REGULAR EXPRESSIONS

ERNesto Gomez

We define a Regular Expression (R.E.) inductively as follows:

Given an alphabet \( \Sigma \):

1. \( a \in \Sigma \) is a regular expression, denoting the set < \( a \) >.
2. \( \varepsilon \) is a regular expression, denoting the set < \( \varepsilon \) >.
3. \( \emptyset \) is a regular expression, denoting the empty set \( \emptyset \).
4. Union: If \( A \) and \( B \) are regular expressions, \( A \cup B \) is a regular expression.
5. Concatenation: If \( A \) and \( B \) are regular expressions, \( AB \) is a regular expression.
6. Star: If \( A \) is a regular expression, \( A^* \) is a regular expression.
7. Nothing else is a regular expression.

1. NFA AND DFA

We define a Non-deterministic Finite Automaton (NFA) as a 5-tuple:

\[ N = (\Sigma, Q, \tau, q_0, F) \] is an NFA, where:
- \( \Sigma \) is an alphabet.
- \( Q \) is a finite set of states.
- \( \tau : Q \times \Sigma \to 2^Q \) is a transition function from the cartesian product of the set of states with the alphabet to the power set of the set of states (The power set of \( Q \) is the set of all subsets of \( Q \)).
- \( q_0 \) is a single initial state.
- \( F \subseteq Q \) is the set of final states.

A Deterministic Finite Automaton (DFA) differs from an NFA only in the transition function: \( \tau : Q \times \Sigma \to Q \). Note that \( Q \subset 2^Q \), therefore a DFA is a restricted case of NFA.

2. CONSTRUCTION OF NFA FROM R.E.

Construction based on notes from Ernst Leiss, Department of Computer Science, University of Houston. The text gives a similar approach in section 3.7 (page 121). The construction given here has the advantage that it avoids the need for \( \varepsilon \) transitions.

The graphs are as described in the text. We use the following format for transition tables:

<table>
<thead>
<tr>
<th>state number</th>
<th>input symbol</th>
<th>accept</th>
</tr>
</thead>
</table>

Date: January 12, 2004.
The start state is always numbered 0. "New state" is the number of the state to transition to on the given "input symbol", "a" denotes a blank entry, indicating that there is no transition.

2.1. Basis: <a>:

\[ \begin{array}{c|c|c|c} \hline \text{state} & a & \text{not a} & \text{accept} \\ \hline 0 & 1 & - & T \\ 1 & - & - & F \\ \hline \end{array} \]

<ε>:

States: \( Q = \{q_0\} \), no transitions. Accepting states: \( F = \{q_0\} \)

∅:

States: \( Q = \{q_0\} \), no transitions. Accepting states: \( F = \emptyset \) (no accepting states).

Table:

\[ \begin{array}{c|c|c} \hline \text{state} & a & \text{accept} \\ \hline 0 & - & T \\ \hline \end{array} \]
2.2. **Induction:** $\Sigma$ is the union of the alphabets of all the NFAs in the following operations.

**Union:** $N_1 = N_\alpha \cup N_\beta$

Description: Merge the initial states of the two automata into a single initial state. The transitions out of the combined initial state are the same as the transitions out of the two original initial states. The automata are otherwise unchanged.

- $Q_1 = (Q_\alpha - q_0 \cup Q_\beta - q_0) + q_0$
- $q_0$ = combination of $q_0 \alpha$ and $q_0 \beta$
- $\tau$: \{ $\tau(0_\alpha, a) = \tau_\alpha(q_0, a) \cup \tau_\beta(q_0, a)$
- $F_1 = F_\alpha \cup F_\beta$

**Algorithm 2.1.** Given: two tables, $T_1$ and $T_2$ representing finite automata $A$ and $B$, to combine into a table $T_3$ representing $A \cup B$.

Assume all states in each table are numbered starting from 0. Let $N_1$ be the highest numbered state in $T_1$. If $K$ is the number of a state in $T_2$, replace number $K$ with $K + N_1$.

Now construct $T_3$ as follows:

1. Start with a copy of $T_1$, with the same states and column entries for all columns labeled with input symbols.
2. If there is an input symbol $x$ that appears as the label of a column in $T_2$ that is not in $T_1$, add a column to $T_3$ and label it with that symbol $x$. Enter a “$\$” in the new column $x$ for all rows representing states in $T_1$.
3. Add rows for each state in $T_2$ other than state 0, with the same entries for all columns labeled with input symbols. If some symbol $y$ appeared as a column label in $T_1$ and not in $T_2$, enter a “$\$” in all such columns.
4. Copy all non-blank entries from columns labeled by input symbols in row 0 of $T_2$ into the corresponding column in row 0 of $T_3$. If some column in row 0 of $T_3$ already has a non-blank entry (that is, there is already a state number there), add the new state number separated by a comma.
5. Enter $T$ in the accept column for any state in $T_3$ copied from either $T_1$ or $T_2$ that had $T$ in its accept column, enter $F$ in the accept column of $T_3$ for all other states.

**Concatenation:** $N_2 = N_\alpha N_\beta$

Description: eliminate the initial state of $N_\beta$. Connect each of the final states of $N_\alpha$ to every one of the states of $N_\beta$ that had transitions from the initial state of $N_\beta$, with transitions on the same characters that are on transitions from $q_0 \beta$. The final states of $N_2$ are the final states of $N_\beta$; if $N_\beta$ accepts $\varepsilon$ then the final states of $N_2$ also include the final states of $N_\alpha$.

- $Q_2 = Q_\alpha \cup Q_\beta - q_0 \beta$
- $q_0 = q_0 \alpha$
- $\tau$: \{ $\tau_\alpha(q_0, a) \cup \tau_\beta(q_0, a)$
- $F_2 = F_\beta$ if $N_\beta$ does not accept $\varepsilon$,
- $F_2 = F_\alpha \cup F_\beta$ if $N_\beta$ accepts $\varepsilon$.

**Algorithm 2.2.** Given: two tables, $T_1$ and $T_2$ representing finite automata $A$ and $B$, to combine into a table $T_3$ representing $AB$. Assume all states in each
table are numbered starting from 0. Let \( N_1 \) be the highest numbered state in \( T_1 \). If \( K \) is the number of a state in \( T_2 \), replace \( K \) with \( K + N_1 \).

Now construct \( T_3 \) as follows:
1. Start with a copy of \( T_1 \), with the same states and column entries for all columns labeled with input symbols. Copy all \( T \) and \( F \) entries from the accept column into the new table.
2. If there is an input symbol \( x \) that appears as the label of a column in \( T_2 \) that is not in \( T_1 \), add a column to \( T_3 \) and label it with that symbol \( x \). Enter a “\( " \) in column \( z \) for all rows representing states in \( T_1 \).
3. Add rows for each state in \( T_2 \) other than state 0, with the same entries for all columns labelled with input symbols. If some symbol \( y \) appeared as a column label in \( T_1 \) and not in \( T_2 \), enter a “\( " \) in all such columns.
4. If state 0 in \( T_2 \) has a non-blank entry in the column for some input symbol \( z \), copy that entry into the row labeled \( z \) for all states of \( T_1 \) marked \( T \) in the accepts column. If some column and row of \( T_3 \) already has a non-blank entry (that is, there is already a state number there), add the new state number separated by a comma.
5. Enter \( T \) in the accept column of \( T_3 \) for any state of \( T_2 \) that had \( T \) in its accept column. If state 0 of \( T_2 \) was not an accepting state, replace the \( T \) entry in all states copied from \( T_1 \) with \( F \), else leave them unchanged. Enter \( F \) in the accept column of \( T_3 \) for any states not otherwise marked.

**Star:** \( N_3 = N^* \)

Description: If \( q_0 \) is not an accepting state, make it an accepting state. Connect each accepting state to the same states which can be reached from \( q_0 \), by transitions on the same characters that are on the transitions from \( q_0 \).
- \( Q_3 = Q \)
- \( q_0 = q_0 \)
- \( \tau_1 : \{ \tau(q,a) ; q \in Q - F \} \)
- \( \tau(q,a) \cup \tau(q_0,a) ; q \in F \}
- \( F_3 = F \cup \{q_0\} \)

**Algorithm 2.3.** Given a table \( T_1 \) representing a finite automaton \( A \), construct a table \( T_2 \) representing \( A^* \).
1. Copy table \( T_1 \) into table \( T_2 \).
2. If a state \( K \) is marked \( T \) in the accept column of \( T_1 \), copy all non-blank entries in the input symbol columns of state 0 into the corresponding column of state \( K \). If some column and row already has a non-blank entry (that is, there is already a state number there), add the new state number separated by a comma.
3. If a state is marked \( T \) in the accept column of \( T_1 \), mark it \( T \) in the accept column of \( T_2 \). If state 0 of \( T_2 \) is not already marked as an accepting state, mark it as \( T \) in the accept column of \( T_2 \). Mark the accept column of all states not already marked with \( F \).

2.3. **Construction of NFA from R.E.** Identify basis items in the expression (letters in \( \Sigma, \varepsilon, \emptyset \)), draw graphs from the basis set for each basis element. Combine pairs of elements using the operations in the recursive construction. Combine the resulting NFAs. Repeat until the entire expression is a single NFA.

2.4. **DFA and minimal DFA:** See algorithm 3.2 on page 118 of text, algorithm 3.6 on page 142.
2.5. **Equivalence of Regular Expression and Finite Automata.** The above algorithms, taken together, constitute a proof that, for every regular expression there is a finite automaton. We have also proved the equivalence of NFA and DFA by providing algorithms to convert from NFA to DFA. It is not necessary to show that from a DFA we can produce an equivalent NFA, because every DFA can be considered to be an NFA in the limiting case in which every combination of state and symbol has only one possible path.

For our purposes in compiler construction, it is sufficient to show the above. However, for completeness, we also need to consider conversion of a finite automaton to a regular expression. See Hopcroft and Ullman, *Introduction to Automata Theory, Languages and Computation*, Theorem 2.4 for a proof that from every DFA we can produce a regular expression.

An alternative way of convincing ourselves that finite automata can always be converted to regular expressions is to devise an algorithm that does this. Following is an approach to such an algorithm.

We represent a FA as a directed graph $G = \{V, A\}$ where states $q_i$ are vertices $V$, paths $q_i \to q_j$ are arcs in $A$, and we annotate the paths with the symbols that cause a transition along that path.

A depth-first traverse of $G$ can be used to generate a regular expression that corresponds to the particular FA. Multiple symbols along a path are related by $\lor = |$ in the regular expression, multiple paths from a state are also connected by $\lor$. The $*$ operator is applied when a path $q_i \to q_j$ connects to a state that is on the stack, to all the symbols (on the stack) between $q_j$ and $q_i$.

Try to devise an algorithm that accomplishes this - consider pushing states and symbols onto a stack while traversing the graph, constructing the regular expression during the backtracking phase of the traverse when popping symbols off the stack.