Kleene's Theorem states the equivalence of the following three statements:

1. A language is regular (i.e., is represented by a regular expression).
2. A language is accepted by a NDFA.
3. A language is accepted by a FA.

In the textbook by Cohen, he states the theorem using TG’s in place of NDFAs. It makes no difference. We could add a fourth statement to the list, but Kleene did not. In trying to stay close to the text, I will restate Kleene's Theorem using TGs, and also as a set of implications.

Restatement of Kleene's Theorem:

1. If a language is regular, there is a TG that accepts it.
2. If a language is accepted by a TG, then there is a FA that accepts it.
3. If a language is accepted by a FA, then it is regular (i.e., there is a regular expression that defines it).

In these notes, I prove statements 2 and 3 above. The proof of statement 1 is very easy and will be added at a later date.

Proof of 2. For any TG M, there is a FA M' such that L(M') = L(M).

This is a constructive proof. Given a TG M, it defines a FA M' that accepts the same language as the TG.

Let M have states s₁, s₂, ..., sₙ and assume that the set of start states of M is S and the set of final states is F.

First, create a TG M₀ exactly like M except that M₀ has a unique start state. M₀ is identical to M except that it has a new start state, s₀, with Λ-transitions to each state of M that is in S, and the states that were start states in M are non-start states in M₀. Since any word accepted by M can be accepted by M₀ by using a Λ-transition to enter the same start state that would lead to its acceptance in M, and since any word that is accepted by M must be accepted by M since it must first reach a state that is a start state in M, without M₀ reading any letters, L(M₀) = L(M).

Next, let M₁ be exactly the same as M₀ except that it contains no edges labeled by strings of length greater than 1. To do this, first let M₁ be a copy of M₀. Then, for each edge in M₁ that is labeled by a string of length > 1, do the following. Suppose the edge from sᵢ to sⱼ in M₁ is labeled by w = a₁a₂a₃...aₖ, where k > 1. Create k-1 new states in M₁ with unique labels, say t₁, t₂, ..., tₖ₋₁, create the transitions δ(sᵢ,a₁) = t₁, δ(t₁,a₂) = t₂, δ(t₂,a₃) = t₃, ..., δ(tₖ₋₂,aₖ₋₁) = tₖ₋₁, and δ(tₖ₋₁,aₖ) = sⱼ, and delete the edge from sᵢ to sⱼ. Then M₁ accepts the same language as M₀ because M₀ can move from sᵢ to sⱼ on w if and only if M₁ can move from sᵢ to sⱼ on w by entering the new intermediate states.
M₁ is now a NDFA with \( \Lambda \)-transitions, called a NDFA-\( \Lambda \). The next step is to build the FA M' that accepts the same language as M₁. For any state s, define

\[
\Lambda - \text{closure} \{ s \} = \{ t \mid \delta(s, \Lambda) = t \lor \exists u \exists t \in \Lambda - \text{closure} \{ s \} \land \delta(u, \Lambda) = t \}\]

Notice that this is a recursive definition of the \( \Lambda \)-closure. The recursion is embedded within the curly braces, but it is nonetheless recursion. In plain words, the \( \Lambda \)-closure of a state s is the set of states that a NDFA-\( \Lambda \) can enter from s without reading any symbols. Now define the \( \Lambda \)-closure of a set of states S:

\[
\Lambda - \text{closure} \{ S \} = \bigcup_{s \in S} \Lambda - \text{closure} \{ s \}
\]

We can now construct the FA M'. The idea is that the states of M' will be sets of states from M₁. The following pseudo-code algorithm constructs the FA M'.

Let \( s_0 \) be the unique start state of M₁.
Let \( S_0 = \Lambda \)-closure(\( s_0 \)) be the start state of M'.
Let Q denote the collection of states of M'. Add \( S_0 \) to Q and mark it unprocessed.

while there is a state-set S in Q that is unprocessed do
  mark S processed;
  for each input symbol a do
    Let T be the set of all states to which there is a transition on 'a' from some state in S;
    Let \( T_\Lambda = \Lambda \)-closure(T);
    if \( T_\Lambda \) is not in Q then
      add \( T_\Lambda \) to Q and mark it unprocessed;
      add a transition from S to \( T_\Lambda \) labeled 'a';
  For each state-set S in Q, if S contains a final state of M, make S a final state of M'.

Claim: \( L(M') = L(M₁) \).

Proof.

Let \( \delta(s,a) \) denote the transition function of M₁. Since \( M₁ \) is a NDFA, \( \delta(s,a) \) is the set of states that can be entered by \( M₁ \) on reading 'a' in state s. The emphasis is on "set" because the transition function is not the same as that of a FA -- it defines a set. The set \( \delta(s,a) \) includes any states that it can reach by following the \( \Lambda \)-transitions in \( M₁ \). By definition, \( \delta^*(s,w) \) is the set of states that can be reached by \( M₁ \) on reading the string w in state s, again including the possibility that it might have used \( \Lambda \)-transitions.

Let \( \delta_M(S,a) \) denote the transition function of M'. From the algorithm above, the transition function \( \delta_M(S,a) \) is defined by

\[
\delta_M(S,a) = \bigcup_{s \in S} \delta(s,a)
\] (1)
because the definition includes the states entered by all \( \Lambda \) transitions. That is why the \( \Lambda \)-closure is computed at each step.

**Claim:** For any string \( w \),

\[
\delta^*_M(S, w) = \bigcup_{s \in S} \delta^*_s(s, w) \quad \text{(2)}
\]

This can be proved by induction on the length of \( w \). It is true for \( |w| = 0 \) since

\[
\delta^*_M(S, \Lambda) = S = \bigcup_{s \in S} \delta^*_s(s, \Lambda) \quad \text{(3)}
\]

because the states in \( M' \) are their own \( \Lambda \)-closures, so it follows from the definition of \( \Lambda \)-closure. Assume it is true for any \( w \) with \( |w| = m \) and let \( w \) be a word of length \( m+1 \). Then \( w = va \), where \( |v| = m \). Hence

\[
\delta^*_M(S, w) = \delta^*_M(S, va) = \delta^*_M(S, v, a) = \bigcup_{s \in S} \delta^*_m(s, v, a) = \bigcup_{s \in S} \delta^*_s(s, w)
\]

The second step used the definition of \( \delta^*_M \) and the third step applied the inductive hypothesis on \( v \). The fourth step used the definition from (1) (and an implicit step I have not included, but which can be proved easily enough.) The last two steps follow from the definition of \( w \) and \( \delta^* \). It follows that the claim is proved.

Since \( S_0 = \Lambda \)-closure\((s_0) \) is the start state of \( M' \),

\[
\delta^*_M(S_0, w) = \delta^*_s(s_0, w)
\]

Also, since \( w \) is in \( L(M') \) if and only if \( \delta^*_M(S_0, w) \) is a final state, from the above, \( w \) is in \( L(M') \) if and only if \( \delta^*(s_0, w) \) contains a final state in \( M_1 \), which is true if and only if \( w \) is in \( L(M_1) \).

**Proof of 3.** If a language \( L \) is accepted by some FA, then there is a regular expression \( r \) such that \( L = \langle r \rangle \).

Let \( L \) be accepted by an FA \( M \) with states \( s_1, s_2, ..., s_n \). Assume that \( s_1 \) is the start state of \( M \) and that the set of final states of \( M \) is denoted \( F \). Define the set \( L(i,j,k) \) to be the set of all words that cause \( M \), starting in state \( s_i \) to enter state \( s_j \) without passing through any of the states \( s_{k+1}, s_{k+2}, ..., s_n \). In other words, \( L(i,j,k) \) is the set of words that start in \( s_i \) and end in \( s_j \) and **pass through** only
the states \( s_1, s_2, ..., s_k \). "Passing through" means entering and leaving, like one does in a toll booth or turnstile. It does not mean "landing there" and staying there.

The language accepted by \( M \) is the set of all words that cause \( M \), when starting in state \( s_i \), to stop in a final state, passing through any of the states of \( M \). This means that, if \( s_i \) is a final state, then \( L(1,f,n) \) consists only of words accepted by \( M \), and that

\[
L = L(1,f_1, n) \cup L(1,f_2, n) \cup \ldots \cup L(1,f_m, n)
\]

where \( F = \{ s_{f_1}, s_{f_2}, \ldots, s_{f_m} \} \).

From the definition of \( L(i,j,k) \) it follows that, for each \( i \) and \( j \), \( 1 \leq i, j \leq n \), \( L(i,j,0) \) is the set of all symbols that label the transitions from \( s_i \) to \( s_j \), and that in addition, if \( i = j \), then the null string is also in this set. Formally,

\[
L(i, j, 0) = \begin{cases} 
\{ a \mid \delta(\sigma_i, a) = \sigma_j \} \cup \Lambda & \text{if } i = j \\
\{ a \mid \delta(\sigma_i, a) = \sigma_j \} & \text{if } i \neq j 
\end{cases} \tag{5}
\]

Furthermore, for all \( k > 0 \), the set \( L(i,j,k) \) can be defined recursively from the following observation. (I will use the language abusively and talk about a word starting in a state or passing through a state or even visiting a state. What this means of course is that the word causes \( M \) to enter a state while reading it, or causes \( M \) to pass through a state while reading it, and so on.)

1. If a word starts in state \( s_i \) and terminates in state \( s_j \) without going through any states \( s_{k+1}, s_{k+2}, \ldots, s_n \), then it falls into one of two cases:

2. It starts in state \( s_i \) and terminates in state \( s_j \) without going through any states \( s_k, s_{k+1}, \ldots, s_n \), or

It starts in state \( s_i \) and terminates in state \( s_j \) and enters state \( s_k \), and then visits other states without passing through any of \( s_{k+1}, s_{k+2}, \ldots, s_n \), possibly passing through \( s_k \) many times, and then returns to \( s_k \) for the last time, and then travels a path to state \( s_j \).

In short, either the word was already in \( L(i,j,k-1) \) (Case 1) or it is in \( L(i,j,k) \) but not in \( L(i,j,k-1) \), and is there because it passes through state \( s_k \), and we can break the word into 3 pieces: the "left" piece \( x \) that first reaches \( s_k \) without going through any states \( s_k, s_{k+1}, \ldots, s_n \), the "middle" piece \( y \) that travels around \( M \) without going through any states \( s_k, s_{k+1}, \ldots, s_n \) until it visits \( s_k \) for the last time, and the "right" piece \( z \) that reaches \( s_j \) from \( s_k \) without passing through any states \( s_k, s_{k+1}, \ldots, s_n \). Since \( x \) in in \( L(i,k,k-1) \), \( y \) is in \( L(k,k,k-1)^* \) and \( y \) is in \( L(k,j,k-1) \), it follows that

\[
L(i,j,k) = L(i,j,k-1) \cup L(i,k,k-1) \cdot L(k,k,k-1)^* \cdot L(k,j,k-1) \tag{6}
\]

**Claim:** For every \( i \) and \( j \), \( 1 \leq i, j \leq n \), and for every \( k \), \( 0 \leq k \leq n \), the set \( L(i,j,k) \) can be represented by a regular expression.

**Proof.**
We can prove this by induction on k.

For each i and j, the set \( L(i,j,0) \) is a finite set and is therefore regular. Let \( r(i, j, 0) \) denote the regular expression such that \( L(i,j,0) = \langle r(i, j, 0) \rangle \).

Assume that the claim is true for \( k-1 \). Then, for any i and j, there exists a regular expression that we can denote \( r(i,j,k-1) \) such that \( L(i,j,k-1) = \langle r(i,j,k-1) \rangle \). From formula (6) and the induction hypothesis it follows that

\[
L[i, j, k] = \langle r[i, j, k-1] \rangle + \langle r[i, k, k-1] \rangle \cdot \langle r[k, j, k-1] \rangle^* \cdot \langle r[k, j, k-1] \rangle
\]

where each of \( r(i,j,k-1), r(i,k,k-1), r(k,k,k-1), \) and \( r(k, j, k-1) \) is a regular expression. Since the right hand side is a regular expression, it follows that \( L(i,j,k) \) is a regular language, and that we can let \( r(i,j,k) \) denote the regular expression that defines it. By the axiom of induction, it is true for all \( k \geq 0 \). Of course, for \( k > n \), the sets do not change since there are no states in the FA numbered higher than \( s_n \), so although in principle all of these sets exist, we are only concerned about the ones for which \( k \leq n \). QED.

The truth of the theorem follows from formulas (4) and (7). Formula (4) states that \( L \) is a finite union of the sets \( L(1,s,n) \) for which \( s \) is a final state of \( M \), and formula (7) states that each of the sets \( L(1,s,n) \) can be represented by regular expressions, so that

\[
L = \langle r(1, f_1, n) + r(1, f_2, n) + \ldots + r(1, f_m, n) \rangle
\]

proving that \( L \) is a regular expression. QED.

The proof of the theorem implicitly defines a tabular algorithm that can be used to construct the regular expression. It also suggests a recursive function that can be used to construct the expression. The most efficient solution, however, would be a dynamic programming solution, combining the simple and inefficient table-driven approach with the recursive solution. I will not describe that algorithm here. For now, I present a recursive algorithm, written in C with pseudo-code.

Let \( M \) have states 1, 2, 3, ..., n. Assume the alphabet is \( \Sigma \). Assume that \( \delta(i,a) \) is the transition function, which can also be represented by a 2D matrix \( \delta[i,a] \).

The main function is \( \text{BuildRE()} \), which takes the FA, and integers i, j, and k, and constructs a string \( re \) that contains the regular expression, fully parenthesized to avoid possible ambiguities. The FA is used inside the function in pseudo-code that looks up all symbols that cause a transition from state i to state j. I leave out necessary declarations and such.
void BuildRE ( FA M, int i, int j, int k, char re[] )
{
    char re1[MAXSIZE];
    char re2[MAXSIZE];
    char re3[MAXSIZE];
    char re4[MAXSIZE];

    if ( k == 0 ) {
        re = { a in SIGMA | M.delta(i,a) == j };
        if ( i == j )
            re = re + 'LAMBDA';
    } else { // k > 0
        BuildRE ( M, i, j, k-1, re1);
        BuildRE ( M, i, k, k-1, re2);
        BuildRE ( M, k, k, k-1, re3);
        BuildRE ( M, k, j, k-1, re4);
        sprintf (re, "(%s)+(%s)(%s)*(%s)", re1, re2, re3, re4 );
    }
}

The main program is simply

void main ()
{
    sprintf(re, "()" );
    for ( i = 1; i <= n; i++ )
        if ( finalstate(i) ) {
            BuildRE( M, 1, i, n, temp_re);
            sprintf(re, "(%s)+(%s)", re, temp_re);
        }
    printf("%s\n", re);
}
Example

We will build the regular expression for the FA below using a table-driven method.

```
Example
We will build the regular expression for the FA below using a table-driven method.
```

```
<table>
<thead>
<tr>
<th>i,j</th>
<th>k</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i,j</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1,1</td>
<td>Λ</td>
<td>Λ</td>
</tr>
<tr>
<td>1,2</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>1,3</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>2,1</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>2,2</td>
<td>Λ</td>
<td>Λ</td>
</tr>
<tr>
<td>2,3</td>
<td>b</td>
<td>b + ab</td>
</tr>
<tr>
<td>3,1</td>
<td>Φ</td>
<td>Φ</td>
</tr>
<tr>
<td>3,2</td>
<td>a + b</td>
<td>a + b</td>
</tr>
<tr>
<td>3,3</td>
<td>Λ</td>
<td>Λ</td>
</tr>
</tbody>
</table>

Note. L(1,3,3) is simplified from a*b + a*b(Λ + (a+b)a*b)* ( Λ + (a+b)a*b). There is no need to calculate any other parts of the table. Since L(M) is the union of L(1,2,3) and L(1,3,3), the final expression is

L(M) = a(aa)* + a*b ( (a+b) a*b )* ( Λ + (a+b)(aa)* )