LECTURE 2

Regular Expressions

1 Where are regular expressions used?

Regular expression is a formalism that is used to capture syntactic rules that allow the compiler to take a steam of characters (your program file from the first character to EOF character) and convert it to a stream of tokens (e.g. ID, relation operator, integer). This process is known as lexical analysis. This is the stage before parsing.

2 Background

Since syntactic analysis deals with strings, we need some notation. Alphabet refers to the set of characters using which a string is built. For example, the alphabet for C++ or Java programs is ASCII. We will use Σ to represent the alphabet. If x and y are two strings, xy is concatenation of those strings. Note that concatenation is *not* commutative. The length of a string x is denoted by |x|. The empty string is denoted using the symbol ε. Length of the empty string $|\varepsilon|=0$. Further, x concatenated with ε, either on the left or on the right, does not change x and results in x. Concatenation operation can be extended to sets as follows. If L and M are two sets of strings, then

LM = {xy | x and y are members of L and M respectively}

For example, if L = {5, 7} and M = { b, p, q }, then LM = {5b, 5p, 5q, 7b, 7p, 7q}. Clearly, LM is not the same as ML. In the following, we will be particularly interested in L concatenated with itself. We will use the notation L^2 to represent LL. Similarly, L^i refers to strings that are obtained by taking any *i* strings from L and concatenating them.

A second useful operation on sets of strings is the union. This is simply a set union. For the example above, the union of L and M is denoted by $L \cup M$ = {5, 7, b, p, q }.

Kleene closure of a set L is denoted by L^* and is $\bigcup_{i=0}^{\infty} L^i$. Here L^0 refers to the singleton set (set containing exactly one element) containing the empty string.

Let us look at an example. Let L = {0, 1}. Then, L^* is the union of

$L^0 = \{\varepsilon\}$
$L^1 = \{0, 1\}$
$L^2 = \{00,01,10,11\}$
$L^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
...
In other words, L^* contains all binary strings including the empty string.

3 Regular Expressions: A Definition

Each regular expression *r* represents a set of strings. We will represent this set by L(r) and call it the language represented by *r*. Regular expressions are built out of simpler regular expressions using a set of defining rules. Here is a definition of regular expressions over the alphabet Σ.

1. ε is regular expression that represents $\{\varepsilon\}$
2. for every symbol *a* from the alphabet Σ, *a* is regular expression that represents {a}
3. (the recursive rule) if *r* and *s* are regular expressions representing languages L(*r*) and L(*s*) then,
   a. *r* | *s* is a regular expression that represents $L(r) \cup L(s)$
   b. *r.s* is a regular expression that represents L(*r*)L(*s*)
   c. *r** is a regular expression that represents (L(*r*))*
   d. (*r*) is a regular expression that represents L(*r*)[1]

4 A Look at Some Regular Expressions

4.1 Set of Identifiers

Many programming languages require that the identifiers must begin with a letter (a-z or A-Z) and can be followed by zero or more letters or other characters. (Here the alphabet is ASCII.)

Let *letter* stand for the regular expression (a|b|…|z|A|B|…|Z). Similarly, let *char* stand for any character from the alphabet. Then the regular expression for the set of all valid identifiers is

$$(letter)\ (letter \mid char)^*$$

4.2 No *a* appears after a *b*

Here the alphabet Σ, implicitly, is {a, b}. We are interested in a regular expression for the infinite set

$$\{\ \varepsilon, a, b, ab, bb, aab, abb, bbb, \ldots\}$$

[1] All this rule says is that we can place parenthesis around regular expressions, for readability, without changing the language they represent.

The regular expression for this set is a*b*.

4.3 No Three Consecutive *1*'s

Basically, what must happen here is that either no *1*, or one *1*, or two *1*'s must separate every consecutive pair of *0*'s. Taking into account the boundary conditions, we get

((ε|1|11)0)*(ε|1|11)

4.4 Odd Number of *0*'s or Odd Number of *1*'s

The alphabet here is {0, 1}. Here the `or` condition is an inclusive or. Hence, we can ignore the 1's condition, solve the problem and `or` it with the flip (regular expression where we replace every 0 with 1 and 1 with a 0) of the first part. (1*01*01*)* produces all strings that have an even number of 0's. Concatenating that with a 01^* gives us the set of strings that have an odd number of 0's. The flip of this regular expression is $(0^*10^*10^*)^*10^*$. The final answer is

(1*01*01*)*01*|(0*10*10*)*10*

5 Can Every Set of Strings be Expressed as a Regular Expression?

While regular expressions suffice to capture the syntactic rules concerning tokens, they are *not* expressive enough to encode grammatical rules such as `every open parenthesis must have matching closed one`. In other words, there is no regular expression for the set

{( ), (( )), ((( ))),…}

This fact follows from the fact that regular expressions are equivalent to finite state automata. And in finite state automaton, each state is intuitively represents the state of the recognizer. Since the recognizer has only finite states, it cannot have enough memory to remember the count of how many open parenthesis it has seen. Hence it cannot match the open ones with the closed ones.

In the next class, we will look at a more powerful formalism, context-free grammars, that can be used to encode syntactic rules that regular expressions cannot handle.