AUTOMATA THEORY

Digital Notes By

BIGHNARAJ NAIK

Assistant Professor

Department of Master in Computer Application

VSSUT, Burla

Syllabus

4th SEMESTER MCA

F.M : 70

MCA 207 **AUTOMATA THEORY (3-1-0)Cr.-4**

Module – I

Introduction to Automata: The Methods Introduction to Finite Automata, Structural Representations, Automata and Complexity.
Proving Equivalences about Sets, The Contrapositive, Proof by Contradiction,
Inductive Proofs: General Concepts of Automata Theory: Alphabets Strings, Languages, Applications of Automata Theory.

Module – II

Finite Automata: The Ground Rules, The Protocol, Deterministic Finite Automata: Definition of a Deterministic Finite Automata, How a DFA Processes Strings, Simpler Notations for DFA's, Extending the Transition Function to Strings, The Language of a DFA
Nondeterministic Finite Automata: An Informal View. The Extended Transition Function, The Languages of an NFA, Equivalence of Deterministic and Nondeterministic Finite Automata.
Finite Automata With Epsilon-Transitions: Uses of $\in$-Transitions, The Formal Notation for an $\in$-NFA, Epsilon-Closures, Extended Transitions and Languages for $\in$-NFA's, Eliminating $\in$-Transitions.

Module – III

Regular Expressions and Languages: Regular Expressions: The Operators of regular Expressions, Building Regular Expressions, Precedence of Regular-Expression Operators, Precedence of Regular-Expression Operators
Finite Automata and Regular Expressions: From DFA's to Regular Expressions, Converting DFA's to Regular Expressions, Converting DFA's to Regular Expressions by Eliminating States, Converting Regular Expressions to Automata.
Algebraic Laws for Regular Expressions:
Properties of Regular Languages: The Pumping Lemma for Regular Languages, Applications of the Pumping Lemma Closure Properties of Regular Languages, Decision Properties of Regular Languages, Equivalence and Minimization of Automata,

Module – IV

Context-Free Grammars and Languages: Definition of Context-Free Grammars, Derivations Using a Grammars Leftmost and Rightmost Derivations, The Languages of a Grammar,
Parse Trees: Constructing Parse Trees, The Yield of a Parse Tree, Inference Derivations, and Parse Trees, From Inferences to Trees, From Trees to Derivations, From Derivation to Recursive Inferences,

Applications of Context-Free Grammars: Parsers, Ambiguity in Grammars and Languages: Ambiguous Grammars, Removing Ambiguity From Grammars, Leftmost Derivations as a Way to Express Ambiguity, Inherent Anbiguity

Module – V

Pushdown Automata: Definition Formal Definition of Pushdown Automata, A Graphical Notation for PDA's, Instantaneous Descriptions of a PDA,

The Languages of a PDA: Acceptance by Final State, Acceptance by Empty Stack, From Empty Stack to Final State, From Final State to Empty Stack

Equivalence of PDA's and CFG's: From Grammars to Pushdown Automata, From PDA's to Grammars

Deterministic Pushdown Automata: Definition of a Deterministic PDA, Regular Languages and Deterministic PDA's, DPDA's and Context-Free Languages, DPDA's and Ambiguous Grammars

Module – VI

Properties of Context-Free Languages: Normal Forms for Context-Free Grammars, The Pumping Lemma for Context-Free Languages, Closure Properties of Context-Free Languages, Decision Properties of CFL's

Module - VII

Introduction to Turing Machines: The Turing Machine: The Instantaneous Descriptions for Turing Machines, Transition Diagrams for Turing Machines, The Language of a Turing Machine, Turing Machines and Halting

Programming Techniques for Turing Machines, Extensions to the Basic Turing Machine,

Restricted Turing Machines, Turing Machines and Computers,

Module - VIII

Undecidability: A Language That is Not Recursively Enumerable, Enumerating the Binary Strings, Codes for Turing Machines, The Diagonalization Language

An Undecidable Problem That Is RE: Recursive Languages, Complements of Recursive and RE languages, The Universal Languages, Undecidability of the Universal Language

Undecidable Problems About Turing Machines: Reductions, Turing Machines That Accept the Empty Language

Post's Correspondence Problem: Definition of Post's Correspondence Problem, The "Modified" PCP

Other Undecidable Problems: Undecidability of Ambiguity for CFG's

REFERENCE BOOKS

1. Hopcroft, Ullman " Theory of Computation & Formal Languages", TMH.
2. FORMAL LANGUAGES AND AUTOMATA THEORY, **H S Behera**, **Janmenjoy Nayak** , **Hadibandhu Pattnayak**, Vikash Publishing, New Delhi.
3. Anand Sharma, "Theory of Automata and Formal Languages", Laxmi Publisher.

Formal language

The alphabet of a formal language is the set of symbols, letters, or tokens from which the strings of the language may be formed; frequently it is required to be finite. The strings formed from this alphabet are called words, and the words that belong to a particular formal language are sometimes called *well-formed words* or *well-formed formulas*. A formal language is often defined by means of a formal grammar such as a regular grammar or context-free grammar, also called its formation rule.

The field of **formal language theory** studies the purely syntactical aspects of such languages—that is, their internal structural patterns. Formal language theory sprang out of linguistics, as a way of understanding the syntactic regularities of natural languages. In computer science, formal languages are often used as the basis for defining programming languages and other systems in which the words of the language are associated with particular meanings or semantics.

A formal languageL over an alphabet Σ is a subset of Σ*, that is, a set of words over that alphabet.

In computer science and mathematics, which do not usually deal with natural languages, the adjective "formal" is often omitted as redundant.

While formal language theory usually concerns itself with formal languages that are described by some syntactical rules, the actual definition of the concept "formal language" is only as above: a (possibly infinite) set of finite-length strings, no more nor less. In practice, there are many languages that can be described by rules, such as regular languages or context-free languages. The notion of a formal grammar may be closer to the intuitive concept of a "language," one described by syntactic rules.

Formal language

A formal grammar (sometimes simply called a grammar) is a set of formation rules for strings in a formal language. The rules describe how to form strings from the language's alphabet that are

valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context—only their form.

A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting must start. Therefore, a grammar is usually thought of as a language generator. However, it can also sometimes be used as the basis for a "recognizer"—a function in computing that determines whether a given string belongs to the language or is grammatically incorrect. To describe such recognizers, formal language theory uses separate formalisms, known as automata theory. One of the interesting results of automata theory is that it is not possible to design a recognizer for certain formal languages.

Alphabet

An alphabet, in the context of formal languages, can be any set, although it often makes sense to use an alphabet in the usual sense of the word, or more generally a character set such as ASCII. Alphabets can also be infinite; e.g. first-order logic is often expressed using an alphabet which, besides symbols such as∧, ¬, □ and parentheses, contains infinitely many elements x0, x1, x2, … that play the role of variables. The elements of an alphabet are called its letters.

word

A word over an alphabet can be any finite sequence, or string, of letters. The set of all words over an alphabet Σ is usually denoted by Σ* (using the Kleene star). For any alphabet there is only one word of length 0, the empty word, which is often denoted by e, ε or λ. By concatenation one can combine two words to form a new word, whose length is the sum of the lengths of the original words. The result of concatenating a word with the empty word is the original word.

Operations on languages

Certain operations on languages are common. This includes the standard set operations, such as union, intersection, and complement. Another class of operation is the element-wise application of string operations.

Examples: suppose L1 and L2 are languages over some common alphabet.

- The concatenation L1L2 consists of all strings of the form vw where v is a string from L1 and w is a string from L2.
- The intersection L1 ∩ L2 of L1 and L2 consists of all strings which are contained in both languages
- The complement ¬L of a language with respect to a given alphabet consists of all strings over the alphabet that are not in the language.
- The Kleene star: the language consisting of all words that are concatenations of 0 or more words in the original language;
- Reversal:
    - Let e be the empty word, then eR = e, and
    - for each non-empty word w = x1…xn over some alphabet, let wR = xn…x1,
    - then for a formal language L, LR = {wR | w □ L}.
- String homomorphism

Such string operations are used to investigate closure properties of classes of languages. A class of languages is closed under a particular operation when the operation, applied to languages in the class, always produces a language in the same class again. For instance, the context-free languages are known to be closed under union, concatenation, and intersection with regular languages, but not closed under intersection or complement. The theory of trios and abstract families of languages studies the most common closure properties of language families in their own right.

Language

“A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols.In general, if $\Sigma$is an alphabet and L is a subset of Σ^*, then L is said to be a *language*over Σ, or simply a language if$\Sigma$is understood. Each element of L is said to be a *sentence*or a *word* or a *string*of the language.

Example 1 ***{0, 11, 001}, {ε, 10}, and {0, 1}* are subsets of {0, 1}*, and so they are languages over the alphabet {0, 1}.***

The empty set Ø and the set {ε} are languages over every alphabet. Ø is a language that contains no string. {ε} is a language that contains just the empty string.

The *union* of two languages L_1 and L_2, denoted $L_1 \cup L_2$, refers to the language that consists of all the strings that are either in L_1 or in L_2, that is, to { x | x is in L_1 or x is in L_2 }. The *intersection* of L_1 and L_2, denoted $L_1 \cap L_2$, refers to the language that consists of all the strings that are both in L_1 and L_2, that is, to { x | x is in L_1 and in L_2 }. The *complementation* of a language L over Σ, or just the complementation of L when Σ is understood, denoted $\overline{L}$, refers to the language that consists of all the strings over Σ that are not in L, that is, to { x | x is in Σ^* but not in L }.

Example 2 ***Consider the languages L_1 = {ε, 0, 1} and L_2 = {ε, 01, 11}. The union of these languages is $L_1 \cup L_2$ = {ε, 0, 1, 01, 11}, their intersection is $L_1 \cap L_2$ = {ε}, and the complementation of L_1 is $\overline{L_1}$ = {00, 01, 10, 11, 000, 001, . . . }.***

Ø ∪ L = L for each language L. Similarly, Ø ∩ L = Ø for each language L. On the other hand, $\overline{\emptyset} = \Sigma^*$ and $\overline{\Sigma^*} = \emptyset$ for each alphabet Σ.

The *difference* of L_1 and L_2, denoted $L_1 - L_2$, refers to the language that consists of all the strings that are in L_1 but not in L_2, that is, to { x | x is in L_1 but not in L_2 }. The *crossproduct* of L_1 and L_2, denoted $L_1 \times L_2$, refers to the set of all the pairs (x, y) of strings such that x is in L_1 and y is in L_2, that is, to the relation { (x, y) | x is in L_1 and y is in L_2 }. The *composition* of L_1 with L_2, denoted L_1L_2, refers to the language { xy | x is in L_1 and y is in L_2 }.

Example 3 ***If L_1 = {ε, 1, 01, 11} and L_2 = {1, 01, 101} then $L_1 - L_2$ = {ε, 11} and $L_2 - L_1$ = {101}.***

On the other hand, if L_1 = {ε, 0, 1} and L_2 = {01, 11}, then the cross product of these languages is $L_1 \times L_2$ = {(ε, 01), (ε, 11), (0, 01), (0, 11), (1, 01), (1, 11)}, and their composition is L_1L_2 = {01, 11, 001, 011, 101, 111}.

L - Ø = L, Ø - L = Ø, ØL = Ø, and {ε}L = L for each language L.

L^i will also be used to denote the composing of i copies of a language L, where L^0 is defined as { ε}. The set $L^0 \cup L^1 \cup L^2 \cup L^3 \ldots$, called the *Kleeneclosure* or just the *closure* of L, will be denoted by L*. The set $L^1 \cup L^2 \cup L^3 \ldots$, called the *positiveclosure* of L, will be denoted by L^+.

L^i consists of those strings that can be obtained by concatenating i strings from L. L* consists of those strings that can be obtained by concatenating an arbitrary number of strings from L.

Example 4 ***Consider the pair of languages L_1 = {ε, 0, 1} and L_2 = {01, 11}. For these languages L_1^2 = {ε, 0, 1, 00, 01, 10, 11}, and L_2^3 = {010101, 010111, 011101, 011111, 110101, 110111, 111101, 111111}. In addition, ε is in L_1*, in L_1^+, and in L_2* but not in L_2^+.***

The operations above apply in a similar way to relations in $\Sigma^* \times \Delta^*$, when Σ and Δ are alphabets. Specifically, the *union* of the relations R_1 and R_2, denoted $R_1 \cup R_2$, is the relation { (x, y) | (x, y) is in R_1 or in R_2 }. The *intersection* of R_1 and R_2, denoted $R_1 \cap R_2$, is the relation { (x, y) | (x, y) is in R_1 and in R_2 }. The *composition* of R_1 with R_2, denoted R_1R_2, is the relation { (x_1x_2, y_1y_2) | (x_1, y_1) is in R_1 and (x_2, y_2) is in R_2 }.

Grammar

It is often convenient to specify languages in terms of grammars. The advantage in doing so arises mainly from the usage of a small number of rules for describing a language with a large number of sentences. For instance, the possibility that an English sentence consists of a subject phrase followed by a predicate phrase can be expressed by a grammatical rule of the form <sentence>→<subject><predicate>. (The names in angular brackets are assumed to belong to the grammar metalanguage.) Similarly, the possibility that the subject phrase consists of a noun phrase can be expressed by a grammatical rule of the form <subject>→<noun>.

G is defined as a mathematical system consisting of a quadruple <N, Σ, P, S>, where

N : is an alphabet, whose elements are called nonterminalsymbols.
Σ : is an alphabet disjoint from N, whose elements are called terminalsymbols.
P :is a relation of finite cardinality on $(N \cup \Sigma)^*$, whose elements are called productionrules. Moreover, each production rule (α, β) in P, denoted $\alpha \rightarrow \beta$, must have at least one nonterminal

symbol in α. In each such production rule, $\alpha$is said to be the left-handside of the production rule, and $\beta$is said to be the right-handside of the production rule.

S is a symbol in N called the start, or sentence, symbol.

Types of grammars

Prescriptive: prescribes authoritative norms for a language

Descriptive: attempts to describe actual usage rather thanenforce arbitrary rules

Formal: a precisely defined grammar, such as context-free

Generative: a formal grammar that can generate naturallanguage expressions

Chomsky hierarchy of languages.

The Chomsky hierarchy consists of the following levels:

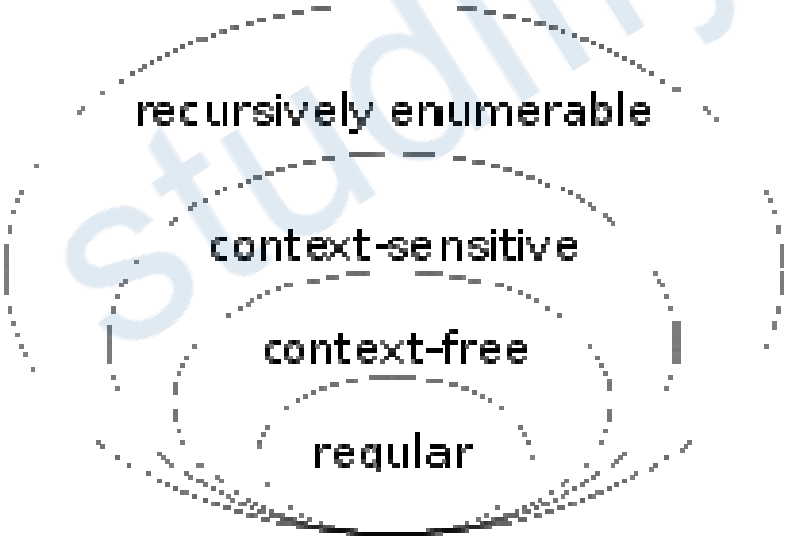
Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which can be decided by an always-halting Turing machine.

Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$with Aa nonterminal and α, $\beta$and γstrings of terminals and nonterminals. The strings $\alpha$and βmay be empty, but γmust be nonempty. The rule $S \rightarrow \epsilon$is allowed if Sdoes not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input.)

Type-2 grammars (context-free grammars) generate the context-free languages. These are defined by rules of the form $A \rightarrow \gamma$with Aa nonterminal and γa string of terminals and nonterminals. These languages are exactly all languages that can be recognized by a non-

deterministic pushdown automaton. Context-free languages are the theoretical basis for the syntax of most programming languages.

Type-3 grammars (regular grammars)generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal. The rule $S \rightarrow \epsilon$is also allowed here if Sdoes not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.



Examples:

1. The language consists of all strings begin with 0.

{0}{0, 1}*

2. The language consists of all strings begin with 0, and end with 1.

{0}{0, 1}*{1}

3. The language consists of all strings with odd lengths.

{0, 1}2n−1, n = 1, 2, . . .

4. The language consists of all strings with substring of three consecutive 0.

{0, 1}*000{0, 1}*

5. The language consists of all strings without substring of three consecutive 0.

{001, 01, 1}*

Regular grammar

A regular grammar is any right-linear or left-linear grammar.

A right regular grammar (also called right linear grammar) is a formal grammar (N, Σ, P, S) such that all the production rules in P are of one of the following forms:

B → a - where B is a non-terminal in N and a is a terminal in Σ

B → aC - where B and C are in N and a is in Σ

B → ε - where B is in N and ε denotes the empty string, i.e. the string of length 0.

In a left regular grammar (also called left linear grammar), all rules obey the forms

A → a - where A is a non-terminal in N and a is a terminal in Σ

A → Ba - where A and B are in N and a is in Σ

A → ε - where A is in N and ε is the empty string.

An example of a right regular grammar G with N = {S, A}, Σ = {a, b, c}, P consists of the following rules

S → aS

S → bA

A → ε

A → cA

andS is the start symbol. This grammar describes the same language as the regular expression a*bc*.

Extended regular grammars

An extended right regular grammar is one in which all rules obey one of

1. B → a - where B is a non-terminal in N and a is a terminal in Σ
2. A → wB - where A and B are in N and w is in Σ*
3. A → ε - where A is in N and ε is the empty string.

Some authors call this type of grammar a right regular grammar (or right linear grammar) and the type above a strictly right regular grammar (or strictly right linear grammar).

An extended left regular grammar is one in which all rules obey one of

1. A → a - where A is a non-terminal in N and a is a terminal in Σ
2. A → Bw - where A and B are in N and w is in Σ*
3. A → ε - where A is in N and ε is the empty string.

Regular expression

A regular expression (or regexp, or pattern, or RE) is a text string that describes some (mathematical) set of strings. A RE r matches a string s if s is in the set of strings described by r. Regular Expressions have their own notation. Characters are single letters for example ‘a’, ‘ ’(single blank space), ‘1’ and ‘-’ (hyphen). Operators are entries in a RE that match one or more characters.

Regular expressions consist of constants and operator symbols that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found as such in most textbooks on formal language theory. Given a finite alphabet Σ, the following constants are defined as regular expressions:

- (**empty set)**∅ denoting the set ∅.
- (**empty string)** ε denoting the set containing only the "empty" string, which has no characters at all.
- (**literal character)** a in Σ denoting the set containing only the character a.

Given regular expressions R and S, the following operations over them are defined to produce regular expressions:

- (**concatenation**) RS denoting the set { αβ | α in set described by expression R and β in set described by S }. For example {"ab", "c"}{"d", "ef"} = {"abd", "abef", "cd", "cef"}.
- (**alternation**) R | S denoting the set union of sets described by R and S. For example, if R describes {"ab", "c"} and S describes {"ab", "d", "ef"}, expression R | S describes {"ab", "c", "d", "ef"}.
- (**Kleene star**) R* denoting the smallest superset of set described by R that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from set described by R. For

example, {"0","1"}* is the set of all finite binary strings (including the empty string), and {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", ... }.

To avoid parentheses it is assumed that the Kleene star has the highest priority, then concatenation and then alternation. If there is no ambiguity then parentheses may be omitted. For example, (ab)c can be written as abc, and a|(b(c*)) can be written as a|bc*.

Examples:

- a|b* denotes {ε, "a", "b", "bb", "bbb", ...}
- (a|b)* denotes the set of all strings with no symbols other than "a" and "b", including the empty string: {ε, "a", "b", "aa", "ab", "ba", "bb", "aaa", ...}
- ab*(c|ε) denotes the set of strings starting with "a", then zero or more "b"s and finally optionally a "c": {"a", "ac", "ab", "abc", "abb", "abbc", ...}

Deterministic finite automaton (D.F.A)

- In the automata theory, a branch of theoretical computer science, a deterministic finite automaton (DFA)—also known as deterministic finite state machine—is a finite state machine that accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string. 'Deterministic' refers to the uniqueness of the computation.
- A DFA has a start state (denoted graphically by an arrow coming in from nowhere) where computations begin, and a set of accept states (denoted graphically by a double circle) which help define when a computation is successful.
- A DFA is defined as an abstract mathematical concept, but due to the deterministic nature of a DFA, it is implementable in hardware and software for solving various specific problems. For example, a DFA can model a software that decides whether or not online user-input such as email addresses are valid.
- DFAs recognize exactly the set of regular languageswhich are, among other things, useful for doing lexical analysis and pattern matching. DFAs can be built from nondeterministic finite automata through the powerset construction.

Formal definition

A deterministic finite automaton M is a 5-tuple, **(Q, Σ, δ, q_0, F)** consisting of

- a finite set of states (Q)
- a finite set of input symbols called the alphabet (Σ)
- a transition function ($\delta : Q \times \Sigma \rightarrow Q$)
- a start state ($q_0 \in Q$)
- a set of accept states ($F \in Q$)

  Let w = a1a2 ... an be a string over the alphabet Σ. The automaton M accepts the string w if a sequence of states, $r_0, r_1, ..., r_n$, exists in Q with the following conditions:
- $r_0 = q_0$
- $r_{i+1} = \delta(r_i, a_{i+1})$, for i = 0, ..., n−1
- $r_n \in F$.

  In words, the first condition says that the machine starts in the start state q_0. The second condition says that given each character of string w, the machine will transition from state to state according to the transition function δ. The last condition says that the machine accepts w if the last input of w causes the machine to halt in one of the accepting states. Otherwise, it is said that the automaton rejects the string. The set of strings M accepts is the language recognized by M and this language is denoted by L(M).

Transition Function Of DFA

A deterministic finite automaton without accept states and without a starting state is known as a transition system or semi automaton.

Given an input symbol $a \in \Sigma$, one may write the transition function as $\delta_a : Q \rightarrow Q$, using the simple trick of currying, that is, writing $\delta(q, a) = \delta_a(q)$ for all $q \in Q$. This way, the transition function can be seen in simpler terms: it's just something that "acts" on a state in Q, yielding another state. One may then consider the result of function composition repeatedly applied to the various functions δ_a, δ_b, and so on. Using this notion we define

$\hat{\delta} : Q \times \Sigma^* \to Q$. Given a pair of letters $a, b \in \Sigma$, one may define a new function $\hat{\delta}$, by insisting that $\hat{\delta}_{ab} = \delta_a \circ \delta_b$, where $\circ$ denotes function composition. Clearly, this process can be recursively continued. So, we have following recursive definition

$\hat{\delta}(q, \epsilon) = q$ where ϵ is empty string and

$\hat{\delta}(q, wa) = \delta_a(\hat{\delta}(q, w))$ where $w \in \Sigma^*, a \in \Sigma$ and $q \in Q$.

- $\hat{\delta}$ is defined for all words $w \in \Sigma^*$

Advantages and disadvantages

- DFAs were invented to model real world finite state machines in contrast to the concept of a Turing machine, which was too general to study properties of real world machines.
- DFAs are one of the most practical models of computation, since there is a trivial linear time, constant-space, online algorithm to simulate a DFA on a stream of input. Also, there are efficient algorithms to find a DFA recognizing:

1. the complement of the language recognized by a given DFA.
2. the union/intersection of the languages recognized by two given DFAs.

- Because DFAs can be reduced to a canonical form (minimal DFAs), there are also efficient algorithms to determine:

1. whether a DFA accepts any strings
2. whether a DFA accepts all strings
3. whether two DFAs recognize the same language
4. the DFA with a minimum number of states for a particular regular language

- DFAs are equivalent in computing power to nondeterministic finite automata (NFAs). This is because, firstly any DFA is also an NFA, so an NFA can do what a DFA can do. Also, given an NFA, using the powerset construction one can build a DFA that

recognizes the same language as the NFA, although the DFA could have exponentially larger number of states than the NFA.

- On the other hand, finite state automata are of strictly limited power in the languages they can recognize; many simple languages, including any problem that requires more than constant space to solve, cannot be recognized by a DFA.
- The classical example of a simply described language that no DFA can recognize is bracket language, i.e., language that consists of properly paired brackets such as word "(( )( ))".
- No DFA can recognize the bracket language because there is no limit to recursion, i.e., one can always embed another pair of brackets inside.
- It would require an infinite amount of states to recognize. Another simpler example is the language consisting of strings of the form $a_n b_n$—some finite number of a's, followed by an equal number of b's.

Nondeterministic finite automaton (N.F.A)

- In the automata theory, a nondeterministic finite automaton (NFA) or nondeterministic finite state machine is a finite state machine where from each state and a given input symbol the automaton may jump into several possible next states.
- This distinguishes it from the deterministic finite automaton (DFA), where the next possible state is uniquely determined.
- Although the DFA and NFA have distinct definitions, a NFA can be translated to equivalent DFA using powerset construction, i.e., the constructed DFA and the NFA recognize the same formal language. Both types of automata recognize only regular languages
- A NFA is represented formally by a 5-tuple, $(Q, \Sigma, \Delta, q0, F)$, consisting of

1. a finite set of states Q
2. a finite set of input symbols Σ
3. a transition relation $\Delta : Q \times \Sigma \rightarrow P(Q)$.
4. an initial (or start) state q0

5. a set of states F distinguished as accepting (or final) states F .

- Here, P(Q) denotes the power set of Q. Let w = a1a2 ... an be a word over the alphabet Σ. The automaton M accepts the word w if a sequence of states, r0,r1, ..., rn, exists in Q with the following conditions:

1. r0 = q0
2. ri+1 ∈ Δ(ri, ai+1), for i = 0, ..., n−1
3. $r_n \in F$.

Implementation

There are many ways to implement a NFA:

- Convert to the equivalent DFA. In some cases this may cause exponential blowup in the size of the automaton and thus auxiliary space proportional to the number of states in the NFA (as storage of the state value requires at most one bit for every state in the NFA)[4]
- Keep a set data structure of all states which the machine might currently be in. On the consumption of the last input symbol, if one of these states is a final state, the machine accepts the string.
- In the worst case, this may require auxiliary space proportional to the number of states in the NFA; if the set structure uses one bit per NFA state, then this solution is exactly equivalent to the above.

- Create multiple copies. For each n way decision, the NFA creates up to $n - 1$ copies of the machine. Each will enter a separate state.
- If, upon consuming the last input symbol, at least one copy of the NFA is in the accepting state, the NFA will accept. (This, too, requires linear storage with respect to the number of NFA states, as there can be one machine for every NFA state.)
- Explicitly propagate tokens through the transition structure of the NFA and match whenever a token reaches the final state. This is sometimes useful when the NFA should encode additional context about the events that triggered the transition.

Decision property of Regular Language

A decision property for a class of languages is an algorithm that takes a formal description of a language (e.g., a DFA) and tells whether or not some property holds.

- Example: Is language L empty?
- You might imagine that the language isdescribed informally, so if the description is “the empty language”then yes, otherwise no.But the representation is a DFA (or aRE that you will convert to a DFA).

Closure Properties

- A closure property of a language classsays that given languages in the class,an operator (e.g., union) producesanother language in the same class.
- Example: the regular languages areobviously closed under union,concatenation, and (Kleene) closure.Use the RE representation of languages.
- The principal closure properties of regular languages are:

  1.The union of two regular languages is regular.

  If L and M are regular languages, then so is L ∪ M.

  2. The intersection of two regular languages is regular.

  If L and M are regular languages, then so is L ∩ M.

  3. The compliment of two regular languages is regular.

  If L is a regular language over alphabet Σ, then Σ*-L is also regular language.

  4. The difference of two regular languages is regular.

  If L and M are regular languages, then so is L - M.

  5. The reversal of a regular language is regular.

  The reversal of a string means that the string is written backward, i.e. reversal of abcde is edcba.

  The reversal of a language is the language consisting of reversal of all its strings, i.e. if L={001,110} then

  L® = {100,011}.

  6.The closure of a regular language is regular.

  If L is a regular language, then so is L*.

7. The concatenation of regular languages is regular.

If L and M are regular languages, then so is L M.

8.The homomorphism of a regular language is regular.

A homomorphism is a substitution of strings for symbol. Let the function h be defined by h(0) = a and h(1) = b then h applied to 0011 is simply aabb.

If h is a homomorphism on alphabet Σ and a string of symbols w = abcd…z then

$$h(w) = h(a)\, h(b)\, h(c)\, h(d) \ldots h(z)$$

The mathematical definition for homomorphism is

$$h: \Sigma^* \rightarrow \Gamma^* \text{ such that } \forall\, x, y \in \Sigma^*$$

A homomorphism can also be applied to a language by applying it to each of strings in the language. Let L be a language over alphabet Σ, and h is a homomorphism on Σ, then

$$h(L) = \{\, h(w) \mid w \text{ is in } L \,\}$$

The theorem can be stated as " If L is a regular language over alphabet Σ, and h is a homomorphism on Σ, then h(L) is also regular " .

9. The inverse homomorphism of two regular languages is regular.

Suppose h be a homomorphism from some alphabet Σ to strings in another alphabet T and L be a language over T then h inverse of L, h′ (L) is set of strings w inΣ^* such that h(w) is in L.

The theorem states that " If h is a homomorphism from alphabet Σ to alphabet T , and L is aregular language on T , then h′(L) is also a regular language.

Pumping lemma for regular languages

- The pumping lemma for regular languages describes an essential property of all regular languages.
- Informally, it says that all sufficiently long words in a regular language may be pumped — that is, have a middle section of the word repeated an arbitrary number of times — to produce a new word which also lies within the same language.
- Specifically, the pumping lemma says that for any regular language L there exists a constant p such that any word w in L with length at least p can be split into three substrings, w = xyz, where the middle portion y must not be empty, such that the words

xz, xyz, xyyz, xyyyz, … constructed by repeating y an arbitrary number of times (including zero times) are still in L. This process of repetition is known as "pumping".

- Moreover, the pumping lemma guarantees that the length of xy will be at most p, imposing a limit on the ways in which w may be split.
- Finite languages trivially satisfy the pumping lemma by having p equal to the maximum string length in L plus one.

  Let L be a regular language. Then there exists an integer $p \geq 1$ depending only on L such that every string w in L of length at least p (p is called the "pumping length") can be written as w = xyz (i.e., w can be divided into three substrings), satisfying the following conditions:

  1. $|y| \geq 1$
  2. $|xy| \leq p$
  3. for all $i \geq 0$, $xy^i z \in L$

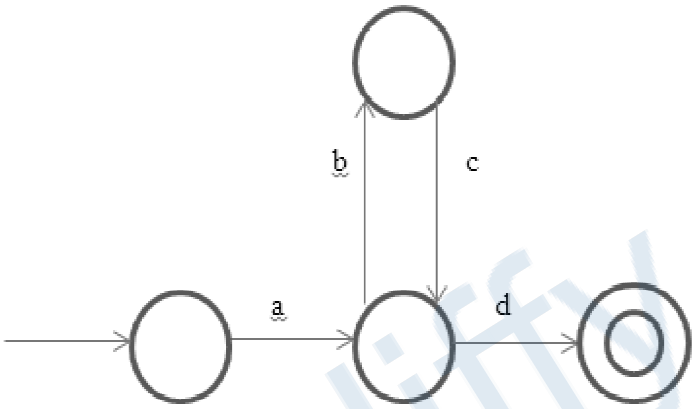
  y is the substring that can be pumped (removed or repeated any number of times, and the resulting string is always in L). (1) means the loop y to be pumped must be of length at least one; (2) means the loop must occur within the first p characters. There is no restriction on x and z.

  In simple words, For any regular language L, any sufficiently long word w(in L) can be split into 3 parts. i.e w = xyz , such that all the strings $xy^k z$ for $k \geq 0$ are also in L.

Proof of the pumping lemma

For every regular language there is a finite state automaton (FSA) that accepts the language. The number of states in such an FSA are counted and that count is used as the pumping length p. For a string of length at least p, let s0 be the start state and let s1, ...,spbe the sequence of the next p states visited as the string is emitted. Because the FSA has only p states, within this sequence of p + 1 visited states there must be at least one state that is repeated. Write S for such a state. The transitions that take the machine from the first encounter of state S to the second encounter of state S match some string. This string is called y in the lemma, and since the machine will match a string without the y portion, or the string y can be repeated any number of times, the conditions of the lemma are satisfied.

For example, the following image shows an FSA.



The FSA accepts the string: abcd. Since this string has a length which is at least as large as the number of states, which is four, the pigeonhole principle indicates that there must be at least one repeated state among the start state and the next four visited states. In this example, only q1 is a repeated state. Since the substring bc takes the machine through transitions that start at state q1 and end at state q1, that portion could be repeated and the FSA would still accept, giving the string abcbcd. Alternatively, the bc portion could be removed and the FSA would still accept giving the string ad. In terms of the pumping lemma, the string abcd is broken into an x portion a, a y portion bc and a z portion d.

DFA minimization

- DFA minimization is the task of transforming a given deterministic finite automaton (DFA) into an equivalent DFA that has minimum number of states. Here, two DFAs are called equivalent if they describe the same regular language.
- For each regular language that can be accepted by a DFA, there exists a DFA with a minimum number of states (and thus a minimum programming effort to create and use) and this DFA is unique (except that states can be given different names.)

- There are three classes of states that can be removed/merged from the original DFA without affecting the language it accepts.
- Unreachable states are those states that are not reachable from the initial state of the DFA, for any input string.
- Dead states are those nonaccepting states whose transitions for every input character terminate on themselves. These are also called Trap states because once entered there is no escape.
- Nondistinguishable states are those that cannot be distinguished from one another for any input string.
- DFA minimization is usually done in three steps, corresponding to the removal/merger of the relevant states. Since the elimination of nondistinguishable states is computationally the most expensive one, it's usually done as the last step.

Left and right linear grammars.

A linear language is a language generated by some linear grammar.

Example

A simple linear grammar is G with N = {S}, Σ = {a, b}, P with start symbol S and rules

$S \rightarrow aSb$

$S \rightarrow \varepsilon$

Two special types of linear grammars are the following:

- the left-linear or left regular grammars, in which all nonterminals in right hand sides are at the left ends;
- the right-linear or right regular grammars, in which all nonterminals in right hand sides are at the right ends.

Collectively, these two special types of linear grammars are known as the regular grammars; both can describe exactly the regular languages.

Another special type of linear grammar is the following:

- linear grammars in which all nonterminals in right hand sides are at the left or right ends, but not necessarily all at the same end.

By inserting new nonterminals, every linear grammar can be brought into this form without affecting the language generated. For instance, the rules of G above can be replaced with

$S \rightarrow aA$

$A \rightarrow Sb$

$S \rightarrow \varepsilon$

Hence, linear grammars of this special form can generate all linear languages.

Left linear Grammar

A -> B a or A -> a,where A and B are in N and a is in S

Right linearGrammar

A -> a B or A -> a,

where A and B are in N and a is in S

Example:

S -> a T
S -> b T
S -> a
S -> b
T -> a T
T -> b T
T -> 1
T
T -> 2
T
T -> a
T -> b
T -> 1
T -> 2

S => a
S => a T => a 1
S => a T => a 1 T

Constructing a Nondeterministic Finite State Automaton from a Right Linear Grammar

Let G = (N, S, P, S).

Construct a nondeterministic finite state automaton

M = (Q, S, d, S, F), where

Q = N U {X}, X not in N or S

F = {X}

d is constructed by

If A -> a B is in P, then B is in d(A,a)

If A -> a is in P, then X is in d (A,a)

S -> a T	**T -> 1 T**
S -> b T	**T -> 2 T**
S -> a	**T -> a**
S -> b	**T -> b**
T -> a T	**T -> 1**
T -> b T	**T -> 2**

d(S,a) = {T, X}
d(S,b) = {T,X}
d(S,1) = F
d(S,2) = F
d(T,a) = {T,X}
d(T,b) = {T,X}
d(T,1) = {T,X}
d(T,2) = {T,X}

A Left Linear Grammar for Identifiers

Example:

S -> S a
S -> S b
S -> S 1
S -> S 2
S -> a
S -> b

S => a
S => S 1 => a 1
S => S 2 => S b 2
=> S 1 b 2 => a 1 b 2

Constructing a Nondeterministic Finite State Automaton from a Left Linear Grammar

Let G = (N, S, P, S).
Construct a nondeterministic finite state automaton M = (Q, S, d, X, F), where
Q = N U {X}, X not in N or S
F = {S}
d is constructed by
If A -> B a is in P, then A is in d(B,a)
If A -> a is in P, then A is in d (X,a)

S -> S a
S -> S b
S -> S 1
S -> S 2
S -> a
S -> b

d(X,a) = {S}
d(X,b) = {S}
d(X,1) = F
d(X,2) = F
d(S,a) = {S}
d(S,b) = {S}
d(S,1) = {S}
d(S,2) = {S}

Context free grammars

A context-free grammar G is defined by the 4-tuple:

G=(V,T,P,S) where

1. V is a finite set; each element v ∈ V is called a non-terminal character or a variable. Each variable represents a different type of phrase or clause in the sentence. Variables are also sometimes called syntactic categories. Each variable defines a sub-language of the language defined by G.
2. T is a finite set of terminals, disjoint from V, which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar G.
3. P is a set of production rule.
4. S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V.

Example:

1. S → x
2. S → y
3. S → z
4. S → S + S
5. S → S - S
6. S → S * S
7. S → S / S
8. S → (S)

This grammar can, for example, generate the string

(x + y) * x - z * y / (x + x)

as follows:

S (the start symbol)

→ S - S (by rule 5)

→ S * S - S (by rule 6, applied to the leftmost S)

→ S * S - S / S (by rule 7, applied to the rightmost S)

→ (S) * S - S / S (by rule 8, applied to the leftmost S)

→ (S) * S - S / (S) (by rule 8, applied to the rightmost S)

→ (S + S) * S - S / (S) (etc.)

→ (S + S) * S - S * S / (S)

→ (S + S) * S - S * S / (S + S)

→ (x + S) * S - S * S / (S + S)

→ (x + y) * S - S * S / (S + S)

→ (x + y) * x - S * y / (S + S)

→ (x + y) * x - S * y / (x + S)

→ (x + y) * x - z * y / (x + S)

→ (x + y) * x - z * y / (x + x)

Problem 1. Give a context-free grammar for the language

$L = \{a^n b^m : n \neq 2m\}$.

Is your grammar ambiguous?

Ans:

A grammar for the language is

S →aaSb | A | B | ab

A →aA | a

B →bB | b

This grammar is unambiguous; it is not hard to prove that every string in the langauge has one

and only one parse tree.

Problem 2. Give a context-free grammar for the language

L = {x ∈ {0, 1}_ : x has the same number of 0's and 1's}

Is your grammar ambiguous?

Ans:

A grammar for the language is

S →S0S1S | S1S0S | ε

This grammar is ambiguous; for example, 0101 has two different parse trees

Problem 3. Prove that L = {aibjck : j = max{i, k}} is not context free.

Ans:

Suppose for contradiction that L were context free. Let N be the "N" of the pumping lemma

for context-free languages. Consider the string $w = a^N b^N c^N$. Suppose w = uvxyz, where $|vxy| \leq N$ and $|vy| \geq 1$. If vy contains only a's or vy contains only c's, then pump up: the string $uv^2xy^2z \notin L$. Suppose vy contains only b's. Then we can pump either way to get a string not in L. Suppose v contains two different letters or y contains two different letters. Then uv^2xy^2z is not even of the form a*b*c*, so certainly it is not in L. Finally, suppose ($v \in a+$ and) $y \in b+$, or $v \in b+$ (and $y \in c+$). Then we can pump down and there will be too few b's. By $|vwy| \leq N$, these are all the possible cases. So in all cases there is some i for which $uv^ixy^iz \notin L$, a contradiction.

Theorem : L ⊆A* is CF iff ∃NPDA M that accepts L.

Proof ->: Given CFL L, consider any grammar G(L) for L. Construct NPDA M that simulates all possible derivations of G. M is essentially a single-state FSM, with a state q that applies one of G's rules at a time. The start state q0 initializes the stack with the content S ¢, where S is the start symbol of G, and ¢ is the bottom of stack symbol. This initial stack content means that M aims to read an input that is an instance of S. In general, the current stack content is a sequence of symbols that represent tasks to be accomplished in the characteristic LIFO order (last-in first-out). The task on top of the stack, say a non-terminal X, calls for the next characters of the input string to be an instance of X. When these characters have been read and verified to be an instance of X, X is popped from the stack, and the new task on top of the stack is started. When ¢ is on top of the stack, i.e. thestack is empty, all tasks generated by the first instance of S have been successfully met, i.e. the input string read so far is an instance of S. M moves to the accept state and stops.

The following transitions lead from q to q:

1) □, X-> w for each rule X -> w. When X is on top of the stack, replace X by a right-hand side for X.

2) a, a -> □ for each a ∈ A. When terminal a is read as input and a is also on top of the stack, pop the stack.

Rule 1 reflects the following fact: one way to meet the task of finding an instance of X as a prefix of the inputstring not yet read, is to solve all the tasks, in the correct order, present in the right-hand side w of the productionX -> w. M can be considered to be a non-deterministic parser for G. A formal proof that M accepts precisely Lcan be done by induction on the length of the derivation of any w ∈ L.

Ambiguous Grammar;

A grammar is said to be ambiguous if more than two parse trees can be constructed from it.

Example 1:

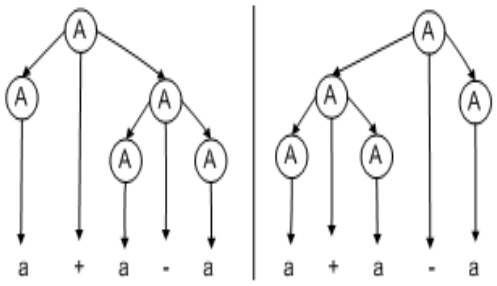
The context free grammar

$$A \rightarrow A+A \mid A-A \mid a$$

is ambiguous since there are two leftmost derivations for the string a + a + a:

$A \rightarrow A+A$	$A \rightarrow A+A$
$\rightarrow a+A$	$\rightarrow A+A+A$ (First A is replaced by A+A. Replacement of the second A would yield a similar derivation)
$\rightarrow a+A+A$	$\rightarrow a+A+A$
$\rightarrow a+a+A$	$\rightarrow a+a+A$
$\rightarrow a+a+a$	$\rightarrow a+a+a$

As another example, the grammar is ambiguous since there are two parse trees for the string a + a − a:



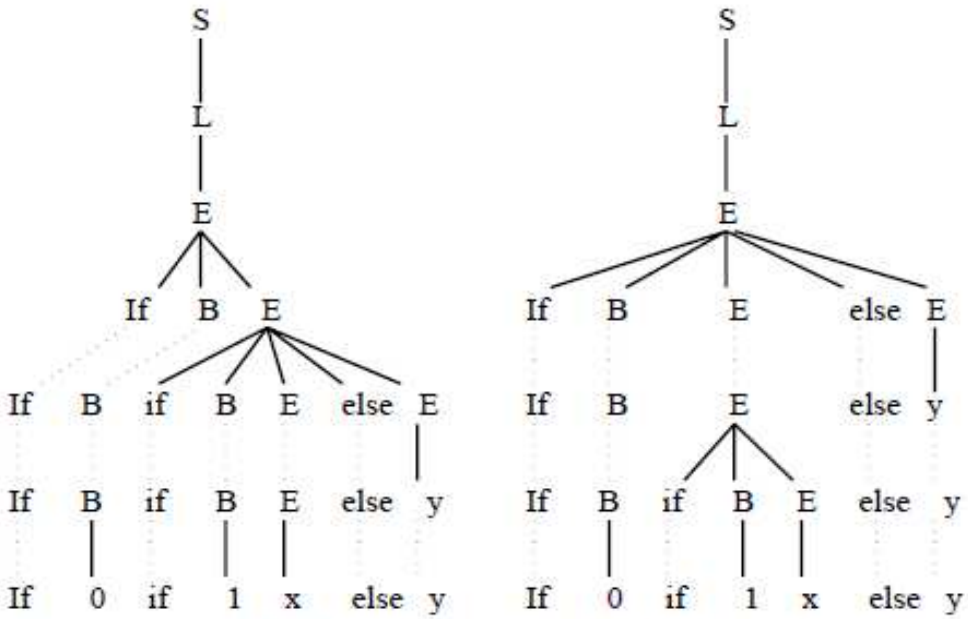
The language that it generates, however, is not inherently ambiguous; the following is a non-ambiguous grammar generating the same language:

$$A \rightarrow A+a \mid A-a \mid a$$

Example 2 : Show that the following grammar is ambigious

$$
\begin{aligned}
S &\longrightarrow L \\
L &\longrightarrow E \\
L &\longrightarrow LE \\
E &\longrightarrow \text{if B E else E} \\
E &\longrightarrow \text{if B E} \\
E &\longrightarrow x \\
E &\longrightarrow y \\
B &\longrightarrow 0 \\
B &\longrightarrow 1
\end{aligned}
$$

Answer :



As you can see we can have two corresponding parse trees for the above grammar, so the grammar is ambiguous.

Removal of Ambiguity

For compiling applications we need to design unambiguous grammar, or to use ambiguous grammar with additional rules to resolve the ambiguity.

1. Associativity of operators.
2. Precedence of operators.
3. Separate rules or Productions.

1. Associativity of Operators:

If operand has operators on both side then by connection, operand should be associated with the operator on the left.

In most programming languages arithmetic operators like addition, subtraction, multiplication, and division are left associative.

- Token string: 9 - 5 + 2
- Production rules
  list → list - digit | digit
  digit → 0 | 1 | 2 | . . . | 9

In the C programming language the assignment operator, =, is right associative. That is, token string a = b = c should be treated as a = (b = c).

- Token string: a = b =c.
- Production rules:
  right → letter = right | letter
  letter → a | b | . . . | z

2. Precedence of Operators:

An expression 9 + 5 * 2 has two possible interpretation:

(9 + 5) * 2 and 9 + (5 * L)

The associativity of '+' and '*' do not resolve this ambiguity. For this reason, we need to know the relative precedence of operators.

The convention is to give multiplication and division higher precedence than addition and subtraction.

Only when we have the operations of equal precedence, we apply the rules of associative.

So, in the example expression: 9 + 5 * 2.

We perform operation of higher precedence i.e., * before operations of lower precedence i.e., +.

Therefore, the correct interpretation is 9 + (5 *).

3. Separate Rule:

Consider the following grammar and language again.

S → IF b THEN S ELSE S
| IF b THEN S
| a

An ambiguity can be removed if we arbitrary decide that an ELSE should be attached to the last preceding THEN.

We can revise the grammar to have two nonterminals S_1 and S_2. We insist that S_2 generates IF-THEN-ELSE, while S_1 is free to generate either kind of statements.

The rules of the new grammar are:

S_1 → IF b THEN S_1
| IF b THEN S_2 THEN S_1
| a

S_2 → IF b THEN S_2 ELSE S_2
| a

Although there is no general algorithm that can be used to determine if a given grammar is ambiguous, it is certainly possible to isolate rules which leads to ambiguity or ambiguous grammar.

Example:

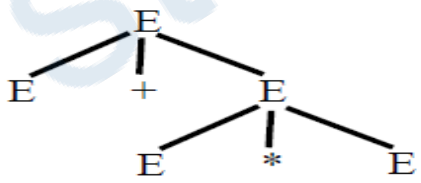
Show that the given grammar is ambiguous and also remove the ambiguity.

$$E \rightarrow I/E+E/E*E/(E)$$
$$I \rightarrow a/b/Ia/Ib/I0/I1$$

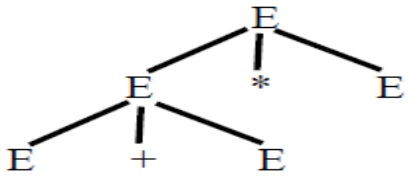
Answer :

Consider the sentential form $E+E*E$. It has two derivations from E.

1. $E \Rightarrow E+E \Rightarrow E+E*E.$
2. $E \Rightarrow E*E \Rightarrow E+E*E.$



(a)



(b)

As two parse trees can be possible so the above given grammar is ambigious.

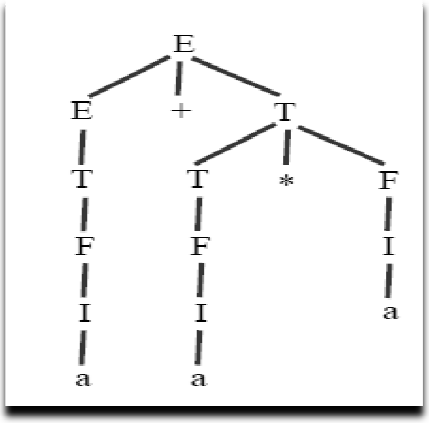
The solution to the problem of enforcing precedence is to introduce several different variables, each of which represents those expressions that share a level of **binding strength**. Specially:

1. A **factor** is an expression that can not be broken apart by any adjacent operator, either a * or a +. The only factors in our expression language are:
   (a) Identifiers. It is not possible to separate the letters of an identifier by attaching an operator.
   (b) Any parenthesized expression, no matter what appears inside the parentheses. It is the purpose of parentheses to prevent what is inside from becoming the operand of any operator outside the parentheses.
2. A **term** is an expression that cannot be broken by the + operator. In our example, where + and * are the only operators, a term is a product of one or more factors. For instance, the term $a * b$ can be **broken** if we use left associativity and place $a1*$ to its left. That is, $a1 * a * b$ is grouped $(a1 * a) * b$, which breaks apart the $a * b$. However, placing additive term, such as $a1+$ to its left or $+a1$ to its right cannot break $a * b$. The proper grouping of $a1 + a * b$ is $a1 + (a * b)$, and the proper grouping of $a * b + a1$ is (a*b)+a1.
3. An **expression** will henceforth refer to any possible expression, including those that can be broken by either an adjacent * or an adjacent +. Thus, an expression for our example is a sum of one or more terms.

An unambiguous expression grammar

$$
\begin{array}{|l|}
\hline
E \rightarrow T/E + T \\
T \rightarrow F/T * F \\
F \rightarrow I/(E) \\
I \rightarrow a/b/Ia/Ib/I0/I1 \\
\hline
\end{array}
$$

Now the same parse tree can be drawn as follows.



Inherent Ambiguity

A context-free language L is said to be inherently ambiguous if all its grammars are ambiguous.

If even one grammar for L is unambiguous, then L is an unambiguous language.

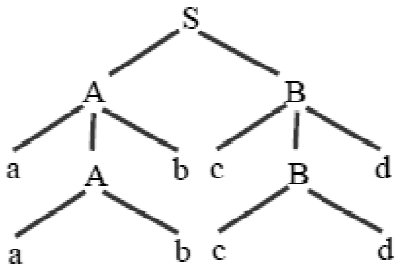
Example :

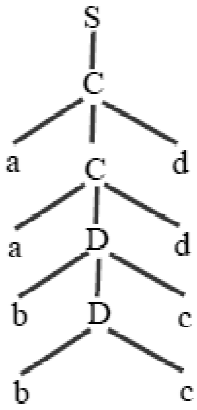
A grammar for an inherently ambiguous language

$$\begin{array}{|l} \hline S \rightarrow AB/C \\ A \rightarrow aAb/ab \\ B \rightarrow cBd/cd \\ C \rightarrow aCd/aDd \\ D \rightarrow bDc/bc \\ \hline \end{array}$$

This grammar is ambiguous. For example, the string $aabbccdd$ has the two leftmost derivations:

1. $S \Rightarrow_{lm} AB \Rightarrow_{lm} aAbB \Rightarrow_{lm} aabbB \Rightarrow_{lm} aabbcBd \Rightarrow_{lm} aabbccdd.$
2. $S \Rightarrow_{lm} C \Rightarrow_{lm} aCd \Rightarrow_{lm} aaDdd \Rightarrow_{lm} aabDdd \Rightarrow_{lm} aabbccdd$





Two parse trees for aabbccdd

Pumping Lemma For CFL

For every CFL L there is a constant n such that every z ∈ L of length |z| ≥ n can be written as z = u v w x y such that the following holds:

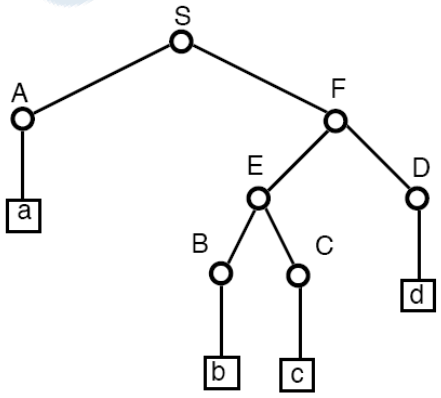
1) v x ≠ □

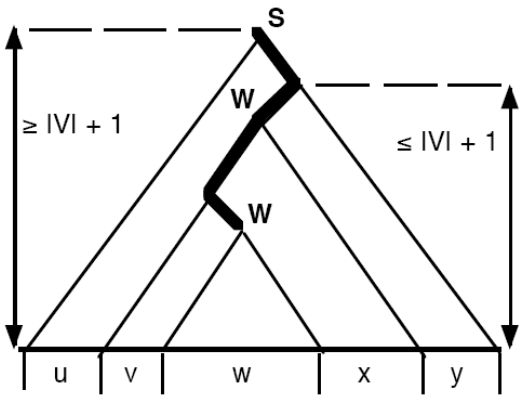
2) |v w x| ≤ n, and

3) uvk w xk y ∈ L for all k ≥ 0.

Proof:

Given CFL L, choose any G = G(L) in Chomsky NF. This implies that the parse tree of any z ∈ L is a binary tree, as shown in the figure below at left. The length n of the string at the leaves and the height h of a binary tree are related by h ≥ log n, i.e. a long string requires a tall parse tree. By choosing the critical length n = 2 |V | + 1 we force the height of the parse trees considered to be h ≥ |V| + 1. On a root-to-leaf path of length ≥ |V| + 1 we encounter at least |V| + 1 nodes labeled by non-terminals. Since G has only |V| distinct non-terminals, this implies that on some long root-to-leaf path we must encounter 2 nodes labeled with the same non-terminal, say W, as shown at right.





For two such occurrences of W (in particular, the two lowest ones), and for some u, v, y, x, w ∈ A*, we have: S ->* u W y, W ->* v W x and W ->* w. But then we also have W ->* v2 W x2, and in general, W ->* vkWxk, and S ->* u vk W xk y and S ->* u vk w xk y for all k ≥ 0,

Example-1 : Let G be a CFG in Chomsky normal form that contains b variables. Show that, if G generates some string with a derivation having at least 2b steps, L(G) is infinite.

Answer:

Since G is a CFG in Chomsky normal form, every derivation can generate at most two non-terminals, so that in any parse tree using G, an internal node can have at most two children. This implies that every parse tree with height k has at most 2^k–1 internal nodes.If G generates some string with a derivation having at least 2^b steps, the parse tree of that string will have at least 2^b internal nodes. Based on the above argument, this parse tree has height is at least b + 1, so that there exists a path from root to leaf containing b + 1 variables. By pigeonhole principle, there is one variable occurring at least twice. So, we can use the technique in the proof of the pumping lemma to construct infinitely many strings which are all in L(G).

Example-2 :

Let C = {xy | x, y 2 {0, 1}_, |x| = |y|, and x 6= y}. Show that C is a context-free language.

Answer:

We observe that a string is in C if and only if it can be written as xywith |x| = |y| such that for some i, the i^{th} character of x is different from the i^{th} character of y. To obtain such a string, we start generating the corresponding i^{th}characters, and fill up the remaining characters. Based on the above idea, we define the CFG for C is as follows:

$S \rightarrow AB \mid BA$

$A \rightarrow XAX \mid 0$

$B \rightarrow XBX \mid 1$

$X \rightarrow 0 \mid 1$

Let A =

Example-3 :

Let A= {wtw^R| w, t ∈ {0, 1}*and |w| = |t| }Prove that A is not a context-free language.

Answer:

Suppose on the contrary that A is context-free. Then, let p be the pumping length for A, such that any string in A of length at least p will satisfy the pumping lemma. Now, we select a string s in A with s = $0^{2p}0^p1^p0^{2p}$. For s to satisfy the pumping lemma,

there is a way that s can be written as uvxyz, with $|vxy| \leq p$ and $|vy| \geq 1$, and for any i, uv^ixy^iz is a string in A. There are only three cases to write s with the above conditions:

Case 1: vy contains only 0s and these 0s are chosen from the last 0^{2p} of s. Let i be a number with $7p > |vy| \times (i + 1) \geq 6p$. Then, either the length of $uv^ixy^iz$is not a multiple of 3, or this string is of the form wtw0 such that $|w| = |t| = |w'|$ with w′is all 0s and w is not all 0s (this is, $w' = w^R$).

Case 2: vy does not contain any 0s in the last 0^2p of s. Then, either the length of uv^2xy^2z is not a multiple of 3, or this string is of the form wtw′ such that $|w| = |t| = |w'|$ with w is all 0s and w′ is not all 0s (that is, $w' = w^R$).

Case 3: vy is not all 0s, and some 0s are from the last 0^2p of s. As |vxy| _ p, vxy in this case must be a substring in 1p′p. Then, either the length of uv2xy2z is not a multiple of 3, or this string is of the form wtw′such that $|w| = |t| = |w'|$ with w is all 0s and w′ is not all 0s (that is, w′ 6= w^R).

In summary, we observe that there is no way s can satisfy the pumping lemma. Thus, a contradiction occurs (where?), and we conclude that A is not a context-free language.

Theorem : L1 = { 0k 1k 2k / k ≥ 0 } is not context free.

Pf (by contradiction): Assume L is CF, let n be the constant asserted by the pumping lemma. Consider z = 0n 1n 2n = u v w x y. Although we don't know where vwx is positioned within z, the assertion $|vw x| \leq n$ implies that v w x contains at most two distinct letters among 0, 1, 2. In other words, one or two of thethree letters 0, 1, 2 is missing in vwx. Now consider u v2 w x2 y. By the pumping lemma, it must be in L. Theassertion $|v x| \geq 1$ implies that u v2 w x2 y is longer than u v w x y. But u v w x y had an equal number of 0s, 1s,and 2s, whereas u v2 w x2 y cannot, since only one or two of the three distinct symbols increased in number. Thiscontradiction proves the theorem.

Theorem : L2 = { w w / w ∈ {0, 1} } is not context free.

Proof (by contradiction): Assume L is CF, let n be the constant asserted by the pumping lemma. Consider z = 0n+1 1n+1 0n+1 1n+1 = u v w x y. Using k = 0, the lemma asserts z0 = u w y ∈ L, but we showthat z0 cannot have the form t t, for any string t, and thus that z0 ∈ L, leading to a contradiction. Recall that $|v wx| \leq n$, and thus, when we delete v and x, we delete symbols that

are within a distance of at most n from eachother. By analyzing three cases we show that, under this restriction, it is impossible to delete symbols in such away as to retain the property that the shortened string z0 = u w x has the form t t. We illustrate this using theexample n = 3, but the argument holds for any n.Given z = 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1, slide a window of length n = 3 across z, and delete any characters youwant from within the window. Observe that the blocks of 0s and of 1s within z are so long that the truncated z,call it z', still has the form "0s 1s 0s 1s". This implies that if z' can be written as z' = t t, then t must have theform t = "0s 1s". Checking the three cases: the window of length 3 lies entirely within the left half of z; thewindow straddles the center of z; and the window lies entirely within the right half of z, we observe that in noneof these cases z' has the form z' = t t, and thus that z0 = u w y ∉ L.

Context sensitive grammars and languages

The rewriting rules B -> w of a CFG imply that a non-terminal B can be replaced by a word w ∈ (V □ A)* "in any context". In contrast, a context sensitive grammar (CSG) has rules of the form: u B v -> u w v, where u, v, w ∈ (V □ A)*, implying that B can be replaced by w only in the context "u on the left, v on the right". It turns out that this definition is equivalent (apart from the nullstring□) to requiring that any CSG rule be of the form v -> w, where v, w ∈ (V □ A)*, and |v| ≤ |w|. This monotonicity property (in any derivation, the current string never gets shorter) implies that the word problem for CSLs: "given CSG G and given w, is w ∈ L(G)?" is decidable. An exhaustive enumeration of all derivations up to the length |w| settles the issue. As an example of the greater power of CSGs over CFGs, recall that we used the pumping lemma to prove that the language $0^k 1^k 2^k$ is not CF.

Parikh's theorem

Parikh's theorem in theoretical computer science says that if we look only at the relative number of occurrences of terminal symbols in a context-free language, without regard to their order, then the language is indistinguishable from a regular language. It is useful for deciding whether or not a string with given number of some terminals is accepted by a context-free grammar. It was first proved by Rohit Parikh in 1961 and republished in 1966.

Definitions

Let $\Sigma = \{a_1, a_2, \ldots, a_k\}$ be an alphabet. The *Parikh vector* of a word is defined as the function $p : \Sigma^* \to \mathbb{N}^k$, given by $p(w) = (\#a_1(w), \#a_2(w), \ldots, \#a_k(w))$, where $\#a_i(w)$ gives the number of occurrences of the letter a_i in the word w. Further, for a language L, $p(L) = \{p(w) | w \in L\}$

A subset of $\mathbb{N}^k$ is said to be *linear* if it is of the form

$u_0 + \langle u_1, \ldots, u_m \rangle = \{u_0 + a_1 u_1 + \ldots + a_m u_m | a_1, \ldots, a_m \in \mathbb{N}\}$ for some vectors $u_0, \ldots, u_m$.

A subset of $\mathbb{N}^k$ is said to be *semi-linear* if it is a union of finitely many linear subsets.

Cantor's theorem

Cantor's theorem states that, for any setA, the set of all subsets of A (the power set of A) has a strictly greater cardinality than A itself. For finite sets, Cantor's theorem can be seen to be true by a much simpler proof than that given below, since in addition to subsets of A with just one member, there are others as well, and since n < 2n for all natural numbers n. But the theorem is true of infinite sets as well. In particular, the power set of a countably infinite set is uncountably infinite.

Proof:

Two sets are equinumerous (have the same cardinality) if and only if there exists a one-to-one correspondence between them. To establish Cantor's theorem it is enough to show that, for any given set A, no function f from A into $P(A)$, the power set of A, can be surjective, i.e. to show the existence of at least one subset of A that is not an element of the image of A under f. Such a subset, $B \in P(A)$, is given by the following construction:

$$B = \{x \in A : x \notin f(x)\}.$$

This means, by definition, that for all x in A, $x \in B$ if and only if $x \notin f(x)$. For all x the sets B and $f(x)$ cannot be the same because B was constructed from elements of A whose images (under f) did not include themselves. More specifically, consider any $x \in A$, then either $x \in f(x)$ or $x \notin f(x)$. In the former case, $f(x)$ cannot equal B because $x \in f(x)$ by assumption and $x \notin B$ by the construction of B. In the latter case, $f(x)$ cannot equal B because $x \notin f(x)$ by assumption and $x \in B$ by the construction of B.

Thus there is no x such that $f(x) = B$; in other words, B is not in the image of f. Because B is in the power set of A, the power set of A has a greater cardinality than A itself.

Another way to think of the proof is that B, empty or non-empty, is always in the power set of A. For f to be onto, some element of A must map to B. But that leads to a contradiction: no element of B can map to B because that would contradict the criterion of membership in B, thus the element mapping to B must not be an element of B meaning that it satisfies the criterion for membership in B, another contradiction. So the assumption that an element of A maps to B must be false; and f can not be onto.

Because of the double occurrence of x in the expression "$x \notin f(x)$", this is a diagonal argument.

Godel Numbering:

In mathematical logic, a Godel numbering is a function that assigns to each symbol and well-formed formula of some formal language a unique natural number, called its Gödel number. The concept was famously used by Kurt Godel for the proof of his incompleteness theorems. (Gödel 1931)

A Gödel numbering can be interpreted as an encoding in which a number is assigned to each symbol of a mathematical notation, after which a sequence of natural numbers can then represent a sequence of strings. These sequences of natural numbers can again be represented by single natural numbers, facilitating their manipulation in formal theories of arithmetic.

Since Godel's paper was published in 1931, the term "Godel numbering" or "Godel code" has been used to refer to more general assignments of natural numbers to mathematical objects.

Gödel used a system based on prime factorization. He first assigned a unique natural number to each basic symbol in the formal language of arithmetic with which he was dealing.

To encode an entire formula, which is a sequence of symbols, Gödel used the following system. Given a sequence $(x_1, x_2, x_3, ..., x_n)$ of positive integers, the Gödel encoding of the sequence is the product of the first *n* primes raised to their corresponding values in the sequence:

$$\operatorname{enc}(x_1, x_2, x_3, \ldots, x_n) = 2^{x_1} \cdot 3^{x_2} \cdot 5^{x_3} \cdots p_n^{x_n}.$$

According to the fundamental theorem of arithmetic, any number obtained in this way can be uniquely factored into prime factors, so it is possible to recover the original sequence from its Gödel number (for any given number n of symbols to be encoded).

Gödel specifically used this scheme at two levels: first, to encode sequences of symbols representing formulas, and second, to encode sequences of formulas representing proofs. This allowed him to show a correspondence between statements about natural numbers and statements about the provability of theorems about natural numbers, the key observation of the proof.

Example:

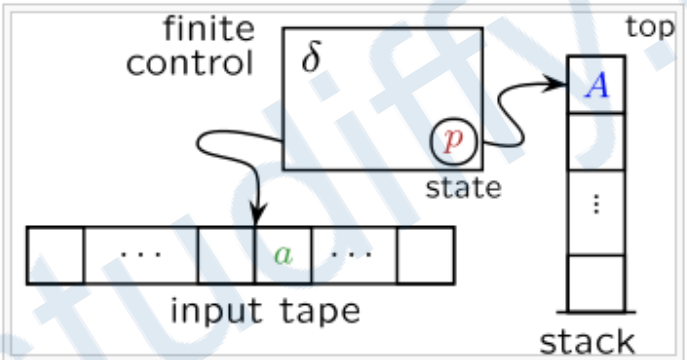
In the specific Gödel numbering used by Nagel and Newman, the Gödel number for the symbol "0" is 6 and the Gödel number for the symbol "=" is 5. Thus, in their system, the Gödel number of the formula "0 = 0" is $2^6 \times 3^5 \times 5^6 = 243{,}000{,}000$.

Pushdown automaton (PDA)

- In computer science, a pushdown automaton (PDA) is a type of automaton that uses a stack for temporary data storage.
- The PDA is used in theories about what can be computed by machines. The PDA is more capable than finite-state machines but less capable than Turing machines. Because its input can be described with a formal grammar, it can be used in parser design. There are two classes of PDAs:
- In deterministic PDAs, the machine has only one possible choice of action for all situations. Their application is limited to deterministic context-free grammars.
- In nondeterministic PDAs, the automaton can have two or more possible choices of action for some or all situations. The choices may or may not be mutually exclusive.
- When they are not, the automaton will create branches, each following one of the correct choices. If more than one of the branches created during the execution of the automaton complete successfully multiple outputs will be produced. This kind of PDAs can handle all context-free grammars.

Operation

- Pushdown automata differ from finite state machines in two ways:
- They can use the top of the stack to decide which transition to take.
- They can manipulate the stack as part of performing a transition.
- Pushdown automata choose a transition by indexing a table by input signal, current state, and the symbol at the top of the stack. This means that those three parameters completely determine the transition path that is chosen. Finite state machines just look at the input signal and the current state: they have no stack to work with. Pushdown automata add the stack as a parameter for choice.



a diagram of the pushdown automaton

- Pushdown automata can also manipulate the stack, as part of performing a transition. Finite state machines choose a new state, the result of following the transition. The manipulation can be to push a particular symbol to the top of the stack, or to pop off the top of the stack. The automaton can alternatively ignore the stack, and leave it as it is. The choice of manipulation (or no manipulation) is determined by the transition table.
- Put together: Given an input signal, current state, and stack symbol, the automaton can follow a transition to another state, and optionally manipulate (push or pop) the stack.
- In general, pushdown automata may have several computations on a given input string, some of which may be halting in accepting configurations while others are not. Thus we have a model which is technically known as a "nondeterministic pushdown automaton" (NDPDA or NPDA).

- Nondeterminism means that there may be more than just one transition available to follow, given an input signal, state, and stack symbol. If in every situation only one transition is available as continuation of the computation, then the result is a deterministic pushdown automaton (DPDA), a strictly weaker device. Unlike finite-state machines, there is no mechanical way to turn a NDPDA into an equivalent DPDA.
- If we allow a finite automaton access to two stacks instead of just one, we obtain a more powerful device, equivalent in power to a Turing machine. A linear bounded automaton is a device which is more powerful than a pushdown automaton but less so than a Turing machine.
- Nondeterministic pushdown automata are equivalent to context-free grammars: for every context-free grammar, there exists a pushdown automaton such that the language generated by the grammar is identical with the language generated by the automaton, which is easy to prove. The reverse is true, though harder to prove: for every pushdown automaton there exists a context-free grammar such that the language generated by the automaton is identical with the language generated by the grammar.

A PDA is formally defined as a 7-tuple:

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

- Q is a finite set of *states*
- Σ is a finite set which is called the *input alphabet*
- Γ is a finite set which is called the *stack alphabet*
- δ is a finite subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$, the *transition relation*, where Γ^* denote the set of strings over Γ and ε denotes the empty string.
- $q_0 \in Q$ is the *start state*
- $Z \in \Gamma$ is the *initial stack symbol*
- $F \subseteq Q$ is the set of *accepting states*

An element $(p, a, A, q, \alpha) \in \delta$ is a transition of M. It has the intended meaning that M, in state $p \in Q$, with $a \in \Sigma \cup \{\varepsilon\}$ on the input and with $A \in \Gamma$ as topmost stack symbol, may read a, change the state to q, pop A, replacing it by pushing $\alpha \in \Gamma^*$. The letter ϵ (epsilon) denotes the empty string and the $(\Sigma \cup \{\varepsilon\})$ component of the transition relation is used to formalize that the PDA can either read a letter from the input, or proceed leaving the input untouched.

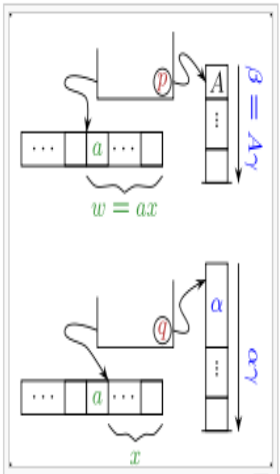
In many texts the transition relation is replaced by an (equivalent) formalization, where

- δ is the *transition function*, mapping $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ into finite subsets of $Q \times \Gamma^*$.

Here $\delta(p, a, A)$ contains all possible actions in state p with A on the stack, while reading a on the input. One writes $(q, \alpha) \in \delta(p, a, A)$ for the function precisely when $(p, a, A, q, \alpha) \in \delta$ for the relation. Note that *finite* in this definition is essential.

Computations

In order to formalize the semantics of the pushdown automaton a description of the current situation is introduced. Any 3-tuple $(p, w, \beta) \in Q \times \Sigma^* \times \Gamma^*$ is called an instantaneous description (ID) of M, which includes the current state, the part of the input tape that has not been read, and the contents of the stack (topmost symbol written first). The transition relation δ defines the step-relation $\vdash_M$ of M on instantaneous descriptions. For instruction $(p, a, A, q, \alpha) \in \delta$ there exists a step $(p, ax, A\gamma) \vdash_M (q, x, \alpha\gamma)$, for every $x \in \Sigma^*$ and every $\gamma \in \Gamma^*$.



a step of the pushdown automaton

In general pushdown automata are nondeterministic meaning that in a given instantaneous description (p, w, β) there may be several possible steps. Any of these steps can be chosen in a computation. With the above definition in each step always a single symbol (top of the stack) is popped, replacing it with as many symbols as necessary. As a consequence no step is defined when the stack is empty.

Computations of the pushdown automaton are sequences of steps. The computation starts in the initial state q_0 with the initial stack symbol Z on the stack, and a string w on the input tape, thus with initial description (q_0, w, Z). There are two modes of accepting. The pushdown automaton either accepts by final state, which means after reading its input the automaton reaches an accepting state (in F), or it accepts by empty stack (ε), which means after reading its input the automaton empties its stack. The first acceptance mode uses the internal memory (state), the second the external memory (stack).

Formally one defines

1. $L(M) = \{w \in \Sigma^* | (q_0, w, Z) \vdash_M^* (f, \varepsilon, \gamma) \text{ with } f \in F \text{ and } \gamma \in \Gamma^*\}$ (final state)
2. $N(M) = \{w \in \Sigma^* | (q_0, w, Z) \vdash_M^* (q, \varepsilon, \varepsilon) \text{ with } q \in Q\}$ (empty stack)

Here $\vdash_M^*$ represents the reflexive and transitive closure of the step relation $\vdash_M$ meaning any number of consecutive steps (zero, one or more).

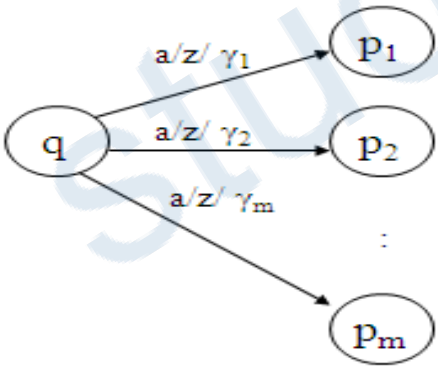
For each single pushdown automaton these two languages need to have no relation: they may be equal but usually this is not the case. A specification of the automaton should also include the intended mode of acceptance. Taken over all pushdown automata both acceptance conditions define the same family of languages.

PDA Transitions:

Two types of PDA transitions:
First:

$\delta(q, a, z) = \{(p1,\gamma1), (p2,\gamma2),\ldots, (pm,\gamma m)\}$

- Current state is q
- Current input symbol is a
- Symbol currently on top of the stack z
- Move to state pi from q
- Replace z with γi on the stack (leftmost symbol on top)
- Move the input head to the next input symbol



Second:

$\delta(q, \varepsilon, z) = \{(p1,\gamma1), (p2,\gamma2),\ldots, (pm,\gamma m)\}$

- Current state is q
- Current input symbol is not considered
- Symbol currently on top of the stack z
- Move to state pi from q
- Replace z with γi on the stack (leftmost symbol on top)
- No input symbol is read

Example 1: (balanced parentheses)

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

δ:

(1) $\delta(q_1, (, \#) = \{(q_1, L\#)\}$

(2) $\delta(q_1,), \#) = \emptyset$ // illegal, string rejected

(3) $\delta(q_1, (, L) = \{(q_1, LL)\}$

(4) $\delta(q_1,), L) = \{(q_1, \varepsilon)\}$

(5) $\delta(q_1, \varepsilon, \#) = \{(q_1, \varepsilon)\}$ // if no character read, & stack hits bottom

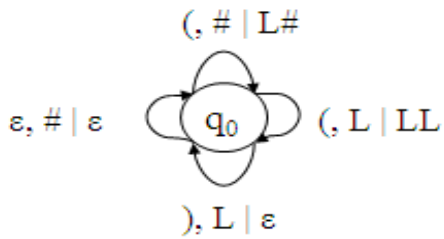
(6) $\delta(q_1, \varepsilon, L) = \emptyset$ // illegal, string rejected

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.

Transition Diagram:



Example Computation:

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

Example 2: For the language {x | x = wcw$ and w in {0,1}*, but sigma={0,1,c}}

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

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)	$\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$	(12)	$\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$
(8)	$\delta(q_2, \varepsilon, R) = \{(q_2, \varepsilon)\}$		

Example Computation:

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

State	Input	Stack	Rule Applied	Rules Applicable
q_1	**0**1c10	R		(1)
q_1	**1**c10	BR	(1)	(10)
q_1	**c**10	GBR	(10)	(6)
q_2	**1**0	GBR	(6)	(12)
q_2	**0**	BR	(12)	(7)
q_2	**ε**	R	(7)	(8)
q_2	ε	ε	(8)	-

Example Computation:

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

State	Input	Stack	Rule Applied
q_1	**1**c1	R	
q_1	**c**1	GR	(9)
q_2	**1**	GR	(6)
q_2	**ε**	R	(12)
q_2	ε	ε	(8)

Example 3 : For the language $\{x \mid x = ww^r \text{ and } w \text{ in } \{0,1\}^*\}$

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

δ:

(1) $\delta(q_1, 0, R) = \{(q_1, BR)\}$

(2) $\delta(q_1, 1, R) = \{(q_1, GR)\}$

(3) $\delta(q_1, 0, B) = \{(q_1, BB), (q_2, \varepsilon)\}$

(4) $\delta(q_1, 0, G) = \{(q_1, BG)\}$

(5) $\delta(q_1, 1, B) = \{(q_1, GB)\}$

(6) $\delta(q_1, 1, G) = \{(q_1, GG), (q_2, \varepsilon)\}$

(7) $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$

(8) $\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$

(9) $\delta(q_1, \varepsilon, R) = \{(q_2, \varepsilon)\}$

(10) $\delta(q_2, \varepsilon, R) = \{(q_2, \varepsilon)\}$

Example Computation:

(1) $\delta(q_1, 0, R) = \{(q_1, BR)\}$

(2) $\delta(q_1, 1, R) = \{(q_1, GR)\}$

(3) $\delta(q_1, 0, B) = \{(q_1, BB), (q_2, \varepsilon)\}$

(4) $\delta(q_1, 0, G) = \{(q_1, BG)\}$

(5) $\delta(q_1, 1, B) = \{(q_1, GB)\}$

(6) $\delta(q_1, 1, G) = \{(q_1, GG), (q_2, \varepsilon)\}$

(7) $\delta(q_2, 0, B) = \{(q_2, \varepsilon)\}$

(8) $\delta(q_2, 1, G) = \{(q_2, \varepsilon)\}$

(9) $\delta(q_1, \varepsilon, R) = \{(q_2, \varepsilon)\}$

(10) $\delta(q_2, \varepsilon, R) = \{(q_2, \varepsilon)\}$

State	Input	Stack	Rule Applied	Rules Applicable
q_1	000000	R		(1), (9)
q_1	00000	BR	(1)	(3), both options
q_1	0000	BBR	(3) option #1	(3), both options
q_1	000	BBBR	(3) option #1	(3), both options
q_2	00	BBR	(3) option #2	(7)
q_2	0	BR	(7)	(7)
q_2	ε	R	(7)	(10)
q_2	ε	ε	(10)	

Example Computation:

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

State	Input	Stack	Rule Applied	
q_1	010010	R		
q_1	10010	BR	(1)	From (1) and (9)
q_1	0010	GBR	(5)	
q_1	010	BGBR	(4)	
q_2	10	GBR	(3) option #2	
q_2	0	BR	(8)	
q_2	ε	R	(7)	
q_2	ε	ε	(10)	

Theorem : 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, Σ, Γ, δ, q, z, Ø) where:

Q = {q}
Σ = T
Γ = V
z = S

δ: for all a in Σ and A in Γ, δ(q, a, A) contains (q, γ) if A –> aγ is in P or rather:
δ(q, a, A) = {(q, γ) | A –> aγ is in P and γ is in Γ*}, 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 4 :

Construct pushdown automata for the following languages. Acceptance either by empty stack or by final state.

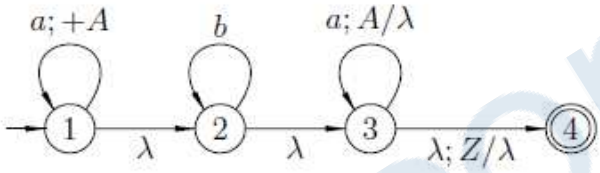
(a) $\{ a^n b^m a^n \mid m, n \in \mathbb{N} \}$

(b) $\{a^i b^j c^k \mid i, j, k \in \mathbb{N}, i + k = j\}$

(c) $\{ a^n b^m \mid n \leq m \leq 2n \}$

Answer:

The pda is depicted by the following diagram. Formally, it consists of the following components: state set $Q = \{1, 2, 3, 4\}$, alphabet $\Sigma = \{a, b\}$, stack alphabet $\Gamma = \{Z, A\}$, initial state 1, final state 4, initial stack symbol Z, and instructions $(1, a, X, 1, AX)$, $(1, \lambda, X, 2, X)$, $(2, b, X, 2, X)$, $(2, \lambda, X, 3, X)$ $(3, a, A, 3, \lambda)$, $(3, \lambda, Z, 3, \lambda)$, for all $X \in \Gamma$. Acceptance either by final state or by empty stack.

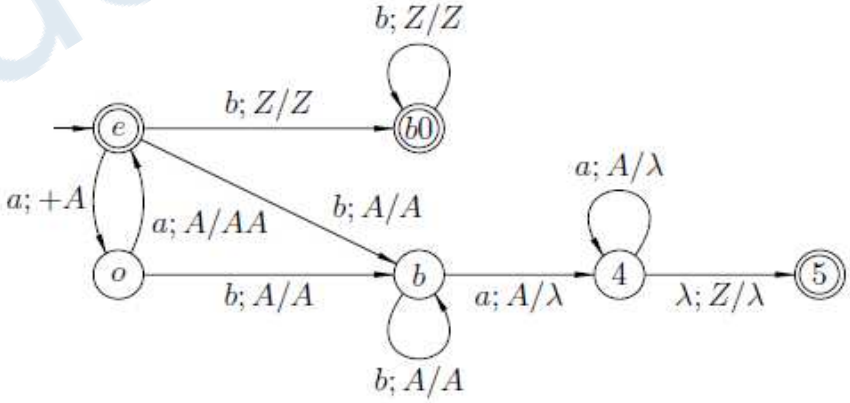


Alternatively, we build a cfg: $S \to aSA$, $S \to T$, $T \to bT$, $T \to \lambda$, $A \to a$.

This translates into the single state pda with instructions (p, a, S, p, SA), (p, λ, S, p, T), (p, b, T, p, T), $(p, \lambda, T, p, \lambda)$, and (p, a, A, p, λ). Initial stack symbol S, acceptance by empty stack.

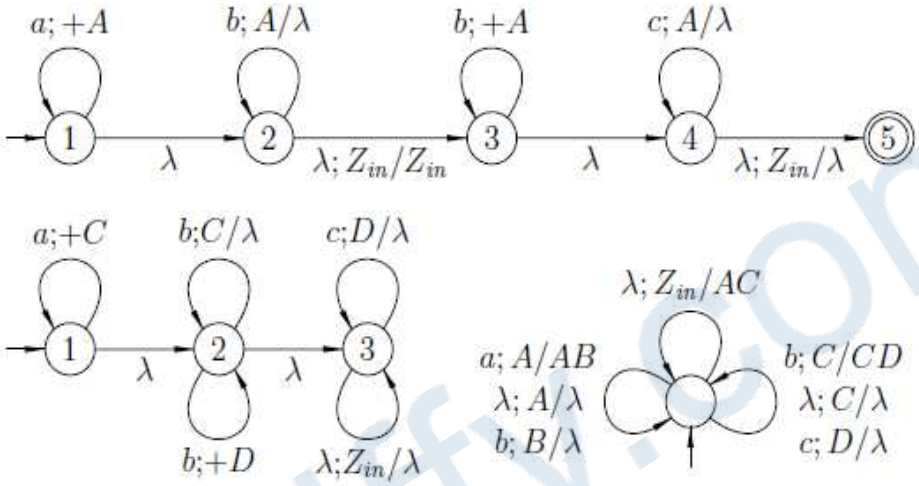
Determinism. The automata above are not deterministic. To accept deterministically we have to realize strings without b's (including λ) are to be accepted too, without recognizing (guessing) the middle of the string. However, strings without b from the regular subset $(aa)^*$. Likewise there is a 'special' subset b^*.

(a)



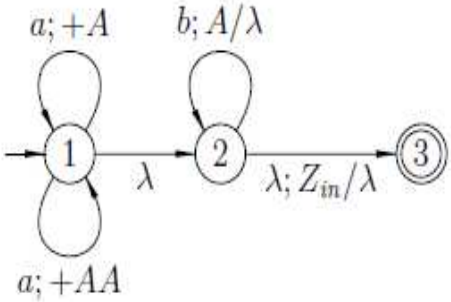
(b)

We give three different pda accepting this language. All variants use the fact that the language can be writen as $\{ a^i b^i b^j c^j \mid i, j \in \mathbb{N} \}$.



(c)

For each a we put nondeterministically one or two symbols onto the stack. Each b removes one symbol.



Parsing:

- In computer science and linguistics, **parsing**, or, more formally, **syntactic analysis**, is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar.
- Parsing is also an earlier term for the diagramming of sentences of natural languages, and is still used for the diagramming of inflected languages, such as the Romance languages or Latin.
- Parsing is a common term used in psycholinguistics when describing language comprehension.
- In this context, parsing refers to the way that human beings, rather than computers, analyze a sentence or phrase (in spoken language or text) "in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc." This term is especially common when discussing what linguistic cues help speakers to parse garden-path sentences.

Types Of Parsing

Top Down Parsing:

- Top-down parsing is a parsing strategy where one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar. LL parsers are a type of parser that uses a top-down parsing strategy.
- Top-down parsing is a strategy of analyzing unknown data relationships by hypothesizing general parse tree structures and then considering whether the known fundamental structures are compatible with the hypothesis. It occurs in the analysis of both natural languages and computer languages.
- Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse-trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.

- In top-down parsing, you start with the start symbol and apply the productions until you arrive at the desired string.
- As an example, let's trace through the two approaches on this simple grammar that recognizes strings consisting of any number of a's followed by at least one (and possibly more) b's:

S –> AB

A –>aA | ε

B –> b | bB

Here is a top-down parse of aaab. We begin with the start symbol and at each step, expand one of the remaining nonterminals by replacing it with the right side of one of its productions. We repeat until only terminals remain. The top-down parse produces a leftmost derivation of the sentence.

S	
AB	S –> AB
aAB	A –>aA
aaAB	A –>aA
aaaAB	A –>aA
aaaεB	A –>ε
aaab	B –> b

Bottom Up Parsing

A bottom-up parse works in reverse. We begin with the sentence of terminals and each step applies a production in reverse, replacing a substring that matches the right side with the nonterminal on the left. We continue until we have substituted our way back to the start symbol. If you read from the bottom to top, the bottom-up parse prints out a rightmost derivation of the sentence.

aaab	
aaaεb	(insert ε)

aaaAb	A –>ε
aaAb	A –>aA
aAb	A –>aA
Ab	A –>aA
AB	B –> b
S	S –> AB

Ogden's lemma

In the theory of formal languages, Ogden's lemma (named after William F. Ogden) provides an extension of flexibility over the pumping lemma for context-free languages.

Ogden's lemma states that if a language L is context-free, then there exists some number $p > 0$ (where p may or may not be a pumping length) such that for any string w of length at least p in L and every way of "marking" p or more of the positions in w, w can be written as

$$w = uxyzv$$

with strings u, x, y, z, and v, such that

1. xz has at least one marked position,
2. xyz has at most p marked positions, and
3. ux^iyz^iv is in L for every $i \geq 0$.

Ogden's lemma can be used to show that certain languages are not context-free, in cases where the pumping lemma for context-free languages is not sufficient. An example is the language $\{a^ib^jc^kd^l : i = 0 \text{ or } j = k = l\}$. It is also useful to prove the inherent ambiguity of some languages.

Proof:

Let b be the maximum number of symbols in the right-hand side of a rule. For any parse tree T of string z, and z has at least p marked positions. We say a leaf in T is marked if its corresponding position in z is marked. We say an internal node of T is marked if the subtrees rooted at two or more of its children each contains a marked leaf. We claim that if every (root-to-

leaf) path in T contains at most i marked internal nodes, T has at most bi marked leaves. Assume that this claim is true (which will be proved shortly by induction). To prove Ogden's lemma, we set $p = b^{|V|+1}$. Then, the minimum marked internal nodes in a path is |V | + 1. By pigeonhole principle, there exists some variable appearing at least twice on that path. Then, by a similar way of proving the original pumping lemma, we can show that z can be written as uvwxy satisfying Ogden's lemma. We now go back to prove the claim. The claim is true for i = 0, since if every path inT has at most 0 marked nodes, T has no marked nodes. Thus, there must be at most b0 = 1 marked leaves (why?). For i ≥1, let q be the unique marked internal node whose ancestors (if exist) are not marked. Then, the number of marked leaves in T is equal to the total number of marked leaves under the children of q. Also, as q is a marked node, in the subtree rooted at any child of q, every path has at most i − 1 marked nodes. By induction, every subtree has at most b^{i-1} marked leaves. As q has at most b children, T thus has at most bi marked leaves.

C(Cocke)Y(Younger)K(Kasami) Algorithm:

- The Cocke–Younger–Kasami (CYK) algorithm (alternatively called CKY) is a parsingalgorithm for context-free grammars. It employs bottom-up parsing and dynamic programming.
- The standard version of CYK operates only on context-free grammars given in Chomsky normal form (CNF). However any context-free grammar may be transformed to a CNF grammar expressing the same language .
- The importance of the CYK algorithm stems from its high efficiency in certain situations.
- The algorithm requires the context-free grammar to be rendered into Chomsky normal form (CNF), because it tests for possibilities to split the current sequence in half. Any context-free grammar that does not generate the empty string can be represented in CNF using onlyproduction rules of the forms . $A \to \alpha$ and $A \to BC$.

  Algorithm:

```
let the input be a string S consisting of n characters: a_1 ... a_n.
let the grammar contain r nonterminal symbols R_1 ... R_r.
This grammar contains the subset R_s which is the set of start symbols.
let P[n,n,r] be an array of booleans. Initialize all elements of P to false.
for each i = 1 to n
  for each unit production R_j -> a_i
    set P[i,1,j] = true
for each i = 2 to n -- Length of span
  for each j = 1 to n-i+1 -- Start of span
    for each k = 1 to i-1 -- Partition of span
      for each production R_A -> R_B R_C
        if P[j,k,B] and P[j+k,i-k,C] then set P[j,i,A] = true
if any of P[1,n,x] is true (x is iterated over the set s, where s are all the indices for R_s) then
  S is member of language
else
  S is not member of language
```

Example:

This is an example grammar:

$$
\begin{aligned}
S &\rightarrow NP\ VP \\
VP &\rightarrow VP\ PP \\
VP &\rightarrow V\ NP \\
VP &\rightarrow eats \\
PP &\rightarrow P\ NP \\
NP &\rightarrow Det\ N \\
NP &\rightarrow she \\
V &\rightarrow eats \\
P &\rightarrow with \\
N &\rightarrow fish \\
N &\rightarrow fork \\
Det &\rightarrow a
\end{aligned}
$$

Now the sentence *she eats a fish with a fork* is analyzed using the CYK algorithm. In the following table, in $P[i, j, k]$, i is the number of the column (starting at the left at 1), and j is the number of the row (starting at the bottom at 1).

CYK table

S						
	VP					
S						
	VP			PP		
S		NP			NP	
NP	V, VP	Det.	N	P	Det	N
she	eats	a	fish	with	a	fork

Turing Machine:

A Turing machine is a device that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computeralgorithm, and is particularly useful in explaining the functions of a CPU inside a computer.

The "Turing" machine was described in 1936 by Alan Turingwho called it an "a-machine" (automatic machine). The Turing machine is not intended as practical computing technology, but

rather as a hypothetical device representing a computing machine. Turing machines help computer scientists understand the limits of mechanical computation.

A Turing machine that is able to simulate any other Turing machine is called a universal Turing machine (UTM, or simply a universal machine). A more mathematically oriented definition with a similar "universal" nature was introduced by Alonzo Church, whose work on lambda calculus intertwined with Turing's in a formal theory of computation known as the Church–Turing thesis. The thesis states that Turing machines indeed capture the informal notion of effective method in logic and mathematics, and provide a precise definition of an algorithm or 'mechanical procedure'.

A Turing machine consists of:

- A **tape** which is divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special *blank* symbol (here written as 'B') and one or more other symbols. The tape is assumed to be arbitrarily extendable to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written to before are assumed to be filled with the blank symbol. In some models the tape has a left end marked with a special symbol; the tape extends or is indefinitely extensible to the right.
- A **head** that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time. In some models the head moves and the tape is stationary.
- A **state register** that stores the state of the Turing machine, one of finitely many. There is one special *start state* with which the state register is initialized. These states, writes Turing, replace the "state of mind" a person performing computations would ordinarily be in.
- A finite **table** (occasionally called an **action table** or **transition function**) of instructions (usually quintuples [5-tuples] : $q_i a_j \rightarrow q_{i1} a_{j1} d_k$, but sometimes 4-tuples) that, given the *state*(q_i) the machine is currently in *and* the *symbol*(a_j) it is reading on the tape (symbol currently under the head) tells the machine to do the following in sequence (for the 5-tuple models):

- Either erase or write a symbol (replacing a_j with a_{j1}), *and then*
- Move the head (which is described by d_k and can have values: 'L' for one step left *or* 'R' for one step right *or* 'N' for staying in the same place), *and then*
- Assume the same or a *new state* as prescribed (go to state q_{i1}).
- Formally a Turing machine can be defined as follows. $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$ where

- Q is a finite, non-empty set of *states*
- Γ is a finite, non-empty set of the *tape alphabet/symbols*
- $b \in \Gamma$ is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation)
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of *input symbols*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *final* or *accepting states*.
- $\delta : Q \setminus F \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a partial function called the *transition function*, where L is left shift, R is right shift. (A relatively uncommon variant allows "no shift", say N, as a third element of the latter set.)

Decision problems, optimization problems

Hamiltonian cycle in a graph G = (V, E): a cycle that contains each of the vertices in V (exactly once)

Traveling salesman problem (TSP):

Weighted graph G = (V, E, w: E ->Reals), V = {1, .., n}, E = { (i, j) | i < j } (often the complete graph Kn).Weight (or length) of a path or cycle = sum of the weights of its edges.

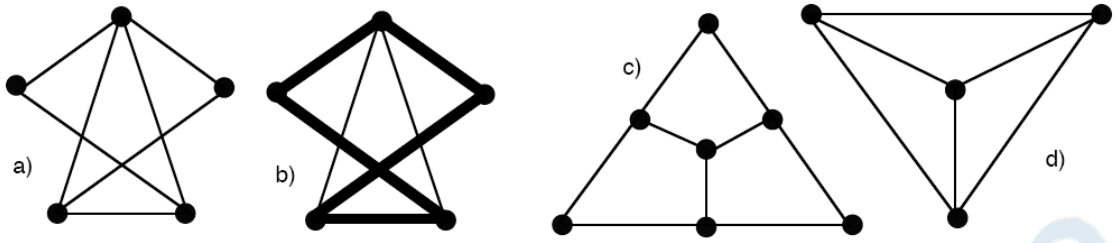
Given G= (V, E, w) find a shortest Hamiltonian cycles, if any exist.

A clique in a graph G = (V, E): a subset V’ □ V such that for all u, v □ V’, (u, v) □ E.

|V’| is the size of the clique. A clique of size k is called a k-clique.

Clique problems: Given G= (V, E), answer questions about the existence of cliques, find maximumclique, enumerate all cliques.

Examples:



Ex: the graph a) shown at left has exactly 1 Hamiltonian cycle, highlighted in b). The graph c) has none. Thecomplete graph K4 has 3 Hamiltonian cycles. d) The complete graph Kn has (n-1)! / 2 Hamiltonian cycles

Ex: Graph a) has three maximum 3-cliques. The only cliques in graph c) are the vertices and the edges, i.e. the1-cliques and the 2-cliques. Graph d) is a 4-clique, and every subset of its vertices is a clique.

Decision problems:

• Given G, does G have a Hamiltonian cycle? Given G and k, does G have a k-clique?

Given G = (V, E, w) and a real number B, does G have a Hamiltonian cycle of length ≤ B?

Finding the answer to a decision problem is often hard, whereas verifying a positive answer is often easy: weare shown an object and merely have to verify that it meets the specifications (e.g. trace the cycle shown in b).Decision problems are naturally formalized in terms of machines accepting languages, as follows: probleminstances (e.g. graphs) are coded as strings, and the code words of all instances that have the answer YES (e.g.have a Hamiltonian cycle) form the language to be accepted.

Optimization problems:

• Given G, construct a maximum clique.

• TSP: Given Kn = (V, E, w) find a Hamiltonian cycle of minimal total length.

Both problems, of finding the answer and verifying it, are usually hard. If I claim to show you a maximumclique, and it contains k vertices, how do you verify that I haven't missed a bigger one? Do you have toenumerate all the subsets of k+1 vertices to be sure that there is no (k+1)-clique? Nevertheless, verifying isusually easier than finding a solution, because the claimed solution provides a bound that eliminates manysuboptimal candidates.

Enumeration problems:

• Given G, construct all Hamiltonian cycles, or all cliques, or all maximum cliques.

Enumeration problems are solved by exhaustive search techniques such as backtrack. They are time consumingbut often conceptually simple, except when the objects must be enumerated in a prescribed order. Enumerationis the technique of last resort for solving decision problems or optimization problems that admit no efficientalgorithms, or for which no efficient algorithm is known. It is an expensive technique, since the number ofobjects to be examined often grows exponentially with the length of their description.

Theorem

The class P of problems solvable in polynomial time

Practically all standard combinatorial algorithms presented in a course on Algorithms and Data Structures runin polynomial time. They are sequential algorithms that terminate after a number of computational steps that isbounded by some polynomial p(n) in the size n of the input data, as measured by the number of data items thatdefine the problem. A computational step is any operation that takes constant time, i.e. time independent of n.

In practical algorithm analysis there is a fair amount of leeway in the definition of "computational step" and"data item". For example, an integer may be considered a single data item, regardless of its magnitude, and anyarithmetic operation on integers as a single step. This is reasonable when we know a priori that all numbersgenerated are bounded by some integer "maxint", and is unreasonable for computations that generate numbersof unbounded magnitude.

In complexity theory based on Turing machines the definition is clear: a computational step is a transitionexecuted, a data item is a character of the alphabet, read or written on a square of tape. The alphabet is usuallychosen to be {0, 1} and the size of data is measured in bits. When studying the class P of problems solvable inpolynomial time, we only consider deterministic TMs that halt on all inputs.

Let tM: A* -> Integers be the number of steps executed by M on input x □ A*.

This chapter deals with TMs whose running time is bounded by some polynomial in the length of the input.

TM M is or runs in polynomial time iff □ polynomial p such □x □ A*: tM(x) ≤ p(|x|)

P = { L □ A* | □ TM M, □ polynomial p such that L = L(M) and □x □ A*: tM (x) ≤ p(|x|) }

Notice that we do not specify the precise version of TM to be used, in particular the number of tapes of M isleft open. This may be surprising in view of the fact that a multi-tape TM is much faster than a single-tape TM.

A detailed analysis shows that “much faster” is polynomially bounded: a single-tape TM S can simulate anymulti-tape TM M with at most a polynomial slow-down: for any multi-tape TM M there is a single-tape TM Sand a polynomial p such that for all x □ A*, tS (x) ≤ p(tM (x)). This simulation property, and the fact thata polynomial of a polynomial is again a polynomial, makes the definition of the class P extremely robust.

The question arises whether the generous accounting that ignores polynomial speed-ups or slow-downs is ofpractical relevance. After all, these are much greater differences than ignoring constant factors as one doesroutinely in asymptotics. The answer is a definite YES, based on several considerations:

1) Practical computing uses random access memory, not sequential storage such as tapes. Thus, the issue ofhow many tapes are used does not even arise. The theory is formulated in terms of TMs, rather than morerealistic models of computation such as conventional programming languages, for the sake of mathematicalprecision. And it turns out that the slowness of tape manipulation gets absorbed, in comparison with a RAM model, by the polynomial transformations we ignore so generously.

2) Most practical algorithms are of low degree, such as O(n), O(n log n), O(n2), or O(n3). Low-degreepolynomials grow slowly enough that the corresponding algorithms are computationally feasible for manyvalues of n that occur in practice. E.g. for n = 1000, n3 = 109 is a number of moderate size when compared toprocessor clock rates of 1 GHz and memories of 1 GByte. Polynomial growth rates are exceedingly slowcompared to exponential growth (consider 21000).

3) This complexity theory, like any theory at all, is a model that mirrors some aspects of reality well, and otherspoorly. It is the responsibility of the programmer or algorithm designer to determine in each specific case,whether or not an algoritm “in P” is practical or not.

Examples of problems (perhaps?) in P:

1) Every context-free language is in P. In Ch5 we saw an O(n3) parsing algorithm that solves the wordproblem for CFLs, where n is the length of the input string.

2) The complexity of problems that involve integers depends on the representation. Fortunately, with the usualradix representation, the choice of radix r > 1 is immaterial (why?). But if we choose an exotic notation,everything might change. For example, if integers were given as a list of their prime factors, many arithmeticproblems would become easier. Paradoxically, if integers were given in the unwieldy unary notation, somethings might also become “easier” according to the measure of this chapter. This is because the length of theunary representation < k >1 of k is exponential in the length of the radix r ≥ 2 representation < k > r. Given anexponentially longer input, a polynomial-time TM is allowed to use an exponentially longer computation timeas compared to the “same” problem given in the form of a concise input.

The following example illustrates the fact that the complexity of arithmetic problems depends on the numberrepresentation chosen. The assumed difficulty of factoring an integer lies at the core of modern cryptography.Factoring algorithms known today require, in the worst case, time exponential in the length, i.e. number of bits,of the radix representation of the number to be factored. But there is no proof that factoring is NP-hard -according to today’s knowledge, there might exist polynomial-time factoring algorithms. This possibilitygained plausibility when it was proven that primality, i.e. the problem of determining whether a natural number(represented in radix notation) is prime or composite.

Theorem :

The class NP of problems solvable in non-deterministic polynomial time

“NP” stands for “non-deterministic polynomial”. It is instructive to introduce two different but equivalentdefinitions of NP, because each definition highlights a different key aspect of NP. The original definitionexplicitly introduces non-deterministic TMs:

NP = { L □ A* | □ NTM N, □ polynomial p such that L = L(N) and □x □ A*: tN (x) ≤ p(|x|) }

Notice that this differs from the definition of P only in the single letter “N” in the phrase “ □ NTM N ..”,indicating that we mean non-deterministic Turing machines. We had seen that deterministic and nondeterministicTMs are equally powerful in the presence of unbounded resources of time and memory. But thereis a huge difference in terms of the time they take for certain computations. A NTM pursues simultaneously anumber of computation paths that can grow exponentially with the length of the computation.Because of many computation paths

pursued simultaneously we must redefine the function tN: A* -> Integersthat measures the number of steps executed. An input x □ A* may be accepted along a short path as well as along a long path, and some other paths may not terminate. Therefore we define tN(x) as the minimumnumber of steps executed along any accepting path for x.

Whereas the original definition of NP in terms of non-deterministic TMs has the intuitive interpretation viaparallel computation, an equivalent definition based on deterministic TMs is perhaps technically simpler tohandle. The fact that these two definitions are equivalent provides two different ways of looking at NP.The motivation for this second definition of NP comes from the observation that it may be difficult to decidewhether a string meeting certain specifications exists; but that it is often easier to decide whether or not a givenstring meets the specifications. In other words, finding a solution, or merely determining whether a solution
exists, is harder than checking a proposed solution's correctness. The fact that this intuitive argument leads to arigorous definition of NP is surprising and useful!

In the following definition, the language L describes a problem class, e.g. all graphs with a desired property ;the string w describes a problem instance, e.g. a specific graph G; the string c = c(w), called a "certificate forw" or a witness, plays the role of a key that "unlocks w": the pair w, c is easier to check than w alone!

Example: for the problems of Hamiltonian cycles and cliques, w = <G> is the representationof graph G, and the certificate c is the representation of a cycle or a clique, respectively. Given c, it is easy toverify that c represents a cycle or a clique in G.

a verifier for L □ A* is a deterministic TM V with the property thatL = { w | □ c(w) □ A* such that V accepts < w, c> }The string c = c(w) is called a certificate for w's membership in L. <w, c> denotes a representation of the pair(w, c) as a single string, e.g. w#c, where a reserved symbol # separates w from its certificate. The idea behindthis concept of certificate is that it is easy to verify w □ L if you are given w's certificate c. If not, you wouldhave to try all strings in the hope of finding the right vertificate, a process that may not terminate. We formalizethe phrase "easy to verify" by requiring that a verifier V is (or runs in) polynomial-time.

Polynomial time reducibility, NP-hard, NP-complete

Df: A function f: A* -> A* is polynomial-time computable iff there is a polynomial p and a DTM M which,when started with w on its tape, halts with f(w) on its tape, and tM(w) ≤ p(|w|).

L is ***polynomial-time reducible*** to L', denoted by L ≤p L' iffthere is polynomial-time computable f: A* -> A* such that □w □ A*, w □ L ifff(w) □ L'.

In other words, the question w □ L?can be answered by deciding f(w) □ L'. Thus, the complexity of decidingmembership in L is at most the complexity of evaluating f plus the complexity of deciding membership in L'.

Since we generously ignore polynomial times, this justifies the notation L ≤p L' .

A remarkable fact makes the theory of NP interesting and rich. With respect to the class NP and the notion ofpolynomial-time reducibility, there are "hardest problems" in the sense that all decision problems in NP arereducible to any one of these"hardest problems".

L' is ***NP-hard***iff □L □ NP, L ≤p L'

L' is ***NP-complete***iff L' □ NP and L' is NP-hard

Theorem :

Satisfiability of Boolean expressions (SAT) is NP-complete

SAT = satisfiability of Boolean expressions: given an arbitrary Boolean expression E over variablesx1, x2, .. , xd, is there an assignment of truth values to x1, x2, ..that makes E true?

It does not matter what Boolean operators occur, conventionally one considers And □, Or □, Not ¬.SAT is the prototypical "hard problem". I.e. the problem that is generally proven to be NP-complete "fromscratch", whereafter all other problems to be proven NP-complete are reduced to SAT. The theory of NPcompletenessbegan with the key theorem (Cook 1971): SAT is NP-complete.Given this central role of SAT it is useful to develop an intuitive understanding of the nature of the problem,including details of measurement and "easy" versions of SAT.

1) It does not matter what Boolean operators occur, conventionally one considers And □, Or □, Not ¬. Specialforms, eg CNF

2) The length n of E can be measured by the number of characters of E. It is more convenient, however, to firsteliminate "Not-chains" ¬¬¬ ..using the identity ¬ ¬ x = x, and to measure the length n of E by thenumber n of occurences of variables (d denotes the number of distinct variables x1, x2, .. xd in E, d ≤ n). It isconvenient to avoid mentioning the unary operator ¬ explicitly by introducing, for each variable x, its negation¬ x as a dependent variable. A variable and its negation are called literals. Thus, an expression E with d
variables has 2d distinct literals that may occur in E.

3) Any Boolean expression E can be evaluated in linear time. Given truth values for x1, x2, ..xd, the noccurrences of literals are the leaves of a binary tree with n -1 internal nodes, each of which represents a binaryBoolean operator. The bottom-up evaluation of this tree requires n-1 operations.

4) Satisfiability of expressions over a constant number d of distinct variables can be decided in linear time. Bytrying all 2d assignments of truth values to the variables, SAT can be decided in time O(2d n), a bound that islinear in n and exponential in d. If we consider d constant, 2d is also a constant - hence this version of thesatisfiability problem can be solved in linear time! This argument shows that, if SAT turns out to be a difficultproblem, this is due to an unbounded growth of the number of distinct variables. Indeed, if d growsproportionately to n, the argument above yields an exponential upper bound of O(2n).

5) In order to express SAT as a language, choose a suitable alphabet A and a coding scheme that assigns to anyexpression E a word code(E) □ A*, and define: SAT = { code(E) | E is a satisfiable Boolean expression }

Theorem :

3-CNF SAT is NP-complete

Pf idea: reduce SAT to 3-CNF SAT, SAT ≤p 3-CNF SAT. To any Boolean expression E we assign inpolynomial time a 3-CNF expression F that is equivalent in the weak sense that either both E and F aresatisfiable, or neither is. Notice that E and F need not be equivalent as Boolean expressions, i.e. they need notrepresent the same function! They merely behave the same w.r.t. satisfiability.Given E, we construct F in 4 steps, illustrated using the example E = ¬ (¬ x □ (y □ z))

1) Use de Morgan's law to push negations to the leaves of the expression tree:

E1 = x □ ¬ (y □ z) = x □ (¬ y □ ¬ z)

2) Assign a new Boolean variable to each internal node of the expression tree, i.e. to each occurrence of anoperator, and use the Boolean operator 'equivalence' □ to state the fact that this variable must be the resultof the corresponding operation: u □ (¬ y □ ¬ z) , w □ x □ u

3) Construct an expression E2 that states that the root of the expression tree must be true, traces the evaluationof the entire tree, node by node, and combines all these assertions using ANDs:

E2 = w □ (w □ (x □ u)) □ (u □ (¬ y □ ¬ z)).

E2 and E are equivalent in the weak sense of simultaneous satisfiability. If E is satisfiable, then E2 is also, bysimply assigning to the new variables u and w the result of the corresponding operation. Conversely, if E2 issatisfiable, then E is also, using the same values of the original variables x, y, z as appear in E2.Notice that E2 is in conjunctive form at the outermost level, but its subexpressions are not, so we need a lasttransformation step.

4) Recall the Boolean identity for implication: a □ b = ¬ a □ b to derive the identities:

a □ (b □ c) = (a □ ¬ b) □ (a □ ¬ c) □ (¬ a □ b □ c)

a □ (b □ c) = (¬ a □ b) □ (¬ a □ c) □ (a □ ¬ b □ ¬ c)

Using these identities on the subexpressions, E2 gets transformed into F in 3-CNF:

F = w □ (w □ ¬ x) □ (w □ ¬ u) □ (¬ w □ x □ u) □ (¬ u □ x) □ (¬ u □ z) □ (u □ y □ z)

Each of the four transformation steps can be done in linear time and lengthens the expression by at most aconstant factor. Thus, the reduction of a Boolean expression E in general form to one, F, in 3-CNF can be donein polynomial time.

Notice the critical role of the integer ’3’in 3-CNF: we need to express the result of a binary operator, such asw □ x □ u, which naturally involves three literals. Thus, it is no surprise that the technique used in the proofabove fails to work for 2-CNF. Indeed, 2-CNF is in P .

In analogy to CNF we define the disjunctive normal form DNF as an OR of terms, each of which is anAND of literals: E = T1 □ T2 □ T3 □ ... where Ti = (L1 □ L2 □ L3 □ ..)

Theorem :

CLIQUE is NP-complete

Proof: Show that 3-CNF ≤p CLIQUE. Given a 3-CNF expression F, construct a graph G = (V, E) and an integerk such that F is satisfiableiff G has a k-clique.

Let F = (z11 □ z12 □ z13) □ (z21 □ z22 □ z23) □ ... □ (zm1 □ zm2 □ zm3), where each zij is a literal.To each occurrence of a literal we assign a vertex, i.e. V = { (1,1), (1,2), (1,3), ... , (m, 1), (m, 2), (m, 3) }

We introduce an edge ((i, j) (p, q)) iff i ≠ p (the two literals are in different clauses) and$z_{ij} \neq \neg z_{pq}$(the 2 literals do not clash, i.e. both can be made true under the same assignment).

Finally, let k, the desired clique size, be = m, the number of clauses.

With this construction of G we observe that F is satisfiable via an assignment A

iff 1) each clause contains a literal that is true under A, say z1, j1, z2, j2, ... , zm, jm

iff 2) there are literals z1, j1, z2, j2, ... , zm, jm no 2 of which are negations of each other

iff 3) there are vertices (1, j1), (2, j2), ... , (m, jm) that are pairwise connected by an edge
iff 4) G has a k-clique.

Some more Examples:

Let L be the set of all strings

$$a^{n_1}ba^{n_2}b\dots a^{n_t}b \in \Sigma^* = \{a,b\}^*$$

where $t, n_1, \dots, n_t \geq 0$ are positive integers such that $n_j \neq j$ for some $j \in \{1, \dots, t\}$. (As usual, a^{n_i} denotes the strings consisting of n_i copies of the letter "a".) In this problem you will prove that L is a context free language. You can break your proof into the following subproblems:

- First prove that the language $L' = \{c^n d^m e : n+1 \neq m\}$ over the alphabet $\{c,d,e\}$ is context free giving a context free grammar that generates L'.

  Solution: The language L' is generated by the following grammar:

  $$\begin{aligned} S &\rightarrow Te|Ue \\ T &\rightarrow cTd|cT|\epsilon \\ U &\rightarrow cUd|Ud|dd \end{aligned}$$

  where T generates the set of all strings $c^n d^m$ with $m \leq n$ and U generates the set of all strings $c^n d^m$ with $m \geq n+2$.

- Then consider c, d, e as variable symbols and extend the grammar from the first part with rules for c, d, e in such a way that the new grammar generates the language L.

  Solution: We add rules to the grammar in such a way that symbols c, d, e are replaced by the languages corresponding to regular expressions a^*b, a, $b(a^*b)^*$. For clarity, we remane symbols c, d, e with the corresponding uppercase letters C, D, E. The resulting grammar is:

$$\begin{aligned} S &\rightarrow TE|UE \\ T &\rightarrow CTD|CT|\epsilon \\ U &\rightarrow CUD|UD|DD \\ C &\rightarrow aC|b \\ D &\rightarrow a \\ E &\rightarrow b|EX \\ X &\rightarrow aX|b \end{aligned}$$

The language generated by the grammar is the set of all strings obtained by concatenating n strings of the form a^*b (corresponding to the C variable symbols), a string $a^m b$ where m is an integer different from $n+1$ (corresponding to the D symbols and the first b in the string generated by E), and finally any number of strings of the form a^*b.

$G = \langle T, N, S, R \rangle$

$T = \{that,\ this,\ a,\ the,\ man,\ book,\ flight,\ meal,\ include,\ read,\ does\}$

$N = \{\text{S, NP, NOM, VP, Det, Noun, Verb, Aux}\}$

$S = S$

$R = \{$

S → NP VP	Det → *that* | *this* | *a* | *the*
S → Aux NP VP	Noun → *book* | *flight* | *meal* | *man*
S → VP	Verb → *book* | *include* | *read*
NP → Det NOM	Aux → *does*
NOM → Noun	
NOM → Noun NOM	
VP → Verb	
VP → Verb NP	

$\}$

Application of grammar rewrite rules

S → NP VP	Det → *that* | *this* | *a* | *the*
S → Aux NP VP	Noun → *book* | *flight* | *meal* | *man*
S → VP	Verb → *book* | *include* | *read*
NP → Det NOM	Aux → *does*
NOM → Noun	
NOM → Noun NOM	
VP → Verb	
VP → Verb NP	

S → NP VP
→ Det NOM VP
→ *The* NOM VP
→ *The* Noun VP
→ *The man* VP
→ *The man* Verb NP
→ *The man read* NP
→ *The man read* Det NOM
→ *The man read this* NOM
→ *The man read this* Noun
→ *The man read this book*

Church Turing Thesis:

In computability theory, the Church–Turing thesis (also known as the Turing-Church thesis, the Church–Turing conjecture, Church's thesis, Church's conjecture, and Turing's thesis) is a combined hypothesis ("thesis") about the nature of functions whose values are effectively calculable; or, in more modern terms, functions whose values are algorithmically computable. In simple terms, the Church–Turing thesis states that a function is algorithmically computable if and only if it is computable by a Turing machine.

Several attempts were made in the first half of the 20th Century to formalize the notion of computability:

- American mathematician Alonzo Church created a method for defining functions called the λ-calculus,

- British mathematician Alan Turing created a theoretical model for a machine, now called a universal Turing machine, that could carry out calculations from inputs,
- Church, along with mathematician Stephen Kleene and logician J.B. Rosser created a formal definition of a class of functions whose values could be calculated by recursion.

All three computational processes (recursion, the λ-calculus, and the Turing machine) were shown to be equivalent—all three approaches define the same class of functions. This has led mathematicians and computer scientists to believe that the concept of computability is accurately characterized by these three equivalent processes. Informally the Church–Turing thesis states that if some method (algorithm) exists to carry out a calculation, then the same calculation can also be carried out by a Turing machine (as well as by a recursively definable function, and by a λ-function).

The thesis can be stated as follows:

- Every effectively calculable function is a computable function.

Turing stated it this way:

- "It was stated ... that 'a function is effectively calculable if its values can be found by some purely mechanical process.' We may take this literally, understanding that by a purely mechanical process one which could be carried out by a machine. The development ... leads to ... an identification of computability† with effective calculability."