Maggie Johnson
CS109

Handout #27

Languages & Regular Expressions

Key Topics

* Theory of Formal Languages
* Some Definitions
* Regular Expressions
* Inductions in Formal Languages
* Other Proofs for Regular Expressions

Theory of Formal Languages

In the English language, we distinguish between three different identities: letter, word, sentence. There is a certain parallelism between the fact that a group of letters make up a word and a group of words make up a sentence. Not all collections of letters form valid words, and not all collections of words form valid sentences.

This situation also exists with programming languages. Certain strings are recognizable words (FOR, WHILE, IF...). Certain strings of words are valid sentences in a programming language. These sentences together form a valid program.

To construct a general language theory that unifies all these examples, it is necessary to adopt a definition of a "universal language structure", i.e, a structure in which the decision of whether a given string of units constitute a valid larger unit is not a matter of guesswork, but is based on explicitly stated rules. The study of such a set of rules operating on a set of symbols is called the **Theory of Formal Languages**. "Formal" in this sense refers to the fact that all the rules for creating valid structures in the language are explicitly stated in terms of what strings of symbols can occur. We are interested in the FORM of the string of symbols, not the meaning. So, our study of formal languages is concerned with language not as a method of communication, but as a valid sequence of symbols generated from the application of formal rules.

This topic is important because we are now starting to study abstract machine models to get a better idea of what computers can and cannot do. All computers require a program to operate, and all programs are written in some language. The first step in understanding abstract machine models is to study the language structures that the machines can and cannot accept as input, and how the machine operates on this input. Formal language theory forms the basis for programming languages and compiler construction.

Some Definitions

1) **Symbol**: this is an abstract entity that we shall not define formally (just as "point" and "line" are note defined in geometry). Letters, digits and punctuation are examples of symbols.

2) An **alphabet** is a finite set of symbols out of which we build larger structures. An alphabet is typically denoted using the Greek sigma Σ, e.g., $\Sigma = \{0,1\}$.

3) **String:** a finite sequence of symbols from a particular alphabet juxtaposed. For example: a, b, c, are symbols and abcb is a string.

4) The **empty string**, denoted ∂, is the string consisting of zero symbols.
5) A **formal language** is the set of strings from some alphabet. This set of strings is usually denoted as $\sum^*$, e.g., if $\sum = \{1,0\}$ then $\sum^* = \{\partial, 0, 1, 00, 01, 10, 11, 000, 001, ...\}$. This language is called the **closure** of the alphabet (where any possible juxtaposition of symbols from $\sum$ is a string of the language, including ∂). The notation is called the **Kleene star** (after the logician who was one of the original workers in this field). Often, we may use a capital letter to represent a language: $L = \{\partial, 0, 1, 00, 01, 10, 11, 000, 001, ...\}$.

If, for some reason, we want to modify the concept of closure to refer to only the inclusion of non-empty strings of a formal language, we use the symbol $\sum^+$. For example, if $\sum = \{x\}$, then $\sum^* = \{\partial, x, xx, xxx, ...\}$, and $\sum^+ = \{x, xx, xxx, xxxx, ... \}$. This is called the **positive closure** of $\sum$.

6) **Words**: strings from a language that represent valid units as specified by the rules of the language. "Rich" is a valid word in English; "Xvcns" is not.

7) The **length** of a string w, denoted |w|, is the number of symbols making up the string. $|\partial| = 0$.

8) A **prefix** of a string is any number of leading symbols of that string, and a **suffix** is any number of trailing symbols. For example, the string abc has prefixes ∂, a, ab, abc; and suffixes ∂, c, bc, abc. A prefix or suffix of a string that is other than the string itself is called a **proper** prefix or suffix.

9) The **concatenation** of two strings is the string formed by writing the first and then the second string right next to each other, with no intervening spaces. The concatenation of *dog* and *house* is *doghouse*. Concatenation of two strings w and z is denoted wz.

10) A **substring** of a string is any sequence of consecutive symbols from the string including ∂. Substrings of abc are ∂, a, b, c, ab, bc, abc.

Regular Expressions

We need to be very exact in the way we define formal languages. Therefore, we will specify some language-defining symbols. Consider the language specified in item (5) above:

$$\sum = \{x\} \qquad \sum^* = \{\partial, x, xx, xxx, ...\}$$

Another way of specifying this language is to use the Kleene star directly applied to a symbol: **x***. This is interpreted to indicate some sequence of x's (maybe none at all).

$$\mathbf{x}^* = \partial, x, xx, xxx, xxxx, ... = \sum^*$$

The Kleene star operator applied to a letter is analogous to applying it to a set as we did earlier. It represents an arbitrary concatenation of copies of that letter (maybe 0 copies). Some other examples:

$$\sum = \{a, b\} \qquad \sum^* = \{\partial, a, b, ab, ba, aaa, bbb, abb, ...\}$$
$$\mathbf{ab}^* = a, ab, abb, abbb, abbbb, ...$$

∑ = {a, b}   ∑* = {∂, a, b, ab, ba, aaa, bbb, abb, ...}
(**ab**)* = ∂, ab, abab, ababab, ...

∑ = {a, b}   ∑* = {∂, a, b, ab, ba, aaa, bbb, abb, ...}
a*b** * = ∂, a, b, aa, ab, bb, aaa, aab, abb, bbb, aaaa, ...

∑ = {a, b}   ∑* = {∂, a, b, ab, ba, aaa, bbb, abb, ...}
? = {∂, a, aa, ab, aaa, aab, aaaa, aaab, abab....} ________________

How would you informally describe this last pattern? ∂ or any string of *a*'s or *b*'s that starts with an *a* and cannot have 2 or more consecutive *b*'s. It will be important to be able to recognize and describe the patterns defined by regular expressions.

Another language-defining symbol is +. This is interpreted as 'or' in an expression that defines a language.

∑ = {a, b, c}
((**a** + **c**) **b***) = a, c, ab, cb, abb, cbb, abbb, cbbb, ...

Notice in the above example that either a or c must be included. ∂ is not a part of this language.

∑ = {a, b}
(**a** + **b**)(**a** + **b**)(**a** + **b**) = aaa, aab, aba, abb, baa, bab, bba, bbb

The above example shows a finite language that contains all strings of a's and b's of length 3. We could denote this $(\mathbf{a} + \mathbf{b})^3$. If we wanted to refer to the set of all possible strings of a's and b's of any length whatsoever: (**a** + **b**)*; this includes ∂.

(**a** + **b**)* **a** (**a** + **b**)* **a** (**a** + **b**)* represents the language of all words that have at least two a's embedded in any number of a's and b's (maybe none). Another representation of a language with at least two a's: **b***ab***a**(**a** + **b**)*. abbbabb and aaaaa are both words from this language. Therefore, these two expressions are **equivalent**, meaning they describe the same language. If we wanted all the words with *exactly* 2 a's, one possibility: **b***ab***ab***. aab, baba, bbbabbbab are all words from this language.

Are the following languages equivalent?

	(ab)*a** and **a(ba)*	_____________
	(a* + b)* and (a + b)*	_____________
	(a*b)*a* and a*(ba*)*	_____________

Note that ∂ can be used in these expressions too. Suppose we wanted to define an expression for a language with ∑ = {a, b} in which a word is either empty, all b's or else there is an a followed by some b's. One possible expression: **b*** + **ab***. The ∂ can be represented by **b***. Another possibility: (∂ + **a**) **b***. Deriving this second expressions seems to indicate some kind of distributive law: ∂**b*** + **ab*** = (∂ + **a**) **b***. This analogy to algebra should be dealt with suspiciously, since addition in algebra never means choice, and algebraic multiplication has properties different than concatenation. For example, ab = ba in algebra, but ab <> ba in formal languages.

What are some algebraic properties that do not apply to regular expressions? ____________

All the expressions derived above are called **regular expressions.** The set of regular expressions can be defined by the following recursive rules:

1) Every symbol of ∑ is a regular expression
2) ∂ is a regular expression
3) if **r1** and **r2** are regular expressions, so are

(**r1**) **r1r2** **r1** + **r2** **r1***

4) Nothing else is a regular expression.

Note that we could include **r1**+ as a part of this definition, but **r1**+ = **r1r1***, so we can define it with what we have.

We have one more task concerning regular expressions, and that is to distinguish between the expressions themselves and the language associated with a regular expression. We need one more definition:

If S and T are sets of strings, we define their product ST = { all combinations of a string from S concatenated with a string from T }

M = {∂, x, xx} N = {∂, y, yy, yyy, yyyy...}
MN = {∂, y, yy, yyy, yyyy...
x, xy, xyy, xyyy, xyyyy...
xx, xxy, xxyy, xxyyy, xxyyyy...}

Now, we can define the **language associated** with a regular expression by these rules:

1) The language associated with a regular expression that is just a single letter, is that one-letter word alone.
2) The language associated with ∂ is {∂}, a one-word language.
3) If **r1** is a regular expression associated with L_1 and **r2** is a regular expression associated with L_1:
i) (**r1**)(**r2**) = L_1L_2
ii) **r1** + **r2** = $L_1 + L_2$
iii) (**r1**)* = $L_1{}^*$

Can every language be represented by a regular expression? Every finite language can, because any finite language can be represented as a list of all its words separated by +. Infinite languages, however, are a bit more complex; we will deal with them later.

Inductions in Formal Languages

1) Given that S = {xx, xxx}, prove that $S^+ = x^n$ for n > 1 (i.e., x^n e S^+)

P(n) denotes: if S = {xx, xxx}, then $S^+ = x^n$ for n > 1

base case: when n = 2, x^2 = xx; when n = 3, x^3 = xxx; both of these strings are in the language.
inductive hypothesis: Assume that for any n > 3, x^n is in S^+, prove that x^{n+1} is in S^+.

PROOF:
There are two possible cases:

a) at least one xx is used in the string: replace this xx with xxx to get x^{n+1}.
b) no xx is used, i.e., the string is all xxx: replace one xxx with 2 xx's to get x^{n+1}.

By the principle of mathematical induction, P(n) is true for all n.

*Do you really need the second base case?*_________________________________

2) P(n) denotes: A palindrome is defined as follows:

a) ∂ is a palindrome.
b) If a is any symbol then a is a palindrome.
c) If a is any symbol and x is a palindrome then axa is a palindrome.
d) Nothing else is a palindrome unless it follows from (a) - (c).

A palindrome is a string that reads the same forward and backward.

(This induction will be on the length of z which is a string that reads the same forward as backward; we will prove that z's being a palindrome follows from (a) - (c).)

base case: if the length of z is 0, we have ∂ which is defined by (a) to be a palindrome; if the
length of z is 1, we have a single symbol which is defined by (b) to be a palindrome.
inductive hypothesis: Assume that w of some length less than z is a palindrome, show that z is also a palindrome.

PROOF:
If $|z| > 1$, then z begins and ends with some symbol a. Thus z = awa where w reads the same forward as backward, and is shorter in length than z. By the induction hypothesis, rules (a)-(c) imply that w is a palindrome. Thus, by rule (c), z = awa is also a palindrome.

By the principle of mathematical induction, P(n) is true for all n.

Other Proofs for Regular Expressions

Two regular expressions R and T are equivalent if the language defined by R (i.e., the set of strings generated by regular expression R) is equal to the language defined by T. To prove equivalences for regular expressions, we use containment proofs from set theory. That is, if S_1 is the set of strings generated by regular expression R, and S_2 is the set of strings generated by regular expression T, we must prove that S1   S2 and S2   S1. Both directions are necessary to prove equality of sets. For some examples of this, refer to *Foundations of Computer Science*.

Bibliography

* The classic references on formal language theory:

M. Davis, E. Weyuker, *Computability, Complexity and Languages: Fundamentals of Theoretical Computer Science*, New York: Academic Press, 1983.

M. Harrison, *Introduction to Formal Language Theory*, Reading, MA: Addison-Wesley, 1978.

J. Hopcroft, J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Reading, MA: Addison-Wesley, 1979.

H. Lewis, C. Papadimitriou, *Elements of the Theory of Computation*, Englewood Cliffs, NJ: Prentice Hall, 1981.

A. Salomaa, *Formal Languages*, New York: Academic Press, 1973.

* Some very readable references (i.e., not quite as dense mathematically):

D. Cohen, *Introduction to Computer Theory*, New York: Wiley, 1986.

T. Sudkamp, *Languages and Machines: An Introduction to the Theory of Computer Science*, Reading, MA: Addison-Wesley, 1988.

D. Wood, *Theory of Computation*, New York: Wiley, 1987.

* Of historical interest:

S. Kleene, "Representation of Events in Nerve Nets and Finite Automata," in C. Shannon and J. McCarthy (eds), *Automata Studies*, Princeton, NJ: Princeton University Press, 1956.

K. Thompson, "Regular Expression Search Algorithm," *Communications of the ACM*, 11: 6 (1968), 419-422.

Historical Notes

Stephen Kleene introduced the notion of regular expressions in one of the fundamental papers of theoretical computer science (see above). The use of regular expressions as a way of describing patterns in strings first appeared in Ken Thompson's QED system (see above). The same ideas later influenced many commands in his UNIX system.