Warm-Up Problem

- What is the definition of an NFA? (Try it without looking!)
- Write an NFA over Σ = {a, b, c} that accepts

  $L = \{\text{abc}\} \cup \{w : w \text{ ends with cc}\}.$

CS 241 Lecture 8

Non-Deterministic Finite Automata with ε transitions

With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

Non-Deterministic Finite Automata

The above idea can be mathematically described as follows:

Definition

An **NFA** is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:

- Σ is a finite non-empty set (alphabet).
- Q is a finite non-empty set of states.
- $q_0 \in Q$ is a start state
- $A \subseteq Q$ is a set of accepting states
- $\delta : (Q \times \Sigma) \to 2^Q$ is our [total] transition function. Note that 2^Q denotes the *power set* of Q, that is, the set of all subsets of Q. This allows us to go to multiple states at once!

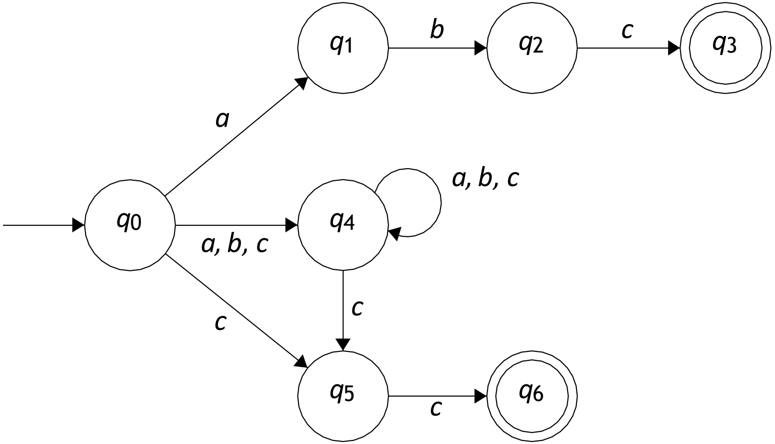
More Examples In Class

Let $\Sigma = \{a, b, c\}$. Write an NFA and the associated DFA for the following examples:

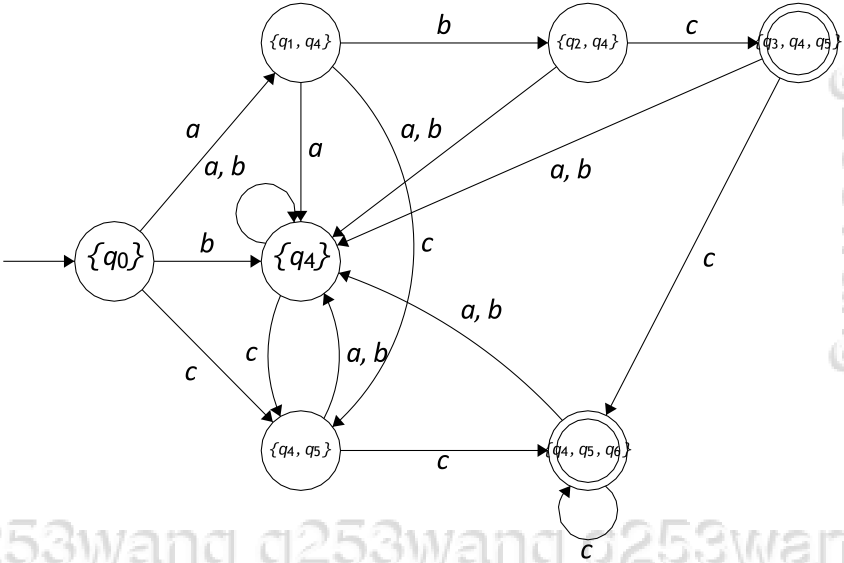
- L = {abc} ∪ {*w* : *w* ends with cc}
- L = {abc} ∪ {*w* : *w* contains cc}
- L = {*w* : *w* contains cab} ∪
  {*w* : *w* contains an even number of bs}
- L = {*w* : *w* contains exactly one abb} ∪
  {*w* : *w* does not contain ac}

First Example (Non-Optimal)

$L = \{abc\} \cup \{w : w \text{ ends with } cc\}$

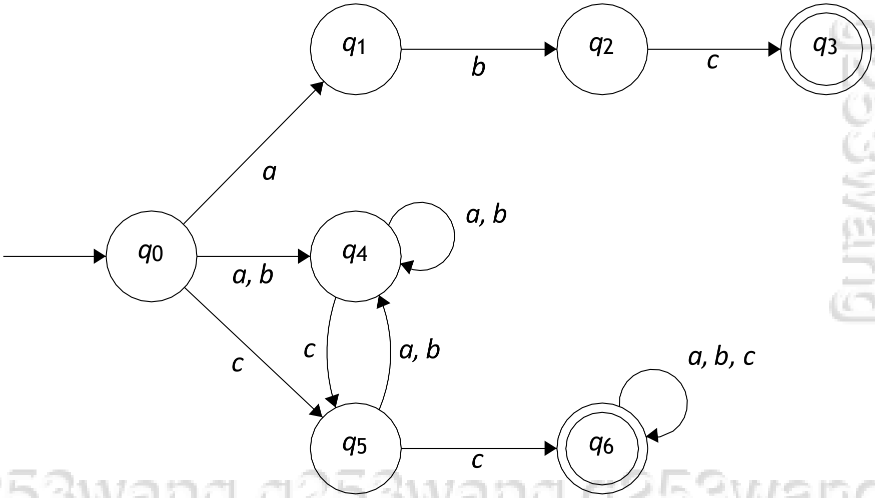


DFA Associated with Previous Example

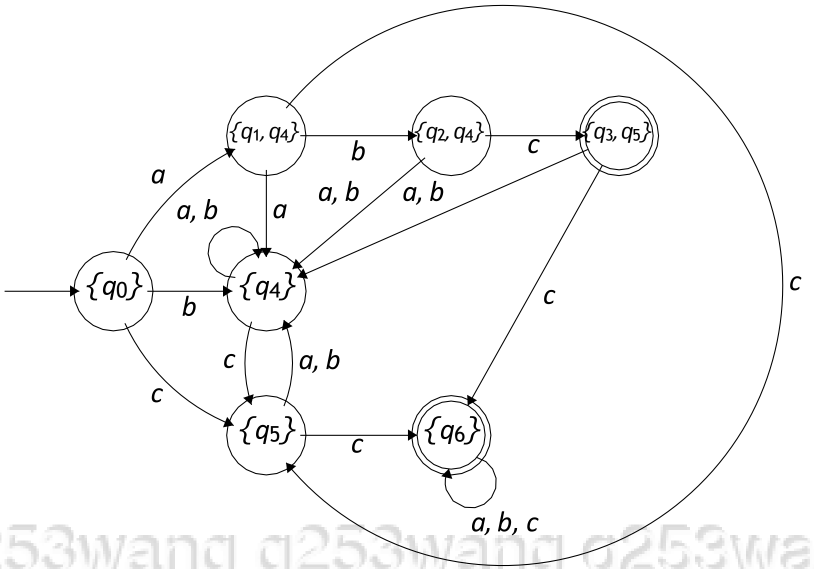


Second Example (Non-Optimal)

$L = \{abc\} \cup \{w : w \text{ contains a copy of } cc\}$



DFA Associated with Previous Example

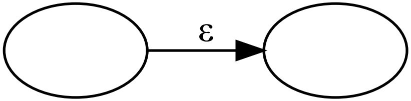


Summary

- From Kleene’s theorem, the set of languages accepted by a DFA are the regular languages
- The set of languages accepted by DFAs are the same as those accepted by NFAs
- Therefore, the set of languages accepted by an NFA are precisely the regular languages!

One More Generalization

- We gained no new computing powers from an NFA.
- What about if we permitted state changes without reading a character?
- These are known as ε transitions:



ε-Non-Deterministic Finite Automata

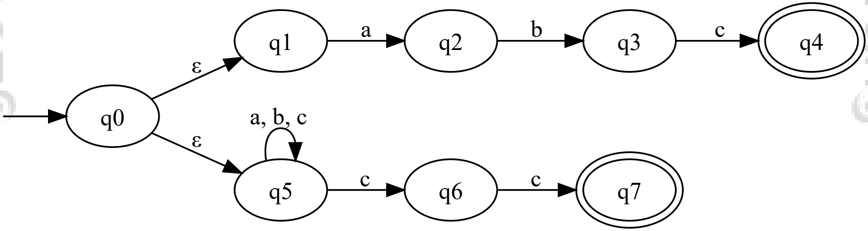
Definition

An ε**-NFA** is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:

- Σ is a finite non-empty set (alphabet) **that does not contain the symbol** ε.
- Q is a finite non-empty set of states.
- $q_0 \in Q$ is a start state
- $A \subseteq Q$ is a set of accepting states
- $\delta : (Q \times \Sigma \cup \{\varepsilon\}) \to 2^Q$ is our [total] transition function. Note that 2^Q denotes the *power set* of Q, that is, the set of all subsets of Q. This allows us to go to multiple states at once!

Why Do This?

These ε-transitions make it trivial to take the union of two NFAs: Example:
L = {abc} ∪ {*w* : *w* ends with cc}



Extending δ For an E-NFA

Define $E(S)$ to be the epsilon closure of a set of states S, that is, the set of all states reachable from S in 0 or more ε transitions.

Note, this implies that $S \subset E(S)$.

Again we can extend the definition of $\delta : (Q \times \Sigma \cup \{\varepsilon\}) \to 2^Q$ to a function $\delta^* : (2^Q \times \Sigma^*) \to 2^Q$ via:

$$\begin{aligned} \delta^* :\ (2^Q \times \Sigma^*) &\to 2^Q \\ (S, \epsilon) &\mapsto E(S) \\ (S, aw) &\mapsto \delta^*\left(\bigcup_{q \in S} E(\delta(q,a)), w\right) \end{aligned}$$

where $a \in \Sigma$. Analogously, we also have:

Definition

An ε-NFA given by $M = (\Sigma, Q, q_0, A, \delta)$ **accepts a string** w if and only if $\delta^*(E\{q_0\}, w) \cap A \neq \emptyset$

Simulating an ε-NFA

Let E(S) to be the epsilon closure of a set of states S. Recall S ⊂ E (S).

Algorithm 4 ϵ-NFA Recognition Algorithm

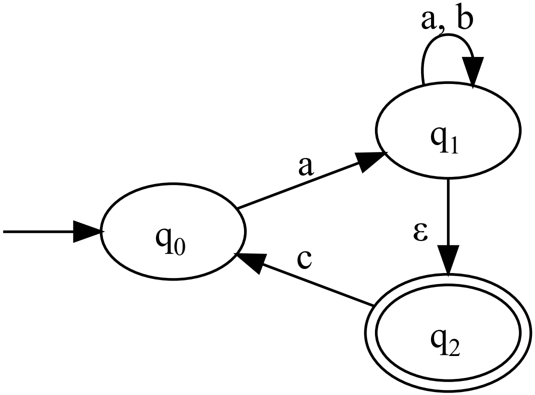
```
1: $w = a_1 a_2 \ldots a_n$
2: $S = E(\{q_0\})$
3: for i in 1 to n do
4:     $S = E\left(\cup_{q \in S} \delta(q, a_i)\right)$
5: end for
6: if $S \cap A \neq \emptyset$ then
7:     Accept
8: else
9:     Reject
10: end if
```

Tracing a Word with Previous Example

Processed	Remaining	S
ε	*abcaccc*	$\{q_0, q_1, q_5\}$
a	*bcaccc*	$\{q_2, q_5\}$
ab	*caccc*	$\{q_3, q_5\}$
abc	*accc*	$\{q_4, q_5, q_6\}$
abca	*ccc*	$\{q_5\}$
abcac	*cc*	$\{q_5, q_6\}$
abcacc	*c*	$\{q_5, q_6, q_7\}$
abcaccc	*ε*	$\{q_5, q_6, q_7\}$

Since $\{q_5, q_6, q_7\} \cap \{q_4, q_7\} \neq \emptyset$, accept.

Exercise: What regular language does this machine represent?



Second Tracing Example

Processed	Remaining	S
ε	$abca$	$\{q_0\}$
a	bca	$\{q_1, q_2\}$
ab	ca	$\{q_1, q_2\}$
abc	a	$\{q_0\}$
$abca$	ε	$\{q_1, q_2\}$

Since $\{q_1, q_2\} \cap \{q_2\} \neq \emptyset$, accept.

Third Tracing Example

Processed	Remaining	S
ε	cc	$\{q_0\}$
c	c	$\{\}$
cc	ε	$\{\}$

Since $\{\} \cap \{q_2\} = \emptyset$, reject.

Equivalences

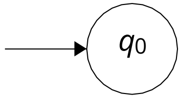
- Using the same technique as for an NFA, every ε-NFA has a corresponding DFA.
- In both cases, this technique can be automated! It doesn't necessarily give the smallest DFA, but it will give a DFA that is valid.
- This, combined with Kleene's Theorem, implies that every language recognized by an ε-NFA is regular.

Equivalences

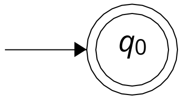
- In fact, if we can show that an ε-NFA exists for every regular expression, then we have proved one direction of Kleene’s. We do this by **structural induction**.

ε-NFAs that Recognize Regular Languages

1. *∅*

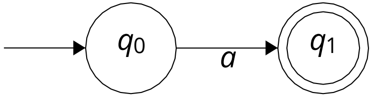


2. *{E}*



ε-NFAs that Recognize Regular Languages

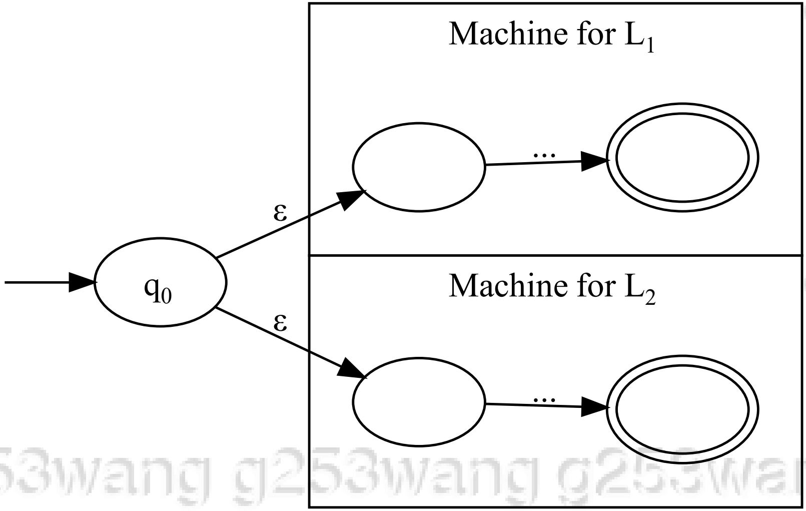
3. *{a}*



$$\begin{tikzcd} \arrow[r] & q_0 \arrow[r, "a"'] & q_1 \end{tikzcd}$$

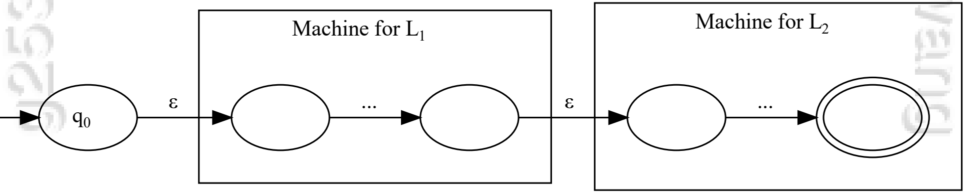
ε-NFAs that Recognize Regular Languages

4. $L_1 \cup L_2$ (that is, given ε-NFAs that recognize L_1 and L_2 already, construct one for this language)



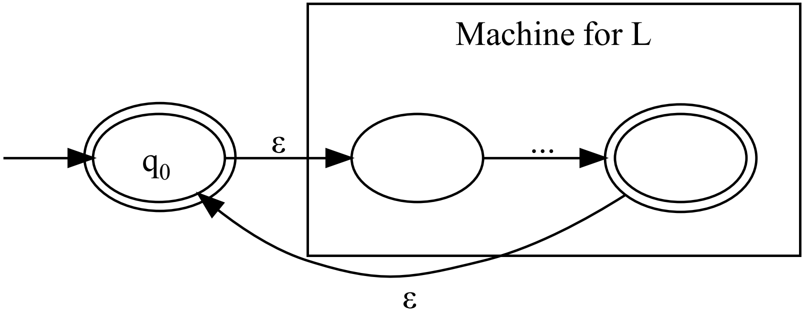
ε-NFAs that Recognize Regular Languages

5. $L_1 L_2$ (that is, given ε-NFAs that recognize L_1 and L_2 already, construct one for this language). In the diagram below, change all accepting states in the ε-NFA for L_1 to non-accepting states and for each such state, add an ε transition to the start state of L_2.



ε-NFAs that Recognize Regular Languages

6. L^*. Assume we have a ε-NFA for L already. Below, from each accepting state, add an ε transition back to the newly created start state.



Summary of Previous Slides

- For each regular language, we can construct an ε-NFA that recognizes the language.
- We can also convert each ε-NFA into a DFA.
- Both processes can be automated.
- There are several ways to go from a DFA to a regular language which we will not discuss here (see the State Removal Method for example).

Scanning

- Is C a regular language? The following are regular:
  - C keywords
  - C identifiers
  - C literals
  - C operators
  - C comments
- Sequences of these are, hence, also regular (Kleene star!). Finite automata can do our tokenization (ie. our scanning).

Scanning

- What about punctuation? Even simpler, set $\Sigma = \{ (\ ,\ ) \}$ and L = { strings with balanced parentheses }. Is L regular? More on this later.

Scanning Continued

- How does our scanner work?
- Our goal is: given some text, break the text up into tokens (eg. (ID, div), (REG, $1), (COMMA, ‘,’) (REG, $2))

Scanning Continued

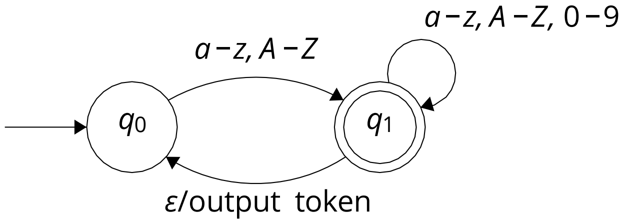
- Problem: Some tokens can be recognized in multiple different ways!
  - E.g., 0x12cc. Could be a single HEXINT, or could be an INT (0) followed by an ID (x) followed by another INT (1) followed by another INT (2) and followed by an ID (cc). (There are lots of other interpretations).
- Which interpretation is correct?

Formalization of our Problem

- Given a regular language L (say, L is all valid MIPS or C tokens), determine if a given word *w* is in LL^* (or in other words, is $w \in L^* \setminus \{\varepsilon\}$?)
  - (We don't want to consider the empty program as being valid, hence why we drop ε).

Concrete Example for C

Consider the language *L* of just ID tokens in C:



With the input *abcde*, this could be considered as anywhere from 1 to 5 different tokens! What do we do?

What do to?

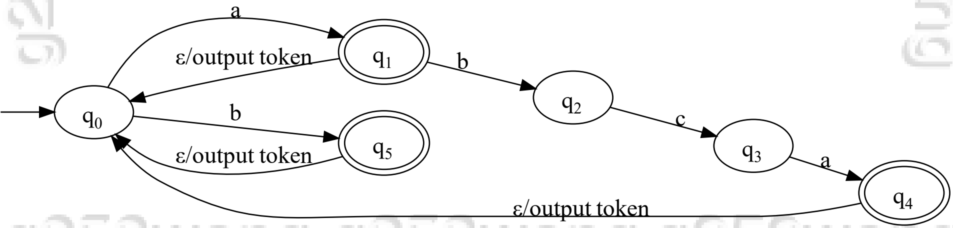
- We will discuss two algorithms: *maximal munch* and *simplified maximal munch*.
- General idea: Consume the largest possible token that makes sense. Produce the token and then proceed.

What do to?

- Difference:
  - Maximal Munch: Consume characters until you no longer have a valid transition. If you have characters left to consume, backtrack to the last valid accepting state and resume.
  - Simplified Maximal Munch: Consume characters until you no longer have a valid transition. If you are currently in an accepting state, produce the token and proceed. Otherwise go to an error state.

DFA for next two slides

- Σ = {a, b, c}, L = {a, b, abca}, consider *w* = ababca.
  - Note that *w* ∈ LL∗. What follows is an ε-NFA for LL∗ based on our algorithm:



Examples

Maximal Munch: $\Sigma = \{a, b, c\}$, $L = \{a, b, abca\}$, w = ababca. Note that $w \in LL*$.

- Algorithm consumes a **and flags this state as it is accepting**, then b then tries to consume a but ends up in an error state.
- Algorithm then backtracks to first a since that was the last accepting state. Token a is output.
- Algorithm then resumes consuming b **and flags this state as it is accepting**, then tries to consume a but ends up in an error state.
- Algorithm then backtracks to first b since that was the last accepting state. Token b is output.
- Algorithm then consumes the second a, the second b, the first c, the third a and runs out of input. This last state is accepting so output our last token abca and accept.

Examples

Simplified Maximal Munch: $\Sigma = \{a, b, c\}$, $L = \{a, b, abca\}$, w = ababca. Note that $w \in LL*$.

- Algorithm consumes a, then b then tries to consume a but ends up in an error state. **Note there is no keeping track of the first accepting state!**
- Algorithm then checks to see if ab is accepting. It is not (as $ab \notin L$).
- Algorithm rejects ababca.
- Note: This gave the wrong answer! However, this algorithm is usually good enough, and is used in practice.

Practical Implications

Consider the following C + + line:

```
vector<pair<string,int>> v;
```

Notice that at the end, there is the token >>! This, on its own, is a valid token! With either algorithm, we would reject this declaration! This was how it was in C++ before C++11. To do this declaration, you needed a space:

```
vector<pair<string,int> > v;
```

Note on Maximal Munch

- We only backtrack exactly once in this course, since once we backtrack, we output a token, and we cannot take this action back.
- It is of course possible to backtrack more, using a stack and buffering the output, but we choose not to do this in this course (or in most practical implementations).

Algorithm 5 Simplified Maximal Munch

```
w = input string
s = $q_0$
repeat
    if $\delta(s, peek(w))$ == ERROR then
        if $s \in A$ then
            Output token for state $s$
            $s = q_0$
        else
            Reject
        end if
    else
        $s = \delta(s, consume(w))$
    end if
until w is empty
if $s \in A$ then
    Output token for state $s$ and Accept
else
    Reject
end if
```

Come Up for Air

Where are we now?

1. Identify tokens (Scanning) [Complete!]
2. Check order of tokens (Syntactic Analysis) [Now]
3. Type Checking (Semantic Analysis) [Later]
4. Code Generation [Also later]

- Syntax: Is the order of the tokens correct? Do parentheses balance?
- Semantics: Does what is written make sense (right type of variables in functions etc.)