Regular Expressions and Regular Grammars

Algebraic Descriptions and Grammatical Foundations

Theory of Computation Slide Sets

Table of Contents

Learning Outcomes

By the end of this slide set, you should be able to:

- interpret a regular expression as a language and apply precedence rules correctly;
- convert a regular expression to an equivalent ε-NFA;
- convert a finite automaton to a regular expression by state elimination;
- recognize right-linear and left-linear grammars and convert between regular grammars and finite automata;
- justify conversions using a path/derivation invariant; and
- distinguish a grammar's syntactic form from its language's class.

Part 1: Inductive Definition of RE

Regular expressions (REs) provide a declarative, algebraic way to describe languages. We define them inductively based on the regular operations.

Part 1: Inductive Definition of RE

Regular expressions (REs) provide a declarative, algebraic way to describe languages. We define them inductively based on the regular operations.

Formal Definition

Given an alphabet Σ, R is a regular expression if R is built by the following rules, and no others. Its denoted language is written $L(R)$.

1. a for $a \in \Sigma$, with $L(a) = \{a\}$;
2. ε, with $L(\varepsilon) = \{\varepsilon\}$;
3. $\emptyset$, with $L(\emptyset) = \emptyset$;
4. $(R_1 \cup R_2)$, with $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$;
5. $(R_1 \circ R_2)$, with $L(R_1 \circ R_2) = L(R_1)L(R_2)$;
6. (R_1^*), with $L(R_1^*) = [L(R_1)]^*$.

Part 1: Inductive Definition of RE

Regular expressions (REs) provide a declarative, algebraic way to describe languages. We define them inductively based on the regular operations.

Formal Definition

Given an alphabet Σ, R is a regular expression if R is built by the following rules, and no others. Its denoted language is written $L(R)$.

1. a for $a \in \Sigma$, with $L(a) = \{a\}$;
2. ε, with $L(\varepsilon) = \{\varepsilon\}$;
3. $\emptyset$, with $L(\emptyset) = \emptyset$;
4. $(R_1 \cup R_2)$, with $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$;
5. $(R_1 \circ R_2)$, with $L(R_1 \circ R_2) = L(R_1)L(R_2)$;
6. (R_1^*), with $L(R_1^*) = [L(R_1)]^*$.

Notation Note

We use $\cup$ for union and ε for the empty string. Some texts instead use $+$ for union and λ for the empty string. Here $R^+ = RR^*$ abbreviates one or more repetitions; this superscript $+$ is not union.

Precedence Rules and Empty-Language Notation

To avoid excessively nested parentheses, we establish an order of operations:

$$\textbf{Star } (*) \quad > \quad \textbf{Concatenation } (\circ) \quad > \quad \textbf{Union } (\cup)$$

For example, $0 \cup 10^*$ is parsed as $0 \cup (1 \circ (0^*))$, not $(0 \cup 1) \circ 0^*$.

Precedence Rules and Empty-Language Notation

To avoid excessively nested parentheses, we establish an order of operations:

$$\textbf{Star } (*) \quad > \quad \textbf{Concatenation } (\circ) \quad > \quad \textbf{Union } (\cup)$$

For example, $0 \cup 10^*$ is parsed as $0 \cup (1 \circ (0^*))$, not $(0 \cup 1) \circ 0^*$.

Teacher's Corner: $\emptyset$ vs. $\{\varepsilon\}$

A frequent source of confusion is the distinction between the expressions $\emptyset$ and ε and the languages they denote.

- $L(\emptyset) = \emptyset$ contains **no strings**.
- $L(\varepsilon) = \{\varepsilon\}$ contains **exactly one string**: the empty string.
- Concatenation identities: $R\varepsilon \equiv R$, but $R\emptyset \equiv \emptyset$, where $\equiv$ means equal languages.
- Union identities: $R \cup \emptyset \equiv R$; $R \cup \varepsilon$ ensures that the empty string is included (it may already be present).

Useful Algebraic Laws for Regular Expressions

These are language equivalences: $R \equiv S$ means $L(R) = L(S)$.

Identity and Annihilation

$$R \cup \emptyset \equiv R, \qquad R \cup R \equiv R,$$
$$R\varepsilon \equiv \varepsilon R \equiv R,$$
$$R\emptyset \equiv \emptyset R \equiv \emptyset.$$

Star and Distribution

$$\emptyset^* \equiv \varepsilon, \qquad \varepsilon^* \equiv \varepsilon,$$
$$R(S \cup T) \equiv RS \cup RT,$$
$$(R \cup S)T \equiv RT \cup ST.$$

Important Non-Law

Concatenation is not commutative in general: $ab \not\equiv ba$. These laws simplify state-elimination labels and help check answers.

Examples of Regular Expressions

Let $\Sigma = \{0, 1\}$. How do we express common patterns algebraically?

Examples of Regular Expressions

Let $\Sigma = \{0, 1\}$. How do we express common patterns algebraically?

1. Strings containing at least one 00 as a substring:

$$(0 \cup 1)^* 00 (0 \cup 1)^*$$

Explanation: The $(0 \cup 1)^*$ acts as a wildcard Σ^* representing any prefix and any suffix.

Examples of Regular Expressions

Let $\Sigma = \{0, 1\}$. How do we express common patterns algebraically?

1. Strings containing at least one 00 as a substring:

$$(0 \cup 1)^* 00 (0 \cup 1)^*$$

Explanation: The $(0 \cup 1)^*$ acts as a wildcard Σ^* representing any prefix and any suffix.

2. Strings that do not contain 00 as a substring:

$$(1 \cup 01)^* (0 \cup \varepsilon)$$

Explanation: Every nonfinal 0 is followed by a 1; at most one unmatched 0 may occur, and only at the end.

Examples of Regular Expressions

Let $\Sigma = \{0, 1\}$. How do we express common patterns algebraically?

1. Strings containing at least one 00 as a substring:

$$(0 \cup 1)^* 00 (0 \cup 1)^*$$

Explanation: The $(0 \cup 1)^*$ acts as a wildcard Σ^* representing any prefix and any suffix.

2. Strings that do not contain 00 as a substring:

$$(1 \cup 01)^* (0 \cup \varepsilon)$$

Explanation: Every nonfinal 0 is followed by a 1; at most one unmatched 0 may occur, and only at the end.

3. Strings with an even number of 0s:

$$1^* (01^* 01^*)^*$$

Explanation: 1s can appear anywhere. 0s must appear in pairs (01^*0) to ensure their total count is even.

Prove the Pattern, Not Just the Examples

Why does $R = (1 \cup 01)^*(\varepsilon \cup 0)$ describe exactly the binary strings with no occurrence of 00?

Soundness: every generated word has the property

Each repeated block is 1 or 01, so it ends in 1. Joining such blocks never creates 00; the optional final 0 cannot do so either.

Prove the Pattern, Not Just the Examples

Why does $R = (1 \cup 01)^*(\varepsilon \cup 0)$ describe exactly the binary strings with no occurrence of 00?

Soundness: every generated word has the property

Each repeated block is 1 or 01, so it ends in 1. Joining such blocks never creates 00; the optional final 0 cannot do so either.

Completeness: every valid word is generated

Scan a word with no 00 from left to right. Take each 1 as a block; pair each nonfinal 0 with its following 1 to make 01. If a final 0 remains, use the optional ending. These blocks form an R-word.

Prove the Pattern, Not Just the Examples

Why does $R = (1 \cup 01)^*(\varepsilon \cup 0)$ describe exactly the binary strings with no occurrence of 00?

Soundness: every generated word has the property

Each repeated block is 1 or 01, so it ends in 1. Joining such blocks never creates 00; the optional final 0 cannot do so either.

Completeness: every valid word is generated

Scan a word with no 00 from left to right. Take each 1 as a block; pair each nonfinal 0 with its following 1 to make 01. If a final 0 remains, use the optional ending. These blocks form an R-word.

Boundary tests: $\varepsilon, 0, 1, 010$ must be included; $00, 1001$ must be excluded. Tests catch errors, but the two inclusions establish correctness for every length.

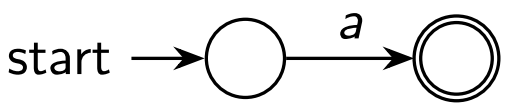
Part 2: RE to NFA — Thompson's Construction

Kleene's Theorem: A language is regular if and only if it is described by a regular expression.
Here we prove the direction *regular expression* $\Rightarrow$ *regular language* by converting any RE into an equivalent ε-NFA. First, we establish the atomic primitives:

Part 2: RE to NFA — Thompson's Construction

Kleene's Theorem: A language is regular if and only if it is described by a regular expression.
Here we prove the direction *regular expression* $\Rightarrow$ *regular language* by converting any RE into an equivalent ε-NFA. First, we establish the atomic primitives:

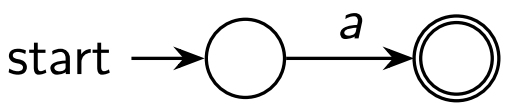
RE: a **for** $a \in \Sigma$



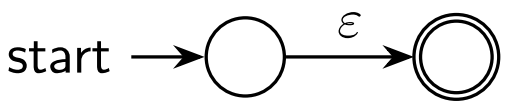
Part 2: RE to NFA — Thompson's Construction

Kleene's Theorem: A language is regular if and only if it is described by a regular expression.
Here we prove the direction *regular expression* $\Rightarrow$ *regular language* by converting any RE into an equivalent ε-NFA. First, we establish the atomic primitives:

RE: a **for** $a \in \Sigma$



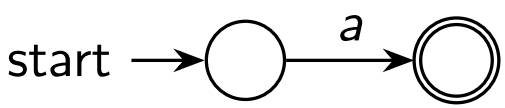
RE: ε



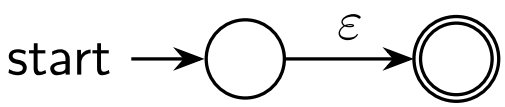
Part 2: RE to NFA — Thompson's Construction

Kleene's Theorem: A language is regular if and only if it is described by a regular expression.
Here we prove the direction *regular expression* $\Rightarrow$ *regular language* by converting any RE into an equivalent ε-NFA. First, we establish the atomic primitives:

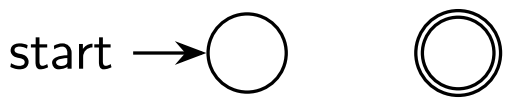
RE: a **for** $a \in \Sigma$



RE: ε



RE: $\emptyset$



Each primitive has one start state and one distinct accept state; the $\emptyset$ machine simply has no path between them.

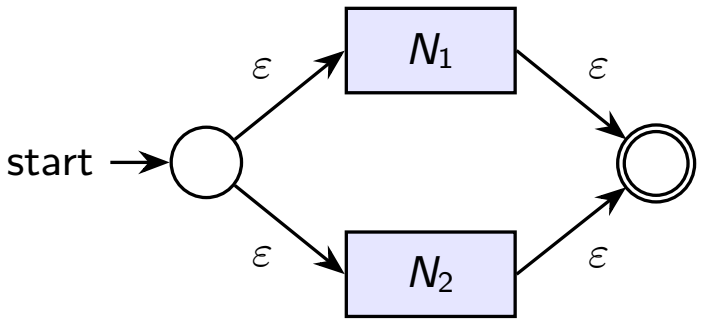
Inductive Steps: Union and Concatenation

Use disjoint copies of the Thompson fragments for R_1 and R_2. Each box's left boundary is its entry; its right boundary is its exit.

Inductive Steps: Union and Concatenation

Use disjoint copies of the Thompson fragments for R_1 and R_2. Each box's left boundary is its entry; its right boundary is its exit.

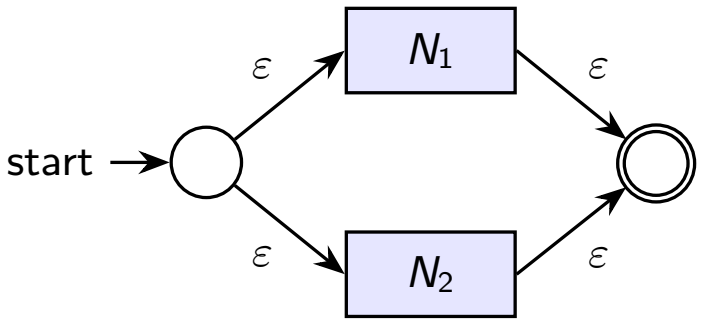
Union ($R_1 \cup R_2$)



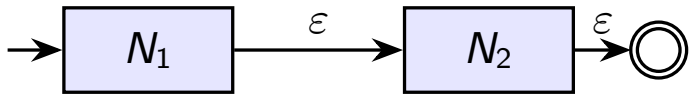
Inductive Steps: Union and Concatenation

Use disjoint copies of the Thompson fragments for R_1 and R_2. Each box's left boundary is its entry; its right boundary is its exit.

Union ($R_1 \cup R_2$)



Concatenation ($R_1 \circ R_2$)



In the combined machine, internal fragment exits lose their accepting status; only the overall exit accepts. The displayed extra final state for concatenation is optional. All joining edges consume no input.

Inductive Steps: The Star Operation

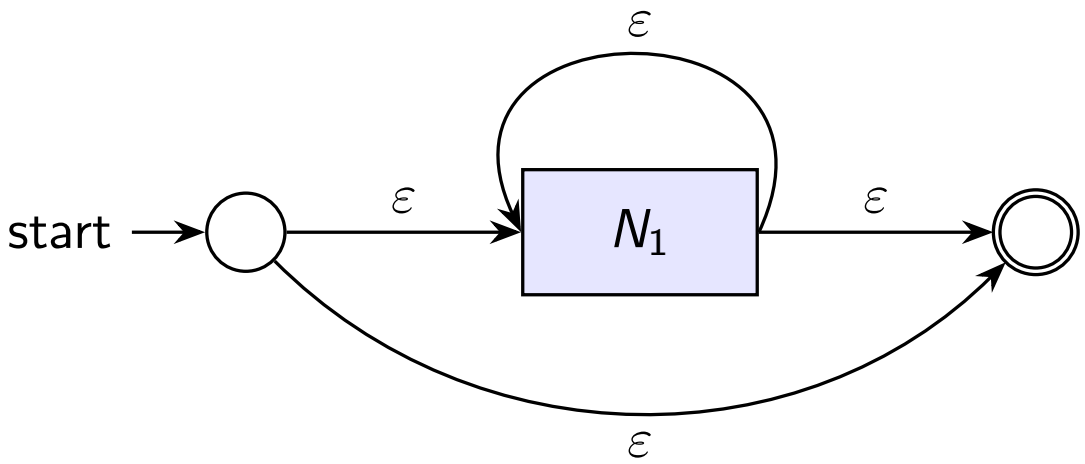
Star Operation (R_1^*)

The Star operation requires us to accept the empty string ε (zero repetitions), or loop back to the beginning of N_1 after a successful pass (one or more repetitions).

Inductive Steps: The Star Operation

Star Operation (R_1^*)

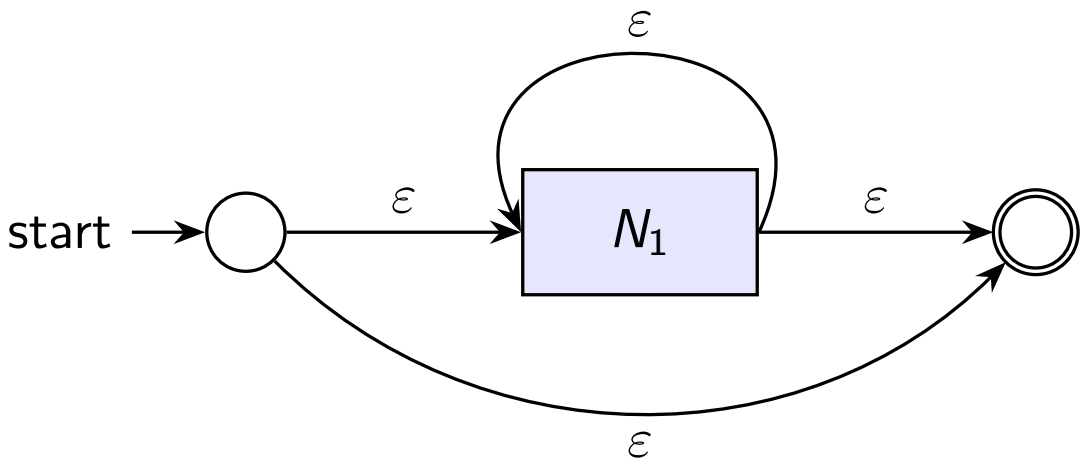
The Star operation requires us to accept the empty string ε (zero repetitions), or loop back to the beginning of N_1 after a successful pass (one or more repetitions).



Inductive Steps: The Star Operation

Star Operation (R_1^*)

The Star operation requires us to accept the empty string ε (zero repetitions), or loop back to the beginning of N_1 after a successful pass (one or more repetitions).



Why new boundary states? The new start has no incoming transitions and the new accept has no outgoing transitions. This preserves the one-entry/one-exit form needed by later inductive steps.

Worked Example: Converting $(ab \cup a)^*$

Let's build an NFA for $(ab \cup a)^*$ stage-by-stage.

Worked Example: Converting $(ab \cup a)^*$

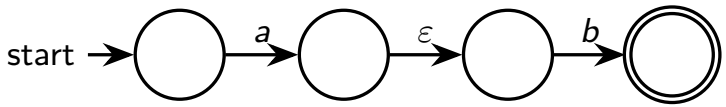
Let's build an NFA for $(ab \cup a)^*$ stage-by-stage.

1. Build separate primitives for the two occurrences of a and for b.

Worked Example: Converting $(ab \cup a)^*$

Let's build an NFA for $(ab \cup a)^*$ stage-by-stage.

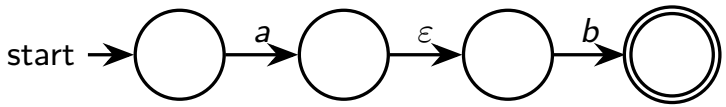
1. **Build separate primitives for the two occurrences of a and for b.**
2. **Concatenate them to obtain ab:**



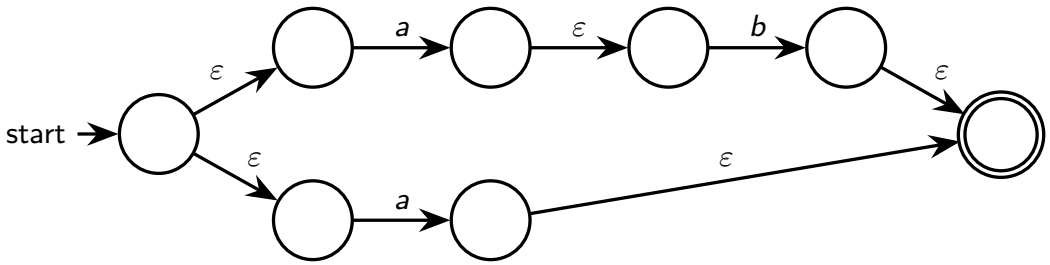
Worked Example: Converting $(ab \cup a)^*$

Let's build an NFA for $(ab \cup a)^*$ stage-by-stage.

1. **Build separate primitives for the two occurrences of a and for b.**
2. **Concatenate them to obtain ab:**

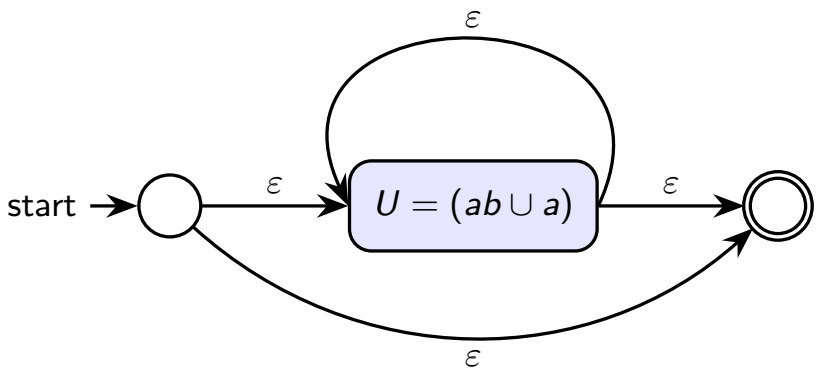


3. **Form $U = (ab \cup a)$:**



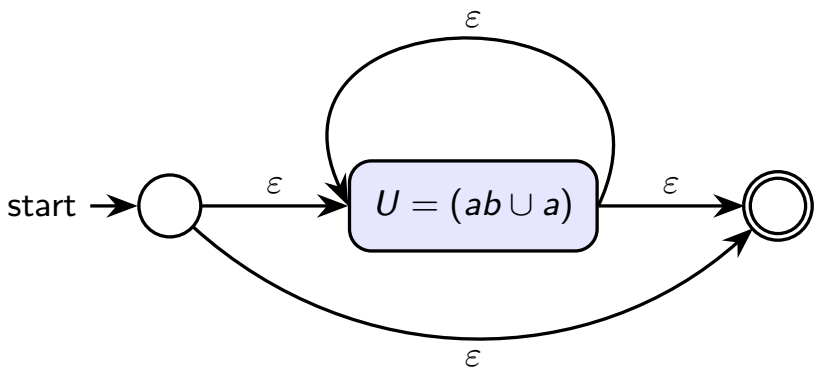
Worked Example: Star Wrapper and a Check

Recall the one-entry/one-exit fragment U for $ab \cup a$. **4. Apply the star wrapper to U:**



Worked Example: Star Wrapper and a Check

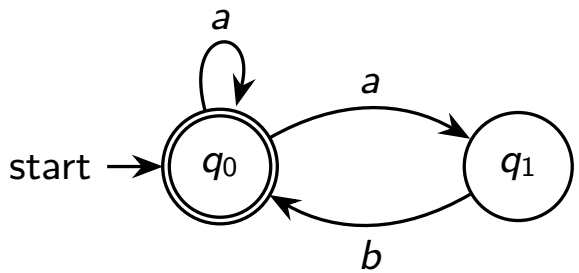
Recall the one-entry/one-exit fragment U for $ab \cup a$. **4. Apply the star wrapper to U:**



Trace the choices

ε takes the bypass. The word $abaab$ is accepted as $(ab)(a)(ab)$: enter U three times, using its exit-to-entry edge between repetitions. The word abb cannot be partitioned into a or ab blocks.

Equivalent compact NFA
(not a Thompson fragment)



Part 3: Generalized NFAs (GNFA)

To convert a finite automaton back into a regular expression, we use a **generalized NFA (GNFA)**.

Part 3: Generalized NFAs (GNFA)

To convert a finite automaton back into a regular expression, we use a **generalized NFA (GNFA)**.

What is a GNFA?

A GNFA has transition arrows labeled by **regular expressions** rather than single symbols. Traversing an edge consumes any string in the language denoted by its label.

Part 3: Generalized NFAs (GNFA)

To convert a finite automaton back into a regular expression, we use a **generalized NFA (GNFA)**.

What is a GNFA?

A GNFA has transition arrows labeled by **regular expressions** rather than single symbols. Traversing an edge consumes any string in the language denoted by its label.

Standard Form for GNFA conversion:

1. A new **start state** with an ε-arrow to the old start state, and no incoming arrows.
2. A single new **accept state** with ε-arrows from the old accept states, and no outgoing arrows.
3. Combine parallel arrows by union and regard a missing arrow as one labeled $\emptyset$. Thus each allowable ordered pair has one RE label; the start has no incoming arrows and the accept state has no outgoing arrows.

The Conversion Algorithm: State Elimination

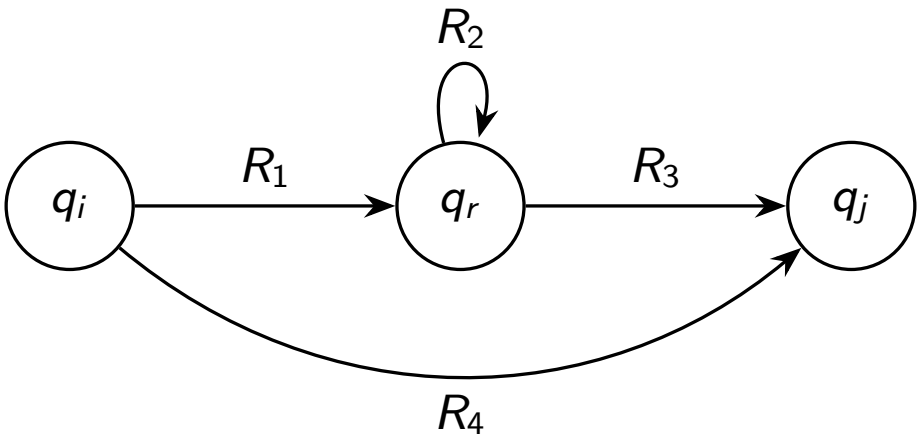
Objective: Reduce the machine to two states (start and accept) by eliminating internal states one at a time.

The Conversion Algorithm: State Elimination

Objective: Reduce the machine to two states (start and accept) by eliminating internal states one at a time.
When we eliminate a state q_r, we update the graph so that its language remains unchanged. To compute the new label on $q_i \to q_j$:

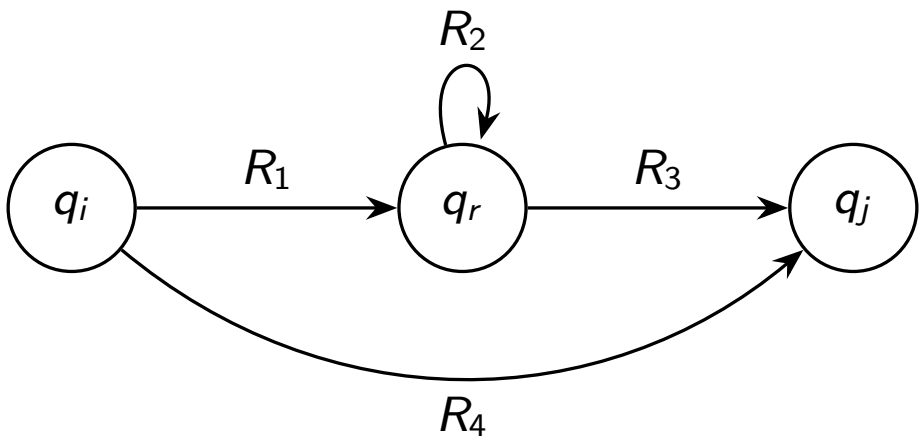
The Conversion Algorithm: State Elimination

Objective: Reduce the machine to two states (start and accept) by eliminating internal states one at a time.
When we eliminate a state q_r, we update the graph so that its language remains unchanged. To compute the new label on $q_i \to q_j$:



The Conversion Algorithm: State Elimination

Objective: Reduce the machine to two states (start and accept) by eliminating internal states one at a time.
When we eliminate a state q_r, we update the graph so that its language remains unchanged. To compute the new label on $q_i \to q_j$:



The Update Formula

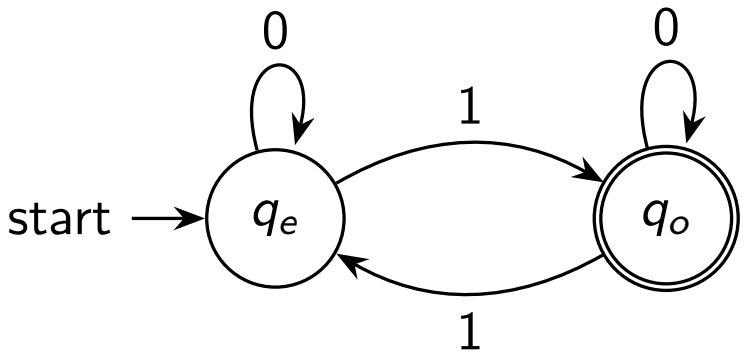
The new label bypassing q_r from q_i to q_j becomes:

$$R' = R_4 \cup R_1 R_2^* R_3$$

Use the **old labels** for every allowable pair of remaining states, including self-loops ($q_i = q_j$). Different orders may produce different-looking but equivalent expressions.

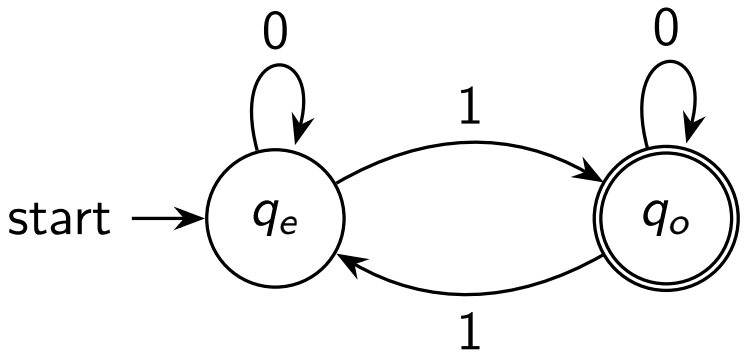
Worked Example: DFA to GNFA

Given DFA: Accepts strings with an odd number of 1s.

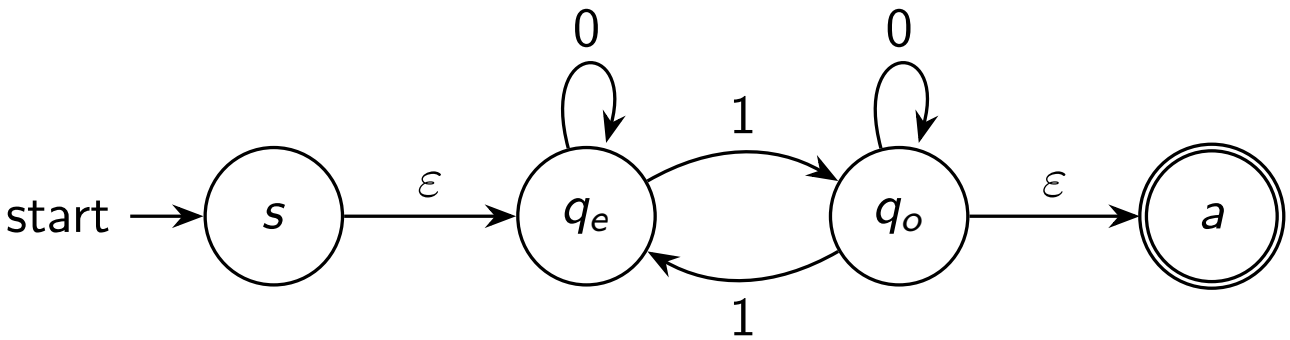


Worked Example: DFA to GNFA

Given DFA: Accepts strings with an odd number of 1s.



Step 1: Convert to Standard GNFA (Add new start s and accept a).



Worked Example: State Elimination

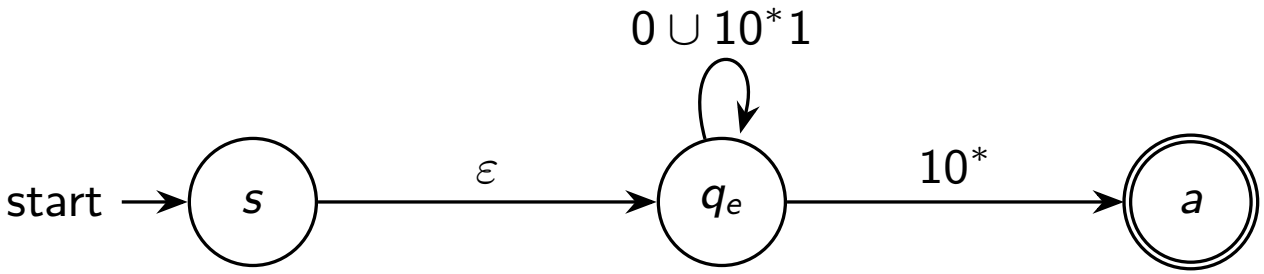
Step 2: Eliminate q_o.
Update the labels on $q_e \to q_e$ and $q_e \to a$.

- $q_e \to q_e$: the bypass through q_o contributes 10^*1. Union with the existing 0 to obtain $0 \cup 10^*1$.
- $q_e \to a$: the bypass contributes $10^*\varepsilon = 10^*$. Union with $\emptyset$ to obtain 10^*.

Worked Example: State Elimination

Step 2: Eliminate q_o.
Update the labels on $q_e \to q_e$ and $q_e \to a$.

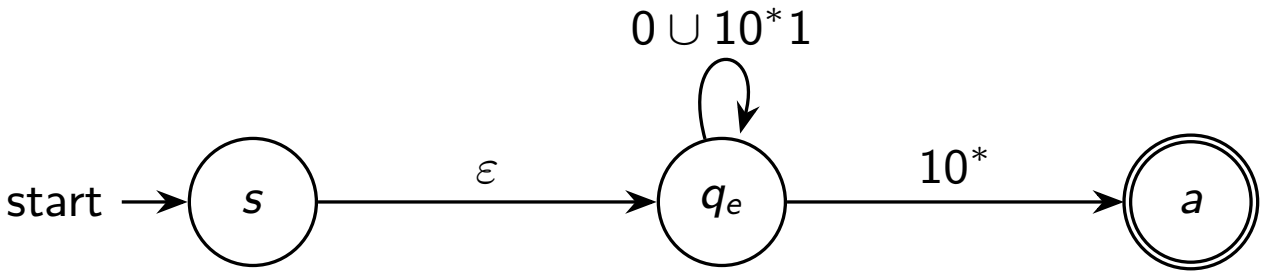
- $q_e \to q_e$: the bypass through q_o contributes 10^*1. Union with the existing 0 to obtain $0 \cup 10^*1$.
- $q_e \to a$: the bypass contributes $10^*\varepsilon = 10^*$. Union with $\emptyset$ to obtain 10^*.



Worked Example: State Elimination

Step 2: Eliminate q_o.
Update the labels on $q_e \to q_e$ and $q_e \to a$.

- $q_e \to q_e$: the bypass through q_o contributes 10^*1. Union with the existing 0 to obtain $0 \cup 10^*1$.
- $q_e \to a$: the bypass contributes $10^*\varepsilon = 10^*$. Union with $\emptyset$ to obtain 10^*.



Step 3: Eliminate q_e.
Now we update the edge $s \to a$.
Path: $s \xrightarrow{\varepsilon} q_e \xrightarrow{(0\cup10^*1)^*} q_e \xrightarrow{10^*} a$.

Final RE: $(0 \cup 10^*1)^*10^*$. Every repeated block contributes zero or two 1's; the final 10^* contributes one. Pairing the first $2k$ ones in any odd-parity word shows completeness. An alternative is $0^*1(0^*10^*1)^*0^*$.

Grammar Notation and Language Generation

We use the Linz-style order $G = (V, T, S, P)$:

- V: a finite set of variables (nonterminals);
- T: a finite terminal alphabet, disjoint from V;
- $S \in V$: the start variable;
- P: finitely many productions $A \to \alpha$, with $A \in V$ and $\alpha \in (V \cup T)^*$, restricted further on the next slide.

Grammar Notation and Language Generation

We use the Linz-style order $G = (V, T, S, P)$:

- V: a finite set of variables (nonterminals);
- T: a finite terminal alphabet, disjoint from V;
- $S \in V$: the start variable;
- P: finitely many productions $A \to \alpha$, with $A \in V$ and $\alpha \in (V \cup T)^*$, restricted further on the next slide.

A step replaces one variable using a production:

$$uAv \Rightarrow u\alpha v \quad \text{if } A \to \alpha \in P.$$

$\Rightarrow^*$ means zero or more steps, and

$$L(G) = \{w \in T^* : S \Rightarrow^* w\}.$$

A terminal-only result is a completed word; aS is not one. $A \to x \mid y$ abbreviates two productions, not a terminal bar.

Part 4: Definitions of Regular Grammars

We have seen that DFAs, NFAs, and REs all describe regular languages. We can also define this class syntactically using restricted grammars.

Part 4: Definitions of Regular Grammars

We have seen that DFAs, NFAs, and REs all describe regular languages. We can also define this class syntactically using restricted grammars.

Regular Grammar Definition

A grammar $G = (V, T, S, P)$ is **regular** if all of its productions are right-linear or all of its productions are left-linear. In both definitions below, $A, B \in V$ and $x \in T^*$.

Right-Linear: All productions are of the form:

- $A \to xB \quad (A, B \in V, x \in T^*)$
- $A \to x$

Left-Linear: All productions are of the form:

- $A \to Bx$
- $A \to x$

Part 4: Definitions of Regular Grammars

We have seen that DFAs, NFAs, and REs all describe regular languages. We can also define this class syntactically using restricted grammars.

Regular Grammar Definition

A grammar $G = (V, T, S, P)$ is **regular** if all of its productions are right-linear or all of its productions are left-linear. In both definitions below, $A, B \in V$ and $x \in T^*$.

Right-Linear: All productions are of the form:

- $A \to xB \quad (A, B \in V,\ x \in T^*)$
- $A \to x$

Left-Linear: All productions are of the form:

- $A \to Bx$
- $A \to x$

Important: At most one variable appears on each right-hand side, and its orientation must be consistent throughout the grammar. Having at most one variable is necessary, but not sufficient, for a grammar to be regular. We allow $x = \varepsilon$, hence unit rules $A \to B$ and terminating rules $A \to \varepsilon$; some texts use a stricter convention.

Right-Linear, Left-Linear, or Merely Linear?

Right-linear example

$$S \to abS \mid a$$

Every variable is at the right end. The generated language is

$$L = (ab)^* a.$$

Left-linear example

$$S \to Aab,$$
$$A \to Aab \mid B,$$
$$B \to a.$$

Every variable is at the left end. Here $L = aab(ab)^*$.

Right-Linear, Left-Linear, or Merely Linear?

Right-linear example

$$S \to abS \mid a$$

Every variable is at the right end. The generated language is

$$L = (ab)^* a.$$

Left-linear example

$$S \to Aab,$$
$$A \to Aab \mid B,$$
$$B \to a.$$

Every variable is at the left end. Here $L = aab(ab)^*$.

Do Not Mix the Two Orientations

The grammar with rules $S \to aA$, $A \to Sb$, and $S \to \varepsilon$ is linear but not a regular grammar: it mixes right- and left-linear productions. In fact, it generates the nonregular language $\{a^n b^n : n \geq 0\}$.

This is a restriction on the *grammar*. A mixed grammar may still generate a regular language; its form alone does not decide that question.

Reading Derivations in a Regular Grammar

Consider the right-linear grammar

$$S \to abS \mid a.$$

Reading Derivations in a Regular Grammar

Consider the right-linear grammar

$$S \to abS \mid a.$$

Each use of $S \to abS$ contributes one block ab and keeps the derivation active. The rule $S \to a$ terminates it.

Sample Derivations

$$S \Rightarrow a,$$
$$S \Rightarrow abS \Rightarrow aba,$$
$$S \Rightarrow abS \Rightarrow ababS \Rightarrow ababa.$$

Reading Derivations in a Regular Grammar

Consider the right-linear grammar

$$S \to abS \mid a.$$

Each use of $S \to abS$ contributes one block ab and keeps the derivation active. The rule $S \to a$ terminates it.

Sample Derivations

$$S \Rightarrow a,$$
$$S \Rightarrow abS \Rightarrow aba,$$
$$S \Rightarrow abS \Rightarrow ababS \Rightarrow ababa.$$

Thus every completed derivation has the form $(ab)^n a$, and every $(ab)^n a$ can be derived. Therefore $L(G) = (ab)^* a$.

Normalizing Long Terminal Blocks

Some definitions permit productions $A \to xB$ and $A \to x$ with $x \in T^*$. Before drawing an NFA, split each long terminal block into one-symbol steps.

Normalizing Long Terminal Blocks

Some definitions permit productions $A \to xB$ and $A \to x$ with $x \in T^*$. Before drawing an NFA, split each long terminal block into one-symbol steps.

General Replacement

For $k \geq 2$, replace

$$A \to a_1 a_2 \cdots a_k B$$

by fresh variables $C_1, \ldots, C_{k-1}$ and rules

$$A \to a_1 C_1, \quad C_1 \to a_2 C_2, \quad \ldots \quad , C_{k-1} \to a_k B.$$

For a terminal-only rule $A \to a_1 \cdots a_k$, use the same chain but end with $C_{k-1} \to a_k$ (or a transition to the new final state in the NFA construction).

Normalizing Long Terminal Blocks

Some definitions permit productions $A \to xB$ and $A \to x$ with $x \in T^*$. Before drawing an NFA, split each long terminal block into one-symbol steps.

General Replacement

For $k \geq 2$, replace

$$A \to a_1 a_2 \cdots a_k B$$

by fresh variables $C_1, \ldots, C_{k-1}$ and rules

$$A \to a_1 C_1, \quad C_1 \to a_2 C_2, \quad \ldots \quad , C_{k-1} \to a_k B.$$

For a terminal-only rule $A \to a_1 \cdots a_k$, use the same chain but end with $C_{k-1} \to a_k$ (or a transition to the new final state in the NFA construction).

Example

With fresh V_2, $V_1 \to abV_0$ becomes $V_1 \to aV_2$ and $V_2 \to bV_0$. The introduced variable records progress through the terminal block.

Theorem: Right-Linear Grammar to NFA

Theorem

If $G = (V, T, S, P)$ is right-linear, then $L(G)$ is a regular language.

Theorem: Right-Linear Grammar to NFA

Theorem

If $G = (V, T, S, P)$ is right-linear, then $L(G)$ is a regular language.

Construction after normalizing terminal blocks:

1. Use one state for each variable and make S the start state.
2. For $A \to aB$, add the transition $A \xrightarrow{a} B$.
3. For $A \to B$, add $A \xrightarrow{\varepsilon} B$.
4. For $A \to a$, add $A \xrightarrow{a} F$ to one new accept state F.
5. For $A \to \varepsilon$, either make A accepting or add $A \xrightarrow{\varepsilon} F$.

Theorem: Right-Linear Grammar to NFA

Theorem

If $G = (V, T, S, P)$ is right-linear, then $L(G)$ is a regular language.

Construction after normalizing terminal blocks:

1. Use one state for each variable and make S the start state.
2. For $A \to aB$, add the transition $A \xrightarrow{a} B$.
3. For $A \to B$, add $A \xrightarrow{\varepsilon} B$.
4. For $A \to a$, add $A \xrightarrow{a} F$ to one new accept state F.
5. For $A \to \varepsilon$, either make A accepting or add $A \xrightarrow{\varepsilon} F$.

Why the Construction Works

In the normalized grammar, $S \Rightarrow^* wA$ exactly when there is a path to variable-state A labeled w (ignoring ε-moves). Reaching an accept state corresponds precisely to completing a derivation $S \Rightarrow^* w$.

Worked Example: Regular Grammar to NFA

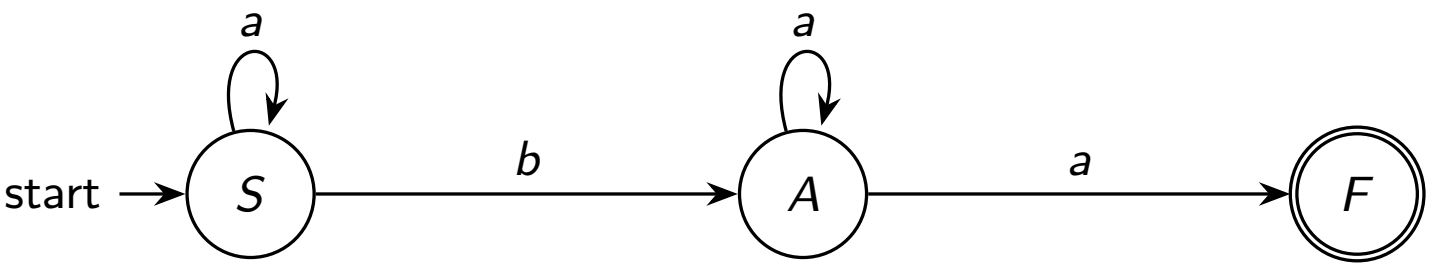
Consider

$$S \to aS \mid bA, \qquad A \to aA \mid a.$$

Worked Example: Regular Grammar to NFA

Consider

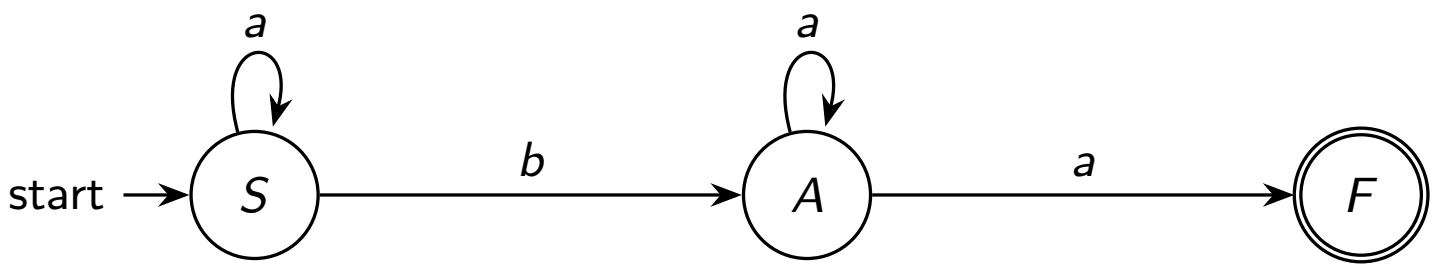
$$S \to aS \mid bA, \qquad A \to aA \mid a.$$



Worked Example: Regular Grammar to NFA

Consider

$$S \to aS \mid bA, \qquad A \to aA \mid a.$$



Derivation

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aabA$$
$$\Rightarrow aabaA \Rightarrow aabaa.$$

Language

$$L(G) = a^* ba^+ = a^* baa^*.$$

Theorem: NFA to Right-Linear Grammar

Theorem

If $L \subseteq \Sigma^*$ is regular, then some right-linear grammar $G = (V, \Sigma, S, P)$ satisfies $L = L(G)$.

Theorem: NFA to Right-Linear Grammar

Theorem

If $L \subseteq \Sigma^*$ is regular, then some right-linear grammar $G = (V, \Sigma, S, P)$ satisfies $L = L(G)$.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA for L.

1. Create one variable Q_i for every automaton state q_i.
2. Use Q_0 as the grammar's start variable.
3. For each transition $q_i \xrightarrow{a} q_j$, add $Q_i \to aQ_j$.
4. For each ε-transition $q_i \xrightarrow{\varepsilon} q_j$, add $Q_i \to Q_j$.
5. For every accepting state q_f, add $Q_f \to \varepsilon$.

Theorem: NFA to Right-Linear Grammar

Theorem

If $L \subseteq \Sigma^*$ is regular, then some right-linear grammar $G = (V, \Sigma, S, P)$ satisfies $L = L(G)$.

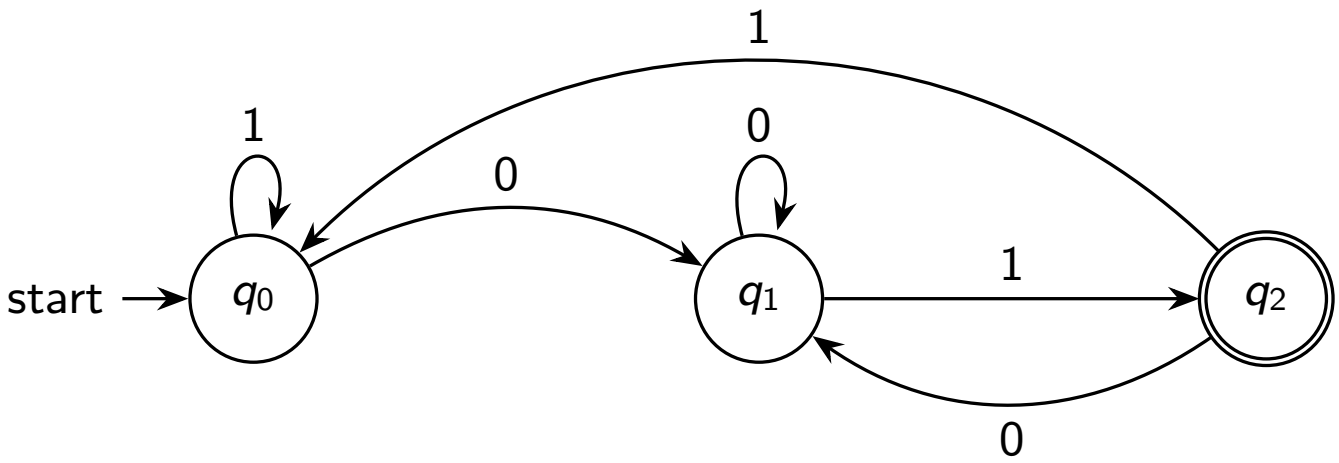
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA for L.

1. Create one variable Q_i for every automaton state q_i.
2. Use Q_0 as the grammar's start variable.
3. For each transition $q_i \xrightarrow{a} q_j$, add $Q_i \to aQ_j$.
4. For each ε-transition $q_i \xrightarrow{\varepsilon} q_j$, add $Q_i \to Q_j$.
5. For every accepting state q_f, add $Q_f \to \varepsilon$.

A path labeled w from q_0 to q_f corresponds step-for-step to a derivation $Q_0 \Rightarrow^* wQ_f \Rightarrow w$.

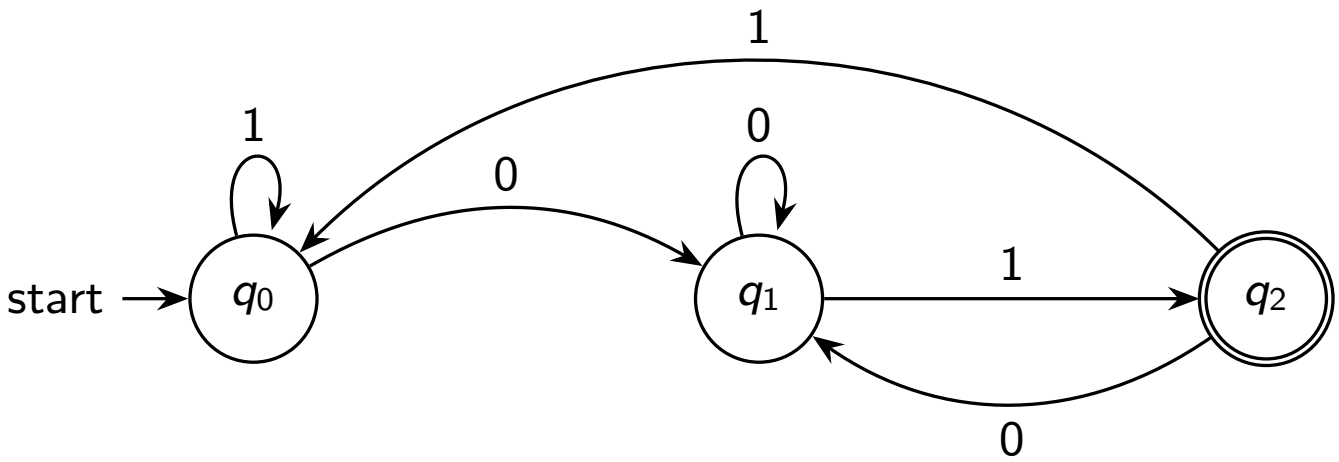
Worked Example: DFA to Right-Linear Grammar

The DFA below accepts exactly the binary strings ending in 01.



Worked Example: DFA to Right-Linear Grammar

The DFA below accepts exactly the binary strings ending in 01.



Replace states by variables Q_0, Q_1, Q_2:

$$Q_0 \to 0Q_1 \mid 1Q_0,$$
$$Q_1 \to 0Q_1 \mid 1Q_2,$$
$$Q_2 \to 0Q_1 \mid 1Q_0 \mid \varepsilon.$$

Left-Linear Grammars and the Equivalence Theorem

Left-Linear Version

A language is regular if and only if it is generated by a left-linear grammar.

Left-Linear Grammars and the Equivalence Theorem

Left-Linear Version

A language is regular if and only if it is generated by a left-linear grammar.

For a finite automaton, build productions in the reverse direction:

- If $q_i \xrightarrow{a} q_j$, add $Q_j \to Q_i a$.
- If $q_i \xrightarrow{\varepsilon} q_j$, add $Q_j \to Q_i$.
- Introduce a new grammar start S with $S \to Q_f$ for each accepting state q_f.
- Add $Q_0 \to \varepsilon$ for the automaton's start state.

Left-Linear Grammars and the Equivalence Theorem

Left-Linear Version

A language is regular if and only if it is generated by a left-linear grammar.

For a finite automaton, build productions in the reverse direction:

- If $q_i \xrightarrow{a} q_j$, add $Q_j \to Q_i a$.
- If $q_i \xrightarrow{\varepsilon} q_j$, add $Q_j \to Q_i$.
- Introduce a new grammar start S with $S \to Q_f$ for each accepting state q_f.
- Add $Q_0 \to \varepsilon$ for the automaton's start state.

Combined Equivalence

For every language L, the following are equivalent:

$$\begin{aligned} L \text{ is regular} &\iff L = L(G_R) \text{ for some right-linear } G_R \\ &\iff L = L(G_L) \text{ for some left-linear } G_L. \end{aligned}$$

Regular-Grammar Checkpoint Exercises

1. Classify each *grammar* as right-linear, left-linear, linear but not a regular grammar, or not linear:
   1. $S \to 0S \mid 1A$, $A \to 1$;
   2. $S \to A0$, $A \to A1 \mid \varepsilon$;
   3. $S \to 0A$, $A \to S1 \mid \varepsilon$.
2. Convert $S \to 0S \mid 1A$, $A \to 0A \mid 1$ into an NFA and describe its language.
3. Construct a right-linear grammar for all binary strings that end in 00.

Regular-Grammar Checkpoint Exercises

1. Classify each *grammar* as right-linear, left-linear, linear but not a regular grammar, or not linear:
   1. $S \to 0S \mid 1A,\ A \to 1$;
   2. $S \to A0,\ A \to A1 \mid \varepsilon$;
   3. $S \to 0A,\ A \to S1 \mid \varepsilon$.
2. Convert $S \to 0S \mid 1A,\ A \to 0A \mid 1$ into an NFA and describe its language.
3. Construct a right-linear grammar for all binary strings that end in 00.

Proof Practice

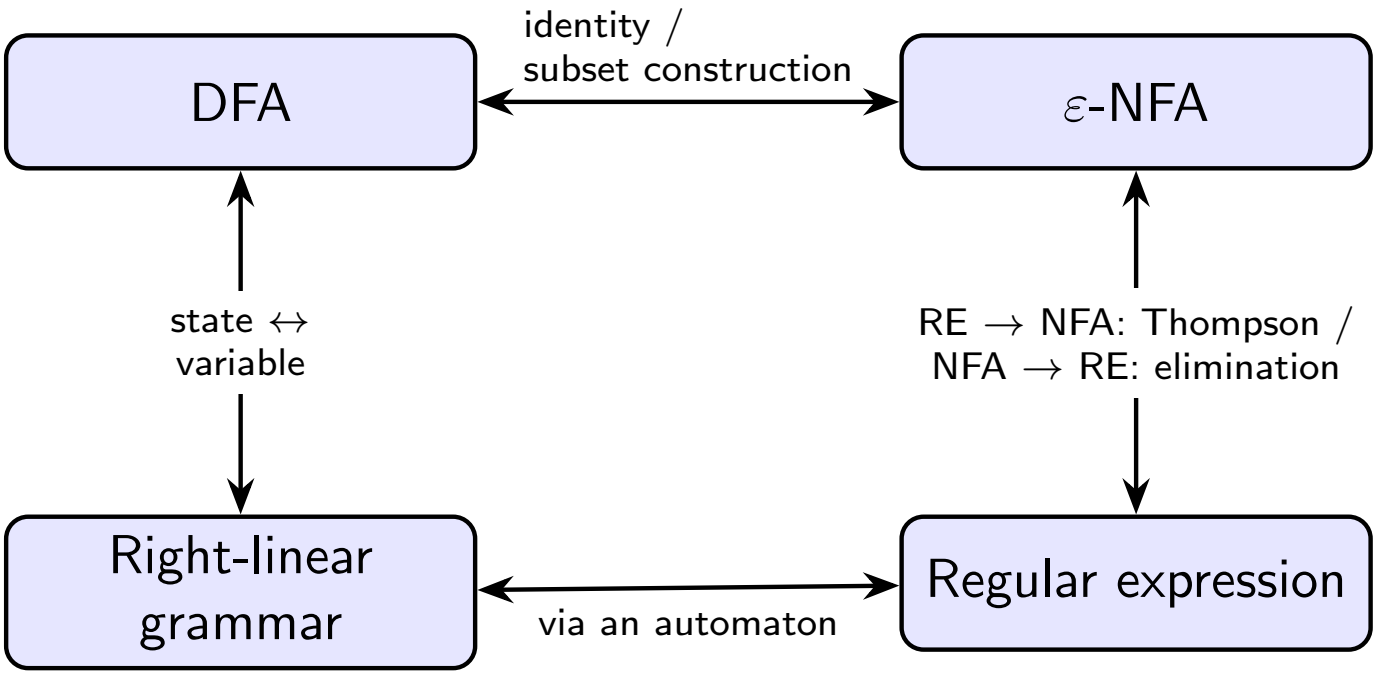
State the invariant that connects a partial derivation $S \Rightarrow^* wA$ with a path in the corresponding NFA.

Part 5: The Equivalence Map

Four formalisms describe exactly the regular languages.

Part 5: The Equivalence Map

Four formalisms describe exactly the regular languages.



Applications of Regular Languages

Why do we care about regular expressions and finite automata?

Applications of Regular Languages

Why do we care about regular expressions and finite automata?

1. Compiler Design (Lexical Analysis)

Compilers use regular expressions to define the tokens of a programming language (e.g., identifiers, constants, and reserved words).
If $D = (0 \cup \cdots \cup 9)$, a simple signed decimal can be described by $(\varepsilon \cup + \cup -)DD^*(\varepsilon \cup .DD^*)$. Tools such as `lex` convert token specifications into efficient finite-automaton-based scanners.

Applications of Regular Languages

Why do we care about regular expressions and finite automata?

1. Compiler Design (Lexical Analysis)

Compilers use regular expressions to define the tokens of a programming language (e.g., identifiers, constants, and reserved words).
If $D = (0 \cup \cdots \cup 9)$, a simple signed decimal can be described by $(\varepsilon \cup + \cup -)DD^*(\varepsilon \cup .DD^*)$. Tools such as `lex` convert token specifications into efficient finite-automaton-based scanners.

2. Pattern Matching and Text Editors

For an RE of syntax-tree size m and an input of length n, Thompson-NFA simulation takes $O(mn)$ time and $O(m)$ working space. For a fixed RE, this is linear in n. A prebuilt DFA needs $O(n)$ matching time, but constructing it can require exponentially many states.

Checkpoint Exercises

1. **GNFA Conversion:** Given a DFA that accepts strings ending in 1, with states q_0 (start) and q_1 (accept), the transitions are $\delta(q_0, 0) = q_0$, $\delta(q_0, 1) = q_1$, $\delta(q_1, 0) = q_0$, and $\delta(q_1, 1) = q_1$.
   Convert this DFA to a regular expression using state elimination.

2. **Grammars:** Write a right-linear grammar for the language:

$$L = \{w \in \{0, 1\}^* \mid w \text{ ends in } 01\}.$$

Practice: Design, Explain, and Find a Counterexample

Work over $\Sigma = \{0, 1\}$ unless stated otherwise.

1. Give an RE for exactly two 1's. Explain why it permits all positions of the 0's but no third 1.
2. Give an RE for words in which every maximal run of 0's has even length. Is that the same as having an even total number of 0's?
3. Disprove $(R \cup S)^* \equiv R^* \cup S^*$ with concrete expressions and a distinguishing word.
4. A student converts $S \to aS \mid \varepsilon$ into an automaton containing only an a-loop at a nonaccepting start state. What is missing?

Practice: Design, Explain, and Find a Counterexample

Work over $\Sigma = \{0, 1\}$ unless stated otherwise.

1. Give an RE for exactly two 1's. Explain why it permits all positions of the 0's but no third 1.
2. Give an RE for words in which every maximal run of 0's has even length. Is that the same as having an even total number of 0's?
3. Disprove $(R \cup S)^* \equiv R^* \cup S^*$ with concrete expressions and a distinguishing word.
4. A student converts $S \to aS \mid \varepsilon$ into an automaton containing only an a-loop at a nonaccepting start state. What is missing?

Discuss before opening the appendix answers

Check ε and the shortest examples first. Then justify every accepted word, not merely a few test cases.

Summary: One Class, Several Useful Representations

- REs describe languages using union, concatenation, and star; an expression is syntax, while $L(R)$ is its meaning.

Summary: One Class, Several Useful Representations

- REs describe languages using union, concatenation, and star; an expression is syntax, while $L(R)$ is its meaning.
- Thompson's construction proves RE $\Rightarrow$ finite automaton by structural induction and one-entry/one-exit fragments.

Summary: One Class, Several Useful Representations

- REs describe languages using union, concatenation, and star; an expression is syntax, while $L(R)$ is its meaning.
- Thompson's construction proves RE $\Rightarrow$ finite automaton by structural induction and one-entry/one-exit fragments.
- State elimination proves the converse by preserving path languages while deleting internal states.

Summary: One Class, Several Useful Representations

- REs describe languages using union, concatenation, and star; an expression is syntax, while $L(R)$ is its meaning.
- Thompson's construction proves RE $\Rightarrow$ finite automaton by structural induction and one-entry/one-exit fragments.
- State elimination proves the converse by preserving path languages while deleting internal states.
- Right-linear grammars follow accepting paths forwards; left-linear grammars can reconstruct the same paths backwards.

Summary: One Class, Several Useful Representations

- REs describe languages using union, concatenation, and star; an expression is syntax, while $L(R)$ is its meaning.
- Thompson's construction proves RE $\Rightarrow$ finite automaton by structural induction and one-entry/one-exit fragments.
- State elimination proves the converse by preserving path languages while deleting internal states.
- Right-linear grammars follow accepting paths forwards; left-linear grammars can reconstruct the same paths backwards.
- Equal expressive power does not imply equal representation size, and a syntactically nonregular grammar may still generate a regular language.

Bridge to Slide Set 4

These constructions become tools for proving closure, deciding questions about regular languages, and identifying the limits of finite memory.

Appendix Guide: Choose a Route

The main lecture ends at the summary. This appendix keeps optional depth available without interrupting the core conversion methods.

Moved out of the main lecture

Empty-string tests; detailed Thompson and state-elimination proofs; the left-linear converse; programming-regex caveats; extra conversion practice; and checkpoint answers.

Suggested enrichment routes

- **Exam preparation:** practice solutions and worked challenges.
- **Algebra:** RE identities, Arden's lemma, and grammar equations.
- **Implementation:** derivatives and position automata.

Problems with textbook attributions are adapted and solved here; the additional extensions are original practice. Closure, decision properties, and the pumping lemma are left to Slide Set 4.

Shorthand, Membership, and the Empty String

Abbreviations do not add expressive power:

$$R^+ = RR^*, \qquad R? = (R \cup \varepsilon), \qquad R^0 = \varepsilon, \quad R^{k+1} = R^k R.$$

Here the superscript $+$ means one or more repetitions, not union. Each occurrence of R may contribute a different word of $L(R)$.

Shorthand, Membership, and the Empty String

Abbreviations do not add expressive power:

$$R^+ = RR^*, \qquad R? = (R \cup \varepsilon), \qquad R^0 = \varepsilon, \quad R^{k+1} = R^k R.$$

Here the superscript $+$ means one or more repetitions, not union. Each occurrence of R may contribute a different word of $L(R)$.

Does an expression accept the empty string?

Write $\nu(R)$ for the truth value of $\varepsilon \in L(R)$. Then

$$\begin{aligned} \nu(a) = \nu(\emptyset) &= \text{false}, \qquad \nu(\varepsilon) = \text{true}, \\ \nu(R \cup S) = \nu(R) \vee \nu(S), &\quad \nu(RS) = \nu(R) \wedge \nu(S), \\ \nu(R^*) &= \text{true}. \end{aligned}$$

Shorthand, Membership, and the Empty String

Abbreviations do not add expressive power:

$$R^+ = RR^*, \qquad R? = (R \cup \varepsilon), \qquad R^0 = \varepsilon, \quad R^{k+1} = R^k R.$$

Here the superscript $+$ means one or more repetitions, not union. Each occurrence of R may contribute a different word of $L(R)$.

Does an expression accept the empty string?

Write $\nu(R)$ for the truth value of $\varepsilon \in L(R)$. Then

$$\begin{aligned} \nu(a) = \nu(\emptyset) &= \text{false}, & \nu(\varepsilon) &= \text{true}, \\ \nu(R \cup S) &= \nu(R) \vee \nu(S), & \nu(RS) &= \nu(R) \wedge \nu(S), \\ \nu(R^*) &= \text{true}. \end{aligned}$$

Thus R^+ can contain ε if R does. In this course, membership matches the *whole word*; a substring search for R instead asks for membership in $\Sigma^* L(R) \Sigma^*$.

Why Thompson's Construction Works

Structural-induction invariant

The fragment N_R has distinct entry and exit states, no edge entering its entry or leaving its exit, and accepts exactly $L(R)$. These boundary conditions hold before the fragment is embedded.

Why Thompson's Construction Works

Structural-induction invariant

The fragment N_R has distinct entry and exit states, no edge entering its entry or leaving its exit, and accepts exactly $L(R)$. These boundary conditions hold before the fragment is embedded.

- **Atoms:** the three primitive diagrams have the required languages $\{a\}$, $\{\varepsilon\}$, and $\emptyset$.
- **Union:** every successful path uses one selected branch.
- **Concatenation:** every successful path splits into a path through N_{R_1} followed by a path through N_{R_2}.
- **Star:** every successful path traverses the body zero or more times, yielding a finite concatenation of words of $L(R)$.

Why Thompson's Construction Works

Structural-induction invariant

The fragment N_R has distinct entry and exit states, no edge entering its entry or leaving its exit, and accepts exactly $L(R)$. These boundary conditions hold before the fragment is embedded.

- **Atoms:** the three primitive diagrams have the required languages $\{a\}$, $\{\varepsilon\}$, and $\emptyset$.
- **Union:** every successful path uses one selected branch.
- **Concatenation:** every successful path splits into a path through N_{R_1} followed by a path through N_{R_2}.
- **Star:** every successful path traverses the body zero or more times, yielding a finite concatenation of words of $L(R)$.

Conversely, each such choice of words gives a successful path. Thus the induction proves equality, not just inclusion. For an RE syntax tree of size m, the construction uses $O(m)$ states and transitions; it need not give the smallest NFA or DFA.

Why State Elimination Preserves the Language

Path invariant

Each label summarizes paths of one or more edges between its endpoints whose internal states have already been eliminated. Initially, labels summarize the original edges; after eliminating q_r, paths may also pass through q_r.

Why State Elimination Preserves the Language

Path invariant

Each label summarizes paths of one or more edges between its endpoints whose internal states have already been eliminated. Initially, labels summarize the original edges; after eliminating q_r, paths may also pass through q_r.

A path between remaining states either avoids q_r, contributing R_{ij}, or enters it, loops there any finite number of times, and leaves:

$$R'_{ij} = R_{ij} \cup R_{ir}(R_{rr})^* R_{rj}.$$

Conversely, every word in this new label expands into an old path.

Why State Elimination Preserves the Language

Path invariant

Each label summarizes paths of one or more edges between its endpoints whose internal states have already been eliminated. Initially, labels summarize the original edges; after eliminating q_r, paths may also pass through q_r.

A path between remaining states either avoids q_r, contributing R_{ij}, or enters it, loops there any finite number of times, and leaves:

$$R'_{ij} = R_{ij} \cup R_{ir}(R_{rr})^* R_{rj}.$$

Conversely, every word in this new label expands into an old path.

Important edge cases

If q_r has no loop, $R_{rr} = \emptyset$ but $(R_{rr})^* = \varepsilon$: passing through once is still possible. If no accepting path exists, the final label is $\emptyset$. If the old start accepts, the new s-to-f path preserves acceptance of ε.

When only s and f remain, their label denotes the original language.

Left-Linear Conversion: Direction Matters

For the automaton with start q_0, loop $q_0 \xrightarrow{a} q_0$, and transition $q_0 \xrightarrow{b} q_1$ to its sole accepting state, use

$$S \to Q_1, \qquad Q_1 \to Q_0 b, \qquad Q_0 \to Q_0 a \mid \varepsilon.$$

For example,

$$S \Rightarrow Q_1 \Rightarrow Q_0 b \Rightarrow Q_0 ab \Rightarrow Q_0 aab \Rightarrow aab.$$

We reconstruct the path backwards, but the terminal word is still in its original order. Both descriptions give $a^* b$.

Left-Linear Conversion: Direction Matters

For the automaton with start q_0, loop $q_0 \xrightarrow{a} q_0$, and transition $q_0 \xrightarrow{b} q_1$ to its sole accepting state, use

$$S \to Q_1, \qquad Q_1 \to Q_0 b, \qquad Q_0 \to Q_0 a \mid \varepsilon.$$

For example,

$$S \Rightarrow Q_1 \Rightarrow Q_0 b \Rightarrow Q_0 ab \Rightarrow Q_0 aab \Rightarrow aab.$$

We reconstruct the path backwards, but the terminal word is still in its original order. Both descriptions give $a^* b$.

Converse: why every left-linear grammar is regular

Reverse every production's right side: $A \to Bx$ becomes $A \to x^R B$, and $A \to x$ becomes $A \to x^R$. The resulting right-linear grammar generates $L(G)^R$, which is regular. Reversing an NFA's edges and exchanging its initial/final boundaries recognizes the reversal; thus $L(G)$ is regular too.

Classical REs Versus Programming Regex

Notation is not computational power

Character classes, optional pieces, and bounded repetitions are shorthand for finite unions and concatenations. They remain regular. Backtracking is an implementation strategy, not a new language operation: it can be slow even for a classical regular expression.

Classical REs Versus Programming Regex

Notation is not computational power

Character classes, optional pieces, and bounded repetitions are shorthand for finite unions and concatenations. They remain regular. Backtracking is an implementation strategy, not a new language operation: it can be slow even for a classical regular expression.

Backreferences really can go beyond regularity

Under whole-string matching, the programming pattern `([01]*)\1` requires two identical binary halves. It denotes $\{ww : w \in \{0,1\}^*\}$, which is not regular. The symbol `\1` is not an operator in our RE definition.

Classical REs Versus Programming Regex

Notation is not computational power

Character classes, optional pieces, and bounded repetitions are shorthand for finite unions and concatenations. They remain regular. Backtracking is an implementation strategy, not a new language operation: it can be slow even for a classical regular expression.

Backreferences really can go beyond regularity

Under whole-string matching, the programming pattern `([01]*)\1` requires two identical binary halves. It denotes $\{ww : w \in \{0,1\}^*\}$, which is not regular. The symbol `\1` is not an operator in our RE definition.

Do not classify every lookaround as nonregular: assertions built only from regular subpatterns can express regular constraints. Separate the *language described* from the *engine's running time*.

Hints: Regular-Grammar Checkpoint

Hints

- Exercise 1(a): right-linear, language 0^*11. Exercise 1(b): left-linear, language 1^*0. Exercise 1(c): mixed linear, language $\{0^{n+1}1^n : n \geq 0\}$, since $S \Rightarrow 0S1$ or $S \Rightarrow 0$.
- For Exercise 2, use states S, A, and a new final state F. The resulting regular expression is 0^*10^*1.
- Exercise 3: $S \to 0S \mid 1S \mid 0A$, $A \to 0B$, $B \to \varepsilon$. Guess the beginning of the final 00.

Hints: Regular-Grammar Checkpoint

Hints

- Exercise 1(a): right-linear, language 0^*11. Exercise 1(b): left-linear, language 1^*0. Exercise 1(c): mixed linear, language $\{0^{n+1}1^n : n \geq 0\}$, since $S \Rightarrow 0S1$ or $S \Rightarrow 0$.
- For Exercise 2, use states S, A, and a new final state F. The resulting regular expression is 0^*10^*1.
- Exercise 3: $S \to 0S \mid 1S \mid 0A$, $A \to 0B$, $B \to \varepsilon$. Guess the beginning of the final 00.

Invariant for the Proof

For every $w \in T^*$ and $A \in V$,

$$S \Rightarrow^* wA \quad \Longleftrightarrow \quad \text{the normalized construction has a path } S \xrightarrow{w} A.$$

Hints for Checkpoint Exercises

Hints

- **Exercise 1:**
  - Add the standard GNFA start (s) and accept (a) states.
  - When ripping q_1, the path $q_0 \to q_1 \to q_0$ creates a loop on q_0 labeled 11^*0.
  - The total loop on q_0 becomes $0 \cup 11^*0$.
  - The path to a is via $q_1 \to a$ which is ε, so $q_0 \to q_1 \to a$ gives 11^*.
  - Eliminating q_0 next gives $(0 \cup 11^*0)^*11^*$, which is equivalent to $(0 \cup 1)^*1$.
- **Exercise 2:** One answer is $S \to 0S \mid 1S \mid 0A$, $A \to 1B$, and $B \to \varepsilon$. The nondeterministic choice on 0 guesses the start of the final suffix 01.

Hints for Checkpoint Exercises

Hints

- **Exercise 1:**
  - Add the standard GNFA start (s) and accept (a) states.
  - When ripping q_1, the path $q_0 \to q_1 \to q_0$ creates a loop on q_0 labeled 11^*0.
  - The total loop on q_0 becomes $0 \cup 11^*0$.
  - The path to a is via $q_1 \to a$ which is ε, so $q_0 \to q_1 \to a$ gives 11^*.
  - Eliminating q_0 next gives $(0 \cup 11^*0)^*11^*$, which is equivalent to $(0 \cup 1)^*1$.
- **Exercise 2:** One answer is $S \to 0S \mid 1S \mid 0A$, $A \to 1B$, and $B \to \varepsilon$. The nondeterministic choice on 0 guesses the start of the final suffix 01.

Final Pro-Tip for State Elimination

Simplify regular-expression labels after each state-elimination update. For instance, replace εR by R and $R \cup \emptyset$ by R. This keeps the final expression readable and easier to verify.

Solutions: Design and Counterexamples

1. Exactly two 1's: $0^*10^*10^*$. The two literal 1's are separated and surrounded by arbitrary 0-blocks.

Solutions: Design and Counterexamples

1. Exactly two 1's: $0^*10^*10^*$. The two literal 1's are separated and surrounded by arbitrary 0-blocks.
2. Even 0-runs: $(1 \cup 00)^*$. Every run is a concatenation of 00 blocks, and any even run can be divided into such blocks. The word 010 has even total zero count but two odd runs, so the conditions differ.

Solutions: Design and Counterexamples

1. Exactly two 1's: $0^*10^*10^*$. The two literal 1's are separated and surrounded by arbitrary 0-blocks.
2. Even 0-runs: $(1 \cup 00)^*$. Every run is a concatenation of 00 blocks, and any even run can be divided into such blocks. The word 010 has even total zero count but two odd runs, so the conditions differ.
3. Let $R = 0$, $S = 1$. The word 01 is in $L((0 \cup 1)^*)$ but not in $L(0^* \cup 1^*)$. Star of a union allows mixed factors.

Solutions: Design and Counterexamples

1. Exactly two 1's: $0^*10^*10^*$. The two literal 1's are separated and surrounded by arbitrary 0-blocks.
2. Even 0-runs: $(1 \cup 00)^*$. Every run is a concatenation of 00 blocks, and any even run can be divided into such blocks. The word 010 has even total zero count but two odd runs, so the conditions differ.
3. Let $R = 0$, $S = 1$. The word 01 is in $L((0 \cup 1)^*)$ but not in $L(0^* \cup 1^*)$. Star of a union allows mixed factors.
4. Make the start state accepting, or add an ε-edge to a fresh accepting state. The omitted rule $S \to \varepsilon$ supplies termination, including acceptance of the empty word.

Practice: Conversion and Grammar Structure

1. Over $\{a, b, c\}$, classify $S \to aA$, $A \to Bb$, $B \to c$. Is its generated language regular?
2. Normalize $S \to abS \mid \varepsilon$ and draw an equivalent NFA. Explain why the variable introduced for ab must be fresh.
3. A GNFA has $s \xrightarrow{a} r$, $r \xrightarrow{b} f$, and $s \xrightarrow{c} f$, with no loop on r. Eliminate r.
4. Construct both a right-linear and a left-linear grammar for a^*b. Give one derivation of aab in each.

Practice: Conversion and Grammar Structure

1. Over $\{a, b, c\}$, classify $S \to aA$, $A \to Bb$, $B \to c$. Is its generated language regular?
2. Normalize $S \to abS \mid \varepsilon$ and draw an equivalent NFA. Explain why the variable introduced for ab must be fresh.
3. A GNFA has $s \xrightarrow{a} r$, $r \xrightarrow{b} f$, and $s \xrightarrow{c} f$, with no loop on r. Eliminate r.
4. Construct both a right-linear and a left-linear grammar for a^*b. Give one derivation of aab in each.

What a complete conversion answer includes

Identify the start and accepting states or start variable, list all rules or transitions, and state the path/derivation invariant. A correct-looking diagram alone does not explain correctness.

Solutions: Conversion and Grammar Structure

1. The grammar is linear but mixes orientations. Its only terminal derivation is $S \Rightarrow aA \Rightarrow aBb \Rightarrow acb$. The singleton language $\{acb\}$ is regular.

Solutions: Conversion and Grammar Structure

1. The grammar is linear but mixes orientations. Its only terminal derivation is $S \Rightarrow aA \Rightarrow aBb \Rightarrow acb$. The singleton language $\{acb\}$ is regular.
2. Use $S \to aC \mid \varepsilon$, $C \to bS$, with fresh C. The NFA has accepting start S, an a-edge to C, and a b-edge back to S. Reusing an existing variable could introduce unintended choices midway through the block.

Solutions: Conversion and Grammar Structure

1. The grammar is linear but mixes orientations. Its only terminal derivation is $S \Rightarrow aA \Rightarrow aBb \Rightarrow acb$. The singleton language $\{acb\}$ is regular.
2. Use $S \to aC \mid \varepsilon$, $C \to bS$, with fresh C. The NFA has accepting start S, an a-edge to C, and a b-edge back to S. Reusing an existing variable could introduce unintended choices midway through the block.
3. The new label is $c \cup a\emptyset^* b \equiv c \cup ab$. No loop means zero repetitions, not the absence of a bypass path.

Solutions: Conversion and Grammar Structure

1. The grammar is linear but mixes orientations. Its only terminal derivation is $S \Rightarrow aA \Rightarrow aBb \Rightarrow acb$. The singleton language $\{acb\}$ is regular.
2. Use $S \to aC \mid \varepsilon$, $C \to bS$, with fresh C. The NFA has accepting start S, an a-edge to C, and a b-edge back to S. Reusing an existing variable could introduce unintended choices midway through the block.
3. The new label is $c \cup a\emptyset^* b \equiv c \cup ab$. No loop means zero repetitions, not the absence of a bypass path.
4. Right-linear: $S \to aS \mid b$, with $S \Rightarrow aS \Rightarrow aaS \Rightarrow aab$. Left-linear: $S \to Ab$, $A \to Aa \mid \varepsilon$, with $S \Rightarrow Ab \Rightarrow Aab \Rightarrow Aaab \Rightarrow aab$.

More RE Laws: Useful but Easy to Misapply

In addition to associativity and commutativity of union, and associativity of concatenation, the following are valid language equivalences:

$$R^* \equiv \varepsilon \cup RR^* \equiv \varepsilon \cup R^*R,$$
$$(R^*)^* \equiv R^*, \qquad R^*R^* \equiv R^*,$$
$$(R \cup S)^* \equiv R^*(SR^*)^*.$$

More RE Laws: Useful but Easy to Misapply

In addition to associativity and commutativity of union, and associativity of concatenation, the following are valid language equivalences:

$$R^* \equiv \varepsilon \cup RR^* \equiv \varepsilon \cup R^*R,$$
$$(R^*)^* \equiv R^*, \qquad R^*R^* \equiv R^*,$$
$$(R \cup S)^* \equiv R^*(SR^*)^*.$$

Proof idea for the last identity: group any sequence of R- and S-factors into an initial run of R-factors followed by zero or more groups consisting of one S-factor and a run of R-factors. Conversely, such groups are sequences of R- and S-factors.

More RE Laws: Useful but Easy to Misapply

In addition to associativity and commutativity of union, and associativity of concatenation, the following are valid language equivalences:

$$R^* \equiv \varepsilon \cup RR^* \equiv \varepsilon \cup R^*R,$$
$$(R^*)^* \equiv R^*, \qquad R^*R^* \equiv R^*,$$
$$(R \cup S)^* \equiv R^*(SR^*)^*.$$

Proof idea for the last identity: group any sequence of R- and S-factors into an initial run of R-factors followed by zero or more groups consisting of one S-factor and a run of R-factors. Conversely, such groups are sequences of R- and S-factors.

Do not distribute star over union or concatenation

$(R \cup S)^* \not\equiv R^* \cup S^*$ in general, and $(RS)^* \not\equiv R^*S^*$ in general. For $R = a$, $S = b$, the word a distinguishes $(ab)^*$ from a^*b^*.

Arden's Lemma: An Algebraic Alternative

Here A, B, X denote *languages*, not grammar variables.

Arden's Lemma

$X = AX \cup B$ always has the least solution $X = A^*B$. If $\varepsilon \notin A$, that solution is unique.

Arden's Lemma: An Algebraic Alternative

Here A, B, X denote *languages*, not grammar variables.

Arden's Lemma

$X = AX \cup B$ always has the least solution $X = A^*B$. If $\varepsilon \notin A$, that solution is unique.

Proof sketch. Substitution verifies that A^*B is a solution. Any solution contains $B, AB, A^2B, \ldots$, hence contains A^*B. If $\varepsilon \notin A$, each recursive factor from A consumes at least one symbol, so repeatedly decomposing a finite word of X must reach B. Thus $X \subseteq A^*B$ too.

Arden's Lemma: An Algebraic Alternative

Here A, B, X denote *languages*, not grammar variables.

Arden's Lemma

$X = AX \cup B$ always has the least solution $X = A^*B$. If $\varepsilon \notin A$, that solution is unique.

Proof sketch. Substitution verifies that A^*B is a solution. Any solution contains $B, AB, A^2B, \ldots$, hence contains A^*B. If $\varepsilon \notin A$, each recursive factor from A consumes at least one symbol, so repeatedly decomposing a finite word of X must reach B. Thus $X \subseteq A^*B$ too.

Order and the hypothesis both matter

For $X = XA \cup B$, the least solution is BA^*. With $A = \{\varepsilon\}$ and $B = \emptyset$, the equation becomes $X = X$: every language is a solution, so uniqueness fails.

Worked Language Equations: Odd Number of Ones

Let X_e and X_o be the languages accepted starting from the even and odd states of the earlier parity DFA. Decompose by the first transition:

$$X_e = 0X_e \cup 1X_o, \qquad X_o = \varepsilon \cup 0X_o \cup 1X_e.$$

Only the accepting state contributes ε.

Worked Language Equations: Odd Number of Ones

Let X_e and X_o be the languages accepted starting from the even and odd states of the earlier parity DFA. Decompose by the first transition:

$$X_e = 0X_e \cup 1X_o, \qquad X_o = \varepsilon \cup 0X_o \cup 1X_e.$$

Only the accepting state contributes ε. Applying Arden's lemma to the second equation gives

$$X_o = 0^*(\varepsilon \cup 1X_e).$$

Substitute into the first equation and distribute:

$$X_e = (0 \cup 10^*1)X_e \cup 10^*.$$

Worked Language Equations: Odd Number of Ones

Let X_e and X_o be the languages accepted starting from the even and odd states of the earlier parity DFA. Decompose by the first transition:

$$X_e = 0X_e \cup 1X_o, \qquad X_o = \varepsilon \cup 0X_o \cup 1X_e.$$

Only the accepting state contributes ε. Applying Arden's lemma to the second equation gives

$$X_o = 0^*(\varepsilon \cup 1X_e).$$

Substitute into the first equation and distribute:

$$X_e = (0 \cup 10^*1)X_e \cup 10^*.$$

The coefficient language excludes ε, so Arden gives

$$X_e = (0 \cup 10^*1)^*10^*,$$

exactly the expression obtained by eliminating the odd state first.

Challenge: Exactly One Occurrence of 00

Task. Design a binary RE with exactly one occurrence of 00, counting overlapping occurrences. Thus 000 has two and must be rejected.

Challenge: Exactly One Occurrence of 00

Task. Design a binary RE with exactly one occurrence of 00, counting overlapping occurrences. Thus 000 has two and must be rejected.

Solution

$$(1 \cup 01)^* \, 00 \, (1 \cup 10)^*.$$

Soundness. The prefix has no 00 and is empty or ends in 1. The suffix has no 00 and is empty or starts in 1. Therefore the displayed middle pair is the only occurrence, including at the joins.

Challenge: Exactly One Occurrence of 00

Task. Design a binary RE with exactly one occurrence of 00, counting overlapping occurrences. Thus 000 has two and must be rejected.

Solution

$$(1 \cup 01)^* \, 00 \, (1 \cup 10)^*.$$

Soundness. The prefix has no 00 and is empty or ends in 1. The suffix has no 00 and is empty or starts in 1. Therefore the displayed middle pair is the only occurrence, including at the joins. **Completeness.** Split a valid word around its unique 00. Its prefix and suffix must have exactly those boundary restrictions; factor them into 1/01 blocks and 1/10 blocks, respectively.

The tempting wrong answer

$(0 \cup 1)^* 00 (0 \cup 1)^*$ guarantees *at least* one occurrence, not exactly one. Test 00, 10010, 000, and 00100.

Original worked exercise in the RE-design style of Linz, Chapter 3, and Sipser, Section 1.3.

Challenge: Binary Numbers at Least 40

Task. Describe binary representations of integers at least 40. Allow leading zeros; the empty word is not a representation here. Let $B = (0 \cup 1)$, and let B^k mean k concatenated copies.

Challenge: Binary Numbers at Least 40

Task. Describe binary representations of integers at least 40. Allow leading zeros; the empty word is not a representation here. Let $B = (0 \cup 1)$, and let B^k mean k concatenated copies.

Split by significant length

$$0^*\left(1B^6B^* \;\cup\; 1(01 \cup 10 \cup 11)B^3\right).$$

- At least seven significant bits: the value is at least 64. These words are $1B^6B^*$.

Challenge: Binary Numbers at Least 40

Task. Describe binary representations of integers at least 40. Allow leading zeros; the empty word is not a representation here. Let $B = (0 \cup 1)$, and let B^k mean k concatenated copies.

Split by significant length

$$0^*\left(1B^6B^* \;\cup\; 1(01 \cup 10 \cup 11)B^3\right).$$

- At least seven significant bits: the value is at least 64. These words are $1B^6B^*$.
- Six significant bits: $40 = 101000_2$. The admissible first three bits are 101, 110, or 111; the last three bits are arbitrary.

Challenge: Binary Numbers at Least 40

Task. Describe binary representations of integers at least 40. Allow leading zeros; the empty word is not a representation here. Let $B = (0 \cup 1)$, and let B^k mean k concatenated copies.

Split by significant length

$$0^*\left(1B^6B^* \;\cup\; 1(01 \cup 10 \cup 11)B^3\right).$$

- At least seven significant bits: the value is at least 64. These words are $1B^6B^*$.
- Six significant bits: $40 = 101000_2$. The admissible first three bits are 101, 110, or 111; the last three bits are arbitrary.
- At most five significant bits: the value is at most 31. These words are excluded. An initial 0^* does not change the value.

Boundary checks: reject 100111 (39), accept 101000 (40) and 00101000, and accept 1000000 (64).

Textbook-style numerical RE challenge; the case analysis and leading-zero convention are made explicit here.

Sipser Challenge: Constrained Positions

Task. Over $\{0, 1\}$, require a 1 at every odd-numbered position, counting from position 1. The empty word satisfies the condition.

Sipser Challenge: Constrained Positions

Task. Over $\{0, 1\}$, require a 1 at every odd-numbered position, counting from position 1. The empty word satisfies the condition. Group positions into pairs, with a possible final odd position:

$$\big(1(0 \cup 1)\big)^*(\varepsilon \cup 1).$$

Every pair starts with 1; conversely, every valid word has this pairing.

Sipser Challenge: Constrained Positions

Task. Over $\{0, 1\}$, require a 1 at every odd-numbered position, counting from position 1. The empty word satisfies the condition. Group positions into pairs, with a possible final odd position:

$$\big(1(0 \cup 1)\big)^*(\varepsilon \cup 1).$$

Every pair starts with 1; conversely, every valid word has this pairing.

Harder extension: also require an even number of zeros

Write $U = 11$ and $V = 10$ for the two pair types. Only a V contributes a zero, so pair the V-blocks:

$$U^*(VU^*VU^*)^*(\varepsilon \cup 1).$$

The optional final 1 does not change the zero count.

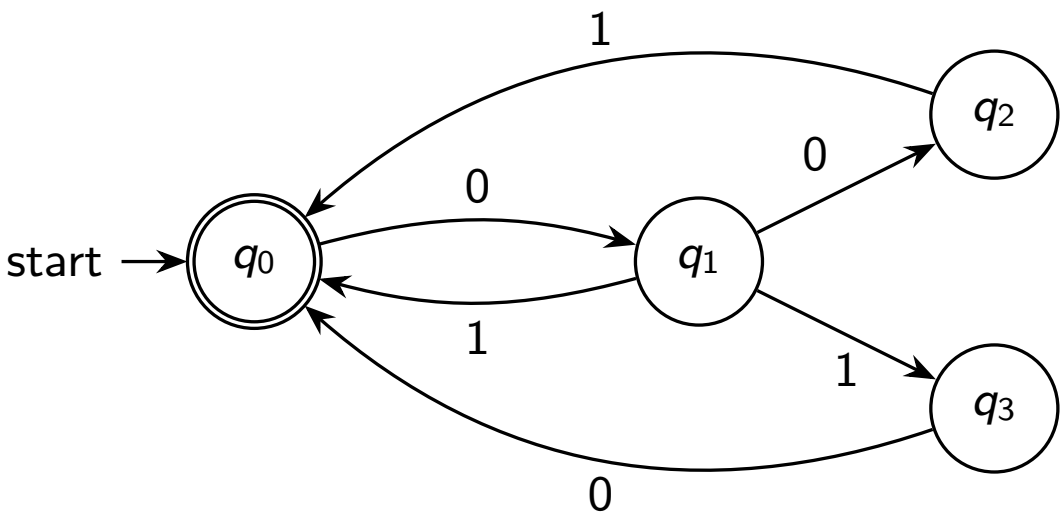
Base problem adapted from Sipser's Exercise 1.18, as identified in the course assignment [PSU13]; the even-zero strengthening and solutions are original.

Sipser Challenge: RE to a Compact NFA

Task. Build an NFA, then a DFA, for $R = (01 \cup 001 \cup 010)^*$.

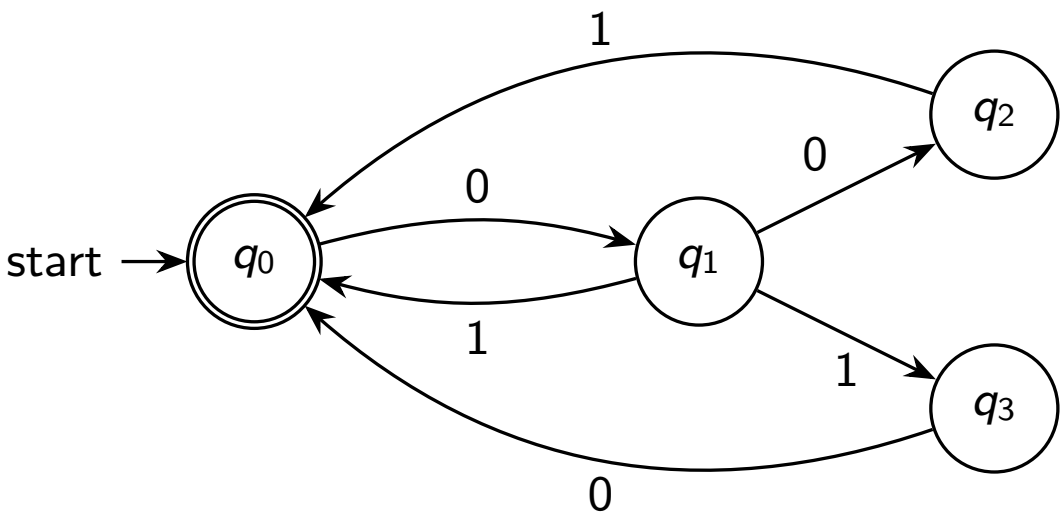
Sipser Challenge: RE to a Compact NFA

Task. Build an NFA, then a DFA, for $R = (01 \cup 001 \cup 010)^*$. Use q_0 as both start and sole accepting state. Each return to q_0 completes one factor. Shared prefixes give this compact NFA:



Sipser Challenge: RE to a Compact NFA

Task. Build an NFA, then a DFA, for $R = (01 \cup 001 \cup 010)^*$. Use q_0 as both start and sole accepting state. Each return to q_0 completes one factor. Shared prefixes give this compact NFA:



From q_1, input 1 may finish 01 or continue toward 010. This nondeterminism is essential to this particular prefix-sharing design. The three first-return paths spell exactly 01, 001, and 010.

Adapted from Sipser's Exercise 1.17, identified in [PSU13]. This is a compact NFA, not the literal Thompson output.

Solution: Reachable Subsets and a Complete DFA

Use the NFA on the preceding slide. For a subset Q', take the union of all possible next states. A subset accepts exactly when it contains q_0.

Solution: Reachable Subsets and a Complete DFA

Use the NFA on the preceding slide. For a subset Q', take the union of all possible next states. A subset accepts exactly when it contains q_0.

DFA state	NFA subset	on 0	on 1	Accept?
A (start)	$\{q_0\}$	B	$\bot$	yes
B	$\{q_1\}$	C	D	no
C	$\{q_2\}$	$\bot$	A	no
D	$\{q_0, q_3\}$	E	$\bot$	yes
E	$\{q_0, q_1\}$	F	D	yes
F	$\{q_1, q_2\}$	C	D	no
$\bot$	$\emptyset$	$\bot$	$\bot$	no

Solution: Reachable Subsets and a Complete DFA

Use the NFA on the preceding slide. For a subset Q', take the union of all possible next states. A subset accepts exactly when it contains q_0.

DFA state	NFA subset	on 0	on 1	Accept?
A (start)	$\{q_0\}$	B	$\perp$	yes
B	$\{q_1\}$	C	D	no
C	$\{q_2\}$	$\perp$	A	no
D	$\{q_0, q_3\}$	E	$\perp$	yes
E	$\{q_0, q_1\}$	F	D	yes
F	$\{q_1, q_2\}$	C	D	no
$\perp$	$\emptyset$	$\perp$	$\perp$	no

All seven subsets are reachable, for example by $\varepsilon, 0, 00, 01, 010, 0100, 1$, respectively.

Check the boundary ambiguity

After 01, both q_0 and q_3 are possible. On the next 0, one computation starts a new factor and another finishes 010: $D \xrightarrow{0} E$. Keeping only one computation would lose words.

Reachable does not mean minimal; minimization is a separate task.

Challenge: Change the Elimination Order

Return to the odd-number-of-1's DFA, with even state q_e (start), odd state q_o (accept), 0-loops, and 1-transitions between them. **Now eliminate q_e before q_o.**

Challenge: Change the Elimination Order

Return to the odd-number-of-1's DFA, with even state q_e (start), odd state q_o (accept), 0-loops, and 1-transitions between them. **Now eliminate q_e before q_o.** After adding GNFA boundaries s, f, the relevant surviving labels are

$$R_{s,o} = 0^*1, \qquad R_{o,o} = 0 \cup 10^*1, \qquad R_{o,f} = \varepsilon.$$

Eliminating q_o next gives

$$0^*1(0 \cup 10^*1)^*.$$

Challenge: Change the Elimination Order

Return to the odd-number-of-1's DFA, with even state q_e (start), odd state q_o (accept), 0-loops, and 1-transitions between them. **Now eliminate q_e before q_o.** After adding GNFA boundaries s, f, the relevant surviving labels are

$$R_{s,o} = 0^*1, \qquad R_{o,o} = 0 \cup 10^*1, \qquad R_{o,f} = \varepsilon.$$

Eliminating q_o next gives

$$0^*1(0 \cup 10^*1)^*.$$

Why it agrees with the main lecture

The main order gave $(0 \cup 10^*1)^*10^*$. In both expressions there is one unpaired 1 and zero or more pairs of 1's, with arbitrary zeros. Any word with an odd number of 1's admits either grouping.

This is a language-equivalence argument, *not* a claim that concatenation is commutative. Elimination order changes the expression, not the recognized language.

Challenge: Solve a Regular Grammar Algebraically

Task. Find an RE for this right-linear grammar:

$$S \to abA \mid B, \qquad A \to aS \mid \varepsilon, \qquad B \to bB \mid \varepsilon.$$

Let X_S, X_A, X_B denote the terminal languages generated by the variables.

Challenge: Solve a Regular Grammar Algebraically

Task. Find an RE for this right-linear grammar:

$$S \to abA \mid B, \qquad A \to aS \mid \varepsilon, \qquad B \to bB \mid \varepsilon.$$

Let X_S, X_A, X_B denote the terminal languages generated by the variables.

$$X_B = b^*, \qquad X_A = aX_S \cup \varepsilon, \qquad X_S = abX_A \cup X_B = abaX_S \cup ab \cup b^*.$$

Arden's lemma applies because $\varepsilon \notin \{aba\}$, giving

$$X_S = (aba)^*(ab \cup b^*).$$

Challenge: Solve a Regular Grammar Algebraically

Task. Find an RE for this right-linear grammar:

$$S \to abA \mid B, \qquad A \to aS \mid \varepsilon, \qquad B \to bB \mid \varepsilon.$$

Let X_S, X_A, X_B denote the terminal languages generated by the variables.

$$X_B = b^*, \qquad X_A = aX_S \cup \varepsilon, \qquad X_S = abX_A \cup X_B = abaX_S \cup ab \cup b^*.$$

Arden's lemma applies because $\varepsilon \notin \{aba\}$, giving

$$X_S = (aba)^*(ab \cup b^*).$$

Independent check. Each return $S \Rightarrow abA \Rightarrow abaS$ contributes aba. Termination contributes either ab via $A \to \varepsilon$, or a word of b^* via $S \to B$. These are all possible choices, proving both inclusions.

Original synthesis exercise combining Linz's regular-grammar constructions with Arden's lemma; unit and empty productions matter.

Brzozowski Derivatives: Reading One Symbol

The derivative $D_a(R)$ describes the possible suffixes after reading a:

$$L(D_a(R)) = \{z : az \in L(R)\}.$$

Recall that $\nu(R)$ means $\varepsilon \in L(R)$.

Brzozowski Derivatives: Reading One Symbol

The derivative $D_a(R)$ describes the possible suffixes after reading a:

$$L(D_a(R)) = \{z : az \in L(R)\}.$$

Recall that $\nu(R)$ means $\varepsilon \in L(R)$.

$$D_a(\emptyset) = D_a(\varepsilon) = \emptyset,$$

$$D_a(b) = \begin{cases} \varepsilon & b = a, \\ \emptyset & b \neq a, \end{cases}$$

$$D_a(R \cup S) = D_a(R) \cup D_a(S),$$

$$D_a(RS) = D_a(R)S \cup \begin{cases} D_a(S) & \nu(R), \\ \emptyset & \text{otherwise}, \end{cases}$$

$$D_a(R^*) = D_a(R)R^*.$$

Brzozowski Derivatives: Reading One Symbol

The derivative $D_a(R)$ describes the possible suffixes after reading a:

$$L(D_a(R)) = \{z : az \in L(R)\}.$$

Recall that $\nu(R)$ means $\varepsilon \in L(R)$.

$$D_a(\emptyset) = D_a(\varepsilon) = \emptyset,$$

$$D_a(b) = \begin{cases} \varepsilon & b = a, \\ \emptyset & b \neq a, \end{cases}$$

$$D_a(R \cup S) = D_a(R) \cup D_a(S),$$

$$D_a(RS) = D_a(R)S \cup \begin{cases} D_a(S) & \nu(R), \\ \emptyset & \text{otherwise}, \end{cases}$$

$$D_a(R^*) = D_a(R)R^*.$$

In concatenation, a is consumed in the first factor unless that factor contributes ε. For membership, differentiate along the input and test whether the final expression is nullable. See [B64].

Worked Derivatives: A DFA for the Earlier RE

Let $R = (ab \cup a)^*$ and $T = (b \cup \varepsilon)R$. Simplifying derivatives gives the following complete transition table:

Expression state	on a	on b	Nullable (accepting)?
R (start)	T	$\emptyset$	yes
T	T	R	yes
$\emptyset$	$\emptyset$	$\emptyset$	no

Worked Derivatives: A DFA for the Earlier RE

Let $R = (ab \cup a)^*$ and $T = (b \cup \varepsilon)R$. Simplifying derivatives gives the following complete transition table:

Expression state	on a	on b	Nullable (accepting)?
R (start)	T	$\emptyset$	yes
T	T	R	yes
$\emptyset$	$\emptyset$	$\emptyset$	no

For example,

$$D_a(R) = (b \cup \varepsilon)R = T, \quad D_b(T) = \varepsilon R \cup D_b(R) = R.$$

Thus $R \xrightarrow{a} T \xrightarrow{b} R \xrightarrow{a} T$ accepts aba, whereas $R \xrightarrow{a} T \xrightarrow{b} R \xrightarrow{b} \emptyset$ rejects abb.

Worked Derivatives: A DFA for the Earlier RE

Let $R = (ab \cup a)^*$ and $T = (b \cup \varepsilon)R$. Simplifying derivatives gives the following complete transition table:

Expression state	on a	on b	Nullable (accepting)?
R (start)	T	$\emptyset$	yes
T	T	R	yes
$\emptyset$	$\emptyset$	$\emptyset$	no

For example,

$$D_a(R) = (b \cup \varepsilon)R = T, \quad D_b(T) = \varepsilon R \cup D_b(R) = R.$$

Thus $R \xrightarrow{a} T \xrightarrow{b} R \xrightarrow{a} T$ accepts aba, whereas $R \xrightarrow{a} T \xrightarrow{b} R \xrightarrow{b} \emptyset$ rejects abb.

Implementation qualification

Equivalent expressions need not look identical. A general derivative construction needs normalization or an equivalence-aware representation; blindly treating every new syntax tree as a distinct state is not enough.

Glushkov's Position Automaton: An Epsilon-Free Route

Mark the m terminal occurrences of an RE with distinct positions. The **position automaton** has these m states plus a new start 0.

- First: positions that can begin a nonempty marked word.
- Last: positions that can end a nonempty marked word.
- $\mathrm{Follow}(i)$: positions that can immediately follow i.

Glushkov's Position Automaton: An Epsilon-Free Route

Mark the m terminal occurrences of an RE with distinct positions. The **position automaton** has these m states plus a new start 0.

- First: positions that can begin a nonempty marked word.
- Last: positions that can end a nonempty marked word.
- $\mathrm{Follow}(i)$: positions that can immediately follow i.

Construction

Add $0 \xrightarrow{\text{symbol}(j)} j$ for $j \in \mathrm{First}$, and $i \xrightarrow{\text{symbol}(j)} j$ for $j \in \mathrm{Follow}(i)$. Accept at every position in Last; also accept at 0 exactly when the RE is nullable.

Glushkov's Position Automaton: An Epsilon-Free Route

Mark the m terminal occurrences of an RE with distinct positions. The **position automaton** has these m states plus a new start 0.

- First: positions that can begin a nonempty marked word.
- Last: positions that can end a nonempty marked word.
- $\mathrm{Follow}(i)$: positions that can immediately follow i.

Construction

Add $0 \xrightarrow{\mathrm{symbol}(j)} j$ for $j \in \mathrm{First}$, and $i \xrightarrow{\mathrm{symbol}(j)} j$ for $j \in \mathrm{Follow}(i)$. Accept at every position in Last; also accept at 0 exactly when the RE is nullable.

The invariant records the last matched occurrence. The marked RE's first/follow/last constraints characterize its nonempty words; removing position indices recovers the original language.

No ε-edges are needed, but the number of edges can be quadratic in m; fewer states does not automatically mean less storage. See [AM06] for this construction and related automata.

Worked Position Automaton: The Earlier RE

Mark the earlier RE as $(a_1 b_2 \cup a_3)^*$. The indices distinguish occurrences of a, not different alphabet symbols.

$$\text{First} = \{1, 3\}, \qquad \text{Last} = \{2, 3\},$$

$$\text{Follow}(1) = \{2\}, \qquad \text{Follow}(2) = \text{Follow}(3) = \{1, 3\}.$$

Worked Position Automaton: The Earlier RE

Mark the earlier RE as $(a_1 b_2 \cup a_3)^*$. The indices distinguish occurrences of a, not different alphabet symbols.

$$\text{First} = \{1, 3\}, \qquad \text{Last} = \{2, 3\},$$

$$\text{Follow}(1) = \{2\}, \qquad \text{Follow}(2) = \text{Follow}(3) = \{1, 3\}.$$

State	on a	on b	Accepting?
0 (start)	$\{1, 3\}$	$\emptyset$	yes
1	$\emptyset$	$\{2\}$	no
2	$\{1, 3\}$	$\emptyset$	yes
3	$\{1, 3\}$	$\emptyset$	yes

Worked Position Automaton: The Earlier RE

Mark the earlier RE as $(a_1 b_2 \cup a_3)^*$. The indices distinguish occurrences of a, not different alphabet symbols.

$$\mathrm{First} = \{1, 3\}, \qquad \mathrm{Last} = \{2, 3\},$$

$$\mathrm{Follow}(1) = \{2\}, \qquad \mathrm{Follow}(2) = \mathrm{Follow}(3) = \{1, 3\}.$$

State	on a	on b	Accepting?
0 (start)	$\{1, 3\}$	$\emptyset$	yes
1	$\emptyset$	$\{2\}$	no
2	$\{1, 3\}$	$\emptyset$	yes
3	$\{1, 3\}$	$\emptyset$	yes

Both $0 \xrightarrow{a} 3$ and $0 \xrightarrow{a} 1 \xrightarrow{b} 2$ accept a factor. Star contributes the transitions back to first positions. Compare: the Thompson construction uses fragment boundaries; the derivative DFA stores residual expressions; this NFA stores matched positions.

Observation: Ambiguity in a Regular Grammar

Informally, a grammar is **ambiguous** if a word has two different parse structures (here, two distinct production-choice sequences). Consider the right-linear grammar

$$S \to aS \mid A, \qquad A \to aA \mid \varepsilon.$$

For example, a has the derivations

$$S \Rightarrow aS \Rightarrow aA \Rightarrow a, \qquad S \Rightarrow A \Rightarrow aA \Rightarrow a.$$

Observation: Ambiguity in a Regular Grammar

Informally, a grammar is **ambiguous** if a word has two different parse structures (here, two distinct production-choice sequences). Consider the right-linear grammar

$$S \to aS \mid A, \qquad A \to aA \mid \varepsilon.$$

For example, a has the derivations

$$S \Rightarrow aS \Rightarrow aA \Rightarrow a, \qquad S \Rightarrow A \Rightarrow aA \Rightarrow a.$$

Its language is a^*, which also has the unambiguous grammar $S \to aS \mid \varepsilon$. For a^n, the first grammar has $n+1$ choices for when to switch from S to A.

Why every regular language has an unambiguous grammar

Convert a DFA to a right-linear grammar. Each input symbol forces the next state-variable, and only the final accepting state can terminate the derivation. The unique accepting path gives a unique derivation.

Ambiguity concerns the chosen description, not regularity of its language. Formal parse-tree definitions belong to the CFG slide set.

References: Enrichment and Exercise Sources

[B64] Janusz A. Brzozowski, *Derivatives of Regular Expressions*, Journal of the ACM 11(4), 1964, pp. 481–494.

[AM06] Cyril Allauzen and Mehryar Mohri, *A Unified Construction of the Glushkov, Follow, and Antimirov Automata*, NYU technical report TR2006-880, 2006.

[PSU13] Tim Sheard, Portland State University CS 311, *Homework 3*, October 16, 2013. Identifies the Sipser 1.17 and 1.18 problems adapted here.

Attribution and scope

The Sipser challenges have independently worked solutions. The other challenges synthesize the RE-design and grammar-conversion themes of Linz and Sipser; no unverified textbook exercise numbers are assigned.

References and Suggested Use

[LR23] Peter Linz and Susan H. Rodger, *An Introduction to Formal Languages and Automata*, 7th ed., Jones & Bartlett Learning, 2023. Chapter 3: regular languages and regular grammars.

[S13] Michael Sipser, *Introduction to the Theory of Computation*, 3rd ed., Cengage, 2013. Section 1.3: regular expressions and finite-automaton equivalence.

[T68] Ken Thompson, *Regular Expression Search Algorithm*, Communications of the ACM 11(6), 1968, pp. 419–422.

[C07] Russ Cox, *Regular Expression Matching Can Be Simple And Fast*, 2007. An implementation-oriented explanation of Thompson simulation.

Use the main sequence for the core lecture and the appendix for solutions and optional enrichment. Specific adapted exercises are attributed on their slides; unnumbered challenges are additional, newly worked practice.