Course 4
Finite Automata/Finite State Machines (cont'd)

The structure and the content of the lecture is based on (1) http://www.eecs.wsu.edu/~ananth/CptS317/Lectures/index.htm, (2) W. Schreiner Computability and Complexity, Lecture Notes, RISC-JKU, Austria

Excursion: Previous lecture

Finite Automaton (FA)

- Informally, a state diagram that comprehensively captures all possible states and transitions that a machine can take while responding to a stream or sequence of input symbols.
- Recognizer for “Regular Languages”

- Deterministic Finite Automata (DFA)
  - The machine can exist in only one state at any given time
- Non-deterministic Finite Automata (NFA)
  - The machine can exist in multiple states at the same time

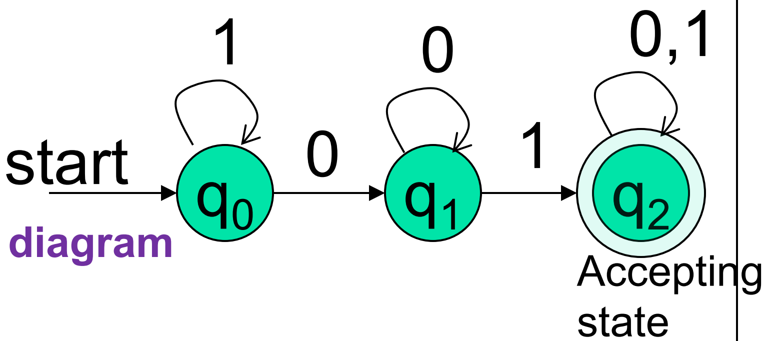
Deterministic Finite Automata - Definition

- A Deterministic Finite Automaton (DFA) consists of:
  - Q - a finite set of states
  - $\sum$ - a finite set of input symbols (alphabet)
  - q_0 - a start state (one of the elements from Q)
  - F - set of accepting states
  - $\delta : Q \times \Sigma \rightarrow Q$ – a transition function which takes a state and an input symbol as an argument and returns a state.
- A DFA is defined by the 5-tuple: $\{Q, \sum, q_0, F, \delta\}$

Regular expression: (01)*01(01)*

DFA for strings containing 01

- What makes this DFA deterministic?



Transition diagram

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0,1\}$
- start state = q_0
- $F = \{q_2\}$

Transition table

symbols

δ	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
$*q_2$	q_2	q_2

Start state

states

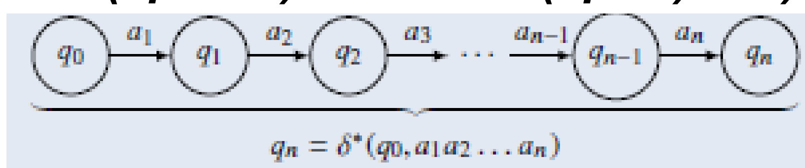
Accepting/final state

What if the language allows empty strings?

Finite Automata (cont’d)

Extension of transitions (δ) to Paths (δ*)

- $\delta^*(q,w)$ = *destination state* from state *q* on input string *w*; *used for determining the language of a DFA (see next slide)*
- *Base case:* $\delta^*(q,\varepsilon) = q$
- *Induction:* $\delta^*(q,wa) = \delta(\delta^*(q,w), a)$



- Example: Work out previous example using the input sequence:
  - $w=10$: $\delta^*(q_0,w) = ?$
  - w=1110: $\delta^*(q_0,w) = ?$

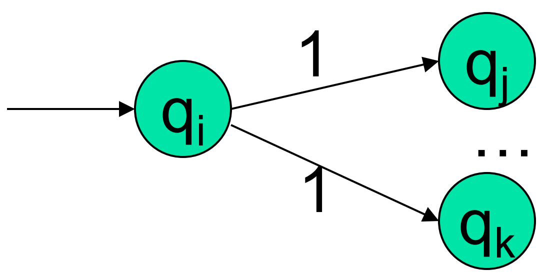
Language of a DFA

A DFA A accepts string *w* if there is a path from q_0 to an accepting (or final) state that is labeled by *w*

- *i.e.,* $L(A) = \{ w \mid \delta^*(q_0,w) \in F \}$
- *i.e., L(A) = all strings that lead to an accepting state from* q_0

Non-deterministic Finite Automata (NFA)

- A Non-deterministic Finite Automaton (NFA)
  - is of course "non-deterministic"
    - Implying that the machine can exist in more than one state at the same time
    - Transitions could be non-deterministic



• Each transition function therefore maps to a <u>set</u> of states

Non-deterministic Finite Automata (NFA)

- A Non-deterministic Finite Automaton (NFA) consists of:
  - Q: a finite set of states
  - Σ: a finite set of input symbols (alphabet)
  - Q_0: a start state
  - F: set of accepting states
  - δ: a transition function, which is a mapping between $Q \times \Sigma$ and a subset of Q
- An NFA is also defined by the 5-tuple:
  - $\{Q, \Sigma, q_0, F, \delta\}$

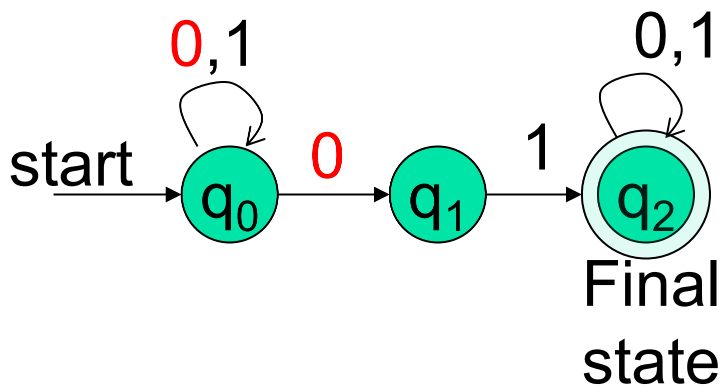
How to use an NFA?

- Input: a word w in Σ^*
- Question: Is w acceptable by the NFA?
- Steps:
  - Start at the "start state" q_0
  - For every input symbol in the sequence w do
    - Determine all possible next states from all current states, given the current input symbol in w and the transition function
  - If after all symbols in w are consumed and if at least one of the current states is a final state then *accept w;*
  - Otherwise, *reject w.*

Regular expression: (0+1)*01(0+1)*

NFA for strings containing 01

Why is this non-deterministic?



What will happen if at state q_1 an input of 0 is received?

- $Q = \{q_0,q_1,q_2\}$
- $\Sigma = \{0,1\}$
- start state = q_0
- $F = \{q_2\}$
- Transition table

symbols

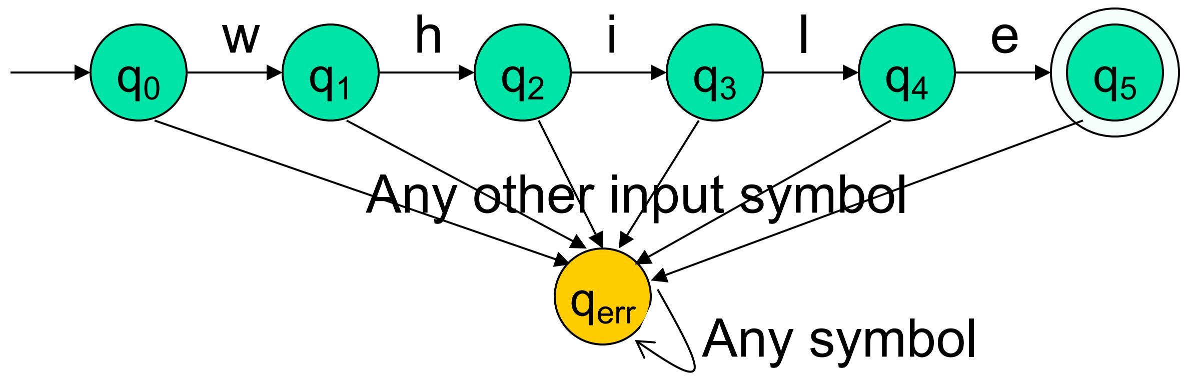
states

δ	0	1
$\rightarrow q_0$	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Φ	$\{q_2\}$
$*q_2$	$\{q_2\}$	$\{q_2\}$

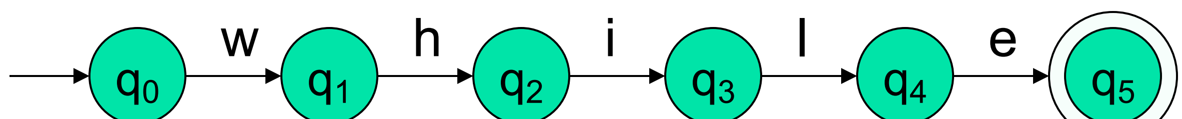
Note: Omitting to explicitly show error states is just a matter of design convenience (one that is generally followed for NFAs), and i.e., this feature should not be confused with the notion of non-determinism.

What is an "error state"?

- A DFA for recognizing the key word "*while*"



- An NFA for the same purpose:



Transitions into a dead state are implicit

Examples of NFA for different types of languages

1. $L(NFA) = \{ab^n \mid n \geq 0\} \cup \{ab^n ab^m \mid n, m \geq 0\}$ (***Union of languages***)
2. $L(NFA) = \{IF, FOR, FORK\}$ ***(Generation of finite language)***
3. $L(NFA) = \{a^n \mid n \geq 1\}$, $L(NFA) = \{a^n \mid n \geq 0\}$ ***(Repetition of symbols)***
4. $L(NFA) = \{w \mid w \in \{a, b, c\}^*\}$, $L(NFA) = \{w \mid w \in \{a, b, c\}^+\}$ (***Mix of letters***)
5. $L(NFA) = \{w \mid w \in \{0, \dots, 9\}^*, w \text{ ends with } 0\}$
6. $L(NFA) = \{a^n b^m c^k \mid m, n, k \geq 1\}$
7. $L(NFA) = \{w \mid w \in \{0,1\}^*, w \text{ contains at most one } 1\}$
8. $L(NFA) = \{w \mid w \text{ contains an even number of zeros and any number of } 1\}$
9. $L(NFA) = \{wabaw' \mid w, w' \in \{a, b, c\}^*\}$
10. $L(NFA) = \{w \mid w \text{ contains an even number of } 0's \text{ and } 1's\}$
11. $L(NFA) = \{a^i b^j \mid i, j > 0\}$
12. $L(NFA) = \{w \in \{a, b, c\}^* \mid w \text{ contains } abba\}$
13. $L(NFA) = \{w \text{ integer constant in } C \text{ with optional sign}\}$
14. $L(NFA) = \{w \text{ is divisible by } 3\}$
15. $L(NFA) = \{w \mid w \equiv 1 \bmod 3\}$

Solution*: see whiteboard.*

Extension of δ to NFA Paths δ*

- <u>Basis:</u> $\delta^* (q,\varepsilon) = \{q\}$

- <u>Induction:</u>
  - Let $\delta^* (q_0,w) = \{p_1,p_2 \ldots,p_k\}$
  - $\delta (p_i,a) = S_i$ for $i=1,2,\ldots,k$
  - *Then,* $\delta^* (q_0,wa) = S_1 \cup S_2 \cup \ldots \cup S_k$

Language of an NFA

- An NFA accepts *w* if *there exists at least one* path from the start state to an accepting (or final) state that is labeled by *w*
- $L(N) = \{ w \mid \delta^*(q_0,w) \cap F \neq \Phi \}$

Advantages & Caveats for NFA

- Great for modeling regular expressions (next lectures)
  - String processing - e.g., grep, lexical analyzer

Finite Automaton (FA)

Construct DFA and NFA for recognizing the language:

$$L(NFA) = \{awa \mid w \in \{b,c\}^* \}$$

Solution: see whiteboard.

In many cases it is easier to construct the NFA for a language.

Theorem:

A language L is accepted by a DFA *if and only if* it is accepted by an NFA, that is $L(DFA) = L(NFA)$.

Proof. See next slide

Equivalence of DFA & NFA

- Theorem:
  - A language L is accepted by a DFA *if and only if* it is accepted by an NFA i.e. $L(DFA) = L(NFA)$.

Should be true for any L

- Proof:
  1. If part i.e $L(DFA) \supseteq L(NFA)$:
     - Prove by showing every NFA can be converted to an equivalent DFA (in the next few slides…)
  2. Only-if part i.e $L(DFA) \subseteq L(NFA)$ is trivial:
     - Every DFA is a special case of an NFA where each state has exactly one transition for every input symbol. Therefore, if L is accepted by a DFA, it is accepted by a corresponding NFA.

□

Proof for the if-part

- <u>If-part:</u> Show that $L(DFA) \supseteq L(NFA)$
- *rephrasing…*
- Given any NFA N, we can construct a DFA D such that L(N)=L(D)
- How to convert an NFA into a DFA?

  - <u>Observation:</u> In an NFA, each transition maps to a ***subset*** of states
  - <u>Idea:</u> Represent:

    each "subset of NFA_states" → a single "DFA_state"

Subset construction

NFA to DFA by subset construction

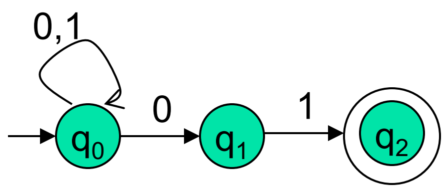
- Let $N = \{Q_N, \Sigma, \delta_N, q_0, F_N\}$
- <u>Goal:</u> Build $D=\{Q_D, \Sigma, \delta_D, \{q_0\}, F_D\}$ s.t. $L(D)=L(N)$
- <u>Construction:</u>
  1. Q_D= all subsets of Q_N (i.e., power set)
  2. F_D=set of subsets S of Q_N s.t. $S \cap F_N \neq \Phi$
  3. δ_D: for each subset S of Q_N and for each input symbol a in Σ:
     - $\delta_D(S,a) = \bigcup_{p \in s} \delta_N(p,a)$

Idea: To avoid enumerating all of power set, do “lazy creation of states”

NFA to DFA construction: Example 1

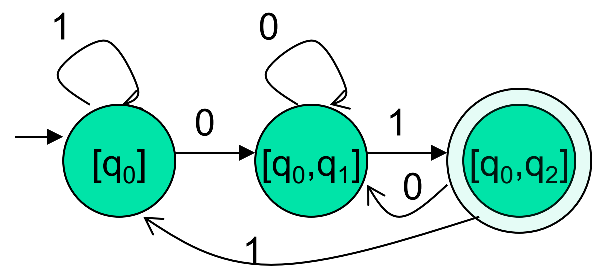
- *L = {w | w ends in 01}*

NFA:



δ_N	0	1
→q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
*q_2	Ø	Ø

DFA:



δ_D	0	1
~~Ø~~	~~Ø~~	~~Ø~~
→$[q_0]$	$\{q_0\}U\{q_1\}$	$\{q_0\}$
~~$[q_1]$~~	~~Ø~~	~~$\{q_2\}$~~
~~*$[q_2]$~~	~~Ø~~	~~Ø~~
$[q_0,q_1]$	$\{q_0,q_1\}U$Ø	$\{q_0\}U\{q_2\}$
*$[q_0,q_2]$	$\{q_0,q_1\}$	$\{q_0\}$
~~*$[q_1,q_2]$~~	~~Ø~~	~~$\{q_2\}$~~
~~*$[q_0,q_1,q_2]$~~	~~$\{q_0\}U\{q_1\}$ U Ø U Ø~~	~~$\{q_0\}$ U $\{q_2\}$ U Ø~~

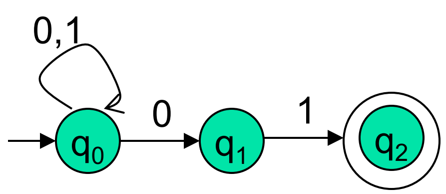
δ_D	0	1
→$[q_0]$	$[q_0,q_1]$	$[q_0]$
$[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_2]$
*$[q_0,q_2]$	$[q_0,q_1]$	$[q_0]$

0. Enumerate all possible subsets
1. Determine transitions
2. Retain only those states reachable from $\{q_0\}$

NFA to DFA: Repeating the example using *LAZY CREATION*

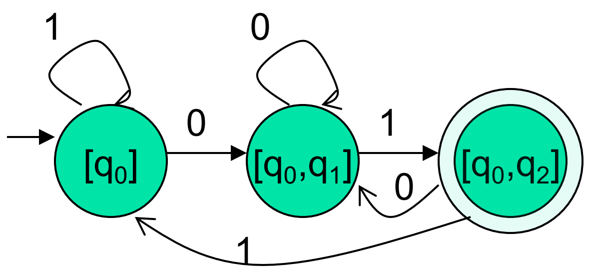
- *L = {w | w ends in 01}*

NFA:



δ_N	0	1
→ q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
$*q_2$	Ø	Ø

DFA:



δ_D	0	1
→ $[q_0]$	$[q_0,q_1]$	$[q_0]$
$[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_2]$
$*[q_0,q_2]$	$[q_0,q_1]$	$[q_0]$

Main Idea:
Introduce states as you go (on a need basis)

Correctness of subset construction

Theorem: If D is the DFA constructed from NFA N by subset construction, then L(D)=L(N)

- Proof:
  - Show that $\delta_D^*(\{q_0\},w) \equiv \delta_N^*(q_0,w)$, for all w
  - Using induction on w's length:
    - Let w = xa
    - $\delta_D^*(\{q_0\},xa) \equiv \delta_D^*(\ \delta_N(q_0,x),\ a\ ) \equiv \delta_N^*\ (q_0,w)$

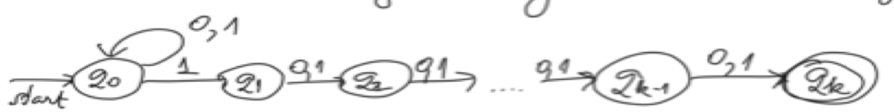
A bad case where #states(DFA)>>#states(NFA)

- Typically (not always) #states(DFA) = #states(NFA), however a DFA has more transitions.
- Worst case: #states(DFA)=2^n, #states(NFA)=n
- However, L = {w | w is a binary string s.t., the k^{th} symbol from its end is a 1}
  - NFA has k+1 states
  - But an equivalent DFA needs to have at least 2^k states
  - *(see next slide)*

Automata recogn. the lang. of binary strings s.t., the kth symbol from its end is a 1

$L = \{ w \mid w$ is a binary string s.t. the k^{th} symb. from the end is $1 \}$

NFA:



$k+1$ states

There is a state q_0 that the NFA is always in regardless of what it was read; if the next input is 1 then the NFA might "guess" that this 1 is the k^{th} symb. from the end (non-deterministic q_0, q_1). Any input from q_i $(i \geq 1)$ has effect on moving on the states.

Example: Instantiate k with 1, 2, ... and check how the NFA works.

DFA:

δ	0	1
~~ϕ~~	~~ϕ~~	~~ϕ~~
$\{q_0\}$	$\{q_0\}$	$\{q_0,q_1\}$
$\{q_1\}$	$\{q_2\}$	$\{q_2\}$
...		
$\{q_k\}$	ϕ	ϕ
$\{q_0,q_1\}$	$\{q_0,q_2\}$	$\{q_0,q_1,q_2\}$
$\{q_0,q_2\}$	$\{q_0,q_3\}$	$\{q_0,q_4\}$
...		

2^{k+1}

Question: Can I get rid of any states?
No!!! All states are readable from each other!

Always 2^{k+1} states for the DFA.

Relationship between regular lang. and type-3 lang.

- **Definition**. A type-3 grammar has a normal form if it has generating rules as follows: $A \to iB, C \to j$, where $\mathrm{A}, B, C \in V_N; i, j \in V_T$, or the completing rule $S \to \lambda$ and in this case S does not appear on the right side of any rule.
- (Parsers for right linear grammars are much simpler. Why?)

Relationship between regular lang. and type-3 lang. (cont'd)

- **Lemma**. Any type-3 grammar admits a normal form.
- **Proof sketch**. Production rules of type $A \to pB,\ p = i_1 \dots i_n$, are replaced by $A \to i_1 Z_1, Z_1 \to i_2 Z_2, \dots, Z_{n-1} \to i_n B$, Z_i are newly introduced non-terminals.
- For a rule $A \to p$ with $|p| > 1$ we do the same except the last rule which will be $Z_k \to i_n$.
- Transform $A \to Bp$ rules (how to convert a left-linear grammar to a right-linear one)

Left linear grammar

- A *left linear grammar* is a linear grammar in which the non-terminal symbol always occurs on the left side.
- Here is a left linear grammar:

$$\boxed{\begin{array}{l} S \rightarrow Aa \\ A \rightarrow ab \end{array}}$$

Right linear grammar

- A *right linear grammar* is a linear grammar in which the non-terminal symbol always occurs on the right side.
- Here is a right linear grammar:

$$S \rightarrow abaA$$
$$A \rightarrow \varepsilon$$

Left linear grammars are evil

- Consider this rule from a left linear grammar:
  `A → Babc`
- Can that rule be used to recognize this string:
  `abbabc` ?
- We need to check the rule for `B`:
  `B → Cb | D`
- Now we need to check the rules for `C` and `D`.
- This is very complicated.
- Left linear grammars require complex parsers.

Right linear grammars are good

- Consider this rule from a right linear grammar:
  `A → abcB`
- Can that rule be used to recognize this string: `abcabb` ?
- We immediately see that the first part of the string – `abc` – matches the first part of the rule. Thus, the problem simplifies to this: can the rule for `B` be used to recognize this string: `abb`
- Parsers for right linear grammars are much simpler.

Convert left linear to right linear

Now we will see an algorithm for converting any left linear grammar to its equivalent right linear grammar.

left linear		right linear
$S \to Aa$ $A \to ab$	$\Rightarrow$	$S \to abaA$ $A \to \varepsilon$

Both grammars generate this language: {aba}

May need to make a new start symbol

The algorithm on the following slides assume that the left linear grammar doesn't have any rules with the start symbol on the right hand side.

- If the left linear grammar has a rule with the start symbol S on the right hand side, simply add this rule:

  $S_0 \rightarrow S$

Symbols used by the algorithm

- Let `S` denote the start symbol
- Let `A`, `B` denote non-terminal symbols
- Let `p` denote zero or more terminal symbols
- Let ε denote the empty symbol

Algorithm

1) If the left linear grammar has a rule $S \rightarrow p$, then make that a rule in the right linear grammar
2) If the left linear grammar has a rule $A \rightarrow p$, then add the following rule to the right linear grammar: $S \rightarrow pA$
3) If the left linear grammar has a rule $B \rightarrow Ap$, add the following rule to the right linear grammar: $A \rightarrow pB$
4) If the left linear grammar has a rule $S \rightarrow Ap$, then add the following rule to the right linear grammar: $A \rightarrow p$

Convert this left linear grammar

left linear

$$S \to Aa$$
$$A \to ab$$

Right hand side has terminals

left linear

$$S \rightarrow Aa$$
$$A \rightarrow ab$$

right linear

$$S \rightarrow abA$$

2) If the left linear grammar has this rule $A \rightarrow p$, then add the following rule to the right linear grammar: $S \rightarrow pA$

Right hand side of S has non-terminal

left linear

$S \rightarrow Aa$
$A \rightarrow ab$

right linear

$S \rightarrow abA$
$A \rightarrow a$

4) If the left linear grammar has `S → Ap`, then add the following rule to the right linear grammar: `A → p`

Equivalent!

left linear

$$S \rightarrow Aa$$
$$A \rightarrow ab$$

right linear

$$S \rightarrow abA$$
$$A \rightarrow a$$

Both grammars generate this language: {aba}

Convert this left linear grammar

original grammar

$S \to Ab$
$S \to Sb$
$A \to Aa$
$A \to a$

make a new start symbol

left linear

$S_0 \to S$
$S \to Ab$
$S \to Sb$
$A \to Aa$
$A \to a$

Convert this

Right hand side has terminals

left linear

$$S_0 \rightarrow S$$
$$S \rightarrow Ab$$
$$S \rightarrow Sb$$
$$A \rightarrow Aa$$
$$A \rightarrow a$$

right linear

$$S_0 \rightarrow aA$$

2) If the left linear grammar has this rule `A → p`, then add the following rule to the right linear grammar: `S → pA`

Right hand side has non-terminal

left linear

$S_0 \rightarrow S$
$S \rightarrow Ab$
$S \rightarrow Sb$
$A \rightarrow Aa$
$A \rightarrow a$

right linear

$S_0 \rightarrow aA$
$A \rightarrow bS$
$A \rightarrow aA$
$S \rightarrow bS$

3) If the left linear grammar has a rule $B \rightarrow Ap$, add the following rule to the right linear grammar: $A \rightarrow pB$

Right hand side of start symbol has non-terminal

left linear	right linear
$S_0 \to S$	$S_0 \to aA$
$S \to Ab$	$A \to bS$
$S \to Sb$	$A \to aA$
$A \to Aa$	$S \to bS$
$A \to a$	$S \to \varepsilon$

4) If the left linear grammar has $S \to Ap$, then add the following rule to the right linear grammar: $A \to p$

Equivalent!

left linear

$$S_0 \rightarrow S$$
$$S \rightarrow Ab$$
$$S \rightarrow Sb$$
$$A \rightarrow Aa$$
$$A \rightarrow a$$

right linear

$$S_0 \rightarrow aA$$
$$A \rightarrow bS$$
$$A \rightarrow aA$$
$$S \rightarrow bS$$
$$S \rightarrow \varepsilon$$

Both grammars generate this language: $\{a^+b^+\}$

Will the algorithm always work?

- We have seen two examples where the algorithm creates a right linear grammar that is equivalent to the left linear grammar.
- But will the algorithm *always* produce an equivalent grammar?
- Yes! Check *Introduction to Formal Languages* by Gyorgy Revesz for the proof.

Convert right linear grammar to left linear grammar

Example:

N = {S, A}, T = {a, b, c}, S, P:

$$\boxed{\begin{array}{l} S \to aS \mid cA \\ A \to bA \mid b \end{array}}$$

Idea:

- Given the grammar construct the FA;
- Switch between start and final states
- Reverse transitions

Relationship between regular lang. and type-3 lang. (cont'd)

- **Theorem**. The family of type-3 languages is equal to the family of regular languages.
- Useful for the constructing a type-3 grammar from an automata and viceversa.

Relationship between regular lang. and type-3 lang. (cont'd)

- $G = (N, T, S, P)$, $\mathrm{FA} = (Q, \Sigma, q_0, F, \delta)$

For any regular grammar G (in normal form) there exists a nondeterministic finite automaton A such that $L(A) = L(G)$:

In grammar G	In automaton A
T	$\Sigma = T$
N	$Q = N \cup \{f\}, F = \{f\}$
S	$q_0 = S$
$q \rightarrow ap$	$p \in \delta(q, a)$
$q \rightarrow a$	$f \in \delta(q, a)$
if $S \rightarrow \epsilon$	add S to F

Relationship between regular lang. and type-3 lang. (cont'd)

- $G = (N, T, S, P)$, $\text{FA} = (Q, \Sigma, q_0, F, \delta)$

For any deterministic finite automaton there exists a regular grammar G such that $L(A) = L(G)$:

In automaton A	In grammar G
Σ	$T = \Sigma$
Q	$N = Q$
q_0	$S = q_0$
$\delta(q, a) = p$	$q \to ap$
$\delta(q, a) \in F$	$q \to a$
if $q_0 \in F$	add rule $q_0 \to \epsilon$

q0 is the start state, not other

Applications

- Text indexing
  - inverted indexing
  - For each unique word in the database, store all locations that contain it using an NFA or a DFA
- Find pattern P in text T
  - Example: Google querying
- Extensions of this idea:
  - PATRICIA tree, suffix tree

But, DFAs and NFAs are equivalent in their power to capture langauges !!

Summary DFA vs NFA

- **DFA**
1. All transitions are deterministic
   - Each transition leads to exactly one state
2. For each state, transition on all possible symbols (alphabet) should be defined
3. Accepts input if the last state visited is in F
4. Sometimes harder to construct because of the number of states
5. Practical implementation is feasible

- **NFA**
1. Some transitions could be non-deterministic
   - A transition could lead to a subset of states
2. Not all symbol transitions need to be defined explicitly (if undefined will go to an error state – this is just a design convenience, not to be confused with "non-determinism")
3. Accepts input if *one of* the last states is in F
4. Generally easier than a DFA to construct
5. Practical implementations limited but emerging (e.g., Micron automata processor)

A few subtle properties of DFAs and NFAs

- The machine never really terminates.
  - It is always waiting for the next input symbol or making transitions.
- The machine decides when to <u>consume</u> the next symbol from the input and when to <u>ignore</u> it.
  - (but the machine can never <u>skip</u> a symbol)
- => A transition can happen even *without* really consuming an input symbol (think of consuming ε as a free token) – if this happens, then it becomes an ε-NFA (**see next lecture**).
- A single transition *cannot* consume more than one (non-ε) symbol.