A Proof of Kleene's Theorem

Rance Cleaveland

Spring 2000

1. Introduction

So far in class, we have concentrated on two classes of languages: the regular languages, which are those that may be defined using regular expressions, and the FA languages, which are those that are accepted by finite automata. Kleene's Theorem states that, in fact, these classes are the same: every regular language may be recognized by some FA, and every FA language may be represented using a regular expression. The book presents one proof of these statements; this handout offers an alternative, and I hope simpler, argument for the former.

On the basis of what we have seen in class, to establish that every regular language can be recognized by an FA it suffices to show how, given a regular expression r, we can build a NFA M such that $\mathcal{L}(r) = \mathcal{L}(M)$. We give such a construction in a couple of steps.

1. We first define a predicate $\surd$ on regular expressions; intuitively, $r\surd$ is intended to hold if $\varepsilon \in \mathcal{L}(r)$.

2. We then define a *ternary* relation $\longrightarrow \subseteq \mathcal{R}(\Sigma) \times \Sigma \times \mathcal{R}(\Sigma)$. Intuitively, $r \xrightarrow{a} r'$ is true if the start state for a NFA for r can have an a-transition to the start state for a NFA for r'. Put differently, if $r \xrightarrow{a} r'$ then a NFA for r should be able to "process" symbol a and then accept all the strings in $\mathcal{L}(r')$.

3. Using these relations, we then show how to build a NFA from r whose states are regular expressions, whose transitions are given by $\longrightarrow$, and whose final states are defined using $\surd$.

2. Formal Definitions

We define $\surd$ recursively using the following rules.

Definition 2.1. Let Σ be an alphabet. Then $r\surd$, where $r \in \mathcal{R}(\Sigma)$, is defined as follows.

- $\varepsilon\surd$.
- $r^*\surd$.

- If $r\surd$ then $(r+s)\surd$ and $(s+r)\surd$.
- If $r\surd$ and $s\surd$ then $(rs)\surd$.

Intuitively, $r\surd$ holds if r is capable of "generating" the empty word, i.e. if $\varepsilon \in \mathcal{L}(r)$. Certainly $\varepsilon \in \mathcal{L}(\varepsilon)$, and $\varepsilon \in \mathcal{L}(r^*)$ regardless of what r is. The definition of $\mathcal{L}(r+s)$ ensures that ε is in the language of $r+s$ if and only if it is in the language of r or s, while in the case of rs ε must be in the language of both. As examples, we have the following.

$$\begin{array}{ll}
\varepsilon a^*\surd & \text{since } \varepsilon\surd \text{ and } a^*\surd. \\
\neg((a+b)\surd) & \text{since neither } a\surd \text{ nor } b\surd. \\
01+(1+01)^*\surd & \text{since } (1+01)^*\surd. \\
\neg(01(1+01)^*\surd) & \text{since } \neg(01\surd).
\end{array}$$

We also use recursion to define $\longrightarrow$.

Definition 2.2. Let Σ be an alphabet. Then for $r, r' \in \mathcal{R}(\Sigma)$ and $a \in \Sigma$, $r \xrightarrow{a} r'$ is defined as follows.

- If $a \in \Sigma$ then $a \xrightarrow{a} \varepsilon$.
- If $r \xrightarrow{a} r'$ then $r+s \xrightarrow{a} r'$.
- If $s \xrightarrow{a} s'$ then $r+s \xrightarrow{a} s'$.
- If $r \xrightarrow{a} r'$ then $rs \xrightarrow{a} r's$.
- If $r\surd$ and $s \xrightarrow{a} s'$ then $rs \xrightarrow{a} s'$.
- If $r \xrightarrow{a} r'$ then $r^* \xrightarrow{a} r'(r^*)$.

The definition of this relation is somewhat complex, but the idea that it is trying to capture is relatively simple: $r \xrightarrow{a} r'$ if one can build words in $\mathcal{L}(r)$ by taking the a labeling $\longrightarrow$ and appending a word from $\mathcal{L}(r')$. So we have the rule $a \xrightarrow{a} \varepsilon$ for $a \in \Sigma$, while the rules for $+$ follow from the fact that $\mathcal{L}(r+s) = \mathcal{L}(r) \cup \mathcal{L}(s)$. The rules for rs in essence state that $ax \in \mathcal{L}(rs)$ can hold either if there is a way of splitting x into x_1, x_2 such that ax_1 is in the language of r and x_2 is in the language of s or if ε is in the language of r and ax is in the language of s. Finally, the rule for r^* essentially permits such regular expressions to "loop". As examples, we have the following.

$$\begin{array}{ll}
0+1 \xrightarrow{0} \varepsilon & \text{by the rules for 0 and } +. \\
(abb+a)^* \xrightarrow{a} \varepsilon bb(abb+a)^* & \text{by the rules for } a\text{, concatenation, } +\text{, and } ^*.
\end{array}$$

In this latter example, note that applying the rules literally requires that we include the ε in $\varepsilon bb(abb+a)^*$. This is because the rule for a says that $a \xrightarrow{a} \varepsilon$, meaning that $abb \xrightarrow{a} \varepsilon bb$, etc. However, when we have εs like this, we will often leave them out; thus we will write $abb \xrightarrow{a} bb$ rather than $abb \xrightarrow{a} \varepsilon bb$.

The following lemmas about $\surd$ and $\longrightarrow$ formally establish the intuitive properties that we wish them to have.

Lemma 2.3. *Let r be a regular expression. Then $r\surd$ if and only if $\varepsilon \in \mathcal{L}(r)$.*

Lemma 2.4. *Let $r \in \mathcal{R}(\Sigma)$ be a regular expression over Σ, $a \in \Sigma$, and $x \in \Sigma^*$. Then $ax \in \mathcal{L}(r)$ if and only if there is an $r' \in \mathcal{R}(\Sigma)$ such that $r \xrightarrow{a} r'$ and $x \in \mathcal{L}(r')$.*

Both lemmas may be proved using strong induction on the size of regular expression r.

3. Building Automata using $\surd$ and $\longrightarrow$

To see how $\surd$ and $\longrightarrow$ may be used to build NFAs, first note how we can use them to determine whether a string is in the language of a regular expression. Consider the following sequence of "transitions" starting from the regular expression $(abb+a)^*$.

$$(abb+a)^* \xrightarrow{a} bb(abb+a)^* \xrightarrow{b} b(abb+a)^* \xrightarrow{b} (abb+a)^* \xrightarrow{a} (abb+a)^*$$

Using Lemma 2.4 (four times!), we can conclude that if $x \in \mathcal{L}((abb+a)^*)$, then $abbax \in \mathcal{L}((abb+a)^*)$ also. In addition, since $(abb+a)^*\surd$, we know from Lemma 2.3 that $\varepsilon \in \mathcal{L}((abb+a)^*)$. Since $abba\varepsilon = abba$, then, it follows that $abba \in \mathcal{L}((abb+a)^*)$.

More generally, if there is a sequence of transitions $r_0 \xrightarrow{a_1} r_1 \cdots \xrightarrow{a_n} r_n$ and $r_n\surd$, then we can assert that $a_1 ... a_n \in \mathcal{L}(r_0)$, and vice versa. This observation suggests the following possible strategy for building a NFA from a regular expression r.

1. Let the states be all possible regular expressions that can be reached by some sequence of $\longrightarrow$-transitions from r.
2. Take r to be the start state.
3. Let the transitions be given by $\longrightarrow$.
4. Let the accepting states be those regular expressions r' for which $r'\surd$ holds.

Of course, this would only work if the set of "all possible regular expressions" mentioned in part 1 is finite, since a NFA is required to have a finite number of states.

To make this construction precise, and to examine the issue of "finiteness" of state sets, we need to define mathematically the set of "all possible regular expressions that can be reached by some sequence". We can do this as follows.

Definition 3.5. Let $r \in \mathcal{R}(\Sigma)$ be a regular expression. Then the set $RS(r) \subseteq \mathcal{R}(\Sigma)$ is defined recursively as follows.

- $r \in RS(r)$.
- If $r_1 \in RS(r)$ and $r_1 \xrightarrow{a} r_2$ for some $a \in \Sigma$, then $r_2 \in RS(r)$.

The RS stands for "reachability set." As an example, note that

$$RS((abb+a)^*) = \{(abb+a)^*, bb(abb+a)^*, b(abb+a)^*\}.$$

The following result indicates that the number of reachable regular expressions is always finite.

Lemma 3.6. *Let r be a regular expression. Then $RS(r)$ is finite.*

Proof. Follows from observations such as

- $RS(r_1+r_2) = RS(r_1) \cup RS(r_2)$ and
- $RS(r^*) = \{\, r'r^* \mid r' \in RS(r) \,\}$.

□

We can now define our NFA construction as follows.

Definition 3.7. Let $r \in \mathcal{R}(\Sigma)$ be a regular expression. Then $NFA(r) = \langle Q, \Sigma, q_0, \delta, A \rangle$ is the NFA defined as follows.

- $Q = RS(r)$.
- $q_0 = r$.
- $\delta(r_1, a) = \{\, r_2 \in Q \mid r_1 \xrightarrow{a} r_2 \,\}$.
- $A = \{\, r' \in Q \mid r' \surd \,\}$.

The next theorem establishes that r and $NFA(r)$ always have the same languages.

Theorem 3.8. *Let r be a regular expression. The $\mathcal{L}(r) = \mathcal{L}(NFA(r))$.*

Proof. Relies on the fact that Lemmas 2.3 and 2.4 guarantee that $x = a_1 ... a_n \in \mathcal{L}(r)$ if and only if there is a regular expression r' such that $r \xrightarrow{a_1} \cdots \xrightarrow{a_n} r'$ and $r' \surd$. □

4. How to Compute $NFA(r)$

It may not be apparent from the discussion up to now, but in fact the construction for $NFA(r)$ given above can be automated; that is, one can come up with a routine for building $NFA(r)$, given r. Before describing how this may be done, we first make precise the notion of "outgoing transitions" from a regular expression and explain how they may be calculated.

Definition 4.9. Let $r \in \mathcal{R}(\Sigma)$ be a regular expression. Then the set of *outgoing transitions* from r is defined as the set $\{\, \langle a, r' \rangle \mid r \xrightarrow{a} r' \,\}$.

Intuitively, the outgoing transitions from r consists of pairs $\langle a, r' \rangle$ that, when combined with r, constitute a valid "transition" $r \xrightarrow{a} r'$. Figure 1 contains a recursive procedure computing for outgoing transitions. The routine, out, uses the structure of r and the rules that define $\longrightarrow$ to guide its computation. For regular expressions of the form $\emptyset, \varepsilon$ and $a \in \Sigma$, the definition of $\longrightarrow$ immediately gives all the transitions. For regular expressions built using $+, \cdot$ and *, one must first recursively compute the outgoing transitions of the subexpressions of r and then combine the results appropriately, based on the rules given in the definition of $\longrightarrow$.

The next lemma states that out correctly computes of outgoing transitions.

$$\mathsf{out}(r) = \begin{cases} \{\,\} & \text{if } r = \emptyset \text{ or } r = \varepsilon \\ \{\langle a, \varepsilon\rangle\} & \text{if } r = a \in \Sigma \\ \mathsf{out}(r_1) \cup \mathsf{out}(r_2) & \text{if } r = r_1 + r_2 \\ \{\,\langle a, r_1' r_2\rangle \mid \langle a, r_1'\rangle \in \mathsf{out}(r_1)\,\} & \\ \quad \cup \{\,\langle a, r_2'\rangle \mid \langle a, r_2'\rangle \in \mathsf{out}(r_2) \wedge r_1 \surd\} & \text{if } r = r_1 r_2 \\ \{\,\langle a, r_1' r_1^*\rangle \mid \langle a, r_1'\rangle \in \mathsf{out}(r_1)\,\} & \text{if } r = r_1^* \end{cases}$$

Figure 1: Calculating the outgoing transitions of regular expressions.

Lemma 4.10. *Let $r \in \mathcal{R}(\Sigma)$ be a regular expression, and let out be as defined in Figure 1. Then $out(r) = \{\,\langle a, r'\rangle \mid r \xrightarrow{a} r'\,\}$.*

Proof. The proof breaks into two pieces. The first requires us to show that every $\langle a, r'\rangle \in \mathsf{out}(r)$ satisfies: $r \xrightarrow{a} r'$. In the second, we establish that whenever $r \xrightarrow{a} r'$, then $\langle a, r'\rangle \in \mathsf{out}(r)$. Both arguments can be carried out using induction, with the first being done on the structure of the definition of out and the second using the definition of $\longrightarrow$.
□

We now sketch a routine for computing $NFA(r)$; it relies on maintaining three sets of regular expressions.

- Q, a set that will eventually contain the states of $NFA(r)$.
- A, a set that will eventually contain the accepting states of $NFA(r)$.
- *toProc*, a subset of Q containing states that have not yet had their transitions computed and thus require some "processing".

The algorithm works as follows. Initially, Q and *toProc* contain only r. While there remains at least one regular expression to process, we remove one such an expression from *toProc* and perform the following. First, we check to see if $\surd$ holds for the expression; if so then we add the expression to the set of accepting states. Then we compute all the "outgoing transitions" from the given expression; the target expressions of these transitions that are not already in Q are added both to Q and to *toProc*, as they have not yet been encountered and thus need their transitions computed. The algorithm terminates when *toProc* is empty. Pseudocode for this procedure may be found in Figure 2, while Figure 3 gives the NFA resulting from applying the procedure to $(abb + a)^*$.

```
procedure NFA ($r$) =
  begin
    $Q := \{r\}$;
    $A := \emptyset$;
    set $\delta(r,a) := \emptyset$ for all $a \in \Sigma$;
    $toProc := \{r\}$;
    while $toProc \neq \emptyset$ do
      begin
        choose $r_1 \in toProc$;
        delete $r_1$ from $toProc$;
        if $r_1\surd$ then add $r_1$ to $A$;
          compute $T = \mathsf{out}(r_1)$;
          for each $\langle a, r_1' \rangle \in T$ do
            begin
              add $r_1'$ into $\delta(r_1, a)$;
              if $r_1' \notin Q$ then add $r_2$ to $Q$ and $toProc$;
            end
      end;
    return NFA $\langle Q, \Sigma, r, \delta, A \rangle$;
  end
```

Figure 2: Procedure for building NFA from regular expression.

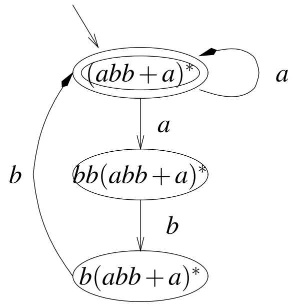

Figure 3: A NFA for $(abb+a)^*$.