Regular Languages

What is this course about???

- What is COMPUTATION?
  - Informally obvious, formally quite difficult to characterize: serious attempts started in the 17th century; continued through the next two; culminated in the work of Gödel, Church and Turing in the first half of the 20th.
  - K. Gödel: around 1930. Formalized computation enough to show that some desired properties DO NOT HOLD.
  - A. Church introduced the lambda calculus (computation by functions) in an attempt (unsuccessful) to get around the difficulties pointed out by Gödel.
  - A. Turing introduced an "ideal machine" that allowed some different questions to be asked, and proved fruitful as a model for "actual" computational machines.

Church & Turing

The two approaches were equivalent in the computations they could represent - very different in what they made easy to talk about

- Turing Machine: decent model for current computer hardware; provides a convenient platform for complexity analysis of algorithms: how much time, how much space.
  - Not very good for proofs of correctness and for the development and analysis of language features. Leads to imperative languages.
- Lambda Calculus: decent model for the study of behavior in terms of functions; has a reasonably clean semantics - both operational and denotational. Leads to functional languages.

TM: 91.502, 91.503
LC: 91.531

Problems

One of the problems brought out by the Gödel-Church-Turing formalisms is that too many interesting questions (can I compute X for this object Y?) have a negative (mostly "undecidable") answer (as we shall see).

Since we just can't give up (unless we are willing to give up most things that have happened in the last three quarters of a century), the next important question becomes:

- how can I change the formalism (by making it, usually, less powerful) in such a way that some of the interesting questions have (usable) answers?

A critical example, for us, is: can I design a language (details later) to describe computations (e.g., C) so that, given a piece of code (= a string of squiggles), I can always answer the question:

- is this string of squiggles a LEGAL PIECE OF CODE (= sentence) in this language?

If the answer is YES, the next question is:

- Is there a limit to the class of languages for which this question is always answerable? (e.g., if I add, remove or change some features, do I lose this capability?)
Regular Languages

Languages and Models of Computation

Chomsky Hierarchy: as part of the development of "formal linguistics", Noam Chomsky (MIT) published a series of papers around 1953-54 that introduced a family of grammars (called generative grammars) providing a hierarchy of features and descriptive power.

The publication of the papers coincided (roughly) with IBM’s release of the first FORTRAN compiler.

The two sets of ideas found use in the new fields of Computer Languages and Compiler Design. It was also found that the grammars correspond to "abstract machines", that can be arranged in the same hierarchy.

We start with the simplest (and useful) family of grammars (= languages) and their corresponding machines.

Regular Languages

The Idea of Language

Symbols: a set of “fundamental squiggles” = alphabet

Strings: ordered, finite collections of symbols

Example:

alphabets:
1) {a, b, c, ..., z}
2) {0, 1, ..., 9}
3) {I, V, X, L, C, D, M} (Roman Numerals)

strings:
1) {t, h, i, s} ≡ this
2) {2, 0, 0, 6} ≡ 2006
3) {V, I, I, I, I, I} ≡ VII

Another alphabet: {0, 1}; the strings are binary numerals.

Proofs: Mathematical Induction

We have a countable collection of sets, \( M_0, M_1, \ldots, M_n, \ldots \) and we want to prove all sets satisfy some property \( P \).

Principle of induction:

Base Case: Prove that \( P \) holds for \( M_0 \).

Induction Assumption:

a) Assume \( P \) holds for some unspecified \( M_n \) or
b) Assume \( P \) holds for all \( M_m, m \leq n \), for some unspecified \( n \).

Induction Step:

In either case prove \( P \) holds for \( M_{n+1} \).

Operations on Strings

Length: number of symbols, with repetitions. Denoted by \( |s| \) denotes the empty string, with no symbols.

Equality: let \( s_1 = x_1 \ldots x_m, s_2 = x_1 \ldots x_n \) be strings.

\[ s_1 = s_2 \iff a) \ |s_1| = |s_2| \]
\[ b) x_i = x_j \ \forall i \]

Concatenation: let \( s_1, s_2 \) be strings as above.

\[ s_1 s_2 = x_1 \ldots x_m x_2 \ldots x_n \]

For any string \( x \):

\[ x^0 = \epsilon, x^1 = x, x^2 = x \cdot x, \ldots, x^n = x^{n-1} \cdot x \]

Substring: let \( s = x_1 \ldots x_n \):

\[ s_i = x_{i-1} \ldots x_n, 1 \leq i \leq n. \]

Prefix: \( s_i = x_1 \ldots x_i \)

Suffix: \( s_i = x_i \ldots x_n \)

Reversal: \( s^R = x_n \ldots x_1 \)

Furthermore, for \( x, y \) strings \( (xy)^R = y^R x^R \).
Regular Languages

Some Proof Techniques - a Grab-Bag

Ex. 1: Let $A = \{a_1, \ldots, a_k\}$ be an alphabet.

Question: how many strings of length $n$ can you build out of this alphabet? $k \times k \times \ldots \times k = k^n$. (Elementary combinatorics)

Ex. 2: prove that $(xy)^n = y^n x^n$.

Proof: by induction on the length of the string. The textbook has

a) Assume $x = \varepsilon$. Then $x^n = \varepsilon$.

b) Assume $x = x_1 x_2 x_3$. Then $x^n = (x_1 x_2 x_3)^n = x_1^n x_2^n x_3^n$.

NOW:

c) Assume that, for all pairs of strings of lengths $n_0, n_1$ such that $n_0 + n_1 = n$ the result holds. Let $x = x_1 \ldots x_p, y = y_1 \ldots y_p$ with $p + q = n + 1$. Then $(xy)^n = (x_1 \ldots x_p y_1 \ldots y_p)^n = (x_1 x_2 \ldots x_p y_1 y_2 \ldots y_p)^n$.

With this definition, we can complete the proof.

$(xy)^n = (x_1 \ldots x_p y_1 \ldots y_p)^n = (x_1 x_2 \ldots x_p y_1 y_2 \ldots y_p)^n$ by associativity of concatenation

$= (y_1 \ldots y_p)^n (x_1 x_2 \ldots x_p)^n x_1 x_2 \ldots x_p$ by induction assumption

$= (y_1 \ldots y_p)^n (x_1 x_2 \ldots x_p)^n$ by the definition of reversal.

Induction proofs will appear all over the course...

Regular Languages

How do you solve word equations?

Problem (Ex. 1.3): Given the alphabet $\{0, 1\}$, and the strings $\{0, 1\}$, find the set of strings $z$ that satisfy the equation $011 = 011z$.

Soin.:

a) $z$ is empty.

b) If not, $011$ is both a prefix and a suffix of $z$. Let $x = 011y$, where $y$ could be $\varepsilon$. From $011z = 011y$, we can “cancel” the leading 011 to obtain $x = y11$, and the equation 011y = y11. $y = \varepsilon$ is a solution of the new equation, providing us with the solution $x = 011$ for the original one. Repeating the process, we can conclude that $x = (011)^n$, $n \geq 0$, provides an infinite number of solutions. It is easy to see that these are the only solutions (prove it... assume $w = 0111011$, where $z \neq (011)^n$ and use the defining equation to strip 011s from both ends).

Languages

Let $\Sigma$ be an alphabet; let $\Sigma^*$ denote the set of all strings over $\Sigma$.

Def.: a language $L$ over $\Sigma$ is subset of $\Sigma^*$.

Basic Operations: $A$ and $B$ denote languages over some $\Sigma$.

Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$

Intersection: $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$

Subtraction: $A \setminus B = \{x \mid x \in A \text{ and } x \notin B\}$ ($A - B$ if $B \subseteq A$)

Complementation: $\overline{A} = \Sigma^* - A$

Concatenation: $A \cdot B = A \cdot B = \{xy \mid x \in A \text{ and } y \in B\}$

Kleene Closure: define $A^* = \{\varepsilon\} \cup A \cdot A$.

$A^* = A^0 \cup A^1 \cup A^2 \cup \ldots$

Positive Closure: $A^+ = A^1 A$.
Regular Languages

Languages as Sets - Containment Proofs

Lemma (Ex. 1.6). For any languages \( A \) and \( B \), \( (A \cup B)^* = A^*(B^*)^* \).

Proof: equality of sets is usually proven by proving two-way containment. \( \varepsilon \) clearly belongs to both sets and we concentrate on the non-empty cases.

1. \( A^*(B^*)^* \subseteq (A \cup B)^* \). Assume \( x \in A^*(B^*)^* \), then \( x \in (A^*B^*)^* \) for some \( n, m \geq 0 \) (by def. of Kleene closure). \( x \) can be written as \( x = x_1 x_2 \ldots x_{n+1} \) where \( x_1, x_2, \ldots, x_n \in A \) and \( y_1, y_2, \ldots, y_m \in B^* \). Each of the \( y_i \)'s can be written as \( y_i = y_{i,0} y_{i,1} y_{i,2} \ldots y_{i,k_i} \) with \( k_i \geq 0 \) and \( y_{i,0} \in B \), \( y_{i,1}, y_{i,2}, \ldots, y_{i,k_i} \in A \). It is clear that \( x = x_1 x_2 \ldots x_{n+1} \) is the concatenation of strings in \( A \cup B \), and the containment follows.

2. \( (A \cup B)^* \subseteq A^*(B^*)^* \). \( x \in (A \cup B)^* \) \( \Rightarrow \exists n \geq 0 \) s.t. \( x \in (A \cup B)^n \). Thus \( x = x_1 x_2 \ldots x_{n+1} \) for some \( x_1, \ldots, x_n \in A \cup B \). Let \( x_{n+1} \) denote the \( x_i \)'s in \( B \), with the other \( x_i \)'s in \( A \). Then \( x = y_{i,0} y_{i,1} y_{i,2} \ldots y_{i,k} y_{i,k+1} \) for some \( y_{i,0}, y_{i,1}, y_{i,2}, \ldots, y_{i,k} \in A \), and \( x \in A^*(B^*)^* \).

Regular Languages

Simplest Class: Regular Languages (Regular Sets)

Def.: let \( \Sigma \) be an alphabet. We consider sets of strings over \( \Sigma \).

1. The empty set \( \emptyset \) is a regular language
2. For every \( a \in \Sigma \), \( \{a\} \) is a regular language
3. If \( A \) and \( B \) are regular languages, \( A \cup B, AB \) and \( A^* \) are regular languages
4. Nothing else is a regular language.

Examples:
1. \( \{\varepsilon\} \) is a regular language -- \( \{\varepsilon\} = \emptyset^* \)
2. \( \{001, 110\} \) is a regular language over \( \{0, 1\} \)
3. Every finite language over an alphabet \( \Sigma \) is regular.

Regular Languages

Another Notation: Regular Expressions

To simplify the notation for Regular Languages we introduce the notion of regular expression over an alphabet \( \Sigma \).

1. \( \emptyset \) is a regular expression that represents the empty set
2. \( x \) is a regular expression that represents the string \( x \)
3. for \( a \in \Sigma \), \( a \) is a regular expression that represents the string \( \langle a \rangle \)
4. if \( r_1 \) and \( r_2 \) are regular expressions representing languages \( A \) and \( B \), respectively, then \( (r_1) + (r_2), (r_1)(r_2) \) and \( (r_1)^* \) are regular expressions representing \( A \cup B, AB \) and \( A^* \), respectively
5. nothing else is a regular expression over \( \Sigma \).

Note: we will often use \( a \) to denote both \( a \) and \( \{a\} \). Confusion is not likely.
Some more details:

- Kleene closure has precedence over concatenation and union
- Concatenation has precedence over union
- Regular expressions satisfy the distributive laws:

1. \( r(s + t) = rs + rt \)
2. \( (r + s)t = rt + st \)

This shows that uniqueness of representation does not exist...

**Question:** Is there a unique minimal representation? How would we characterize minimality? By smallest number of symbols used?

We will introduce another notation...

**Ex. 1.16:** Find a regular expression for the set of binary strings with no occurrence of the substring 001.

**Soln.** A string in this language cannot have a substring 00 unless it is a suffix, in which case it can be even longer. The set of strings with no substring 00 can be represented by \((01 + 1)^*(c + 0)\). We can now add any further number of zeros as a suffix: \((01 + 1)^*(c + 0)^0 = (01 + 1)^*0^*\).

**Ex. 1.20:** Find a regular expression for the set of all strings over the alphabet that represent correct additions over binary expansions of integers, with leading zeros added when necessary. For example, the relation implies that the string is in the set.
Regular Languages

**Soln.:**
1. The language is closed under string concatenation - since there are no "implicit" carries.
2. It thus suffices to look at the minimal strings in the language, since all others will arise from their concatenation.
Consider the leftmost symbol: it must be one of
\[
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1
\end{bmatrix}
\]
since all others imply a carry still to be handled. In the first three cases, the "digit" itself is a minimal string in the language. In the fourth case, we must figure out all the minimal strings that have that as their leftmost symbol.

Regular Languages

The only possibility for the rightmost symbol is
\[
\begin{bmatrix}
1 \\
0
\end{bmatrix}
\]
since all others are either minimal strings themselves (not generating a carry), and thus removable, or not possible, since they would involve a carry from farther to the right. The second rightmost symbol must exist, and has four choices:
\[
\begin{bmatrix}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1
\end{bmatrix}
\]
From these choices we get the following minimal strings:
\[
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 0
\end{bmatrix}
\]

More Techniques: Structural Induction

This is an induction technique that uses the structure of a set, through its inductive definition, rather than its cardinality. The idea is to prove that, if two sets used to construct a larger one both possess a given property, the construction preserves the property.

The induction is not on the cardinality of the sets in question, but on the number of operations applied from the "base sets" (so the indexing is by number of operations and may not be quite as "obvious" as in the straight Mathematical Induction method).

♦ Since multiple different operations can be applied, the proof of the induction step must be repeated for each operation.
Regular Languages

Structural Induction: Example Proof

Proposition: the suffixes of strings in a regular language form a regular language. Let \( L \) denote the set of suffixes \( \{w \mid aw \in L, \text{ for some } a \in \Sigma^*\} \).

Proof:
1. (basis 1): \( \emptyset' = \emptyset \) - the suffixes of the empty language form a regular language (= set).
2. (basis 2): for any \( a \in \Sigma \), \( \{a\}' = \{(e, a)\} \) is regular (since any finite language is regular).
3. (induction step): Let \( L_1, L_2 \) be regular, with the property that \( L_1' \) and \( L_2' \) are also regular. To show: \( (L_1 \cup L_2)' \), \( (L_1 \cup L_2)' \), \( (L_1)' \) are regular.

Note the 3 induction steps, corresponding to the 3 operations.

Regular Languages

Structural Induction - continued

c) Continued
\[
(L_1')' = (e) \cup (L_1') \cup (L_1 L_2') \cup \ldots \cup (L_1 L_2') \cup \ldots
\]
\[
= (e) \cup (L_1 L_2) \cup \ldots \cup (L_1 L_2) \cup \ldots
\]
\[
= (e) \cup (L_1 L_2)^\omega \cup \ldots
\]
\[
= (L_1') L_2^\omega
\]

All three new sets are thus regular sets and the "structural induction" proof is complete.

Regular Languages

Structural Induction - continued

Ex. 1.22: Every regular language has a regular expression in disjunctive normal form \( \alpha_1 + \alpha_2 + \ldots + \alpha_n \), in which each \( \alpha_i \), for \( i = 1, \ldots, n \), does not contain the operator +.

Proof:
a. Basis: \( \emptyset \) has \( \emptyset \) as R.E. in disjunctive normal form.
b. Basis: For any \( a \), \( \{a\} \) has \( a \) as R.E. in disjunctive normal form.
c. Induction Step: suppose \( L_1 \) and \( L_2 \) have regular expressions in disjunctive normal form...
   a. \( L_1 \cup L_2 \)
   b. \( L_1 L_2 \)
   c. \( L_1^{-1} \)
A directed graph (digraph) is a pair \((V, E)\) of sets, such that each element of \(E\) (edges) is an ordered pair of elements of \(V\) (vertices). An edge \((u, v)\) is directed (from \(u\) to \(v\)), and is called an in-edge to \(v\) and out-edge from \(u\). A loop is an edge that starts and ends at the same vertex (e.g., \((u, u)\)). A path is a finite sequence of vertices \((v_1, v_2, \ldots, v_n)\) such that each \((v_i, v_{i+1})\) is an edge. A path starting and ending at the same vertex is a cycle.

If the edges have labels, we talk about a labeled digraph.

Regular Languages

Graph Representations: Definitions

The labels are taken from \(\{\varepsilon\} \cup \Sigma\). Given a regular expression \(r\), its labeled digraph representation \(G_r\) can be constructed as:

1. Start with two special vertices, the initial vertex and the final vertex and join them with an edge labeled \(r\).
2. Repeat the following until every edge has a label that does not contain operation symbols +, ⋅, or *:
   a) replace each edge with label \(fg\) by two edges with labels \(f\) and \(g\):
   b) replace each edge with label \(f\) by an additional vertex and two edges with labels \(f\) and \(g\):
   c) replace each edge with label \(f\) by an additional vertex and three edges with labels \(f\) and \(g\):
3. Delete all edges with label \(\emptyset\).

Languages, Regular Expressions and Graphs

**Theorem 1.23.** Let \(r\) be a regular expression. A string \(x\) belongs to the language \(L(r)\) if and only if there is a path in \(G_r\) from the initial vertex to the final vertex whose associated string is \(x\).

**Proof.** Let \(v_i\) be the initial vertex and \(v_f\) be the final vertex in \(G_r\). Consider the statement:

S: \(x \in L(r) \iff \text{there exists a path } (v_1, v_2, \ldots, v_f) \text{ in the digraph } G \text{ such that } x \in L(r_1) L(r_2) \ldots L(r_k) \text{ where } r_i \text{ is the label of the edge } (v_i, v_{i+1}), i = 1, \ldots, k-1.\)

We show that, by a structural induction, the construction of the graph from the regular expression always satisfies S.

Step 1 of the construction leads to a graph that satisfies S, and thus the construction satisfies S at the beginning of step 2. Step 2 involves the replacement of single edges with two edges, two edges and a
Regular Languages

vertex, or three edges and a vertex. Each allows us to rewrite the language $L(r)$ as a concatenation of languages corresponding to subexpressions of the original expression. Thus, at the end of each replacement of an edge by one of the constructions of step 2; S still holds. It will also hold for the graph obtained when step 2 can no longer be applied. Step 3 requires the elimination of all $\emptyset$-edges. Each $\emptyset$-edge corresponds to a language $L(\emptyset)$, with no strings at all. At this point, each edge of $G(r)$ is labeled by exactly one symbol from $(\varepsilon) \cup \Sigma$, and S implies that $x \in L(r) \Rightarrow$ there exists a path $(v_1, v_2, \ldots, v_k = v_f)$ in the digraph $G(r)$ whose associated string is exactly $x$.

Ex.: Construct $G(r)$ for $r = (11 + 0)^*(00 + 1)^*$.

Theorem 1.25. Let $r$ be a regular expression. Then an $\varepsilon$-edge $(u, v)$ in $G(r)$ which is a unique out-edge from a nonfinal vertex $u$ or a unique in-edge to a non-initial vertex $v$ can be shrunk into a single vertex, still preserving the property of Theorem 1.23. (If one of the endpoints of the $\varepsilon$-edge is the initial vertex or the final vertex, then so is the resulting vertex).

Proof. Omitted - see textbook.

Ex.: Construct $G(r)$ for $r = a'b(c + da'b)'$.