Regular Expressions,
Regular Languages

18 February 2020
A plea

Restore my faith in humanity.
Return the classroom stapler.
Assignment 1
   Graded and returned

Assignment 2
   Being graded

Assignment 3
   Due today
   Corrections due Thursday

Exam 1
   Looming
Try these out! (But not on the exam.)

“Present-day computers are built of transistors and wires, but they could just as well be built, according to the same principles, from valves and water pipes, or from sticks and string. The principles are the essence of what makes a computer compute. One of the most remarkable things about computers is that their essential nature transcends technology.”

W. Daniel Hillis, *The Pattern on the Stone*
Kleene’s theorem
KLEEVEN'S THEOREM: Regular expressions and finite automata have equivalent descriptive power.

That is, the languages recognized by DFAs and NFAs and those described by REs are exactly the regular languages.
Regular Expression

$\epsilon
\rightarrow
NFA-\epsilon
\rightarrow
DFA

\begin{align*}
((a \cup b)^* \cup c)^* & \cup a b^* (c \cup b)^*
\end{align*}$
Proof

We will prove this set of equivalences by

- Showing how to construct a DFA from an NFA-\(\varepsilon\). (Already done!)
- Showing how to construct an NFA-\(\varepsilon\) from a regular expression
- Showing how to construct a regular expression from a finite automaton
Constructing an NFA-ε from a regular expression
Cover the six cases in the formal (recursive) definition of REs.

Base cases:

1. $R = a$

   ![Diagram for $R = a$]

2. $R = \varepsilon$

   ![Diagram for $R = \varepsilon$]

3. $R = \emptyset$

   ![Diagram for $R = \emptyset$]
4 \( R = (R_1 \cup R_2) \)

*The class of regular languages is closed under the union operation.*

For languages represented by \( R_1 \) and \( R_2 \), take their NFAs \( N_1 \) and \( N_2 \) and combine them into one new NFA \( N \).

\( N \) must accept input if either \( N_1 \) or \( N_2 \) accepts input.
5 \[ R = (R_1 \cdot R_2) \]

The class of regular languages is closed under the concatenation operation.

For languages represented by \( R_1 \) and \( R_2 \), take their NFAs \( N_1 \) and \( N_2 \) and combine them sequentially into one new NFA \( N \).

\[ N \]

\[ N_1 \quad N_2 \]

\[ \varepsilon \]
6 \( R = (R_1)^* \)

The class of regular languages is closed under the star operation.

For a language represented by \( R_1 \), modify \( N_1 \) to accept \( (R_1)^* \).
Example

$(ab \cup a)^*$
Example

\((ab \cup a)^*\)

\[ a \quad \xrightarrow{a} \quad \text{circle} \]

\[ b \quad \xrightarrow{b} \quad \text{circle} \]

\[ ab \quad \xrightarrow{a} \quad \varepsilon \quad \xrightarrow{b} \quad \text{circle} \]

\[ ab \cup a \quad \varepsilon \quad \xrightarrow{a} \quad \text{circle} \]

\[ (ab \cup a)^* \quad \varepsilon \quad \xrightarrow{a} \quad \varepsilon \quad \xrightarrow{b} \quad \text{circle} \]

Solution from Sipser
The construction we’re using in this proof is, approximately, **Thompson’s Algorithm**.

The “Thompson” is computing pioneer Ken Thompson, a co-inventor of UNIX.
Thompson’s Algorithm is used by actual regular expression matchers to convert regular expressions into finite automata that can be run.

Since the 1970s, `grep` has converted each regular expression into a finite automaton that it runs to do the search.

```
$ grep "[DN]FAs are (fun|cool)" notes.txt
```
This construction is *awesome*.

But we can also prove that the language of a regular expression is a regular language without doing a proof by construction.

How?
This construction is *awesome*.

But we can also prove that the language of a regular expression is a regular language without doing a proof by construction.

How?

*Proof by induction! Left as an exercise.*
We will prove this set of equivalences by

- Showing how to construct a DFA from an NFA-ε. (Already done!)
- Showing how to construct an NFA-ε from a regular expression
- Showing how to construct a regular expression from a finite automaton
Constructing a regular expression from a NFA
Generalizing NFAs
Generalizing NFAs
Generalizing NFAs

These are all regular expressions!
Generalizing NFAs

Note: NFAs aren’t allowed to have transitions like these. This is just a thought experiment.
Generalizing NFAs

\[
\begin{align*}
q_0 & \xrightarrow{ab \cup b} q_1 \\
q_2 & \xrightarrow{a} q_3 \\
q_3 & \xrightarrow{ab^*} q_2 \\
\end{align*}
\]
Generalizing NFAs

\[ q_0 \xrightarrow{\text{start}} q_1 \xrightarrow{a} q_2 \xrightarrow{a^*b?a^*} q_3 \xrightarrow{ab^*} q_0 \]

\[ a \ a \ a \ a \ b \ a \ a \ b \ b \ b \ b \]
Generalizing NFAs

\[
\begin{array}{c}
q_0 \quad \quad \quad \quad q_1 \\
\downarrow \quad \quad \quad \quad \quad \downarrow \\
a \quad \quad \quad \quad a^*b?a^* \\
q_2 \quad \quad \quad \quad q_3 \\
\end{array}
\]

\[ (ab \cup b) \]

\[
\begin{array}{c}
\text{start} \\
q_0 \\
\end{array}
\]
Generalizing NFAs
Generalizing NFAs

Diagram:

- States: $q_0$, $q_1$, $q_2$, $q_3$
- Transitions:
  - $q_0$ to $q_1$: $ab \cup b$
  - $q_0$ to $q_2$: $a$
  - $q_1$ to $q_3$: $ab^*$
  - $q_2$ to $q_3$: $a^*b?a^*$

Input: $a a a b a a a b b b b$
Generalizing NFAs

```
start
q0  -- ab u b --> q1
     /  \\
    /    \\
   a    a*b
    \
  q2  -- a*b?a* --> q3
      /    \\
     /      \\
    ab*     \\
```

```
a a a b a a b b b
```
Generalizing NFAs

\[
\begin{align*}
q_0 & \xrightarrow{ab \cup b} q_1 \\
q_2 & \xrightarrow{a^*b?a^*} q_3 \\
& \xrightarrow{ab^*} q_3
\end{align*}
\]
Generalizing NFAs

\[ q_0 \xrightarrow{ab \cup b} q_1 \xrightarrow{ab^*} q_3 \]

\[ q_2 \xrightarrow{a^*b?a^*} q_3 \]

\( a \ a \ a \ b \ a \ a \ b \ b \ b \ b \)
Generalizing NFAs

\[
\begin{align*}
q_0 \xrightarrow{a} q_2 \\
q_2 \xrightarrow{a^*b?a^*} q_3 \\
q_1 \xrightarrow{ab\cup b} q_0 \\
start \xrightarrow{ab^*} q_3
\end{align*}
\]
Generalizing NFAs

\[
\begin{align*}
\text{start} & \quad \rightarrow \quad q_0 \\
q_0 & \quad \rightarrow \quad q_1 \quad \text{ab} \cup b \\
q_0 & \quad \rightarrow \quad q_2 \quad a \\
q_2 & \quad \rightarrow \quad q_3 \quad a^*b?a^* \\
q_3 & \quad \rightarrow \quad q_1 \quad ab^* \\
\end{align*}
\]
Key idea 1: Imagine that we can label transitions in an NFA with arbitrary regular expressions.
Generalizing NFAs

Is there a simple regular expression for the language of this generalized NFA?
Generalizing NFAs

Is there a simple regular expression for the language of this generalized NFA?
Generalizing NFAs

Is there a simple regular expression for the language of this generalized NFA?
Generalizing NFAs

Is there a simple regular expression for the language of this generalized NFA?
Key idea 2: If we can convert an NFA into a generalized NFA that looks like this,

![Diagram](image)

then we can easily read off a regular expression for the original NFA.
From GNFAs to regular expressions

\[ R_{00}, R_{01}, R_{11}, \text{ and } R_{10} \text{ are arbitrary regular expressions.} \]
From GNFAs to regular expressions

Can we get a clean regular expression from this NFA?
From GNFA to regular expressions

Key idea 3: Transform a GNFA so it looks like this:
From GNFAs to regular expressions

First add new start and accept states
From GNFAs to regular expressions

First add new start and accept states
From GNFAs to regular expressions

*First add new start and accept states*
From GNFA to regular expressions

Could we eliminate this state from the GNFA?
From GNFA to regular expressions
We can use concatenation and Kleene closure to skip this state.
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions
From GNFAs to regular expressions

We can use union to combine these transitions.
Could we eliminate this state from the GNFA?
Could we eliminate this state from the GNFA?

\[ R_{11} \cup R_{10} R_{00}^* R_{01} \]
From GNFA to regular expressions

\[ R_{00}^* R_{01} \]

\[ R_{11} \cup R_{10} R_{00}^* R_{01} \]
From GNFA\textsubscript{s} to regular expressions

What should we put on this transition?

\[
\begin{align*}
q_s & \quad R_{00}^* R_{01} & q_1 \quad \varepsilon & \quad q_f \\
R_{11} \cup R_{10} R_{00}^* R_{01} & \end{align*}
\]
From GNFAs to regular expressions

\[
R_{00}^* R_{01} \left( R_{11} \cup R_{10} R_{00}^* R_{01} \right)^* \epsilon
\]
From GNFAs to regular expressions

\[
R_{00}^* R_{01} (R_{11} \cup R_{10} R_{00}^* R_{01})^* \epsilon
\]
From GNFAs to regular expressions

\[ R_{00}^* R_{01} (R_{11} \cup R_{10} R_{00}^* R_{01})^* \varepsilon \]
From GNFAs to regular expressions

\[ R_{00}^* R_{01} (R_{11} \cup R_{10} R_{00}^* R_{01})^* \]
From GNFAs to regular expressions

\[
\begin{align*}
&\text{start} \\
&\text{from } q_s \\
&\quad \rightarrow \quad R_{00}^* R_{01} (R_{11} \cup R_{10} R_{00}^* R_{01})^* \\
&\quad \rightarrow \quad q_f
\end{align*}
\]
From GNFAs to regular expressions

After:

Before:
The state-elimination algorithm

1. Start with an NFA $M$ for the language $L$, which we’ll use as a generalized NFA (GNFA).

2. Add a new start state $q_s$ and accept state $q_f$ to $M$.
   - Add an $\varepsilon$-transition from $q_s$ to the old start state of $M$.
   - Add $\varepsilon$-transitions from each accepting state of $M$ to $q_f$, then mark them as not accepting.

3. Repeatedly remove states other than $q_s$ and $q_f$ from the NFA by “shortcutting” them until only $q_s$ and $q_f$ remain.

4. The transition from $q_s$ to $q_f$ is now a regular expression equivalent to the original NFA.
Eliminating a state

Before

After

Before:
- $q_i$ to $q_j$ via $R_{rip}$
- $q_i$ to $q_j$ via $R_{out}$
- $q_i$ to $q_i$ via $R_{stay}$

After:
- $(R_{in} (R_{stay})^* R_{out}) \cup R_{other}$ from $q_i$ to $q_j$
Eliminating a state

To eliminate a state $q_{rip}$ from the automaton, do the following for each pair of states $q_i$ and $q_j$, where there’s a transition from $q_i$ into $q_{rip}$ and a transition from $q_{rip}$ into $q_j$:

Let $R_{in}$ be the regex. on the transition from $q_i$ to $q_{rip}$.
Let $R_{out}$ be the regex. on the transition from $q_{rip}$ to $q_j$.

If there is a regular expression $R_{stay}$ on a transition from $q_{rip}$ to itself,

Add a new transition from $q_i$ to $q_j$ labeled $((R_{in})(R_{stay})^*(R_{out}))$.

Otherwise,

Add a new transition from $q_i$ to $q_j$ labeled $((R_{in})(R_{out}))$.

If a pair of states has multiple transitions between them labeled $R_1$, $R_2$, …, $R_k$, replace them with a single transition labeled $R_1 \cup R_2 \cup \cdots \cup R_k$. 

See diagram on previous slide
Example conversion
Example conversion

Add new start and end state
Example conversion

Add new start and end state

Remove state 2
Example conversion

Add new start and end state

Remove state 2
Example conversion

Add new start and end state

Remove state 2
Example conversion

Add new start and end state

Remove state 2

b(aub)*
Example conversion

Add new start and end state

Remove state 2

Remove state 1
Example conversion

Add new start and end state

Remove state 1

Remove state 2

Remove state 1

Example conversion

Add new start and end state

Remove state 2

Remove state 1
Example conversion

Add new start and end state

Remove state 2

Remove state 1
Exercise
Step 1: Modify to create a unique start and end state:
Step 2: Eliminate state $q_0$. 
Step 3: Eliminate state $q_1$. 

$\begin{align*}
\text{Step 3: Eliminate state } q_1. \\
\text{Start state: } S \\
\text{Final state: } F \\
\text{Transitions:} \\
a^*bb^* \\
a^*bb^*a \\
(\varepsilon a^*b) b^* \\
(b \cup a^*b) b^* a
\end{align*}$
Step 4: Eliminate state $q_2$.

In class we removed the states in the order $q_0$, $q_2$, $q_1$, which gave a smaller regular expression.
We will prove this set of equivalences by

- Showing how to construct a DFA from an NFA-ε. (Already done!)
- Showing how to construct an NFA-ε from a regular expression
- Showing how to construct a regular expression from a finite automaton

Proof
The regular languages are recognized by DFAs, NFAs, and REs.
The following are all equivalent:

$L$ is a regular language.

There is a DFA $D$ such that $L(D) = L$.

There is an NFA $N$ such that $L(N) = L$.

There is a regular expression $R$ such that $L(R) = L$. 
Why this matters

The equivalence of regular expressions and finite automata has practical relevance.

Tools like `grep` and `flex` that use regular expressions capture all the power available via DFAs and NFAs.

This is also hugely significant theoretically:

The regular languages can be assembled “from scratch” using a small number of operations!
Beyond the Valley of the Regular Languages
DFAs correspond to computers with finite memory.
The equivalence of DFAs and NFAs tells us that given finite memory, nondeterminism doesn’t increase computational power.

(Though it might save on memory.)

The equivalence of DFAs and regular expressions tells us that all problems solvable by finite computers can be assembled out of smaller building blocks.
Is every language regular?
To prove a language is regular, we can construct a DFA, NFA, or RE to recognize it.

Or directly use known closure properties.
To prove a language is not regular, we need to show that it’s impossible to construct a DFA, NFA, or regular expression for it.

This kind of argument is challenging – how can we show that we wouldn’t be able to devise a finite automaton for it if we tried harder?
Proof strategy: *Infinite states*
A simple language

Let $\Sigma = \{a, b\}$ and consider this language:

$$L = \{a^ib^i \mid i \in \mathbb{N}_0\}$$

$L$ is the language of all strings of $i$ as followed by $i$ bs:

$$\{\varepsilon, ab, aabb, aaabbb, \ldots\}$$

Is this language regular?
How many states are needed to recognize \( \{a^i b^i\} \)?
This language is not regular!

Intuitive explanation:

Imagine a finite automaton to accept this language. When any DFA for L is run on any two of the strings $\varepsilon$, $a$, $aa$, $aaa$, $aaaa$, etc., the DFA must end in different states.

Suppose $a^n$ and $a^m$ end up in the same state, where $m \neq n$.

Then $a^n b^n$ and $a^m b^n$ will end up in the same state.

The DFA will either accept a string not in the language or reject a string in the language, which it shouldn’t be able to do.

*We can’t place all these strings into different states; there are only finitely many states!*
(More) formal proof

Consider the language \{a^i b^i \mid i \geq 0\} and a DFA to recognize it.

For any \(i\), let \(q_i\) be the state entered after processing \(a^i\).
That is, \(\hat{\delta}(q_0, a^i) = q_i\)

Consider any \(i\) and \(j\) such that \(i \neq j\).

\[ \hat{\delta}(q_0, a^i b^i) \neq \hat{\delta}(q_0, a^j b^i), \]

since the former is accepting and the latter is rejecting.

\[ \hat{\delta}(q_0, a^i b^i) = \hat{\delta}(\hat{\delta}(q_0, a^i), b^i) = \hat{\delta}(q_i, b^i), \]

by definition of \(\hat{\delta}\) and definition of \(q_i\), respectively.

\[ \hat{\delta}(q_0, a^j b^i) = \hat{\delta}(\hat{\delta}(q_0, a^j), b^i) = \hat{\delta}(q_j, b^i), \]

by the same reasoning.
Since inputs $a^i$ and $a^j$ lead to different states on the same input, the states must be different: $q_i \neq q_j$.

Since $i$ and $j$ were arbitrary, and since there are an infinite number of ways to pick them, \textit{there must be an infinite number of states.}

Thus there is no \textit{finite-state} machine to recognize this language, and the language cannot be regular.
The intuition for this proof is helpful to think about. However, actually writing one of these proofs becomes difficult for more complicated languages.

We can take the idea of the number of states required for a DFA to recognize a language and develop a powerful proof framework: the Pumping Lemma.
Acknowledgments

This lecture incorporates material from:

Nancy Ide
Keith Schwarz
Michael Sipser
Jeffrey Ullman