Lecture 29: Kleene’s theorem

So far: three ways to specify strings

- **DFA** (low level: easy to evaluate/analyze; hard to create)
- **NFA**
- **RE** (high level: difficult to evaluate/analyze; easy to create*)

Claim (Kleene’s theorem):

The following are equivalent:

- $L$ is DFA-rec
- $L$ is NFA-rec
- $L$ is regular

* mostly
Converting RE to NFA

**Claim:** If $L$ is regular, then $L$ is NFA-recognizable.

**Proof:** Suppose $L$ is regular, so that there exists $r \in RE$ with $L(r) = L$. We want to show that $L$ is NFA-recognizable, i.e. that there exists an NFA $N$ with $L(N) = L(r)$.

Stated another way, we want to show $\forall r \in RE, \exists N \in NFA, L(N) = L(r)$.

Let $P(r)$ be the statement “$\exists N \in NFA, L(N) = L(r)$.”

By definition, $L(r) = \{w \in \Sigma^* | r \vdash w\}$, where $r$ is a regular expression.

Let $N = N_1 \cup \epsilon$.

Then $L(N) = L(r)$ by inspection.

Let $N = N_1 \cup \epsilon$.

Let $N = N_1 \cup \epsilon$.

Assume $P(r_1)$ and $P(r_2)$.

So $\exists N_1 \cup \epsilon \in NFA$ with $L(N_1) = L(r_1)$ and $\exists N_2 \cup \epsilon \in NFA$ with $L(N_2) = L(r_2)$.

Let $N = N_1 \cup N_2 \cup \epsilon$.

So $\exists N \in NFA$ with $L(N) = L(r_1 \cup r_2)$ (which is $L(r_1) \cup L(r_2)$).

**P(r+)**

Let $N = N_1 \cup \epsilon$.

Assume $P(r)$.

Let $N = N_1 \cup \epsilon$.

Let $N = N_1 \cup \epsilon$.

Let $N$ contain $N_1 \cup N_2$.

- Make all states of $N_1$ non-accepting.
- Add $\epsilon$-trans from accept. of $N_1$ to start of $N_2$.

$L(N) = L(r^+)$. 

\[ r = x_1 x_2 \ldots x_n | x_i \in L(r) \text{ where } x_i \neq \epsilon \text{ is accepted.} \]
Converting NFA to RE

**Claim:** If \( L \) is NFA-recognizable, then \( L \) is regular.

**Proof:** Suppose \( L \) is NFA-recognizable, so that there exists an NFA \( N \) with \( L(N) = L \). We want to show that \( L \) is regular, by constructing an \( r \in RE \) with \( L(r) = L(N) = L \).

A generalized NFA is like an NFA, but allows edges to label transitions.

A string \( x \) should be accepted if there's a path \( q_0 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_m \) with \( r_i = a_i \) making \( x \).

Step 1: simplify \( N \) so there is only one accept state, only one start state, no edges out of accept state, no edges into start state.

\[
(atbc*ef)(def)^*(def+g) + bcp^*h
\]
An unpopular opinion: don't use most RE libraries in real life.

```
sequence

sequence [optional (repeat (digit, 3)), optional ('-'), repeat (digit, 3), optional(4), repeat (digit, 3)]

digit * 3

optional_first = optional(repeat (digit, 3))
```
Turing machines (beyond the scope of 2800)

We can add a few features to DFA to make them as powerful as general purpose computers:

- Move both directions while processing input
- Move past the end of the input
- Overwrite the input
Some results about turing machines:

- There are (important) unrecognizable languages
  - Determining the behaviour of programs
- Turing machines can simulate any computer processor and programs in any programming language
- Nondeterminism (and other features) don’t change what’s computable
- Lots of interesting questions about what’s efficiently computable
- Turing machines can be simulated by a complicated set of rules for moving around three pebbles
- Turing machines can be simulated by crabs

Take 4810 (automata theory) or 4820 (algorithms) for more!