Regular Expressions and Regular Grammars

Algebraic Descriptions and Grammatical Foundations

Computer Science Theory

Slide Set 3

Table of Contents

Part 1: Inductive Definition of RE

Self-Study Header: Formal Syntax of Regular Expressions

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

Self-Study Header: Formal Syntax of Regular Expressions

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:

1. a for some $a \in \Sigma$ (representing the language $\{a\}$)[cite: 817].
2. ε (representing the language $\{\varepsilon\}$)[cite: 817].
3. $\emptyset$ (representing the empty language)[cite: 817].
4. $(R_1 \cup R_2)$, where R_1, R_2 are REs[cite: 817].
5. $(R_1 \circ R_2)$, where R_1, R_2 are REs[cite: 817].
6. (R_1^*), where R_1 is an RE[cite: 817].

Part 1: Inductive Definition of RE

Self-Study Header: Formal Syntax of Regular Expressions

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:

1. a for some $a \in \Sigma$ (representing the language $\{a\}$)[cite: 817].
2. ε (representing the language $\{\varepsilon\}$)[cite: 817].
3. $\emptyset$ (representing the empty language)[cite: 817].
4. $(R_1 \cup R_2)$, where R_1, R_2 are REs[cite: 817].
5. $(R_1 \circ R_2)$, where R_1, R_2 are REs[cite: 817].
6. (R_1^*), where R_1 is an RE[cite: 817].

Notation Note

We will use $\cup$ for union and ε for the empty string to align with modern texts like Sipser[cite: 817]. Be aware that older texts, such as Linz, use $+$ for union and λ for the empty string[cite: 353].

Precedence Rules and the Empty Sets

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 $(0 \cup 1) \circ 0^*$[cite: 818].

Precedence Rules and the Empty Sets

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 $(0 \cup 1) \circ 0^*$[cite: 818].

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

A frequent source of confusion is the distinction between $\emptyset$ and ε[cite: 817].

- $\emptyset$ represents the language containing **no strings**[cite: 817].
- ε represents the language containing **exactly one string** (the empty string)[cite: 817].
- Concatenation identities: $R \circ \varepsilon = R$, but $R \circ \emptyset = \emptyset$[cite: 819].
- Union identities: $R \cup \emptyset = R$, but $R \cup \varepsilon$ adds the empty string to $L(R)$[cite: 819].

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[cite: 818].

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[cite: 818].

2. Strings that do NOT contain '00' as a substring:

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

Explanation: Every '0' must be immediately followed by a '1', except potentially for a single '0' at the very end of the string.

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[cite: 818].

2. Strings that do NOT contain '00' as a substring:

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

Explanation: Every '0' must be immediately followed by a '1', except potentially for a single '0' at the very end of the string.

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.

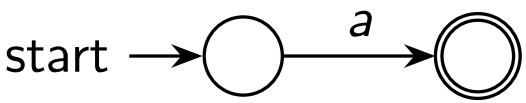
Part 2: RE to NFA — The Building Blocks

Theorem: A language is regular if and only if some regular expression describes it[cite: 819].
We prove the forward direction by showing how to convert any RE into an NFA.
First, we establish the atomic primitives:

Part 2: RE to NFA — The Building Blocks

Theorem: A language is regular if and only if some regular expression describes it[cite: 819].
We prove the forward direction by showing how to convert any RE into an NFA.
First, we establish the atomic primitives:

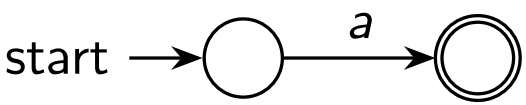
RE: a **for** $a \in \Sigma$



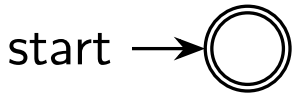
Part 2: RE to NFA — The Building Blocks

Theorem: A language is regular if and only if some regular expression describes it[cite: 819].
We prove the forward direction by showing how to convert any RE into an NFA.
First, we establish the atomic primitives:

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



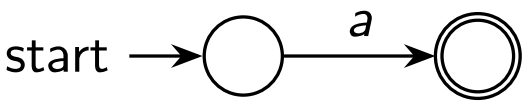
RE: ε



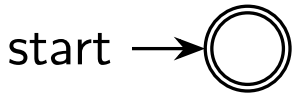
Part 2: RE to NFA — The Building Blocks

Theorem: A language is regular if and only if some regular expression describes it[cite: 819].
We prove the forward direction by showing how to convert any RE into an NFA.
First, we establish the atomic primitives:

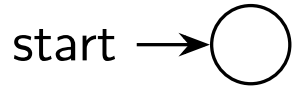
RE: a for $a \in \Sigma$



RE: ε



RE: $\emptyset$



Notice these atomic machines all have a single start state and at most one accept state. This standardization helps our induction.

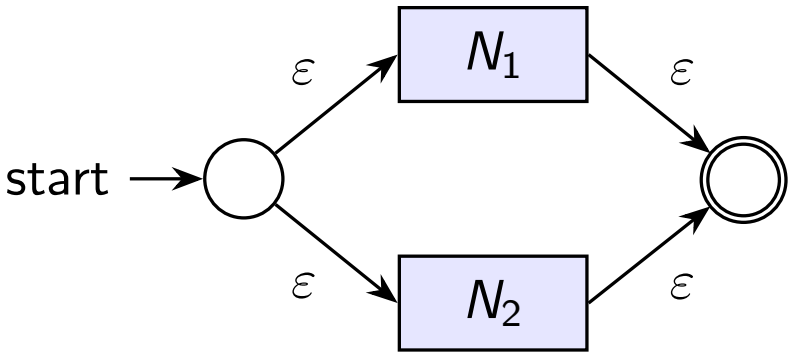
Inductive Steps: Union and Concatenation

Assume we have NFAs N_1 and N_2 for expressions R_1 and R_2.

Inductive Steps: Union and Concatenation

Assume we have NFAs N_1 and N_2 for expressions R_1 and R_2.

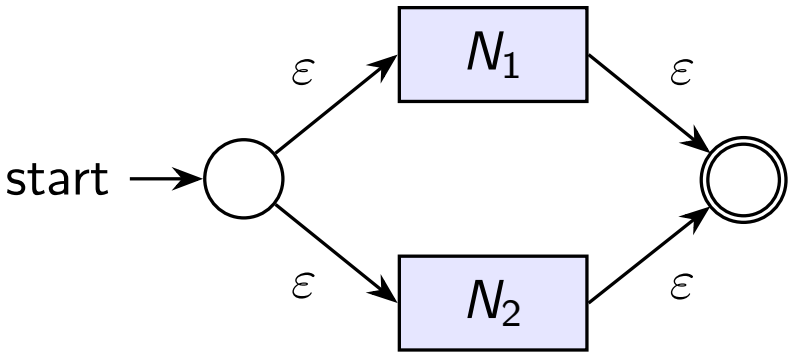
Union ($R_1 \cup R_2$)



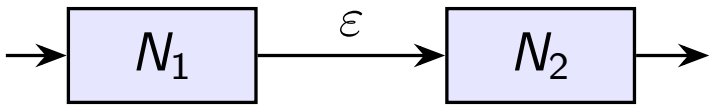
Inductive Steps: Union and Concatenation

Assume we have NFAs N_1 and N_2 for expressions R_1 and R_2.

Union ($R_1 \cup R_2$)



Concatenation ($R_1 \circ R_2$)



We use ε-transitions to seamlessly connect the sub-machines without accidentally reading an input symbol.

Inductive Steps: The Star Operation

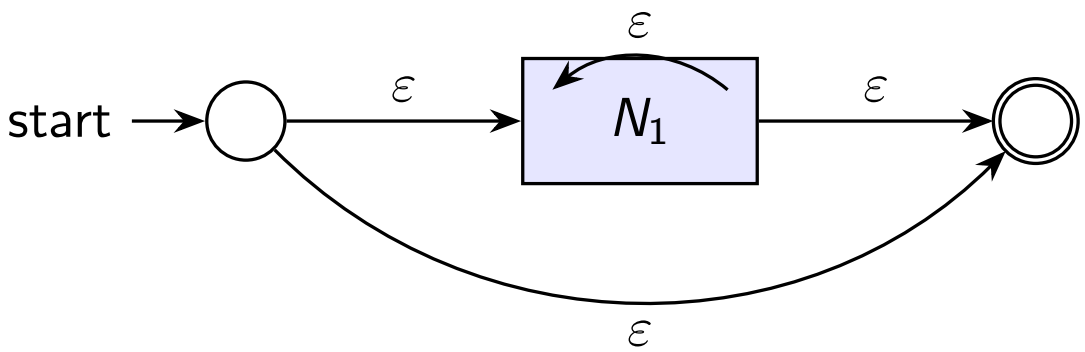
Star Operation (R_1^*)

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^*)

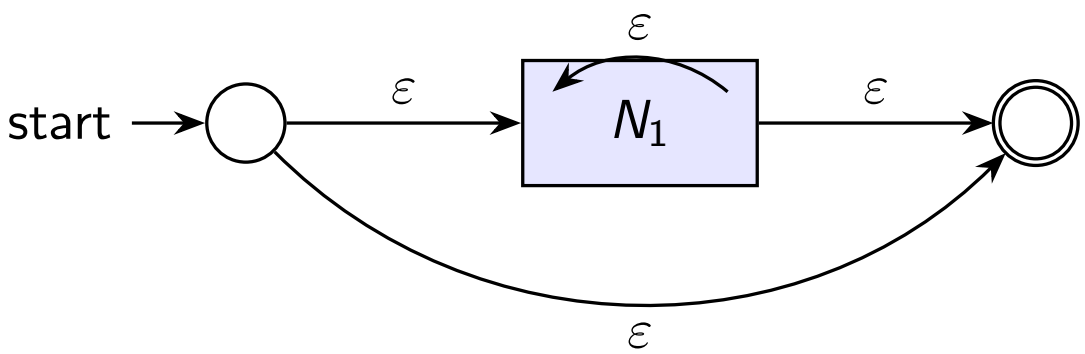
The Star operation requires us to accept the empty string ε (zero repetitions), or loop back to the beginning of N_1 after a successful pass (one or more repetitions).



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).



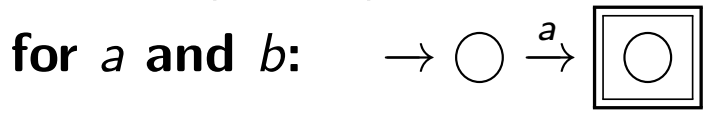
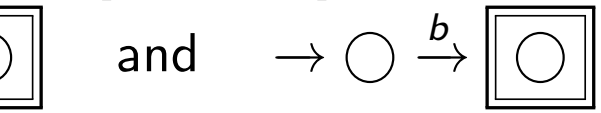
Why a new start state? If we simply made the old start state an accept state, incoming loops might inadvertently accept prefixes we didn't intend! Adding a dedicated start/accept state avoids this.

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

Let's build an NFA for $(ab \cup a)^*$ stage-by-stage[cite: 821].

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

Let's build an NFA for $(ab \cup a)^*$ stage-by-stage[cite: 821].

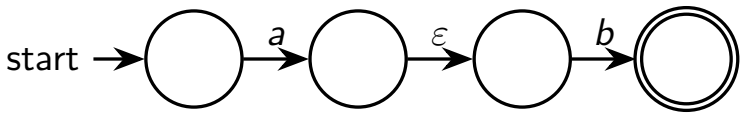
1. Primitives for a and b:  and 

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

Let's build an NFA for $(ab \cup a)^*$ stage-by-stage[cite: 821].

1. **Primitives for a and b:** 

2. **Concatenation ab:**

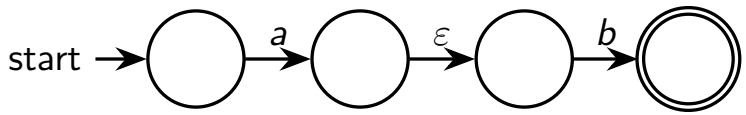


Worked Example: Converting $(ab \cup a)^*$

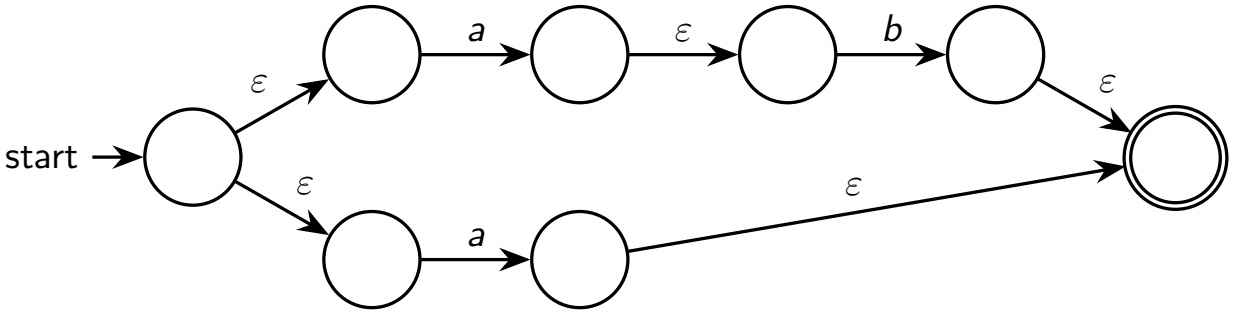
Let's build an NFA for $(ab \cup a)^*$ stage-by-stage[cite: 821].

1. **Primitives for a and b:** 

2. **Concatenation ab:**



3. **Union $(ab \cup a)$:**



(Applying the Star wrapper simply adds ε-loops from the end of this block back to the beginning, plus a master bypass).

Part 3: Generalized NFAs (GNFA)

To convert a DFA back into a Regular Expression, we use a **Generalized NFA (GNFA)**.

Part 3: Generalized NFAs (GNFA)

To convert a DFA back into a Regular Expression, we use a **Generalized NFA (GNFA)**.

What is a GNFA?

A GNFA is an NFA where the transition arrows are labeled with **regular expressions** instead of single symbols[cite: 823]. It reads blocks of symbols matching the regex on the transition[cite: 823].

Part 3: Generalized NFAs (GNFA)

To convert a DFA back into a Regular Expression, we use a **Generalized NFA (GNFA)**.

What is a GNFA?

A GNFA is an NFA where the transition arrows are labeled with **regular expressions** instead of single symbols[cite: 823]. It reads blocks of symbols matching the regex on the transition[cite: 823].

Standard Form for GNFA conversion:

1. A new **start state** with an ε-arrow to the old start state, and no incoming arrows[cite: 823].
2. A single new **accept state** with ε-arrows from the old accept states, and no outgoing arrows[cite: 823].
3. Except for the start and accept states, exactly one arrow exists from every state to every other state (including itself), potentially labeled $\emptyset$[cite: 823].

The Conversion Algorithm: The "Rip" Process

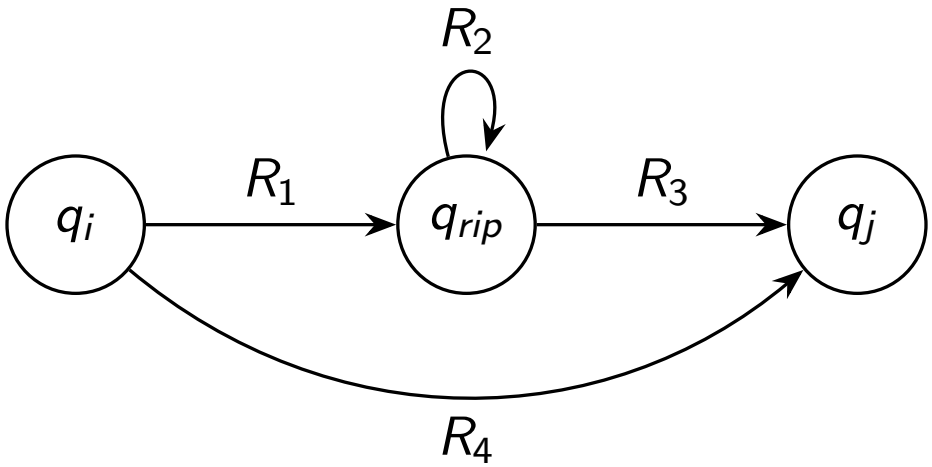
Objective: Reduce the machine down to 2 states (Start and Accept) by "ripping" out internal states one by one[cite: 824].

The Conversion Algorithm: The "Rip" Process

Objective: Reduce the machine down to 2 states (Start and Accept) by "ripping" out internal states one by one[cite: 824].
When we rip out a state q_{rip}, we must repair the graph so the language remains unchanged. To compute the new label for the edge $q_i \to q_j$:

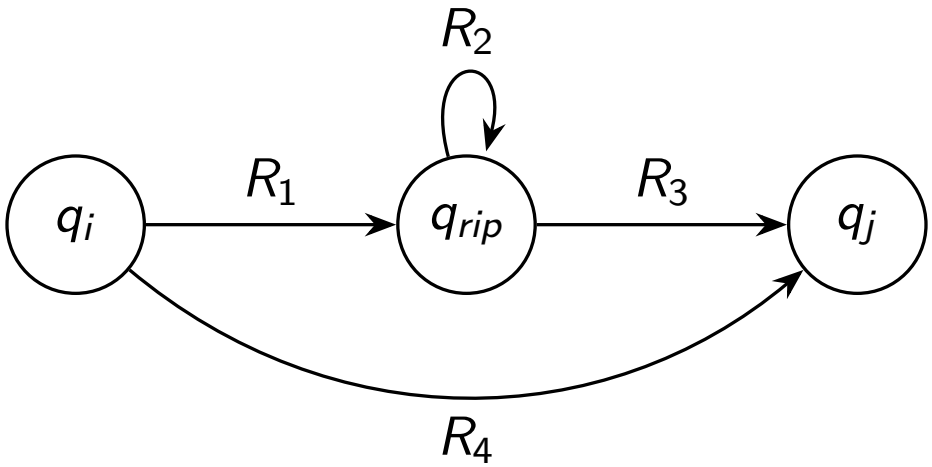
The Conversion Algorithm: The "Rip" Process

Objective: Reduce the machine down to 2 states (Start and Accept) by "ripping" out internal states one by one[cite: 824].
When we rip out a state q_{rip}, we must repair the graph so the language remains unchanged. To compute the new label for the edge $q_i \to q_j$:



The Conversion Algorithm: The "Rip" Process

Objective: Reduce the machine down to 2 states (Start and Accept) by "ripping" out internal states one by one[cite: 824].
When we rip out a state q_{rip}, we must repair the graph so the language remains unchanged. To compute the new label for the edge $q_i \to q_j$:



The Update Formula

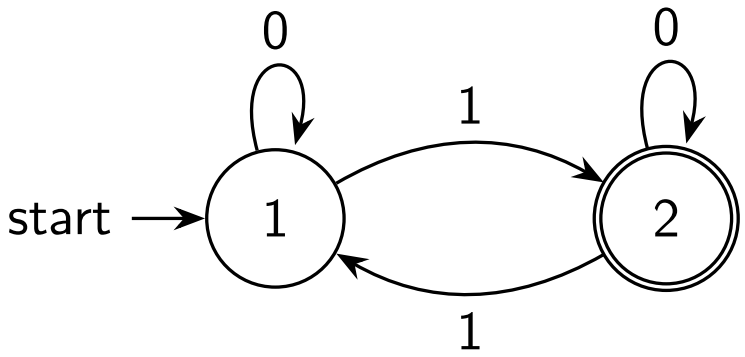
The new label bypassing q_{rip} from q_i to q_j becomes:

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

We compute this update for **every pair** (q_i, q_j) of remaining states[cite: 825, 826].

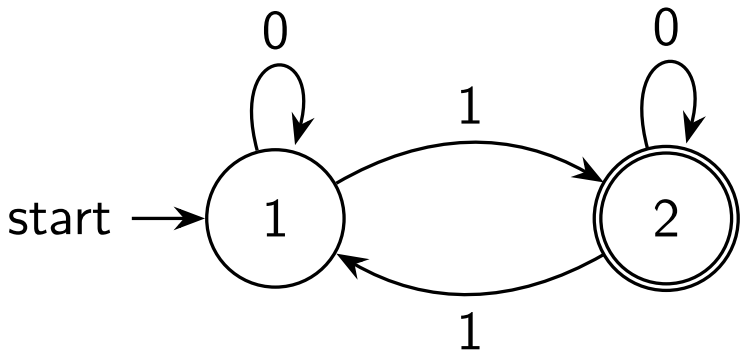
Worked Example: DFA to GNFA

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

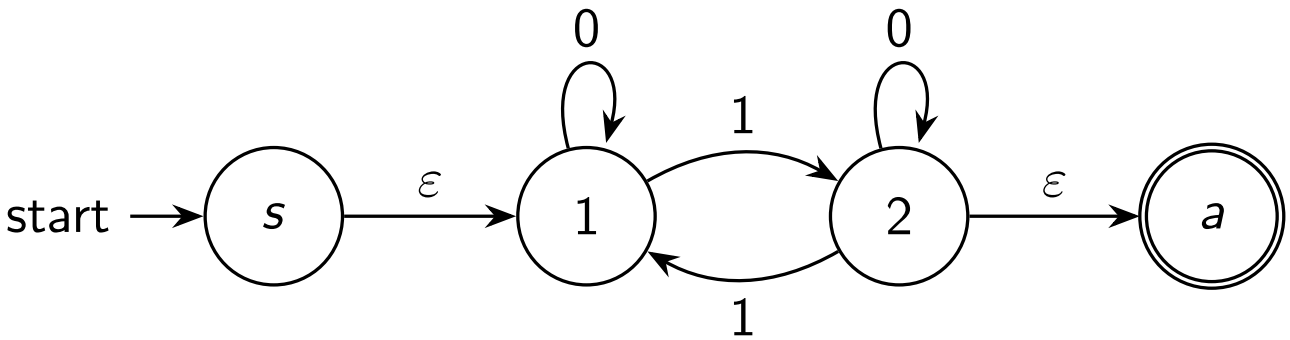


Worked Example: DFA to GNFA

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

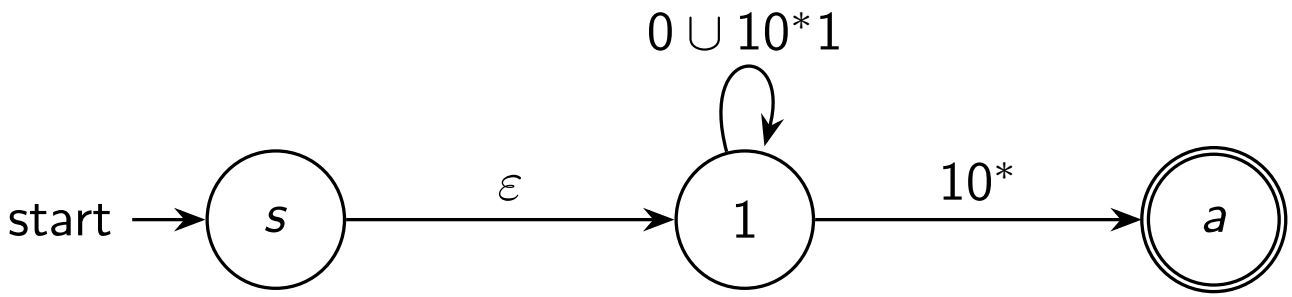
Step 2: Rip State 2. We need to update edges: $(1 \to 1)$, $(1 \to a)$.

- $1 \to 1$: Path $1 \xrightarrow{1} 2 \xrightarrow{0^*} 2 \xrightarrow{1} 1$ becomes 10^*1. Union with existing 0: $0 \cup 10^*1$.
- $1 \to a$: Path $1 \xrightarrow{1} 2 \xrightarrow{0^*} 2 \xrightarrow{\varepsilon} a$ becomes 10^*. Union with existing $\emptyset$: 10^*.

Worked Example: State Elimination

Step 2: Rip State 2. We need to update edges: $(1 \to 1)$, $(1 \to a)$.

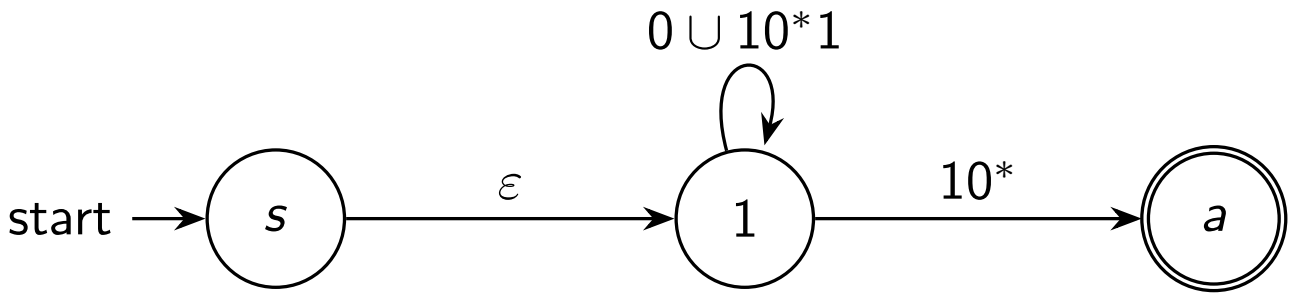
- $1 \to 1$: Path $1 \xrightarrow{1} 2 \xrightarrow{0^*} 2 \xrightarrow{1} 1$ becomes 10^*1. Union with existing 0: $0 \cup 10^*1$.
- $1 \to a$: Path $1 \xrightarrow{1} 2 \xrightarrow{0^*} 2 \xrightarrow{\varepsilon} a$ becomes 10^*. Union with existing $\emptyset$: 10^*.



Worked Example: State Elimination

Step 2: Rip State 2. We need to update edges: $(1 \to 1)$, $(1 \to a)$.

- $1 \to 1$: Path $1 \xrightarrow{1} 2 \xrightarrow{0^*} 2 \xrightarrow{1} 1$ becomes 10^*1. Union with existing 0: $0 \cup 10^*1$.
- $1 \to a$: Path $1 \xrightarrow{1} 2 \xrightarrow{0^*} 2 \xrightarrow{\varepsilon} a$ becomes 10^*. Union with existing $\emptyset$: 10^*.



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

Final RE: $(0 \cup 10^*1)^*10^*$

Part 4: Definitions of Regular Grammars

Self-Study Header: Restricting Context-Free Grammars

We've seen that DFAs, NFAs, and REs all represent Regular Languages. We can also define Regular Languages syntactically using restricted grammars.

Part 4: Definitions of Regular Grammars

Self-Study Header: Restricting Context-Free Grammars

We've seen that DFAs, NFAs, and REs all represent Regular Languages. We can also define Regular Languages syntactically using restricted grammars.

Regular Grammar Definition

A grammar $G = (V, T, S, P)$ is **regular** if it is either right-linear or left-linear[cite: 353].

Right-Linear: All productions are of the form[cite: 353]:

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

Left-Linear: All productions are of the form[cite: 353]:

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

Part 4: Definitions of Regular Grammars

Self-Study Header: Restricting Context-Free Grammars

We've seen that DFAs, NFAs, and REs all represent Regular Languages. We can also define Regular Languages syntactically using restricted grammars.

Regular Grammar Definition

A grammar $G = (V, T, S, P)$ is **regular** if it is either right-linear or left-linear[cite: 353].

Right-Linear: All productions are of the form[cite: 353]:

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

Left-Linear: All productions are of the form[cite: 353]:

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

Important: In a regular grammar, at most one variable appears on the right side of any production, and it must consistently be on the extreme left or right[cite: 353].

Equivalence Theorem: Grammars and Automata

Theorem: A language L is regular if and only if there exists a regular grammar G such that $L = L(G)$[cite: 355, 359].

Equivalence Theorem: Grammars and Automata

Theorem: A language L is regular if and only if there exists a regular grammar G such that $L = L(G)$[cite: 355, 359].

Mapping a DFA to a Right-Linear Grammar: Let $M = (Q, \Sigma, \delta, q_0, F)$. We build $G = (V, T, S, P)$:

1. $V = Q$ (Every state is a variable)[cite: 357].
2. $T = \Sigma$ (The alphabet is the terminals).
3. $S = q_0$ (Start state is the start variable)[cite: 357].
4. For every transition $\delta(q_i, a) = q_j$, add rule: $q_i \to aq_j$[cite: 357].
5. For every accept state $q_k \in F$, add rule: $q_k \to \varepsilon$ (or $q_k \to \lambda$)[cite: 357].

Equivalence Theorem: Grammars and Automata

Theorem: A language L is regular if and only if there exists a regular grammar G such that $L = L(G)$[cite: 355, 359].

Mapping a DFA to a Right-Linear Grammar: Let $M = (Q, \Sigma, \delta, q_0, F)$. We build $G = (V, T, S, P)$:

1. $V = Q$ (Every state is a variable)[cite: 357].
2. $T = \Sigma$ (The alphabet is the terminals).
3. $S = q_0$ (Start state is the start variable)[cite: 357].
4. For every transition $\delta(q_i, a) = q_j$, add rule: $q_i \to aq_j$[cite: 357].
5. For every accept state $q_k \in F$, add rule: $q_k \to \varepsilon$ (or $q_k \to \lambda$)[cite: 357].

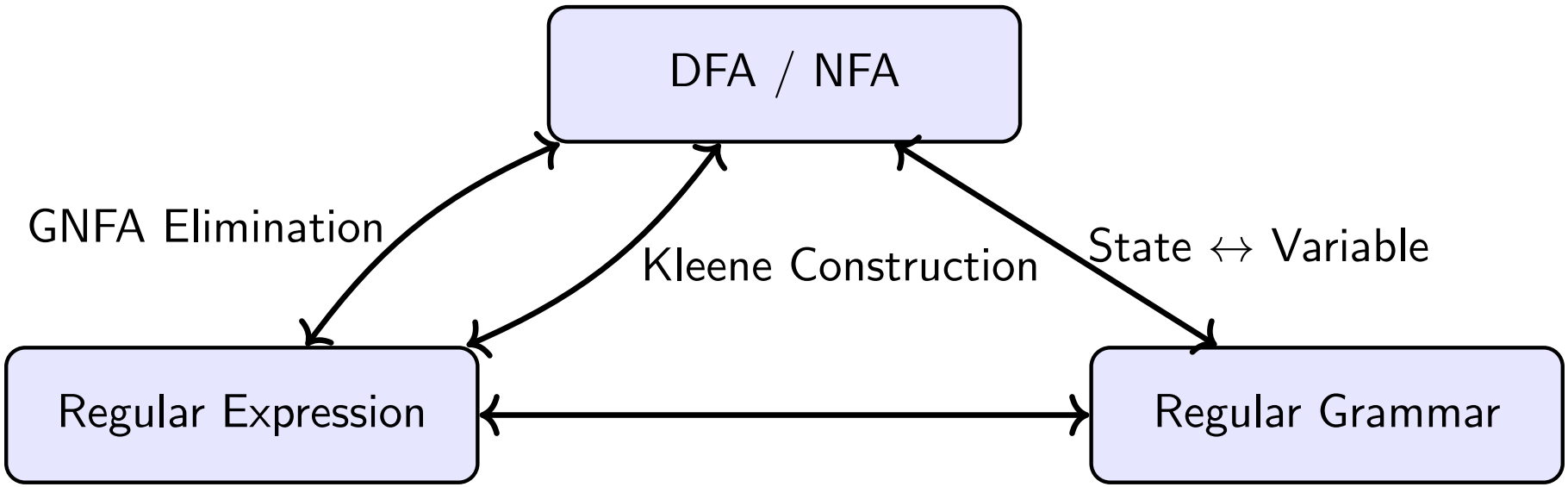
This directly mirrors the sequence of states an automaton passes through when accepting a string!

Part 5: The Equivalence Map

We have established four equivalent ways to describe Regular Languages!

Part 5: The Equivalence Map

We have established four equivalent ways to describe 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., variables, constants, reserved words).
Example: A numeric constant can be defined as $+ \cup - \cup \varepsilon \circ D^+ \cup D^+ \circ .D^*$[cite: 819]. Tools like `lex` automatically convert these REs into DFAs to parse source code rapidly.

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., variables, constants, reserved words).
Example: A numeric constant can be defined as $+ \cup - \cup \varepsilon \circ D^+ \cup D^+ \circ .D^*$[cite: 819]. Tools like `lex` automatically convert these REs into DFAs to parse source code rapidly.

2. Pattern Matching and Text Editors

Command-line utilities in UNIX, such as `grep` and `awk`, and modern languages like Perl, utilize RE engines for powerful string search algorithms[cite: 816, 350]. Automata theory provides the algorithms to execute these searches in linear time relative to text length.

Checkpoint Exercises

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

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

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

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^*.
- **Exercise 2:** You need three variables (e.g., S, A, B). S loops on anything, but can transition to A on a '0'. A transitions to B on a '1'. B is the only state that derives ε.

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^*.
- **Exercise 2:** You need three variables (e.g., S, A, B). S loops on anything, but can transition to A on a '0'. A transitions to B on a '1'. B is the only state that derives ε.

Final Pro-Tip for State Elimination

Always simplify your Regular Expression labels at each step of the GNFA rip process. For instance, simplify εR to R, and $R \cup \emptyset$ to R. If you leave them unsimplified, the final formula becomes completely unreadable!