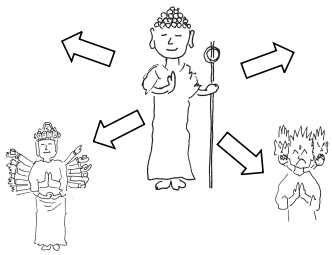
Lecture 2: Formal Language Theory — Regular Language

Formal Language Theory is one of the important fields on foundations of computing theory. This field were developed in the early stage of computer science, and most of the key results were obtained during that time. Thus, they are not new, but they should be learned as liberal arts for science and engineering students of this computer age.

In our lecture, we study "regular language" (also called "regular set"), the most fundamental concept in Formal Language Theory. It also provides us with a good example illustrating different ways to understand some objects that have been often used in computer science.

For more details and for further reading on Formal Language Theory, please refer to some standard textbooks [1].



In CS some object appears in different ways like Kannon who appears in several different ways in front of Human beings

An Important Message from O.W: For those who are not studying CS as a major subject, one book covering many subjects on theoretical computer science would be better to read instead of reading text books on some specific topics. For such books, I would recommend two textbooks [3, 5]. You can find some chapters on Formal Language Theory.)

1 Definitions of Regular Languages

We show three different definitions of Regular Languages. We begin with some very basic notions and notations of Formal Language Theory.

- *alphabet* is a finite set of symbols, which is usually fixed in each context.
  notation: Σ is often used to denote a given alphabet.
  examples: $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$, $\Sigma = \{\mathtt{0}, \mathtt{1}\}$.
- *string* is a sequence of finite number of symbols of Σ.
  notation: use here u, v, w, ... to denote strings.
  by $|u|$ we mean the length of string u.
  examples: $x = \mathtt{abcbbc}$, $y = \mathtt{010010}$.

null string (or, *empty string*) is the length 0 string.
  notation: use here ε to denote the null string.
- *language* is a set of strings.
- *concatenation* is to connect two strings to make one string.
  notation: we use here $\cdot$ for the concatenation operator, which is often omitted though.
  examples: for $x = \mathtt{ab}$, $y = \mathtt{cd}$, $x \cdot y = \mathtt{abcd}$

We extend the concatenation operator for languages. For any pair of languages S and T, define $S \cdot T$ by

$$S \cdot T \;=\; \big\{\, u \cdot v \;:\; u \in S, v \in T \,\big\}.$$

Concatenating one language (for some fixed number of times) is expressed simply by a power notation. For any language S, define $S^0 = \{\, \varepsilon \,\}$, and for any integer $t \geq 0$, define $S^t = S \cdot S^{t-1}$. Then for any language S we by S^* we mean

$$S^* \;=\; \bigcup_{i \geq 0} S^i.$$

This is just a set of strings obtained by concatenating strings in S for $i \geq 0$ times. The operator $*$ is called *Kleene closure* or *star operator*.

Now we state three different definitions for the notion of "regular language." Below we fix our alphabet Σ to either $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ or $\{\mathtt{0}, \mathtt{1}\}$.

Definition 1. *A set S of strings is* regular *if it satisfies one of the following conditions:*
(1) S is $\emptyset$, $\{\varepsilon\}$, or $\{e\}$ for some $e \in \Sigma$;
(2) $S = U \cup V$ for some regular languages U and V;
(3) $S = U \cdot V$ for some regular languages U and V; and
(4) $S = U^*$ for some regular language U.

This immediately gives the following way to express regular languages.

Definition 2. *A* regular expression *and its* interpretation *L is defined as follows:*
(1) Symbols $\emptyset$ and ε, and each element e of Σ is respectively a regular expression, where $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$, and $L(e) = \{e\}$;
(2) For any regular expressions r and s, $r + s$ is a regular expression, where $L(r + s) = L(r) \cup L(s)$;
(3) For any regular expressions r and s, $r \cdot s$ is a regular expression, where $L(r \cdot s) = L(r) \cdot L(s)$; and
(4) For any regular expression r, r^* is a regular expression, where $L(r^*) = L(r)^*$.

Example 1.

$$(\mathtt{a} + \mathtt{b} + \mathtt{c})^* \qquad \mathtt{a}(\mathtt{ba})^*\mathtt{a} + \mathtt{b}(\mathtt{ab})^*\mathtt{b}$$
$$(\mathtt{0} + \mathtt{1})^*\mathtt{1}(\mathtt{0} + \mathtt{1})(\mathtt{0} + \mathtt{1})(\mathtt{0} + \mathtt{1})$$

□

Next we define regular languages by certain methods to generate/specify languages — grammar. A grammar is a way to specify a language by showing a way to derive (i.e., generate) each sentence (i.e., string) of the language. Here we formally express this notion.

Definition 3. *A* regular grammar *G is a triple (S, N, P), where N is a set of* nonterminals*, $S \in N$ is a* start nonterminal*, and P is a set of* production rules *that are one of the following types:*
(1) $A \to \varepsilon$ for some $A \in N$;
(2) $A \to e$ for some $A \in N$ and $e \in \Sigma$; and
(3) $A \to B$ for some $A, B \in N$, or $A \to eB$ for some $A, B \in N$ and $e \in \Sigma$.
Remark. *We assume that P has at least one production rule of the form $S \to \cdots$.*

Consider any regular grammar $G = (S, N, P)$. Each production rule of P means a rule to replace the left hand side nonterminal by the right hand side: for example, $A \to eB$ means to replace A with eB. A *derivation* is to apply this type of replacements for some number of times starting from the initial nonterminal S ending with some string w (without any nonterminal). We write $S \to^* w$ if w is *derived* (or *generated*) by some derivation. Let $L(G)$ denote the set of strings w such that $S \to^* w$ holds, and $L(G)$ is called the language *generated/specified* by grammar G. Then we may define "regular language" as follows: a language is *regular* if it is generated by some regular grammar.

Example 2. Consider $G_1 = (S, \{S, A, B, C\}, P)$, where P consists of the following rules:

$$S \to A, \quad S \to B, \quad S \to C, \quad S \to \varepsilon$$
$$A \to \mathtt{a}S, \quad B \to \mathtt{b}S, \quad C \to \mathtt{c}S$$

Consider $G_2 = (S, \{S, X, Y, \}, P)$, with following (extended) production rules:

$$S \to \mathtt{a}X, \quad S \to \mathtt{b}Y$$
$$X \to \mathtt{ab}X, \quad X \to \varepsilon, \quad Y \to \mathtt{ba}Y, \quad Y \to \varepsilon$$

□

Finally we give a way to define "recognition method" for regular languages. In general, a "recognition method" of a language L is a way to determine whether a given $w \in L$. We introduce here to "finite automata" for describing recognition methods for regular languages.

Definition 4. *A (*deterministic*)* finite automaton *(in short,* DFA*) is a 4-tuple $M = (Q, \delta, q_0, F)$, where Q is a finite set of* states*, $q_0 \in Q$ is an* initial state*, $F \subseteq Q$ is a set of* final states*, and δ: $Q \times \Sigma \to Q$ is a (*state*)* transition function*.*

Remark. *We would usually use "nondeterministic" finite automata, which is essentially the same computational power as DFA but quite useful for designing a method for recognizing a regular language specified by either a regular expression or a regular grammar. But in this lecture we omit this generalized automata.*

Example 3. It would be much easier to explain/understand if we state DFA by a figure. Here we give two examples. One is for recognizing $\mathtt{a}(\mathtt{ba})^*\mathtt{a} + \mathtt{b}(\mathtt{ab})^*\mathtt{b}$, and the other is for recognizing the set (denoted by L_{bin3}) of binary strings that corresponds to natural numbers divisible by 3. (Please copy the figure below that I draw on the black board.)

□

We explain how to use DFA for language recognition. Consider any DFA $M = (Q, q_0, \delta, F)$ that is designed for recognizing some language L. The task of M is, for a given string w (i.e., any element of Σ^*), to determine whether $w \in L$ or not. For this, we first introduce a way to define an *extended* state transition function δ^*. By using δ, we define δ^* inductively as follows: For any $q \in Q$, $\delta^*(q, \varepsilon)$ is defined by $\delta^*(q, \varepsilon) = q$, and for any $q \in Q$, $e \in \Sigma$, and $w \in \Sigma^*$, define $\delta^*(q, ew)$ by

$$\delta^*(q, ew) \;=\; \delta^*(\delta(q, e), w).$$

Then for given w, we say that M *accepts* w (meaning "determine that $w \in L$") if $\delta^*(q_0, w) \in F$. Let $L(M)$ denote the set of strings accepted by M; that is, $L(M)$ is the language that is *recognized by* M. Finally, we give the third definition of "regular language": A language L is *regular* if it is recognized by some DFA.

> **An Important Message from O.W:** I think that this third definition is very characteristics in computer science. In CS, we always try to obtain some "computational" method that can be implemented on computers.

2 Derivatives of Regular Languages

Let us see yet another view of regular languages. This view is provided by the following "derivative" notion introduced by Brzozowski [2]. (Some of the following explanation is from [4], but note that symbol $\approx$ has different meaning in [4].) Here again we fix our alphabet Σ to either $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ or $\{\mathtt{0}, \mathtt{1}\}$.

Definition 5. *For any language (i.e., set of strings) L and any $u \in \Sigma^*$, define the u-*derivative of L *by*

$$\partial_u L \;=\; \{\, v \,:\, uv \in L \,\}.$$

Definition 6. *For any language L, and define equivalence*[1] *relation $\approx_L$ on Σ as follows: for any u and v in Σ^*,*

$$u \approx_L v \iff \partial_u L = \partial_v L.$$

Then we can give yet another definition of "regular language": A language L is *regular* if Σ^* is divided into a finite number of equivalence classes by $\approx_L$.

Example 4. Let us consider the set L_{bin3} of Example 3. For analyzing equivalence classes such as $\Sigma^*/\approx_{L_{\mathrm{bin3}}}$, Fact 1 below is useful. □

Fact 1. *For any language L, we have $\partial_\varepsilon L = L$, and for any $e \in \Sigma$ and any $w \in \Sigma^*$, we have $\partial_{ew} L = \partial_w(\,\partial_e L\,)$.*

[1] An equivalence relation is a binary relation satisfying (i) reflexibility, (ii) symmetry, and (iii) transitivity.

We can use this derivative operator for deriving DFA recognizing a language expressed by a regular expression. Below is a set of key rules for this derivation. (We use r and s to denote regular expressions.)

e-derivative calculation rule (where $e \in \Sigma$)

$$
\begin{array}{lcl}
\partial_e \varepsilon & = & \emptyset \\
\partial_e \emptyset & = & \emptyset \\
\partial_e a & = & \begin{cases} \varepsilon, & \text{if } a = e, \text{ and} \\ \emptyset, & \text{otherwise} \end{cases} \\
\partial_e (r + s) & = & \partial_e r + \partial_e s \\
\partial_e (r \cdot s) & = & \begin{cases} (\partial_e r) \cdot s + \partial_e s, & \text{if } \varepsilon \in L(r), \text{ and} \\ (\partial_e r) \cdot s, & \text{otherwise} \end{cases} \\
\partial_e (r^*) & = & (\partial_e r) \cdot r^*
\end{array}
$$

You may want to try this set of rules with some regular expressions. Then you will soon see what sort of equivalence classes that a given regular expression defines, and how these equivalence classes can be used to design your target DFA. I would like to leave such examination to you.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1986.

[2] J.A. Brzozowski, Derivatives of regular expressions, *J. ACM* 11(4): 481–494, 1964.

[3] A. Maruoka, *Concise Guide to Computation Theory*, Springer, 2010.

[4] S. Owens, J. Reppy, and A. Turon, *J. Functional Programming* 19 (2): 173–190, 2009.

[5] M. Sipser, *Introduction to the Theory of Computation*, PWS Publishing Company, 2012 (3rd Edition is available).

Homework assignment from this lecture[2]

Q1.1. Prove that a language L is regular if and only if Σ^* is divided into a finite number of equivalence classes by $\approx_L$. Then explain with some example why the number of equivalence classes cannot be finite for non-regular languages; estimate the number of equivalence classes in terms of the string length. (You may use any one of the first three definitions of regular language; but maybe the definition by DFA would be much easier.)

[2]The requirement of Part I (for grading) is to submit a report for two of three homework assignments given at Lecture 2, Lecture 3, and Lecture 5 by the end of November. You may use Japanese to give your answer.

Q1.2. Explain some example of using regular languges (and their expressions) to capture/formulate some computational task. (Something unrelated to string processing such as string matching.)