

ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028



A T M E

College of Engineering

**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
(DATA SCIENCE)**

(ACADEMIC YEAR 2023-24)

LESSON NOTES

SUBJECT: AUTOMATA THEORY AND COMPILER DESIGN

SUB CODE: 21CS51

SEMESTER: V- 2021 CBCS Scheme

INSTITUTIONAL MISSION AND VISION

Objectives

- ☐ To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- ☐ To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- ☐ To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels
- ☐ To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- ☐ To cultivate strong community relationships and involve the students and the staff in local community service.
- ☐ To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

Vision

- ☐ Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.

- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING AND ENGINEERING
(DATA SCIENCE &ENGINEERING)**

Vision of The Department

- To impart technical education in the field of data science of excellent quality with a high level of professional competence, social responsibility, and global awareness among the students

Mission

- To impart technical education that is up to date, relevant and makes students competitive and employable at global level
- To provide technical education with a high sense of discipline, social relevance in an intellectually, ethically and socially challenging environment for better tomorrow
- Educate to the global standards with a benchmark of excellence and to kindle the spirit of innovation.

Program Outcomes(PO)

- **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

- **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
- **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
- **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
- **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
- **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

- **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
- **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

- PSO1: Develop relevant programming skills to become a successful data scientist
- PSO2: Apply data science concepts and algorithms to solve real world problems of the society
- PSO3: Apply data science techniques in the various domains like agriculture, education healthcare for better society

Program Educational Objectives (PEOs):

PEO1: Develop cutting-edge skills in data science and its related technologies, such as machine learning, predictive analytic, and data engineering.

PEO2: Design and develop data-driven solutions to real-world problems in a business, research, or social environment.

PEO3: Apply data engineering and data visualization techniques to discover, investigate, and interpret data.

PEO4: Demonstrate ethical and responsible data practices in problem solving

PEO5: Integrate fields within computer science, optimization, and statistics to develop better solutions

V Semester

AUTOMATA THEORY AND COMPILER DESIGN			
Course Code	21CS51	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:0:0	SEE Marks	50
Total Hours of Pedagogy	40	Total Marks	100
Credits	03	Exam Hours	03
Course Learning Objectives			
CLO 1. Introduce the fundamental concepts of Automata Theory, Formal Languages and compiler design			
CLO 2. Principles Demonstrate Application of Automata Theory and Formal Languages in the field of compiler design			
CLO 3. Develop understanding of computation through Push Down Automata and Turing Machines			
CLO 4. Introduce activities carried out in different phases of Phases compiler			
CLO 5. Identify the undecidability problems.			
Teaching-Learning Process (General Instructions)			
These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes.			
1. Lecturer method (L) needs not to be only a traditional lecture method, but alternative effective teaching methods could be adopted to attain the outcomes.			
2. Use of Video/Animation to explain functioning of various concepts.			
3. Encourage collaborative (Group Learning) Learning in the class.			
4. Ask at least three HOT (Higher order Thinking) questions in the class, which promotes critical thinking.			
5. Adopt Problem Based Learning (PBL), which fosters students' Analytical skills, develop design thinking skills such as the ability to design, evaluate, generalize, and analyze information rather than simply recall it.			
6. Introduce Topics in manifold representations.			
7. Show the different ways to solve the same problem with different approaches and encourage the students to come up with their own creative ways to solve them.			
8. Discuss how every concept can be applied to the real world - and when that's possible, it helps improve the students' understanding.			
Module-1			
Introduction to Automata Theory: Central Concepts of Automata theory, Deterministic Finite Automata(DFA), Non- Deterministic Finite Automata(NFA) ,Epsilon- NFA, NFA to DFA Conversion, Minimization of DFA			
Introduction to Compiler Design: Language Processors, Phases of Compilers			
Textbook 1: Chapter1 – 1.5, Chapter2 – 2.2,2.3,2.5 Chapter4 –4.4			
Textbook 2: Chapter1 – 1.1 and 1.2			
Teaching-Learning Process	Chalk and board, Active Learning, Problem based learning		
Module-2			
Regular Expressions and Languages: Regular Expressions, Finite Automata and Regular Expressions, Proving Languages Not to Be Regular			
Lexical Analysis Phase of compiler Design: Role of Lexical Analyzer, Input Buffering , Specification of Token, Recognition of Token.			

Textbook 1: Chapter3 – 3.1, 3.2, Chapter4- 4.1	
Textbook 2: Chapter3- 3.1 to 3.4	
Teaching-Learning Process	Chalk and board, Active Learning, Demonstration
Module-3	
Context Free Grammars: Definition and designing CFGs, Derivations Using a Grammar, Parse Trees, Ambiguity and Elimination of Ambiguity, Elimination of Left Recursion, Left Factoring.	
Syntax Analysis Phase of Compilers: part-1: Role of Parser , Top-Down Parsing	
Textbook 1: Chapter 5 – 5.1.1 to 5.1.6, 5.2 (5.2.1, 5.2.2), 5.4	
Textbook 2: Chapter 4 – 4.1, 4.2, 4.3 (4.3.2 to 4.3.4) ,4.4	
Teaching-Learning Process	Chalk and board, Problem based learning, Demonstration
Module-4	
Push Down Automata: Definition of the Pushdown Automata, The Languages of a PDA.	
Syntax Analysis Phase of Compilers: Part-2: Bottom-up Parsing, Introduction to LR Parsing: SLR, More Powerful LR parsers	
Textbook1: Chapter 6 – 6.1, 6.2	
Textbook2: Chapter 4 – 4.5, 4.6, 4.7 (Up to 4.7.4)	
Teaching-Learning Process	Chalk & board, Problem based learning
Module-5	
Introduction to Turing Machine: Problems that Computers Cannot Solve, The Turing machine, problems, Programming Techniques for Turing Machine, Extensions to the Basic Turing Machine	
Undecidability : A language That Is Not Recursively Enumerable, An Undecidable Problem That Is RE.	
Other Phases of Compilers: Syntax Directed Translation- Syntax-Directed Definitions, Evaluation Orders for SDD's. Intermediate-Code Generation- Variants of Syntax Trees, Three-Address Code.	
Code Generation- Issues in the Design of a Code Generator	
Textbook1: Chapter 8 – 8.1, 8.2,8.3,8.4 Chapter 9 – 9.1,9.2	
Textbook2: Chapter 5 – 5.1, 5.2, Chapter 6- 6.1,6.2 Chapter 8- 8.1	
Teaching-Learning Process	Chalk and board, MOOC
Course Outcomes At the end of the course the student will be able to: CO 1. Acquire fundamental understanding of the core concepts in automata theory and Theory of Computation CO 2. Design and develop lexical analyzers, parsers and code generators CO 3. Design Grammars and Automata (recognizers) for different language classes and become knowledgeable about restricted models of Computation (Regular, Context Free) and their relative powers. CO 4. Acquire fundamental understanding of the structure of a Compiler and Apply concepts automata theory and Theory of Computation to design Compilers CO 5. Design computations models for problems in Automata theory and adaptation of such model in the field of compilers	
Assessment Details (both CIE and SEE) The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/	

course if the student secures not less than 35% (18 Marks out of 50) in the semester-end examination (SEE), and a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together

Continuous Internal Evaluation:

Three Unit Tests each of **20 Marks (duration 01 hour)**

1. First test at the end of 5th week of the semester
2. Second test at the end of the 10th week of the semester
3. Third test at the end of the 15th week of the semester

Two assignments each of **10 Marks**

1. First assignment at the end of 4th week of the semester
2. Second assignment at the end of 9th week of the semester

Group discussion/Seminar/quiz any one of three suitably planned to attain the COs and POs for **20 Marks (duration 01 hours)**

1. At the end of the 13th week of the semester

The sum of three tests, two assignments, and quiz/seminar/group discussion will be out of 100 marks and will be **scaled down to 50 marks**

(to have a less stressed CIE, the portion of the syllabus should not be common /repeated for any of the methods of the CIE. Each method of CIE should have a different syllabus portion of the course).

CIE methods /question paper has to be designed to attain the different levels of Bloom's taxonomy as per the outcome defined for the course.

Semester End Examination:

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the subject (**duration 03 hours**)

1. The question paper will have ten questions. Each question is set for 20 marks and Marks scored shall be proportionally reduced to 50 marks
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.
3. The students have to answer 5 full questions, selecting one full question from each module

Suggested Learning Resources:

Textbooks

1. John E Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, "Introduction to Automata Theory, Languages and Computation", Third Edition, Pearson.
2. Alfred V.Aho, Monica S.Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers Principles, Techniques and Tools", Second Edition, Pearson.

Reference:

1. Elaine Rich, "Automata, Computability and complexity", 1st Edition, Pearson Education, 2018.
2. K.L.P Mishra, N Chandrashekar, 3rd Edition, "Theory of Computer Science", PHI, 2012.
3. Peter Linz, "An introduction to Formal Languages and Automata", 3rd Edition, Narosa Publishers, 1998.
4. K Muneeswaran, "Compiler Design", Oxford University Press 2013.

Weblinks and Video Lectures (e-Resources):

1. <https://nptel.ac.in/courses/106/106/106106049/#>
2. <https://nptel.ac.in/courses/106/104/106104123/>
3. <https://www.jflap.org/>

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

MODULE I:

1. Introduction
2. Why study the Theory of Computation?
3. Strings
4. Languages
5. A Finite State Machines (FSM)
 - 5.1. Deterministic FSM
 - 5.2. Nondeterministic FSMs
 - 5.3. Simulators for FSMs
 - 5.4. Minimizing FSMs
6. Finite State Transducers
7. Bidirectional Transducers.

1. Introduction

The term "Automata" is derived from the Greek word "αὐτόματα" which means "self-acting". An **automaton** (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

An automaton with a finite number of states is called a **Finite Automaton (FA)** or **Finite State Machine (FSM)**.

2. Why to study Theory of Computation?

Theory of computation is mainly concerned with the study of how problems can be solved using algorithms. It is the study of mathematical properties both of problems and of algorithms for solving problems that depend on neither the details of today's technology nor the programming language.

It is still useful in two key ways:

- It provides a set of **abstract structures** that are useful for solving certain classes of problems. These abstract structures can be implemented on whatever hardware/software platform is available
- It defines provable limits to **what can be computed** regardless of processor speed or memory size. An understanding of these limits helps us to focus our design effort in areas in which it can pay off, rather than on the computing equivalent of the search for a perpetual motion machine.

The goal is to discover fundamental properties of the problems like:

- Is there any computational **solution** to the problem? If not, is there a restricted but useful variation of the problem for which a solution does exist?
- If a solution exists, can it be implemented using some **fixed amount of memory**?
- If a solution exists, how **efficient** is it? More specifically, how do its time and space requirements grow as the size of the problem grows?
- Are there groups of problems that are **equivalent** in the sense that if there is an efficient

solution to one member of the group there is an efficient solution to all the others?

Applications of theory of computation:

- ***Development of Machine Languages:*** Enables both machine-machine and person-machine communication. Without them, none of today's applications of computing could exist. Example: Network communication protocols, HTML etc.
- ***Development of modern programming languages:*** Both the design and the implementation of modern programming languages rely heavily on the theory of context-free languages. Context-free grammars are used to document the languages syntax and they form the basis for the parsing techniques that all compilers use.
- ***Natural language processing:*** It is a field of computer science, artificial intelligence, and computational linguistics concerned with the interactions between computers and human (natural) languages.
- ***Automated hardware systems:*** Systems as diverse as parity checkers, vending machines, communication protocols, and building security devices can be straightforwardly described as finite state machines, which is a part of theory of computation.
- ***Video Games:*** Many interactive video games use large nondeterministic finite state machines.
- ***Security*** is perhaps the most important property of many computer systems. The undecidability results of computation show that there cannot exist a general-purpose method for automatically verifying arbitrary security properties of programs.
- ***Artificial intelligence:*** Artificial intelligence programs solve problems in task domains ranging from medical diagnosis to factory scheduling. Various logical frameworks have been proposed for representing and reasoning with the knowledge that such programs exploit.
- ***Graph Algorithms:*** Many natural structures, including ones as different as organic molecules and computer networks can be modeled as graphs. The theory of complexity tells us that, is there exist efficient algorithms for answering some important questions about graphs. Some questions are "hard", in the sense that no efficient algorithm for them is known nor is one likely to be developed.

3. Strings

Alphabet

Definition: An **alphabet** is any finite set of symbols denoted by Σ (Sometimes also called as **characters** or **symbols**).

Example: $\Sigma = \{a, b, c, d\}$ is an alphabet set where 'a', 'b', 'c', and 'd' are symbols.

String

Definition: A **string** is a finite sequence of symbols taken from Σ .

Example: ‘cabcad’ is a valid string on the alphabet set $\Sigma = \{a, b, c, d\}$

Alphabet name	Alphabet symbols	Example strings
The lower case English alphabet	$\{a, b, c, \dots, z\}$	$\epsilon, aabbccg, aaaaa$
The binary alphabet	$\{0, 1\}$	$\epsilon, 0, 001100, 11$
A star alphabet	$\{\star, \odot, \star, \star, \star, \star\}$	$\epsilon, \odot\odot, \odot\star\star\star\star\star$
A music alphabet	$\{o, \text{quarter note}, \text{eighth note}, \text{sixteenth note}, \text{beamed eighth notes}, \text{beamed sixteenth notes}, \bullet\}$	$\epsilon, \text{quarter note}, \text{eighth note}, \text{sixteenth note}, \text{beamed eighth notes}, \text{beamed sixteenth notes}, \bullet$

3.1.

Functions on Strings

Length of a String

Definition: It is the number of symbols present in a string. (Denoted by $| \cdot |$).

Examples: If $s = \text{‘cabcad’}$, $|s| = 6$; Also $|11001101| = 7$

If $|s| = 0$, it is called an empty string, denoted by ϵ . $|\epsilon| = 0$

Concatenation of strings: The *concatenation* of two strings s and t , written s/t or simply st , is the string formed by appending t to s . For example, if $x = \text{good}$ and $y = \text{bye}$, then $xy = \text{goodbye}$. So $|xy| = |x| + |y|$.

The empty string, ϵ , is the **identity** for concatenation of strings. ($xe = ex = x$).

Concatenation, as a function defined on strings is associative. $(st)w = s(tw)$.

String Replication

For each string w and each natural number i , the string w^i is defined as:

Example: $a^3 = aaa$, $(bye)^2 = \text{byebye}$, $a^0b^3 = bbb$

$$w^0 = \epsilon$$

$$w^{i+1} = w^i w$$

String Reversal: For each string w , the reverse of w , written as w^R , is defined as:

Theorem: If w and x are strings, then $(wx)^R = x^R w^R$.

For example, $(\text{nametag})^R = (\text{tag})^R (\text{name})^R = \text{gateman}$.

Proof: The proof is by induction on $|x|$:

Base case: $|x| = 0$. Then $x = \epsilon$, and $(wx)^R = (w\epsilon)^R = (w)^R = \epsilon w^R = \epsilon^R w^R = x^R w^R$.

Prove: $\forall n \geq 0 ((|x| = n) \rightarrow ((wx)^R = x^R w^R)) \rightarrow ((|x| = n + 1) \rightarrow ((wx)^R = x^R w^R))$.

Consider any string x , where $|x| = n + 1$. Then $x = ua$ for some character a and $|u| = n$. So:

$$\begin{aligned}
 (wx)^R &= (w(ua))^R && \text{rewrite } x \text{ as } ua \\
 &= ((wu)a)^R && \text{associativity of concatenation} \\
 &= a(wu)^R && \text{definition of reversal} \\
 &= a(u^R w^R) && \text{induction hypothesis} \\
 &= (au^R)w^R && \text{associativity of concatenation} \\
 &= (ua)^R w^R && \text{definition of reversal} \\
 &= x^R w^R && \text{rewrite } ua \text{ as } x
 \end{aligned}$$

3.2. Relations on strings

Substring: A string s is a substring of a string t iff s occurs contiguously as part of t .

For example: aaa is a substring of $aaabbbbaaa$, $aaaaaa$ is not a substring of $aaabbbbaaa$

Proper Substring: A string r is a proper substring of a string t , iff t is a substring of t and $s \neq t$. Every string is a substring (although not a proper substring) of itself. The empty string, ϵ , is a substring of every string.

Prefix: A string s is a prefix of t , iff $\exists x \in \Sigma^* (t = sx)$. A string s is a proper prefix of a string t iff s is a prefix of t and $s \neq t$. Every string is a prefix (although not a proper prefix) of itself. The empty string ϵ , is a prefix of every string. For example. the prefixes of $abba$ are: ϵ , a , ab , abb , $abba$.

Suffix: A string s is a suffix of t , iff $\exists x \in \Sigma^* (t = xs)$. A string s is a proper suffix of a string t iff s is a suffix of t and $s \neq t$. Every string is a suffix (although not a proper suffix) of itself. The empty string ϵ , is a suffix of every string. For example. the suffixes of $abba$ are: ϵ , a , ba , bba , $abba$.

4. Languages

A language is a (finite or infinite) set of strings over a finite alphabet Σ . When we are talking about more than one language, we will use the notation Σ_L , to mean the alphabet from which the strings in the language L are formed.

Let $\Sigma = \{a, b\}$. $\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab \}$.

Some examples of languages over Σ are:

Φ , $\{\epsilon\}$, $\{a, b\}$, $\{\epsilon, a, aa, aaa, aaaa, aaaaa, \dots\}$, $\{\epsilon, a, aa, aaa, aaaa, aaaaa, \dots\}$

4.1. Techniques for Defining Languages

There are many ways. Since languages are sets. we can define them using any of the set-defining techniques

Ex-1: All a 's Precede All b 's,

$L = \{w \in \{a,b\}^* : \text{an } a \text{'s precede all } b \text{'s in } w\}$. The strings ϵ , a , aa , $aabbb$, and bb are in L . The strings aba , ba , and abc are not in L .

Ex-2: Strings that end in 'a'

$L = \{x : \exists y \in \{a, b\}^*, (x = ya)\}$. The strings a , aa , aaa , $baaa$ and ba are in L . The strings ϵ , bab , and bca are not in L . L consists of all strings that can be formed by taking some string in $\{a, b\}^*$ and concatenating a single a onto the end of it.

Ex-3: Empty language

$L = \{\} = \Phi$, the language that contains no strings. **Note:** $L = \{\epsilon\}$ the language that contains a single string, ϵ . Note that L is different from Φ .

Ex-4: Strings of all 'a' s containing zero or more 'a' s

Let $L = \{a^n : n \geq 0\}$. $L = (\epsilon, a, aa, aaa, aaaa, aaaaa, \dots)$

Ex-5: We define the following languages in terms of the prefix relation on strings:

$L_1 = \{w \in \{a, b\}^* : \text{no prefix of } w \text{ contains } b\} = \{ \epsilon, a, aa, aaa, aaaa, aaaaa, aaaaaa, \dots \}$.

$L_2 = \{w \in \{a, b\}^* : \text{no prefix of } w \text{ starts with } b\} = \{w \in \{a, b\}^* : \text{the first character of } w \text{ is } a\} \cup \{\epsilon\}$.

$L_3 = \{w \in \{a, b\}^* : \text{every prefix of } w \text{ starts with } b\} = \emptyset$. L_3 is equal to \emptyset because ϵ is a prefix of every string. Since ϵ does not start with b , no strings meet L_3 's requirement.

Languages are sets. So, a computational definition of a language can be given in two ways;

- a **language generator**, which enumerates (lists) the elements of the language, or
- a **language recognizer**, which decides whether or not a candidate string is in the language and returns *True* if it is and *False* if it isn't.

For example, the logical definition. $L = \{x : \exists y \in \{a, b\}^* (x = ya)\}$ can be turned into either a language generator (enumerator) or a language recognizer.

In some cases, when considering an enumerator for a language, we may care about the order in which the elements of L are generated. If there exists a total order D of the elements of Σ^* , then we can use D to define on L a total order called **lexicographic order** (written $<_L$):

- Shorter strings precede longer ones: $\forall x (\forall y ((|x| < |y|) \Rightarrow (x <_L y)))$ and
- Of strings that are the same length sort them in dictionary order using D .

Let $L = \{w \in \{a, b\}^* : \text{all } a\text{'s precede all } b\text{'s}\}$. The lexicographic enumeration of L is:

$\epsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb, aaaa, aaab, aabb, abbb, bbbb, aaaaa, \dots$

4.2. Cardinality of a Language

- Cardinality refers to the number of strings in the language.
- The smallest language over any alphabet is \emptyset , whose cardinality is 0.
- The largest language over any alphabet Σ is Σ^* . Suppose that $\Sigma = \emptyset$, then $\Sigma^* = \{\epsilon\}$ and $|\Sigma^*| = 1$. In general, $|\Sigma^*|$ is infinite.

Theorem: If $\Sigma \neq \emptyset$ then Σ^* is countably infinite.

Proof: The elements of Σ^* can be lexicographically enumerated by a straightforward procedure that:

- Enumerates all strings of length 0, then length 1, then length 2, and so forth.
- Within the strings of a given length, enumerates them in dictionary order.

This enumeration is infinite since there is no longest string in Σ^* . By Theorem A.1, since there exists an infinite enumeration of Σ^* , it is countably infinite.

4.4. Functions on Languages

Since languages are sets, all of the standard set operations are well-defined on languages.

Union, intersection, difference and complement are quite useful

Let: $\Sigma = \{a, b\}$.
 $L_1 = \{\text{strings with an even number of a's}\}$.
 $L_2 = \{\text{strings with no b's}\} = \{\epsilon, a, aa, aaa, aaaa, aaaaa, aaaaaa, \dots\}$.
 $L_1 \cup L_2 = \{\text{all strings of just a's plus strings that contain b's and an even number of a's}\}$.
 $L_1 \cap L_2 = \{\epsilon, aa, aaaa, aaaaaa, aaaaaaaa, \dots\}$.
 $L_2 - L_1 = \{a, aaa, aaaaa, aaaaaaa, \dots\}$.
 $\neg(L_2 - L_1) = \{\text{strings with at least one b}\} \cup \{\text{strings with an even number of a's}\}$.

Concatenation

Let L_1 and L_2 be two languages defined over some alphabet Σ . Then their concatenation, written $L_1 L_2$ is:

$$L_1 L_2 = \{w \in \Sigma^* : \exists s \in L_1 (\exists t \in L_2 (w = st))\}.$$

Example: Let: $L_1 = \{\text{cat, dog, mouse, bird}\}$. $L_2 = \{\text{bone, food}\}$.

$L_1 L_2 = \{\text{catbone, catfood, dogbone, dogfood, mousebone, mousefood, birdbone, birdfood}\}$.

The language $\{\epsilon\}$ is the **identity for concatenation** of languages. So for all languages L ,

$$L\{\epsilon\} = \{\epsilon\}L = L.$$

The language Φ is a zero for concatenation of languages. So, for all languages L , $L\Phi = \Phi L = \Phi$. That Φ is a zero follows from the definition of the concatenation of two languages as the set consisting of all strings that can be formed by selecting some string 's' from the first language and some string 't' from the second language and then concatenating them together. There are no ways to select a string from the empty set.

Concatenation on languages is **associative**. So, for all languages $L_1 L_2$ and L_3 :

$$((L_1 L_2) L_3 = L_1 (L_2 L_3)).$$

Reverse

Let L be a language defined over some alphabet Σ . Then the reverse of L , written L^R is:

$$L^R = \{w \in \Sigma^* : w = x^R \text{ for some } x \in L\}.$$

In other words, L^R is the set of strings that can be formed by taking some string in L and reversing it

Theorem: If L_1 and L_2 are languages, then $(L_1 L_2)^R = L_2^R L_1^R$.

Proof: If x and y are strings, then $\forall x (\forall y ((xy)^R = y^R x^R))$ Theorem 2.1

$$(L_1 L_2)^R = \{(xy)^R : x \in L_1 \text{ and } y \in L_2\}$$

Definition of concatenation
of languages

$$= \{y^R x^R : x \in L_1 \text{ and } y \in L_2\}$$

Lines 1 and 2

$$= L_2^R L_1^R$$

Definition of concatenation
of languages

Kleene Star

Definition: The **Kleene star** denoted by Σ^* , is a unary operator on a set of symbols or strings, Σ , that gives the infinite set of all possible strings of all possible lengths over Σ including ϵ .

Representation: $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$ where Σ^p is the set of all possible strings of length p .

Example: If $\Sigma = \{a, b\}$, $\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, \dots\}$

Let $L = \{\text{dog, cat, fish}\}$. Then:
 $L^* = \{\epsilon, \text{dog, cat, fish, dogdog, dogcat, ...},$
 $\text{fishdog, ...}, \text{fishcatfish, fishdogfishcat, ...}\}$.

Kleene Closure / Plus

Definition: The set Σ^+ is the infinite set of all possible strings of all possible lengths over Σ excluding ϵ . $\Sigma^+ = \Sigma^* - \{\epsilon\}$ Representation:

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$

Example: If $\Sigma = \{a, b\}$, $\Sigma^+ = \{a, b, aa, ab, ba, bb, \dots\}$

Closure: A set S is closed under the operation $@$ if for every element x & y in S , $x@y$ is also an element of S .

4.5. A Language Hierarchy

A Machine-Based Hierarchy of Language Classes are shown in the diagram.

We have four language classes:

1. **Regular languages**, which can be accepted by some finite state machine.
2. **Context-free languages**, which can be accepted by some pushdown automaton.
3. **Decidable** (or simply D) languages, which can be decided by some Turing machine that always halts.
4. **Semi-decidable** (or SD) languages, which can be semi-decided by some Turing machine that halts on all strings in the language.

Each of these classes is a proper subset of the next class, as illustrated in the Figure.

As we move outward in the language hierarchy, we have access to tools with greater and expressive power. We can define $A^n B^n C^n$ as a decidable language but not as a context-free or a regular one. These matters because expressiveness generally comes at a price. The price may be: Computational efficiency, decidability and clarity.

- *Computational efficiency:* Finite state machines run in time that is linear in the length of the input string. A general context-free parser based on the idea of a pushdown automaton requires time that grows as the cube of the length of the input string. A

Turing machine may require time that grows exponentially (or faster) with the length of the input string.

- *Decidability*: There exist procedures to answer many useful questions about finite state machines. For example, does an FSM accept some particular string? Is an FSM minimal? Are two FSMs identical? A subset of those questions can be answered for pushdown automata. None of them can be answered for Turing machines.
- *Clarity*: There exist tools that enable designers to draw and analyze finite state machines. Every regular language can also be described using the regular expression pattern language. Every context-free language, in addition to being recognizable by some pushdown automaton, can be described with a context-free grammar

5. Finite State Machines (FSM)

A *finite state machines* (or FSM) is a computational device whose input is a string and whose output is one of two values; Accept and Reject. FSMs are also sometimes called *finite state automata* or *FSAs*.

5.1. Deterministic FSM

- We begin by defining the class of FSMs whose behavior is **deterministic**.
- These machines, makes exactly one move at each step
- The move is determined by the current state and the next input character.

Definition: Deterministic Finite State Machine (DFSM) is $M: M = (K, \Sigma, \delta, s, A)$, where:
 K is a finite set of states
 Σ is an alphabet
 $s \in K$ is the initial state
 $A \subseteq K$ is the set of accepting states, and
 δ is the transition function from $(K \times \Sigma)$ to K

Configuration: A *Configuration* of a DFSM M is an element of $K \times \Sigma^*$. Configuration captures the two things that make a difference to M 's future behavior: i) its current state, the input that remains to be read.

The *Initial Configuration* of a DFSM M , on input w , is (s_M, w) , where s_M is start state of M

The transition function δ defines the operation of a DFSM M one step at a time. δ is set of all pairs of states in M & characters in Σ . (Current State, Current Character) \rightarrow New State

Relation ‘yields’: Yields-in-one-step relates configuration, to configuration-1 to configuration- 2 iff M can move from configuration-1, to configuration-2 in one step. Let c be any element of Σ and let w be any element of Σ^* , then,

$$(q_1, cw) \vdash_M (q_2, w) \text{ iff } ((q_1, c), q_2) \in \delta$$

\vdash_M^* is the reflexive, transitive closure of \vdash_M

Complete vs Incomplete FSM

Complete FSM: A transition is defined for every possible state and every possible character in the alphabet. Note: This can cause FSM to be larger than necessary, but ALWAYS processes the entire string

Incomplete FSM: One which defines a transition for every possible state & every possible character in the alphabet which can lead to an accepting state Note: If no transition is defined, the string is *Rejected*

Computation: A Computation by M is a finite sequence of configurations C_0, C_1, \dots, C_n for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ϵ) , for some state $q \in K_M$
 - ϵ indicates empty string, entire string is processed & implies a complete DFSM
- $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \dots \vdash_M C_n$.

However, M **Halts** when the last character has to be processed or a next transition is not defined

Acceptance / Rejection

A DFSM M , **Accepts** a string w iff $(s, w) \vdash_M^* (q, \epsilon)$, for some $q \in A_M$.

A DFSM M , **Rejects** a string w iff $(s, w) \vdash_M^* (q, \epsilon)$, for some $q \notin A_M$.

The **language accepted by M** , denoted $L(M)$, is the set of all strings accepted by M .

Regular languages

A language is regular iff it is accepted by some DFSM. Some examples are listed below.

- $\{w \in \{a, b\}^* \mid \text{every } a \text{ is immediately followed by } b\}$.
- $\{w \in \{a, b\}^* \mid \text{every } \mathbf{a \text{ region}} \text{ in } w \text{ is of even length}\}$
- binary strings with odd parity.

Designing Deterministic Finite State Machines

Given some language L . how should we go about designing a DFSM to accept L ? In general. as in any design task. There is no magic bullet. But there are two related things that it is helpful to think about:

- Imagine any DFSM M that accepts L . As a string w is being read by M , what properties of the part of w that has been seen so far are going to have any bearing on the ultimate answer that M needs to produce? Those are the properties that M needs to record.
- If L is infinite but M has a finite number of states, strings must "cluster". In other words, multiple different strings will all drive M to the same state. Once they have done that, none of their differences matter anymore. If they've driven M to the same state, they share a fate. No matter what comes next, either all of them cause M to accept or all of them cause M to reject.

MODULE II:

1. Regular Expression
2. Equivalence of Re and NFA
3. DFA
4. Pumping lemma for regular languages
5. Grammars
6. Parse Trees

1. Regular Expressions

- A regular expression is used to specify a language, and it does so precisely.
- Regular expressions are very intuitive.
- Regular expressions are very useful in a variety of contexts.
- Given a regular expression, an NFA- ϵ can be constructed from it automatically.
- Thus, so can an NFA, a DFA, and a corresponding program, all automatically!

Definition:

- Let Σ be an alphabet. The regular expressions over Σ are:
 - \emptyset Represents the empty set $\{ \}$
 - ϵ Represents the set $\{\epsilon\}$
 - a Represents the set $\{a\}$, for any symbol a in Σ

Let r and s be regular expressions that represent the sets R and S , respectively.

- $r+s$ Represents the set $R \cup S$ (precedence 3)
- rs Represents the set RS (precedence 2)
- r^* Represents the set R^* (highest precedence)
- (r) Represents the set R (not an op, provides precedence)

- If r is a regular expression, then $L(r)$ is used to denote the corresponding language.
- **Examples:** Let $\Sigma = \{0, 1\}$

$(0 + 1)^*$	All strings of 0's and 1's
$0(0 + 1)^*$	All strings of 0's and 1's, beginning with a 0
$(0 + 1)^*1$	All strings of 0's and 1's, ending with a 1
$(0 + 1)^*0(0 + 1)^*$	All strings of 0's and 1's containing at least one 0
$(0 + 1)^*0(0 + 1)^*0(0 + 1)^*$	All strings of 0's and 1's containing at least two 0's
$(0 + 1)^*01^*01^*$	All strings of 0's and 1's containing at least two 0's
$(1 + 01^*0)^*$	All strings of 0's and 1's containing an even number of 0's

$1^*(01^*01^*)^*$
 $(1^*01^*0)^*1^*$

All strings of 0's and 1's containing an even number of 0's
 All strings of 0's and 1's containing an even number of 0's

Identities:

$$\emptyset u = u\emptyset = \emptyset$$

Multiply by 0

- $u + \emptyset = u$

- $u + u = u$

8. $u^* = (u^*)^*$

9. $u(v+w) = uv+uw$

10. $(u+v)w = uw+vw$

11. $(uv)^*u = u(vu)^*$

12. $(u+v)^* = (u^*+v)^*$

$$= u^*(u+v)^*$$

$$= (u+vu^*)^*$$

$$= (u^*v^*)^*$$

$$= u^*(vu^*)^*$$

$$= (u^*v)^*u^*$$

2. Equivalence of Regular Expressions and NFA-ε

- **Note:** Throughout the following, keep in mind that a string is accepted by an NFA-ε if there exists a path from the start state to a final state.
- **Lemma 1:** Let r be a regular expression. Then there exists an NFA-ε M such that $L(M) = L(r)$. Furthermore, M has exactly one final state with no transitions out of it.
- **Proof:** (by induction on the number of operators, denoted by $OP(r)$, in r).
- **Basis:** $OP(r) = 0$

Then r is either \emptyset , ϵ , or a , for some symbol a in Σ

- **Inductive Hypothesis:** Suppose there exists a $k \geq 0$ such that for any regular expression r where $0 \leq OP(r) \leq k$, there exists an NFA- ϵ such that $L(M) = L(r)$. Furthermore, suppose that M has exactly one final state.
- **Inductive Step:** Let r be a regular expression with $k + 1$ operators ($OP(r) = k + 1$), where $k + 1 \geq 1$.

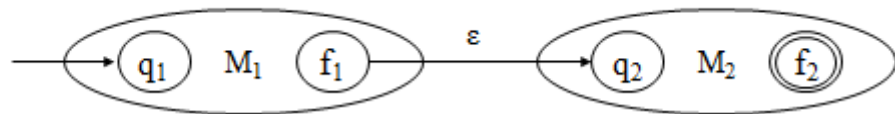
Case 1) $r = r_1 + r_2$

Since $OP(r) = k + 1$, it follows that $0 \leq OP(r_1), OP(r_2) \leq k$. By the inductive hypothesis there exist NFA- ϵ machines M_1 and M_2 such that $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. Furthermore, both M_1 and M_2 have exactly one final state.

Case 2) $r = r_1 r_2$

Since $OP(r) = k + 1$, it follows that $0 \leq OP(r_1), OP(r_2) \leq k$. By the inductive hypothesis there exist NFA- ϵ machines M_1 and M_2 such that $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. Furthermore, both M_1 and M_2 have exactly one final state.

Construct M as:



Case 3) $r = r_1^*$

Since $OP(r) = k + 1$, it follows that $0 \leq OP(r_1) \leq k$. By the inductive hypothesis there exists an NFA- ϵ machine M_1 such that $L(M_1) = L(r_1)$. Furthermore, M_1 has exactly one final state.

- **Example:**

Problem: Construct FA equivalent to RE, $r = 0(0+1)^*$

Solution:

$$\begin{aligned}
 r &= r_1 r_2 \\
 r_1 &= 0 \\
 r_2 &= (0+1)^* \\
 r_2 &= r_3^* \\
 r_3 &= 0+1 \\
 r_3 &= r_4 + r_5 \\
 r_4 &= 0 \\
 r_5 &= 1
 \end{aligned}$$

Transition graph:

3. Definitions Required to Convert a DFA to a Regular Expression

- Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA with state set $Q = \{q_1, q_2, \dots, q_n\}$, and define:
 $R_{i,j} = \{x \mid x \text{ is in } \Sigma^* \text{ and } \delta(q_i, x) = q_j\}$
 $R_{i,j}$ is the set of all strings that define a path in M from q_i to q_j .

- Lemma 2:** Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA. Then there exists a regular expression r such that $L(M) = L(r)$.

if $i=j$. Case 1) No transitions from q_i to q_j and $i \neq j$ $r_{i,j}^0 = \emptyset$

Case 2) At least one ($m \geq 1$) transition from q_i to q_j and $i \neq j$

$$r_{i,j}^0 = a_1 + a_2 + a_3 + \dots + a_m \quad \text{where } \delta(q_i, a_p) = q_j, \text{ for all } 1 \leq p \leq m$$

Case 3) No transitions from q_i to q_j and $i = j$

$$r_{i,j}^0 = \epsilon$$

Case 4) At least one ($m \geq 1$) transition from q_i to q_j and $i = j$

$$r_{i,j}^0 = a_1 + a_2 + a_3 + \dots + a_m + \epsilon \quad \text{where } \delta(q_i, a_p) = q_j \text{ for all } 1 \leq p \leq m$$

- Inductive Hypothesis:**

Suppose that $R_{i,j}^{k-1}$ can be represented by the regular expression $r_{i,j}^{k-1}$ for all $1 \leq i, j \leq n$, and some $k \geq 1$.

- Inductive Step:**

Consider $R_{i,j}^k = R_{i,k}^{k-1} (R_{k,k}^{k-1})^* R_{k,j}^{k-1} \cup R_{i,j}^{k-1}$. By the inductive hypothesis there exist regular expressions $r_{i,k}^{k-1}$, $r_{k,k}^{k-1}$, $r_{k,j}^{k-1}$, and $r_{i,j}^{k-1}$ generating $R_{i,k}^{k-1}$, $R_{k,k}^{k-1}$, $R_{k,j}^{k-1}$, and $R_{i,j}^{k-1}$, respectively. Thus, if we let

$$r_{i,j}^k = r_{i,k}^{k-1} (r_{k,k}^{k-1})^* r_{k,j}^{k-1} + r_{i,j}^{k-1}$$

then $r_{i,j}^k$ is a regular expression generating $R_{i,j}^k$, i.e., $L(r_{i,j}^k) = R_{i,j}^k$.

- Finally, if $F = \{q_{j1}, q_{j2}, \dots, q_{jr}\}$, then

$$r_{1,j1}^n + r_{1,j2}^n + \dots + r_{1,jr}^n$$

is a regular expression generating $L(M)$.•

4. Pumping Lemma for Regular Languages

- Pumping Lemma relates the size of string accepted with the number of states in a DFA
- What is the largest string accepted by a DFA with n states?
- Suppose there is no loop?
Now, if there is a loop, what type of strings are accepted *via* the loop(s)?
- **Lemma:** (the pumping lemma)

Let M be a DFA with $|Q| = n$ states. If there exists a string x in $L(M)$, such that $|x| \geq n$, then there exists a way to write it as $x = uvw$, where u, v , and w are all in Σ^* and:

- $1 \leq |uv| \leq n$
 - $|v| \geq 1$
 - such that, the strings $uv^i w$ are also in $L(M)$, for all $i \geq 0$
- Let:
 - $u = a_1 \dots a_s$
 - $v = a_{s+1} \dots a_t$
- Since $0 \leq s < t \leq n$ and $uv = a_1 \dots a_t$ it follows that:
 - $1 \leq |v|$ and therefore $1 \leq |uv|$
 - $|uv| \leq n$ and therefore $1 \leq |uv| \leq n$
- In addition, let:
 - $w = a_{t+1} \dots a_m$
- It follows that $uv^i w = a_1 \dots a_s (a_{s+1} \dots a_t)^i a_{t+1} \dots a_m$ is in $L(M)$, for all $i \geq 0$.

In other words, when processing the accepted string x , the loop was traversed once, but could have been traversed as many times as desired, and the resulting string would still be accepted.

4.1 Closure Properties of Regular Languages

1. Closure Under Union

- If L and M are regular languages, so is $L \cup M$.
- Proof: Let L and M be the languages of regular expressions R and S , respectively.
- Then $R+S$ is a regular expression whose language is $L \cup M$.

2. Closure Under Concatenation and Kleene Closure

- RS is a regular expression whose language is LM .
- R^* is a regular expression whose language is L^* .

3. Closure Under Intersection

- If L and M are regular languages, then so is $L \cap M$.
- Proof: Let A and B be DFA's whose languages are L and M , respectively.

4. Closure Under Difference

- If L and M are regular languages, then so is $L - M =$ strings in L but not M .
- Proof: Let A and B be DFA's whose languages are L and M , respectively.

5. Closure Under Complementation

- The complement of language L (w.r.t. an alphabet Σ such that Σ^* contains L) is $\Sigma^* - L$.
- Since Σ^* is surely regular, the complement of a regular language is always regular.

6. Closure Under Homomorphism

- If L is a regular language, and h is a homomorphism on its alphabet, then $h(L) = \{h(w) \mid w \text{ is in } L\}$ is also a regular language.

5. Grammar

- **Definition:** A grammar G is defined as a 4-tuple, $G = (V, T, S, P)$

Where,

- V is a finite set of objects called variables,
- T is a finite set of objects called terminal symbols,
- $S \in V$ is a special symbol called start variable,
- P is a finite set of productions.

Assume that V and T are non-empty and disjoint.

- **Example:**

Consider the grammar $G = (\{S\}, \{a, b\}, S, P)$ with P given by

$S \rightarrow aSb, \quad S \rightarrow \epsilon$.

For instance, we have $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$.

It is not hard to conjecture that $L(G) = \{a^n b^n \mid n \geq 0\}$.

5.1

Right, Left-Linear Grammar

- **Right-linear Grammar:** A grammar $G = (V, T, S, P)$ is said to be right-linear if all productions are of the form:

$$A \rightarrow xB,$$

$$A \rightarrow x,$$

Where $A, B \in V$ and $x \in T^*$.

○

Example#1:

$S \rightarrow abS \mid a$ is an example of a right-linear grammar.

- Can you figure out what language it generates?
- $L = \{w \in \{a,b\}^* \mid w \text{ contains alternating } a\text{'s and } b\text{'s, begins with an } a, \text{ and ends with a } b\}$
 $\cup \{a\}$
- $L((ab)^*a)$

- **Left-linear Grammar:** A grammar $G = (V, T, S, P)$ is said to be left-linear if all productions are of the form:

$$A \rightarrow Bx,$$

$$A \rightarrow x,$$

Where $A, B \in V$ and $x \in T^*$.

○ **Example#2:**

$$S \rightarrow Aab$$

$$A \rightarrow Aab \mid aB$$

$$B \rightarrow a$$

is an example of a left-linear grammar.

- Can you figure out what language it generates?
- $L = \{w \in \{a,b\}^* \mid w \text{ is } aa \text{ followed by at least one set of alternating } ab\text{'s}\}$
- $L(aaab(ab)^*)$

○ **Example#3:**

Consider the grammar

$$S \rightarrow A$$

$$A \rightarrow aB \mid \lambda$$

$$B \rightarrow Ab$$

This grammar is NOT regular.

- No "mixing and matching" left- and right-recursive productions.

5.2 Regular Grammar

- A linear grammar is a grammar in which at most one variable can occur on the right side of any production without restriction on the position of this variable.
- An example of linear grammar is $G = (\{S, S1, S2\}, \{a, b\}, S, P)$ with
 $S \rightarrow S1ab$,
 $S1 \rightarrow S1ab \mid S2$,
 $S2 \rightarrow a$.
- A **regular grammar** is one that is either right-linear or left-linear.

5.3 Testing Equivalence of Regular Languages

- Let L and M be reg langs (each given in some form).

To test if $L = M$

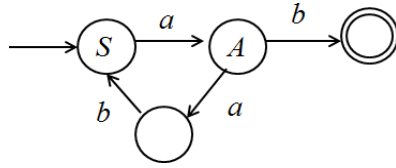
1. Convert both L and M to DFA's.
2. Imagine the DFA that is the union of the two DFA's (never mind there are two start states)
3. If TF-algo says that the two start states are distinguishable, then $L \neq M$, otherwise, $L = M$.

We can "see" that both DFA accept $L(\epsilon + (0+1)^*0)$. The result of the TF-algo is

Therefore the two automata are equivalent.

5.4 Regular Grammars and NFA's

- It's not hard to show that regular grammars generate and nfa's accept the same class of languages: the regular languages!
- It's a long proof, where we must show that
 - Any finite automaton has a corresponding left- or right-linear grammar,
 - And any regular grammar has a corresponding nfa.
- **Example:**
 - We get a feel for this by example.



Let $S \rightarrow aA$ $A \rightarrow abS \mid b$ CONTEXT FREE-GRAMMAR

- **Definition:** Context-Free Grammar (CFG) has 4-tuple: $G = (V, T, P, S)$

Where,

- | | | |
|---|---|--|
| V | - | A finite set of variables or <i>non-terminals</i> |
| T | - | A finite set of <i>terminals</i> (V and T do not intersect) |
| P | - | A finite set of <i>productions</i> , each of the form $A \rightarrow \alpha$,
Where A is in V and α is in $(V \cup T)^*$
Note: that α may be ϵ . |
| S | - | A starting non-terminal (S is in V) |

- **Example#1 CFG:**

$G = (\{S\}, \{0, 1\}, P, S)$

P:

(1) $S \rightarrow 0S1$

(2) $S \rightarrow \epsilon$

or just simply $S \rightarrow 0S1 \mid \epsilon$

- **Example Derivations:**

$S \Rightarrow 0S1$ (1)

$S \Rightarrow \epsilon$ (2)

$\Rightarrow 01$ (2)

$S \Rightarrow 0S1$ (1)

$\Rightarrow 00S11$ (1)

$\Rightarrow 000S111$ (1)

$\Rightarrow 000111$ (2)

- Note that G “generates” the language $\{0^k1^k \mid k \geq 0\}$

6. Derivation (or Parse) Tree

- **Definition:** Let $G = (V, T, P, S)$ be a CFG. A tree is a derivation (or parse) tree if:
 - Every vertex has a label from $V \cup T \cup \{\epsilon\}$
 - The label of the root is S
 - If a vertex with label A has children with labels X_1, X_2, \dots, X_n , from left to right, then

$$A \rightarrow X_1, X_2, \dots, X_n$$
 must be a production in P
 - If a vertex has label ϵ , then that vertex is a leaf and the only child of its’ parent
- More Generally, a derivation tree can be defined with any non-terminal as the root.

Definition: A derivation is *leftmost (rightmost)* if at each step in the derivation a production is applied to the leftmost (rightmost) non-terminal in the sentential form.

- The first derivation above is **leftmost**, second is **rightmost** and the third is neither.

MODULE III:

- 1. Context Free Grammar**
- 2. Minimization of Context Free Grammar**
- 3. Chomsky Normal Form**
- 4. Pumping Lemma for Context-Free Languages**
- 5. Pushdown Automata (PDA)**

1. Ambiguity in Context Free Grammar

- **Definition:** Let G be a CFG. Then G is said to be ambiguous if there exists an x in $L(G)$ with >1 leftmost derivations. Equivalently, G is said to be ambiguous if there exists an x in $L(G)$ with >1 parse trees, or >1 rightmost derivations.
- Note: Given a CFL L , there may be more than one CFG G with $L = L(G)$. Some are ambiguous and some are not.
- Definition: Let L be a CFL. If every CFG G with $L = L(G)$ is ambiguous, then L is inherently ambiguous.
- **Example:** Consider the string $aaab$ and the preceding grammar.
- The string has two left-most derivations, and therefore has two distinct parse trees and is ambiguous.

1.1 Eliminations of Useless Symbols

- **Definition:**
Let $G = (V, T, S, P)$ be a context-free grammar. A variable $A \in V$ is said to be useful if and only if there is at least one $w \in L(G)$ such that

$$S \Rightarrow^* xAy \Rightarrow^* w$$

with $x, y \in (V \cup T)^*$.

In words, a variable is useful if and only if it occurs in at least one derivation. A variable that is not useful is called useless. A production is useless if it involves any useless variable

- For a grammar with productions

$$S \rightarrow aSb \mid \lambda \mid A$$

$$A \rightarrow aA$$

A is useless variable and the production $S \rightarrow A$ plays no role since A cannot be eventually transformed into a terminal string; while A can appear in a sentential form derived from S , this sentential form can never lead to sentence!

Hence, removing $S \rightarrow A$ (and $A \rightarrow aA$) does not change the language, but does simplify the grammar.

- For a grammar with productions

$$S \rightarrow A$$

$$A \rightarrow aA \mid \lambda$$

$$B \rightarrow bA$$

B is useless so is the production $B \rightarrow bA$! Observe that, even though a terminal string can be derived from B , there is no way to get to B from S , i.e. cannot achieve

$$S \Rightarrow^* xBy.$$

- Example:**

Eliminate useless symbols and productions from $G = (V, T, S, P)$, where

$V = \{S, A, B, C\}$, $T = \{a, b\}$ and

P consists of

$$S \rightarrow aS \mid A \mid C$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

$$C \rightarrow aCb$$

First, note that the variable C cannot lead to any terminal string, we can then remove C and its associated productions, we get G_1 with $V_1 = \{S, A, B\}$, $T_1 = \{a\}$ and P_1 consisting of

$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

Next, we identify variables that cannot be reached from the start variable. We can create a dependency graph for V_1 . For a context-free grammar, a dependency graph has its vertices labeled with variables with an edge between any two vertices I and J if there is a production of the form

$$I \rightarrow xJy$$

Consequently, the variable B is shown to be useless and can be removed together with its associated production.

The resulting grammar $G' = (V', T', S, P')$ is with $V' = \{S, A\}$, $T' = \{a\}$ and P' consisting of

$$\begin{aligned} S &\rightarrow aS \mid A \\ A &\rightarrow a \end{aligned}$$

1.2 Eliminations of λ -Production

- **Definition :**

- a) Any production of a context-free grammar of the form

$$A \rightarrow \lambda$$

is called a λ -production.

- b) Any variable A for which the derivation

$$A \Rightarrow^* \lambda$$

is possible is called nullable.

- If a grammar contains some λ -productions or nullable variables but does not generate the language that contains an empty string, the λ -productions can be removed!

- **Example:**

Consider the grammar, G with productions

$$S \rightarrow aS_1b$$

$$S_1 \rightarrow aS_1b \mid \lambda$$

$L(G) = \{a^n b^n \mid n \geq 1\}$ which is a λ -free language. The λ -production can be removed after adding new productions obtained by substituting λ for S_1 on the right hand side.

We get an equivalent G' with productions

$$S \rightarrow aS_1b \mid ab$$

$$S_1 \rightarrow aS_1b \mid ab$$

- **Theorem:**

Let G be any context-free grammar with $\lambda \notin L(G)$. There exists an equivalent grammar G' without λ -productions.

Proof :

Find the set V_N of all nullable variables of G

1. For all productions $A \rightarrow \lambda$, put A in V_N

2. Repeat the following step until no further variables are added to V_N :

For all productions

$$B \rightarrow A_1 A_2 \dots A_n$$

where A_1, A_2, \dots, A_n are in V_N , put B in V_N .

With the resulting V_N , P' can be constructed by looking at all productions in P of the form

$$A \rightarrow x_1 x_2 \dots x_m, m \geq 1$$

where each $x_i \in V \cup T$.

For each such production of P , we put in P' the production plus all productions generated by replacing nullable variables with λ in all possible combinations. However, if all x_i are nullable, the resulting production $A \rightarrow \lambda$ is not put in P' .

- **Example:**

For the grammar G with

$$S \rightarrow ABaC$$

$$A \rightarrow BC$$

$$B \rightarrow b \mid \lambda$$

$$C \rightarrow D \mid \lambda$$

$$D \rightarrow d$$

the nullable variables are A , B , and C .

The equivalent grammar G' without λ -productions has P' containing

$$S \rightarrow ABaC \mid BaC \mid AaC \mid ABa \mid aC \mid Ba \mid Aa \mid a$$

$$A \rightarrow BC \mid C \mid B$$

$$B \rightarrow b$$

$$C \rightarrow D$$

$$D \rightarrow d$$

1.3 Eliminations of MODULE-Production

- **Definition:**

Any production of a context-free grammar of the form

$$A \rightarrow B$$

where $A, B \in V$ is called a MODULE-production.

- **Theorem:**

Let $G = (V, T, S, P)$ be any context-free grammar without λ -productions. There exists a context-free grammar $G' = (V', T', S, P')$ that does not have any MODULE-productions and that is equivalent to G .

Proof:

First of all, Any MODULE-production of the form $A \rightarrow A$ can be removed without any effect. We then need to consider productions of the form $A \rightarrow B$ where A and B are different variables.

Straightforward replacement of B (with $x_1 = x_2 = \lambda$) runs into a problem when we have

$$A \rightarrow B$$

$$B \rightarrow A$$

We need to find for each A , all variables B such that

$$A \Rightarrow^* B$$

This can be done via a dependency graph with an edge (I, J) whenever the grammar G has a MODULE-production $I \rightarrow J$; $A \Rightarrow^* B$ whenever there is a walk from A to B in the graph.

The new grammar G' is generated by first putting in P' all non-MODULE-productions of P . Then, for all A and B with $A \Rightarrow^* B$, we add to P'

$$A \rightarrow y_1 | y_2 | \dots | y_n$$

where $B \rightarrow y_1 | y_2 | \dots | y_n$ is the set of all rules in P' with B on the left. Note that the rules are taken from P' , therefore, none of y_i can be a single variable! Consequently, no MODULE-productions are created by this step.

- **Example:**

Consider a grammar G with

$$\begin{aligned} S &\rightarrow Aa | B \\ A &\rightarrow a | bc | B \\ B &\rightarrow A | bb \end{aligned}$$

We have $S \Rightarrow^* A$, $S \Rightarrow^* B$, $A \Rightarrow^* B$ and $B \Rightarrow^* A$.

First, for the set of original non-MODULE-productions, we have

$$\begin{aligned} S &\rightarrow Aa \\ A &\rightarrow a | bc \\ B &\rightarrow bb \end{aligned}$$

We then add the new rules

$$\begin{aligned} S &\rightarrow a | bc | bb \\ A &\rightarrow bb \\ B &\rightarrow a | bc \end{aligned}$$

We finally obtain the equivalent grammar G' with P' consisting of

$$\begin{aligned} S &\rightarrow Aa | a | bc | bb \\ A &\rightarrow a | bc | bb \\ B &\rightarrow bb | a | bc \end{aligned}$$

Notice that B and its associated production become useless.

2 Minimization of Context Free Grammar

- **Theorem:**

Let L be a context-free language that does not contain λ . There exists a context-free grammar that generates L and that does not have any useless productions, λ -productions or MODULE-productions.

Proof:

We need to remove the undesirable productions using the following sequence of steps.

1. Remove λ -productions
2. Remove MODULE-productions
3. Remove useless productions

3. Chomsky Normal Form

- **Definition:**

A context-free grammar is in Chomsky normal form if all productions are of the form

$$A \rightarrow BC$$

or

$$A \rightarrow a$$

where $A, B, C \in V$, and $a \in T$.

Note: that the number of symbols on the right side of productions is strictly limited; not more than two symbols.

- **Example:**

The following grammar is in Chomsky normal form.

$$S \rightarrow AS \mid a$$

$$A \rightarrow SA \mid b$$

On the other hand, the grammar below is not.

$$S \rightarrow AS \mid AAS$$

$$A \rightarrow SA \mid aa$$

- **Theorem:**

Any context-free grammar $G = (V, T, S, P)$ with $\lambda \notin L(G)$ has an equivalent grammar $G' = (V', T', S, P')$ in Chomsky normal form.

Proof:

First we assume (based on previous Theorem) without loss of generality that G has no λ -productions and no MODULE-productions. Then, we show how to construct G' in two steps.

Step 1:

Construct a grammar $G_1 = (V_1, T, S, P_1)$ from G by considering all productions in P of the form

$$A \rightarrow x_1x_2\dots x_n$$

Where each x_i is a symbol either in V or in T .

Note that if $n = 1$, x_1 must be a terminal because there is no MODULE-productions in G . In this case, put the production into P_1 .

If $n \geq 2$, introduce new variables B_a for each $a \in T$. Then, for each production of the form $A \rightarrow x_1x_2\dots x_n$, we shall remove all terminals from productions whose right side has length greater than one

This is done by putting into P_1 a production

$$A \rightarrow C_1C_2\dots C_n$$

Where

$$C_i = x_i \text{ if } x_i \in V$$

And

$$C_i = B_a \text{ if } x_i = a$$

And, for every B_a , we also put into P_1 a production

$$B_a \rightarrow a$$

As a consequence of Theorem 6.1, it can be claimed that

$$L(G_1) = L(G)$$

Step 2:

The length of right side of productions is reduced by means of additional variables wherever necessary. First of all, all productions with a single terminal or two variables ($n = 2$) are put into P' . Then, for any production with $n > 2$, new variables D_1, D_2, \dots are introduced and the following productions are put into P' .

$$A \rightarrow C_1D_1$$

$$D_1 \rightarrow C_2D_2$$

...

$$D_{n-2} \rightarrow C_{n-1}C_n$$

G' is clearly in Chomsky normal form.

• **Example:**

Convert to Chomsky normal form the following grammar G with productions.

$$S \rightarrow ABa$$

$$A \rightarrow aab$$

$$B \rightarrow Ac$$

Solution:

Step 1:

New variables B_a, B_b, B_c are introduced and a new grammar G_1 is obtained.

$$S \rightarrow ABB_a$$

$$A \rightarrow B_aB_aB_b$$

$$B \rightarrow AB_c$$

$$B_a \rightarrow a$$

$$B_b \rightarrow b$$

$$B_c \rightarrow c$$

Step 2:

Additional variables are introduced to reduce the length of the first two productions making them into the normal form, we finally obtain G' .

$$S \rightarrow AD_1$$

$$D_1 \rightarrow BB_a$$

$$A \rightarrow B_aD_2$$

$$D_2 \rightarrow B_aB_b$$

$$B \rightarrow AB_c$$

$$B_a \rightarrow a$$

$$B_b \rightarrow b$$

$$B_c \rightarrow c$$

Greibach normal form

- **Definition:**

A context-free grammar is said to be in Greibach normal form if all productions have the form

$$A \rightarrow ax$$

where $a \in T$ and $x \in V^*$

Note that the restriction here is not on the number of symbols on the right side, but rather on the positions of the terminals and variables.

- **Example:**

The following grammar is not in Greibach normal form.

$$S \rightarrow AB$$

$$A \rightarrow aA \mid bB \mid b$$

$$B \rightarrow b$$

It can, however, be converted to the following equivalent grammar in Greibach normal form.

$$S \rightarrow aAB \mid bBB \mid bB$$

$$A \rightarrow aA \mid bB \mid b$$

$$B \rightarrow b$$

- **Theorem:**

For every context-free grammar G with $\lambda \notin L(G)$, there exists an equivalent grammar G' in Greibach normal form.

Conversion

- Convert from Chomsky to Greibach in two steps:
 1. From Chomsky to intermediate grammar
 - a) Eliminate direct left recursion
 - b) Use $A \rightarrow uBv$ rules transformations to improve references (explained later)

2. From intermediate grammar into Greibach

1.a) Eliminate direct left recursion

Step1:

- Before

$$A \rightarrow A\underline{a} \mid \underline{b}$$

- After

$$A \rightarrow \underline{b}Z \mid \underline{b}$$

$$Z \rightarrow \underline{a}Z \mid \underline{a}$$

- Remove the rule with direct left recursion, and create a new one with recursion on the right

Step2:

- Before

$$A \rightarrow A\underline{a} \mid A\underline{b} \mid \underline{b} \mid \underline{c}$$

- After

$$A \rightarrow \underline{b}Z \mid \underline{c}Z \mid \underline{b} \mid \underline{c}$$

$$Z \rightarrow \underline{a}Z \mid \underline{b}Z \mid \underline{a} \mid \underline{b}$$

- Remove the rules with direct left recursion, and create new ones with recursion on the right

Step3:

- Before

$$A \rightarrow A\underline{B} \mid \underline{B}A \mid \underline{a}$$

$$B \rightarrow \underline{b} \mid \underline{c}$$

- After

$$A \rightarrow \underline{B}A\underline{Z} \mid \underline{a}Z \mid \underline{B}A \mid \underline{a}$$

$$Z \rightarrow \underline{B}Z \mid \underline{B}$$

$$B \rightarrow \underline{b} \mid \underline{c}$$

Transform $A \rightarrow uBv$ rules

- Before

$$A \rightarrow uBb$$

$$B \rightarrow w_1 / w_1 \mid \dots \mid w_n$$

- After

$$\text{Add } A \rightarrow uw_1b / uw_1b \mid \dots \mid uw_nb$$

$$\text{Delete } A \rightarrow uBb$$

Background Information for the Pumping Lemma for Context-Free Languages

- **Definition:** Let $G = (V, T, P, S)$ be a CFL. If every production in P is of the form

$$A \rightarrow BC$$

or

$$A \rightarrow a$$

where A, B and C are all in V and a is in T , then G is in Chomsky Normal Form (CNF).

- **Example:**

$$S \rightarrow AB \mid BA \mid aSb$$

$$A \rightarrow a$$

$$B \rightarrow b$$

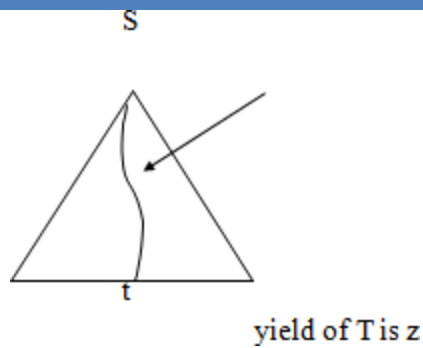
- **Theorem:** Let L be a CFL. Then $L - \{\epsilon\}$ is a CFL.
- **Theorem:** Let L be a CFL not containing $\{\epsilon\}$. Then there exists a CNF grammar G such that $L = L(G)$.
- **Definition:** Let T be a tree. Then the height of T , denoted $h(T)$, is defined as follows:
 - If T consists of a single vertex then $h(T) = 0$
 - If T consists of a root r and subtrees T_1, T_2, \dots, T_k , then $h(T) = \max_i \{h(T_i)\} + 1$
- **Lemma:** Let G be a CFG in CNF. In addition, let w be a string of terminals where $A \Rightarrow^* w$ and w has a derivation tree T . If T has height $h(T) \geq 1$, then $|w| \leq 2^{h(T)-1}$.
- **Proof:** By induction on $h(T)$ (exercise).
- **Corollary:** Let G be a CFG in CNF, and let w be a string in $L(G)$. If $|w| \geq 2^k$, where $k \geq 0$, then any derivation tree for w using G has height at least $k+1$.
- **Proof:** Follows from the lemma.

4. Pumping Lemma for Context-Free Languages

- **Lemma:**
 Let $G = (V, T, P, S)$ be a CFG in CNF, and let $n = 2^{|V|}$. If z is a string in $L(G)$ and $|z| \geq n$, then there exist strings u, v, w, x and y in T^* such that $z = uvwxy$ and:
 - $|vx| \geq 1$ (i.e., $|v| + |x| \geq 1$)
 - $|vwx| \leq n$
 - uv^iwx^iy is in $L(G)$, for all $i \geq 0$
- **Proof:**
 Since $|z| \geq n = 2^k$, where $k = |V|$, it follows from the corollary that any derivation tree for z has height at least $k+1$.

By definition such a tree contains a path of length at least $k+1$.

Consider the longest such path in the tree:



Such a path has:

- Length $\geq k+1$ (i.e., number of edges in the path is $\geq k+1$)
- At least $k+2$ nodes
- 1 terminal

At least $k+1$ non-terminals

- Since there are only k non-terminals in the grammar, and since $k+1$ appear on this long path, it follows that some non-terminal (and perhaps many) appears at least twice on this path.
- Consider the first non-terminal that is repeated, when traversing the path from the leaf to the root.

This path, and the non-terminal A will be used to break up the string z .

- **In addition, (2) also tells us:**

$$S \Rightarrow^* uAy \quad (1)$$

$$\Rightarrow^* uvAxy \quad (2)$$

$$\Rightarrow^* uv^2Ax^2y \quad (2)$$

$$\Rightarrow^* uv^2wx^2y \quad (3)$$

- **More generally:**

$$S \Rightarrow^* uv^iwx^iy \quad \text{for all } i \geq 1$$

- **And also:**

$$S \Rightarrow^* uAy \quad (1)$$

$$\Rightarrow^* uwy \quad (3)$$

- **Hence:**

$$S \Rightarrow^* uv^iwx^iy \quad \text{for all } i \geq 0$$

- **Consider the statement of the Pumping Lemma:**

- *What is n ?*
 $n = 2^k$, where k is the number of non-terminals in the grammar.

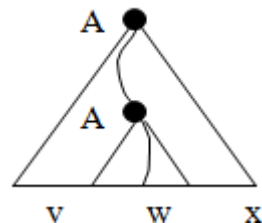
- *Why is $|v| + |x| \geq 1$?*

Since the height of this subtree is ≥ 2 , the first production is $A \rightarrow V_1 V_2$. Since no non-terminal derives the empty string (in CNF), either V_1 or V_2 must derive a non-empty v or x . More specifically, if w is generated by V_1 , then x contains at least one symbol, and if w is generated by V_2 , then v contains at least one symbol.

- *Why is $|vwx| \leq n$?*

Observations:

- The repeated variable was the first repeated variable on the path from the bottom, and therefore (by the pigeon-hole principle) the path from the leaf to the second occurrence of the non-terminal has length at most $k+1$.
- Since the path was the largest in the entire tree, this path is the longest in the subtree rooted at the second occurrence of the non-terminal. Therefore the subtree has height $\leq k+1$. From the lemma, the yield of the subtree has length $\leq 2^k = n$.



CFL Closure Properties

- **Theorem#1:**

The context-free languages are closed under concatenation, union, and Kleene closure.

- **Proof:**

Start with 2 CFL $L(H1)$ and $L(H2)$ generated by $H1 = (N1, T1, R1, s1)$ and $H2 = (N2, T2, R2, s2)$.

Assume that the alphabets and rules are disjoint.

Concatenation:

Formed by $L(H1) \cdot L(H2)$ or a string in $L(H1)$ followed by a string in $L(H2)$ which can be

generated by $L(H3)$ generated by $H3 = (N3, T3, R3, s3)$. $N3 = N1 \cup N2$, $T3 = T1 \cup T2$, $R3 = R1 \cup R2 \cup \{s3 \rightarrow s1s2\}$ where $s3 \rightarrow s1s2$ is a new rule introduced. The new rule generates a string of $L(H1)$ then a string of $L(H2)$. Then $L(H1) \cdot L(H2)$ is context-free.

Union:

Formed by $L(H1) \cup L(H2)$ or a string in $L(H1)$ or a string in $L(H2)$. It is generated by $L(H3)$ generated by $H4 = (N4, T4, R4, s4)$ where $N4 = N1 \cup N2$, $T4 = T1 \cup T2$, and $R4 = R1 \cup R2 \cup \{s4 \rightarrow s1, s4 \rightarrow s2\}$, the new rules added will create a string of $L(H1)$ or $L(H2)$. Then $L(H1) \cup L(H2)$ is context-free.

Kleene:

Formed by $L(H1)^*$ is generated by the grammar $L(H5)$ generated by $H5 = (N1, T1, R5, s1)$ with $R5 = R1 \cup \{s1 \rightarrow e, s1 \rightarrow s1s1\}$. $L(H5)$ includes e , every string in $L(H1)$, and through $i-1$ applications of $s1 \rightarrow s1s1$, every string in $L(H1)^i$. Then $L(H1)^*$ is generated by $H5$ and is context-free.

- **Theorem#2:**

The set of context-free languages is not closed under complementation or intersection.

- **Proof:**

Intersections of two languages $L1 \cap L2$ can be defined in terms of the Complement and Union operations as follows:

$$L1 \cap L2 = \Sigma^* - (\Sigma^* - L1) \cup (\Sigma^* - L2)$$

Therefore if CFL are closed under intersection then it is closed under complement and if closed under complement then it is closed under intersection.

The proof is just showing two context-free languages that their intersection is not a context-free language.

Choose $L1 = \{anbncm \mid m, n \geq 0\}$ is generated by grammar $H1 = \{N1, T1, R1, s1\}$, where

$$\begin{aligned} N1 &= \{s, A, B\} \\ T1 &= \{a, b, c\} \\ R1 &= \{s \rightarrow AB, \\ &A \rightarrow aAb, \\ &A \rightarrow e, \\ &B \rightarrow Bc, \\ &B \rightarrow e\}. \end{aligned}$$

Choose $L2 = \{ambncn \mid m, n$

$$\begin{aligned} N1 &= \{s, A, B\} \\ T1 &= \{a, b, c\} \\ R2 &= \{s \rightarrow AB, \\ &A \rightarrow aA, \\ &A \rightarrow e, \\ &B \rightarrow bBc, \\ &B \rightarrow e\}. \end{aligned}$$

$H2 = \{N2, T2, R2, s2\}$, where

Thus $L1$ and $L2$ are both context-free.

The intersection of the two languages is $L3 = \{anbncn \mid n \text{ already been proven earlier in this paper to be not context-free. Therefore CFL are not closed under intersections, which also means that it is not closed under complementation.}$

5. Pushdown Automata (PDA)

- **Informally:**

- A PDA is an NFA- ϵ with a stack.
- Transitions are modified to accommodate stack operations.

- **Questions:**

- What is a stack?
- How does a stack help?

- A DFA can “remember” only a finite amount of information, whereas a PDA can “remember” an infinite amount of (certain types of) information.

- **Example:**

$$\{0^n 1^n \mid 0 \leq n\}$$

Is *not* regular.

$$\{0^n 1^n \mid 0 \leq n \leq k, \text{ for some fixed } k\}$$

Is regular, for any fixed k .

- **For $k=3$:**

$$L = \{\epsilon, 01, 0011, 000111\}$$

- In a DFA, each state remembers a finite amount of information.
- To get $\{0^n 1^n \mid 0 \leq n\}$ with a DFA would require an infinite number of states using the preceding technique.
- An infinite stack solves the problem for $\{0^n 1^n \mid 0 \leq n\}$ as follows:
 - Read all 0's and place them on a stack
 - Read all 1's and match with the corresponding 0's on the stack
- Only need two states to do this in a PDA
- Similarly for $\{0^n 1^m 0^{n+m} \mid n, m \geq 0\}$

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 <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
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, where

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

- Consider the various parts of δ :

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

- Q on the LHS means that at each step in a computation, a PDA must consider its' current state.
- Γ on the LHS means that at each step in a computation, a PDA must consider the symbol on top of its' stack.
- $\Sigma \cup \{\epsilon\}$ on the LHS means that at each step in a computation, a PDA may or may not consider the current input symbol, i.e., it may have epsilon transitions.
- “Finite subsets” on the RHS means that at each step in a computation, a PDA will have several

options.

—Q on the RHS means that each option specifies a new state.

— Γ^* on the RHS means that each option specifies zero or more stack symbols that will replace the top stack symbol.

• **Two types of PDA transitions #1:**

$$\delta(q, a, z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_m, \gamma_m)\}$$

—Current state is q

—Current input symbol is a

—Symbol currently on top of the stack z

—Move to state p_i from q

—Replace z with γ_i on the stack (leftmost symbol on top)

—Move the input head to the next input symbol

• **Two types of PDA transitions #2:**

$$\delta(q, \epsilon, z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_m, \gamma_m)\}$$

—Current state is q

—Current input symbol is not considered

—Symbol currently on top of the stack z

—Move to state p_i from q

—Replace z with γ_i on the stack (leftmost symbol on top)

—**No input symbol is read**

• **Example:** (balanced parentheses)

$$M = (\{q_1\}, \{“(”, “)”\}, \{L, \#\}, \delta, q_1, \#, \emptyset)$$

δ :

- | | |
|-----|---|
| (1) | $\delta(q_1, (, \#) = \{(q_1, L\#)\}$ |
| (2) | $\delta(q_1,), \#) = \emptyset$ |
| (3) | $\delta(q_1, (, L) = \{(q_1, LL)\}$ |
| (4) | $\delta(q_1,), L) = \{(q_1, \epsilon)\}$ |
| (5) | $\delta(q_1, \epsilon, \#) = \{(q_1, \epsilon)\}$ |
| (6) | $\delta(q_1, \epsilon, L) = \emptyset$ |

• **Goal:** (acceptance)

—Terminate in a non-null state

—Read the entire input string

—Terminate with an empty stack

• Informally, a string is accepted if there exists a computation that uses up all the input and leaves the stack empty.

• Example Computation:

<u>Current Input</u>	<u>Stack</u>	<u>Transition</u>
((()	#	
()	L#	(1) - Could have applied rule
))	LL#	(3) (5), but it would have
)	L#	(4) done no good
ϵ	#	(4)
ϵ	-	(5)

• **Example PDA #1:** For the language $\{x \mid x = wcw^r \text{ and } w \text{ in } \{0,1\}^*\}$

$M = (\{q_1, q_2\}, \{0, 1, c\}, \{R, B, G\}, \delta, q_1, R, \emptyset)$

δ :

(1) $\delta(q_1, 0, R) = \{(q_1, BR)\}$	(9) $\delta(q_1, 1, R) = \{(q_1, GR)\}$
(2) $\delta(q_1, 0, B) = \{(q_1, BB)\}$	(10) $\delta(q_1, 1, B) = \{(q_1, GB)\}$
(3) $\delta(q_1, 0, G) = \{(q_1, BG)\}$	(11) $\delta(q_1, 1, G) = \{(q_1, GG)\}$
(4) $\delta(q_1, c, R) = \{(q_2, R)\}$	
(5) $\delta(q_1, c, B) = \{(q_2, B)\}$	
(6) $\delta(q_1, c, G) = \{(q_2, G)\}$	
(7) $\delta(q_2, 0, B) = \{(q_2, \epsilon)\}$	(12) $\delta(q_2, 1, G) = \{(q_2, \epsilon)\}$
(8) $\delta(q_2, \epsilon, R) = \{(q_2, \epsilon)\}$	

• **Notes:**

—Only rule #8 is non-deterministic.

—Rule #8 is used to pop the final stack symbol off at the end of a computation.

• **Example Computation:**

- | | | | |
|-----|--|------|---|
| (1) | $\delta(q_1, 0, R) = \{(q_1, BR)\}$ | (9) | $\delta(q_1, 1, R) = \{(q_1, GR)\}$ |
| (2) | $\delta(q_1, 0, B) = \{(q_1, BB)\}$ | (10) | $\delta(q_1, 1, B) = \{(q_1, GB)\}$ |
| (3) | $\delta(q_1, 0, G) = \{(q_1, BG)\}$ | (11) | $\delta(q_1, 1, G) = \{(q_1, GG)\}$ |
| (4) | $\delta(q_1, c, R) = \{(q_2, R)\}$ | | |
| (5) | $\delta(q_1, c, B) = \{(q_2, B)\}$ | | |
| (6) | $\delta(q_1, c, G) = \{(q_2, G)\}$ | | |
| (7) | $\delta(q_2, 0, B) = \{(q_2, \epsilon)\}$ | (12) | $\delta(q_2, 1, G) = \{(q_2, \epsilon)\}$ |
| (8) | $\delta(q_2, \epsilon, R) = \{(q_2, \epsilon)\}$ | | |

<u>State</u>	<u>Input</u>	<u>Stack</u>	<u>Rule Applied</u>	<u>Rules Applicable</u>
q ₁	01c10	R	-	(1)
q ₁	1c10	BR	(1)	(10)
q ₁	c10	GBR	(10)	(6)
q ₂	10	GBR	(6)	(12)
q ₂	0	BR	(12)	(7)
q ₂	ε	R	(7)	(8)
q ₂	ε	ε	(8)	-

• **Example Computation:**

- | | | | |
|-----|--|------|---|
| (1) | $\delta(q_1, 0, R) = \{(q_1, BR)\}$ | (9) | $\delta(q_1, 1, R) = \{(q_1, GR)\}$ |
| (2) | $\delta(q_1, 0, B) = \{(q_1, BB)\}$ | (10) | $\delta(q_1, 1, B) = \{(q_1, GB)\}$ |
| (3) | $\delta(q_1, 0, G) = \{(q_1, BG)\}$ | (11) | $\delta(q_1, 1, G) = \{(q_1, GG)\}$ |
| (4) | $\delta(q_1, c, R) = \{(q_2, R)\}$ | | |
| (5) | $\delta(q_1, c, B) = \{(q_2, B)\}$ | | |
| (6) | $\delta(q_1, c, G) = \{(q_2, G)\}$ | | |
| (7) | $\delta(q_2, 0, B) = \{(q_2, \epsilon)\}$ | (12) | $\delta(q_2, 1, G) = \{(q_2, \epsilon)\}$ |
| (8) | $\delta(q_2, \epsilon, R) = \{(q_2, \epsilon)\}$ | | |

<u>State</u>	<u>Input</u>	<u>Stack</u>	<u>Rule Applied</u>
q ₁	1c1	R	
q ₁	c1	GR	(9)
q ₂	1	GR	(6)
q ₂	ε	R	(12)
q ₂	ε	ε	(8)

• **Definition:** $\mid\!\!\!-\ast$ is the reflexive and transitive closure of $\mid\!\!\!-$.

— $I \mid\!\!\!-\ast I$ for each instantaneous description I

— If $I \mid\!\!\!- J$ and $J \mid\!\!\!-\ast K$ then $I \mid\!\!\!-\ast K$

• Intuitively, if I and J are instantaneous descriptions, then $I \xrightarrow{*} J$ means that J follows from I by zero or more transitions.

• **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ be a PDA. The *language accepted by empty stack*, denoted $L_E(M)$, is the set

$$\{w \mid (q_0, w, z_0) \xrightarrow{*} (p, \varepsilon, \varepsilon) \text{ for some } p \in Q\}$$

• **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ be a PDA. The *language accepted by final state*, denoted $L_F(M)$, is the set

$$\{w \mid (q_0, w, z_0) \xrightarrow{*} (p, \varepsilon, \gamma) \text{ for some } p \in F \text{ and } \gamma \in \Gamma^*\}$$

• **Definition:** Let $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ be a PDA. The *language accepted by empty stack and final state*, denoted $L(M)$, is the set

$$\{w \mid (q_0, w, z_0) \xrightarrow{*} (p, \varepsilon, \varepsilon) \text{ for some } p \in F\}$$

• **Lemma 1:** Let $L = L_E(M_1)$ for some PDA M_1 . Then there exists a PDA M_2 such that $L = L_F(M_2)$.

• **Lemma 2:** Let $L = L_F(M_1)$ for some PDA M_1 . Then there exists a PDA M_2 such that $L = L_E(M_2)$.

• **Theorem:** Let L be a language. Then there exists a PDA M_1 such that $L = L_F(M_1)$ if and only if there exists a PDA M_2 such that $L = L_E(M_2)$.

• **Corollary:** The PDAs that accept by empty stack and the PDAs that accept by final state define the same class of languages.

• **Note:** Similar lemmas and theorems could be stated for PDAs that accept by both final state and empty stack.

Greibach Normal Form (GNF)

• **Definition:** Let $G = (V, T, P, S)$ be a CFL. If every production in P is of the form

$$A \rightarrow a\alpha$$

Where A is in V, a is in T, and α is in V^* , then G is said to be in Greibach Normal Form (GNF).

• **Example:**

$$\begin{aligned} S &\rightarrow aAB \mid bB \\ A &\rightarrow aA \mid a \end{aligned}$$

$$B \rightarrow bB \mid c$$

• **Theorem:** Let L be a CFL. Then $L - \{\epsilon\}$ is a CFL.

• **Theorem:** Let L be a CFL not containing $\{\epsilon\}$. Then there exists a GNF grammar G such that $L = L(G)$.

• **Lemma 1:** Let L be a CFL. Then there exists a PDA M such that $L = L_E(M)$.

• **Proof:** Assume without loss of generality that ϵ is not in L . The construction can be modified to include ϵ later.

Let $G = (V, T, P, S)$ be a CFG, and assume without loss of generality that G is in GNF.
Construct $M = (Q, \Sigma, \Gamma, \delta, q, z, \emptyset)$ where:

$$\begin{aligned} Q &= \{q\} \\ \Sigma &= T \\ \Gamma &= V \\ z &= S \end{aligned}$$

δ : for all a in Σ and A in Γ , $\delta(q, a, A)$ contains (q, γ) if $A \rightarrow a\gamma$ is in P or rather:
 $\delta(q, a, A) = \{(q, \gamma) \mid A \rightarrow a\gamma \text{ is in } P \text{ and } \gamma \text{ is in } \Gamma^*\}$, for all a in Σ and A in Γ

• For a given string x in Σ^* , M will attempt to simulate a leftmost derivation of x with G .

• **Example #1:** Consider the following CFG in GNF.

$$\begin{array}{ll} S \rightarrow aS & G \text{ is in GNF} \\ S \rightarrow a & L(G) = a^+ \end{array}$$

Construct M as:

$$\begin{aligned} Q &= \{q\} \\ \Sigma &= T = \{a\} \\ \Gamma &= V = \{S\} \\ z &= S \end{aligned}$$

$$\begin{aligned} \delta(q, a, S) &= \{(q, S), (q, \epsilon)\} \\ \delta(q, \epsilon, S) &= \emptyset \end{aligned}$$

• **Example #2:** Consider the following CFG in GNF.

$$\begin{array}{ll} (1) & S \rightarrow aA \\ (2) & S \rightarrow aB \\ (3) & A \rightarrow aA \\ (4) & A \rightarrow aB \end{array} \quad \begin{array}{l} G \text{ is in GNF} \\ L(G) = a^+b^+ \end{array}$$

- (5) $B \rightarrow bB$
- (6) $B \rightarrow b$

Construct M as:

$$\begin{aligned} Q &= \{q\} \\ \Sigma = T &= \{a, b\} \\ \Gamma = V &= \{S, A, B\} \\ z &= S \end{aligned}$$

- (1) $\delta(q, a, S) = \{(q, A), (q, B)\}$ From productions #1 and 2, $S \rightarrow aA$, $S \rightarrow aB$
 - (2) $\delta(q, a, A) = \{(q, A), (q, B)\}$ From productions #3 and 4, $A \rightarrow aA$, $A \rightarrow aB$
 - (3) $\delta(q, a, B) = \emptyset$
 - (4) $\delta(q, b, S) = \emptyset$
 - (5) $\delta(q, b, A) = \emptyset$
 - (6) $\delta(q, b, B) = \{(q, B), (q, \epsilon)\}$ From productions #5 and 6, $B \rightarrow bB$, $B \rightarrow b$
 - (7) $\delta(q, \epsilon, S) = \emptyset$
 - (8) $\delta(q, \epsilon, A) = \emptyset$
 - (9) $\delta(q, \epsilon, B) = \emptyset$
- Recall $\delta: Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow$ finite subsets of $Q \times \Gamma^*$

• For a string w in $L(G)$ the PDA M will simulate a leftmost derivation of w .

– If w is in $L(G)$ then $(q, w, z_0) \vdash^* (q, \epsilon, \epsilon)$

– If $(q, w, z_0) \vdash^* (q, \epsilon, \epsilon)$ then w is in $L(G)$

• Consider generating a string using G . Since G is in GNF, each sentential form in a *leftmost* derivation has form:

• And each step in the derivation (i.e., each application of a production) adds a terminal and some non-terminals.

$$A_1 \rightarrow t_{i+1} \alpha$$

$$\Rightarrow t_1 t_2 \dots t_i t_{i+1} \alpha A_1 A_2 \dots A_m$$

• Each transition of the PDA simulates one derivation step. Thus, the i^{th} step of the PDAs' computation corresponds to the i^{th} step in a corresponding leftmost derivation.

• After the i^{th} step of the computation of the PDA, $t_1 t_2 \dots t_{i+1}$ are the symbols that have already

been read by the PDA and $\alpha A_1 A_2 \dots A_m$ are the stack contents.

- For each leftmost derivation of a string generated by the grammar, there is an equivalent accepting computation of that string by the PDA.
- Each sentential form in the leftmost derivation corresponds to an instantaneous description in the PDA's corresponding computation.
- For example, the PDA instantaneous description corresponding to the sentential form:

$$\Rightarrow t_1 t_2 \dots t_i A_1 A_2 \dots A_m$$

would be: $(q, t_{i+1} t_{i+2} \dots t_n, A_1 A_2 \dots A_m)$

- **Example:** Using the grammar from example #2:

$$\begin{aligned} S &\Rightarrow aA & (1) \\ &\Rightarrow aaA & (3) \\ &\Rightarrow aaaA & (3) \\ &\Rightarrow aaaaB & (4) \\ &\Rightarrow aaaabB & (5) \\ &\Rightarrow aaaabb & (6) \end{aligned}$$

- The corresponding computation of the PDA:

$$\begin{aligned} \bullet (q, aaaabb, S) & \quad \vdash (q, aaabb, A) & (1)/1 \\ & \quad \vdash (q, aabb, A) & (2)/1 \\ & \quad \vdash (q, abb, A) & (2)/1 \\ & \quad \vdash (q, bb, B) & (2)/2 \\ & \quad \vdash (q, b, B) & (6)/1 \\ & \quad \vdash (q, \varepsilon, \varepsilon) & (6)/2 \end{aligned}$$

–String is read

–Stack is emptied

–Therefore the string is accepted by the PDA

- **Example #3:** Consider the following CFG in GNF.

$$\begin{aligned} (1) \quad & S \rightarrow aABC \\ (2) \quad & A \rightarrow a \\ (3) \quad & B \rightarrow b \\ (4) \quad & C \rightarrow cAB \\ (5) \quad & C \rightarrow cC \end{aligned} \quad \text{G is in GNF}$$

Construct M as:

$$Q = \{q\}$$

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

$$\Gamma = V = \{S, A, B, C\}$$

$$z = S$$

(1)	$\delta(q, a, S) = \{(q, ABC)\}$	$S \rightarrow aABC$	(9)	$\delta(q, c, S) = \emptyset$
(2)	$\delta(q, a, A) = \{(q, \varepsilon)\}$	$A \rightarrow a$	(10)	$\delta(q, c, A) = \emptyset$
(3)	$\delta(q, a, B) = \emptyset$		(11)	$\delta(q, c, B) = \emptyset$
(4)	$\delta(q, a, C) = \emptyset$	$C \rightarrow cAB cC$	(12)	$\delta(q, c, C) = \{(q,$
		$AB), (q, C)\}$	(13)	$\delta(q, \varepsilon, S) = \emptyset$
(5)	$\delta(q, b, S) = \emptyset$		(14)	$\delta(q, \varepsilon, A) = \emptyset$
(6)	$\delta(q, b, A) = \emptyset$		(15)	$\delta(q, \varepsilon, B) = \emptyset$
(7)	$\delta(q, b, B) = \{(q, \varepsilon)\}$	$B \rightarrow b$	(16)	$\delta(q, \varepsilon, C) = \emptyset$
(8)	$\delta(q, b, C) = \emptyset$			

• **Notes:**

- Recall that the grammar G was required to be in GNF before the construction could be applied.
- As a result, it was assumed at the start that ε was not in the context-free language L.

• **Suppose ε is in L:**

1) First, let $L' = L - \{\varepsilon\}$

Fact: If L is a CFL, then $L' = L - \{\varepsilon\}$ is a CFL.

By an earlier theorem, there is GNF grammar G such that $L' = L(G)$.

2) Construct a PDA M such that $L' = L_E(M)$

How do we modify M to accept ε ?

Add $\delta(q, \varepsilon, S) = \{(q, \varepsilon)\}$? No!

• **Counter Example:**

Consider $L = \{\varepsilon, b, ab, aab, aaab, \dots\}$

Then $L' = \{b, ab, aab, aaab, \dots\}$

• **The GNF CFG for L' :**

- (1) $S \rightarrow aS$
- (2) $S \rightarrow b$

• **The PDA M Accepting L' :**

$$\begin{aligned} Q &= \{q\} \\ \Sigma = T &= \{a, b\} \\ \Gamma = V &= \{S\} \\ z &= S \end{aligned}$$

$$\begin{aligned} \delta(q, a, S) &= \{(q, S)\} \\ \delta(q, b, S) &= \{(q, \varepsilon)\} \\ \delta(q, \varepsilon, S) &= \emptyset \end{aligned}$$

• If $\delta(q, \varepsilon, S) = \{(q, \varepsilon)\}$ is added then:

$$L(M) = \{\varepsilon, a, aa, aaa, \dots, b, ab, aab, aaab, \dots\}$$

3) Instead, add a new *start* state q' with transitions:

$$\delta(q', \varepsilon, S) = \{(q', \varepsilon), (q, S)\}$$

• **Lemma 1:** Let L be a CFL. Then there exists a PDA M such that $L = L_E(M)$.

• **Lemma 2:** Let M be a PDA. Then there exists a CFG grammar G such that $L_E(M) = L(G)$.

• **Theorem:** Let L be a language. Then there exists a CFG G such that $L = L(G)$ iff there exists a PDA M such that $L = L_E(M)$.

• **Corollary:** The PDAs define the CFLs.

Equivalence of CFG to PDAs

- **Example:** Consider the grammar for arithmetic expressions we introduced earlier. It is reproduced below for convenience. $G = (\{E, T, F\}, \{n, v, +, *, (,)\}, P, E)$, where

E = {	1:	E	→	E	+	T,
	2:	E	→			T,
	3:	T	→	T*		F,
	4:	T	→			F,
	5:	F	→			n,
	6:	F	→			v,
	7:	F	→	(E),
						}

Suppose the input to our parser is the expression, $n^*(v+n*v)$. Since G is unambiguous this expression has only one leftmost derivation, $p = 2345712463456$. We describe the behavior of the PDA in general, and then step through its moves using this derivation to guide the computation.

- **PDA Simulator:**

- Step 1: Initialize the stack with the start symbol (E in this case). The start symbol will serve as the bottom of stack marker (Z_0).
- Step 2: Ignoring the input, check the top symbol of the stack.
 - Case (a) Top of stack is a nonterminal, “X”: non-deterministically decide which X-rule to use as the next step of the derivation. After selecting a rule, replace X in the stack with the rightpart of that rule. If the stack is non-empty, repeat step 2. Otherwise, halt (input may or may not be empty.)
 - Case(b) Top of stack is a terminal, “a”: Read the next input. If the input matches a, then pop the stack and repeat step 2. Otherwise, halt (without popping “a” from the stack.)
- This parsing algorithm by showing the sequence of configurations the parser would assume in an accepting computation for the input, $n^*(v+n*v)$. Assume “ q_0 ” is the one and only state of this PDA.

- p (leftmost derivation in G) = 2345712463456

(q_0 , $n^*(v+n*v)$, E)

$2 \Rightarrow_M (q_0, n^*(v+n*v), T)$

$3 \Rightarrow_M (q_0, n^*(v+n*v), T^*F)$

$4 \Rightarrow_M (q_0, n^*(v+n*v), F^*F)$

$5 \Rightarrow M (q_0, n^*(v+n*v), n^*F)$	$read \Rightarrow M (q_0, *(v+n*v), *F)$
	$read \Rightarrow M (q_0, (v+n*v), F)$
$7 \Rightarrow M (q_0, (v+n*v), (E))$	$read \Rightarrow M (q_0, v+n*v, E))$
$1 \Rightarrow M (q_0, v+n*v, E+T))$	
$2 \Rightarrow M (q_0, v+n*v, T+T))$	
$4 \Rightarrow M (q_0, v+n*v, F+T))$	
$6 \Rightarrow M (q_0, v+n*v, v+T))$	$read \Rightarrow M (q_0, +n*v, +T))$
	$read \Rightarrow M (q_0, n*v, T))$
$3 \Rightarrow M (q_0, n*v, T*F))$	
$4 \Rightarrow M (q_0, n*v, F*F))$	
$5 \Rightarrow M (q_0, n*v, n^*F))$	$read \Rightarrow M (q_0, *v, *F))$
	$read \Rightarrow M (q_0, v, F))$
$6 \Rightarrow M (q_0, v, v))$	$read \Rightarrow M (q_0,),))$
	$read \Rightarrow M (q_0, l, l) \text{ accept!}$

Deterministic PDAs and DCFLs

- **Definition:** A *Deterministic Pushdown Automaton* (DPDA) is a 7-tuple,

$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, A),$

where

$Q =$ finite set of states,

$\Sigma =$ input alphabet,

$\Gamma =$ stack alphabet,

$q_0 \in Q =$ the initial state,

$Z_0 \in \Gamma =$ bottom of stack marker (or initial stack symbol), and

$\delta: Q \times (\Sigma \cup \{L\}) \times \Gamma \rightarrow Q \times \Gamma^* =$ the transition function (not necessarily total).

Specifically,

[1] if $d(q, a, Z)$ is defined for some $a \in \Sigma$ and $Z \in \Gamma$, then $d(q, L, Z) = \Phi$ and $|d(q, a, Z)| = 1$.

[2] Conversely, if $d(q, L, Z) \neq \Phi$, for some Z , then $d(q, a, Z) = \Phi$, for all $a \in \Sigma$, and $|d(q, L, Z)| = 1$.

- **NOTE:** DPDAs can accept their input either by final state or by empty stack – just as for the non-deterministic model. We therefore define D_{stk} and D_{ste} , respectively, as the corresponding families of Deterministic Context-free Languages accepted by a DPDA by empty stack and final state.

MODULE IV:

- 1. Turing Machines**
- 2. The Halting Problem**
- 3. The Universal language**
- 4. A Church- Turing thesis**
- 5. Linear Bounded Automata.**

1. Turing Machines (TM)

- **Generalize the class of CFLs:**
- Recursively enumerable languages are also known as *type 0* languages.
- Context-sensitive languages are also known as *type 1* languages.
- Context-free languages are also known as *type 2* languages.
- Regular languages are also known as *type 3* languages.
- 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.

1.1 Deterministic Turing Machine (DTM)

- Two-way, infinite tape, broken into cells, each containing one symbol.
- Two-way, read/write tape head.
- Finite control, i.e., a program, containing the position of the read head, current symbol being scanned, and the current state.
- 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.
- 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.

1.2 Formal Definition of a DTM

- A DTM is a seven-tuple:

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

Q	A <u>finite</u> set of states
Γ	A <u>finite</u> tape alphabet
B	A distinguished blank symbol, which is in Γ
Σ	A <u>finite</u> input alphabet, which is a subset of $\Gamma - \{B\}$
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 <i>mapping</i> 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:** $\{0^n 1^n \mid n \geq 1\}$

	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	-	-	-	-	-

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

	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	-	-	-	-	-

- The TM basically matches up 0's and 1's
- q₁ is the “scan right” state
- q₂ is the “scan left” state
- q₄ is the final state

- **Example #2:** $\{w \mid w \text{ is in } \{0,1\}^* \text{ and } w \text{ ends with a } 0\}$

0
00
10
10110
Not ϵ

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

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

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

$F = \{q_2\}$

	0	1	B
q ₀	(q ₀ , 0, R)	(q ₀ , 1, R)	(q ₁ , B, L)
q ₁	(q ₂ , 0, R)	-	-
q ₂	-	-	-

- q₀ is the “scan right” state
- q₁ is the verify 0 state

- **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

$$q_0 w \vdash^* \alpha_1 p \alpha_2$$

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

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

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 an TM need have any exiting 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 there exists 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 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 a non-final (non-accepting) state.
- 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.
- **Observation:** Let L be an r.e. language. Then there is an infinite list M_0, M_1, \dots of TMs such that $L = L(M_i)$.
- **Question:** Let L be a recursive language, and M_0, M_1, \dots a list of all TMs such that $L = L(M_i)$, and choose any $i \geq 0$. Does M_i always halt?

Answer: Maybe, maybe not, but *at least one in the list does*.

- **Question:** Let L be a recursive enumerable language, and M_0, M_1, \dots a list of all TMs such that $L = L(M_i)$, and choose any $i \geq 0$. Does M_i always halt?

Answer: Maybe, maybe not. Depending on L , none might halt or some may halt.

- If L is also recursive then L is recursively enumerable.

Question: Let L be a recursive enumerable language that is not recursive (L is in r.e. – r), and M_0, M_1, \dots a list of all TMs such that $L = L(M_i)$, and choose any $i \geq 0$. Does M_i always halt?

Answer: No! If it did, then L would not be in r.e. – r, it would be recursive.

- **Let M be a TM.**

- Question: Is $L(M)$ r.e.?
Answer: Yes! By definition it is!
- Question: Is $L(M)$ recursive?
Answer: Don't know, we don't have enough information.
- Question: Is $L(M)$ in r.e. – r?
Answer: Don't know, we don't have enough information.

- **Let M be a TM that halts on all inputs:**

- Question: Is $L(M)$ recursively enumerable?
Answer: Yes! By definition it is!
- Question: Is $L(M)$ recursive?
Answer: Yes! By definition it is!
- Question: Is $L(M)$ in r.e. – r?
Answer: No! It can't be. Since M always halts, $L(M)$ is recursive.

- **Let M be a TM.**

- As noted previously, $L(M)$ is recursively enumerable, but may or may not be recursive.
- Question: Suppose that $L(M)$ is recursive. Does that mean that M always halts?
Answer: Not necessarily. However, some TM M' must exist such that $L(M') = L(M)$ and M' always halts.
- Question: Suppose that $L(M)$ is in r.e. – r. Does M always halt?
Answer: No! If it did then $L(M)$ would be recursive and therefore not in r.e. – r.

- **Let M be a TM, and suppose that M loops forever on some string x .**

- Question: Is $L(M)$ recursively enumerable?
Answer: Yes! By definition it is.
- Question: Is $L(M)$ recursive?
Answer: Don't know. Although M doesn't always halt, some other TM M' may exist such that $L(M') = L(M)$ and M' always halts.
- Question: Is $L(M)$ in r.e. – r?
Answer: Don't know.

Closure Properties for Recursive and Recursively Enumerable Languages

- **TMs Model General Purpose Computers:**
 - If a TM can do it, so can a GP computer
 - If a GP computer can do it, then so can a TM

If you want to know if a TM can do X , then some equivalent question are:

- *Can a general purpose computer do X ?*
- *Can a C/C++/Java/etc. program be written to do X ?*

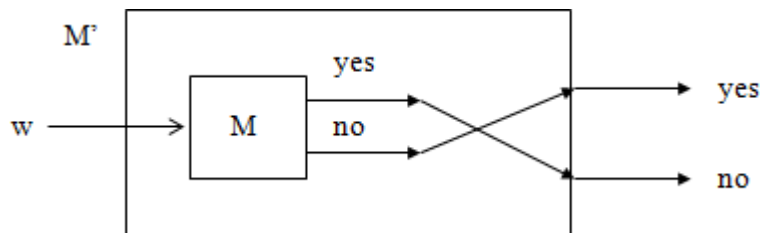
For example, is a language L recursive?

- *Can a C/C++/Java/etc. program be written that always halts and accepts L ?*

- **TM Block Diagrams:**
 - If L is a recursive language, then a TM M that accepts L and always halts can be pictorially represented by a “chip” that has one input and two outputs.
 - If L is a recursively enumerable language, then a TM M that accepts L can be pictorially represented by a “chip” that has one output.
 - Conceivably, M could be provided with an output for “no,” but this output cannot be counted on. Consequently, we simply ignore it.
- **Theorem:** The recursive languages are closed with respect to complementation, i.e., if L is a recursive language, then so is

Proof: Let M be a TM such that $L = L(M)$ and M always halts. Construct TM M' as

follows:



- **Note That:**
 - M' accepts iff M does not
 - M' always halts since M always halts

From this it follows that the complement of L is recursive. •

- **Theorem:** The recursive languages are closed with respect to union, i.e., if L_1 and L_2 are recursive languages, then so is

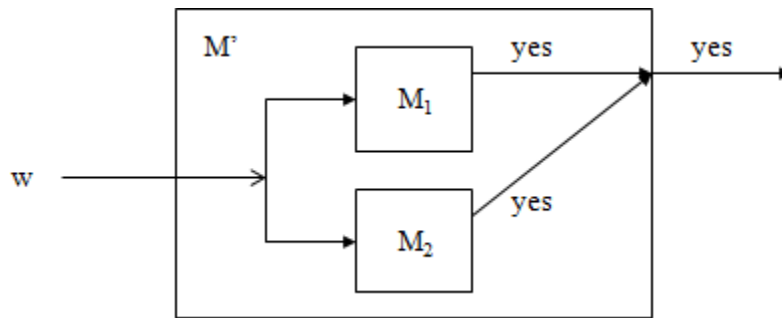
Proof: Let M_1 and M_2 be TMs such that $L_1 = L(M_1)$ and $L_2 = L(M_2)$ and M_1 and M_2 always halts. Construct TM M' as follows:

- **Note That:**
 - $L(M') = L(M_1) \cup L(M_2)$
 - $L(M')$ is a subset of $L(M_1) \cup L(M_2)$
 - $L(M_1) \cup L(M_2)$ is a subset of $L(M')$
 - M' always halts since M_1 and M_2 always halt

It follows from this that $L_3 = L_1 \cup L_2$ is recursive.

- **Theorem:** The recursive enumerable languages are closed with respect to union, i.e., if L_1 and L_2 are recursively enumerable languages, then so is $L_3 = L_1 \cup L_2$

Proof: Let M_1 and M_2 be TMs such that $L_1 = L(M_1)$ and $L_2 = L(M_2)$. Construct M' as follows:



- **Note That:**
 - $L(M') = L(M_1) \cup L(M_2)$
 - $L(M')$ is a subset of $L(M_1) \cup L(M_2)$
 - $L(M_1) \cup L(M_2)$ is a subset of $L(M')$
 - M' halts and accepts iff M_1 or M_2 halts and accepts

It follows from this that is recursively enumerable.

2. The Halting Problem – Background

- **Definition:** A decision problem 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 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
(un)Decidable	-	(non)Recursive

Statement of the Halting Problem

- **Practical Form: (P1)**
Input: Program P and input I.
Question: Does P terminate on input I?
- **Theoretical Form: (P2)**
Input: Turing machine M with input alphabet Σ and string w in Σ^* .
Question: Does M halt on w?
- **A Related Problem We Will Consider First: (P3)**
Input: Turing machine M with input alphabet Σ and one final state, and string w in Σ^* .
Question: Is w in L(M)?
- **Analogy:**
Input: DFA M with input alphabet Σ and string w in Σ^* .
Question: Is w in L(M)?

Is this problem decidable? Yes!

- **Over-All Approach:**
 - We will show that a language L_d is not recursively enumerable
 - From this it will follow that L_d is not recursive
 - Using this we will show that a language L_u is not recursive
 - From this it will follow that the halting problem is undecidable.

3. The Universal Language

- Define the language L_u as follows:
$$L_u = \{x \mid x \text{ is in } \{0, 1\}^* \text{ and } x = \langle M, w \rangle \text{ where } M \text{ is a TM encoding and } w \text{ is in } L(M)\}$$
- Let x be in $\{0, 1\}^*$. Then either:
 1. x doesn't have a TM prefix, in which case x is **not** in L_u
 2. x has a TM prefix, i.e., $x = \langle M, w \rangle$ and either:
 - a) w is not in L(M), in which case x is **not** in L_u
 - b) w is in L(M), in which case x is in L_u

- **Compare P3 and L_u :**

(P3):

Input: Turing machine M with input alphabet Σ and one final state, and string w in Σ^* .

- **Notes:**

- L_u is P3 expressed as a language
- Asking if L_u is recursive is the same as asking if P3 is decidable.
- We will show that L_u is not recursive, and from this it will follow that P3 is undecidable.
- From this we can further show that the halting problem is undecidable.
- Note that L_u is recursive if M is a DFA.

4. 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.
- If something can be “computed” it can be computed by a Turing machine.
- Note that this is called a *Thesis*, not a theorem.
- It can’t be proved, because the term “can be computed” is too vague.
- But it is universally accepted as a true statement.
- Given the *Church-Turing Thesis*:
 - What does this say about "computability"?
 - Are there things even a Turing machine can't do?
 - If there are, then there are things that simply can't be "computed."
 - Not with a Turing machine

- Not with your laptop
 - Not with a supercomputer
 - There ARE things that a Turing machine can't do!!!
- The ***Church-Turing Thesis***:
 - In other words, there is no problem for which we can describe an algorithm that can't be done by a Turing machine.

The Universal Turing machine

- If Tm's are so damned powerful, can't we build one that simulates the behavior of any Tm on any tape that it is given?
- Yes. This machine is called the ***Universal Turing machine***.
- How would we build a Universal Turing machine?
 - We place an encoding of any Turing machine on the input tape of the Universal Tm.
 - The tape consists entirely of zeros and ones (and, of course, blanks)
 - Any Tm is represented by zeros and ones, using unary notation for elements and zeros as separators.
- Every Tm instruction consists of four parts, each represented as a series of **1**'s and separated by **0**'s.
- Instructions are separated by **00**.
- We use unary notation to represent components of an instruction, with
 - $0 = 1$,
 - $1 = 11$,
 - $2 = 111$,

➤ $3 = 1111$,

➤ $n = 111...111$ ($n+1$ 1's).

- We encode q_n as $\underline{n+1}$ 1's
- We encode symbol a_n as $\underline{n+1}$ 1's
- We encode move left as 1, and move right as 11

1111011101111101110100101101101101100

q_3, a_2, q_4, a_2, L q_0, a_1, q_1, a_1, R

- Any Turing machine can be encoded as a unique long string of zeros and ones, beginning with a 1.
- Let T_n be the Turing machine whose encoding is the number n .

5. Linear Bounded Automata

- A Turing machine that has the length of its tape limited to the length of the input string is called a linear-bounded automaton (LBA).
- A linear bounded automaton is a 7-tuple *nondeterministic* Turing machine $M = (Q, S, G, d, q_0, q_{\text{accept}}, q_{\text{reject}})$ except that:
 - a. There are two extra tape symbols $<$ and $>$, which are not elements of G .
 - b. The TM begins in the configuration $(q_0 \leq x >)$, with its tape head scanning the symbol $<$ in cell 0. The $>$ symbol is in the cell immediately to the right of the input string x .
 - c. The TM cannot replace $<$ or $>$ with anything else, nor move the tape head left of $<$ or right of $>$.

Context-Sensitivity

- *Context-sensitive production* any production $\alpha \rightarrow \beta$ satisfying $|\alpha| \leq |\beta|$.
- *Context-sensitive grammar* any generative grammar $G = \langle \Sigma, \Delta, \Pi, \Gamma \rangle$ such that every production in Π context-sensitive.
- No empty productions.

Context-Sensitive Language

- Language L *context-sensitive* if there exists context-sensitive grammar G such that either $L = L(G)$ or $L = L(G) \cup \{ \}$.

- **Example:**

The language $L = \{a^n b^n c^n : n \geq 1\}$ is a C.S.L. the grammar is

$$S \rightarrow abc / aAbc,$$

$$Ab \rightarrow bA,$$

$$AC \rightarrow Bbcc,$$

$$bB \rightarrow Bb,$$

$$aB \rightarrow aa / aaA$$

The derivation tree of $a^3 b^3 c^3$ is looking to be as following

$$S \Rightarrow aAbc$$

$$\Rightarrow abAc$$

$$\Rightarrow abBbcc$$

$$\Rightarrow aBbbcc \quad \Rightarrow aaAbbcc$$

$$\Rightarrow aabAbcc$$

$$\Rightarrow aabbAcc \quad \Rightarrow aabbBbcc$$

$$\Rightarrow aabBbbccc$$

$$\Rightarrow aaBbbbccc$$

$$\Rightarrow aaabbbccc$$

CSG = LBA

- A language is accepted by an LBA iff it is generated by a CSG.
- Just like equivalence between CFG and PDA
- Given an $x \in \text{CSG } G$, you can intuitively see that an LBA can start with S , and nondeterministically choose all derivations from S and see if they are equal to the input string x . Because CSG's are non-contracting, the LBA only needs to generate derivations of length $\leq |x|$. This is because if it generates a derivation longer than $|x|$, it will never be able to shrink to the size of $|x|$.

MODULE V

1. Chomsky Hierarchy Languages
2. Turing Reducibility
3. The Class P

1. Chomsky Hierarchy of Languages

- A containment hierarchy (strictly nested sets) of classes of formal grammars

The Hierarchy

<u>Class</u>	<u>Grammars</u>	<u>Languages</u>	<u>Automaton</u>
Type-0 Unrestricted		Recursively enumerable (Turing-recognizable)	Turing machine
	none	Recursive (Turing-decidable)	Decider
Type-1 Context-sensitive		Context-sensitive	Linear-bounded
Type-2	Context-free	Context-free	Pushdown
Type-3	Regular	Regular	Finite

Type 0 Unrestricted:

Languages defined by Type-0 grammars are accepted by Turing machines .

Rules are of the form: $\alpha \rightarrow \beta$, where α and β are arbitrary strings over a vocabulary V and $\alpha \neq \varepsilon$

Type 1 Context-sensitive:

Languages defined by Type-1 grammars are accepted by linear-bounded automata.

Syntax of some natural languages (Germanic)

Rules are of the form:

$$\alpha A \beta \rightarrow \alpha B \beta$$

$$S \rightarrow \varepsilon$$

where

Type 2 Context-free:

Languages defined by Type-2 grammars are accepted by push-down automata.

Natural language is almost entirely definable by type-2 tree structures

Rules are of the form:

$$A \rightarrow \alpha$$

Where

$$A \in N$$

$$\alpha \in (N \cup \Sigma)^*$$

Type 3 Regular:

Languages defined by Type-3 grammars are accepted by finite state automata

Most syntax of some informal spoken dialog

Rules are of the form:

$$A \rightarrow \varepsilon$$

$$A \rightarrow \alpha$$

$$A \rightarrow \alpha B$$

where

$$A, B \in N \text{ and } \alpha \in \Sigma$$

The Universal Turing Machine

- If Tm's are so damned powerful, can't we build one that simulates the behavior of any Tm on any tape that it is given?

- Yes. This machine is called the *Universal Turing machine*.
 - How would we build a Universal Turing machine?
 - We place an encoding of any Turing machine on the input tape of the Universal Tm.
 - The tape consists entirely of zeros and ones (and, of course, blanks)
 - Any Tm is represented by zeros and ones, using unary notation for elements and zeros as separators.
 - Every Tm instruction consists of four parts, each a represented as a series of **1**'s and separated by **0**'s.
 - Instructions are separated by **00**.
 - We use unary notation to represent components of an instruction, with
 - $0 = \mathbf{1}$,
 - $1 = \mathbf{11}$,
 - $2 = \mathbf{111}$,
 - $3 = \mathbf{1111}$,
 - $n = \mathbf{111...111}$ ($n+1$ 1's).
 - We encode q_n as $n + 1$ 1's
 - We encode symbol a_n as $n + 1$ 1's
 - We encode move left as 1, and move right as 11
- 1111011101111101110100101101101101100**
- | | |
|-------------------------|-------------------------|
| q_3, a_2, q_4, a_2, L | q_0, a_1, q_1, a_1, R |
|-------------------------|-------------------------|
- Any Turing machine can be encoded as a unique long string of zeros and ones, beginning with a **1**.
 - Let T_n be the Turing machine whose encoding is the number n .

2 Turing Reducibility

- A language A is Turing reducible to a language B , written $A \leq_T B$, if A is decidable relative to B
- Below it is shown that E_{TM} is Turing reducible to EQ_{TM}
- Whenever A is mapping reducible to B , then A is Turing reducible to B
 - The function in the mapping reducibility could be replaced by an oracle
- An oracle Turing machine with an oracle for EQ_{TM} can decide E_{TM}

$T^{\text{EQ-TM}} = \text{"On input } \langle M \rangle$

1. Create TM M_1 such that $L(M_1) = \emptyset$

M_1 has a transition from start state to reject state for every element of Σ

1. Call the EQ_{TM} oracle on input $\langle M, M_2 \rangle$

2. If it accepts, accept; if it rejects, reject"

- $T^{\text{EQ-TM}}$ decides E_{TM}
- E_{TM} is decidable relative to EQ_{TM}
- **Applications**
 - If $A \leq_T B$ and B is decidable, then A is decidable
 - If $A \leq_T B$ and A is undecidable, then B is undecidable
 - If $A \leq_T B$ and B is Turing-recognizable, then A is Turing-recognizable
 - If $A \leq_T B$ and A is non-Turing-recognizable, then B is non-Turing-recognizable

3. The class P

A decision problem D is *solvable in polynomial time* or *in the class P*, if there exists an algorithm A such that

- A Takes instances of D as inputs.
- A always outputs the correct answer "Yes" or "No".
- There exists a polynomial p such that the execution of A on inputs of size n always terminates in $p(n)$ or fewer steps.
- **EXAMPLE:** The Minimum Spanning Tree Problem is in the class P.

The class P is often considered as synonymous with the class of computationally feasible problems, although in practice this is somewhat unrealistic.

The class NP

A decision problem is *nondeterministically polynomial-time solvable* or *in the class NP* if there exists an algorithm A such that

- A takes as inputs potential witnesses for "yes" answers to problem D .
- A correctly distinguishes true witnesses from false witnesses.

- There exists a polynomial p such that for each potential witnesses of each instance of size n of D , the execution of the algorithm A takes at most $p(n)$ steps.
- Think of a non-deterministic computer as a computer that magically “guesses” a solution, then has to verify that it is correct
 - If a solution exists, computer always guesses it
 - One way to imagine it: a parallel computer that can freely spawn an infinite number of processes
 - Have one processor work on each possible solution
 - All processors attempt to verify that their solution works
 - If a processor finds it has a working solution
 - So: **NP** = problems *verifiable* in polynomial time
 - Unknown whether **P** = **NP** (most suspect not)

NP-Complete Problems

- We will see that NP-Complete problems are the “hardest” problems in NP:
 - If any *one* NP-Complete problem can be solved in polynomial time.
 - Then *every* NP-Complete problem can be solved in polynomial time.
 - And in fact *every* problem in **NP** can be solved in polynomial time (which would show **P** = **NP**)
 - Thus: solve hamiltonian-cycle in $O(n^{100})$ time, you’ve proved that **P** = **NP**. Retire rich & famous.
- The crux of NP-Completeness is *reducibility*
 - Informally, a problem P can be reduced to another problem Q if *any* instance of P can be “easily rephrased” as an instance of Q , the solution to which provides a solution to the instance of P
 - *What do you suppose “easily” means?*
 - This rephrasing is called *transformation*
 - Intuitively: If P reduces to Q , P is “no harder to solve” than Q
- An example:
 - P : Given a set of Booleans, is at least one TRUE?
 - Q : Given a set of integers, is their sum positive?

- Transformation: $(x_1, x_2, \dots, x_n) = (y_1, y_2, \dots, y_n)$ where $y_i = 1$ if $x_i = \text{TRUE}$, $y_i = 0$ if $x_i = \text{FALSE}$
- Another example:
 - Solving linear equations is reducible to solving quadratic equations
 - *How can we easily use a quadratic-equation solver to solve linear equations?*
- Given one NP-Complete problem, we can prove many interesting problems NP-Complete
 - Graph coloring (= register allocation)
 - Hamiltonian cycle
 - Hamiltonian path
 - Knapsack problem
 - Traveling salesman
 - Job scheduling with penalties, etc.

NP Hard

- **Definition:** Optimization problems whose decision versions are NP- complete are called *NP-hard*.
- **Theorem:** If there exists a polynomial-time algorithm for finding the optimum in any *NP-hard* problem, then $P = NP$.

Proof: Let E be an *NP-hard* optimization (let us say minimization) problem, and let A be a polynomial-time algorithm for solving it. Now an instance J of the corresponding decision problem D is of the form (I, c) , where I is an instance of E , and c is a number. Then the answer to D for instance J can be obtained by running A on I and checking whether the cost of the optimal solution exceeds c . Thus there exists a polynomial-time algorithm for D , and *NP-completeness* of the latter implies $P = NP$.