Last class:

- Introduced nondeterministic finite automata with ε-transitions

Today:

- Prove an NFA-ε is no more powerful than an NFA
- Introduce regular expressions

Fun:

- Assignment 1 is due
- Late deadline is next class
- Assignment 2 out
- Exam 1 in approx. two weeks

Recall

A language $L$ is called a regular language if there exists a DFA $D$ such that $L(D) = L$.

A nondeterministic finite automaton (NFA) is like a DFA but can have missing transitions or multiple transitions defined on the same input symbol.

An NFA accepts if any possible series of choices leads to an accepting state.

An NFA-ε may follow any number of ε-transitions at any point without consuming any input.
Formal definition of NFA-ε

A nondeterministic finite automaton with ε-transitions is a five-tuple \((Q, \Sigma, q_0, \delta, F)\) where

- \(Q\) is a finite set of states
- \(\Sigma\) is a finite alphabet of symbols
- \(q_0 \in Q\) is the start state
- \(F \subseteq Q\) is the set of accepting states
- \(\delta\) is the transition function from \(Q \times (\Sigma \cup \{\epsilon\})\) to \(\mathcal{P}(Q)\)

DFAs and NFA-εs

ε-transitions are a convenience, but they don’t increase the power of FAs

For any NFA-ε, there is an equivalent DFA, i.e., one that accepts the same language.

The construction is similar to the NFA-to-DFA construction

Creating a DFA from an NFA-ε

1. Compute the ε-closure for each state, i.e., the set of states reachable from that state following only ε-transitions.
2. The start state of the DFA is now the ε-closure of \(q_0, E(\{q_0\})\)
3. Define \(\delta\) for each \(a \in \Sigma\) and each ε-closed set \(S\):
   - If a state \(p \in S\) can reach state \(q\) on input \(a\) (not \(\epsilon\!\!\)), then add a transition on input \(a\) from \(S\) to \(E(q)\)
4. The set of final states for the DFA now includes those sets that contain at least one accepting state of the NFA-ε.

ε-closure example

Find the set \(T = E(\{s\})\)

- \(T = \{s\}\) initial step
- \(T = \{s, w\}\) add \(\delta(s, \epsilon)\)
- \(T = \{s, w, q_0\}\) add \(\delta(w, \epsilon)\)
- \(T = \{s, w, q_0, p, t\}\) add \(\delta(q_0, \epsilon)\)
- \(\delta(p, \epsilon) = \delta(t, \epsilon) = \emptyset\), so we are done: \(E(\{s\}) = T = \{s, w, q_0, p, t\}\)
Example

1. Compute ε-closure for all states:
   \[ E(q) = \{ q \} \]
   \[ E(r) = \{ r, s \} \]
   \[ E(s) = \{ r, s \} \]

2. Start state = \( E(q) = \{ q \} \)

3. Compute \( \delta \) for input and set of states from step 1.
   \[ \delta(q, 0) = E(s) = \{ r, s \} \]
   \[ \delta(q, 1) = E(r) = \{ r, s \} \]
   \[ \delta(r, s, 0) = E(q) = \{ q \} \]
   \[ \delta(r, s, 1) = E(q) = \{ q \} \]

4. Final states \( F_0 = \{ r, s \} \)

Problem

Convert this NFA-ε to a DFA

Step 1

\[ E(q_0) = \{ q_0, q_1, q_6 \} \]
\[ E(q_1) = \{ q_1 \} \]
\[ E(q_2) = \{ q_2, q_3 \} \]
\[ E(q_3) = \{ q_3 \} \]
\[ E(q_4) = \{ q_4, q_5 \} \]
\[ E(q_5) = \{ q_5 \} \]
\[ E(q_6) = \{ q_6 \} \]
\[ E(q_7) = \{ q_7, q_5 \} \]

Step 2

Start = \( E(q_0) = \{ q_0, q_1, q_6 \} \)

Step 3

\[ \delta(q_0, q_1) = E(q_2, q_7) = \{ q_2, q_3, q_7, q_5 \} \]
\[ \delta(q_2, q_3, q_7, q_5, a) = E(q_4) = \{ q_4, q_5 \} \]
\[ \delta(q_4, q_3, \emptyset) = \emptyset \]

Step 4

Final = \{ \{ q_0, q_7, q_5 \}, \{ q_4, q_5 \} \}
DFAs, NFAs, and NFAs with $\epsilon$-transitions all describe the same class of languages.

Thus to show a language is a regular language, you can just build an NFA-$\epsilon$ that recognizes it, rather than a DFA.

Many times it is more convenient to build an NFA-$\epsilon$ rather than a DFA, especially if you want to keep track of multiple possibilities.

We've seen we can show a language is regular by
constructing a DFA for it
constructing an NFA for it (with or without $\epsilon$ transitions)

We can also show a language is regular by constructing it out of other regular languages using closure properties.

Regular expressions are a concise notation for describing how to assemble a larger language out of smaller pieces.
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

Operators and operands

Regular expressions are built by combining three kinds of simple operands:

For any symbol $a \in \Sigma$, the regular expression $a$ represents the language $\{a\}$
The symbol $\varepsilon$ is a regular expression representing the language $\{\varepsilon\}$
The symbol $\emptyset$ is a regular expression for the empty language $\emptyset$

These are combined using symbols for the regular operations union, concatenation, and Kleene star.

Operation: Union

If $R_1$ and $R_2$ are regular expressions, $(R_1 \cup R_2)$ is a regular expression for the union of the languages of $R_1$ and $R_2$: $L((R_1 \cup R_2)) = L(R_1) \cup L(R_2)$. 
Operation: Concatenation

When we concatenate strings $w$ and $x$, we write $wx$ or $w \cdot x$ and the result is all the symbols from $w$ followed by all the symbols from $x$, e.g.,

- $w = \text{wonder}$
- $x = \text{woman}$
- $wx = \text{wonderwoman}$

This is like the $+$ operator in some programming languages, e.g., in Python,

```python
> w = 'wonder'
> x = 'woman'
> w+x
'wonderwoman'
```

Operation: Concatenation, continued

The concatenation of two languages $L_1$ and $L_2$ is the language $L_1L_2 = \{wx \mid w \in L_1 \text{ and } x \in L_2\}$

E.g., consider the languages

- $\text{Noun} = \{\text{Puppy, Rainbow, Whale, ...}\}$
- $\text{Verb} = \{\text{Hugs, Juggles, Loves, ...}\}$
- $\text{Det} = \{A, The\}$

The language $\text{DetNounVerbDetNoun}$ is

```python
{A\text{PuppyHugsTheWhale, TheRainbowJugglesTheRainbow, TheWhaleLovesAPuppy, ...} }
```

Operation: Concatenation, continued

Two views of $L_1L_2$:

- The set of all strings that can be made by concatenating a string in $L_1$ with a string in $L_2$.
- The set of strings that can be split into two pieces: a piece from $L_1$ followed by a piece from $L_2$.

Conceptually it’s similar to the Cartesian product of two sets, only with strings.

Operation: Concatenation, continued

If $R_1$ and $R_2$ and regular expressions, $(R_1 \cdot R_2)$ is a regular expression for the concatenation of the languages of $R_1$ and $R_2$. 
Operation: Kleene star

We can concatenate a language with itself.

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

\( LL \) is the language 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 \( L \):

\[ \{aaaaaa, aaaab, aabaa, aabb, baaaa, baab, bbaa, bbb\} \]

Etc.

Operation: Kleene star, continued

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 the empty string.

\( L^{n+1} = LL^n \)

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

---

Operation: Kleene star, continued

An important operation on languages is the **Kleene closure**, defined as \( L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots \)

Intuitively, it gives all possible ways of concatenating any number of copies of strings in \( L \) together.

Operation: Kleene star, continued

If \( R \) is a regular expression, \((R^*)\) is a regular expression for the Kleene closure of the language of \( R \).
Formal definition of regular expressions

**DEFINITION.** $R$ is a *regular expression* if $R$ is

<table>
<thead>
<tr>
<th>Basis</th>
<th>Recursive cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $a$ for some $a \in \Sigma$</td>
<td>4 $(R_1 \cup R_2)$, where $R_1$ and $R_2$ are regular expressions</td>
</tr>
<tr>
<td>2 $\varepsilon$</td>
<td>5 $(R_1 \cdot R_2)$, where $R_1$ and $R_2$ are regular expressions</td>
</tr>
<tr>
<td>3 $\emptyset$</td>
<td>6 $(R_1^*)$, where $R_1$ is a regular expression</td>
</tr>
</tbody>
</table>

Every regular expression arises by a finite number of applications of these six rules.

Order of operations

We can omit parentheses (and the concatenation operator) to make regular expressions more compact, but this makes them ambiguous without defining precedence.

0 Parentheses – $(R)$

1 Kleene star – $R^*$

2 Concatenation – $R_1 R_2$ or $R_1 \cdot R_2$

3 Union – $R_1 \cup R_2$

Empty strings, empty sets

Do not confuse the regular expressions:

- $\varepsilon$ – the language containing only the empty string
- $\emptyset$ – the language containing no strings

**Identities:**

- $R \cup \emptyset = R$
- $R \cdot \varepsilon = R$

**Example:** To prove $((a(b^*))ua)$ is a regular expression over $\Sigma = \{a, b\}$, show it can be constructed according to the rules:

1 $b$ is regular by Rule 1
2 $(b^*)$ is regular by Rule 6
3 $a$ is regular by Rule 1
4 $(a(b^*))$ is regular by Rule 5
5 $((a(b^*))ua)$ is regular by Rule 4 applied to expressions (4) and (3)
Examples

\( L(001) = \{001\} \)

\( L(0 \cup 1^*) = \{0, 1, 10, 100, 1000, \ldots\} \)

\( L(((0(0 \cup 1))^*) = \) the set of strings of 0s and 1s, of even length, such that every odd position has a 0

A few more examples…

\( ab^*a \)

\( a^*b^* \)

\((ab)^*\)

Is this the same as \(a^*b^*\)?

\( a^*b^*a^* \)

Is baa in this?

\( L = \{x^{\text{odd}}\} = x(xx)^* \) or \( (xx)^*x \) but not \( x^*xx^* \)

All strings of as and bs of exactly length 3

\( L = \{aba, baa, aba, bab, bba, bbb\} \)

or \((a \cup b)(a \cup b)(a \cup b)\)

or \((a \cup b)^*\)

Convenient shorthands

\( R^+ = R R^* \) – that is, one or more strings from language \( R \) concatenated together.

\( R^k = \) concatenation of \( k \) \( R \)s.
Equality of REs

Two regular expressions $s$ and $t$ are *equal* if and only if $L(s) = L(t)$

Two regular expressions can look quite different yet describe the same language.

Example:

\[ s = (a \cup b)^* \]
\[ \text{and} \]
\[ t = (b \cup aa^*b)^*a^* \]

Acknowledgments

This lecture incorporates material from:

- Nancy Ide
- Keith Schwarz
- Michael Sipser
- Jeffrey Ullman