

Lecture 1 — August 28

Lecturer: Xiang-Yang Li

Scribe: Xiang-Yang Li

1.1 Preface

As students from computer science department, I am sure that everyone is familiar with computers. We know that the computers can do lots of things for us: edit a report, calculate some mathematical functions, organize our schedule, and so on. Actually, all what computer does is to output something (which may be empty) under some given input (which also may be empty). In other words, the computer does is a computation. In this course, we try to understand what is a computation?

What is a computer? How can I know? What is a computation?

This book attempts to answer these questions. We will gather intuitions about what would constitute a computation, perhaps by thinking of tests we would make on the computer. But in order to see if our intuitions actually are correct, we will need to phrase them in a precise enough fashion that we will be able to reason unambiguously about them. That is, we will have to gather our intuitions into a mathematical model.

If the computer matches our model, then we will conclude that it does computations. This will answer the above question. But this will leave new questions, such as

What can the computer compute?

Is there anything the computer cannot compute?

Are there limits to how quickly it can compute?

And the ultimate question will be whether we can develop a general model of computation, which will be the standard against which we will measure any claim, alien or otherwise, that a device is a computer. That is, the big question will be

What is a computation?

We assume that the reader has some insight into what a computation is. A sufficiently advanced intuition would be something like:

1.2 Introduction

We will study the relationships between languages, machines, and grammars.

A *language* is a set of strings over a finite alphabet, where a *string* is the concatenation of zero or more symbols from the alphabet.

Machines will always include a means of reading input from an input tape, one symbol at a time, and will contain some amount of finite state control. Additional storage and types of storage, additional heads and the ability to move the head(s) in different ways provide for variations in the machine models, and may allow machines to perform harder tasks or perform the same tasks faster. Machines with output capabilities may also be considered as generators of languages (they output exactly the strings of the language delimited in some fashion) or computers (given an input string,

the machine may produce an output string and halt, or if the function is not defined for that input string, it may not halt). For the most part, we will consider machines as language recognizers, that is, given an input string, the machine will execute for some number of steps and halt in an accepting state or not (it may not halt, or it may halt in a non-accepting state).

Grammars consist of two sets of symbols, *terminals* (corresponding to the language's alphabet) and *non-terminals* (which act as variables), along with *rules* for how to start and how to replace non-terminals with other strings. Exercising a sequence of these production rules from the start symbol may produce a string of terminals, in which case that string is said to be in the language of that grammar and the sequence is said to be a *derivation* of that string. Determining if a string is in the language generated by a grammar consists of parsing the string, that is, finding a derivation for the string. These three ways of defining languages (directly, or by describing a recognizing machine or a generating grammar) are closely related, and the nature of their relationships will constitute a major portion of our study.

Additional properties are also of interest, and by establishing the relationships between languages, machines and grammars, we obtain tools to explore these other properties. Languages, machines and grammars may be organized into classes depending upon properties that they have, and in many cases, the classes of languages will correspond to classes of machines and classes of grammars. Determining to which class a particular language belongs is one type of question we will often ask. We will also define operations on languages, by which one or more language may be combined or altered in a systematic way to produce another language (perhaps the same one). A particularly interesting set of questions have to do with closure properties of classes of languages. A class of language is closed under some operation if, given a language in the class, application of the operation always produces another language (perhaps the same one) within the same class. There are subtle but important differences between questions about languages, machines, grammars and classes of these. The student should gain an appreciation of these differences in addition to acquiring techniques to answer the questions that arise.

Special questions of great importance include whether a language is *computable* (that is, is there any machine that can recognize it among the fleet of machine models we know), whether a language is recognizable by a machine that always halts (decidability), and whether a language has an “efficient” machine (one that operates in polynomial time: tractability). Once we have gained a solid foundation in the concepts of language theory, we will explore the structure of classes of languages.

1.3 Mathematical Preliminaries

Below are some preliminary definitions provided to remind the student of the basic mathematics we will need in our explorations. This summary will be rather abstract - refer to the book for examples.

1.3.1 Sets

A set is a finite or infinite collection (zero, a finite number, a countably infinite number or an uncountably infinite number) of objects in which order has no significance, and multiplicity is generally also ignored (unlike a list or multiset). Set theory is founded by **George Cantor**. We will always assume the existence of a universal set, U , relative to which the complement operation

will be defined. A set may be defined by listing the elements (if it is finite), by describing some means of generating or recognizing the elements, or by a “set former”. The latter is of the form

$$S = \{x \text{ in } A \mid P(x)\},$$

read as “ S consists of x in A such that x has property P .” $P(x)$ is a predicate that describes what must be true of x in order for it to be in the set S .

A set S is often represented by all its elements or $\{x \mid x \text{ belongs to } S\}$. For example, the set of natural numbers is often represented as $\mathcal{N} = \{i \mid i \text{ is a integer and } i > 0\}$. Standard sets of numbers we will use include

- The symbol Φ is used to denote the set containing no elements, called the *empty set*.
- R (the real numbers),
- Z (the integers),
- N (the natural numbers, $N = \{1, 2, 3, \dots\}$),
- Q (the rational numbers, $Q = \{\frac{p}{q} \mid p \in Z \text{ and } q \in Z \text{ and } q \neq 0\}$)
- S^+ (the positive members of the set S).
- S^- (the negative members of the set S).

Symbols related to sets include

- $S_1 \cap S_2 = \{x \mid x \text{ is in } S_1 \text{ and } x \text{ is in } S_2\}$, which means “and” or intersection,
- $S_1 \cup S_2 = \{x \mid x \text{ is in } S_1 \text{ or } x \text{ is in } S_2\}$, which means “or” or union,
- $S_1 - S_2$, or $S_1 \setminus S_2 = \{x \text{ in } U \mid x \text{ is in } S_1 \text{ and } x \text{ is not in } S_2\}$ which means “minus” or difference,
- $\bar{S} = \{x \text{ in } U \mid x \text{ is not in } S\}$, which means complementation,
- Subset $S_1 \sqsubseteq S_2$ means that each element in S_1 is in S_2 .
- Proper subset $S_1 \subset S_2$ means that each element in S_1 is in S_2 and there is at least one element from S_1 is not in S_2 ,
- $x \in S$ means that element x belongs to set S ,
- $x \notin S$ means that element x does not belong to set S ,
- Power set: $2^S = \{A \mid A \text{ is a subset of } S\}$,

- The *Cartesian product* of two sets S_1 and S_2 is a set of ordered pair of elements one from S_1 and one from S_2 . We write it as

$$S_1 \times S_2 = \{(x, y) \mid x \in S_1, y \in S_2\}.$$

This notion is extended to the Cartesian product of more than two sets. In other words, we have

$$S_1 \times S_2 \times \dots \times S_n = \{(x_1, x_2, \dots, x_n) \mid x_1 \in S_1, x_2 \in S_2, \dots, x_n \in S_n\}.$$

Two sets A and B are said to be *disjoint* iff $A \cap B = \phi$.

Then operation on these sets using the \cap and \cup operators is *commutative*, *associative* and *distributive*.

- Commutative: $A \cup B = B \cup A$, $A \cap B = B \cap A$.
- Associative: $(A \cup B) \cup C = A \cup (B \cup C)$, $(A \cap B) \cap C = A \cap (B \cap C)$
- Distributive: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$, $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.
- De Morgan's Laws: $\overline{A \cup B} = \overline{A} \cap \overline{B}$, $\overline{A \cap B} = \overline{A} \cup \overline{B}$

Let $S = \{a, b\}$, then

$$2^S = \{\phi, \{a\}, \{b\}, \{a, b\}\}$$

The size of a set S is indicated by $|S|$ or by $\#S$, which is the number of elements in the set.

- $|A \cup B| = |A| + |B| - |A \cap B|$, moreover $|S_1 \cup S_2 \cup \dots \cup S_n| = \sum |S_i| - \sum |S_i \cap S_j| + \sum |S_i \cap S_j \cap S_k| \dots$
- If sets A and B are disjoint, then $|A \cup B| = |A| + |B|$
- $|A \times B| = |A| * |B|$, moreover $|S_1 \times S_2 \times \dots \times S_n| = |S_1| \times |S_2| \times \dots \times |S_n|$
- $|2^A| = 2^{|A|}$

1.3.2 Logic

Logic deals with statements and their truth sets. A statement may have variables in it that may be *bound* or *free*. Binding follows scoping rules as it does in statically scoped programming languages, with the addition of the two quantifiers, for all (\forall) and exists (\exists). The order of quantifiers matters a great deal: the following statements $S1$ and $S2$ are not at all the same.

- $S1$: For all x in N there exists y in N such that $y = 2x$.
- $S2$: There exists y in N for all x in N such that $y = 2x$.

Statements have associated with them truth sets, for which the statement is true, when an element from the truth set is bound to the statement's variables. For $S1$, the truth set

$$T(S1) = \{(x, y) \mid y = 2x\}$$

For $S2$, the truth set is empty. A statement is said to be satisfied by an assignment if the assignment is in the truth set, that is, if substitution of the values in the assignment for the variables in the statement produces a true statement.

One statement is said to imply another statement, written $S1 \implies S2$, if $T(S1)$ is contained in $T(S2)$. Two statements are equivalent, written $S1 \iff S2$, iff $S1 \implies S2$ and $S2 \implies S1$ (their truth sets are equal). Operators on statements include *and* \wedge (upside down V), *OR* \vee , and *NOT*. Thus $P \implies Q$ is the same as $(\text{NOT } P) \vee Q$.

1.3.3 Relations

A *relation* on a set S is a set of ordered pairs of elements from S . Therefore, the relation is a subset of $S \times S$.

An equivalence relation on a set S is a relation R satisfying certain properties. Write xRy to mean (x, y) is an element of R , and we say x is related to y , then the properties are

Reflexive: xRx for all $x \in S$,

Symmetric: xRy implies yRx for all $x \in S$ and $y \in S$.

Transitive: xRy and yRz imply xRz for all $x \in S$, $y \in S$ and $z \in S$.

Where these three properties are completely independent. Other notations are also often used to indicate a relation, e.g., $a \equiv b$ or $a \sim b$.

Good reference for set theory: <http://mathworld.wolfram.com/Set.html>

1.3.4 Functions

A function is a special type of relation, one for which the second element of each pair is uniquely determined by the first element. Let f be a function defined on a set A and taking values in a set B . Set A is called the *domain* of f ; set B is called the *range* of f .

If function f is defined for each element of A , then f is said to be a *total function*; otherwise, f is said to be a *partial function*. Depending on the domain and the codomain, properties a function $f : A \rightarrow B$ may have include the following.

- *Partial:* if there is some a in A such that there is no b in B such that $f(a) = b$ (i.e., f is not defined at a).
- *Total:* if f is defined on all of A (is not partial).
- *Many-to-one:* f is many-to-one if there exist a, a' in A , $a \neq a'$, and $f(a) = f(a')$, that is, there are at least two distinct elements of the domain that have the same image under f .

- *Injection* (one-to-one): iff for all (a, b) and (a', b') in f , $b = b' \implies a = a'$, that is, for every element of B there is at most one pre-image in A .
- *Surjection* (onto): iff for all b in B there is an a in A such that $f(a) = b$, that is, the codomain is the range.
- *Bijection*: iff f is both one-to-one and onto.

1.3.5 Countable Sets

Definition 1. Any set which can be put in a one-to-one correspondence with the natural numbers (or integers) so that a prescription can be given for identifying its members one at a time is called a countably infinite set. A set is countable if it is countably infinite or it is finite. Once one countable set is given, any other set which can be put into a one-to-one correspondence with is also countable.

Examples of countable sets include the integers and rational numbers.

Example 1. Prove that countably infinite buses each containing countably infinite passengers can be arranged to countably infinite hotel rooms.

PROOF. We prove this by constructing such an assignment as follows. The basic idea of this construction is following. The number of buses is countably infinite, therefore, there is a one-to-one function mapping each bus to an integer. We call the mapped integer the *index* of that bus. Similarly, each passenger in every bus will have an index. Thus we can represent each passenger by (b, p) , where b is the index of the bus containing the passenger, p is the index of that passenger in bus b . We then try to find a one-to-one mapping from (b, p) to integers. We assign the p th passenger from the b th bus an integer

$$\frac{(b + p - 1)(b + p - 2)}{2} + p.$$

We then show that the above mapping is really a one-to-one mapping, which is left as an off-class question. \square

1.3.6 Graph and Trees

A graph G is a mathematical object composed of points known as *vertices* or *nodes* and lines connecting some (possibly empty) subset of them, known as *edges*. Formally, a graph is a binary relation on a set of vertices. If this relation is symmetric, the graph is said to be *undirected*; otherwise, the graph is said to be *directed*. Graphs in which at most one edge connects any two nodes are said to be *simple graphs*. Vertices are usually not allowed to be self-connected, but this restriction is sometimes relaxed to allow such "loops". The edges of a graph may be assigned specific values or labels, in which case the graph is called a *labeled graph*.

Unless stated otherwise, the unqualified term "graph" usually refers to a simple graph. A non-simple graph with no loops but which can contain more than one edge between any two points is called a *multigraph*.

1.3.7 Proof Techniques

A proof is an argument or explanation that is sufficiently detailed and logical such that other people with enough background are convinced of the truth of the statement that is to be proven. To prove some statements, we always assume that some other statements are true and can be used to prove the statement that need to be proved. Formal proof systems rely on *axioms* (statements taken to be true without proof), proved theorems, lemmas, corollaries (statements have been proved to be true), and *rules* that allow one or more proved statements to produce new statements.

Proofs may be best written in an hierarchical fashion. That is, the prove should provide the basic idea in the proof, an outline of the major steps. The basic proof strategies are: *prove by induction*, *prove by contradiction*, *prove by construction*, and *prove by pigeonhole principle*. For each major step, some substeps are often necessary to justify that step. A step that is obvious to someone may be scanned, while one that is not obvious may be expanded and investigated in greater detail. When constructing a proof, the most essential step is to state clearly the statement to be proven and any assumptions. When the proof is completed, the reader should be able to identify how the argument supports the truth of the statement, and ideally, the reader should gain some insight into the nature of the structures involved.

In this class, we will see some widely used proof techniques. We review them briefly here.

Proof by Induction

Induction is a technique by which the truth of a number of statements can be inferred from the truth of a few specific instances. Suppose that we want to prove a sequence of statements P_1, P_2, P_3, \dots to be true; or we want to prove a statement is true for a sequence of instances. Suppose that the following conditions holds:

1. For some $k > 1$, we know that $P_1, P_2, P_3, \dots, P_k$ are true.
2. For any $n > k$, the truths of $P_1, P_2, P_3, \dots, P_n$ imply the truth of P_{n+1} .

We then use induction to show that every statement in the sequence is true.

The first condition of the induction is called **basis**, and the second condition is called the **inductive step**, which is usually the major step in the proofs by induction.

Example 2. Prove that

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}.$$

PROOF. See the textbook for the proofs. □

Exercise 1. Prove that $(xy)^R = y^R x^R$.

Proof by Contradiction

Proof by contradiction is another powerful technique that often works when everything else fails. Suppose that we want to prove that some statement P is true. We then assume for the moment, that P is false and to see whether this will lead to a contradictive statement. If it does, then the only assumption to be blamed is the starting assumption that P is false. We then can conclude that P is true.

Example 3. *Prove that $\sqrt{2}$ is not a rational number.*

PROOF. Please see the textbook for the proofs. □

Exercise 2. *Prove that real numbers in range $[0, 1]$ is not countable.*

Exercise 3. *Prove that there are lots of functions that cannot be computed by program.*

Prove by Pigeonhole Principle

Criterion 1.3.1. *Pigeonhole Principle:*

1.4 Basic Concepts and Terminologies

There are three fundamental concepts:

1.4.1 Languages

- A language is a subset of the set of all possible strings formed from a given set of symbols.
- There must be a membership criterion for determining whether a particular string in the set.

An alphabet is a finite, nonempty set of symbols. We use Σ to denote this alphabet.

Remark 1. *Symbols may be more than one English letter long, e.g. while is a single symbol in Pascal.*

Definition 2. *A string is a finite sequence of symbols from Σ .*

Notation 1. *The length of a string s , denoted $|s|$, is the number of symbols in it.*

The empty string is the string of length zero. For convenience we usually write it as λ .

Notation 2. Σ^* denotes the set of all sequences of strings that are composed of zero or more symbols of Σ ; Σ^+ denotes the set of all sequences of strings composed of one or more symbols of Σ . That is, $\Sigma^+ = \Sigma^* - \{\lambda\}$.

Definition 3. *A language is a subset of Σ^* .*

The concatenation of two strings is formed by joining the sequence of symbols in the first string with the sequence of symbols in the second string. If a string S can be formed by concatenating two strings A and B , i.e., $S = AB$, then A is called a prefix of S and B is called a suffix of S .

The reverse of a string S , denoted by S^R , is obtained by reversing the sequence of symbols in the string. For example, if $S = abcd$, then $S^R = dcba$.

Any string that belongs to a language is said to be a *word* or a *sentence* of that language.

Operations on Languages

Languages are sets. Therefore, any operation that can be performed on sets can be performed on languages.

If L , L_1 and L_2 are languages, then

- The union $L_1 \cup L_2$ is a language.
- The intersection $L_1 \cap L_2$ is a language.
- The difference $L_1 - L_2$ is a language.
- The complement of L , \bar{L} , is a language.
- The concatenation $L_1 L_2$ is a language. (The strings of $L_1 L_2$ are strings that have a word of L_1 as a prefix and a word of L_2 as a suffix.)
- L^n , the catenation of L with itself n times, is a language.
- $L^* = \bigcup_{n \geq 0} L^n$, the star closure of L , is a language.
- $L^+ = \bigcup_{n \geq 1} L^n$, the positive closure of L , is a language.

1.4.2 Grammars

- A grammar is a formal system for accepting or rejecting strings.
- A grammar may be used as the membership criterion for a language.

Definition 4. A grammar G is a quadruple $G = (V, T, S, P)$, where

- V is a finite set of (meta)symbols, or variables.
- T is a finite set of terminal symbols.
- $S \in V$ is a distinguished element of V called the start symbol.
- P is a finite set of productions (or rules).

A production has the form $X \rightarrow Y$, where $X \in (V \cup T)^+$, and $Y \in (V \cup T)^*$. We'll put this in words

$X \in (V \cup T)^+$: X is a member of the set of strings composed of any mixture of variables and terminal symbols, but X is not the empty string.

$Y \in (V \cup T)^*$: Y is a member of the set of strings composed of any mixture of variables and terminal symbols; Y is allowed to be the empty string.

Derivations

Productions are rules that can be used to define the strings belonging to a language. Suppose language L is defined by a grammar $G = (V, T, S, P)$. You can find a string belonging to this language as follows:

1. Start with a string w consisting of only the start symbol S .
2. Find a substring x of w that matches the left-hand side of some production p in P .
3. Replace the substring x of w with the right-hand side of production p .
4. Repeat steps 2 and 3 until the string consists entirely of symbols of T (that is, it doesn't contain any variables).
5. The final string belongs to the language L .

Each application of step 3 above is called a *derivation* step. Suppose w is a string that can be written as uxv , where

- u and v are elements of $(V \cup T)^*$
- x is an element of $(V \cup T)^+$

In addition, assume that there is a production $x \rightarrow y$ in P . Then we can write

$$uxv \Longrightarrow uyv$$

and we say that uxv directly derives uyv .

Notation:

- $S \Longrightarrow T$: S derives T (or T is derived from S) in exactly one step.
- $S \xRightarrow{*} T$: S derives T (or T is derived from S) in zero or more steps.
- $S \xRightarrow{+} T$: S derives T (or T is derived from S) in one or more steps.

1.4.3 Automata

- An automaton is a simplified, formalized model of a computer.
- An automaton may be used to compute the membership function for a language.
- Automata can also compute other kinds of things.

An automaton is a simple model of a computer. There is no formal definition for "automaton"—instead, there are various kinds of automata, each with its own formal definition. Generally, an automaton

- has some form of input,

- has some form of output,
- has internal states,
- may or may not have some form of storage,
- is hard-wired rather than programmable.

An automaton that computes a Boolean (yes-no) function is called an *acceptor*. Acceptors may be used as the membership criterion of a language. An automaton that produces more general output (typically a string) is called a *transducer*.