Welcome to
Cpts 317

Monday Feb 3, 2020
(aka post Super Bowl Day)

Lemma of the day:
If a language is regular, then it is described by a regular expression
Summary of equivalences of four different "notions" for regular languages.
Theorem

A language is regular iff some regular expression describes it.

Lemma 1

If a language is described by a regular expression, then it is regular.

Lemma 2

If a language is regular, then it is described by a regular expression.
* We saw proof of Lemma 1 in last lecture

* We started proving Lemma 2:
  broke procedure into two
  1) DFA $\rightarrow$ Generalized NFA (GNFA)
  2) GNFA $\rightarrow$ regular expression

* We defined GNFA, and saw the simplified GNFA format.
To convert a DFA into a GNFA in the special form

1. Add a new start state with an ε arrow to the old start state, and a new accept state with ε arrows from the old accept state.
2. If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), replace each with a single arrow whose label is the union of the previous labels.
3. Add arrows labelled ε between states that had no arrows.
To convert a GNFA into a regular expression

* Suppose the GNFA has \( k \) states. Then we know that \( k \geq 2 \)
  (because a GNFA must have a start state and an accept state, and they must be different from each other)

* If \( k > 2 \), we construct an equivalent GNFA with \( k-1 \) states.
* This step can be repeated on the new GNFA until it is reduced to two states.
If \( k = 2 \), the GNFA has a single arrow that goes from the start state to the accept state. The label of this arrow is the equivalent regular expression.

Example: Stage in converting a DFA with 3 states to an equivalent reg. exp.
* The crucial step is constructing an equivalent GNFA with one fewer state when \( k > 2 \).

* We do this by selecting a state, tipping it out of the machine, and repairing the remainder so that the same language is still recognized.

* Any state will do, provided that it is not the start state or accept state.

* We are guaranteed that such a state will exist because \( k > 2 \).

Let us call the removed state \( \ell_0 \).
After removing \( q_i \), we repair the machine by altering the regular expressions that label each of the remaining arrows.

The new labels compensate for the absence of \( q_i \) by adding back the lost computation.

The new label going from a state \( q_i \) to a state \( q_j \) is a regular expression that describes all strings that would take the machine from \( q_i \) to \( q_j \) either directly or via \( q_i \).
Example

In the old machine, if

1. $g_i$ goes to $g_{trip}$ with an arrow labeled $P_1$,
2. $g_{trip}$ goes to itself with an arrow labeled $P_2$,
3. $g_{trip}$ goes to $g_j$ with an arrow labeled $P_3$, and
4. $g_i$ goes to $g_j$ with an arrow labeled $P_4$.

Then, in the new machine, the arrow from $g_i$ to $g_j$ gets the label $(R_1)(R_2)^x(R_3)(R_4)$.
We make this change for each arrow going from any state $q_i$ to any state $q_j$, including the case where $q_i = q_j$.

The new machine recognizes the original language.

We will now see how this idea is carried out formally — i.e. prog.
Proof:
We start with a formal definition for GNFA.

A GNFA is a 5-tuple

\((Q, \Sigma, S, q_{\text{start}}, q_{\text{accept}})\) where

1. \(Q\) is the finite set of states,
2. \(\Sigma\) is the input alphabet,
3. \(S: (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \rightarrow R\) is the transition function,
4. \(q_{\text{start}}\) is the start state, and
5. \(q_{\text{accept}}\) is the accept state

\(R\) is the collection of all regular expressions over alphabet \(\Sigma\).
A GNFA accepts a string \( w \in \Sigma^* \)

if \( w = w_1 w_2 \ldots w_k \), where

each \( w_i \) is in \( \Sigma^* \) and a sequence of

states, \( q_0, q_1, \ldots, q_k \) exists such that

1. \( q_0 = q_{\text{start}} \) is the start state

2. \( q_k = q_{\text{accept}} \) is an accept state, \( q \)

3. for each \( i \), we have

\( w_i \in L(R_i) \), where \( R_i = S(q_{in}, q_i) \)

(low: \( R_i \) is the expression on the
arrow from \( q_{in} \) to \( q_i \) )
Next, we prove that CONVERT returns a correct value.

**Claim**

for any GNFA $G$, $\text{CONVERT}(G)$ is equivalent to $G$.

**Proof**: By induction

**Basis**: $k = 2$.

If $G$ has only two states, it can have only a single arrow, going from the start state to the accept state. The regular expression label on this arrow describes all strings that allow $G$ to get to the accept state. Hence this expression is equivalent to $G$. 
Convert (G):

1. Let k be the number of states of G.
2. If k = 2, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with R. Return R.

3. If k > 2, select any state $q_{np}$, $q_{np}$ different from $q_{start}$ and $q_{accept}$ and let $G'$ be the GNFA $(Q', \Sigma, \delta', q_{start}, q_{accept})$ where $Q' = Q - \{ q_{np} \}$, and for any $q_i \in Q' - \{ q_{accept} \}$ and any $q_i \in Q' - \{ q_{start} \}$, let $\delta'(q_i, q_i) = (R_1)(R_2)(R_3)(R_4)$, for $R_1 = \delta(q_{start}, q_{accept})$, $R_2 = \delta(q_{np}, q_{np})$, $R_3 = \delta(q_{np}, q_i)$, $R_4 = \delta(q_i, q_i)$.

4. Compute Convert ($G'$) and return this value.
Returning to the proof of Lemma 2.

Let $M$ be the DFA for language $A$. We convert $M$ to a GNFA $G$ by adding a new start state and a new accept state and additional transition arrows as necessary.

We use the procedure \texttt{CONVERT (G)} which takes a GNFA and returns an equivalent regular expression.

(Heading: \texttt{CONVERT uses recursion})
Induction Step:

Assume that the claim is true for \( k-1 \) states & use this to prove the claim is true for \( k \) states.

First, we show that \( G \) and \( G' \) recognize the same language.

Suppose that \( G \) accepts an input \( w \).
Then in an accepting branch of the Catenation,
\( G \) enters a sequence of states
\[ G_{\text{start}}, q_1, q_2, \ldots, q_{\text{accept}} \]
If none of them is \( q_{\text{top}} \), clearly \( G' \) also accepts \( w \). (Because each of the new regular expression labeling the arrows of \( G' \) contains the old regular expression as part of a union.)
If \( Q_{\text{rip}} \) does appear, removing each run of consecutive \( Q_{\text{rip}} \) states forms an accepting computation for \( Q' \).

The states \( q_i \) and \( q_j \) bracketing a run have a new regular expression on the arrow between them that describes all strings taking \( q_i \) to \( q_j \) via \( Q_{\text{rip}} \) on \( Q \).

so \( Q' \) accepts \( w \).
Conversely, suppose $G'$ accepts an input $w$. As each arrow between any two states $q_i$ and $q_j$ in $G'$ describes the collection of strings taking $q_i$ to $q_j$ in $G$, either directly or via $q_{k-1}$, $G$ must also accept $w$. Thus $G$ and $G'$ are equivalent.

The induction hypothesis states that when the algorithm calls itself recursively on input $G'$, the result is a regular expression that is equivalent to $G'$ because $G'$ has $k-1$ states. Hence the regular expression also is equivalent to $G$, and the algorithm is proved correct.
Example 1 (2-state DFA to equiv. reg. exp.)

(a) 2-state DFA

(b) Equiv. reg. exp. for (a)

(c) 1-state DFA

(d) Equiv. reg. exp. for (c)
Example 2. (3-state DFA to equiv. ref. exp.)

(a) b
(a (aa ub)* a6 Ub) ((ba Va) (aa ub)* a6 Ub)*

(b) b

(b) bb

(c) bb

(d) b

(e) b

(f) b

(g) b

(h) b

(i) b