4CS4-6: Theory of Computation (Regular Expressions)

Lecture 03: Jan. 18, 2019

Prof. K.R. Chowdhary *: Professor of CS*

3.1 Introduction

A regular expression (regex), often called a *pattern*, is an expression that describes a set of strings. They are used to give a concise description of a set, without having to list its all elements. For example, a set containing three strings "Raj", "Rajan", and "Rajam" can be described by the pattern "Raj(ε|an|am)". The vertical bar separates alternatives, and the parentheses are used to define the scope and precedence of operators. In most formalism, if there is any regex that matches a particular set, then there are an infinite number of such expressions. The quantifier $*$ (asterisk) as power to an expression indicates that the expression repeats indefinite number of times. For example, "ab^*cd" matches all the strings *acd*, *abcd*, *abbcd*, *abbbcd*, etc. A '+'sign is used in place of separator '|'. Thus, the regex $ab + cd + efg$ matches all the strings *ab*, *cd*, efg. A sign of concatenation, '$\circ$'is used to join the strings. Thus, the regex $ab \circ cde$ matches with the string *abcde*. For the sake of convenience, in this text, a reference to strings *ab*, *cd*, efg will mean "ab", "cd", "efg", respectively. Various constructs, like $+$, $-$, $*$ and / can be combined to form arbitrarily complex expressions very much like one constructs arithmetic expressions from numbers and operators $+$, $-$, $*$ and /.

In the design of sequential circuits, first step is obtaining an unambiguous description of a circuit's behavior. For a certain class of problems, language of regular expressions greatly simplifies this first step synthesis. In general, richer the language, the easier it will be to write the problem specifications.

Regular expressions have number of applications in computer science; for example, they can be used for designing sequential circuits, and in the *lexical analysis* of programs. The other applications are pattern recognition and pattern matching, text editing and in bibliographic search systems. Common to all these approaches is that regular expressions must be converted into a finite automaton. Ken Thompson built the notations of regular expressions into the editor *QED* as a means to match patterns in text files [thompson]. Later he added this capability into the *Unix* editor *ed*. The regex were later used in design of other Unix tools, like - *grep* (global regular expression print), *expr* (expression evaluation), *awk*, *vi* (visual editor), *lex* (lexical analyzer) and in the scripting languages *Perl* and *Tcl*.

3.2 More on Regular Expressions

To begin our study of finite representations, we consider expressions in the form of symbols that describe how language can be built up using the operations. Suppose that we have the language,

$$L = \{w \in \{0,1\}^* \mid w \text{ has two or three occurrences of 1, first and second of which are not consecutive}\}.$$

This language can be described using only *singleton* sets and the symbols: $\cup$, $\circ$, and ϕ as below.

$$\{0\}^* \circ \{1\} \circ \{0\}^*\{0\} \circ \{1\} \circ \{0\}^* \circ ((\{1\} \circ \{0\}^*) \cup \phi)$$

It can be easily verified that the language represented by the above expression is precisely the L, and can also be represented in a simplified and compact form $0^*10^*010^*(10^* + \phi)$, called *regular expression*. A regular expression describes a language exclusively by means of language alphabets and ϕ and possibly combined with symbols, $\circ$, $\cup$ and $*$.

Let us first decide some notations for regular expressions. The Regular expressions over the alphabet are defined inductively.

Definition 3.1 *Regular Expression: A regular expression is defined recursively as follows:*

1. *ϕ, ε, and each $a \in \Sigma$ is a regular expression,*
2. *If α and β are regular expressions then so are $\alpha \circ \beta$, $\alpha \cup \beta$, α^*, β^*,*
3. *Nothing other than (1) and (2) above is a regular expression.*

Every regular expression represents a language according to the interpretation of symbol $\cup$ and $*$, which are set union and Kleene star respectively, and juxtaposition of expressions using concatenation operator $\circ$. Formally, a relation exists between the regular expression and the language it represents. This relation is expressed by the function $\mathcal{L}$, such that if α is any regular expression then $\mathcal{L}(\alpha)$ is the language represented by it. In other words $\mathcal{L}$ is a function from regex strings to the language.

Definition 3.2 *Language over a Regular expression: Given a regular expression, the language over this can be inductively defined as follows:*

1. *$\mathcal{L}(\phi) = \phi$, $\mathcal{L}(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}(a) = \{a\}$, for $a \in \Sigma$*
2. *If α and β are regular expressions, then*
   (a) *$\alpha\beta$ is regular expression, and corresponding regular language is $\mathcal{L}(\alpha\beta) = \mathcal{L}(\alpha)\mathcal{L}(\beta)$,*
   (b) *$\alpha \cup \beta$ is regular expression and corresponding regular language is $\mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$,*

(c) α^* is regular expression, and corresponding regular language is $\mathcal{L}(\alpha^*) = (\mathcal{L}(\alpha))^*$.

The statement (1) above defines the language $\mathcal{L}$ as ϕ, $\{\varepsilon\}$, and $\{a\}$ for the regular expression ϕ, ε, a, respectively. The expressions (2) define the language for the regular expression obtained by combining the regular expressions. The combination is done using operators of $+$, $\circ$, and $*$. Therefore, every regular expression is mapped to some language, and the mapping is *bijective*. This language is called *regular language*. The regular expression $\alpha \cup \beta$ is also represented as $\alpha + \beta$.

Example 3.3 *What are the strings in the language represented by the regular expression $((a \cup b)^* a)$?*

Solution: The language or the set of strings for this regular expression can be obtained using definition (3.2), as follows:

$$\begin{aligned}
\mathcal{L}((a \cup b)^* a) &= \mathcal{L}((a \cup b)^*)\mathcal{L}(a) \\
&= (\mathcal{L}(a \cup b))^*)\{a\} \\
&= (\mathcal{L}(a) \cup \mathcal{L}(b))^*\{a\} \\
&= (\{a\} \cup \{b\})^*\{a\} \\
&= \{(a + b)\}^*\{a\} \\
&= \{a, aa, ba, aaa, aba, baa, bba, \dots\}
\end{aligned}$$

Any language that can be represented by a regular expression can also be represented by infinitely many regular expressions. For example, the regular expressions $\alpha \cup (\beta \cup \gamma)$ and $\alpha \cup (\beta \cup \gamma)$ represent the same language. Normally we omit extra parentheses '('and ')'in regular expressions, and treat $(a \cup b) \cup c$ as a regular expression $a + b + c$.

The regular expressions and the languages they represent can be defined formally and unambiguously.

3.3 Recognizers and Generators

Let us come back to the general scheme for representation of the language L over Σ as,

$$L = \{w \in \Sigma^* \mid w \text{ has property P}\}.$$

What properties of P should we consider to define the language L? For example, what makes the properties like - "w consists of number of 0s followed by equal number of 1s", as obvious candidates for regular expression. We emphasize only one property P such that for the strings possessing property P to be admissible as specification for a language, there must be an algorithm for deciding whether it belongs to the language. The algorithm, which is in a position to decide precisely that "the string w is member of L", is called *language recognition device*. Consider that it is required to recognize the following language L, where,

$$L = \{w \in \{0, 1\}^* \mid w \text{ have 11 as sub-strings}\}.$$

A regular expression for this is $(0+1)^*11(0+1)^*)$.

These regular expressions are useful important and useful means of representing languages all the times. Both the language recognizers and language generators are *finite specification* of languages.

Finally, though the regular expressions are parenthesized expressions, for the convenience and ease of readability, the parenthesis are removed and the meaning remains unchanged. For example, the regular expression $((((1)^*1)0 \cup (1(1)^*))$ can be represented by parenthesis-free regular expression $1^*10 + 11^*$. Even after obtaining regular expressions in this form they can be further simplified using the following rules.

$1^*(1+\varepsilon) = 1^*$, where is null string,

$1^*1^* = 1^*$

$0^* + 1^* = 1^* + 0^*$

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

In the following section, some examples explain how you can create regular expressions, once the language specifications are given.

Example 3.4 *Determine regular expression for the language of strings of* 0*s and* 1*s such that all strings are of even length.*

Solution: The possible even length strings are 00, 01, 10, 11. Thus, regular expression from either of these strings is - the strings repeated zero or more number of times. Hence desired regular expression is,

$$(00 \cup 01 \cup 10 \cup 11)^* = (00 + 01 + 10 + 11)^*.$$

Example 3.5 *Determine the regular expression for the language* L *in which every string contains at least one occurrence of* 1*, for* $\Sigma = \{0, 1\}$*.*

Solution: The possible regular expressions can be, $(0+1)^*1(0+1)^*$.

It should be noted that there are more than one ways to obtain the language for the given specifications. Accordingly, $0^*1(0+1)^*$ and $(0+1)^*10^*$ are also the solutions for this problem.

Example 3.6 *Given a language* $L = \{w \mid w \in (0+1)^*$ *and* w *is of odd length*$\}$*, determine the regular expression.*

Solution: The answers are: $(0+1)(00+01+10+11)^*$ and $(00+01+10+11)^*(0+1)$.

3.4 Regular Languages

If $\mathcal{R}$ is a set of Regular Languages over some alphabet set Σ, then we define some relations as follows:

1. $\phi \subseteq \mathcal{R}$,
2. $\{\varepsilon\} \subseteq \mathcal{R}$,
3. $\{a\} \subseteq \mathcal{R}$, for all $a \in \Sigma$, and
4. If $L_1 \subseteq \mathcal{R}$ and $L_2 \subseteq \mathcal{R}$, then $L_1 \cup L_2 \subseteq \mathcal{R}$, $L_1 \circ L_2 \subseteq \mathcal{R}$, $L_1^* \subseteq \mathcal{R}$.

In other words, a language is regular if it is built using only the elementary languages and operations of union, concatenation, and Kleene Star over them.

Theorem 3.7 *The class of regular languages are closed under the operation of union, concatenation and $*$ (Kleene Star).*

Proof: The theorem states that if L_1 and L_2 are regular languages, then so are $L_1 \cup L_2$, L_1L_2, L_1^* and L_2^*.

The proof is direct in the definition of regular expressions. Let R_1 and R_2 be regular expressions corresponding to regular languages L_1 and L_2. Therefore, $R_1 \cup R_2$, R_1R_2, R_1^* and R_2^* are regular expressions as per the definition (3.1). The operations on regular sets, $R_1 \cup R_2$, R_1R_2, R_1^* and R_2^* produce the strings which belong to regular languages $\mathcal{L}(R_1 \cup R_2)$, $\mathcal{L}(R_1)\mathcal{L}(R_2)$, $\mathcal{L}(R_1^*)$, $\mathcal{L}(R_2^*)$, respectively. This proves that, the regular languages are closed under the operation of union, concatenation, and Kleene star. □

In addition, the regular languages are closed on *intersection*, *complementation*, and *word-reversal*. These topics are discussed in the next chapters.

Creating regular expression for finite language is straightforward. Consider that there is a finite language L (i.e., a language with only finitely many words). To make a regular expression that defines language L, we simply insert plus sign between all these words. If $L = \{a, b, ab, bb, ba\}$ then corresponding regular expression is simply $a + b + ab + bb + ba$.

Theorem 3.8 *Every finite language is regular.*

Proof: The word "finite" here means exactly n elements, for $n \in \mathbb{N}$. This suggests proof by *induction*.

The statement to be proved is - "for every $n \geq 0$, every language having exactly n elements corresponds to some regular expression".

Basis Step: Consider $n = 0$, hence there are 0 strings. Therefore, the language with zero number of strings is ϕ, which, according to the definition (3.1) corresponds to a regular expression ϕ.

Induction Hypothesis: Let k $(k > 0)$ be an arbitrary integer. Assume that any language with exactly k elements is regular.

Induction step: We want to show that any language L with $k + 1$ number of elements is regular. Let L be such a language. We may express L as a union of two languages L_1 and

L_2, where L_1 has k elements and L_2 has one element. By our inductive hypothesis, L_1 corresponds to some regular expression r_1 and let r_2 is a regular expression corresponding to L_2. Hence, as per the definition (3.2) of regular languages, $r_1 + r_2$ is regular expressions corresponding to the languages $L_1 \cup L_2$, which is L, having $k+1$ number of elements. Similar way, we can construct all the languages of finite number of strings, and all these will be regular. □

Example 3.9 *All languages of finite length strings are regular.*

The solution is left as an exercise.

3.5 Countable Sets

The *countable sets* or *enumerable* or *denumerable* sets are those sets elements can be counted. For example, set $\mathbb{N} = \{0, 1, 2, 3, \dots\}$. Thus any set, whose elements can be paired with the elements of $\mathbb{N}$ (having *bijection*) is countable set. The set $\{2, 3, 1, 5, 7, \dots\}$, which is exhaustively listed, can be paired with $\mathbb{N}$ as

$$\{(2, 0), (3, 1), (1, 2), (5, 3), (7, 4), \dots\}.$$

Similarly, the set of all words over $\{a, b\}$ can be enumerated by systematically pairing all the fixed length elements of $\mathbb{N}$, followed with elements of next higher length, and so on, as follows:

$$\{(\varepsilon, 0), (a, 1), (b, 2), (aa, 3), (ab, 4), \dots\}.$$

Both the examples above are of regular sets. Hence, we conclude that (these) regular sets can be enumerated. In fact, a set of regular sets is always enumerable. Consequently, the sets of regular expressions are also enumerable.

However, we know from chapter 1 that "set of subsets of enumerable set is not enumerable". This subset is all possible languages. Thus, given that, all languages are uncountable, and Regular languages are countable, we can easily conclude that, there are many languages, that are *non-regular*.

An alternate argument for the above is as follows: Each regular language corresponds to a regular expression. All the regualr expressions can be exhaustively listed, by listing the regular expressions in order of their size, and those of the same size can be listed lexicographically. Thus regular expressions are countable. Hence, the regular languages are also countable. However, we have studied that all possible languages are not countable as they are exponential power over 2 of Σ^*. This, concludes that there are languages other than regular languages.

3.6 Some applications of FA

The *regular expression* was first used as a description of finite state automata by Kleene. He defined regular expressions and regular sets of sequences and proved that:

1. Any set of sequences recognized by a finite state machine is a regular set of sequences,
2. A finite state machine can be constructed to recognize any regular set of sequences described by a finite regular expression. Later it was established that a regular expression characterizing the events recognized by a finite state machine can be obtained directly from the state diagram.

3.6.1 Design of sequential circuits

There are software as well as hardware design related problems, which can be simplified by automatic conversion of regular expression notation to efficient computer's program implementation or some hardware circuit implementation. Finite automaton is an algorithmic tool. A regular expression can be used for logic design with respect to sequential circuit, as well as for operating system design with respect to process status and transitions.

In the design of sequential circuits, the first step consists of obtaining an unambiguous description of the circuit behavior. For a certain class of problems, the language of regular expressions greatly simplifies this first step of synthesis. In general, richer the language, the easier it is to write the problem specification. It is possible to obtain state diagrams for sequential circuits from such regular expressions.

A finite automaton can be used in conceptual realization of hardware components like *flip-flops*, *counters*, *adders* and various *combinational* and *sequential* circuits. The following is an example showing conceptual design of a sequential of a sequential circuit for 3-bit binary counter.

Example 3.10 *Consider a 3-bit binary counter (figure 3.1), which counts from* 000 *to* 111*, going through* 8 *states* q_0 *to* q_7*. An input* 1 *every time causes the movement of FA to next state and input* 0 *does not change a state. The counting restarts from state* q_0*, which corresponds to count* 0 *(binary* 000*), and state* q_7 *is last state, corresponding to count* 111*.*

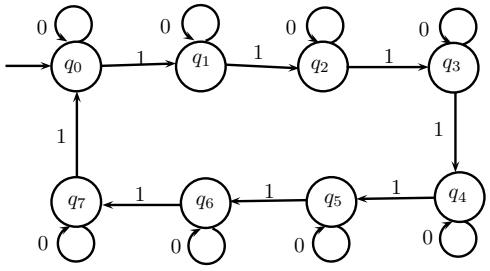

Figure 3.1: FA for 3-bit binary counter.

3.6.2 Lexical analysis

One of the applications of finite-automata is to automatically generate lexical processors. Here, a program accepts an input description of the multi-character items or of words allowable in a language, given in terms of a subset of regular expressions. The output of the systems is a lexical processor, which reads a string of characters and combines them into the

item as defined by the regular expression. Each output item is identified by a code number together with a pointer to a block of storage containing the characters and the character count in the item.

The processors produced by this program are based on finite state machines. Each state of a "machine" corresponds to a unique condition in the lexical processing of a character string. At each transition, appropriate actions are taken based on the particular character read.

The functioning of a compiler can be described in the following four logically separable steps:

1. Read the characters of the source language and assembles them into meaningful words or items (lexical analysis);
2. Group the "words" of the language into phrases and sentences for analysis (parsing);
3. Extract the meaning of the source language sentences (modeling) ;
4. Produce the appropriate object code.

Step 4, in the general case, represents the algorithm for the particular problem and as such is not amenable to general solutions. Step 3, the analysis of the parsed language, again is not amenable to general solutions. Step 2 is much more amenable to generalized solution. Very general systems exist for producing special-purpose processors for parsing language. Such automatically produced parsers are being used in compilers and other computer problem-solving applications.

With some exceptions, lexical properties have been assigned a very minor role in computer languages, and lexical processing has been incorporated as an incidental part of the syntactic-analysis programs. There are some good reasons for separating these functions both logically and programmatically.

- A large portion of compiler's time is consumed in lexical analysis, making it essential that this function be as efficient (fast) as possible.
- The development of effective languages requires attention to the lexical as well as the syntactic properties of the languages. Separating the two functions promotes recognition of this fact, and allows the functions to be investigated independently.
- Separation allows the development of systems for automatic syntactic and lexical analysis. The third point is particularly important, since the existence of such systems allows the language developer or compiler writer to experiment with various lexical and/or syntactic schemes without the burden of the immense programming times which would otherwise be required.

3.6.3 Lexical analyzers

The tokens of a programming language are expressible in the form of regular sets. For example, for the C language the identifiers are expressible as sequence of letters in upper or lower case, followed by any combination of digits, underscores and letters. Optionally an underscore may precede all above characters. Thus, a C identifier can be represented as:

$$\{\_\}^*\{letter\} \circ \{letter \cup digit \cup \_\}^*$$

When expressed as regular expression, the above becomes,

$$\_^*(letter) \circ (letter + digit + \_)^*$$

A somewhat similar but specific regular expression exits for keywords and data units in the programs. Therefore, a FA can be used to recognize different tokens in the programming languages which are different strings constituting the sequence of symbols used to form identifiers, operators, keywords, data sets, etc. The corresponding programs are called *Lexical Analyzers* and the process involved is called *Lexical Analysis*.

The programs, called *Lexical Analyzer Generators*, receive the input as a sequence of regular expressions describing the various types of tokens in a particular language, and produce a single FA, which recognizes all these tokens. The lexical analyzer generator converts the regular expressions to NFA (non-deterministic FA) with null transitions. These NFA are converted to DFA for implementation purpose. The DFA so constructed has number of final states, where each final state indicates some particular token is found. These lexical analyzers are produced directly in the form of object codes or in some form of high-level language code, which is later translated into machine language.

The lexical analyzers are used for completion of first phase of compilation process called Lexical Analysis, which identities the tokens in the input source program. The output of the Lexical Analysis is in form of tokens in some kind of representation, which is passed to next phase of compilation called *syntax analysis* or *parsing*. This phase assembles the tokens together and identities then in the form of statements as well as stores the statement in the form of a tree, called *syntax tree*.