A *regular expression* is a shorthand notation for representing a language.

There are many different kinds of regular expressions, but in this course, we only use one type based on union, concatenation, and Kleene closure. (We *don’t* allow the operations of complement, intersection, or backreferencing, which are allowed in other kinds of regular expressions.)

Examples:

- $(0 \cup 1)^*$: set of all finite strings over $\{0, 1\}$.
- $1(0 \cup 1)^*$: set of all finite strings over $\{0, 1\}$ that start with a 1.
- $((a \cup b)(a \cup b))^*$: set of all strings over $\{a, b\}$ of even length.
More examples

- \((a \cup b)((a \cup b)(a \cup b))^*\): set of all strings over \{a, b\} of odd length.

- \((a \cup b \cup \epsilon)(a \cup b \cup \epsilon)(a \cup b \cup \epsilon)\): set of all strings over \{a, b\} of length \(\leq 3\).

  You could also write this as

  \[\epsilon \cup a \cup b \cup (a \cup b)(a \cup b) \cup (a \cup b)(a \cup b)(a \cup b)\].

- \((a \cup b)^*aa(a \cup b)^*\): set of all strings over \{a, b\} containing \(aa\) as a substring.
Some harder examples

1. Strings over \{a, b\} having no substring equal to \textit{bb};

2. Strings over \{a, b\} having an odd number of \textit{b’s};

3. Nonempty strings over \{a, b, c\} in which all \textit{a’s} (if any) precede all \textit{b’s} (if any), which in turn precede all \textit{c’s} (if any);

Try to construct regular expressions for each of these, then compare your answer to the ones in the next frame.
1. Strings over \( \{a, b\} \) having no substring equal to \( bb \):

The idea is to consider a typical such string and “factorize” it. There are at least three ways to factor it.

The first way breaks up the string into blocks of \( a \)’s (at least one), followed by a \( b \): \( aa \cdots ab \, aa \cdots ab \cdots \). We also have to handle the possibility that the string begins with a \( b \) or ends with \( a \)’s. This gives the expression \( (b \cup \epsilon)(aa^*b)^*a^* \).

The second way breaks up the string into blocks of either \( a \) or \( ab \). We also have to handle the possibility that the string begins with \( b \). This gives the expression \( (b \cup \epsilon)(a \cup ab)^* \).

The third way breaks up the string into blocks of either \( a \) or \( ba \). We also have to handle the possibility that the string ends with \( b \). This gives the expression \( (a \cup ba)^*(b \cup \epsilon) \).
2. Strings over \( \{a, b\} \) having an odd number of \( b \)'s:

Again, the idea is to “factorize” the typical string fulfilling the specification. We can identify the position of each \( b \) in the string. In between each \( b \) there can only be \( a \)'s.

To get an odd number of \( b \)'s, we’ll pair the \( b \)'s together, and have one left over at the end. This gives

\[
a^* b(a^* ba^* b)^* a^*.
\]
3. Nonempty strings over \{a, b, c\} in which all a’s (if any) precede all b’s (if any), which in turn precede all c’s (if any).

If the strings could be empty, then it’s easy: \(a^*b^*c^*\).

But disallowing \(\epsilon\) makes it harder.

One possible solution ensures that there’s at least one a, or at least one b, or at least one c:

\[aa^*b^*c^* \cup a^*bb^*c^* \cup a^*b^*cc^*.\]

However, there’s a shorter regular expression that uses only 13 symbols. Can you find it? I don’t know any solution using less than 13 symbols.
Of course, there are many possible regular expressions for every regular language. Some are better than others...

If every string in the language $L(r)$ is obtainable in essentially one way from a regular expression $r$, then $r$ is said to be unambiguous.

Otherwise, it is ambiguous.

For example, $(1 \cup 11)^*$ is ambiguous, because 11 can either be obtained by 11, or by concatenating 1 with 1.

There is no requirement at all that regular expressions that you construct in this course be unambiguous.
What is a regular expression, formally speaking?

It is a *string*. If the regular expression $r$ specifies a subset of $\Sigma^*$, then $r$ is defined over the alphabet

$$\Sigma \cup \{\cup, *, \epsilon, \emptyset, (, )\}.$$ 

With every regular expression $r$ we associate a language $L(r)$, which is the language it specifies.

- $\emptyset$ denotes the language $\emptyset$;
- $\epsilon$ denotes the language $\{\epsilon\}$;
- $a$ for $a \in \Sigma$ denotes the language $\{a\}$;
If $r_1, r_2$ are regular expressions, then $r_1 \cup r_2$ denotes the language $L(r_1) \cup L(r_2)$;

(Some textbooks may use the symbol $+$ instead of $\cup$.)

If $r_1, r_2$ are regular expressions, then $(r_1)(r_2)$ denotes the language $L(r_1)L(r_2)$;

If $r$ is a regular expression, then $(r)^*$ denotes the language $L(r)^*$.

Parentheses may be omitted if they are not needed.

If a regular expression is not fully parenthesized, then we use some precedence rules: $\ast$ is done first, concatenation is done next, and union is done last. So $a \cup bc^*$ is understood as $a \cup (b(c^*))$. 
We now come to the second big theorem about finite automata: the class of languages recognized by DFA’s (or NFA’s, or $\epsilon$-NFA’s) is the same as the class of languages specified by regular expressions.

To prove this, we’ll show

- how to “compile” a regular expression $r$ into an $\epsilon$-NFA recognizing the language $L(r)$, and
- how to produce a regular expression specifying the language given by an $\epsilon$-NFA (or DFA or NFA) $M$. 
The basic idea is that we’re going to parse the regular expression down to its constituent parts (leaves), then construct the $\epsilon$-NFA from the leaves up.

Another way to say this is that we’re going to show by induction on the number of operators in the regular expression $r$, that there is an $\epsilon$-NFA $M$ such that $L(M) = L(r)$.

Life will be simpler if we restrict the $\epsilon$-NFA $M$ that we construct somewhat. Let’s also ensure that, at each stage, the automaton $M$ we construct

- has exactly one final state;
- has no transitions out of its final state;
- has no transitions into its initial state;
Regular expression to $\epsilon$-NFA

The base case is where $r$ has 0 operators. Then there are three cases: either $r = \emptyset$, or $r = \epsilon$, or $r = a$ for some $a \in \Sigma$. The $\epsilon$-NFA corresponding to each of these is below:

For $\emptyset$:

For $\epsilon$:

For $a$:
Now assume the claim is true for \( r \) with \(< n\) operators. We’ll prove it for \( r \) with \( n\) operators. Identify the last operator to be performed in \( r \). Then write \( r \) either as

\[
\begin{align*}
\triangleright & \quad r = r_1 \cup r_2; \\
\triangleright & \quad r = (r_1)(r_2) ; \text{ or} \\
\triangleright & \quad r = (r_1)^*
\end{align*}
\]

By induction we know there are \( \epsilon\)-NFA \( M_1 \) and \( M_2 \) such that \( L(M_1) = L(r_1) \) and \( L(M_2) = L(r_2) \). We depict them as follows:

Now we can “rewire” these automata to get automata for each of the cases.
Regular expression to $\epsilon$-NFA

For union, we wire them up in parallel:
For concatenation, we wire them up in series:
For Kleene closure, we use the construction below.

Question: why do we need the separate arrow labeled $\epsilon$ at the bottom of the figure?
This completes the proof, and also gives an algorithm for constructing the $\epsilon$-NFA: parse the regular expression and build it iteratively using the constructions above.

From this we get

**Corollary.** When run on a regular expression with $n$ operators, the algorithm given by the proof above constructs an $\epsilon$-NFA with at most $4n + 2$ states and $5n + 1$ transitions.

**Proof.** By induction on $n$. The base case is $n = 0$. In each of the three base cases of the proof, the algorithm gives an $\epsilon$-NFA with at most 2 states and 1 transition.
Now assume the corollary is true for $< n$ operators. We prove it for $n$.
For the cases of union and concatenation, say $r_1$ has $n_1$ operators and $r_2$ has $n_2$ operators and $r$ has $n = n_1 + n_2 + 1$ operators.

Then by induction

1. $r_1$ is compiled into an $\epsilon$-NFA with at most $4n_1 + 2$ states and $5n_1 + 1$ transitions; and
2. $r_2$ is compiled into an $\epsilon$-NFA with at most $4n_2 + 2$ states and $5n_2 + 1$ transitions.
Case 1: \( r = r_1 \cup r_2 \). By the construction \( r \) is compiled into an \( \epsilon \)-NFA with at most
\[
4n_1 + 2 + 4n_2 + 2 + 2 = 4(n_1 + n_2 + 1) + 2 = 4n + 2 \text{ states and}
\]
\[
5n_1 + 1 + 5n_2 + 1 + 4 = 5(n_1 + n_2 + 1) + 1 = 5n + 1 \text{ transitions.}
\]

Case 2: \( r = (r_1)(r_2) \). By the construction \( r \) is compiled into an \( \epsilon \)-NFA with at most
\[
4n_1 + 2 + 4n_2 + 2 + 2 = 4n + 2 \text{ states and}
\]
\[
5n_1 + 1 + 5n_2 + 1 + 3 = 5(n_1 + n_2 + 1) = 5n \text{ transitions.}
\]
The Kleene closure case

In this case \( r_1 \) has \( n_1 \) operators and \( r = (r_1)^* \) has \( n = n_1 + 1 \) operators.

**Case 3:** \( r = (r_1)^* \). Then by the construction \( r \) is compiled into an \( \epsilon \)-NFA with at most \( 4n_1 + 1 + 2 = 4(n_1 + 1) - 1 = 4n - 1 \) states and \( 5n_1 + 1 + 4 = 5(n_1 + 1) = 5n \) transitions.

In each case, then, the bound holds. The proof is complete.
Efficient conversion

Thus our algorithm to convert from a regular expression to an $\epsilon$-NFA is very efficient: it runs quickly, and the $\epsilon$-NFA it constructs is linear in the size of the original regular expression.

Unfortunately, converting in the other direction, from automaton to regular expression, can result in exponential blowup, in the worst case. There is no way around this. Too bad!
We describe how to convert from a (possibly nondeterministic) finite automaton to a regular expression. This method is somewhat easier to use than the one described in John Watrous’ course notes, pp. 43–44. But you should read that proof, too!

The idea is to expand our notion of $\epsilon$-NFA to allow transitions on arbitrary regular expressions, not simply single symbols or $\epsilon$.

Then we successively eliminate states, replacing transitions that enter and leave a state with a more complicated regular expression, until eventually there are only two states left: a start state, an accepting state, and a single transition connecting them, labeled with a regular expression.
Automaton to regular expression

To begin with, the automaton should have

▷ a start state $s$ that has no transitions into $s$ (including self-loops), and which is not accepting.
  
  ▷ If your automaton does not obey this property, add a new, non-accepting start state $s'$, and add an $\epsilon$-transition from $s'$ to $s$.

▷ The automaton should also have a single accepting state $q$ with no transitions leaving $q$ (including self-loops).
  
  ▷ If your automaton does not obey this property, add a new accepting state $q'$, change all other accepting states to non-accepting, and add $\epsilon$-transitions from them to $q'$.

Note that these changes don’t change the language recognized by the automaton.
Automaton to regular expression

For example, if your automaton is:

```
1 2
a,b  b
3
```

then you should modify it as follows:

```
0 1
ε  a,b  b
2
3
a
```

```
Automaton to regular expression

Next, repeat the following two steps, which together eliminate one state, until there are exactly two states left with a single transition connecting them:

1. Pick a non-start, non-accepting state q to eliminate. The state q will have i transitions in and j transitions out. Each will be labelled with a regular expression. For each of the ij combinations of transitions into and out of q, replace

\[ p \rightarrow q \rightarrow p \]

with

\[ p \rightarrow r \]

Of course, you might need to insert some parens around A, B, and C.
2. If many different transitions go between a pair of states, replace them with a single transition labeled by the union (\(\cup\)) of the individual transition labels.

For example, replace

\[ p \xrightarrow{A} r \quad B \]

with

\[ p \xrightarrow{A \cup B} r \]
Here’s an example of the method. Start with the NFA

and eliminate state 2 to get
Now eliminate state 3 to get

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

So a regular expression for the language recognized by the original automaton is \((b \cup ab^*a)a*b\).

In general, if you choose a different order to eliminate the states, you could get a different regular expression.