Formal Languages

In computer science theory, an *alphabet* is a set of characters and a *formal language* (or just *language*) is a set of strings that is only allowed to use characters from the alphabet. Typically A is used to represent the alphabet and L is used to represent the language.

Example: L={00,01,10,11} is a language over alphabet A={0,1}. All four elements of L are strings using A.

Since a string of length zero is legal but hard to see on paper, λ (a greek lambda) is used to represent it. (Some books use greek epsilon ε instead, so you may sometimes see that.)

In CSC 28 we looked at two tools for specifying languages: the regular expressions (RE) and finite automata (FA). Here is a quick review of the two.

Regular expressions

The *syntax* of a RE is the rules for assembling a legal one.

- If x is in alphabet A, then x is a RE.
- ( ) and λ are REs.
- If R is a RE, then R^* and (R) are both REs.
- If R_1 and R_2 are REs then R_1R_2 and R_1+R_2 are both REs.

Nothing else is a RE. In other words, every RE using alphabet A can be built up by repeatedly applying these rules.

The *semantics* of a RE is the meaning of the RE. The key idea here is that every RE represents a set of strings.

- If x is in alphabet A, then RE x represents the set {x} (ie, a set with one string of length 1).
- ( ) represents a set with no strings in it { }.
- λ represents a set with one length 0 string in it {λ}.
- If R is a RE representing the set of strings X, then (R) represents X too. Parentheses around REs are used only for clarity and/or changing the order of evaluation.
- If R is a RE representing the set of strings X, then R^* represents the set of all strings you can make by taking zero or more copies of strings from X and concatenating them together. (This is called the Kleene star operation.)
- If R_1 and R_2 are REs representing sets of strings X and Y, respectively, then R_1R_2 represents the set of strings you could create by starting with any string from X and following it with any string from Y. (This is called the concatenation operation.)
- If R_1 and R_2 are REs representing sets of strings X and Y, respectively, then R_1+R_2 represents the set of strings $X \cup Y$. (This is called the union operation.)

The order of evaluation is handled similarly to the PEMDAS you learned for algebra. Parenthesis can be used to make it clear or change what order the operations should be applied, but if parentheses are not used, Kleene star has highest precedence (like exponentiation would in standard algebra), concatenation is next (like multiplication would come next in algebra), and union has lowest (like addition would in algebra).

Note! If you have some experience using regular expressions in computer applications, you will notice that this module is not quite the same. REs for computer applications have lots of extra convenience features and slightly different notation. For this module you are restricted to just use the features described above.

Examples

It is fairly simple to take a RE and figure out what set of strings it represents. Just follow these steps.

1. Turn all of the "atomic" REs into the sets they represent (ie, *x*, ( ), and λ to {*x*}, { }, and {λ}).
2. Apply the Kleene star to the sets its attached to.
3. Apply concatenation to pairs of adjacent sets.
4. Union sets that have a + between them.
5. Do these operations in a different order if parenthesis say so.

Example: ba* becomes {b}{a}* after turning a and b into their sets.
{b}{a}* becomes {b}{λ,a,aa,aaa,...} after applying the Kleene star.
{b}{λ,a,aa,aaa,...} becomes {b,ba,baa,baaa,...} after concatenating.
So, the RE ba* represents the set of all strings that are 1 b followed by any number of a's (including zero a's).

Example: (a+b)* becomes ({a}+{b})* becomes {a,b}* becomes {λ,a,b,aa,ab,ba,bb,aaa,aab,aba,abb,...}.
So, (a+b)* represents the set of all strings of all lengths you can make using a and b.

Example: a(a+b)*a+b(a+b)*b becomes {a}({a}+{b})*{a}+{b}({a}+{b})*{b}
which becomes {a}{a,b}*{a}+{b}{a,b}*{b}
which becomes {a}{λ,a,b,aa,ab,ba,bb,...}{a}+{b}{λ,a,b,aa,ab,ba,bb,...}{b}
which becomes {a,aa,ab,aaa,aab,aba,abb,...}{a}+{b,ba,bb,baa,bab,bba,bbb,...}{b}
which becomes {aa,aaa,aba,aaaa,aaba,abaa,abba,...}+{bb,bab,bbb,baab,babb,bbab,bbbb,...}
which becomes {aa,bb,aaa,aba,bab,bbb,aaaa,aaba,abaa,abba,...,baab,babb,bbab,bbbb,...}.
So a(a+b)*a+b(a+b)*b represents the set of all strings of length 2 or more that begin and end in the same letter.

Two things I want you to notice in these examples. Kleene star always includes λ because you are allowed to take zero items from the set, and by convention this means the empty string. Also, I'm listing my sets in length order: shorter strings are listed first and same length strings are listed in sorted order (I will ask you to do this on quizzes too).

Finite Automata

The formal definition of a finite automata (sometimes called a deterministic finite automata and abbreviated FA or DFA) has five parts.

A, and alphabet
S, a finite set of state labels
s_i, a start state that must be an element of S
Y, a set if final or "accept" states that must be a subset of S
$F : S \times A \rightarrow S$, a function that maps a state and an alphabet character to a state

The way a FA works is that it's presented with a string over the alphabet as input and the FA outputs "accept" or "reject" according to the following algorithm.

```
run_FA(input):
  cur_state = si      // the start state
  for each character c in input:
     cur_state = F(cur_state, c)
  if cur_state is in Y
     output "accept"
  else
     output "reject"
```

The way a FA specifies a language is that we say "the language of the FA" is the set of strings for which it outputs "accept". If *M* is a FA and *M*(*s*) represents the output of *M* when given *s* as input, then the language defined by *M*, is expressed using set notation as *L*(*M*) = { s | *M*(*s*) == "accept" }.

The definition above, where FA *M* is defined as a five-tuple (*A*, *S*, s_i, *Y*, *F*) is quite abstract and mathematical. That's how it needs to be if you want to, for example, have a computer simulate the FA. But humans do better with graphical representations of FA, which you will see in other videos and documents in this module.

Generators and Recognizers

In a way you can think of a RE as a "generator" of a language because our algorithm for determining the meaning of a RE starts with a compact notation and generates a sometimes much larger set of strings.

On the other hand, a FA doesn't generate anything. But an FA does have the ability to recognize strings by outputting "accept". In this sense it "recognizes" a particular language.

Something you will look at more closely in CSC 135, is the fact that REs and FAs have the same expressive power. For every language that a RE can generate, there is a FA that recognizes it, and vice versa.

Notation

When talking about strings and sets of strings, the following notations are sometimes used.

An *alphabet* is a set of arbitrary symbols. It's usually alphanumeric, so above I called the contents characters. But, technically they could be any symbols, like *A* = {♣,♠,♥,♦}. To keep the problem size small, alphabets used for teaching tend to be small, like {0,1}, {a,b}, or {a,b,c}, but in real life they are often quite large like the Unicode character set which contains thousands of characters, symbols and emojis.

A *string* is a sequence of symbols from the alphabet. I use λ to represent the length 0 string. Don't confuse λ, { }, and {λ}. You can differentiate them in your mind based on their type and/or size. λ is a string, { } is a set with 0 elements, {λ} is a set with 1 element.

If *s* and *t* are strings, then *st* is their concatenation. So if *s* = abc and *t* = def, then *st* = abcdef.

If *s* is a string or symbol, s^n is *s* repeated *n* times. So, a^3 = aaa and $(ab)^3$ = ababab.

If *S* and *T* are sets of strings, then *ST* = { *st* | *s* ∈ *S* and *t* ∈ *T* } (ie, all strings that can be made from one element of *S* followed by one element of *T*). For example, {ab,bc}{a,b} = {aba, abb, bca, bcb}.

If S is a set of strings or symbols, S^* = {λ} ∪ { s | $s \in S$ } ∪ { st | $s,t \in S$ } ∪ { stu | $s,t,u \in S$ } ∪ ... (ie, S^* is the set of strings that can be made by copying zero or more elements of S and concatenating them). For example $\{a,b\}^*$ = {λ, a, b, aa, ab, ba, bb, aaa, ...} and $\{a,ab\}^*$ = {λ, a, aa, ab, aaa, aab, aba, aaaa, ...}. Sometimes you see this notation to assert that a string s is over a particular alphabet, $s \in A^*$.

If M is a recognizer (such as a DFA) then $L(M)$ is the set of strings accepted by M. If R is a generator (such as a regular expression) then $L(R)$ is the set of strings that can be generated by R.