Regular Sets and Expressions

Finite automata are important in science, mathematics, and engineering. Engineers like them because they are superb models for circuits (And, since the advent of VLSI systems sometimes finite automata *are* circuits!) Computer scientists adore them because they adapt very nicely to algorithm design, for example the lexical analysis portion of compiling and translation. Mathematicians are intrigued by them too due to the fact that there are several nifty mathematical characterizations of the sets they accept. This is partially what this section is about.

We shall build expressions from the symbols 0, 1, +, and & using the operations of union, concatenation, and Kleene closure. Several intuitive examples of our notation are:

a) 01 means a zero followed by a one (concatenation)
b) 0+1 means either a zero or a one (union)
c) 0^* means ^ + 0 + 00 + 000 + ... (Kleene closure)

With parentheses we can build larger expressions. And we can associate meanings with our expressions. Here's how:

Expression	**Set Represented**
$(0+1)^*$	**all strings over {0,1}.**
$0^*10^*10^*$	**strings containing exactly two ones.**
$(0+1)^*11$	**strings which end with two ones.**

That is the intuitive approach to these new expressions or formulas.
Now for a precise, formal view. Several definitions should do the job.

Definition. *0, 1, ε, and $\varnothing$ are* ***regular expressions.***

Definition. *If α and β are regular expressions, then so are*

$$(\alpha\beta),\ (\alpha + \beta),\ \text{and}\ (\alpha)^*.$$

OK, fine. Regular expressions are strings put together with zeros, ones, epsilons, stars, plusses, and matched parentheses in certain ways. But why did we do it? And what do they mean? We shall answer this with a list of what various general regular expressions represent. First, let us define what some specific regular expressions represent.

a) 0 represents the set {0}
b) 1 represents the set {1}
c) ε represents the set {ε} (the empty string)
d) $\emptyset$ represents the empty set

Now for some general cases. If α and β are regular expressions representing the sets A and B, then:

a) $(\alpha\beta)$ represents the set AB
b) $(\alpha + \beta)$ represents the set $A \cup B$
c) $(\alpha)^*$ represents the set A^*

The sets which can be represented by regular expressions are called *regular sets*. When writing down regular expressions to represent regular sets we shall often drop parentheses around concatenations. Some examples are $11(0 + 1)^*$ (the set of strings beginning with two ones), 0^*1^* (all strings which contain a possibly empty sequence of zeros followed by a possibly null string of ones), and the examples mentioned earlier. We also should note that {0,1} is not the only alphabet for regular sets. Any finite alphabet may be used.

From our precise definitions of the regular expressions and the sets they represent we can derive the following nice characterization of the regular sets. Then, very quickly we shall relate them to finite automata.

Theorem 1. *The class of regular sets is the smallest class containing the sets {0}, {1}, {ε}, and $\emptyset$ which is closed under union, concatenation, and Kleene closure.*

See why the above characterization theorem is true? And why we left out the proof? Anyway, that is all rather neat but, what exactly does it have to do with finite automata?

Theorem 2. *Every regular set can be accepted by a finite automaton.*

Proof. The singleton sets {0}, {1}, {ε}, and $\emptyset$ can all be accepted by finite automata. The fact that the class of sets accepted by finite automata is closed under union, concatenation, and Kleene closure completes the proof.

Just from closure properties we know that we can build finite automata to accept all of the regular sets. And this is indeed done using the constructions from the theorems. For example, to build a machine accepting $(a + b)a^*b$, we design:

> M_a which accepts {a},
> M_b which accepts {b},
> M_{a+b} which accepts {a, b} (from M_a and M_b),
> M_{a*} which accepts a^*,
> and so forth

until the desired machine has been built. This is easily done automatically, and is not too bad after the final machine is reduced. But it would be nice though to have some algorithm for converting regular expressions directly to automata. The following algorithm for this will be presented in intuitive terms in language reminiscent of language parsing and translation.

Initially, we shall take a regular expression and break it into subexpressions. For example, the regular expression $(aa + b)^*ab(bb)^*$ can be broken into the three subexpressions: $(aa + b)^*$, ab, and $(bb)^*$. (These can be broken down later on in the same manner if necessary.) Then we number the symbols in the expression so that we can distinguish between them later. Our three subexpressions now are: $(a_1a_2 + b_1)^*$, a_3b_2, and $(b_3b_4)^*$.

Symbols which lead an expression are important as are those which end the expression. We group these in sets named *FIRST* and *LAST*. These sets for our subexpressions are:

Expression	***FIRST***	***LAST***
$(a_1a_2 + b_1)^*$	a_1 , b_1	a_2 , b_1
a_3b_2	a_3	b_2
$(b_3b_4)^*$	b_3	b_4

Note that since the *FIRST* subexpression contained a union there were two symbols in its *FIRST* set. The *FIRST* set for the entire expression is: $\{a_1 , a_3 , b_1\}$. The reason that a_3 was in this set is that since the first subexpression was starred, it could be skipped and thus the first symbol of the next subexpression could be the first symbol for the entire expression. For similar reasons, the *LAST* set for the whole expression is $\{b_2 , b_4\}$.

Formal, precise rules do govern the construction of the *FIRST* and *LAST* sets. We know that *FIRST*(a) = {a} and that we always build *FIRST* and *LAST* sets from the bottom up. Here are the remaining rules for *FIRST* sets.

Definition. *If α and β are regular expressions then:*

a) $\text{FIRST}(\alpha + \beta) = \text{FIRST}(\alpha) \cup \text{FIRST}(\beta)$

b) $\text{FIRST}(\alpha^*) = \text{FIRST}(\alpha) \cup \{\varepsilon\}$

c) $$\text{FIRST}(\alpha\beta) = \begin{cases} \text{FIRST}(\alpha) \text{ if } \varepsilon \notin \text{FIRST}(\alpha) \\ \text{FIRST}(\alpha) \cup \text{FIRST}(\beta) \text{ otherwise} \end{cases}$$

Examining these rules with care reveals that the above chart was not quite what the rules call for since empty strings were omitted. The correct, complete chart is:

Expression	***FIRST***	***LAST***
$(a_1a_2 + b_1)^*$	a_1, b_1, ε	a_2 , b_1, ε
a_3b_2	a_3	b_2
$(b_3b_4)^*$	b_3, ε	b_4, ε

Rules for the *LAST* sets are much the same in spirit and their formulation will be left as an exercise.

One more notion is needed, the set of symbols which might follow each symbol in any strings generated from the expression. We shall first provide an example and explain in a moment.

Symbol	a_1	a_2	a_3	b_1	b_2	b_3	b_4
FOLLOW	a_2	a_1, a_3, b	b_2	a_1, a_3, b_1	b_3	b_4	b_3

Now, how did we do this? It is almost obvious if given a little thought. The *FOLLOW* set for a symbol is all of the symbols which could come next. The algorithm goes as follows. To find *FOLLOW*(a), we keep breaking the expression into subexpressions until the symbol a is in the *LAST* set of a subexpression. Then *FOLLOW*(a) is the *FIRST* set of the next subexpression. Here is an example. Suppose that we have $\alpha\beta$ as our expression and know that $a \in$ *LAST*(α). Then *FOLLOW*(a) = *FIRST*(β). In most cases, this is the way it we compute *FOLLOW* sets.

But, there are three exceptions that must be noted.

1) If an expression of the form $a\gamma^*$ is in α then we must also include the *FIRST* set of this starred subexpression γ.

2) If α is of the form β^* then *FOLLOW*(a) also contains α's *FIRST* set.

3) If the subexpression to the right of α has an ε in its *FIRST* set, then we keep on to the right unioning *FIRST* sets until we no longer find an ε in one.

Another example. Let's find the *FOLLOW* set for b_1 in the regular expression $(a_1 + b_1a_2^*)^*b_2^*(a_3 + b_3)$. First we break it down into subexpressions until b_1 is in a *LAST* set. These are:

$$(a_1 + b_1\, a_2{}^*)^* \qquad\qquad b_2{}^* \qquad\qquad (a_3 + b_3)$$

Their *FIRST* and *LAST* sets are:

Expression	*FIRST*	*LAST*
$(a_1 + b_1a_2^*)^*$	a_1, b_1, ε	$a_1, b_1, a_2, \varepsilon$
b_2^*	b_2, ε	b_2, ε
$(a_3 + b_3)$	a_3, b_3	a_3, b_3

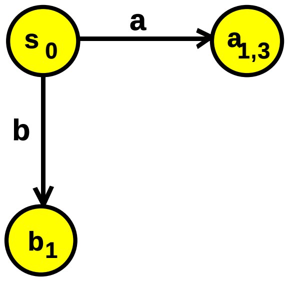
Since b_1 is in the *LAST* set of asubexpression which is starred then we place that subexpression's *FIRST* set $\{a_1, b_1\}$ into *FOLLOW*(b_1). Since a_2^* came after b_1 and was starred we must include a_2 also. We also place the *FIRST* set of the next subexpression (b_2^*) in the *FOLLOW* set. Since that set contained an ε, we must put the next *FIRST* set in also. Thus in this example, all of the *FIRST* sets are combined and we have:

$$FOLLOW(b_1) = \{a_1, b_1, a_2, b_2, a_3, b_3\}$$

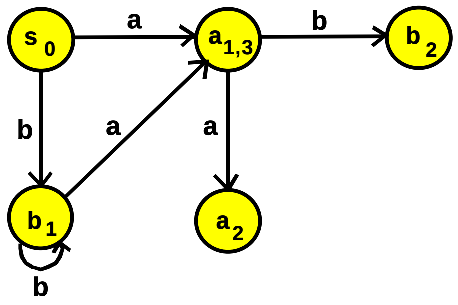
Several other *FOLLOW* sets are:

$$FOLLOW(a_1) = \{a_1, b_1, b_2, a_3, b_3\}$$
$$FOLLOW(b_2) = \{b_2, a_3, b_3\}$$

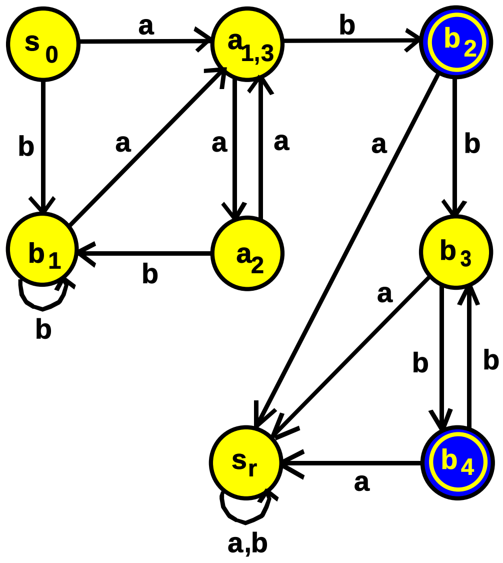
After computing all of these sets it is not hard to set up a finite automaton for any regular expression. Begin with a state named s_0. Connect it to states denoting the *FIRST* sets of the expression. (By *sets* we mean: split the *FIRST* set into two parts, one for each type of symbol.) Our first example $(a_1a_2 + b_1)^*a_3b_2(b_3b_4)^*$ provides:



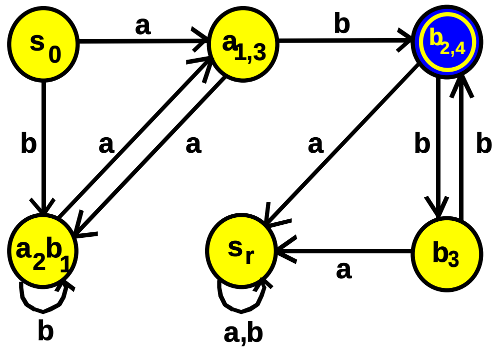
Next, connect the states just generated to states denoting the *FOLLOW* sets of all their symbols. Again, we have:



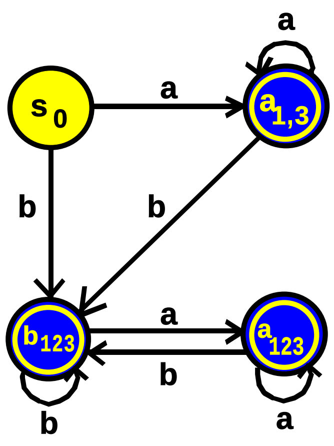
Continue on until everything is connected. Any edges missing at this point should be connected to a rejecting state named s_r. The states containing symbols in the expression's *LAST* set are the accepting states. The complete construction for our example $(aa + b)^*ab(bb)^*$ is:



This construction did indeed produce an equivalent finite automaton, and in not too inefficient a manner. Though if we note that b_2 and b_4 are basically the same, and that b_1 and a_2 are similar, we can easily streamline the automaton to:



Our construction method provides:



for our final example. There is a very simple equivalent machine.
Try to find it!

We now close this section with the equivalence theorem concerning finite automata and regular sets. Half of it was proven earlier in the section, but the translation of finite automata into regular expressions remains. This is not included for two reasons. First, that it is *very* tedious, and secondly that nobody ever actually does that translation for any practical reason! (It is an interesting demonstration of a correctness proof which involves several levels of iteration and should be looked up by the interested reader.)

> **Theorem 3.** *The regular sets are exactly those sets accepted by finite automata.*