Advanced Theory of Computation

By

Dr. Noman Hasany

Chairman & Assoc. Prof. (*CS*), SSUET

Course Outline

- Automata theory, formal languages, Turing machines, computability theory and reducibility, computational complexity, determinism, non-determinism, time hierarchy, space hierarchy, NP completeness, selected advanced topics.

Purpose and motivation

- Theory of Computation, tries to answer the following questions:
  - What are the ***mathematical*** properties of computer hardware and software?
  - What is a ***computation*** and what is an ***algorithm***? Can we give rigorous mathematical definitions of these notions?
  - What are the ***limitations*** of computers? Can "everything" be computed? (As we will see, the answer to this question is "no".)

Purpose of this course

Develop formal mathematical models of computation that reflect real-world computers.

History

- This field of research was started by mathematicians and logicians in the 1930's, when they were trying to understand the meaning of a "computation".
- A central question asked was whether all mathematical problems can be solved in a systematic way. The research that started in those days led to computers as we know them today.

Division of Theory of Computation

- Theory of Computation can be divided into the following three areas:
  - Complexity Theory,
  - Computability Theory, and
  - Automata Theory.

Complexity Theory

The main question asked in this area is “What makes some problems computationally hard and other problems easy?”

Complexity: “easy” problems

- Informally, a problem is called “easy”, if it is efficiently solvable.
  - Examples of “easy” problems are
    (i) sorting a sequence of, say, 1,000,000 numbers,
    (ii) searching for a name in a telephone directory, and
    (iii) computing the fastest way to drive from a city to another.

Complexity: “hard” problems

- A problem is called “hard”, if it cannot be solved efficiently, or if we don’t know whether it can be solved efficiently.
  - Examples of “hard” problems are

  (i) time table scheduling for all courses at a large university according to different criteria,

  (ii) factoring a 300-digit integer into its prime factors (hard to find patterns), and

  (iii) computing a layout for chips in VLSI.

Central Question in Complexity

Classify problems according to their degree of “difficulty”. Give a rigorous proof that problems that seem to be “hard” are really “hard”.

Computability theory

- In the 1930's, Gödel, Turing, and Church discovered that some of the fundamental mathematical problems cannot be solved by a "computer". (This may sound strange, because computers were invented only in the 1940's).
- BUT…
  - The theoretical models that were proposed in order to understand solvable and unsolvable problems led to the development of real computers

Central Question in Computability

Classify problems as being solvable or unsolvable.

Automata theory

- Automata Theory deals with definitions and properties of different types of “computation models”.
- Examples of such models are:
  - Finite Automata. These are used in text processing, compilers, and hardware design.
  - Context-Free Grammars. These are used to define programming languages and in Artificial Intelligence.
  - Turing Machines. These form a simple abstract model of a “real” computer, such as your PC at home.

Central Question in Automata Theory

Do these models have the same power, or can one model solve more problems than the other?

About the course…

- **These topics form the core of computer science.**
- Is it for math lovers, and boring for others?
- No. Because of the following:
  - This course is about the fundamental capabilities and limitations of computers.
  - It is about mathematical properties of computer hardware and software (*not just mathematical principles irrelevant to computers*).

AUTOMATA THEORY

Notation

An **alphabet** Σ is a finite set (e.g., Σ = {0,1})

A **string** over Σ is a finite-length sequence of elements of Σ

For x a string, |x| is the length of x

The unique string of length 0 will be denoted by **ε** or λ and will be called the empty or null string

A **language** over Σ (*sigma*) is a set of strings over Σ

L(M) = the language of machine M
= set of all strings machine M accepts

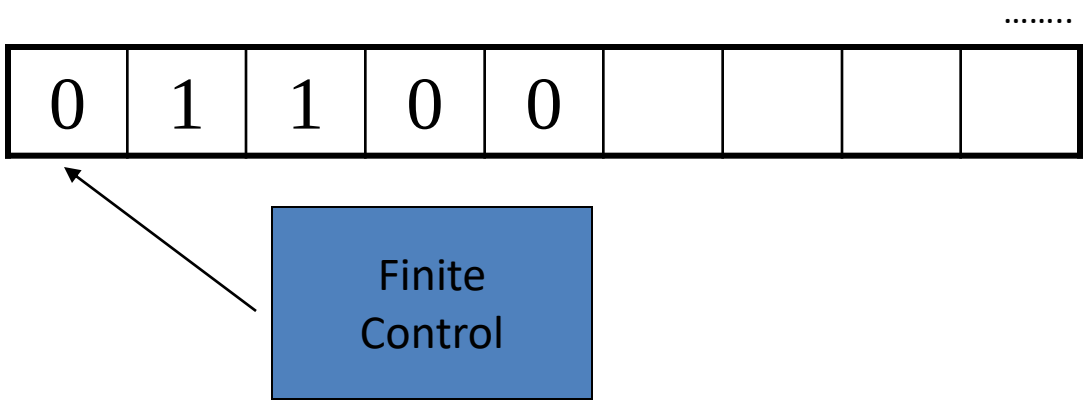
Finite Automaton

- A finite automaton has a set of states, and its "control" moves from state to state in response to external "inputs."
- One of the crucial distinctions among classes of finite automata is whether that control is "deterministic," meaning that the automaton cannot be in more than one state at any one time, or "nondeterministic," meaning that it may be in several states at once.

Deterministic Finite State Automata (DFA)

- The term "deterministic" refers to the fact that on each input there is one and only one state to which the automaton can transition from its current state.

Deterministic Finite State Automata (DFA)



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

Formal Definition of a DFA

- A DFA is a five-tuple:

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

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

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

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

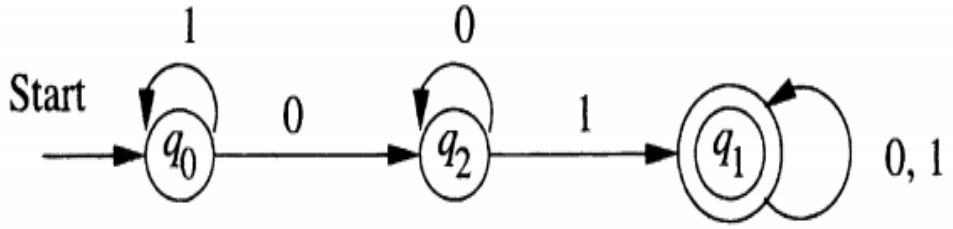


Figure 2.4: The transition diagram for the DFA accepting all strings with a substring 01

Transition Tables

A *transition table* is a conventional, tabular representation of a function like δ that takes two arguments and returns a value. The rows of the table correspond to the states, and the columns correspond to the inputs. The entry for the row corresponding to state q and the column corresponding to input a is the state $\delta(q, a)$.

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Transition table for the DFA of Example 2.1

Regular Expressions

- The language accepted by finite **automata** can be easily described by simple **expressions** called **Regular Expressions**.

RE & DFA relation

- Specification
- Recognition

Regular Expressions and Languages

- Regular expressions also may be thought of as a "programming language," in which we express some important applications, such as text-search applications or compiler components.
- Regular expressions are closely related to nondeterministic finite automata and can be thought of as a "user-friendly" alternative to the NFA notation for describing software components.

Regular Expressions and Languages

- DFA and NFA are machine-like descriptions of languages, whereas, REs are algebraic description of languages.
- Regular expressions offer something that automata do not: a declarative way to express the strings we want to accept. Thus, regular expressions serve as the input language for many systems that process strings. Examples include:

RE applications

1. Search commands such as the UNIX grep or equivalent commands for finding strings that one sees in Web browsers or text-formatting systems.

These systems use a regular-expression-like notation for describing patterns that the user wants to find in a file. Different search systems convert the regular expression into either a DFA or an NFA, and simulate that automaton the file being searched. For example, in Windows OS:

C:\>dir *sheet*.xl?

–the asterisk (*) represents zero or more characters, and the question mark (?) substitutes for any single character.

RE applications

2. Lexical-analyzer generators, such as Lex or Flex. Recall that a *lexical analyzer* is the component of a compiler that breaks the source program into logical units (called tokens) of one or more characters that have a shared significance. Examples of tokens include keywords (e.g., while), identifiers (e.g., any letter followed by zero or more letters and/or digits), and signs, such as + or <=. A lexical-analyzer generator accepts descriptions of the forms of tokens, which are essentially regular expressions, and produces a DFA that recognizes which token appears next on the input.

The Operators of Regular Expressions

- 1. The *union* of two languages L and M, denoted L u M, is the set of strings that are in either L or M, or both. For example, if
  - L = {001, 10, 111} and
  - M = {ε, 001} then
  - L U M = {ε, 10 , 001 ,111}

The Operators of Regular Expressions

- 2. The *concatenation* of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with string in M.
- For example, if
  - L = {001, 10, 111} and
  - M = {ε,001}, then
  - L.M, or just LM = {001, 10, 111,001001, 10001, 111001}

Since **ε** is the identity for concatenation, the resulting strings are the same as the strings of L.

The Operators of Regular Expressions

- 3. The closure (or star or Kleene closure) of a language L is denoted L* and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (i.e., the same string may be selected more than once) and concatenating all of them.

The Operators of Regular Expressions

- if L = {0, 1}, then L* is all strings of 0's and 1's. If L={0,11}, then L* consists of those strings of 0's and 1 's such that the 1 's come in pairs, e.g., 011, 11110, and ε but not 01011 or 101. More formally, L* is the infinite union

$$U_{i>0} L^i,$$

  - where $L^0 = \{\varepsilon\}$, $L^1 = L$, and L^i, for i >1 is LL…L (the concatenation of 'i' copies of L).

Question

- What is L^{+}?
- Represent in terms of primitive operators.

Building Regular Expressions

BASIS: The basis consists of three parts:

1. The constants ϵ and $\emptyset$ are regular expressions, denoting the languages $\{\epsilon\}$ and $\emptyset$, respectively. That is, $L(\epsilon) = \{\epsilon\}$, and $L(\emptyset) = \emptyset$.

2. If a is any symbol, then $\mathbf{a}$ is a regular expression. This expression denotes the language $\{a\}$. That is, $L(\mathbf{a}) = \{a\}$. Note that we use boldface font to denote an expression corresponding to a symbol. The correspondence, e.g. that $\mathbf{a}$ refers to a, should be obvious.

3. A variable, usually capitalized and italic such as L, is a variable, representing any language.

Building Regular Expressions

INDUCTION: There are four parts to the inductive step, one for each of the three operators and one for the introduction of parentheses.

1. If E and F are regular expressions, then $E + F$ is a regular expression denoting the union of $L(E)$ and $L(F)$. That is, $L(E+F) = L(E) \cup L(F)$.

2. If E and F are regular expressions, then EF is a regular expression denoting the concatenation of $L(E)$ and $L(F)$. That is, $L(EF) = L(E)L(F)$.

3. If E is a regular expression, then E^* is a regular expression, denoting the closure of $L(E)$. That is, $L(E^*) = \big(L(E)\big)^*$.

4. If E is a regular expression, then (E), a parenthesized E, is also a regular expression, denoting the same language as E. Formally; $L\big((E)\big) = L(E)$.

Example

- Example 3.2: Let us write a regular expression for the set of strings that consist of alternating 0's and 1 's. First, let us develop a regular expression for the language consisting of the single string 01. We can then use the star operator to get an expression for all strings of the form 0101…01.

Example…

- The basis rule for regular expressions tells us that 0 and 1 are expressions denoting the languages {0} and {1}, respectively. If we concatenate the two expressions, we get a regular expression for the language {01 }; this expression is 01.

Example…

- To get all strings consisting of zero or more occurrences of 01, the regular expression is (01)*. Note that we first put parentheses around 01, to avoid confusing with the expression 01*, whose language is all strings consisting of a 0 and any number of 1s.
- L((0I)*) is not exactly the language. It includes only those strings of alternating 0's and 1's that begin with 0 and end with 1.

Example…

- However, to consider the possibility that there is a 1 at the beginning and/or a 0 at the end. One approach is to construct three more regular expressions that handle other possibilities. That is, (**10**)* represents strings that begin with 1 and end with 0, while **0**(**10**)* can be used for strings that both begin and end with **0** and **1**(**01**)* serves for strings that begin and end with **1**. The entire regular expression is

  (01)* + (10)* + 0(10)* + 1(01)*

The Operators of Regular Expressions

Example 3.1 : Since the idea of the closure of a language is somewhat tricky, let us study a few examples. First, let $L = \{0, 11\}$. $L^0 = \{\epsilon\}$, independent of what language L is; the 0th power represents the selection of zero strings from L. $L^1 = L$, which represents the choice of one string from L. Thus, the first two terms in the expansion of L^* give us $\{\epsilon, 0, 11\}$.

Next, consider L^2. We pick two strings from L, with repetitions allowed, so there are four choices. These four selections give us $L^2 = \{00, 011, 110, 1111\}$. Similarly, L^3 is the set of strings that may be formed by making three choices of the two strings in L and gives us

$$\{000, 0011, 0110, 1100, 01111, 11011, 11110, 111111\}$$

Precedence of Regular-Expression Operators

- 1. The star operator is of highest precedence.
  2. Next in precedence comes the concatenation or "dot" operator. That is, all expressions that are juxtaposed (adjacent, with no intervening operator) are grouped together.
  3. Finally, all unions (+ operators) are grouped with their operands. Since union is also associative, no matter in which order consecutive unions are grouped, but we shall assume grouping from the left.

Example

- Example 3.3: The expression 01 * + 1 is grouped

$$(0(1^*)) + 1$$

- To group the dot before the star, could have used parentheses, as (01)* + 1. (WRONG)
- To group the union first, we could have added parentheses around the union to make the expression 0(1 * + 1). (WRONG)
-

Exercises

- Write regular expressions for the following language:
  - The set of strings of 0's and 1's with at most one pair of consecutive 1's.