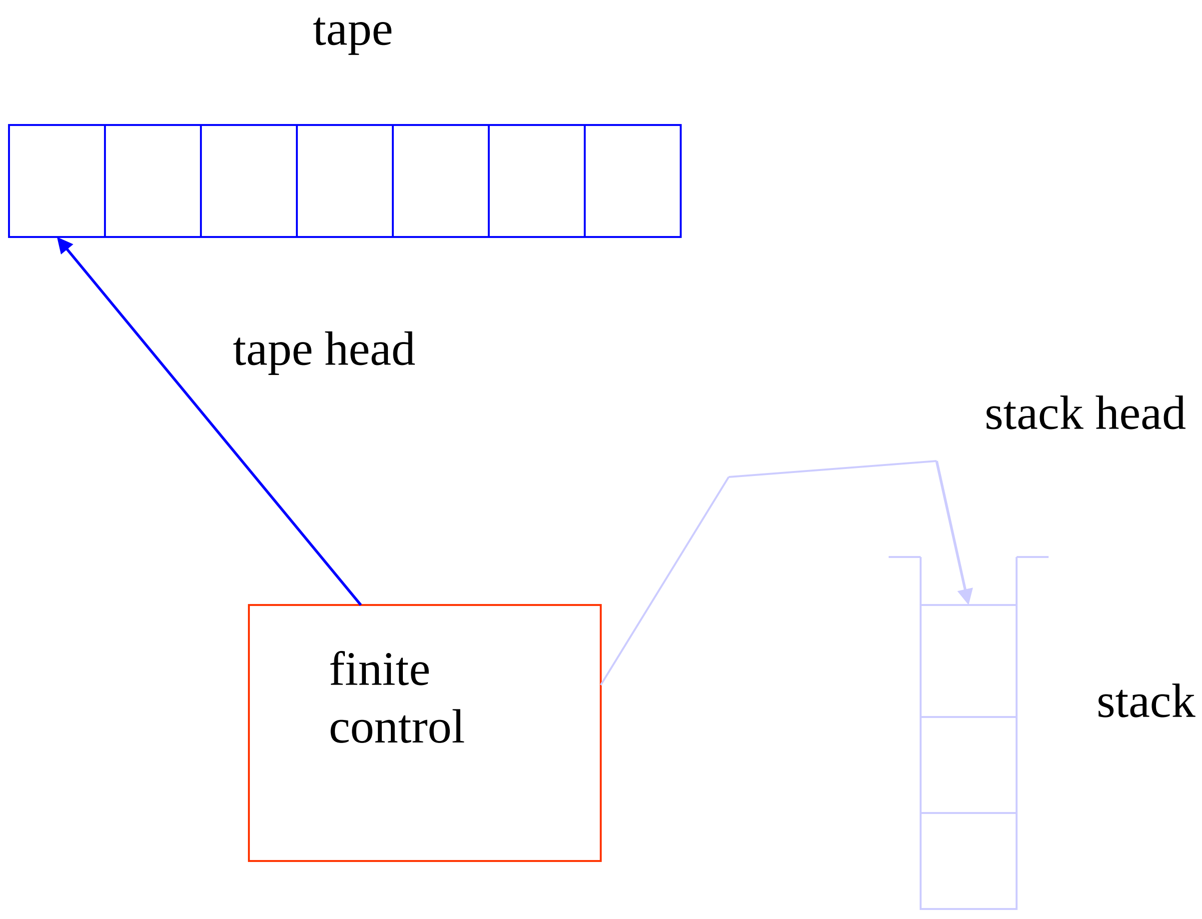
- 1. If the top of the stack is a terminal symbol, compare it to the next input symbol; pop it off the stack if the same. It is an error if the symbols do not match.
- 2. If the top of the stack is a nonterminal symbol X, replace X on the stack with some string α, where α is the right hand side of some production $X \rightarrow \alpha$.

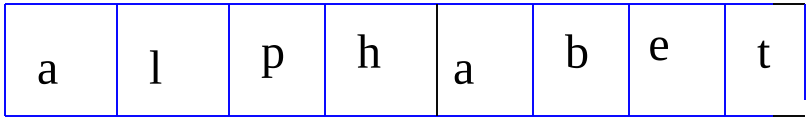
- This PDA now simulates the leftmost derivation for some context-free grammar.
- This construction actually develops a nondeterministic PDA that is equivalent to the corresponding BNF grammar. (i.e., step 2 may have multiple options.)

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).


tape
tape head
stack head
finite
control
stack



The tape is divided into finitely many cells.

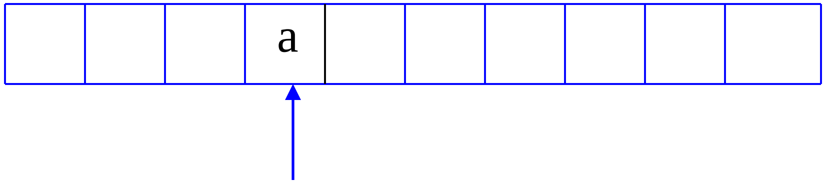
Each cell contains a symbol in an alphabet Σ.

The stack head always scans the top symbol of the stack. It performs two basic operations:

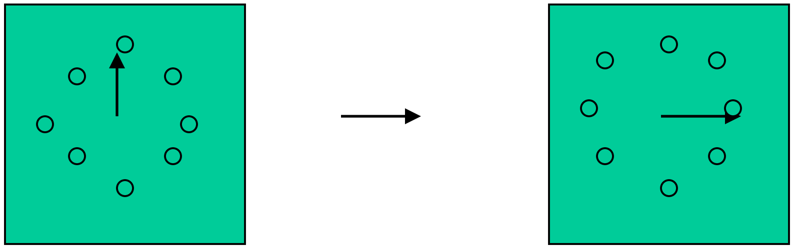
Push: add a new symbol at the top.

Pop: read and remove the top symbol.

Alphabet of stack symbols: Γ

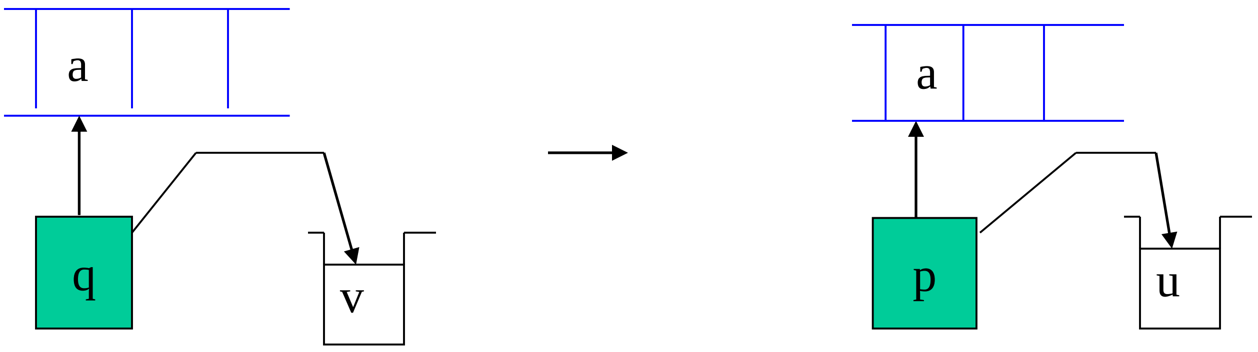


- The head scans at a cell on the tape and can *read* a symbol on the cell. In each move, the head can move to the right cell.

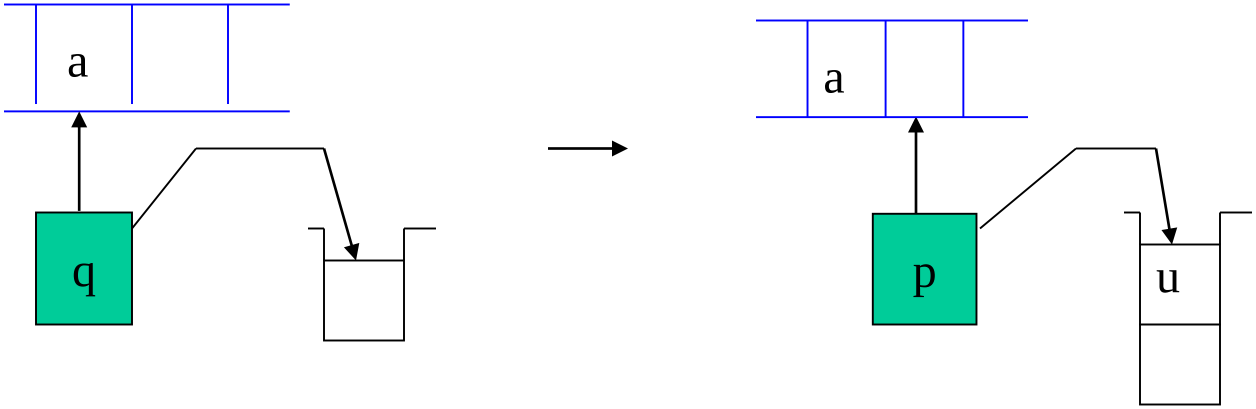


- The finite control has finitely many states which form a set Q. For each move, the state is changed according to the evaluation of a *transition function*

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow 2^{Q \times (\Gamma \cup \{\varepsilon\})}.$$



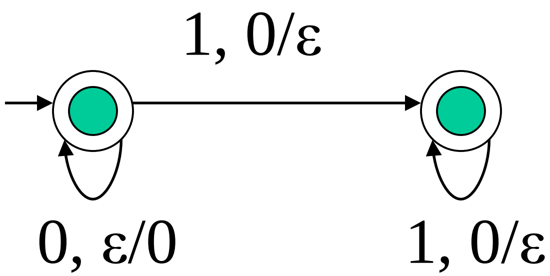
- $(p, u) \in \delta(q, \varepsilon, v)$ means that this a ε-move.



- $(p, u) \in \delta(q, a, \varepsilon)$ means that a push operation performs at stack.

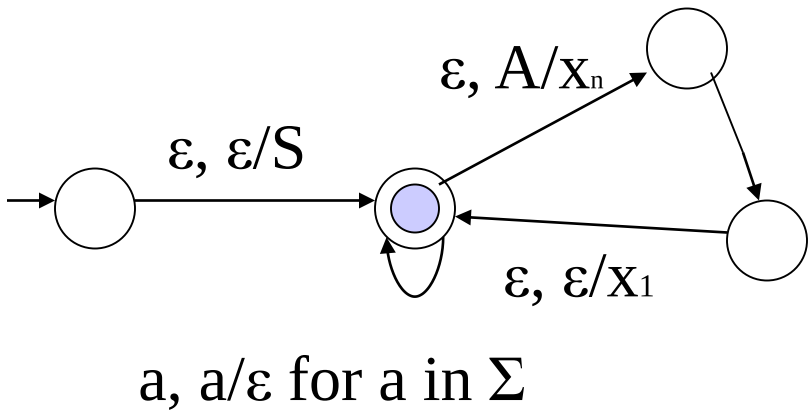
Example 1. Construct PDA to accept $L = \{0^n 1^n \mid n \geq 0\}$

Solution 1.



Theorem Every CFL can be accepted by a PDA.

Proof. Consider a CFL L = L(G) for a CFG G = (V, Σ, R, S).

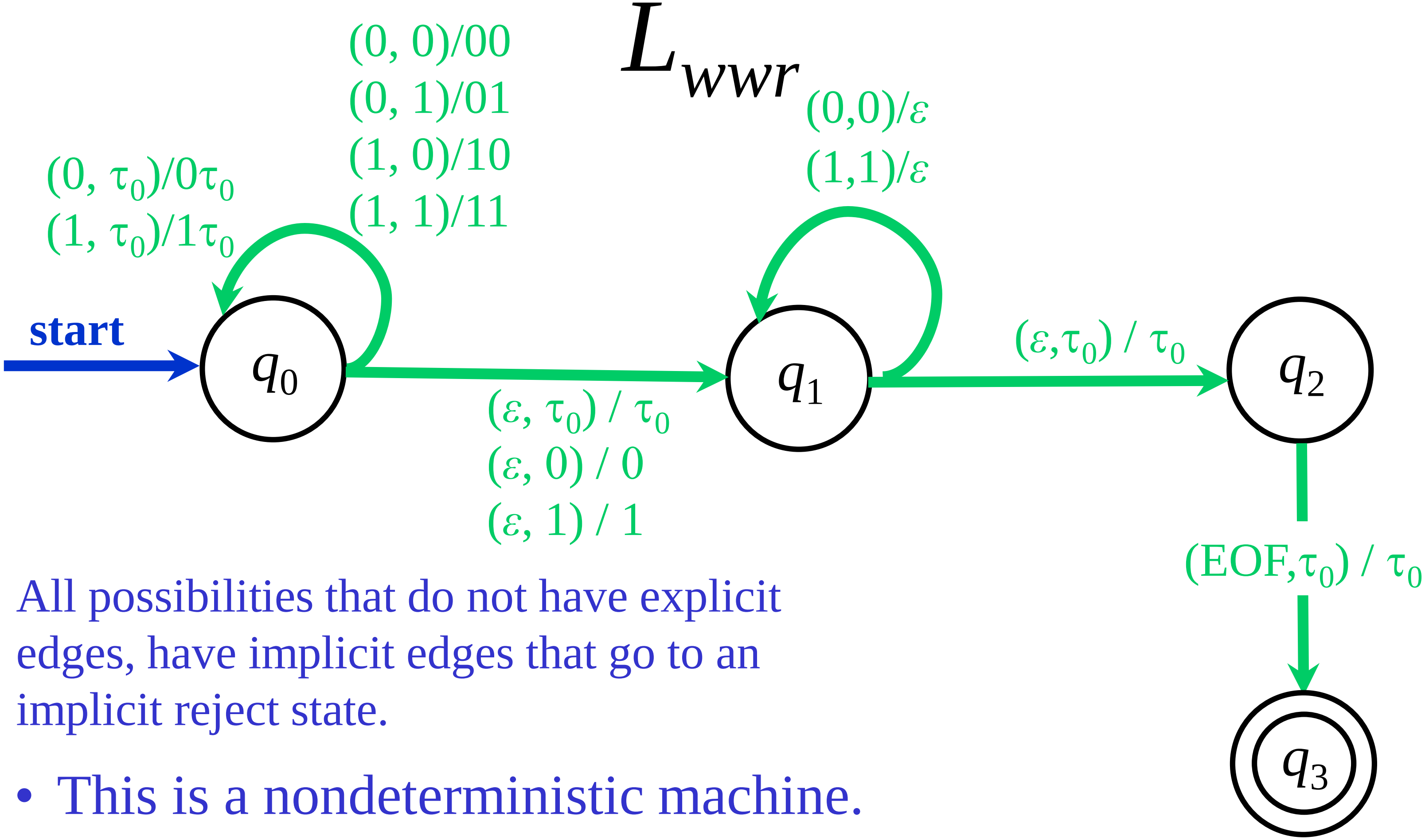


$A \rightarrow x_1\ x_2 \ldots x_n$ in R

Theorem

A language L is CFL $\Leftrightarrow$ it can be accepted by a PDA.

Graphical Notation for PDA of L_{wwr}



All possibilities that do not have explicit edges, have implicit edges that go to an implicit reject state.

- This is a nondeterministic machine.
- Think of the machine as following all possible paths.
- Kill a path if it leads to a reject state.
- If any path leads to an accept state, then the machine

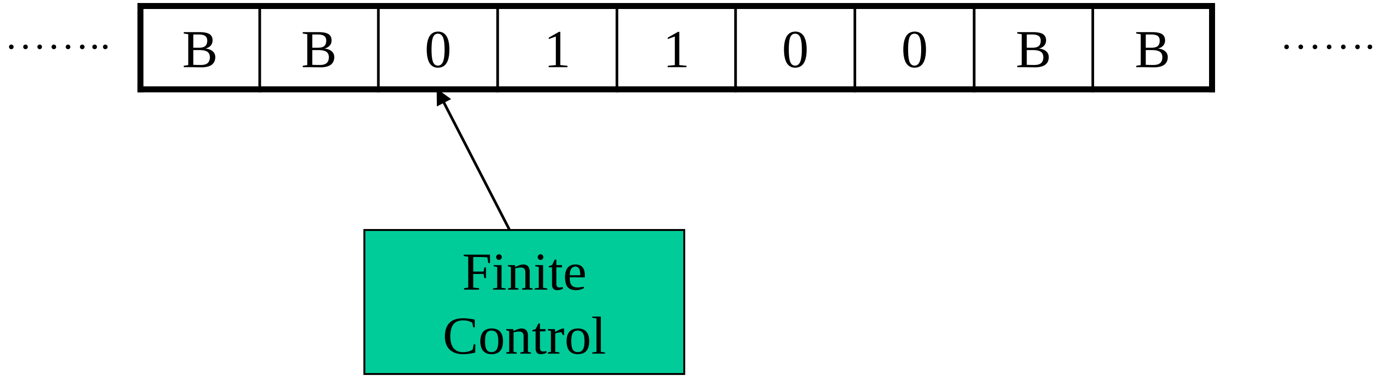
- TMs model the computing capability of a general purpose computer, which informally can be described as:
  - Effective procedure
    - Finitely describable
    - Well defined, discrete, “mechanical” steps
    - Always terminates
  - Computable function
    - A function computable by an effective procedure

- 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 $Q \times \Gamma \to Q \times \Gamma \times \{L,R\}$

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

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

	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)

$q_0 0011$	$Xq_1 011$	$X0q_1 11$	$Xq_2 0Y1$	$q_2 X0Y1$
$Xq_0 0Y1$	$XXq_1 Y1$	$XXYq_1 1$	$XXq_2 YY$	$Xq_2 XYY$
$XXq_0 YY$	$XXYq_3 Y$	$XXYYq_3$	$XXYYBq_4$	

- 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 $qx_1x_2\ldots x_{i-1}x_ix_{i+1}\ldots x_n \vdash pByx_2\ldots x_{i-1}x_ix_{i+1}\ldots x_n$

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

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

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

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

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

$x_1x_2\ldots x_nq \vdash x_1x_2\ldots x_nyp$

- **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM, and let w be a string in Σ^*. Then w is *accepted* by M iff

$$q_0w \vdash^* \alpha_1 p \alpha_2$$

where p is in F and α_1 and α_2 are in Γ^*

- **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM. The *language accepted by M*, denoted L(M), is the set

$$\{w \mid w \text{ is in } \Sigma^* \text{ and } w \text{ is accepted by } M\}$$

- **Notes:**
  - In contrast to FA and PDAs, if a TM simply *passes through* a final state then the string is accepted.
  - Given the above definition, no final state of a TM need to have any transitions. *Henceforth, this is our assumption*.
  - **If x is NOT in L(M) then M may enter an infinite loop, or halt in a non-final state**.
  - Some TMs halt on ALL inputs, while others may not. In either case the language defined by TM is still well defined.

- **Definition:** Let *L* be a language. Then *L* is *recursively enumerable* if <u>there exists</u> a TM *M* such that L = L(M).

  - If *L* is r.e. then L = L(M) for some TM *M*, and
    - If *x* is in *L* then *M* halts in a final (accepting) state.
    - If *x* is not in *L* then *M* may halt in a non-final (non-accepting) state or no transition is available, or loop forever.

- **Definition:** Let *L* be a language. Then *L* is *recursive* if there exists a TM *M* such that L = L(M) and M halts on all inputs.

  - If *L* is recursive then L = L(M) for some TM *M*, and
    - If *x* is in *L* then *M* halts in a final (accepting) state.
    - If *x* is not in *L* then *M* halts in a non-final (non-accepting) state or no transition is available (does <u>not</u> go to infinite loop).

Notes:

- The set of all recursive languages is a subset of the set of all recursively enumerable languages

- Terminology is easy to confuse: A *TM* is not recursive or recursively enumerable, rather a *language* is recursive or recursively enumerable.

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

References

- cs.www.umd.edu › users › mvz › cmsc330-f06
- www.iitg.ac.in › gkd › oct › oct11 ›
- nptel.ac.in › content › storage2 › courses › module3
- www.geeksforgeeks.org
- cs.www3.stonybrook.edu › ~cse350 › slides › cfg3
- www.slideshare.net › rajendranjrf › chomsky-greibach-normal-forms
- cse.www.iitd.ernet.in › ~naveen › courses › COL352 › slides
- cs.www.rpi.edu › ~moorthy › Courses › modcomp › slides › NFA
- cs.people.nctu.edu.tw › ~lwhsu › course › slides
- cs.www.rpi.edu › ~moorthy › Courses >modcomp>slides>PDA
- cs.people.nctu.edu.tw › ~lwhsu › course › slides
- cs.www.unm.edu › ~joel › Pushdown Automaton