Regular Expressions

19 September 2023
Where are we?
A language $L$ is a *regular language* if there is a DFA $D$ such that $L(D) = L$. 
THEOREM The following are equivalent:

- $L$ is a regular language.
- There is a DFA for $L$.
- There is an NFA for $L$. 

If $w \in \Sigma^*$ and $x \in \Sigma^*$, then $wx$ is the *concatenation* of $w$ and $x$.

If $L_1$ and $L_2$ are languages over $\Sigma$, the *concatenation* of $L_1$ and $L_2$ is the language $L_1L_2$, defined as

$$L_1L_2 = \{wx \mid w \in L_1 \text{ and } x \in L_2\}.$$

For example, if $L_1 = \{a, \text{ba}, \text{bb}\}$ and $L_2 = \{\text{aa}, \text{bb}\}$, then

$$L_1L_2 = \{\text{aaa, abb, baaa, babb, bbbaa, bbbb}\}.$$
Lots of concatenation

Consider the language \( L = \{aa, b\} \)

\( LL \) is the set of strings formed by concatenating pairs of strings in \( L \):

\[ \{aaaa, aab, baa, bb\} \]

\( LLL \) is the set of strings formed by concatenating triples of strings in \( L \):

\[ \{aaaaaa, aaaaab, aabaa, aabb, baaaa, baab, bbbaa, bbb\} \]

\( LLLLL \) is the set of strings formed by concatenating quadruples of strings in \( L \).
We can define what it means to “exponentiate” a language as follows:

\[ L^0 = \{ \varepsilon \} \]

**Base case:** Any string formed by concatenating zero strings together is just the empty string.

\[ L^{n+1} = LL^n \]

**Recursive case:** Concatenating \( n + 1 \) strings together works by concatenating \( n \) strings, then concatenating one more.
We can define what it means to “exponentiate” a language as follows:

\[ L^0 = \{ \epsilon \} \]

*Base case:* Any string formed by concatenating zero strings together is just the empty string.

\[ L^{n+1} = L \cdot L^n \]

*Recursive case:* Concatenating \( n + 1 \) strings together works by concatenating \( n \) strings, then concatenating one more.

*The only string you can form by gluing no strings together is the empty string!*
An important operation on languages is the **Kleene closure**, which is defined as

\[ L^* = \{ w \in \Sigma^* \mid \exists n \in \mathbb{N}_0 . w \in L^n \}. \]

That is, a word is in \( L^* \) iff it’s in

- the language \( L^0 \) or
- the language \( L^1 \) or
- the language \( L^2 \) or …
$L^*$ consists of all the possible ways of concatenating zero or more strings in $L$.

If $L = \{a, bb\}$, then $L^* = \{
\varepsilon,

a, bb,

aa, abb, bba, bbbb,

aaa, aabb, abba, abbbb, bbba, bbabb, bbbba, bbbbbbb,

...\}
Last class, we saw that the class of regular languages (\textsc{REG}) is \textit{closed} under the following operations:

- Complement
- Union
- Intersection
- Concatenation
- Kleene star
That is, if $L_1$ and $L_2$ are regular languages over an alphabet $\Sigma$, then so are the following languages:

- **Complement** $\overline{L_1}$
- **Union** $L_1 \cup L_2$
- **Intersection** $L_1 \cap L_2$
- **Concatenation** $L_1 L_2$
- **Kleene star** $L_1^*$
Another view of regular languages
We’ve seen we can show a language is regular by constructing a DFA for it or constructing an NFA for it.

We can also show a language is regular by using closure properties to build it out of other regular languages.
This is a bottom-up approach to the regular languages:

Start with a small set of simple languages we know to be regular.

Use closure properties to combine these to form more elaborate languages.
Regular expressions provide a concise notation for describing this way of building regular languages out of simpler pieces.

They’re use just about everywhere:

- They’re built into JavaScript and used for data validation.
- They’re used in the Unix grep tool to search for strings and flex to build compilers.
- They’re used to clean and scrape data for large-scale analysis projects.
For the moment, put aside anything you know about writing regular expressions when programming.
### Atomic regular expressions

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \varepsilon )</td>
<td>( { \varepsilon } )</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>( { \alpha } )</td>
</tr>
</tbody>
</table>

for any \( \alpha \in \Sigma \)
### Compound regular expressions

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ (R_1 \cup R_2) $</td>
<td>$ L(R_1) \cup L(R_2) $</td>
</tr>
<tr>
<td>$ (R_1 \circ R_2) $</td>
<td>$ L(R_1) \circ L(R_2) $</td>
</tr>
<tr>
<td>$ (R_1^*) $</td>
<td>$ L(R_1)^* $</td>
</tr>
</tbody>
</table>

for any regular expressions $ R_1 $ and $ R_2 $
**DEFINITION**  \( R \) is a *regular expression* if \( R \) is

1. \( \alpha \) for some \( \alpha \in \Sigma \)
2. \( \varepsilon \)
3. \( \emptyset \)

**Basis**

4. \( (R_1 \cup R_2) \), where \( R_1 \) and \( R_2 \) are regular expressions
5. \( (R_1 \circ R_2) \), where \( R_1 \) and \( R_2 \) are regular expressions
6. \( (R_1^*) \), where \( R_1 \) is a regular expression

**Recursive cases**
As with strings and languages, we can omit the $\circ$ and just indicate concatenation by writing two regular expressions next to each other.

$$ab = a \circ b$$
Order of operations

We can omit parentheses to make regular expressions more compact, but this makes them ambiguous unless we define precedence:

1. Parentheses \((R)\)
2. Kleene star \(R^*\)
3. Concatenation \(R_1 \circ R_2\) or \(R_1 R_2\)
4. Union \(R_1 \cup R_2\)
ab*cu\d
ab*cud

a(b*)cud
ab*cud

a(b*)cud

(a°(b*))cud
\( ab^*c \cup d \)

\( a(b^*)c \cup d \)

\( (a \circ (b^*))c \cup d \)

\( ((a \circ (b^*)) \circ c) \cup d \)
ab\ast c\cup d

a(b\ast)c\cup d

(a\circ(b\ast))c\cup d

((a\circ(b\ast))\circ c)d
This is the fully parenthesized version, following our formal, recursive definition of regular expressions.
Examples

$L(oh) = \{oh\}$

$L(ohuawww^*) = \{oh, aww, awww, awwww, awww, \ldots\}$

$L((oua)(huwww^*)) =$
Examples

\[ L(\text{oh}) = \{\text{oh}\} \]

\[ L(\text{ohuawww*}) = \{\text{oh}, \text{aww}, \text{awww}, \text{awwww}, \text{awww}, \ldots\} \]

\[ L((\text{oua})(\text{huwww*})) = \{ \]
  \[ \text{oh}, \]
  \[ \text{oww, owww, owww, \ldots,} \]
  \[ \text{ah}, \]
  \[ \text{aww, awww, awww, \ldots} \]
\[ \} \]
Designing regular expressions
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\}$. 
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid w$ contains $aa$ as a substring$\}$.

$$(a \cup b)^* aa (a \cup b)^*$$
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\}$.

$$(a \cup b)^* aa (a \cup b)^*$$
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* | w$ contains $aa$ as a substring$\}$.

$(a \cup b)^* aa (a \cup b)^*$

$bbabbbbaabab$

$aaaa$

$bbbbbbabbbbaabbbbb$
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\}.$

$$(a \cup b)^*aa(a \cup b)^*$$

$bbabbbbaabab$

$aaaa$

$bbbbbbabbbbaabbbbb$
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid w \text{ contains } aa \text{ as a substring}\}$.

A convenient shorthand!

$\Sigma^*aa\Sigma^*$

bbabbbbaabab

aaaa

bbbbbabbbbbaabbbbb
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$. 
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$. 

We write $|w|$ to denote the length of the string $w$. 
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$. 
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$. 
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$. 
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$. 

aaaa
baba
bbbb
baaa
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$. 
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

$\Sigma^4$

aaaaa
baba
bbbb
baaa
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

Another convenient shorthand!
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid |w| = 4\}$.

$\Sigma^4$

Another convenient shorthand!

aaaa
baba
bbbb
baaa
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* | w \text{ contains at most one } a\}$. 
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

Here are some candidate regular expressions for $L$. Which are correct?

- $\Sigma^*a\Sigma^*$
- $b^*ab^*ub^*$
- $b^*(a\cup\varepsilon)b^*$
- $b^*a^*b\varepsilon b^*$
- $b^*(a^*\cup\varepsilon)b^*$
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

$b^* (a \cup \varepsilon) b^*$
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

\[ b^* (a \cup \varepsilon) b^* \]
Designing regular expressions

Let \( \Sigma = \{a, b\} \).

Let \( L = \{ w \in \Sigma^* \mid w \text{ contains at most one } a \} \).

\[
b^* (a \cup \varepsilon) b^*
\]

- bbbbabbb
- bbbbbbb
- abbb
- a
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

$$b^* (a \cup \varepsilon) b^*$$

$bbbbbabbb$

$bbbbbb$

$abbb$

$a$
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid w \text{ contains at most one } a\}$.

$$b^*a?b^*$$

- bbbbbabbb
- bbbbbbb
- abbb
- a
Designing regular expressions

Let $\Sigma = \{a, b\}$.

Let $L = \{w \in \Sigma^* \mid w$ contains at most one $a\}$.

$\textbf{b}^*\textbf{a}\textbf{b}^*$

Another convenient shorthand!
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^*(.aa^*)*$$

mvassar@vassar.edu

matthew.vassar@vassarbrewery.com

matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^* (.aa^*)^*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, \ ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^*(.aa^*)*@$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, , @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

\[ \text{aa}^* (\text{aa}^*)^* @ \]

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^*(.aa^*)*@aa^*.aa^*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^*(.aa^*)*@aa^*.aa^*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$aa^*(.aa^*)*@aa^*.aa^*(.aa^*)^*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

```
aa*(.aa*)*@aa*.aa*(.aa*)*
```

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$a^+ (.aa^*)*@aa^*.aa^* (.aa^*)^*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

You guessed it – another shorthand!

$$a^+(aa^*)*@aa^*.aa^*(aa^*)^*$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, \_, @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$a^+(a^+)\ast@a^+.a^+(a^+)\ast$$

mvassar@vassar.edu

matthew.vassar@vassarbrewery.com

matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$a^+ (.a^+) @ a^+ . a^+ (.a^+)^*$$

mvassar@vassar.edu

matthew.vassar@vassarbrewery.com

matt@cs.vassar.edu
A more elaborate design

Let $\Sigma = \{a, ., @\}$, where $a$ represents “any letter”.

Let’s make a regular expression for email addresses.

$$a^+ (.a^+)* @a^+ (.a^+)^+$$

mvassar@vassar.edu
matthew.vassar@vassarbrewery.com
matt@cs.vassar.edu
For comparison

\[ a^+ (.a^+)\ast @a^+ (.a^+) \]
Shorthands summary

$\Sigma$ is a shorthand for “any character in $\Sigma$”

$R^n$ is a shorthand for $RR\ldots R$ ($n$ times)

$R?$ is shorthand for $(R\cup \varepsilon)$ – that is, zero or one copies of $R$.

$R^+$ is a shorthand for $RR^*$ – that is, one or more copies of $R$. 
Optional interlude:
Unix regular expressions
From the beginning (of time), Unix has used regular expressions in many places, including the **grep** command.

**grep** = global (search for a) regular expression and print

Many Unix commands use an extended RE notation, but it still expresses only the regular languages.
A big difference is that Unix regular expressions typically match *anywhere* in a string, whereas our regular expressions must match the *entire* string.
Unix regular expression notation

$R_1 \cup R_2$ is written as $R_1 | R_2$

$\Sigma$ is written as \text{"\}.\)
Unix regular expression notation

\[ [a_1 a_2 \ldots a_n] = a_1 \cup a_2 \cup \cdots \cup a_n. \]

As a shorthand, you can also specify ranges of ASCII characters, e.g.,

- \([a-z]\) = any lowercase letter

- \([a-zA-Z]\) = any letter
Unix regular expression notation

Since characters like brackets, dashes, and dots have special meaning, if you want to match them, you need to quote with backslash (\).
Perl, Python, Emacs, …

Include additional extensions, notably character classes like \b for word boundary characters, \w for word characters, etc.

With each implementation of regular expressions, they become less standard, so what you write for one language or application won’t work in another.
| **grep** | **Julia Evans**  
|----------|------------------|
| grep lets you search files for text  
$ grep bananas foo.txt  
Here are some of my favourite grep command line arguments!  
| **-E** aka egrep  
Use if you want regexps like ".+" to work. Otherwise you need to use ".\+"  
| **-r** Recursive! Search all the files in a directory.  
| **-v** Invert match: find all lines that don't match  
| **-0** Only print the matching part of the line (not the whole line)  
| **-l** Only show the filenames of the files that matched  
| **-a** Search binaries: treat binary data like it's text instead of ignoring it!  
| **-A** Show context for your search.  
$ grep -A 3 foo  
will show 3 lines of context after a match  
| **-F** Don't treat the match string as a regex  
eg $ grep -F ...  
| **grep alternatives**  
ack 
ag 
ripgrep  
(better for searching code!)  
|
The power of regular expressions
Regular Languages
Regular Languages

*Languages you can build a DFA for*
Regular Languages

Languages you can build a DFA for

Languages you can build an NFA for
Regular Languages

Languages you can build a DFA for

Languages you can build an NFA for

Languages you can write a regex for
Regular Languages

Languages you can build a DFA for

Languages you can build an NFA for

Languages you can write a regex for
Regular Languages

Languages you can build a DFA for

Languages you can build an NFA for

Languages you can write a regex for
Languages you can build a DFA for

Languages you can build an NFA for

Regular Languages

Languages you can write a regex for
Regular Languages

Languages you can build a DFA for

Languages you can build an NFA for

Languages you can write a regex for
THEOREM  If \( R \) is a regular expression, then \( L(R) \) is regular.
THEOREM  If $R$ is a regular expression, then $L(R)$ is regular.

PROOF IDEA  Use induction!

The atomic regular expressions all represent regular languages.

The combination steps represent closure properties.

So, anything you can make from them must be regular!
In practice, many regex matchers – including `grep` – use an algorithm called *Thompson’s algorithm* to convert regular expressions into equivalent finite automata.

The “Thompson” is computing pioneer Ken Thompson, a co-inventor of Unix.
Example

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

That ends the first part of the proof of Theorem 1.54, giving the easier direction of the if and only if condition. Before going on to the other direction, let's consider some examples whereby we use this procedure to convert a regular expression to an NFA.

**Example 1.56**

We convert the regular expression \((ab \cup a)^*\) to an NFA in a sequence of stages. We build up from the smallest subexpressions to larger subexpressions until we have an NFA for the original expression, as shown in the following diagram.

Note that this procedure generally doesn't give the NFA with the fewest states. In this example, the procedure gives an NFA with eight states, but the smallest equivalent NFA has only two states. Can you find it?

**Solution from Sipser**
Regular Languages

Languages you can build a DFA for

Languages you can build an NFA for

Languages you can write a regex for
Regular Languages

Languages you can build a DFA for

Languages you can build an NFA for

Languages you can write a regex for
THEOREM If $L$ is a regular language, then there is a regular expression for $L$.

PROOF IDEA Show how to convert an arbitrary NFA into a regular expression.
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
Generalizing NFAs

\[
q_0 \xrightarrow{a} q_2 \xrightarrow{a \ast \ast} q_3 \xrightarrow{ab^*} q_1 \xrightarrow{ab \cup b} q_0
\]

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

\[
\begin{align*}
\text{start} & \quad q_0 & \quad ab \cup b & \quad q_1 \\
\quad a & \quad q_2 & \quad a^*b?a^* & \quad q_3 \\
\quad ab^* & \quad q_3 & \quad a^*b?a^* & \quad q_2
\end{align*}
\]

Input sequence: \texttt{a a a b a a b b b}
Generalizing NFAs

The diagram illustrates a nondeterministic finite automaton (NFA) with states $q_0$, $q_1$, $q_2$, and $q_3$. The transitions are as follows:

- From $q_0$ to $q_1$: transition on $ab$ or $b$
- From $q_0$ to $q_2$: transition on $a$
- From $q_2$ to $q_3$: transition on $a^*b?a^*$
- From $q_3$: transition on $ab^*$

The automaton starts at state $q_0$.
Generalizing NFAs

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

![Diagram of a generalized NFA with states and transitions labeled.]

- Start state: $q_0$
- Transitions:
  - $q_0 \xrightarrow{a} q_2$
  - $q_0 \xrightarrow{ab} q_1$
  - $q_2 \xrightarrow{a*b?a*} q_3$
  - $q_1 \xrightarrow{ab} q_1$

Example input sequence: $abaababa$
Generalizing NFAs

![Diagram of a generalized NFA]

- Start state: \( q_0 \)
- Transitions:
  - \( q_0 \) to \( q_1 \) on \( ab \cup b \)
  - \( q_2 \) to \( q_3 \) on \( a^*b?a^* \)
  - \( q_2 \) to \( q_3 \) on \( ab^* \)

Input sequence: a a a b a a a b b b

Output: a
Generalizing NFAs
Generalizing NFAs

\[
\begin{array}{c}
\text{start} \\
q_0 \quad ab \cup b \quad q_1 \\
\downarrow a \quad a*b?a* \\
q_2 \quad q_3 \\
\end{array}
\]

\[
a \quad a \quad a \quad b \quad a \quad a \quad b \quad b \quad b
\]
Key idea 1: Imagine that we can label transitions in an NFA with arbitrary regular expressions.
Is there a simple regular expression for the language of this generalized NFA?
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

$R_{00}$, $R_{01}$, $R_{11}$, and $R_{10}$ are variables for arbitrary regular expressions.
From GNFAs to regular expressions

Can we get a clean regular expression from this NFA?
From GNFAs 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

\[ R_{00} \]
\[ R_{01} \]
\[ R_{10} \]
\[ R_{11} \]
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

Could we eliminate this state from the GNFA?
From GNFAs to regular expressions
From GNFAs 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.
From GNFAs to regular expressions

Could we eliminate this state from the GNFA?

Could we eliminate this state from the GNFA?
Could we eliminate this state from the GNFA?
From GNFAs to regular expressions

\[ q_s \xrightarrow{R_{00}^* R_{01}} q_1 \xrightarrow{\varepsilon} q_f \]

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

What should we put on this transition?

\[
q_s \xrightarrow{R_{00}^* R_{01}} q_1 \xrightarrow{\varepsilon} q_f
\]

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

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

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

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

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

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

**Before:**

**After:**
The state-elimination algorithm

1. Start with an NFA $N$ 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 $N$.
   
   Add an $\varepsilon$-transition from $q_s$ to the old start state of $N$.
   
   Add $\varepsilon$-transitions from each accept state of $N$ to $q_f$, then mark them as not accept states.

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.
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 $R_1 \cup R_2 \cup \cdots \cup R_k$. 
Our transformations

- Direct conversion
- State elimination
- Subset construction
- Thompson's algorithm
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!
Next time

Is every language regular?

What, if anything, can’t we solve with DFAs, NFAs, and regular expressions?
Regular expression crossword puzzle

1. /\d+/ (spell out the emoji, write an equivalent regex)
2. /\d+/ (spell out the emoji, write an equivalent regex)
3. /\d+/ (spell out the emoji, write an equivalent regex)
4. /\d+/ (spell out the emoji, write an equivalent regex)
5. /\d+/ (spell out the emoji, write an equivalent regex)
6. /\d+/ (spell out the emoji, write an equivalent regex)
7. /\d+/ (spell out the emoji, write an equivalent regex)
8. /\d+/ (spell out the emoji, write an equivalent regex)
9. /\d+/ (spell out the emoji, write an equivalent regex)
Acknowledgments

This lecture incorporates material from:

Nancy Ide, Vassar College
Keith Schwarz, Stanford University
Michael Sipser, *Introduction to the Theory of Computation*
Jeffrey Ullman, Stanford University