REGULAR LANGUAGES, FINITE AUTOMATA, AND LEXICAL ANALYSIS

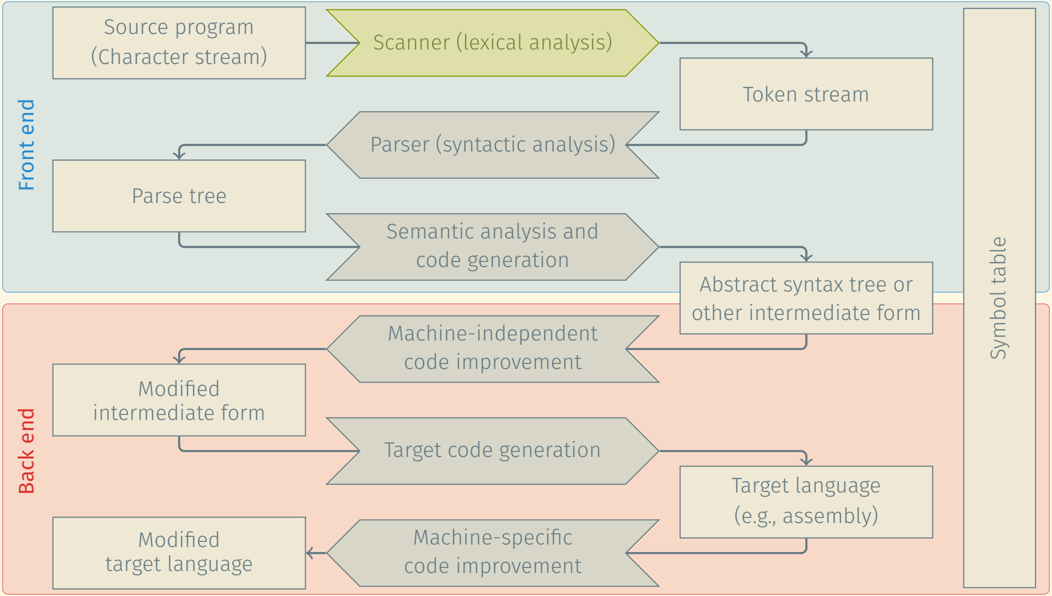
PRINCIPLES OF PROGRAMMING LANGUAGES

Norbert Zeh

Winter 2018

Dalhousie University

PROGRAM TRANSLATION FLOW CHART



Goal

Transform input text into much more compact token stream (keywords, parentheses, operators, identifiers, ...).

Goal

Transform input text into much more compact token stream (keywords, parentheses, operators, identifiers, ...).

```
[class] [DictEntry] [{]
    [int] [key] [;] // The key
    [float] [value] [;] // The associated value
[}] [;]
```

Goal

Transform input text into much more compact token stream (keywords, parentheses, operators, identifiers, ...).

```
class DictEntry {
    int key ; // The key
    float value ; // The associated value
} ;
```

```
kwClass identifier '{'
    identifier identifier ';'
    identifier identifier ';'
'}' ';'
```

Goal

Transform input text into much more compact token stream (keywords, parentheses, operators, identifiers, …).

$\boxed{\text{class}}$ $\boxed{\text{DictEntry}}$ $\boxed{\{}$	kwClass identifier '{'
$\boxed{\text{int}}$ $\boxed{\text{key}}$ $\boxed{;}$ // The key	identifier identifier ';'
$\boxed{\text{float}}$ $\boxed{\text{value}}$ $\boxed{;}$ // The associated value	identifier identifier ';'
$\boxed{\}}$ $\boxed{;}$	'}' ';'

Tools

- Regular expressions
- **Finite automata:** Very simple and efficient machines just powerful enough to carry out lexical analysis

ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA
- Building a scanner
  - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
  - Minimizing the DFA
- Limitations of regular languages

ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA
- Building a scanner
    - Regular expression $\rightarrow$ NFA $\rightarrow$ DFA
    - Minimizing the DFA
- Limitations of regular languages

Definition: Regular Language

Definition: Regular Language

Base cases:

Definition: Regular Language

Base cases: $\emptyset$, are regular languages

Definition: Regular Language

Base cases: $\emptyset$, $\{\epsilon\}$,   are regular languages

Definition: Regular Language

Base cases: $\emptyset$, $\{\epsilon\}$, and $\{a\}$ are regular languages, where $a \in \Sigma$.

Definition: Regular Language

Base cases: $\emptyset$, $\{\epsilon\}$, and $\{a\}$ are regular languages, where $a \in \Sigma$.

Induction: If A and B are regular languages, then

Definition: Regular Language

Base cases: $\emptyset$, $\{\epsilon\}$, and $\{a\}$ are regular languages, where $a \in \Sigma$.

Induction: If A and B are regular languages, then

- $A \cup B$ is a regular language,

Definition: Regular Language

Base cases: $\emptyset$, $\{\epsilon\}$, and $\{a\}$ are regular languages, where $a \in \Sigma$.

Induction: If A and B are regular languages, then

- $A \cup B$ is a regular language,
- $AB = \{ab \mid a \in A, b \in B\}$ is a regular language,
  (ab is the concatenation of a and b)

Definition: Regular Language

Base cases: $\emptyset$, $\{\epsilon\}$, and $\{a\}$ are regular languages, where $a \in \Sigma$.

Induction: If A and B are regular languages, then

- $A \cup B$ is a regular language,
- $AB = \{ab \mid a \in A, b \in B\}$ is a regular language, (ab is the concatenation of a and b)
- $A^* = A^0 \cup A^1 \cup A^2 \cup \cdots$ is a regular language:
  - $A^0 = \{\epsilon\}$
  - $A^i = \{\sigma_1\sigma_2 \mid \sigma_1 \in A^{i-1}, \sigma_2 \in A\}$

Definition: Regular Language

Base cases: $\emptyset$, $\{\epsilon\}$, and $\{a\}$ are regular languages, where $a \in \Sigma$.

Induction: If A and B are regular languages, then

- $A \cup B$ is a regular language,
- $AB = \{ab \mid a \in A, b \in B\}$ is a regular language, (ab is the concatenation of a and b)
- $A^* = A^0 \cup A^1 \cup A^2 \cup \cdots$ is a regular language:
  - $A^0 = \{\epsilon\}$
  - $A^i = \{\sigma_1\sigma_2 \mid \sigma_1 \in A^{i-1}, \sigma_2 \in A\}$

This is the only way to produce infinite regular languages!

EXAMPLES OF REGULAR LANGUAGES

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$
- $\{a, b, c\}^*$

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$
- $\{a, b, c\}^*$
- $\{01^n0 \mid n \geq 0\}$

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$
- $\{a, b, c\}^*$
- $\{01^n0 \mid n \geq 0\}$
- Set of all positive integers in decimal representation

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$
- $\{a, b, c\}^*$
- $\{01^n0 \mid n \geq 0\}$
- Set of all positive integers in decimal representation
- $\{0^m1^n \mid m \geq 0, n \geq 0\}$

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$
- $\{a, b, c\}^*$
- $\{01^n0 \mid n \geq 0\}$
- Set of all positive integers in decimal representation
- $\{0^m1^n \mid m \geq 0, n \geq 0\}$
- $\{a^kb^mc^n \mid k \geq 0, m \geq 0, n \geq 0\}$

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$
- $\{a, b, c\}^*$
- $\{01^n0 \mid n \geq 0\}$
- Set of all positive integers in decimal representation
- $\{0^m1^n \mid m \geq 0, n \geq 0\}$
- $\{a^kb^mc^n \mid k \geq 0, m \geq 0, n \geq 0\}$
- $\{(^n)^n \mid n \geq 0\}$

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$
- $\{a, b, c\}^*$
- $\{01^n0 \mid n \geq 0\}$
- Set of all positive integers in decimal representation
- $\{0^m1^n \mid m \geq 0, n \geq 0\}$
- $\{a^kb^mc^n \mid k \geq 0, m \geq 0, n \geq 0\}$
- $\cancel{\{(^n)^n \mid n \geq 0\}}$

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$
- $\{a, b, c\}^*$
- $\{01^n0 \mid n \geq 0\}$
- Set of all positive integers in decimal representation
- $\{0^m1^n \mid m \geq 0, n \geq 0\}$
- $\{a^kb^mc^n \mid k \geq 0, m \geq 0, n \geq 0\}$
- $\cancel{\{(^n)^n \mid n \geq 0\}}$
- $\{a^p \mid p \text{ is a prime number}\}$

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$
- $\{a, b, c\}^*$
- $\{01^n0 \mid n \geq 0\}$
- Set of all positive integers in decimal representation
- $\{0^m1^n \mid m \geq 0, n \geq 0\}$
- $\{a^kb^mc^n \mid k \geq 0, m \geq 0, n \geq 0\}$
- $\cancel{\{(^n)^n \mid n \geq 0\}}$
- $\cancel{\{a^p \mid p \text{ is a prime number}\}}$

EXAMPLES OF REGULAR LANGUAGES

- $\{a, b, ab\}$
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$
- $\{a, b, c\}^*$
- $\{01^n0 \mid n \geq 0\}$
- Set of all positive integers in decimal representation
- $\{0^m1^n \mid m \geq 0, n \geq 0\}$
- $\{a^kb^mc^n \mid k \geq 0, m \geq 0, n \geq 0\}$
- $\cancel{\{(^n)^n \mid n \geq 0\}}$
- ~~$\{a^p \mid p$ is a prime number$\}$~~
- All syntactically correct C programs

EXAMPLES OF REGULAR LANGUAGES

- {a, b, ab}
- Any finite language!
- $\{0\}^*$
- $\{0, 1\}^*$
- $\{a, b, c\}^*$
- $\{01^n0 \mid n \geq 0\}$
- Set of all positive integers in decimal representation
- $\{0^m1^n \mid m \geq 0, n \geq 0\}$
- $\{a^kb^mc^n \mid k \geq 0, m \geq 0, n \geq 0\}$
- $\cancel{\{(^n)^n \mid n \geq 0\}}$
- ~~$\{a^p \mid p$ is a prime number$\}$~~
- ~~All syntactically correct C programs~~

ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA
- Building a scanner
    - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
    - Minimizing the DFA
- Limitations of regular languages

ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA

- Building a scanner
  - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
  - Minimizing the DFA

- Limitations of regular languages

… are a notation for specifying regular languages.

… are a notation for specifying regular languages.

Definition: Regular expression

… are a notation for specifying regular languages.

Definition: Regular expression

Base cases: $\emptyset$, ϵ, and a are regular expressions, where $a \in \Sigma$.

… are a notation for specifying regular languages.

Definition: Regular expression

Base cases: $\emptyset$, ϵ, and a are regular expressions, where $a \in \Sigma$.

Induction: If A and B are regular expressions, then

… are a notation for specifying regular languages.

Definition: Regular expression

Base cases: $\emptyset$, ϵ, and a are regular expressions, where $a \in \Sigma$.

Induction: If A and B are regular expressions, then

- $A|B$ is a regular expression,

... are a notation for specifying regular languages.

Definition: Regular expression

Base cases: $\emptyset$, ϵ, and a are regular expressions, where $a \in \Sigma$.

Induction: If A and B are regular expressions, then

- $A|B$ is a regular expression,
- AB is a regular expression,

... are a notation for specifying regular languages.

Definition: Regular expression

Base cases: $\emptyset$, ϵ, and a are regular expressions, where $a \in \Sigma$.

Induction: If A and B are regular expressions, then

- $A|B$ is a regular expression,
- AB is a regular expression,
- (A) is a regular expression,

… are a notation for specifying regular languages.

Definition: Regular expression

Base cases: $\emptyset$, ϵ, and a are regular expressions, where $a \in \Sigma$.

Induction: If A and B are regular expressions, then

- $A|B$ is a regular expression,
- AB is a regular expression,
- (A) is a regular expression,
- $A*$ is a regular expression.

… are a notation for specifying regular languages.

Definition: Regular expression

Base cases: $\emptyset$, ϵ, and a are regular expressions, where $a \in \Sigma$.

Induction: If A and B are regular expressions, then

- $A|B$ is a regular expression,
- AB is a regular expression,
- (A) is a regular expression,
- $A*$ is a regular expression.

Interpretation:

- Precedence: Kleene star (*), Concatenation, Union (|)
- Parentheses indicate grouping

Every regular expression R defines a corresponding regular language $\mathcal{L}(R)$:

Every regular expression R defines a corresponding regular language $\mathcal{L}(R)$:

$$R = \emptyset \qquad \Longrightarrow \qquad \mathcal{L}(R) = \emptyset$$

Every regular expression R defines a corresponding regular language $\mathcal{L}(R)$:

$$R = \emptyset \qquad \Longrightarrow \qquad \mathcal{L}(R) = \emptyset$$

$$R = \epsilon \qquad \Longrightarrow \qquad \mathcal{L}(R) = \{\epsilon\}$$

Every regular expression R defines a corresponding regular language $\mathcal{L}(R)$:

$$R = \emptyset \implies \mathcal{L}(R) = \emptyset$$

$$R = \epsilon \implies \mathcal{L}(R) = \{\epsilon\}$$

$$R = \mathrm{a} \implies \mathcal{L}(R) = \{\mathrm{a}\}$$

REGULAR EXPRESSIONS MODEL REGULAR LANGUAGES

Every regular expression R defines a corresponding regular language $\mathcal{L}(R)$:

$$
\begin{aligned}
R &= \emptyset & \Longrightarrow \quad & \mathcal{L}(R) = \emptyset \\
R &= \epsilon & \Longrightarrow \quad & \mathcal{L}(R) = \{\epsilon\} \\
R &= \mathrm{a} & \Longrightarrow \quad & \mathcal{L}(R) = \{\mathrm{a}\} \\
R &= A|B & \Longrightarrow \quad & \mathcal{L}(R) = \mathcal{L}(A) \cup \mathcal{L}(B)
\end{aligned}
$$

Every regular expression R defines a corresponding regular language $\mathcal{L}(R)$:

$$
\begin{aligned}
R &= \emptyset & \Longrightarrow && \mathcal{L}(R) &= \emptyset \\
R &= \epsilon & \Longrightarrow && \mathcal{L}(R) &= \{\epsilon\} \\
R &= \mathrm{a} & \Longrightarrow && \mathcal{L}(R) &= \{\mathrm{a}\} \\
R &= A|B & \Longrightarrow && \mathcal{L}(R) &= \mathcal{L}(A) \cup \mathcal{L}(B) \\
R &= AB & \Longrightarrow && \mathcal{L}(R) &= \mathcal{L}(A)\mathcal{L}(B)
\end{aligned}
$$

Every regular expression R defines a corresponding regular language $\mathcal{L}(R)$:

$$
\begin{array}{lcl}
R = \emptyset & \Longrightarrow & \mathcal{L}(R) = \emptyset \\
R = \epsilon & \Longrightarrow & \mathcal{L}(R) = \{\epsilon\} \\
R = \mathrm{a} & \Longrightarrow & \mathcal{L}(R) = \{\mathrm{a}\} \\
R = A|B & \Longrightarrow & \mathcal{L}(R) = \mathcal{L}(A) \cup \mathcal{L}(B) \\
R = AB & \Longrightarrow & \mathcal{L}(R) = \mathcal{L}(A)\mathcal{L}(B) \\
R = (A) & \Longrightarrow & \mathcal{L}(R) = \mathcal{L}(A)
\end{array}
$$

REGULAR EXPRESSIONS MODEL REGULAR LANGUAGES

Every regular expression R defines a corresponding regular language $\mathcal{L}(R)$:

$$
\begin{aligned}
R &= \emptyset & \implies & \quad \mathcal{L}(R) = \emptyset \\
R &= \epsilon & \implies & \quad \mathcal{L}(R) = \{\epsilon\} \\
R &= \mathrm{a} & \implies & \quad \mathcal{L}(R) = \{\mathrm{a}\} \\
R &= A|B & \implies & \quad \mathcal{L}(R) = \mathcal{L}(A) \cup \mathcal{L}(B) \\
R &= AB & \implies & \quad \mathcal{L}(R) = \mathcal{L}(A)\mathcal{L}(B) \\
R &= (A) & \implies & \quad \mathcal{L}(R) = \mathcal{L}(A) \\
R &= A* & \implies & \quad \mathcal{L}(R) = \mathcal{L}(A)^*
\end{aligned}
$$

EXAMPLES OF REGULAR EXPRESSIONS

$(0|1)*$

EXAMPLES OF REGULAR EXPRESSIONS

$(0\|1)*$	all binary strings

EXAMPLES OF REGULAR EXPRESSIONS

$(0\|1)*$	all binary strings
$(0\|1)*0$	

EXAMPLES OF REGULAR EXPRESSIONS

(0|1)*	all binary strings
(0|1)*0	all binary strings that end in 0

EXAMPLES OF REGULAR EXPRESSIONS

$(0\|1)*$	all binary strings
$(0\|1)*0$	all binary strings that end in 0
$(0\|1)00*$	

EXAMPLES OF REGULAR EXPRESSIONS

$(0\|1)*$	all binary strings
$(0\|1)*0$	all binary strings that end in 0
$(0\|1)00*$	all binary strings that start with 0 or 1, followed by one or more 0s

EXAMPLES OF REGULAR EXPRESSIONS

$(0\|1)*$	all binary strings
$(0\|1)*0$	all binary strings that end in 0
$(0\|1)00*$	all binary strings that start with 0 or 1, followed by one or more 0s
$0\|1(0\|1)*$	

EXAMPLES OF REGULAR EXPRESSIONS

$(0\|1)*$	all binary strings
$(0\|1)*0$	all binary strings that end in 0
$(0\|1)00*$	all binary strings that start with 0 or 1, followed by one or more 0s
$0\|1(0\|1)*$	all binary numbers without leading 0s

EXAMPLES OF REGULAR EXPRESSIONS

$(0\|1)*$	all binary strings
$(0\|1)*0$	all binary strings that end in 0
$(0\|1)00*$	all binary strings that start with 0 or 1, followed by one or more 0s
$0\|1(0\|1)*$	all binary numbers without leading 0s

What regular expression describes the set $\mathcal{L}$ of binary strings that do not contain 101 as a substring?

- $1001001110 \in \mathcal{L}$
- $000100\mathbf{101}00 \notin \mathcal{L}$

$(0\|1)*$	all binary strings
$(0\|1)*0$	all binary strings that end in 0
$(0\|1)00*$	all binary strings that start with 0 or 1, followed by one or more 0s
$0\|1(0\|1)*$	all binary numbers without leading 0s

What regular expression describes the set $\mathcal{L}$ of binary strings that do not contain 101 as a substring?

- $1001001110 \in \mathcal{L}$
- $000100101 00 \notin \mathcal{L}$

$$(0|\epsilon)(1|000*)*(0|\epsilon)$$

REGULAR EXPRESSIONS IN PRACTICE (1)

No ϵ or $\emptyset$:

REGULAR EXPRESSIONS IN PRACTICE (1)

No ϵ or $\emptyset$:

- The empty string is represented as the empty string: $a(b|)$ instead of $a(b|\epsilon)$ to express the language $\{a, ab\}$.

REGULAR EXPRESSIONS IN PRACTICE (1)

No ϵ or $\emptyset$:

- The empty string is represented as the empty string: a(b|) instead of $a(b|\epsilon)$ to express the language $\{a, ab\}$.
- The empty language is not very useful in practice.

No ϵ or $\emptyset$:

- The empty string is represented as the empty string: $a(b|)$ instead of $a(b|\epsilon)$ to express the language $\{a, ab\}$.
- The empty language is not very useful in practice.

Additional repetition constructs:

REGULAR EXPRESSIONS IN PRACTICE (1)

No ϵ or $\emptyset$:

- The empty string is represented as the empty string: $\mathrm{a}(\mathrm{b}|)$ instead of $\mathrm{a}(\mathrm{b}|\epsilon)$ to express the language $\{\mathrm{a}, \mathrm{ab}\}$.
- The empty language is not very useful in practice.

Additional repetition constructs:

- $R+ = RR*$: one or more repetitions of R

REGULAR EXPRESSIONS IN PRACTICE (1)

No ϵ or $\emptyset$:

- The empty string is represented as the empty string: a(b|) instead of a(b|ϵ) to express the language $\{a, ab\}$.
- The empty language is not very useful in practice.

Additional repetition constructs:

- $R+ = RR*$: one or more repetitions of R
- $R? = (R|)$: zero or one repetition of R

REGULAR EXPRESSIONS IN PRACTICE (1)

No ϵ or $\emptyset$:

- The empty string is represented as the empty string: $\mathrm{a}(\mathrm{b}|)$ instead of $\mathrm{a}(\mathrm{b}|\epsilon)$ to express the language $\{\mathrm{a}, \mathrm{ab}\}$.
- The empty language is not very useful in practice.

Additional repetition constructs:

- $R+ = RR*$: one or more repetitions of R
- $R? = (R|)$: zero or one repetition of R
- $R\{n\}$, $R\{,n\}$, $R\{m,\}$, $R\{m,n\}$: n, up to n, at least m, between m and n repetitions of R

No ϵ or $\emptyset$:

- The empty string is represented as the empty string: $\mathrm{a}(\mathrm{b}|)$ instead of $\mathrm{a}(\mathrm{b}|\epsilon)$ to express the language $\{\mathrm{a}, \mathrm{ab}\}$.
- The empty language is not very useful in practice.

Additional repetition constructs:

- $R+ = RR*$: one or more repetitions of R
- $R? = (R|)$: zero or one repetition of R
- $R\{n\}$, $R\{,n\}$, $R\{m,\}$, $R\{m,n\}$: n, up to n, at least m, between m and n repetitions of R

Some capabilities beyond regular languages:

- Allow, for example, recognition of languages such as $\alpha\beta\alpha$, for $\alpha, \beta \in \Sigma^*$.

REGULAR EXPRESSIONS IN PRACTICE (2)

Character classes allow us to write tedious expressions such as $a|b|\cdots|z$ more easily.

REGULAR EXPRESSIONS IN PRACTICE (2)

Character classes allow us to write tedious expressions such as $a|b|\cdots|z$ more easily.

Examples:

- **"Recent" years:** $199(6|7|8|9)|20(0(0|1|2|3|4|5|6|7|8|9)|1(0|1|2|3|4|5|6|7|8))$

REGULAR EXPRESSIONS IN PRACTICE (2)

Character classes allow us to write tedious expressions such as $a|b|\cdots|z$ more easily.

Examples:

- **"Recent" years:** $199(6|7|8|9)|20(0(0|1|2|3|4|5|6|7|8|9)|1(0|1|2|3|4|5|6|7|8))$
  $\rightarrow 199[6\text{–}9]|20(0[0\text{–}9]|1[0\text{–}8])$
- **Identifier in C:** $(a|b|\cdots|z|A|B|\cdots|Z|\_)(a|b|\cdots|z|A|B|\cdots|Z|0|1|\cdots|9|\_)*$

REGULAR EXPRESSIONS IN PRACTICE (2)

Character classes allow us to write tedious expressions such as $a|b|\cdots|z$ more easily.

Examples:

- **"Recent" years:** 199(6|7|8|9)|20(0(0|1|2|3|4|5|6|7|8|9)|1(0|1|2|3|4|5|6|7|8))
  $\longrightarrow$ 199[6–9]|20(0[0–9]|1[0–8])
- **Identifier in C:** $(a|b|\cdots|z|A|B|\cdots|Z|\_)(a|b|\cdots|z|A|B|\cdots|Z|0|1|\cdots|9|\_)*$
  $\longrightarrow$ [a–zA–Z_][a–zA–Z0–9_]$*$

REGULAR EXPRESSIONS IN PRACTICE (2)

Character classes allow us to write tedious expressions such as $a|b|\cdots|z$ more easily.

Examples:

- **"Recent" years:** 199(6|7|8|9)|20(0(0|1|2|3|4|5|6|7|8|9)|1(0|1|2|3|4|5|6|7|8))
  $\rightarrow$ 199[6–9]|20(0[0–9]|1[0–8])
- **Identifier in C:** (a|b|$\cdots$|z|A|B|$\cdots$|Z|_)(a|b|$\cdots$|z|A|B|$\cdots$|Z|0|1|$\cdots$|9|_)$*$
  $\rightarrow$ [a–zA–Z_][a–zA–Z0–9_]$*$
- **Anything but a lowercase letter:** [^a–z]

REGULAR EXPRESSIONS IN PRACTICE (2)

Character classes allow us to write tedious expressions such as $a|b|\cdots|z$ more easily.

Examples:

- **"Recent" years:** 199(6|7|8|9)|20(0(0|1|2|3|4|5|6|7|8|9)|1(0|1|2|3|4|5|6|7|8))
  $\longrightarrow$ 199[6–9]|20(0[0–9]|1[0–8])
- **Identifier in C:** $(a|b|\cdots|z|A|B|\cdots|Z|\_)(a|b|\cdots|z|A|B|\cdots|Z|0|1|\cdots|9|\_)*$
  $\longrightarrow$ [a–zA–Z_][a–zA–Z0–9_]*
- **Anything but a lowercase letter:** [^a–z]
- **Any letter:** .

REGULAR EXPRESSIONS IN PRACTICE (2)

Character classes allow us to write tedious expressions such as $a|b|\cdots|z$ more easily.

Examples:

- **"Recent" years:** $199(6|7|8|9)|20(0(0|1|2|3|4|5|6|7|8|9)|1(0|1|2|3|4|5|6|7|8))$
  $\rightarrow 199[6–9]|20(0[0–9]|1[0–8])$
- **Identifier in C:** $(a|b|\cdots|z|A|B|\cdots|Z|\_)(a|b|\cdots|z|A|B|\cdots|Z|0|1|\cdots|9|\_)*$
  $\rightarrow [a–zA–Z\_][a–zA–Z0–9\_]*$
- **Anything but a lowercase letter:** [^a–z]
- **Any letter:** .
- **Digit, non-digit:** \d, \D

REGULAR EXPRESSIONS IN PRACTICE (2)

Character classes allow us to write tedious expressions such as a|b| · · · |z more easily.

Examples:

- **“Recent” years:** 199(6|7|8|9)|20(0(0|1|2|3|4|5|6|7|8|9)|1(0|1|2|3|4|5|6|7|8))
  $\longrightarrow$ 199[6–9]|20(0[0–9]|1[0–8])
- **Identifier in C:** (a|b| · · · |z|A|B| · · · |Z|_)(a|b| · · · |z|A|B| · · · |Z|0|1| · · · |9|_)∗
  $\longrightarrow$ [a–zA–Z_][a–zA–Z0–9_]∗
- **Anything but a lowercase letter:** [^a–z]
- **Any letter:** .
- **Digit, non-digit:** \d, \D
- **Whitespace, non-whitespace:** \s, \S

REGULAR EXPRESSIONS IN PRACTICE (2)

Character classes allow us to write tedious expressions such as $a|b|\cdots|z$ more easily.

Examples:

- **“Recent” years:** 199(6|7|8|9)|20(0(0|1|2|3|4|5|6|7|8|9)|1(0|1|2|3|4|5|6|7|8))
  $\rightarrow$ 199[6–9]|20(0[0–9]|1[0–8])
- **Identifier in C:** $(a|b|\cdots|z|A|B|\cdots|Z|\_)(a|b|\cdots|z|A|B|\cdots|Z|0|1|\cdots|9|\_)*$
  $\rightarrow$ [a–zA–Z_][a–zA–Z0–9_]*
- **Anything but a lowercase letter:** [^a–z]
- **Any letter:** .
- **Digit, non-digit:** \d, \D
- **Whitespace, non-whitespace:** \s, \S
- **Word character, non-word character:** \w, \W

- Special characters: ‘(’ ‘)’ ‘{’ ‘}’ ‘[’ ‘]’ ‘,’ ‘;’ ‘:’

- Special characters: ‘(’ ‘)’ ‘{’ ‘}’ ‘[’ ‘]’ ‘,’ ‘;’ ‘:’
- Mathematical operators: ‘+|-|*|/|=|+=|-=|*=|/=’

REGULAR EXPRESSIONS FOR LEXICAL ANALYSIS

- **Special characters:** ‘(’ ‘)’ ‘{’ ‘}’ ‘[’ ‘]’ ‘,’ ‘;’ ‘:’
- **Mathematical operators:** ‘+|-|*|/|=|+=|-=|*=|/=’
- **Keywords (C++):** ‘class|struct|union|if|else|for|while|...’

REGULAR EXPRESSIONS FOR LEXICAL ANALYSIS

- **Special characters:** ‘(’ ‘)’ ‘{’ ‘}’ ‘[’ ‘]’ ‘,’ ‘;’ ‘:’
- **Mathematical operators:** ‘+|-|*|/|=|+=|-=|*=|/=’
- **Keywords (C++):** ‘class|struct|union|if|else|for|while|...’
- **Integer:** ‘[+-]?\d+’

- **Special characters:** '(' ')' '{' '}' '[' ']' ',' ';' ':'
- **Mathematical operators:** '+|-|*|/|=|+=|-=|*=|/='
- **Keywords (C++):** 'class|struct|union|if|else|for|while|...'
- **Integer:** '[+-]?\d+'
- **Float:** '[+-]?\d*\.?\d+([Ee][+-]?\d+)?'

- **Special characters:** '(' ')' '{' '}' '[' ']' ',' ';' ':'
- **Mathematical operators:** '+|-|*|/|=|+=|-=|*=|/='
- **Keywords (C++):** 'class|struct|union|if|else|for|while|...'
- **Integer:** '[+-]?\d+'
- **Float:** '[+-]?\d*\.?\d+([Ee][+-]?\d+)?'
- **Identifier (C++):** '[a-zA-Z_][a-zA-Z0-9_]*'

- **Special characters:** '(' ')' '{' '}' '[' ']' ',' ';' ':'
- **Mathematical operators:** '+|-|*|/|=|+=|-=|*=|/='
- **Keywords (C++):** 'class|struct|union|if|else|for|while|...'
- **Integer:** '[+-]?\d+'
- **Float:** '[+-]?\d*\.?\d+([Ee][+-]?\d+)?'
- **Identifier (C++):** '[a-zA-Z_][a-zA-Z0-9_]*'
- ...

REGULAR EXPRESSIONS FOR LEXICAL ANALYSIS

- **Special characters:** ‘(’ ‘)’ ‘{’ ‘}’ ‘[’ ‘]’ ‘,’ ‘;’ ‘:’
- **Mathematical operators:** ‘+|-|*|/|=|+=|-=|*=|/=’
- ~~**Keywords (C++):** ‘class|struct|union|if|else|for|while|...’~~
- **Integer:** ‘[+-]?\d+’
- **Float:** ‘[+-]?\d*\.?\d+([Ee][+-]?\d+)?’
- **Identifier (C++):** ‘[a-zA-Z_][a-zA-Z0-9_]*’
- ...

ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA
- Building a scanner
  - Regular expression $\rightarrow$ NFA $\rightarrow$ DFA
  - Minimizing the DFA
- Limitations of regular languages

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA
- Building a scanner
  - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
  - Minimizing the DFA
- Limitations of regular languages

DETERMINISTIC FINITE AUTOMATON (DFA)

… is a simple type of machine that can be used to decide regular languages.

… is a simple type of machine that can be used to decide regular languages.

Definition:
Deterministic finite automaton (DFA)

A tuple $D = (S, \Sigma, \delta, s_0, F)$:

- Set S of states
- Finite alphabet Σ
- Transition function $\delta : S \times \Sigma \to S$
- Initial state $s_0 \in S$
- Set of final states $F \subseteq S$

... is a simple type of machine that can be used to decide regular languages.

Definition:
Deterministic finite automaton (DFA)

A tuple $D = (S, \Sigma, \delta, s_0, F)$:

- Set S of states
- Finite alphabet Σ
- Transition function $\delta : S \times \Sigma \to S$
- Initial state $s_0 \in S$
- Set of final states $F \subseteq S$

Tabular representation:

Graph representation:

... is a simple type of machine that can be used to decide regular languages.

Definition:
Deterministic finite automaton (DFA)

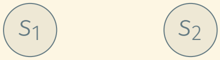
A tuple $D = (S, \Sigma, \delta, s_0, F)$:

- Set S of states
- Finite alphabet Σ
- Transition function $\delta : S \times \Sigma \to S$
- Initial state $s_0 \in S$
- Set of final states $F \subseteq S$

Tabular representation:

S	s_1	
	$*s_2$	

Graph representation:



... is a simple type of machine that can be used to decide regular languages.

Definition:
Deterministic finite automaton (DFA)

A tuple $D = (S, \Sigma, \delta, s_0, F)$:

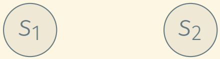
- Set S of states
- Finite alphabet Σ
- Transition function $\delta : S \times \Sigma \to S$
- Initial state $s_0 \in S$
- Set of final states $F \subseteq S$

Tabular representation:

Σ

	0	1
s_1		
$*s_2$		

Graph representation:



DETERMINISTIC FINITE AUTOMATON (DFA)

… is a simple type of machine that can be used to decide regular languages.

Definition:
Deterministic finite automaton (DFA)

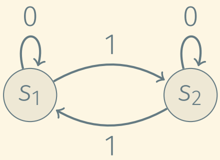
A tuple $D = (S, \Sigma, \delta, s_0, F)$:

- Set S of states
- Finite alphabet Σ
- Transition function $\delta : S \times \Sigma \to S$
- Initial state $s_0 \in S$
- Set of final states $F \subseteq S$

Tabular representation:

δ	0	1
s_1	s_1	s_2
$*s_2$	s_2	s_1

Graph representation:



... is a simple type of machine that can be used to decide regular languages.

Definition:
Deterministic finite automaton (DFA)

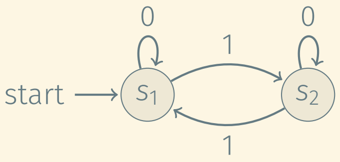
A tuple $D = (S, \Sigma, \delta, s_0, F)$:

- Set S of states
- Finite alphabet Σ
- Transition function $\delta : S \times \Sigma \to S$
- Initial state $s_0 \in S$
- Set of final states $F \subseteq S$

Tabular representation:

δ	0	1
$\to s_1$	s_1	s_2
$*s_2$	s_2	s_1

Graph representation:



DETERMINISTIC FINITE AUTOMATON (DFA)

… is a simple type of machine that can be used to decide regular languages.

Definition: Deterministic finite automaton (DFA)

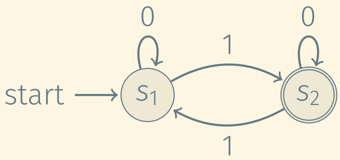
A tuple $D = (S, \Sigma, \delta, s_0, F)$:

- Set S of states
- Finite alphabet Σ
- Transition function $\delta : S \times \Sigma \to S$
- Initial state $s_0 \in S$
- Set of final states $F \subseteq S$

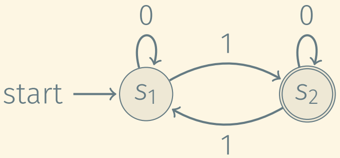
Tabular representation:

δ	0	1
$\to s_1$	s_1	s_2
$*s_2$	s_2	s_1

Graph representation:



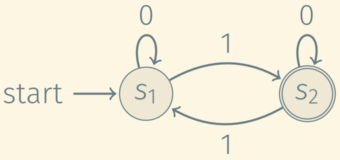
THE LANGUAGE DECIDED BY A DFA (1)



THE LANGUAGE DECIDED BY A DFA (1)

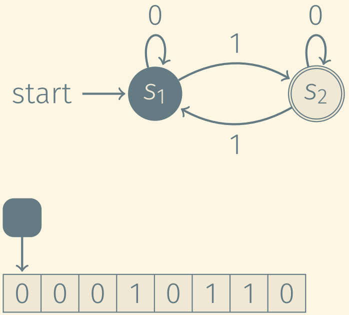
Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



Acceptance/rejection of a string: Intuition

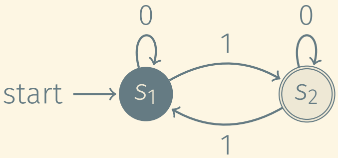
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.

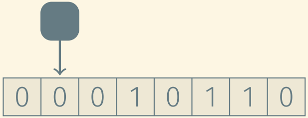


THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.

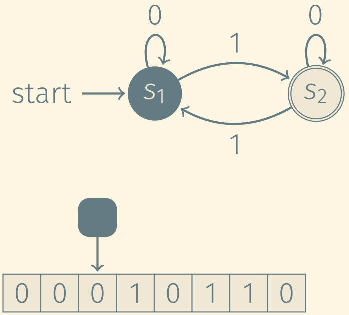




THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

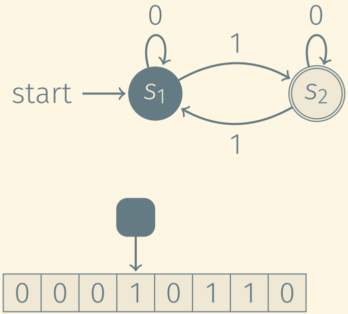
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



THE LANGUAGE DECIDED BY A DFA (1)

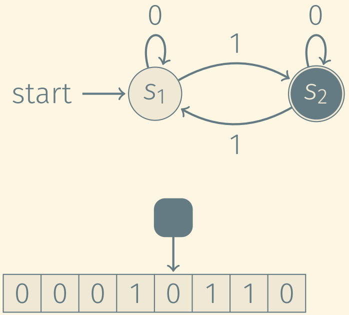
Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



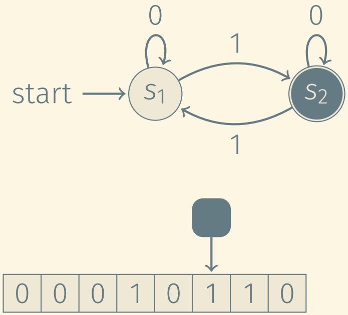
Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



Acceptance/rejection of a string: Intuition

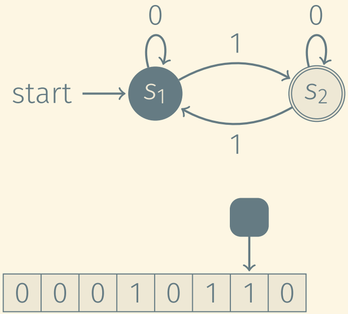
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

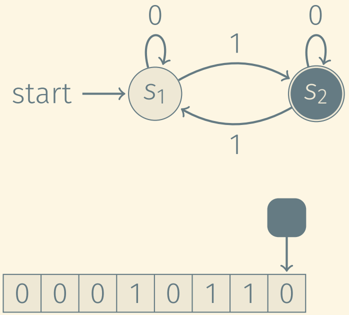
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

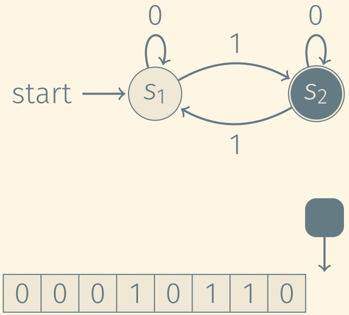
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

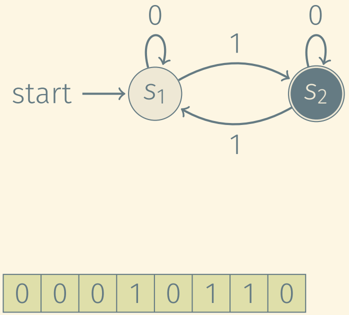
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

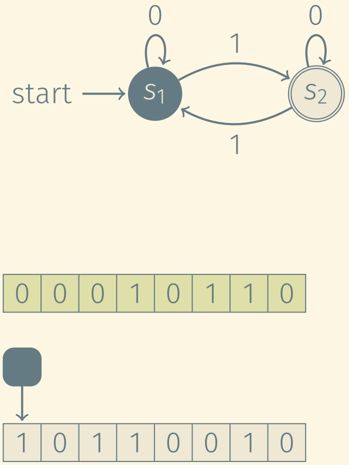
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



THE LANGUAGE DECIDED BY A DFA (1)

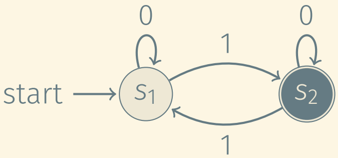
Acceptance/rejection of a string: Intuition

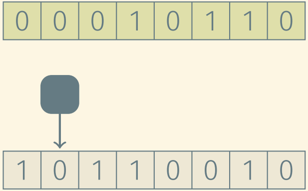
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.

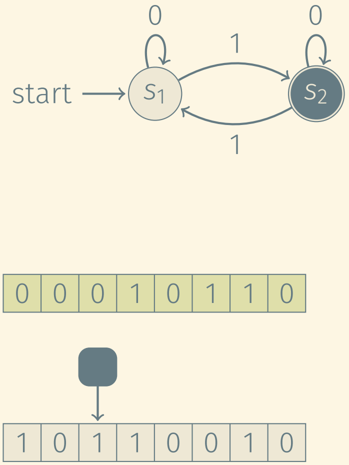




THE LANGUAGE DECIDED BY A DFA (1)

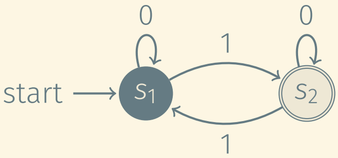
Acceptance/rejection of a string: Intuition

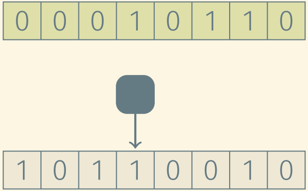
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.

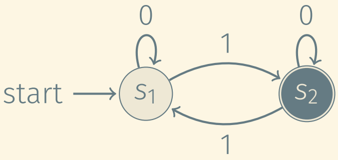


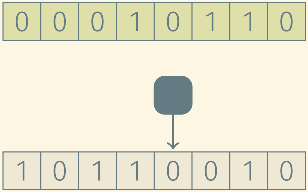


THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

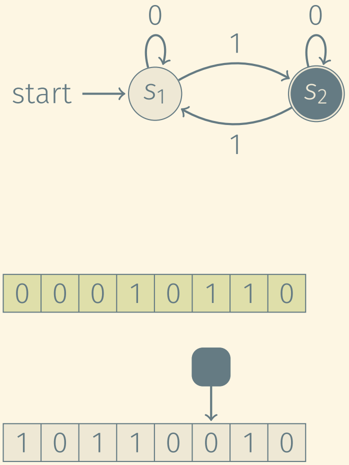
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.





Acceptance/rejection of a string: Intuition

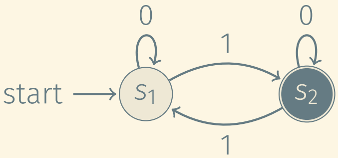
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.

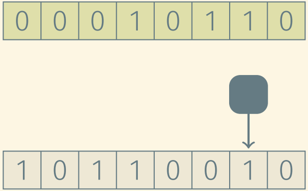


THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

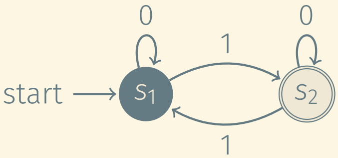
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.

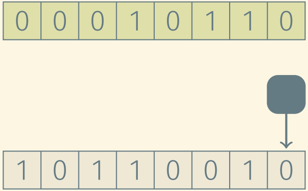




Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.

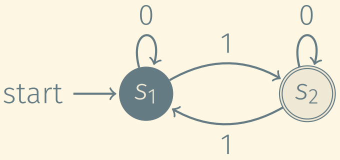


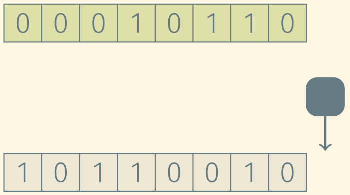


THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

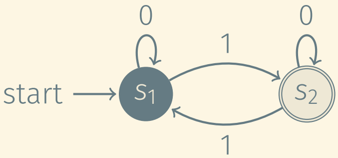
A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.





Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.



0	0	0	1	0	1	1	0

1	0	1	1	0	0	1	0

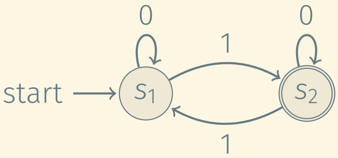
THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.

Language decided by a DFA

$\mathcal{L}(D) = \{\sigma \in \Sigma^* \mid D \text{ accepts } \sigma\}$



0	0	0	1	0	1	1	0

1	0	1	1	0	0	1	0

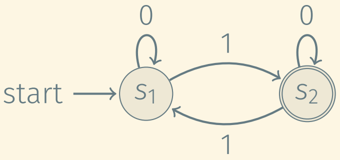
THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.

Language decided by a DFA

$\mathcal{L}(D) = \{\sigma \in \Sigma^* \mid D \text{ accepts } \sigma\}$



0	0	0	1	0	1	1	0

1	0	1	1	0	0	1	0

This DFA decides the language of all binary strings with ...

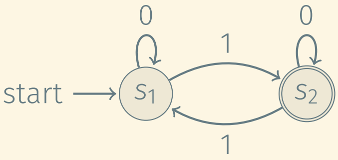
THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.

Language decided by a DFA

$\mathcal{L}(D) = \{\sigma \in \Sigma^* \mid D \text{ accepts } \sigma\}$



0	0	0	1	0	1	1	0

1	0	1	1	0	0	1	0

This DFA decides the language of all binary strings with an odd number of 1s.

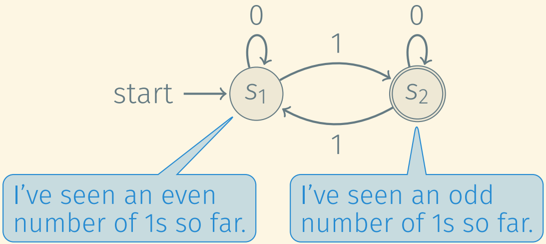
THE LANGUAGE DECIDED BY A DFA (1)

Acceptance/rejection of a string: Intuition

A DFA $D = (S, \Sigma, \delta, s_0, F)$ accepts a string $\sigma \in \Sigma^*$ if, after starting in state s_0 and reading σ, it finishes in an accepting state. Otherwise, it rejects σ.

Language decided by a DFA

$\mathcal{L}(D) = \{\sigma \in \Sigma^* \mid D \text{ accepts } \sigma\}$



0	0	0	1	0	1	1	0

1	0	1	1	0	0	1	0

This DFA decides the language of all binary strings with an odd number of 1s.

THE LANGUAGE DECIDED BY A DFA (2)

A transition function for strings

A transition function for strings

$\delta^* : S \times \Sigma^* \to S$

THE LANGUAGE DECIDED BY A DFA (2)

A transition function for strings

$\delta^* : S \times \Sigma^* \to S$

$\delta^*(s, \sigma)$: the state reached after consuming σ if we start in state s.

THE LANGUAGE DECIDED BY A DFA (2)

A transition function for strings

$\delta^* : S \times \Sigma^* \to S$

$\delta^*(s, \sigma)$: the state reached after consuming σ if we start in state s.

- $\delta^*(s, \epsilon) = s$

THE LANGUAGE DECIDED BY A DFA (2)

A transition function for strings

$\delta^* : S \times \Sigma^* \to S$

$\delta^*(s, \sigma)$: the state reached after consuming σ if we start in state s.

- $\delta^*(s, \epsilon) = s$
- $\delta^*(s, x\sigma) = \delta^*(\delta(s, x), \sigma)$

A transition function for strings

$\delta^* : S \times \Sigma^* \rightarrow S$

$\delta^*(s, \sigma)$: the state reached after consuming σ if we start in state s.

- $\delta^*(s, \epsilon) = s$
- $\delta^*(s, x\sigma) = \delta^*(\delta(s, x), \sigma)$
  $\delta^*(s, x_1x_2 \ldots x_n) = \delta^*(\delta(s, x_1), x_2x_3 \ldots x_n)$

THE LANGUAGE DECIDED BY A DFA (2)

A transition function for strings

$\delta^* : S \times \Sigma^* \longrightarrow S$

$\delta^*(s, \sigma)$: the state reached after consuming σ if we start in state s.

- $\delta^*(s, \epsilon) = s$
- $\delta^*(s, x\sigma) = \delta^*(\delta(s, x), \sigma)$
  $\delta^*(s, x_1 x_2 \ldots x_n) = \delta^*(\delta(s, x_1), x_2 x_3 \ldots x_n)$

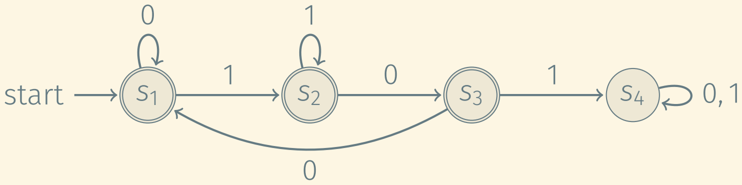
Language decided by a DFA

For a DFA $D = (S, \Sigma, \delta, s_0, F)$,

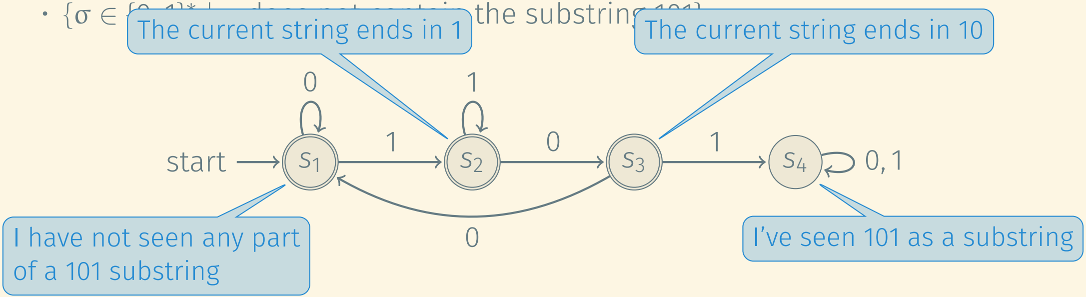
$$\mathcal{L}(D) = \{\sigma \in \Sigma^* \mid \delta^*(s_0, \sigma) \in F\}.$$

EXAMPLES OF DFA

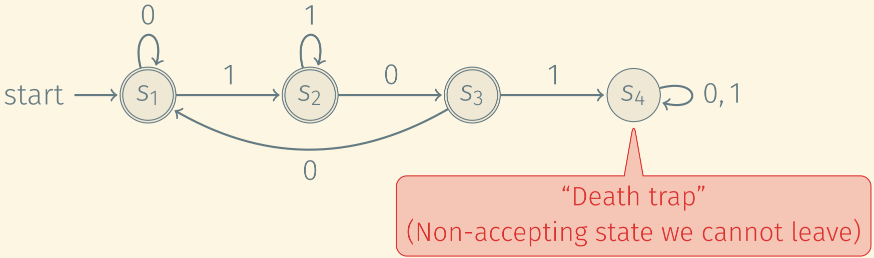
- $\{\sigma \in \{0,1\}^* \mid \sigma \text{ does not contain the substring } 101\}$



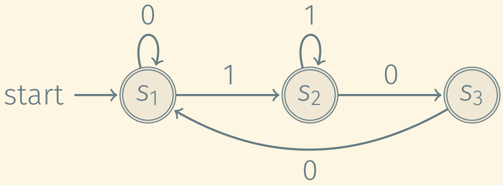
- $\{\sigma \in \{0,1\}^* \mid \sigma$ does not contain the substring $101\}$



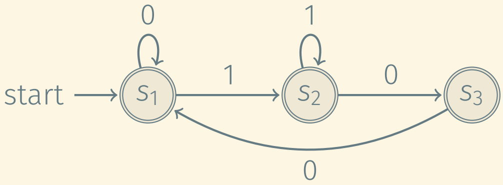
- $\{\sigma \in \{0,1\}^* \mid \sigma$ does not contain the substring 101$\}$



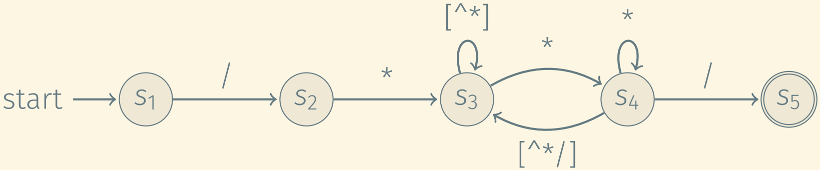
- $\{\sigma \in \{0,1\}^* \mid \sigma$ does not contain the substring $101\}$



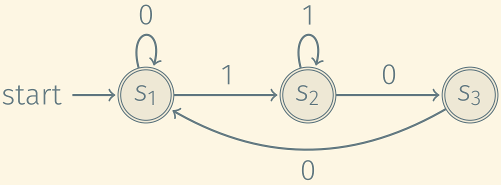
- $\{\sigma \in \{0,1\}^* \mid \sigma$ does not contain the substring 101$\}$



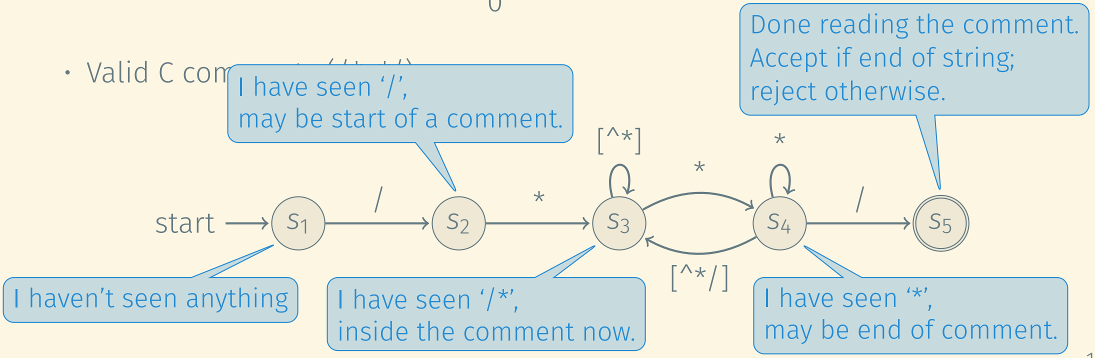
- Valid C comments (/*...*/)



- $\{\sigma \in \{0,1\}^* \mid \sigma \text{ does not contain the substring } 101\}$



- Valid C com



ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA
- Building a scanner
  - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
  - Minimizing the DFA
- Limitations of regular languages

ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA
- Building a scanner
  - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
  - Minimizing the DFA
- Limitations of regular languages

NON-DETERMINISTIC FINITE AUTOMATON (NFA) (1)

For a DFA $D = (S, \Sigma, \delta, s_0, F)$, reading a string σ puts the DFA into a unique state $\delta^*(s_0, \sigma)$. That's why it's called "deterministic".

NON-DETERMINISTIC FINITE AUTOMATON (NFA) (1)

For a DFA $D = (S, \Sigma, \delta, s_0, F)$, reading a string σ puts the DFA into a unique state $\delta^*(s_0, \sigma)$. That's why it's called "deterministic".

A non-deterministic finite automaton (NFA) has the ability to choose between multiple states to transition to after reading a character and may even "spontaneously" transition to a new state without reading anything.

NON-DETERMINISTIC FINITE AUTOMATON (NFA) (1)

For a DFA $D = (S, \Sigma, \delta, s_0, F)$, reading a string σ puts the DFA into a unique state $\delta^*(s_0, \sigma)$. That's why it's called "deterministic".

A non-deterministic finite automaton (NFA) has the ability to choose between multiple states to transition to after reading a character and may even "spontaneously" transition to a new state without reading anything.

$\Rightarrow \delta^*(s_0, \sigma)$ is potentially one of many states.

NON-DETERMINISTIC FINITE AUTOMATON (NFA) (1)

For a DFA $D = (S, \Sigma, \delta, s_0, F)$, reading a string σ puts the DFA into a unique state $\delta^*(s_0, \sigma)$. That's why it's called "deterministic".

A non-deterministic finite automaton (NFA) has the ability to choose between multiple states to transition to after reading a character and may even "spontaneously" transition to a new state without reading anything.

$\Rightarrow$ $\delta^*(s_0, \sigma)$ is potentially one of many states.
Formally, $\delta^*(s_0, \sigma)$ is a set of states.

For a DFA $D = (S, \Sigma, \delta, s_0, F)$, reading a string σ puts the DFA into a unique state $\delta^*(s_0, \sigma)$. That's why it's called "deterministic".

A non-deterministic finite automaton (NFA) has the ability to choose between multiple states to transition to after reading a character and may even "spontaneously" transition to a new state without reading anything.

$\Rightarrow$ $\delta^*(s_0, \sigma)$ is potentially one of many states.
Formally, $\delta^*(s_0, \sigma)$ is a set of states.

An NFA $N = (S, \Sigma, \delta, s_0, F)$ accepts σ if $\delta^*(s_0, \sigma) \cap F \neq \emptyset$.
(N has the ability to reach an accepting state while reading σ, assuming it makes the right choices.)

NON-DETERMINISTIC FINITE AUTOMATON (NFA) (2)

Definition:
Non-deterministic finite automaton (NFA)

A tuple $D = (S, \Sigma, \delta, s_0, F)$:

- Set of states S
- Finite alphabet Σ
- Transition function $\delta : S \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^S$
- Initial state $s_0 \in S$
- Set of final states $F \subseteq S$

NON-DETERMINISTIC FINITE AUTOMATON (NFA) (2)

Definition:
Non-deterministic finite automaton (NFA)

A tuple $D = (S, \Sigma, \delta, s_0, F)$:

- Set of states S
- Finite alphabet Σ
- Transition function $\boxed{\delta : S \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^S}$
- Initial state $s_0 \in S$
- Set of final states $F \subseteq S$

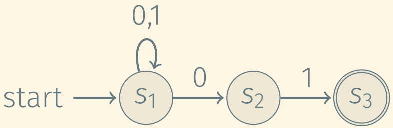
NON-DETERMINISTIC FINITE AUTOMATON (NFA) (2)

Definition:
Non-deterministic finite automaton (NFA)

A tuple $D = (S, \Sigma, \delta, s_0, F)$:

- Set of states S
- Finite alphabet Σ
- Transition function $\boxed{\delta : S \times (\Sigma \cup \{\epsilon\}) \to 2^S}$
- Initial state $s_0 \in S$
- Set of final states $F \subseteq S$

δ	0	1	ϵ
$\to s_1$	$\{s_1, s_2\}$	$\{s_1\}$	$\emptyset$
s_2	$\emptyset$	$\{s_3\}$	$\emptyset$
$*s_3$	$\emptyset$	$\emptyset$	$\emptyset$

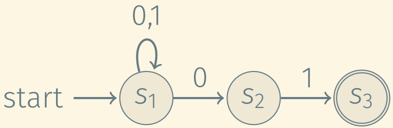


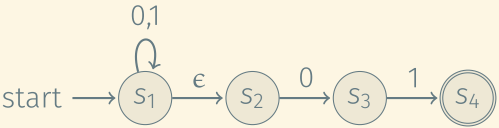
Definition:
Non-deterministic finite automaton (NFA)

A tuple $D = (S, \Sigma, \delta, s_0, F)$:

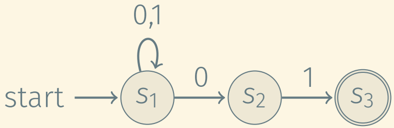
- Set of states S
- Finite alphabet Σ
- Transition function $\boxed{\delta : S \times (\Sigma \cup \{\epsilon\}) \to 2^S}$
- Initial state $s_0 \in S$
- Set of final states $F \subseteq S$

δ	0	1	ϵ
$\to s_1$	$\{s_1\}$	$\{s_1\}$	$\{s_2\}$
s_2	$\{s_3\}$	$\emptyset$	$\emptyset$
s_3	$\emptyset$	$\{s_4\}$	$\emptyset$
$*s_4$	$\emptyset$	$\emptyset$	$\emptyset$

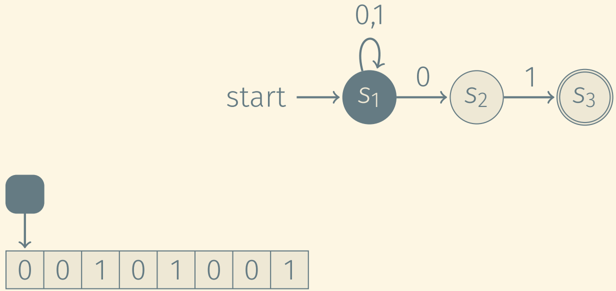




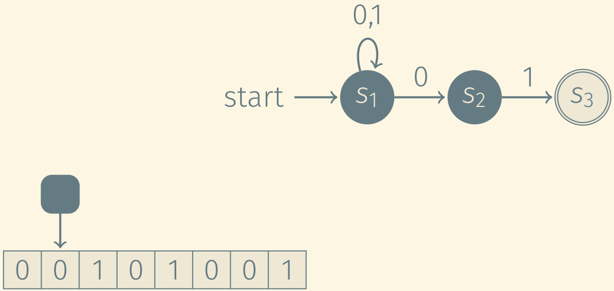
THE LANGUAGE DECIDED BY AN NFA (1)

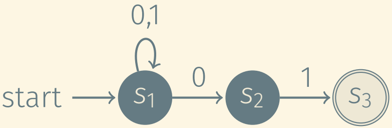


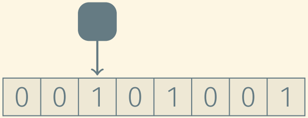
THE LANGUAGE DECIDED BY AN NFA (1)



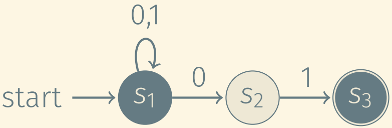
THE LANGUAGE DECIDED BY AN NFA (1)

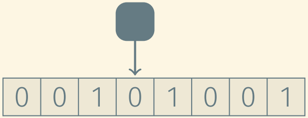



0,1
start
s_1
0
s_2
1
s_3

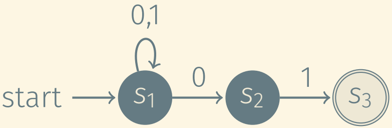

0 0 1 0 1 0 0 1

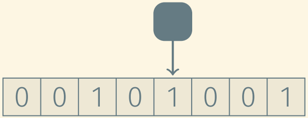
THE LANGUAGE DECIDED BY AN NFA (1)



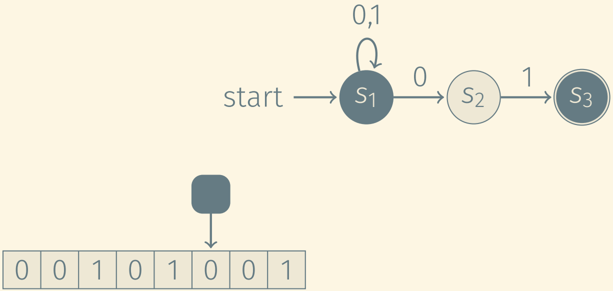


THE LANGUAGE DECIDED BY AN NFA (1)

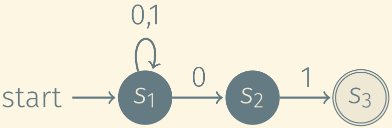


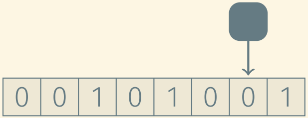


THE LANGUAGE DECIDED BY AN NFA (1)

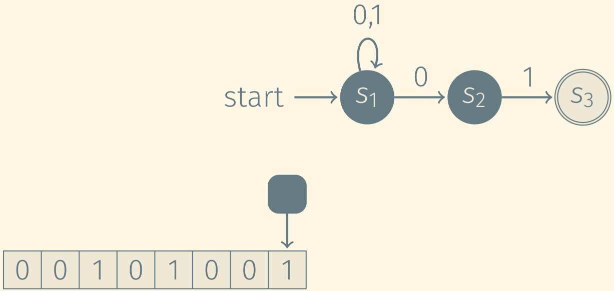


THE LANGUAGE DECIDED BY AN NFA (1)

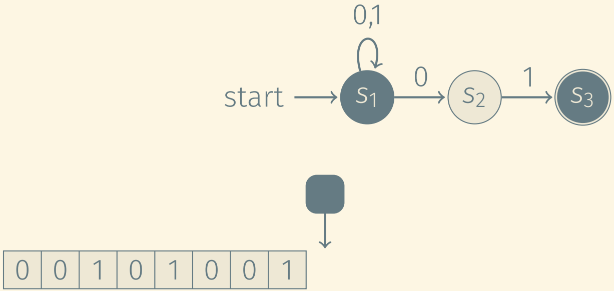




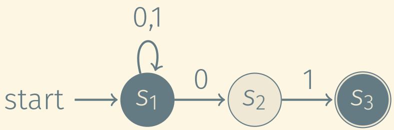
THE LANGUAGE DECIDED BY AN NFA (1)



THE LANGUAGE DECIDED BY AN NFA (1)

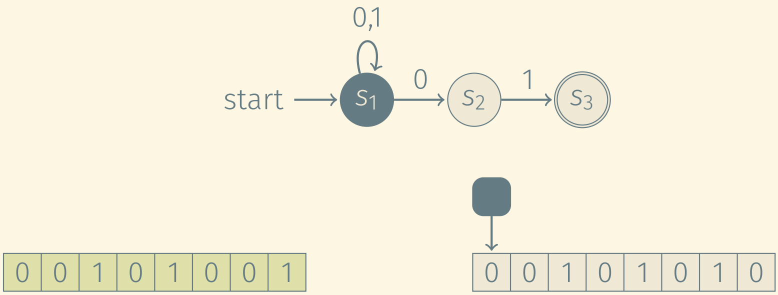


THE LANGUAGE DECIDED BY AN NFA (1)

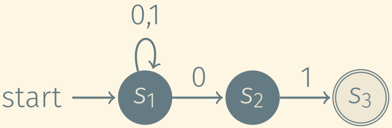


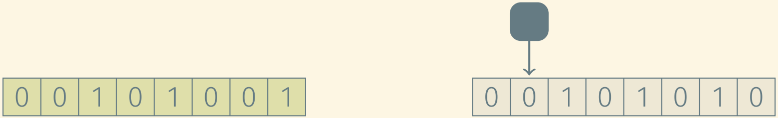
0	0	1	0	1	0	0	1

THE LANGUAGE DECIDED BY AN NFA (1)

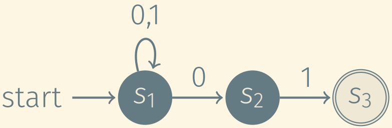


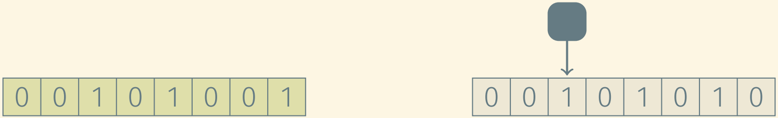
THE LANGUAGE DECIDED BY AN NFA (1)



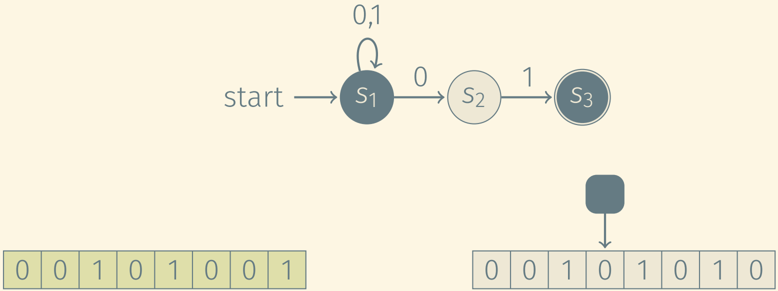


THE LANGUAGE DECIDED BY AN NFA (1)

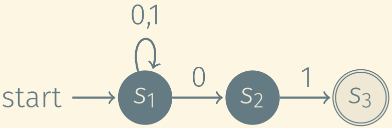


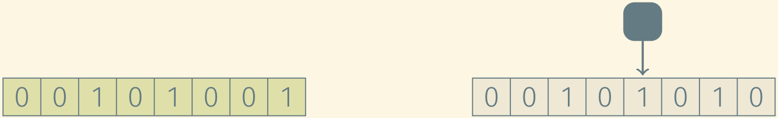


THE LANGUAGE DECIDED BY AN NFA (1)

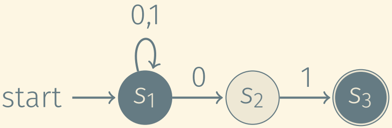


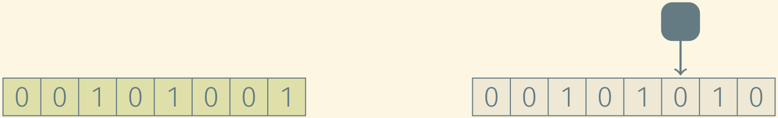
THE LANGUAGE DECIDED BY AN NFA (1)



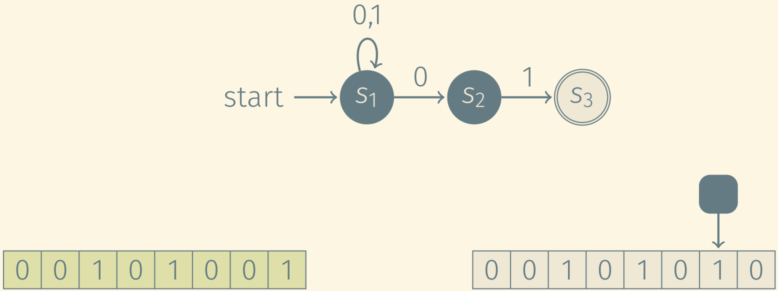


THE LANGUAGE DECIDED BY AN NFA (1)

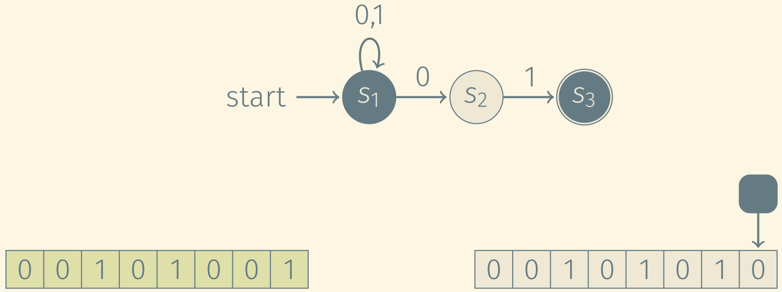


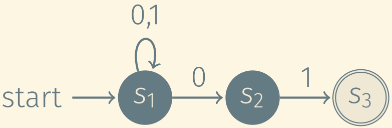


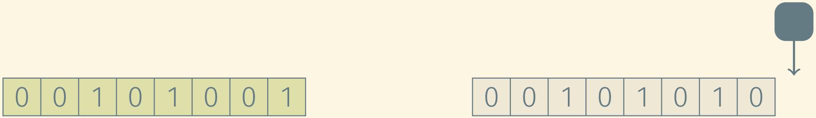
THE LANGUAGE DECIDED BY AN NFA (1)



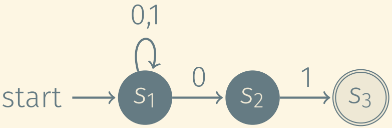
THE LANGUAGE DECIDED BY AN NFA (1)




0,1
start
0
1
s_1
s_2
s_3


0 0 1 0 1 0 0 1
0 0 1 0 1 0 1 0

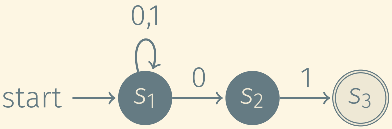
THE LANGUAGE DECIDED BY AN NFA (1)



0	0	1	0	1	0	0	1

0	0	1	0	1	0	1	0

THE LANGUAGE DECIDED BY AN NFA (1)

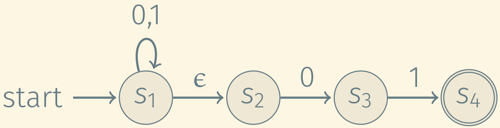


0	0	1	0	1	0	0	1

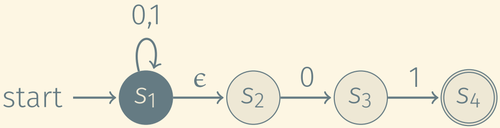
0	0	1	0	1	0	1	0

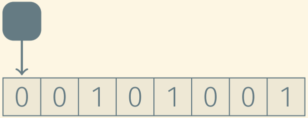
$$\mathcal{L}(N) = \mathcal{L}((0|1)*01)$$

THE LANGUAGE DECIDED BY AN NFA (2)

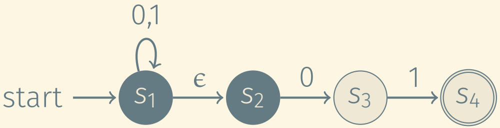


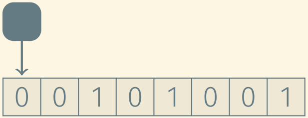
THE LANGUAGE DECIDED BY AN NFA (2)



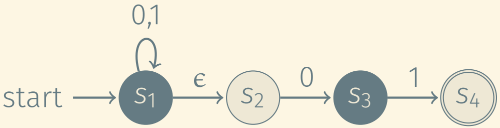


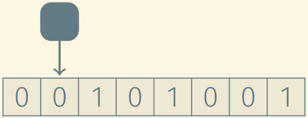
THE LANGUAGE DECIDED BY AN NFA (2)



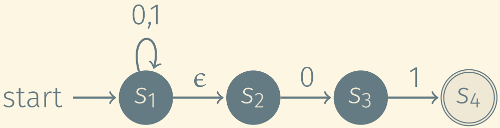


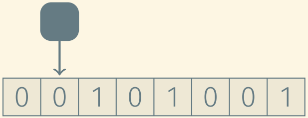
THE LANGUAGE DECIDED BY AN NFA (2)



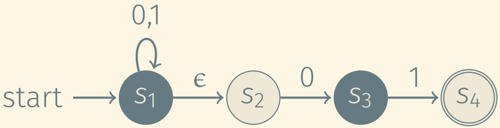


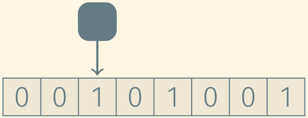
THE LANGUAGE DECIDED BY AN NFA (2)



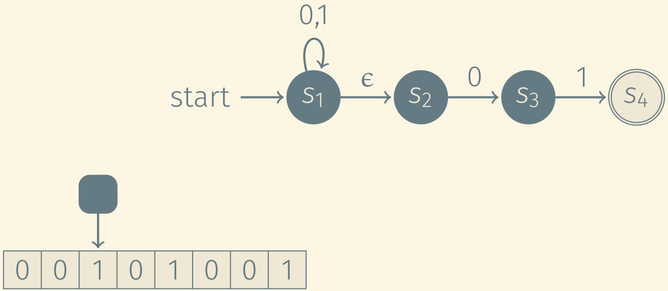


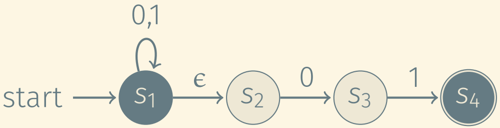
THE LANGUAGE DECIDED BY AN NFA (2)

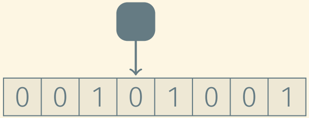




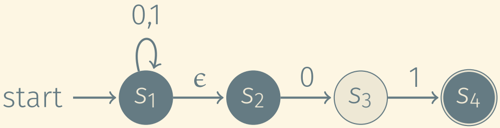
THE LANGUAGE DECIDED BY AN NFA (2)

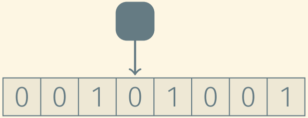



0,1
start
s_1
ϵ
s_2
0
s_3
1
s_4

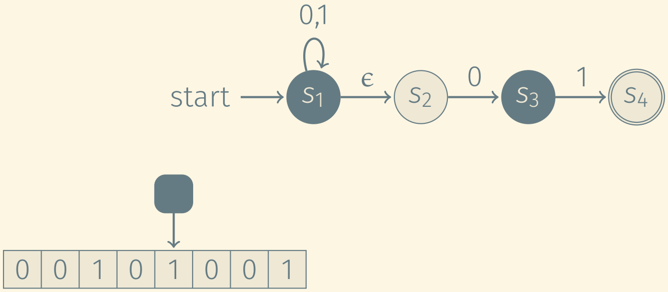

0 0 1 0 1 0 0 1

THE LANGUAGE DECIDED BY AN NFA (2)

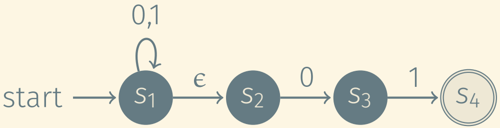


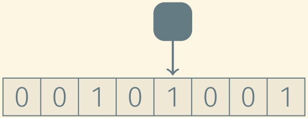


THE LANGUAGE DECIDED BY AN NFA (2)

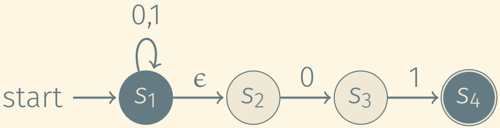


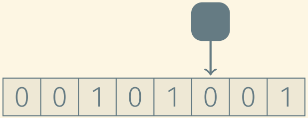
THE LANGUAGE DECIDED BY AN NFA (2)



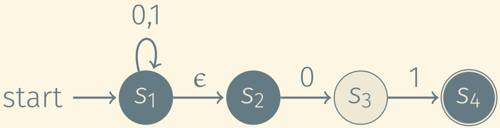


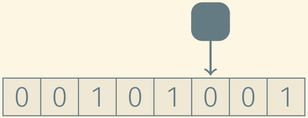
THE LANGUAGE DECIDED BY AN NFA (2)

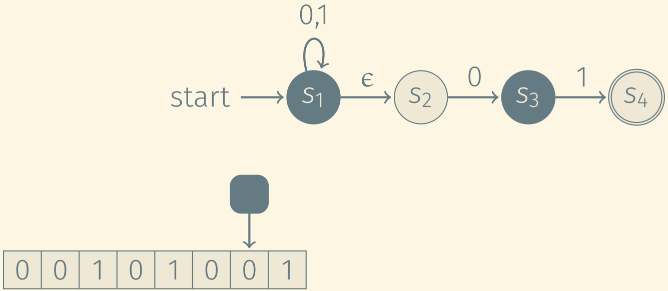




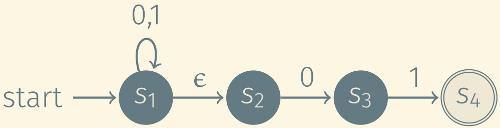
THE LANGUAGE DECIDED BY AN NFA (2)

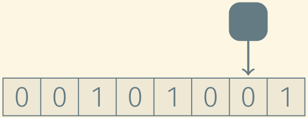




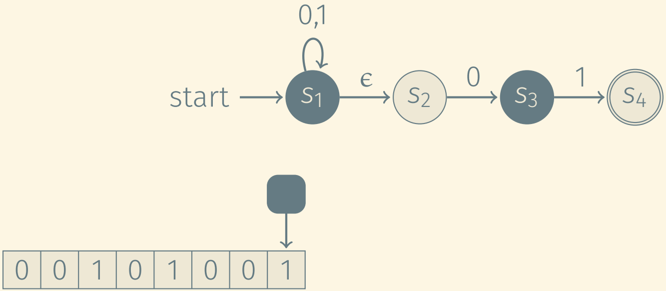

0,1
start
s_1
ϵ
s_2
0
s_3
1
s_4
0 0 1 0 1 0 0 1

THE LANGUAGE DECIDED BY AN NFA (2)

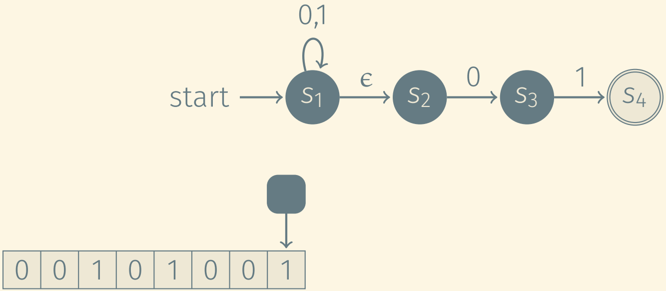




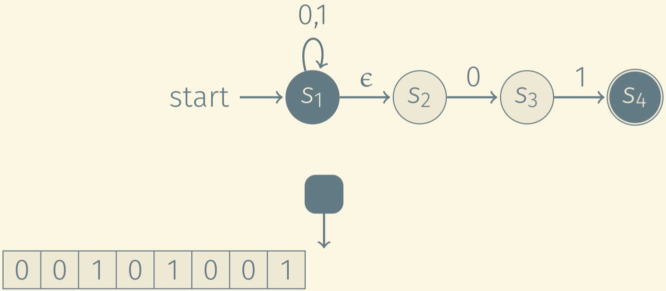
THE LANGUAGE DECIDED BY AN NFA (2)



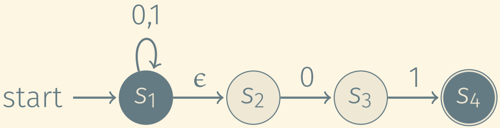
THE LANGUAGE DECIDED BY AN NFA (2)



THE LANGUAGE DECIDED BY AN NFA (2)

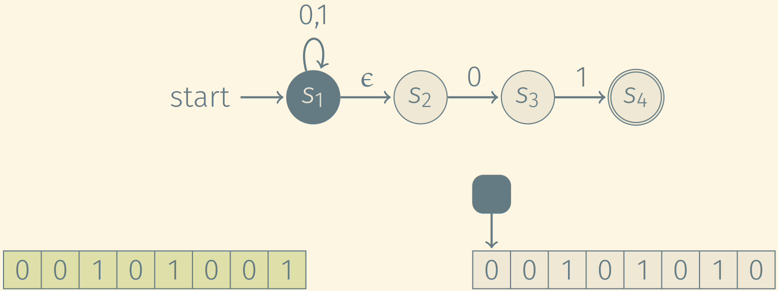


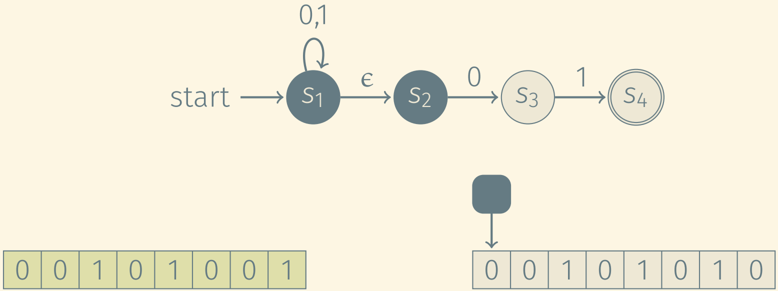
THE LANGUAGE DECIDED BY AN NFA (2)



0	0	1	0	1	0	0	1

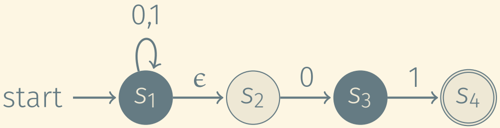
THE LANGUAGE DECIDED BY AN NFA (2)

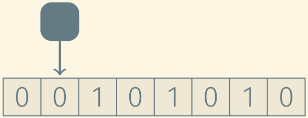




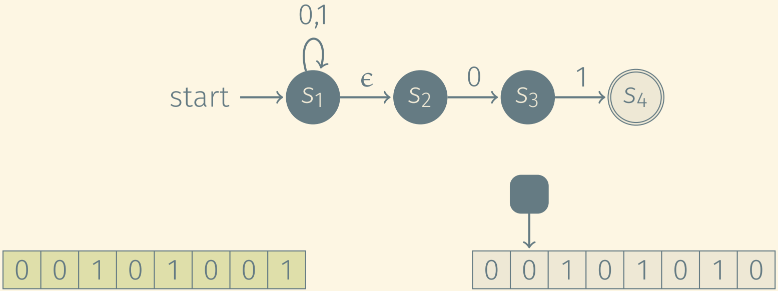
0,1
start
s_1
ϵ
s_2
0
s_3
1
s_4
0 0 1 0 1 0 0 1
0 0 1 0 1 0 1 0

THE LANGUAGE DECIDED BY AN NFA (2)

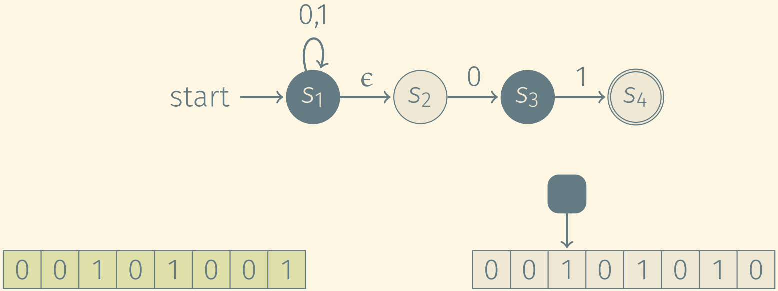




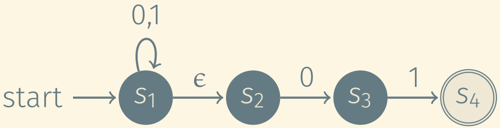
THE LANGUAGE DECIDED BY AN NFA (2)

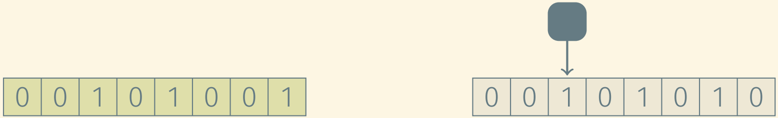


THE LANGUAGE DECIDED BY AN NFA (2)

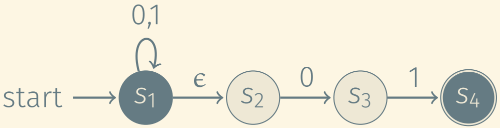


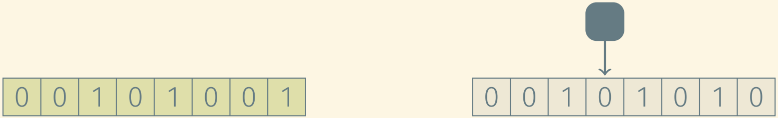
THE LANGUAGE DECIDED BY AN NFA (2)



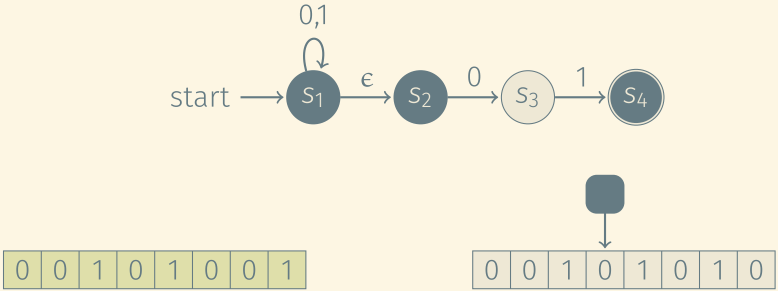


THE LANGUAGE DECIDED BY AN NFA (2)

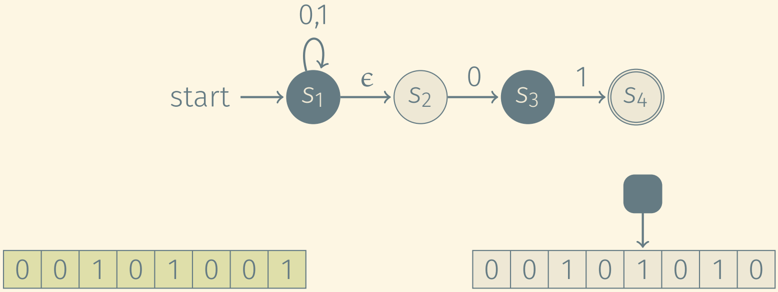




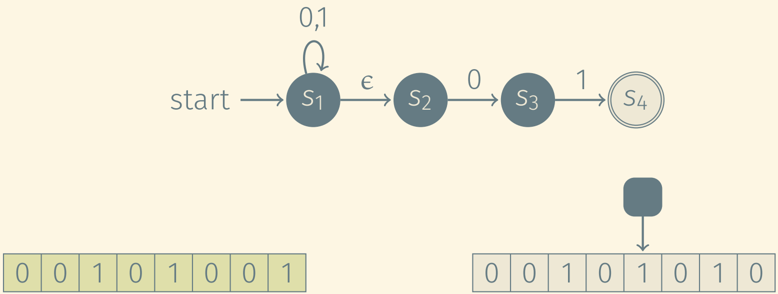
THE LANGUAGE DECIDED BY AN NFA (2)



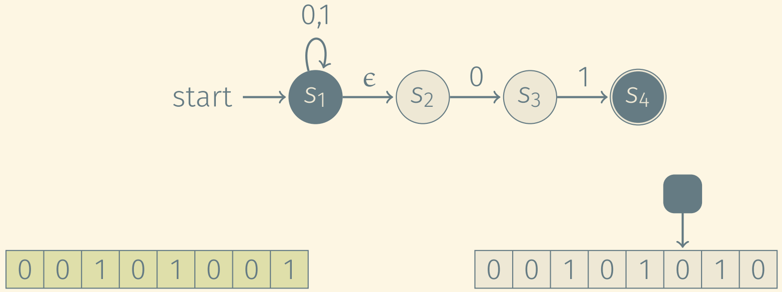
THE LANGUAGE DECIDED BY AN NFA (2)



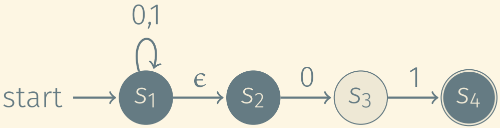
THE LANGUAGE DECIDED BY AN NFA (2)

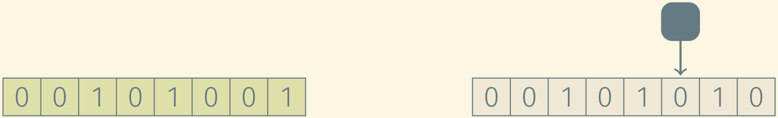


THE LANGUAGE DECIDED BY AN NFA (2)

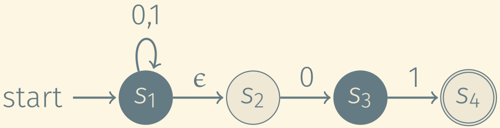


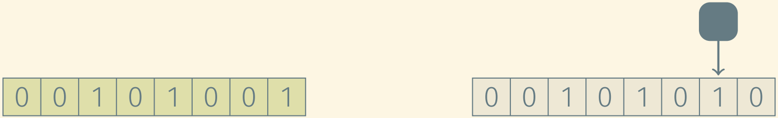
THE LANGUAGE DECIDED BY AN NFA (2)



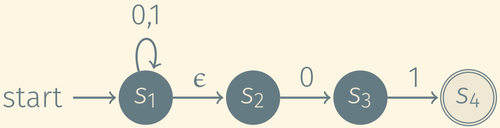


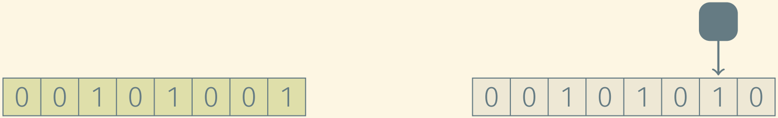
THE LANGUAGE DECIDED BY AN NFA (2)



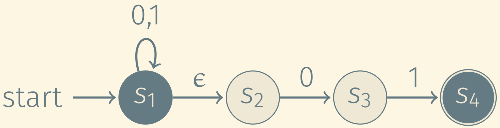


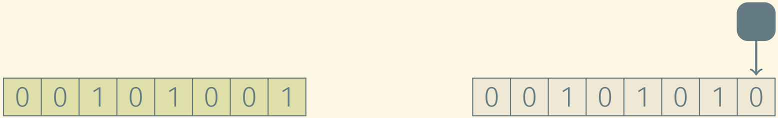
THE LANGUAGE DECIDED BY AN NFA (2)



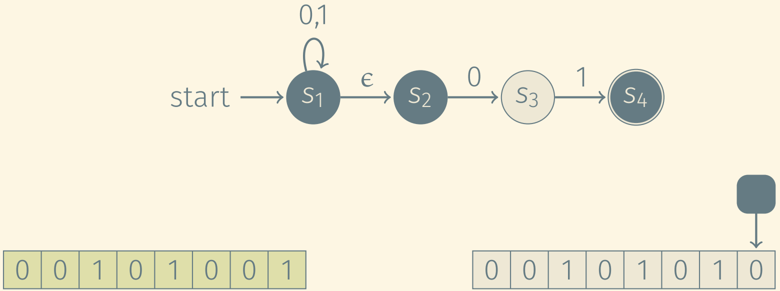


THE LANGUAGE DECIDED BY AN NFA (2)

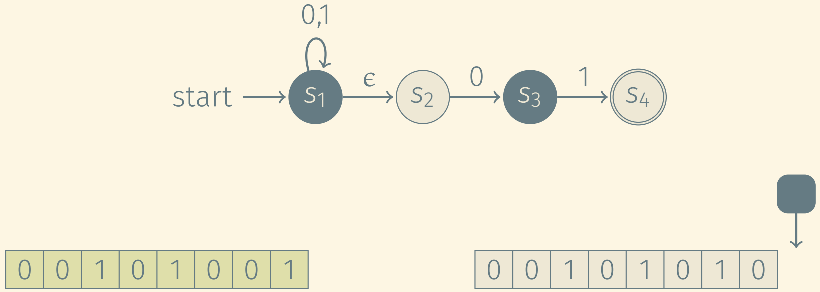




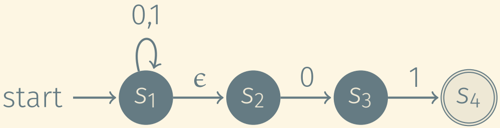
THE LANGUAGE DECIDED BY AN NFA (2)

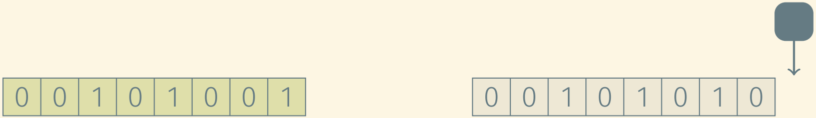


THE LANGUAGE DECIDED BY AN NFA (2)

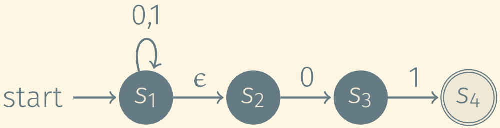


THE LANGUAGE DECIDED BY AN NFA (2)





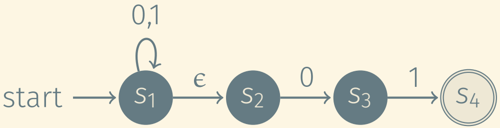
THE LANGUAGE DECIDED BY AN NFA (2)



0	0	1	0	1	0	0	1

0	0	1	0	1	0	1	0

THE LANGUAGE DECIDED BY AN NFA (2)



0	0	1	0	1	0	0	1

0	0	1	0	1	0	1	0

$$\mathcal{L}(N) = \mathcal{L}((0|1)*01)$$

THE LANGUAGE DECIDED BY AN NFA (3)

THE LANGUAGE DECIDED BY AN NFA (3)

Language decided by a DFA

For a DFA $D = (S, \Sigma, \delta, s_0, F)$,

$$\mathcal{L}(D) = \{\sigma \in \Sigma^* \mid \delta^*(s_0, \sigma) \in F\}.$$

THE LANGUAGE DECIDED BY AN NFA (3)

Language decided by a DFA

For a DFA $D = (S, \Sigma, \delta, s_0, F)$,

$$\mathcal{L}(D) = \{\sigma \in \Sigma^* \mid \delta^*(s_0, \sigma) \in F\}.$$

What should this definition look like for an NFA?

Language decided by a DFA

For a DFA $D = (S, \Sigma, \delta, s_0, F)$,

$$\mathcal{L}(D) = \{\sigma \in \Sigma^* \mid \delta^*(s_0, \sigma) \in F\}.$$

What should this definition look like for an NFA?

- What *is* $\delta^*(s_0, \sigma)$?

- How should it be related to F for N to accept σ?

THE LANGUAGE DECIDED BY AN NFA (3)

Language decided by a DFA

For a DFA $D = (S, \Sigma, \delta, s_0, F)$,

$$\mathcal{L}(D) = \{\sigma \in \Sigma^* \mid \delta^*(s_0, \sigma) \in F\}.$$

What should this definition look like for an NFA?

- What *is* $\delta^*(s_0, \sigma)$?

  The set of states reachable from s_0 by reading σ.

- How should it be related to F for N to accept σ?

Language decided by an NFA

For an NFA $N = (S, \Sigma, \delta, s_0, F)$,

$$\mathcal{L}(N) = \{\sigma \in \Sigma^* \mid \delta^*(s_0, \sigma) \cap F \neq \emptyset\}.$$

What should this definition look like for an NFA?

- What *is* $\delta^*(s_0, \sigma)$?

  The set of states reachable from s_0 by reading σ.

- How should it be related to F for N to accept σ?

ϵ-CLOSURE

Definition: ϵ-Closure

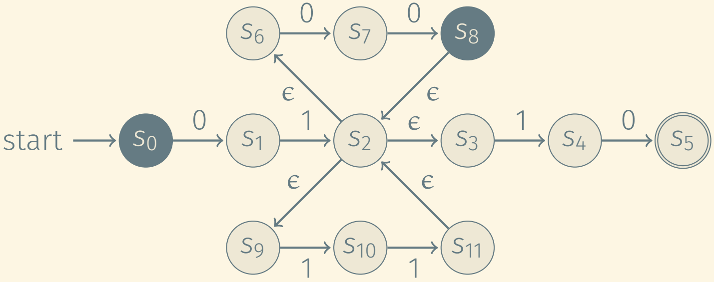
For some subset $S' \subseteq S$ of states, $\text{ECLOSE}(S')$ is the set of all states that can be reached from states in S' using only ϵ-transitions.

Formally, $\text{ECLOSE}(S')$ is the smallest superset $\text{ECLOSE}(S') \supseteq S'$ such that $\delta(s, \epsilon) \subseteq \text{ECLOSE}(S')$ for all $s \in \text{ECLOSE}(S')$.

Definition: ϵ-Closure

For some subset $S' \subseteq S$ of states, ECLOSE(S') is the set of all states that can be reached from states in S' using only ϵ-transitions.

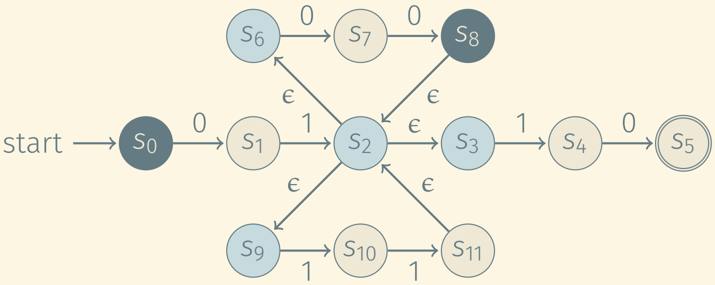
Formally, ECLOSE(S') is the smallest superset ECLOSE(S') $\supseteq S'$ such that $\delta(s, \epsilon) \subseteq$ ECLOSE(S') for all $s \in$ ECLOSE(S').



Definition: ϵ-Closure

For some subset $S' \subseteq S$ of states, ECLOSE(S') is the set of all states that can be reached from states in S' using only ϵ-transitions.

Formally, ECLOSE(S') is the smallest superset ECLOSE$(S') \supseteq S'$ such that $\delta(s, \epsilon) \subseteq$ ECLOSE(S') for all $s \in$ ECLOSE(S').



A transition function for strings

$\delta^*(s, \sigma) =$ the set of states reachable from s by reading σ.

THE LANGUAGE DECIDED BY AN NFA (4)

A transition function for strings

$\delta^*(s, \sigma)$ = the set of states reachable from s by reading σ.

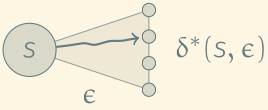
- $\delta^*(s, \epsilon) =$

THE LANGUAGE DECIDED BY AN NFA (4)

A transition function for strings

$\delta^*(s, \sigma)$ = the set of states reachable from s by reading σ.

- $\delta^*(s, \epsilon) = \mathrm{ECLOSE}(\{s\})$

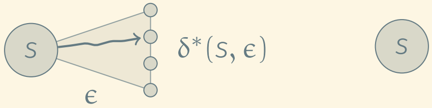


THE LANGUAGE DECIDED BY AN NFA (4)

A transition function for strings

$\delta^*(s, \sigma)$ = the set of states reachable from s by reading σ.

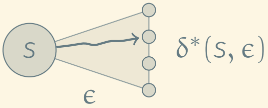
- $\delta^*(s, \epsilon) = \mathrm{ECLOSE}(\{s\})$
- $\delta^*(s, x\sigma) =$

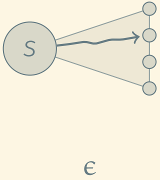


A transition function for strings

$\delta^*(s, \sigma)$ = the set of states reachable from s by reading σ.

- $\delta^*(s, \epsilon) = \mathrm{ECLOSE}(\{s\})$
- $\delta^*(s, x\sigma) = \bigcup_{s_1 \in \mathrm{ECLOSE}(\{s\})}$

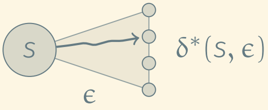


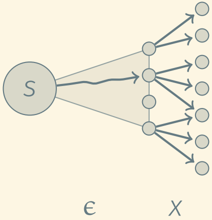


A transition function for strings

$\delta^*(s,\sigma)$ = the set of states reachable from s by reading σ.

- $\delta^*(s,\epsilon) = \mathrm{ECLOSE}(\{s\})$
- $\delta^*(s,x\sigma) = \bigcup_{s_1 \in \mathrm{ECLOSE}(\{s\})} \bigcup_{s_2 \in \delta(s_1,x)}$

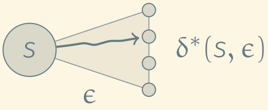


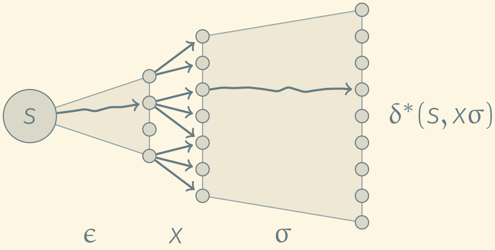


A transition function for strings

$\delta^*(s, \sigma)$ = the set of states reachable from s by reading σ.

- $\delta^*(s, \epsilon) = \text{ECLOSE}(\{s\})$
- $\delta^*(s, x\sigma) = \bigcup_{s_1 \in \text{ECLOSE}(\{s\})} \bigcup_{s_2 \in \delta(s_1, x)} \delta^*(s_2, \sigma)$





ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA
- Building a scanner
    - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
    - Minimizing the DFA
- Limitations of regular languages

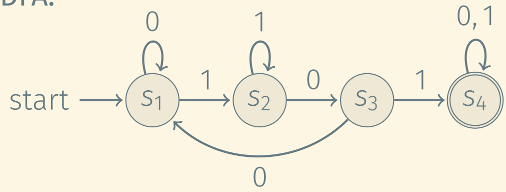
ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA
- Building a scanner
  - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
  - Minimizing the DFA
- Limitations of regular languages

NFA CAN BE MORE CONVENIENT THAN DFA (1)

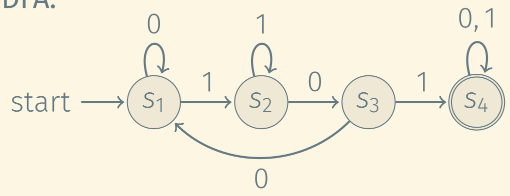
- All binary strings that have 101 as a substring.

DFA:

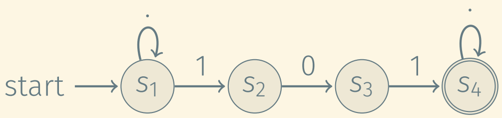


- All binary strings that have 101 as a substring.

DFA:

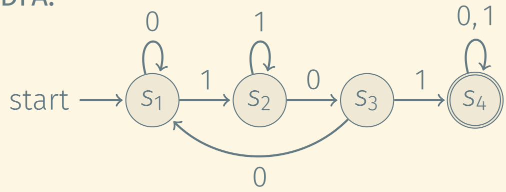


NFA:

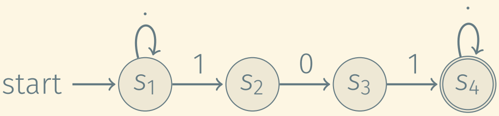


- All binary strings that have 101 as a substring.

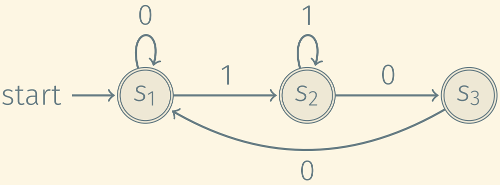
DFA:



NFA:

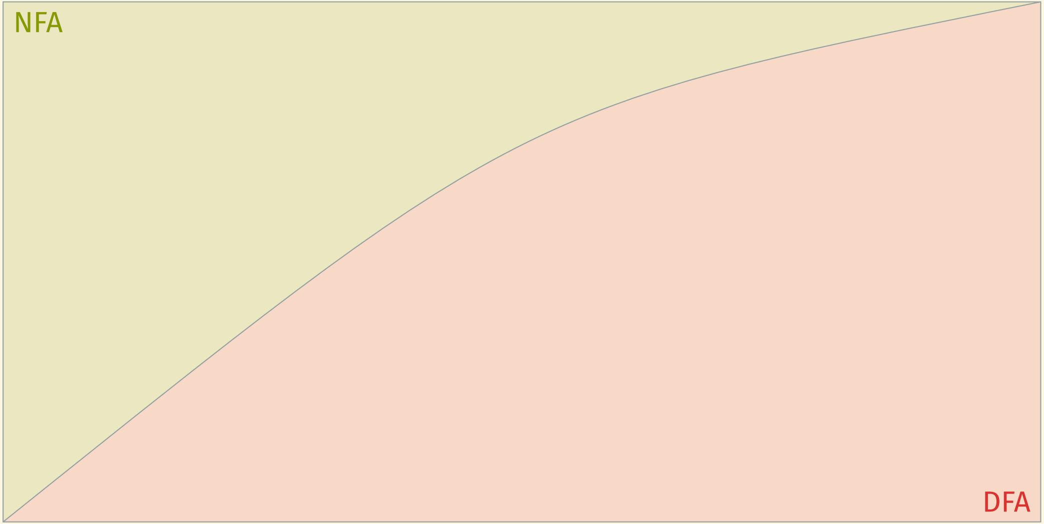


- All binary strings that do not have 101 as a substring.



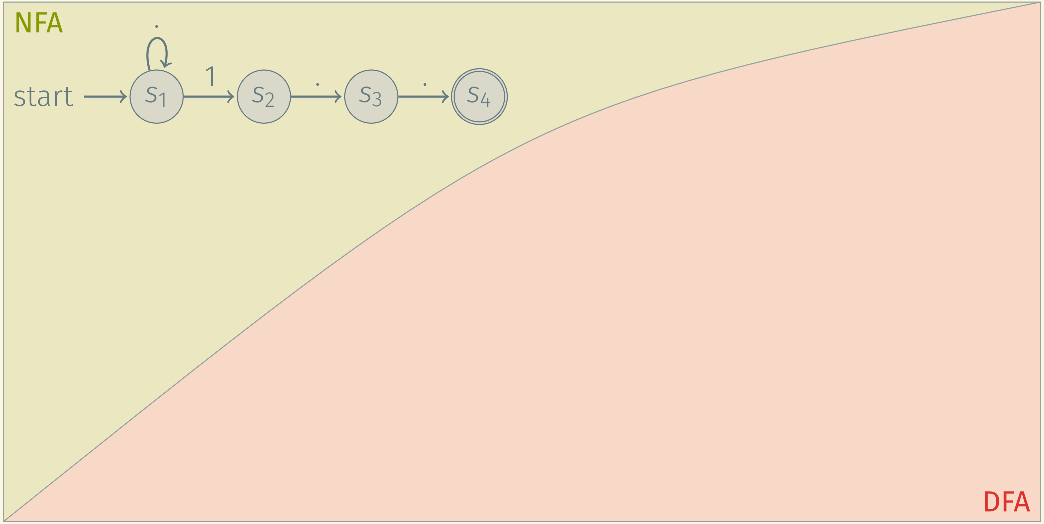
NFA CAN BE MORE CONVENIENT THAN DFA (2)

A more compelling example: $\mathcal{L}(.*1..)$



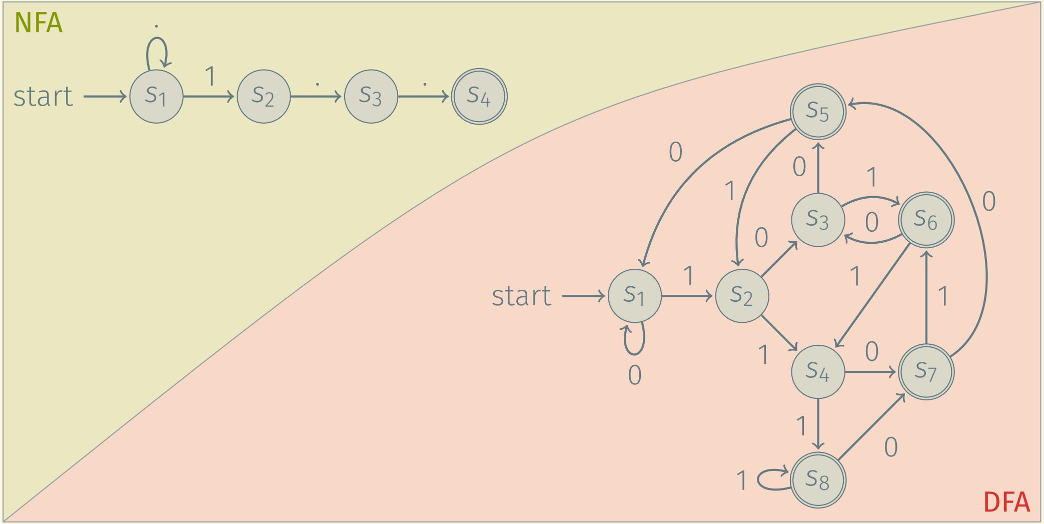
NFA CAN BE MORE CONVENIENT THAN DFA (2)

A more compelling example: $\mathcal{L}(.*1..)$



NFA CAN BE MORE CONVENIENT THAN DFA (2)

A more compelling example: $\mathcal{L}(.*1..)$



NFA CAN BE MORE CONVENIENT THAN DFA (3)

When testing for the presence of patterns, NFA are more convenient than DFA. They only have to guess right where the pattern starts! This does not work for testing for their absence.

NFA CAN BE MORE CONVENIENT THAN DFA (3)

When testing for the presence of patterns, NFA are more convenient than DFA. They only have to guess right where the pattern starts! This does not work for testing for their absence.

Testing for the presence of patterns is the common case in parsing programming languages (keywords, identifiers, ...).

NFA CAN BE MORE CONVENIENT THAN DFA (3)

When testing for the presence of patterns, NFA are more convenient than DFA. They only have to guess right where the pattern starts! This does not work for testing for their absence.

Testing for the presence of patterns is the common case in parsing programming languages (keywords, identifiers, …).

But … computers are not good at guessing!

NFA CAN BE MORE CONVENIENT THAN DFA (3)

When testing for the presence of patterns, NFA are more convenient than DFA. They only have to guess right where the pattern starts! This does not work for testing for their absence.

Testing for the presence of patterns is the common case in parsing programming languages (keywords, identifiers, ...).

But ... computers are not good at guessing!
$\longrightarrow$ We need to construct DFA.

ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA
- Building a scanner
  - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
  - Minimizing the DFA
- Limitations of regular languages

ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA

- Building a scanner
  - Regular expression → NFA → DFA
  - Minimizing the DFA

- Limitations of regular languages

ARE NFA MORE POWERFUL THAN DFA?

No!

No!

Theorem

The following statements are equivalent:

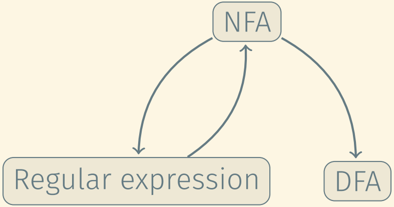
- $\mathcal{L}$ *is a regular language.*
- $\mathcal{L}$ *can be decided by a DFA.*
- $\mathcal{L}$ *can be decided by an NFA.*

No!

Theorem

The following statements are equivalent:

- *$\mathcal{L}$ is a regular language.*
- *$\mathcal{L}$ can be decided by a DFA.*
- *$\mathcal{L}$ can be decided by an NFA.*



Proof outline:

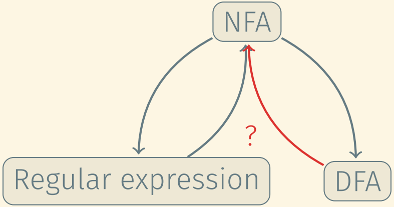
- Given an NFA N, construct a DFA D with $\mathcal{L}(D) = \mathcal{L}(N)$.
- Given a regular expression R, construct an NFA that decides $\mathcal{L}(R)$.
- Given an NFA N, construct a regular expression R with $\mathcal{L}(R) = \mathcal{L}(N)$.

No!

Theorem

The following statements are equivalent:

- *$\mathcal{L}$ is a regular language.*
- *$\mathcal{L}$ can be decided by a DFA.*
- *$\mathcal{L}$ can be decided by an NFA.*



Proof outline:

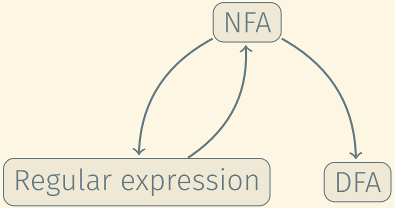
- Given an NFA N, construct a DFA D with $\mathcal{L}(D) = \mathcal{L}(N)$.
- Given a regular expression R, construct an NFA that decides $\mathcal{L}(R)$.
- Given an NFA N, construct a regular expression R with $\mathcal{L}(R) = \mathcal{L}(N)$.

No!

Theorem

The following statements are equivalent:

- *$\mathcal{L}$ is a regular language.*
- *$\mathcal{L}$ can be decided by a DFA.*
- *$\mathcal{L}$ can be decided by an NFA.*



Proof outline:

- Given an NFA N, construct a DFA D with $\mathcal{L}(D) = \mathcal{L}(N)$.
- Given a regular expression R, construct an NFA that decides $\mathcal{L}(R)$.
- Given an NFA N, construct a regular expression R with $\mathcal{L}(R) = \mathcal{L}(N)$.

FROM NFA TO DFA

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (2^S, \Sigma, \gamma, t_0, G)$

- $2^S = \{S' \mid S' \subseteq S\}$ (set of all subsets of S)

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (2^S, \Sigma, \gamma, t_0, G)$

- $2^S = \{S' \mid S' \subseteq S\}$ (set of all subsets of S)

Problem: $|2^S| = 2^{|S|}$

FROM NFA TO DFA

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (2^S, \Sigma, \gamma, t_0, G)$

- $2^S = \{S' \mid S' \subseteq S\}$ (set of all subsets of S)

Problem: $|2^S| = 2^{|S|}$
$\rightarrow$ Can we construct only the subset of states in 2^S that are reachable from t_0?

INITIAL STATE AND ACCEPTING STATES OF THE DFA

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (2^S, \Sigma, \gamma, t_0, G)$

INITIAL STATE AND ACCEPTING STATES OF THE DFA

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (2^S, \Sigma, \gamma, t_0, G)$

Initial state of DFA:

INITIAL STATE AND ACCEPTING STATES OF THE DFA

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (2^S, \Sigma, \gamma, t_0, G)$

Initial state of DFA: The states the NFA can reach without consuming any input.

INITIAL STATE AND ACCEPTING STATES OF THE DFA

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (2^S, \Sigma, \gamma, t_0, G)$

Initial state of DFA: The states the NFA can reach without consuming any input.

$$t_0 = \mathrm{ECLOSE}(\{s_0\})$$

INITIAL STATE AND ACCEPTING STATES OF THE DFA

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (2^S, \Sigma, \gamma, t_0, G)$

Initial state of DFA: The states the NFA can reach without consuming any input.

$$t_0 = \text{ECLOSE}(\{s_0\})$$

Accepting states of the DFA:

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \to D = (2^S, \Sigma, \gamma, t_0, G)$

Initial state of DFA: The states the NFA can reach without consuming any input.

$$t_0 = \mathrm{ECLOSE}(\{s_0\})$$

Accepting states of the DFA: All subsets of S that include an accepting state.

INITIAL STATE AND ACCEPTING STATES OF THE DFA

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \to D = (2^S, \Sigma, \gamma, t_0, G)$

Initial state of DFA: The states the NFA can reach without consuming any input.

$$t_0 = \mathrm{ECLOSE}(\{s_0\})$$

Accepting states of the DFA: All subsets of S that include an accepting state.

$$G = \{S' \subseteq S \mid S' \cap F \neq \emptyset\}$$

TRANSITION FUNCTION OF THE DFA (1)

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (2^S, \Sigma, \gamma, t_0, G)$

TRANSITION FUNCTION OF THE DFA (1)

Idea: Each DFA state represents a set of NFA states the NFA can be in after reading some string.

Trivial construction: $N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (2^S, \Sigma, \gamma, t_0, G)$

We want $\gamma^*(t_0, \sigma) = \gamma^*(\mathrm{ECLOSE}(\{s_0\}), \sigma) = \delta^*(s_0, \sigma)$.

TRANSITION FUNCTION OF THE DFA (2)

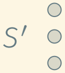
A transition function for sets of states of the NFA

$\delta^*(S', \sigma) =$ the set of states reachable from any state in S' by reading σ.

A transition function for sets of states of the NFA

$\delta^*(S', \sigma)$ = the set of states reachable from any state in S' by reading σ.

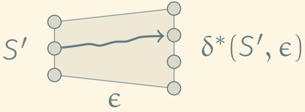
- $\delta^*(S', \epsilon) =$



A transition function for sets of states of the NFA

$\delta^*(S', \sigma)$ = the set of states reachable from any state in S' by reading σ.

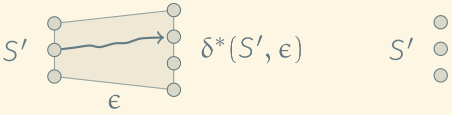
- $\delta^*(S', \epsilon) = \text{ECLOSE}(S')$



A transition function for sets of states of the NFA

$\delta^*(S', \sigma) =$ the set of states reachable from any state in S' by reading σ.

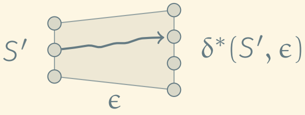
- $\delta^*(S', \epsilon) = \mathrm{ECLOSE}(S')$
- $\delta^*(S', x\sigma) =$

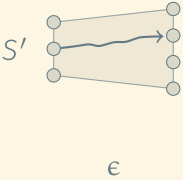


A transition function for sets of states of the NFA

$\delta^*(S', \sigma) =$ the set of states reachable from any state in S' by reading σ.

- $\delta^*(S', \epsilon) = \mathrm{ECLOSE}(S')$
- $\delta^*(S', x\sigma) = \bigcup_{s_1 \in \mathrm{ECLOSE}(S')}$



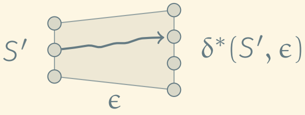


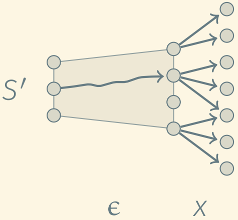
TRANSITION FUNCTION OF THE DFA (2)

A transition function for sets of states of the NFA

$\delta^*(S', \sigma)$ = the set of states reachable from any state in S' by reading σ.

- $\delta^*(S', \epsilon) = \mathrm{ECLOSE}(S')$
- $\delta^*(S', x\sigma) = \bigcup_{s_1 \in \mathrm{ECLOSE}(S')} \bigcup_{s_2 \in \delta(s_1, x)}$



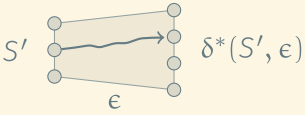


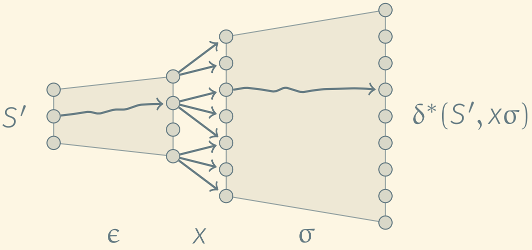
TRANSITION FUNCTION OF THE DFA (2)

A transition function for sets of states of the NFA

$\delta^*(S', \sigma)$ = the set of states reachable from any state in S' by reading σ.

- $\delta^*(S', \epsilon) = \mathrm{ECLOSE}(S')$
- $\delta^*(S', x\sigma) = \bigcup_{s_1 \in \mathrm{ECLOSE}(S')} \bigcup_{s_2 \in \delta(s_1, x)} \delta^*(s_2, \sigma)$





TRANSITION FUNCTION OF THE DFA (3)

Transition function of the DFA

Let

- $\gamma(S', x) = \bigcup_{s \in S'} \mathrm{ECLOSE}(\delta(s, x))$,

Transition function of the DFA

Let

- $\gamma(S', x) = \bigcup_{s \in S'} \mathrm{ECLOSE}(\delta(s, x))$,
- $\gamma^*(S', \epsilon) = S'$, and
- $\gamma^*(S', x\sigma) = \gamma^*(\gamma(S', x), \sigma)$.

Transition function of the DFA

Let

- $\gamma(S', x) = \bigcup_{s \in S'} \mathrm{ECLOSE}(\delta(s, x))$,
- $\gamma^*(S', \epsilon) = S'$, and
- $\gamma^*(S', x\sigma) = \gamma^*(\gamma(S', x), \sigma)$.

Then

$$\delta^*(S', \sigma) = \gamma^*(\mathrm{ECLOSE}(S'), \sigma).$$

TRANSITION FUNCTION OF THE DFA (3)

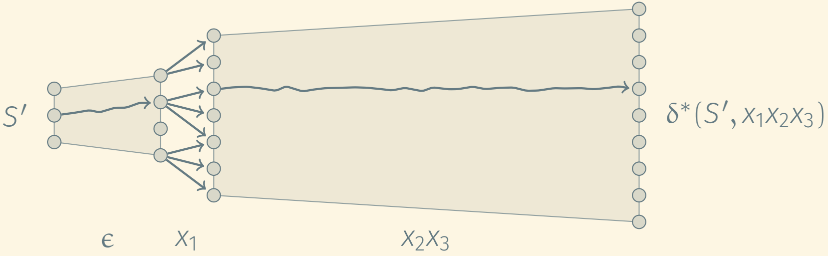
Transition function of the DFA

Let

- $\gamma(S', x) = \bigcup_{s \in S'} \text{ECLOSE}(\delta(s, x))$,
- $\gamma^*(S', \epsilon) = S'$, and
- $\gamma^*(S', x\sigma) = \gamma^*(\gamma(S', x), \sigma)$.

Then

$$\delta^*(S', \sigma) = \gamma^*(\text{ECLOSE}(S'), \sigma).$$



TRANSITION FUNCTION OF THE DFA (3)

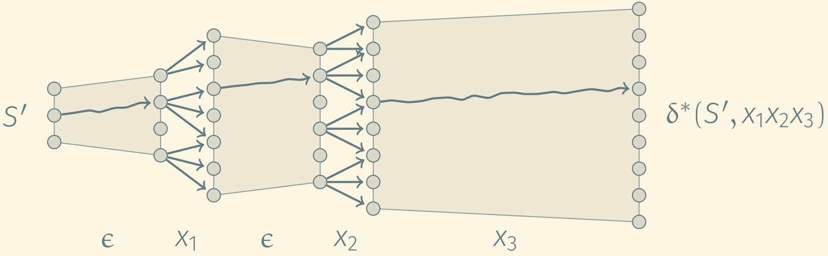
Transition function of the DFA

Let

- $\gamma(S', x) = \bigcup_{s \in S'} \mathrm{ECLOSE}(\delta(s, x))$,
- $\gamma^*(S', \epsilon) = S'$, and
- $\gamma^*(S', x\sigma) = \gamma^*(\gamma(S', x), \sigma)$.

Then

$$\delta^*(S', \sigma) = \gamma^*(\mathrm{ECLOSE}(S'), \sigma).$$



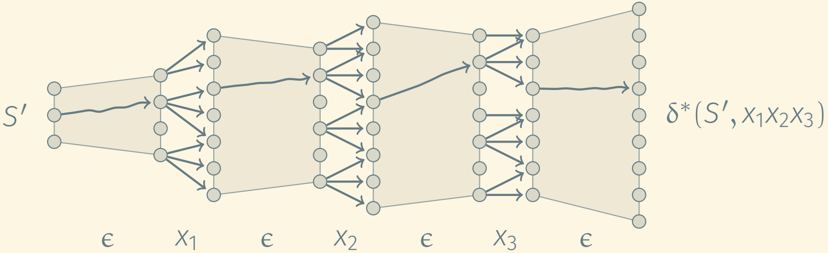
Transition function of the DFA

Let

- $\gamma(S', x) = \bigcup_{s \in S'} \text{ECLOSE}(\delta(s, x))$,
- $\gamma^*(S', \epsilon) = S'$, and
- $\gamma^*(S', x\sigma) = \gamma^*(\gamma(S', x), \sigma)$.

Then

$$\delta^*(S', \sigma) = \gamma^*(\text{ECLOSE}(S'), \sigma).$$



TRANSITION FUNCTION OF THE DFA (3)

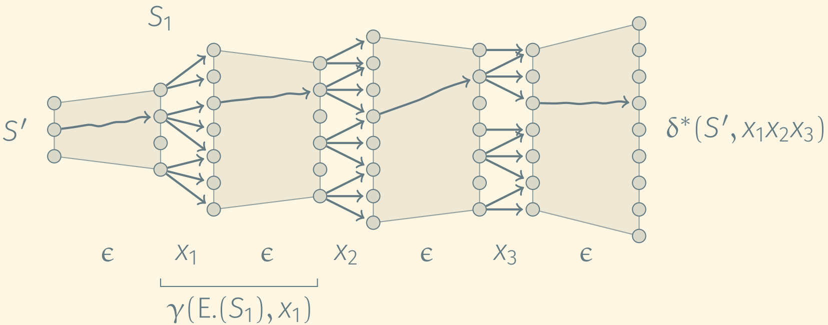
Transition function of the DFA

Let

- $\gamma(S', x) = \bigcup_{s \in S'} \text{ECLOSE}(\delta(s, x))$,
- $\gamma^*(S', \epsilon) = S'$, and
- $\gamma^*(S', x\sigma) = \gamma^*(\gamma(S', x), \sigma)$.

Then

$$\delta^*(S', \sigma) = \gamma^*(\text{ECLOSE}(S'), \sigma).$$



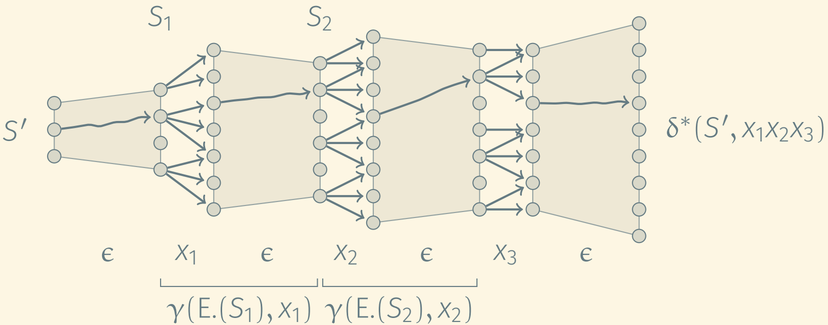
Transition function of the DFA

Let

- $\gamma(S', x) = \bigcup_{s \in S'} \text{ECLOSE}(\delta(s, x))$,
- $\gamma^*(S', \epsilon) = S'$, and
- $\gamma^*(S', x\sigma) = \gamma^*(\gamma(S', x), \sigma)$.

Then

$$\delta^*(S', \sigma) = \gamma^*(\text{ECLOSE}(S'), \sigma).$$



TRANSITION FUNCTION OF THE DFA (3)

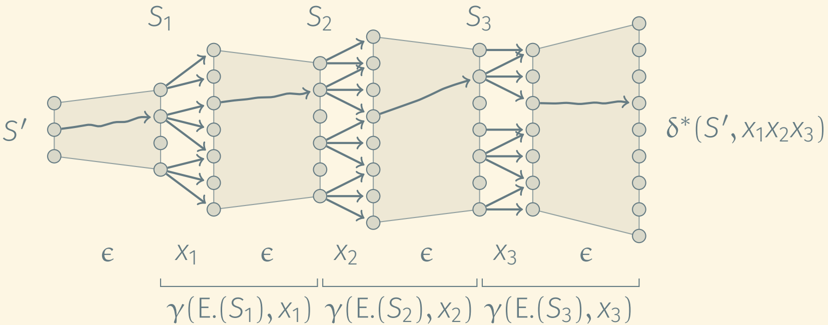
Transition function of the DFA

Let

- $\gamma(S', x) = \bigcup_{s \in S'} \text{ECLOSE}(\delta(s, x))$,
- $\gamma^*(S', \epsilon) = S'$, and
- $\gamma^*(S', x\sigma) = \gamma^*(\gamma(S', x), \sigma)$.

Then

$$\delta^*(S', \sigma) = \gamma^*(\text{ECLOSE}(S'), \sigma).$$



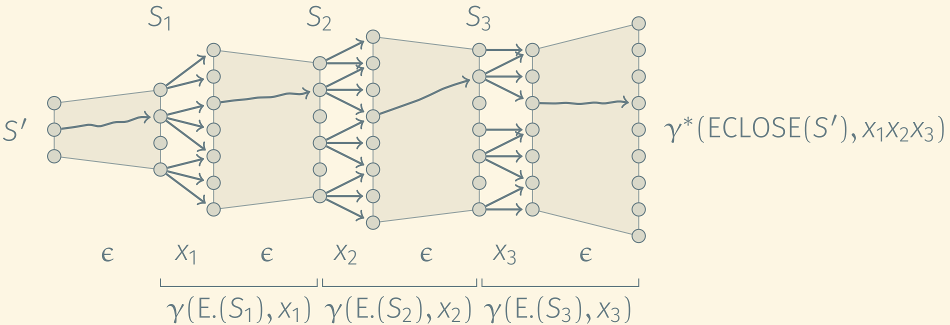
Transition function of the DFA

Let

- $\gamma(S', x) = \bigcup_{s \in S'} \text{ECLOSE}(\delta(s, x))$,
- $\gamma^*(S', \epsilon) = S'$, and
- $\gamma^*(S', x\sigma) = \gamma^*(\gamma(S', x), \sigma)$.

Then

$$\delta^*(S', \sigma) = \gamma^*(\text{ECLOSE}(S'), \sigma).$$



FROM NFA TO DFA: CONSTRUCTING ONLY THE STATES WE NEED

$N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (T, \Sigma, \gamma, t_0, G)$

FROM NFA TO DFA: CONSTRUCTING ONLY THE STATES WE NEED

$N = (S, \Sigma, \delta, s_0, F) \longrightarrow D = (T, \Sigma, \gamma, t_0, G)$

So far, $T = 2^S$, but only a subset of states may be reachable from t_0.
$\longrightarrow$ We would like to choose T to be this subset.

FROM NFA TO DFA: CONSTRUCTING ONLY THE STATES WE NEED

$N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (T, \Sigma, \gamma, t_0, G)$

So far, $T = 2^S$, but only a subset of states may be reachable from t_0.
$\rightarrow$ We would like to choose T to be this subset.

Obvious idea: Construct D with $T = 2^S$, then throw away the states not reachable from t_0.

$N = (S, \Sigma, \delta, s_0, F) \rightarrow D = (T, \Sigma, \gamma, t_0, G)$

So far, $T = 2^S$, but only a subset of states may be reachable from t_0.
$\rightarrow$ We would like to choose T to be this subset.

Obvious idea: Construct D with $T = 2^S$, then throw away the states not reachable from t_0.
Too costly!

FROM NFA TO DFA: CONSTRUCTING ONLY THE STATES WE NEED

$N = (S, \Sigma, \delta, s_0, F) \longrightarrow D = (T, \Sigma, \gamma, t_0, G)$

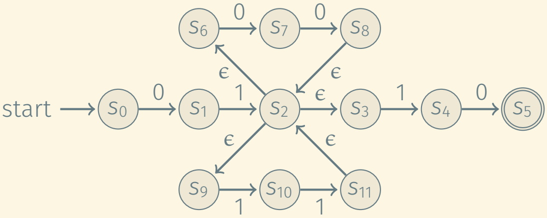
So far, $T = 2^S$, but only a subset of states may be reachable from t_0.
$\longrightarrow$ We would like to choose T to be this subset.

Obvious idea: Construct D with $T = 2^S$, then throw away the states not reachable from t_0.
Too costly!

Almost as obvious: Generate only the states we can reach from t_0:

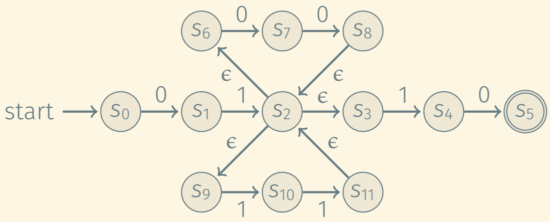
- Start with $T = \{t_0\}$ and a queue $Q = \{t_0\}$ of new states.
- While $Q \neq \emptyset$:
  - Remove some $t \in Q$ from Q.
  - For each $x \in \Sigma$, add $\gamma(t, x)$ to T, and to Q if $\gamma(t, x)$ was not in T before.

FROM NFA TO DFA: EXAMPLE

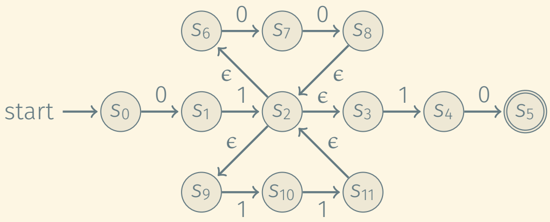


	δ	0	1
$\rightarrow$			

FROM NFA TO DFA: EXAMPLE

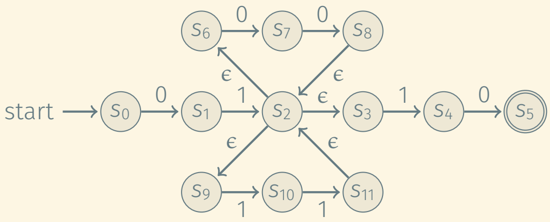


	δ	0	1
$\rightarrow$	$\{s_0\}$		



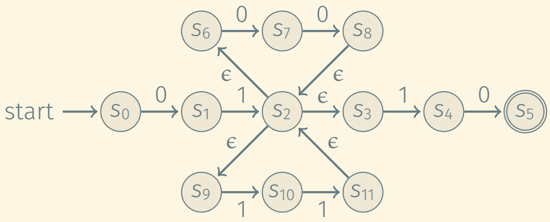
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	

FROM NFA TO DFA: EXAMPLE



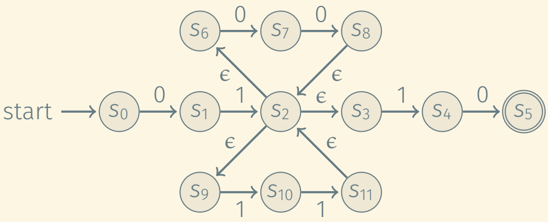
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$

FROM NFA TO DFA: EXAMPLE



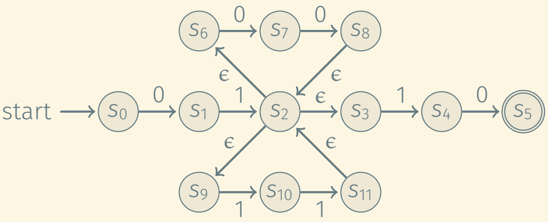
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$		
	$\emptyset$		

FROM NFA TO DFA: EXAMPLE



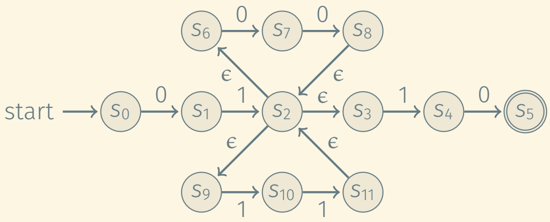
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	
	$\emptyset$		

FROM NFA TO DFA: EXAMPLE



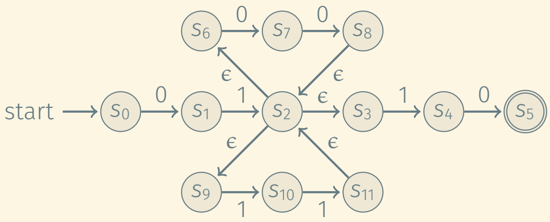
	δ	0	1
→	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$		

FROM NFA TO DFA: EXAMPLE

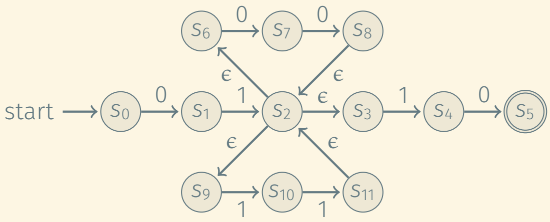


	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$		
	$\{s_2, s_3, s_6, s_9\}$		

FROM NFA TO DFA: EXAMPLE

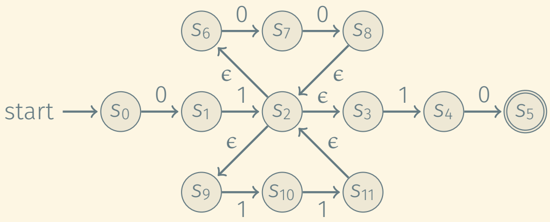


	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$		



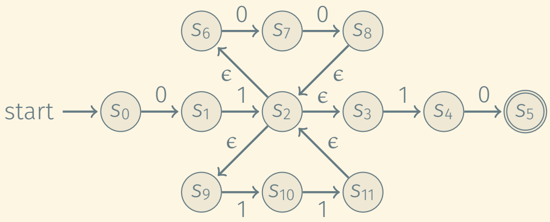
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	

FROM NFA TO DFA: EXAMPLE



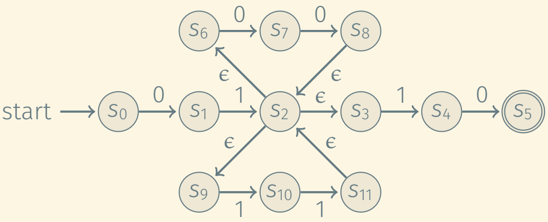
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$

FROM NFA TO DFA: EXAMPLE



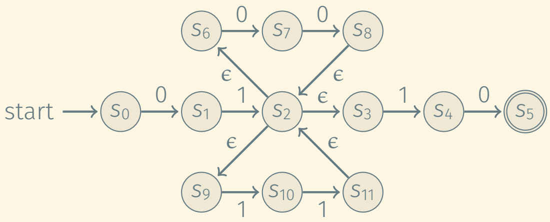
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$		
	$\{s_4, s_{10}\}$		

FROM NFA TO DFA: EXAMPLE



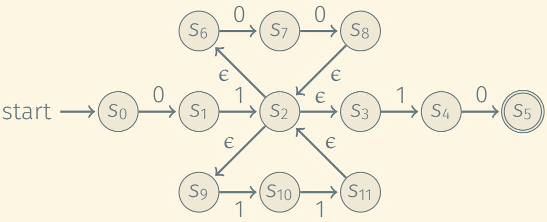
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	
	$\{s_4, s_{10}\}$		

FROM NFA TO DFA: EXAMPLE



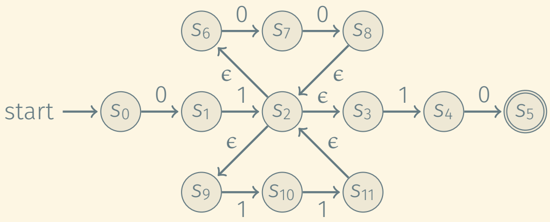
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$		

FROM NFA TO DFA: EXAMPLE



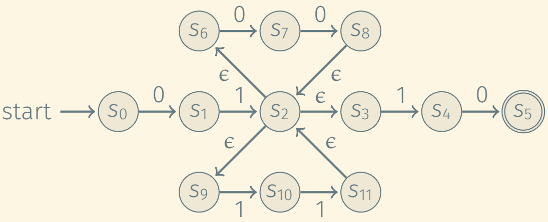
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$		
	$\{s_2, s_3, s_6, s_8, s_9\}$		

FROM NFA TO DFA: EXAMPLE



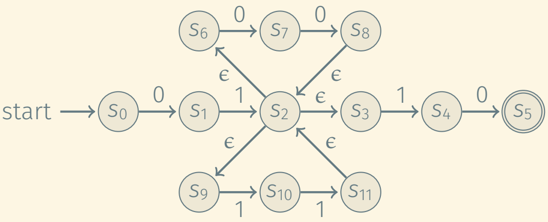
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$	$\{s_5\}$	
	$\{s_2, s_3, s_6, s_8, s_9\}$		

FROM NFA TO DFA: EXAMPLE



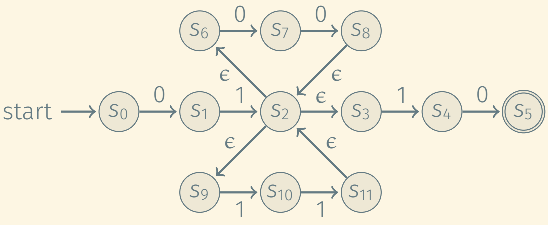
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$	$\{s_5\}$	$\{s_2, s_3, s_6, s_9, s_{11}\}$
	$\{s_2, s_3, s_6, s_8, s_9\}$		

FROM NFA TO DFA: EXAMPLE



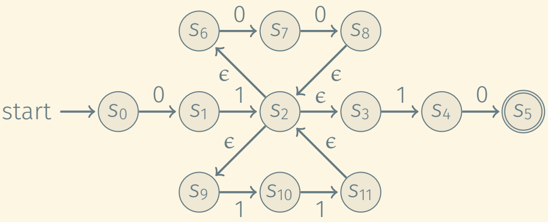
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$	$\{s_5\}$	$\{s_2, s_3, s_6, s_9, s_{11}\}$
	$\{s_2, s_3, s_6, s_8, s_9\}$		
	$\{s_5\}$		
	$\{s_2, s_3, s_6, s_9, s_{11}\}$		

FROM NFA TO DFA: EXAMPLE



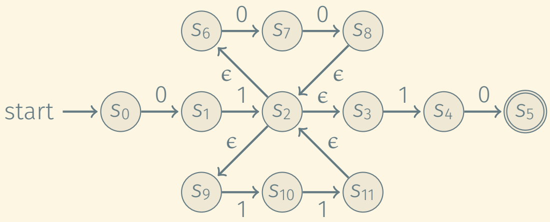
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$	$\{s_5\}$	$\{s_2, s_3, s_6, s_9, s_{11}\}$
	$\{s_2, s_3, s_6, s_8, s_9\}$	$\{s_7\}$	
	$\{s_5\}$		
	$\{s_2, s_3, s_6, s_9, s_{11}\}$		

FROM NFA TO DFA: EXAMPLE



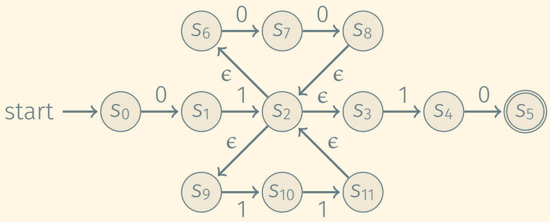
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$	$\{s_5\}$	$\{s_2, s_3, s_6, s_9, s_{11}\}$
	$\{s_2, s_3, s_6, s_8, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_5\}$		
	$\{s_2, s_3, s_6, s_9, s_{11}\}$		

FROM NFA TO DFA: EXAMPLE



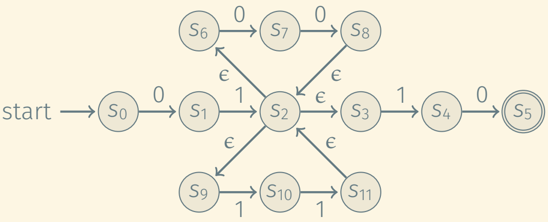
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$	$\{s_5\}$	$\{s_2, s_3, s_6, s_9, s_{11}\}$
	$\{s_2, s_3, s_6, s_8, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_5\}$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9, s_{11}\}$		

FROM NFA TO DFA: EXAMPLE



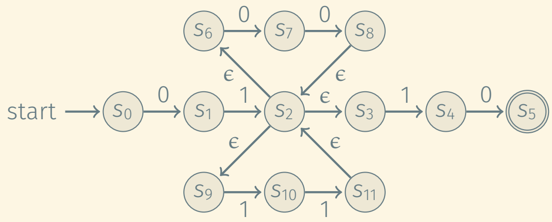
	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$	$\{s_5\}$	$\{s_2, s_3, s_6, s_9, s_{11}\}$
	$\{s_2, s_3, s_6, s_8, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_5\}$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9, s_{11}\}$	$\{s_7\}$	

FROM NFA TO DFA: EXAMPLE

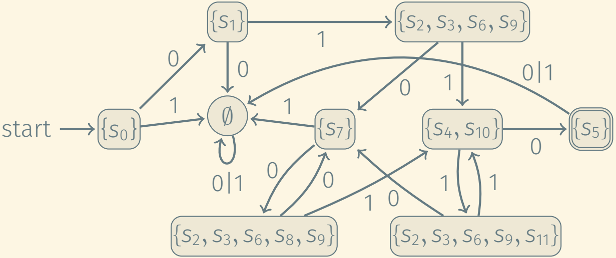


	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$	$\{s_5\}$	$\{s_2, s_3, s_6, s_9, s_{11}\}$
	$\{s_2, s_3, s_6, s_8, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_5\}$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9, s_{11}\}$	$\{s_7\}$	$\{s_4, s_{10}\}$

FROM NFA TO DFA: EXAMPLE



	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$	$\{s_5\}$	$\{s_2, s_3, s_6, s_9, s_{11}\}$
	$\{s_2, s_3, s_6, s_8, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
*	$\{s_5\}$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9, s_{11}\}$	$\{s_7\}$	$\{s_4, s_{10}\}$



	δ	0	1
$\rightarrow$	$\{s_0\}$	$\{s_1\}$	$\emptyset$
	$\{s_1\}$	$\emptyset$	$\{s_2, s_3, s_6, s_9\}$
	$\emptyset$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
	$\{s_7\}$	$\{s_2, s_3, s_6, s_8, s_9\}$	$\emptyset$
	$\{s_4, s_{10}\}$	$\{s_5\}$	$\{s_2, s_3, s_6, s_9, s_{11}\}$
	$\{s_2, s_3, s_6, s_8, s_9\}$	$\{s_7\}$	$\{s_4, s_{10}\}$
*	$\{s_5\}$	$\emptyset$	$\emptyset$
	$\{s_2, s_3, s_6, s_9, s_{11}\}$	$\{s_7\}$	$\{s_4, s_{10}\}$

FROM NFA TO DFA: EXPONENTIAL NUMBER OF STATES (1)

Our construction aims to avoid constructing an NFA with an exponential number of states.

This works for most languages (e.g., the ones used in lexical analysis).

FROM NFA TO DFA: EXPONENTIAL NUMBER OF STATES (1)

Our construction aims to avoid constructing an NFA with an exponential number of states.

This works for most languages (e.g., the ones used in lexical analysis).

However, there are languages where a DFA needs exponentially more states than an NFA:

$$\mathcal{L}(.{*}1.\{n-1\})$$

FROM NFA TO DFA: EXPONENTIAL NUMBER OF STATES (1)

Our construction aims to avoid constructing an NFA with an exponential number of states.

This works for most languages (e.g., the ones used in lexical analysis).

However, there are languages where a DFA needs exponentially more states than an NFA:

$$\mathcal{L}(.*1.\{n-1\})$$

The NFA has $n+1$ states:

FROM NFA TO DFA: EXPONENTIAL NUMBER OF STATES (1)

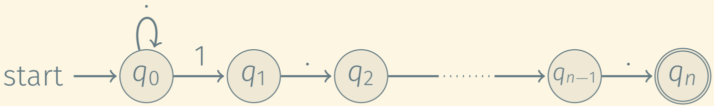
Our construction aims to avoid constructing an NFA with an exponential number of states.

This works for most languages (e.g., the ones used in lexical analysis).

However, there are languages where a DFA needs exponentially more states than an NFA:

$$\mathcal{L}(.{*}1.\{n-1\})$$

The NFA has $n+1$ states:



Claim

Any DFA that decides $\mathcal{L} = \mathcal{L}(.*1.\{n-1\})$ has at least 2^n states. ($\Sigma = \{0, 1\}$.)

Claim

Any DFA that decides $\mathcal{L} = \mathcal{L}(.*1.\{n-1\})$ has at least 2^n states. ($\Sigma = \{0, 1\}$.)

Proof:

Claim

Any DFA that decides $\mathcal{L} = \mathcal{L}(.{*}1.\{n-1\})$ has at least 2^n states. ($\Sigma = \{0, 1\}$.)

Proof:

- Assume there exists a DFA $D = (S, \Sigma, \delta, s_0, F)$ with $\mathcal{L}(D) = \mathcal{L}$ and $|S| < 2^n$.

FROM NFA TO DFA: EXPONENTIAL NUMBER OF STATES (2)

Claim

Any DFA that decides $\mathcal{L} = \mathcal{L}(.{*}1.\{n-1\})$ has at least 2^n states. ($\Sigma = \{0, 1\}$.)

Proof:

- Assume there exists a DFA $D = (S, \Sigma, \delta, s_0, F)$ with $\mathcal{L}(D) = \mathcal{L}$ and $|S| < 2^n$.

$\Rightarrow$ There exist two strings $\sigma_1 \neq \sigma_2 \in \Sigma^n$ such that $\delta^*(s_0, \sigma_1) = \delta^*(s_0, \sigma_2)$.

Claim

Any DFA that decides $\mathcal{L} = \mathcal{L}(.{*}1.\{n-1\})$ has at least 2^n states. ($\Sigma = \{0, 1\}$.)

Proof:

- Assume there exists a DFA $D = (S, \Sigma, \delta, s_0, F)$ with $\mathcal{L}(D) = \mathcal{L}$ and $|S| < 2^n$.

$\Rightarrow$ There exist two strings $\sigma_1 \neq \sigma_2 \in \Sigma^n$ such that $\delta^*(s_0, \sigma_1) = \delta^*(s_0, \sigma_2)$.

- Since $\sigma_1 \neq \sigma_2$, w.l.o.g. $\sigma_1 = .\{m\}0.\{n-m-1\}$ and $\sigma_2 = .\{m\}1.\{n-m-1\}$.

FROM NFA TO DFA: EXPONENTIAL NUMBER OF STATES (2)

Claim

Any DFA that decides $\mathcal{L} = \mathcal{L}(.{*}1.\{n-1\})$ has at least 2^n states. ($\Sigma = \{0, 1\}$.)

Proof:

- Assume there exists a DFA $D = (S, \Sigma, \delta, s_0, F)$ with $\mathcal{L}(D) = \mathcal{L}$ and $|S| < 2^n$.

$\Rightarrow$ There exist two strings $\sigma_1 \neq \sigma_2 \in \Sigma^n$ such that $\delta^*(s_0, \sigma_1) = \delta^*(s_0, \sigma_2)$.

- Since $\sigma_1 \neq \sigma_2$, w.l.o.g. $\sigma_1 = .\{m\}0.\{n-m-1\}$ and $\sigma_2 = .\{m\}1.\{n-m-1\}$.
- Since $\delta^*(s_0, \sigma_1) = \delta^*(s_0, \sigma_2)$, we also have $\delta^*(s_0, \sigma_1 0^m) = \delta^*(s_0, \sigma_2 0^m)$.

FROM NFA TO DFA: EXPONENTIAL NUMBER OF STATES (2)

Claim

Any DFA that decides $\mathcal{L} = \mathcal{L}(.{*}1.\{n-1\})$ has at least 2^n states. ($\Sigma = \{0, 1\}$.)

Proof:

- Assume there exists a DFA $D = (S, \Sigma, \delta, s_0, F)$ with $\mathcal{L}(D) = \mathcal{L}$ and $|S| < 2^n$.

$\Rightarrow$ There exist two strings $\sigma_1 \neq \sigma_2 \in \Sigma^n$ such that $\delta^*(s_0, \sigma_1) = \delta^*(s_0, \sigma_2)$.

- Since $\sigma_1 \neq \sigma_2$, w.l.o.g. $\sigma_1 = .\{m\}0.\{n-m-1\}$ and $\sigma_2 = .\{m\}1.\{n-m-1\}$.
- Since $\delta^*(s_0, \sigma_1) = \delta^*(s_0, \sigma_2)$, we also have $\delta^*(s_0, \sigma_1 0^m) = \delta^*(s_0, \sigma_2 0^m)$.

$\Rightarrow$ Either D accepts both $\sigma_1 0^m$ and $\sigma_2 0^m$ or D rejects both $\sigma_1 0^m$ and $\sigma_2 0^m$.

Claim

Any DFA that decides $\mathcal{L} = \mathcal{L}(.*1.\{n-1\})$ has at least 2^n states. ($\Sigma = \{0, 1\}$.)

Proof:

- Assume there exists a DFA $D = (S, \Sigma, \delta, s_0, F)$ with $\mathcal{L}(D) = \mathcal{L}$ and $|S| < 2^n$.

$\Rightarrow$ There exist two strings $\sigma_1 \neq \sigma_2 \in \Sigma^n$ such that $\delta^*(s_0, \sigma_1) = \delta^*(s_0, \sigma_2)$.

- Since $\sigma_1 \neq \sigma_2$, w.l.o.g. $\sigma_1 = .\{m\}0.\{n-m-1\}$ and $\sigma_2 = .\{m\}1.\{n-m-1\}$.
- Since $\delta^*(s_0, \sigma_1) = \delta^*(s_0, \sigma_2)$, we also have $\delta^*(s_0, \sigma_1 0^m) = \delta^*(s_0, \sigma_2 0^m)$.

$\Rightarrow$ Either D accepts both $\sigma_1 0^m$ and $\sigma_2 0^m$ or D rejects both $\sigma_1 0^m$ and $\sigma_2 0^m$.

- However, $\sigma_1 0^m \notin \mathcal{L}$ and $\sigma_2 0^m \in \mathcal{L}$.

Claim

Any DFA that decides $\mathcal{L} = \mathcal{L}(.{*}1.\{n-1\})$ has at least 2^n states. ($\Sigma = \{0, 1\}$.)

Proof:

- Assume there exists a DFA $D = (S, \Sigma, \delta, s_0, F)$ with $\mathcal{L}(D) = \mathcal{L}$ and $|S| < 2^n$.

$\Rightarrow$ There exist two strings $\sigma_1 \neq \sigma_2 \in \Sigma^n$ such that $\delta^*(s_0, \sigma_1) = \delta^*(s_0, \sigma_2)$.

- Since $\sigma_1 \neq \sigma_2$, w.l.o.g. $\sigma_1 = .\{m\}0.\{n-m-1\}$ and $\sigma_2 = .\{m\}1.\{n-m-1\}$.
- Since $\delta^*(s_0, \sigma_1) = \delta^*(s_0, \sigma_2)$, we also have $\delta^*(s_0, \sigma_1 0^m) = \delta^*(s_0, \sigma_2 0^m)$.

$\Rightarrow$ Either D accepts both $\sigma_1 0^m$ and $\sigma_2 0^m$ or D rejects both $\sigma_1 0^m$ and $\sigma_2 0^m$.

- However, $\sigma_1 0^m \notin \mathcal{L}$ and $\sigma_2 0^m \in \mathcal{L}$.

$\Rightarrow$ D does not decide $\mathcal{L}$.

FROM REGULAR EXPRESSION TO NFA

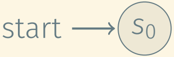
Base cases:

Base cases:

$\emptyset$

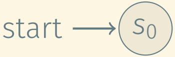
Base cases:

$\emptyset$



Base cases:

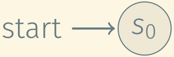
$\emptyset$



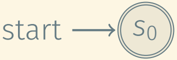
ϵ

Base cases:

$\emptyset$

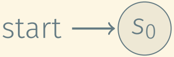


ϵ

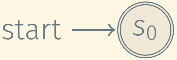


Base cases:

$\emptyset$



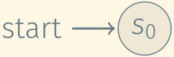
ϵ



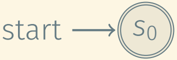
$x \quad (x \in \Sigma)$

Base cases:

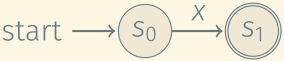
$\emptyset$



ϵ

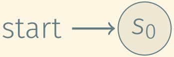


$x \quad (x \in \Sigma)$

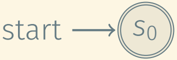


Base cases:

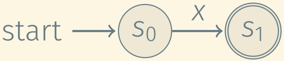
$\emptyset$



ϵ



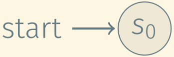
$x \quad (x \in \Sigma)$



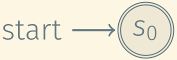
Inductive steps:

Base cases:

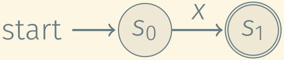
$\emptyset$



ϵ



$x \quad (x \in \Sigma)$

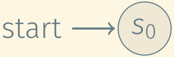


Inductive steps:

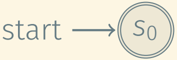
$A|B$

Base cases:

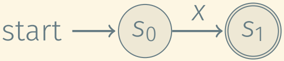
$\emptyset$



ϵ

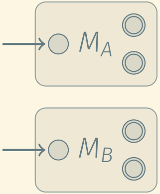


$x \quad (x \in \Sigma)$



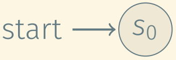
Inductive steps:

$A|B$

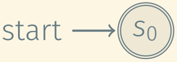


Base cases:

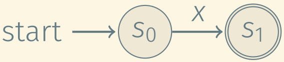
$\emptyset$



ϵ

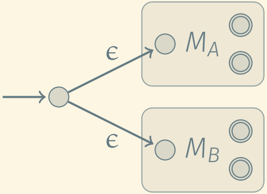


$x \quad (x \in \Sigma)$



Inductive steps:

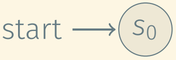
$A|B$



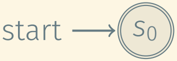
FROM REGULAR EXPRESSION TO NFA

Base cases:

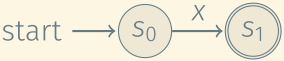
$\emptyset$



ϵ

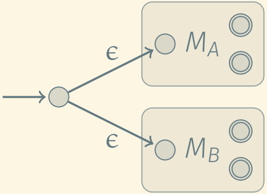


$x \quad (x \in \Sigma)$



Inductive steps:

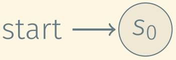
$A|B$



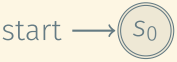
AB

Base cases:

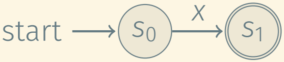
$\emptyset$



ϵ

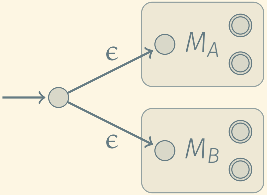


$x \quad (x \in \Sigma)$

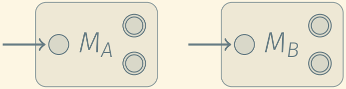


Inductive steps:

$A|B$



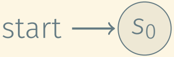
AB



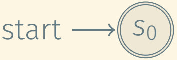
FROM REGULAR EXPRESSION TO NFA

Base cases:

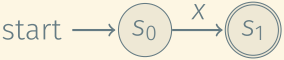
$\emptyset$



ϵ

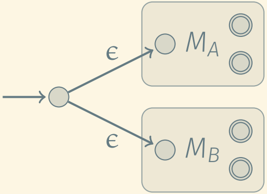


$x \quad (x \in \Sigma)$

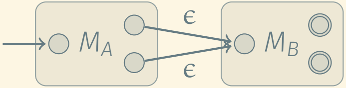


Inductive steps:

$A|B$

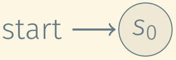


AB

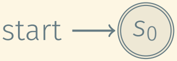


Base cases:

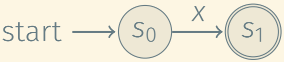
$\emptyset$



ϵ

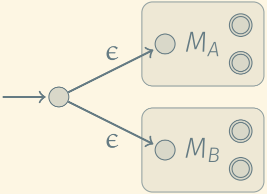


$x \quad (x \in \Sigma)$

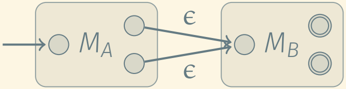


Inductive steps:

$A|B$



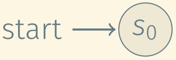
AB



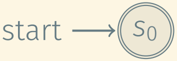
$A*$

Base cases:

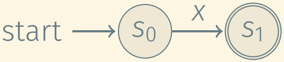
$\emptyset$



ϵ

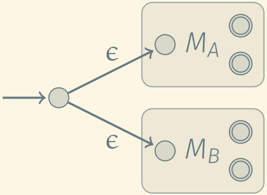


$x \quad (x \in \Sigma)$

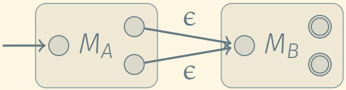


Inductive steps:

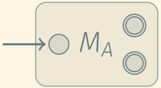
$A|B$



AB

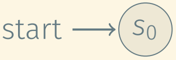


$A*$

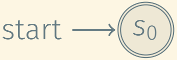


Base cases:

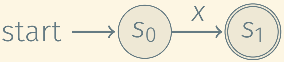
$\emptyset$



ϵ

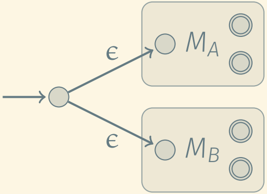


$x \quad (x \in \Sigma)$

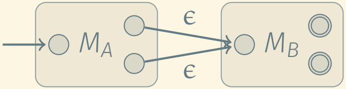


Inductive steps:

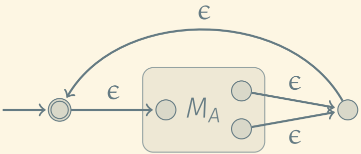
$A|B$



AB



$A*$



FROM REGULAR EXPRESSION TO NFA: EXAMPLE

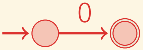
$$(0|\epsilon)(1|000*)*(0|\epsilon)$$

FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$$(0|\epsilon)(1|000*)*(0|\epsilon)$$

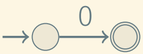
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$$(0|\epsilon)(1|000*)*(0|\epsilon)$$



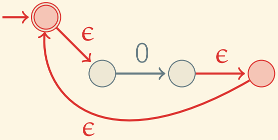
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$$(0|\epsilon)(1|000*)*(0|\epsilon)$$



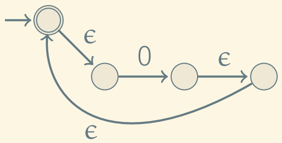
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$$(0|\epsilon)(1|000*)*(0|\epsilon)$$



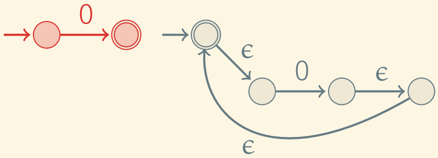
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$

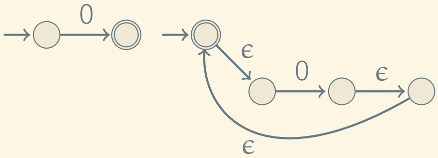


FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$$(0|\epsilon)(1|000*)*(0|\epsilon)$$

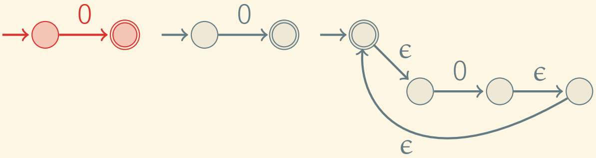


$$(0|\epsilon)(1|000*)*(0|\epsilon)$$



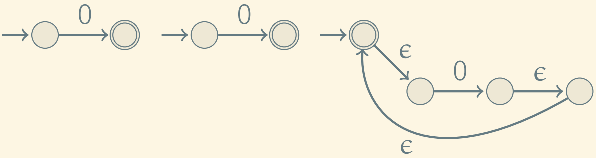
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$



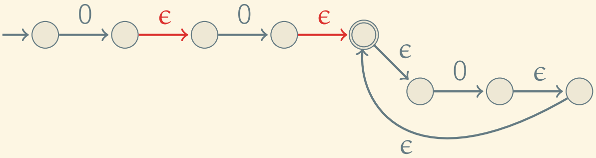
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$



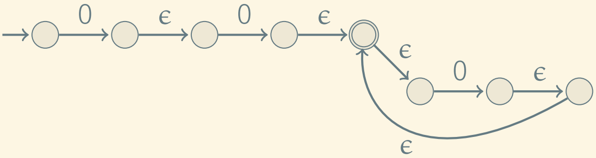
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$



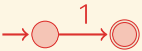
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

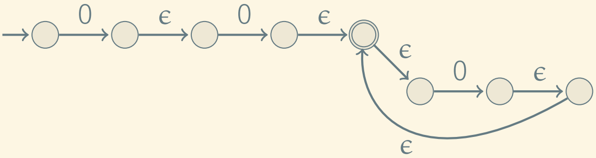
$(0|\epsilon)(1|000*)*(0|\epsilon)$



FROM REGULAR EXPRESSION TO NFA: EXAMPLE

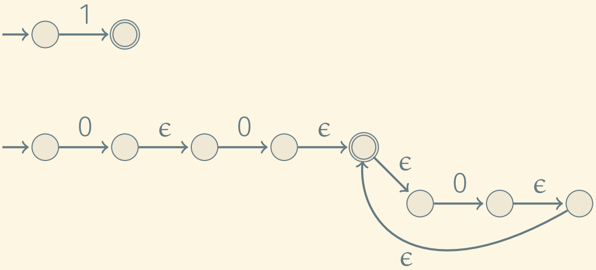
$(0|\epsilon)(1|000*)*(0|\epsilon)$



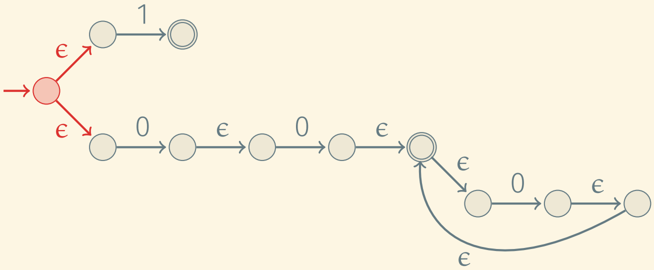


FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$

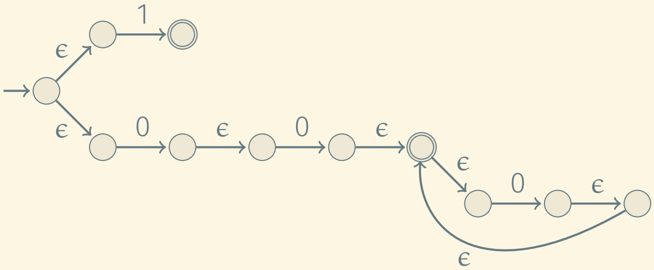


$(0|\epsilon)(1|000*)*(0|\epsilon)$



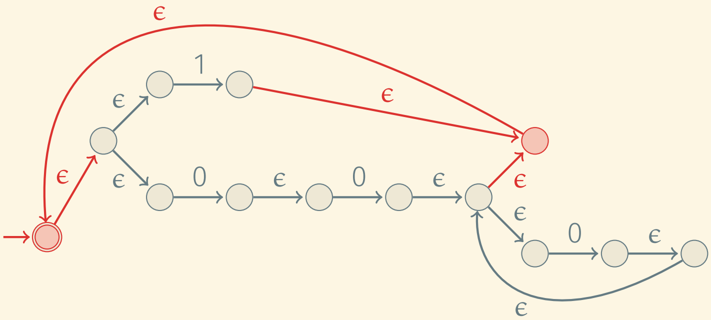
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$$(0|\epsilon)(1|000*)*(0|\epsilon)$$



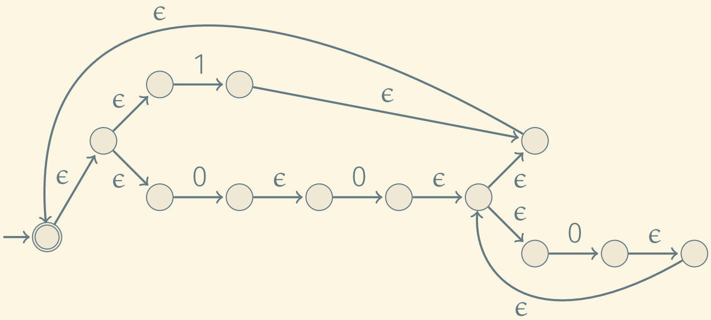
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$



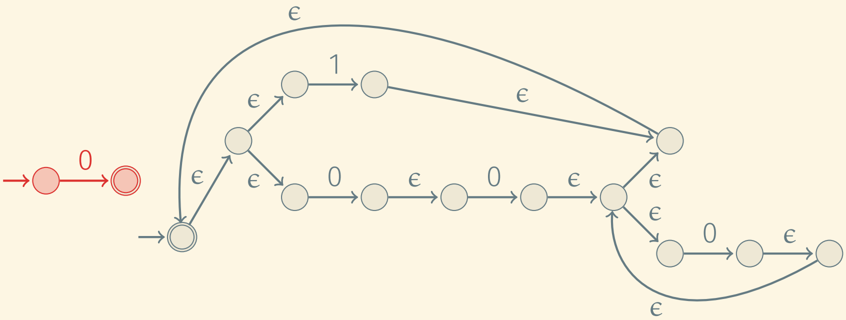
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$

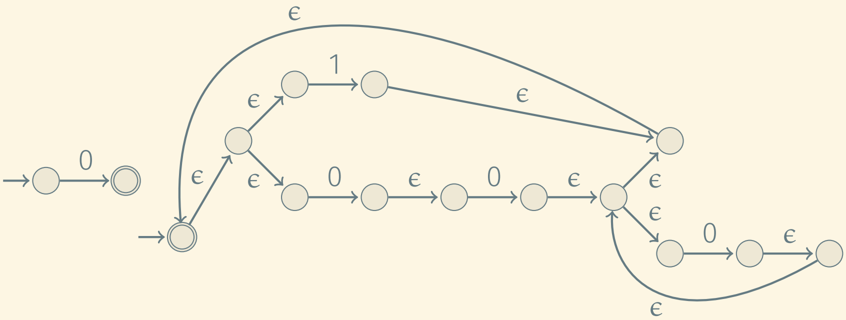


FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$

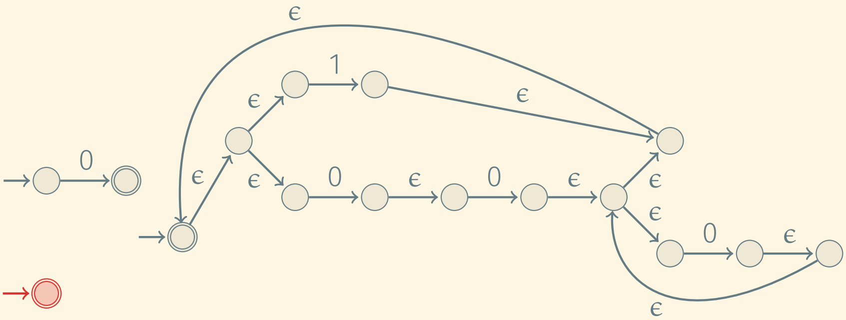


$(0|\epsilon)(1|000*)*(0|\epsilon)$



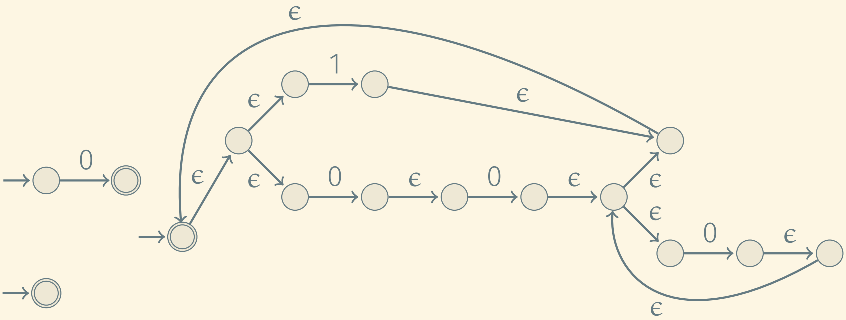
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$



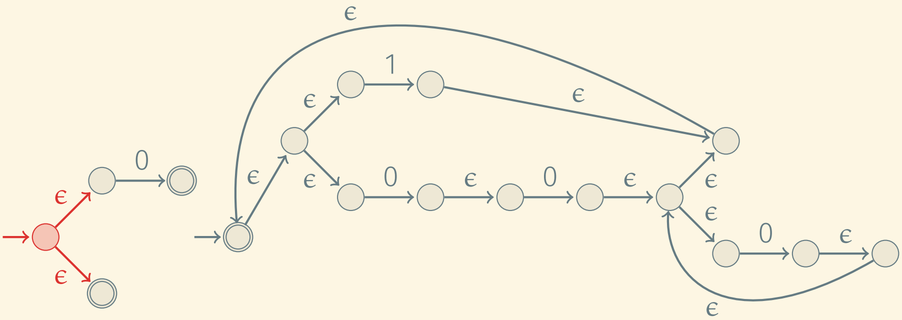
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$



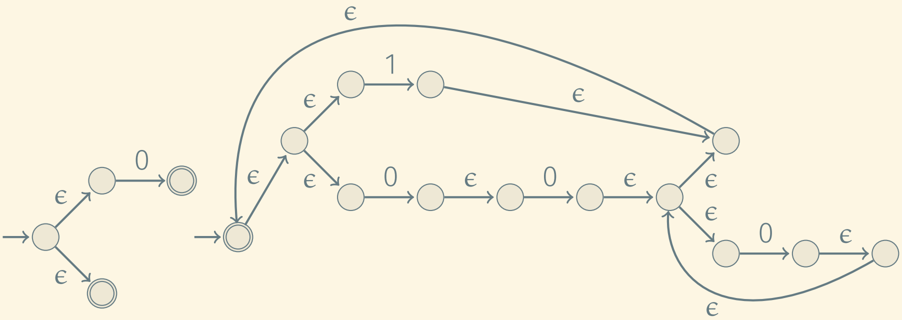
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$



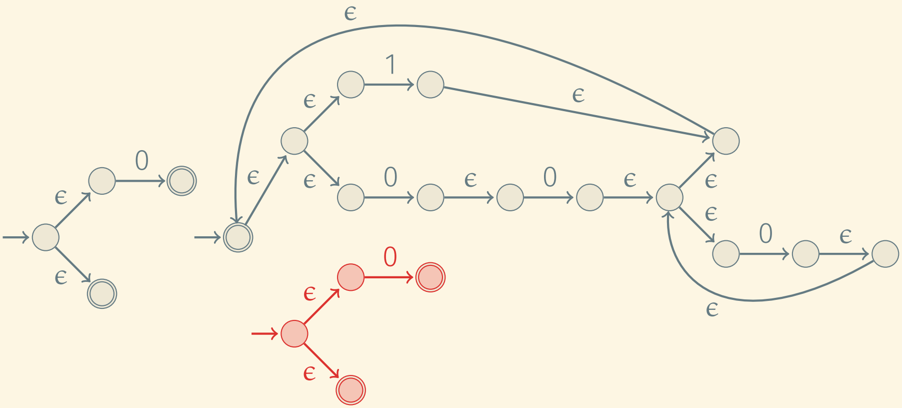
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*\color{red}{(0|\epsilon)}$

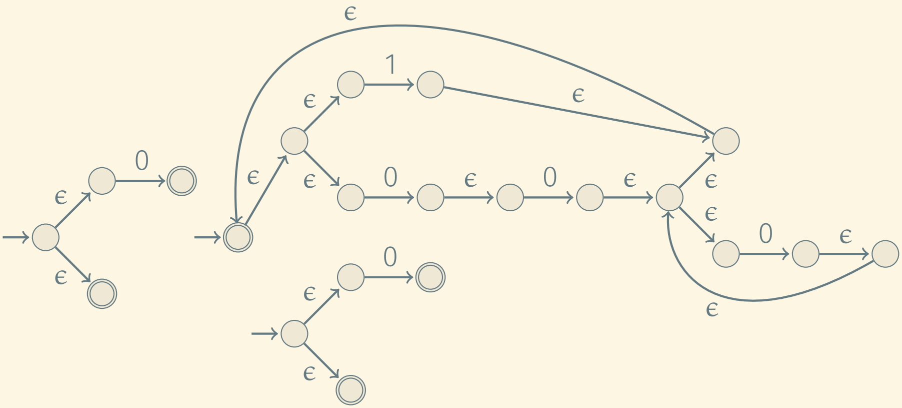


FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$

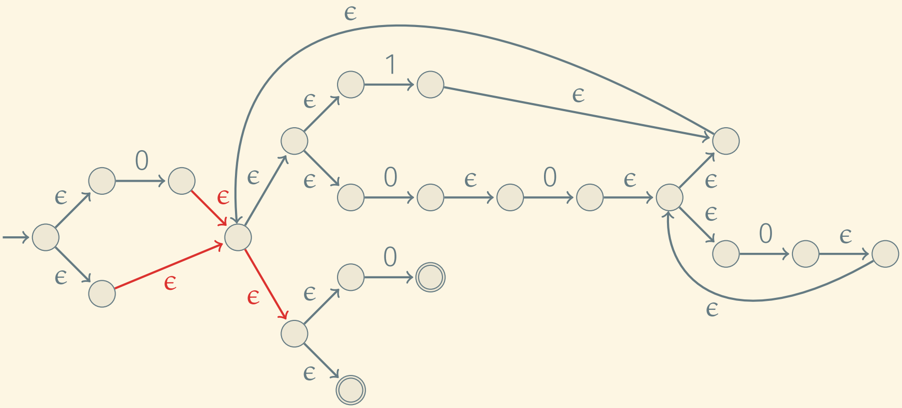


$(0|\epsilon)(1|000*)*(0|\epsilon)$



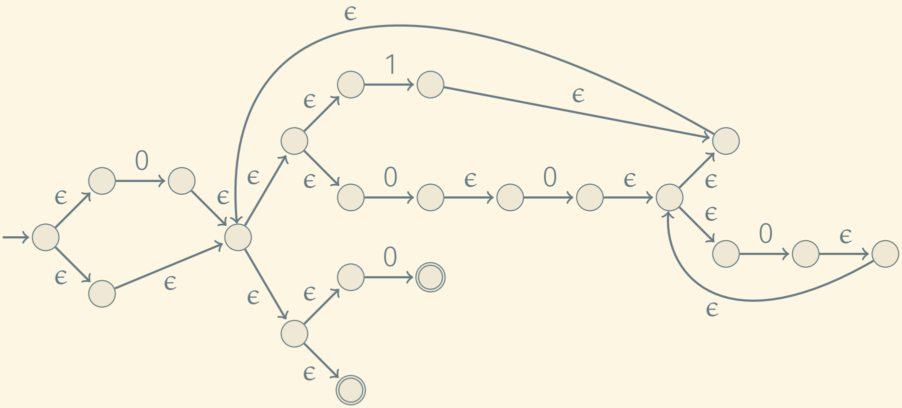
FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$



FROM REGULAR EXPRESSION TO NFA: EXAMPLE

$(0|\epsilon)(1|000*)*(0|\epsilon)$

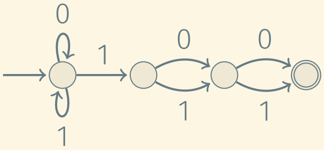


FROM NFA TO REGULAR EXPRESSION (1)

Acceptance of a string by (D/N)FA: Intuition

Concatenating the labels of the edges of a path in a (D/N)FA $N = (S, \Sigma, \delta, s_0, F)$ produces a string, called the label of the path.

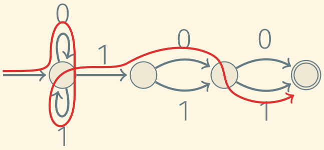
N accepts a string σ if there exists a path from s_0 to an accepting state whose label is σ.



Acceptance of a string by (D/N)FA: Intuition

Concatenating the labels of the edges of a path in a (D/N)FA $N = (S, \Sigma, \delta, s_0, F)$ produces a string, called the label of the path.

N accepts a string σ if there exists a path from s_0 to an accepting state whose label is σ.



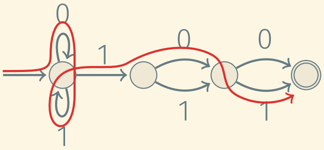
Accepts 01101

FROM NFA TO REGULAR EXPRESSION (1)

Acceptance of a string by (D/N)FA: Intuition

Concatenating the labels of the edges of a path in a (D/N)FA $N = (S, \Sigma, \delta, s_0, F)$ produces a string, called the label of the path.

N accepts a string σ if there exists a path from s_0 to an accepting state whose label is σ.

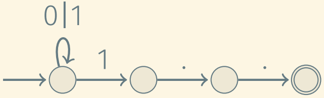


Accepts 01101

Definition: Regular Expression NFA (RNFA)

An RNFA N is a finite automaton whose edges are labelled with regular expressions.

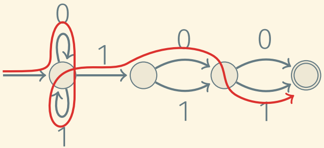
N accepts a string σ if there exists a path from s_0 to an accepting state whose label is R and $\sigma \in \mathcal{L}(R)$.



Acceptance of a string by (D/N)FA: Intuition

Concatenating the labels of the edges of a path in a (D/N)FA $N = (S, \Sigma, \delta, s_0, F)$ produces a string, called the label of the path.

N accepts a string σ if there exists a path from s_0 to an accepting state whose label is σ.

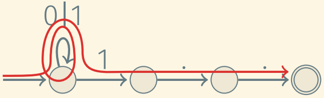


Accepts 01101

Definition: Regular Expression NFA (RNFA)

An RNFA N is a finite automaton whose edges are labelled with regular expressions.

N accepts a string σ if there exists a path from s_0 to an accepting state whose label is R and $\sigma \in \mathcal{L}(R)$.



Accepts $01101 \in \mathcal{L}((0|1)(0|1)1..)$

FROM NFA TO REGULAR EXPRESSION (2)

Proof idea:

$$\text{NFA} \longrightarrow \text{RFA}_1 \longrightarrow \text{RFA}_2 \longrightarrow \cdots \longrightarrow \text{RFA}_n$$

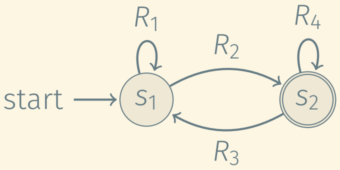
$$\mathcal{L}(\text{NFA}) = \mathcal{L}(\text{RFA}_1) = \mathcal{L}(\text{RFA}_2) = \cdots = \mathcal{L}(\text{RFA}_n)$$

Proof idea:

$$\text{NFA} \longrightarrow \text{RFA}_1 \longrightarrow \text{RFA}_2 \longrightarrow \cdots \longrightarrow \text{RFA}_n$$

$$\mathcal{L}(\text{NFA}) = \mathcal{L}(\text{RFA}_1) = \mathcal{L}(\text{RFA}_2) = \cdots = \mathcal{L}(\text{RFA}_n)$$

RFA_n has two states, an initial state and an accepting state:

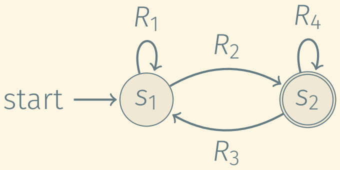


FROM NFA TO REGULAR EXPRESSION (2)

Proof idea:

$$\mathrm{NFA} \longrightarrow \mathrm{RFA}_1 \longrightarrow \mathrm{RFA}_2 \longrightarrow \cdots \longrightarrow \mathrm{RFA}_n$$
$$\mathcal{L}(\mathrm{NFA}) = \mathcal{L}(\mathrm{RFA}_1) = \mathcal{L}(\mathrm{RFA}_2) = \cdots = \mathcal{L}(\mathrm{RFA}_n)$$

RFA_n has two states, an initial state and an accepting state:



$$\mathcal{L}(\mathrm{NFA}) = \mathcal{L}(\mathrm{RFA}_n) = \mathcal{L}((R_1|R_2R_4{*}R_3){*}R_2R_4{*})$$

NFA $\longrightarrow$ RFA$_1$:

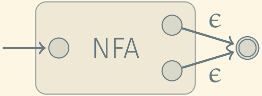
- $\mathcal{L}(\text{NFA}) = \mathcal{L}(\text{RFA}_1)$
- RFA$_1$ has one initial and one accepting state.

NFA $\longrightarrow$ RFA$_1$:

- $\mathcal{L}(\text{NFA}) = \mathcal{L}(\text{RFA}_1)$
- RFA$_1$ has one initial and one accepting state.

NFA $\longrightarrow$ RFA$_1$:

- $\mathcal{L}(\text{NFA}) = \mathcal{L}(\text{RFA}_1)$
- RFA$_1$ has one initial and one accepting state.



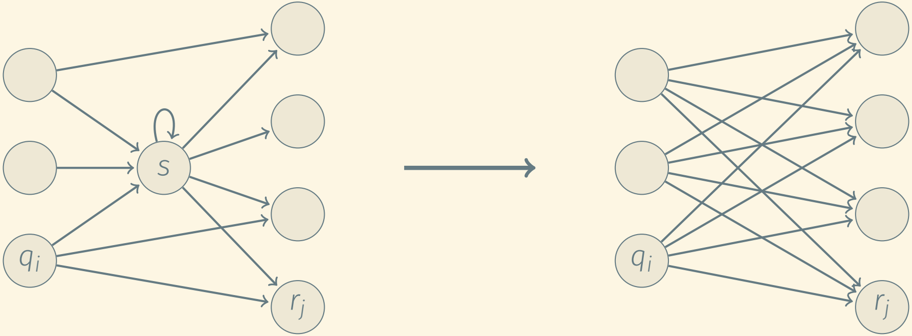
FROM NFA TO REGULAR EXPRESSION (4)

$\text{RFA}_k \longrightarrow \text{RFA}_{k+1}$:

- $\mathcal{L}(\text{RFA}_k) = \mathcal{L}(\text{RFA}_{k+1})$
- RFA_{k+1} has one state less than RFA_k.

$\text{RFA}_k \longrightarrow \text{RFA}_{k+1}$:

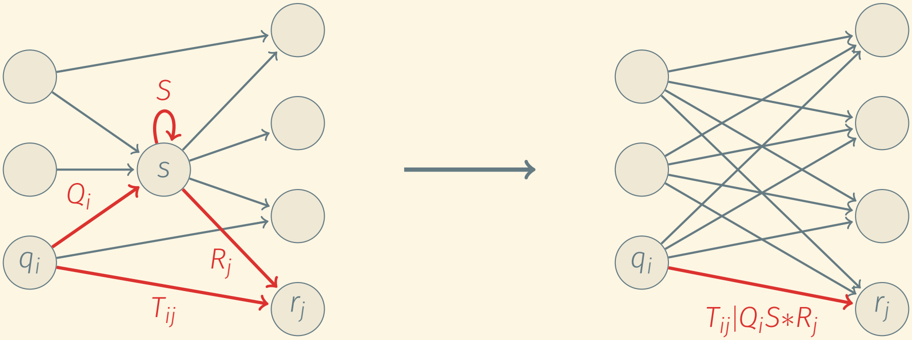
- $\mathcal{L}(\text{RFA}_k) = \mathcal{L}(\text{RFA}_{k+1})$
- RFA_{k+1} has one state less than RFA_k.



FROM NFA TO REGULAR EXPRESSION (4)

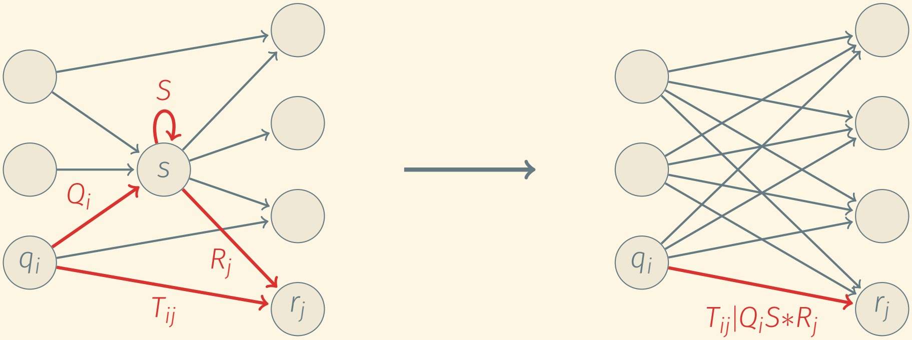
$\mathrm{RFA}_k \longrightarrow \mathrm{RFA}_{k+1}$:

- $\mathcal{L}(\mathrm{RFA}_k) = \mathcal{L}(\mathrm{RFA}_{k+1})$
- RFA_{k+1} has one state less than RFA_k.



$\mathrm{RFA}_k \longrightarrow \mathrm{RFA}_{k+1}$:

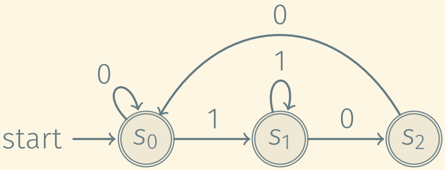
- $\mathcal{L}(\mathrm{RFA}_k) = \mathcal{L}(\mathrm{RFA}_{k+1})$
- RFA_{k+1} has one state less than RFA_k.



Note: This may create loops because some states may simultaneously be in- and out-neighbours of s.

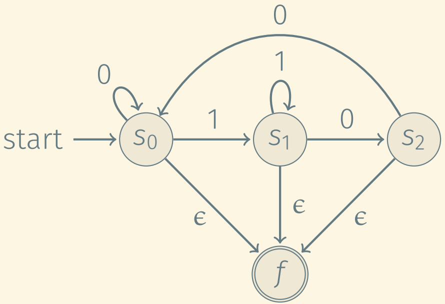
FROM NFA TO REGULAR EXPRESSION: EXAMPLE

$\mathcal{L}((0|\epsilon)(1|000*)*(0|\epsilon))$: All strings that do not contain 101 as a substring

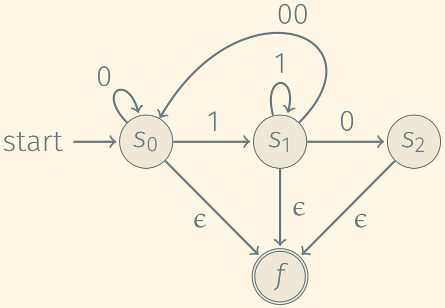


FROM NFA TO REGULAR EXPRESSION: EXAMPLE

$\mathcal{L}((0|\epsilon)(1|000*)*(0|\epsilon))$: All strings that do not contain 101 as a substring

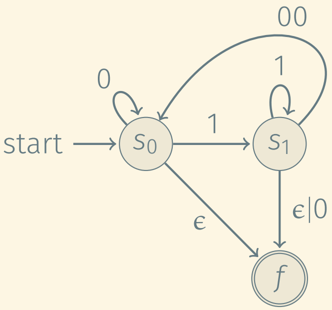


$\mathcal{L}((0|\epsilon)(1|000*)*(0|\epsilon))$: All strings that do not contain 101 as a substring

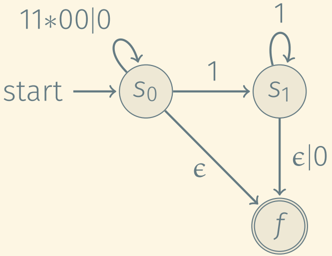


FROM NFA TO REGULAR EXPRESSION: EXAMPLE

$\mathcal{L}((0|\epsilon)(1|000*)*(0|\epsilon))$: All strings that do not contain 101 as a substring

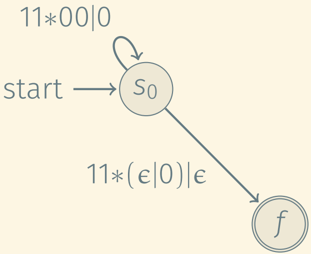


$\mathcal{L}((0|\epsilon)(1|000*)*(0|\epsilon))$: All strings that do not contain 101 as a substring

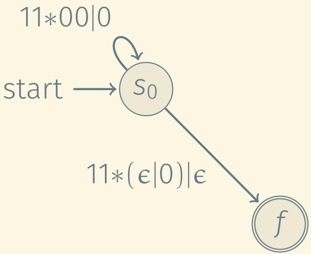


FROM NFA TO REGULAR EXPRESSION: EXAMPLE

$\mathcal{L}((0|\epsilon)(1|000*)*(0|\epsilon))$: All strings that do not contain 101 as a substring



$\mathcal{L}((0|\epsilon)(1|000*)*(0|\epsilon))$: All strings that do not contain 101 as a substring



Regular expression: $(11*00|0)*(11*(\epsilon|0)|\epsilon)$

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA

- Building a scanner
  - Regular expression $\rightarrow$ NFA $\rightarrow$ DFA
  - Minimizing the DFA

- Limitations of regular languages

ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA

- Building a scanner
  - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
  - Minimizing the DFA

- Limitations of regular languages

Scanner

A scanner produces a token (token type, value) stream from a character stream.

Scanner

A scanner produces a token (token type, value) stream from a character stream.

Modes of operation

- Complete pass produces token stream, which is then passed to the parser.
- Parser calls scanner to request next token

In either case, the scanner greedily recognizes the longest possible token.

Scanner implementation

- Hand-written, ad-hoc: Usually when speed is a concern.
- From regular expression using scanner generator: More convenient. Result:
  - Case statements representing transitions of the DFA.
  - Table representing the DFA's transition function plus driver code to implement the DFA.

Workflow

Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA $\longrightarrow$ minimized DFA

Workflow

Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA $\longrightarrow$ minimized DFA

Extensions to pure DFA:

Workflow

Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA $\longrightarrow$ minimized DFA

Extensions to pure DFA:

- Not enough to accept a token; need to know which token was accepted and its value:
  - One accepting state per token type
  - Return string read along the path to the accepting state

Workflow

Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA $\longrightarrow$ minimized DFA

Extensions to pure DFA:

- Not enough to accept a token; need to know which token was accepted and its value:
  - One accepting state per token type
  - Return string read along the path to the accepting state
- Keywords are not identifiers:
  - Look up identifier in keyword table (e.g., hash table) to see whether it is in fact a keyword

Workflow

Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA $\longrightarrow$ minimized DFA

Extensions to pure DFA:

- Not enough to accept a token; need to know which token was accepted and its value:
  - One accepting state per token type
  - Return string read along the path to the accepting state
- Keywords are not identifiers:
  - Look up identifier in keyword table (e.g., hash table) to see whether it is in fact a keyword
- “Look ahead” to distinguish tokens with common prefix (e.g., 100 vs 100.5):
  - Try to find the longest possible match by continuing to scan from an accepting state.
  - Backtrack to last accepting state when “stuck”.

EXTENDED EXAMPLE: AN INCOMPLETE SCANNER FOR PASCAL (1)

Regular expressions for the different tokens:

lparen:	`\(`
rparen:	`\)`
lbrac:	`\[`
rbrac:	`\]`
comma:	`,`
dot:	`\.`
dotdot:	`\.\.`
lt:	`<`
le:	`<=`
ident:	`[A-Za-z][A-Za-z0-9_]*`
int:	`[+-]?[0-9]+`
real:	`[+-]?[0-9]+(\.[0-9]+)?([Ee][+-]?[0-9]+)?`

…

Construction of the DFA:

Construction of the DFA:

- Turn each regular expression into an NFA,
  label each accepting state with the token represented by this expression.

Construction of the DFA:

- Turn each regular expression into an NFA,
  label each accepting state with the token represented by this expression.
- Add an NFA that consumes spaces and comments.

Construction of the DFA:

- Turn each regular expression into an NFA,
  label each accepting state with the token represented by this expression.
- Add an NFA that consumes spaces and comments.
- Join the NFA using ϵ-edges from a new start state to their start states.

Construction of the DFA:

- Turn each regular expression into an NFA,
  label each accepting state with the token represented by this expression.
- Add an NFA that consumes spaces and comments.
- Join the NFA using ϵ-edges from a new start state to their start states.
- Add ϵ-transitions from the accepting states of the spaces/comments NFA to the start state.

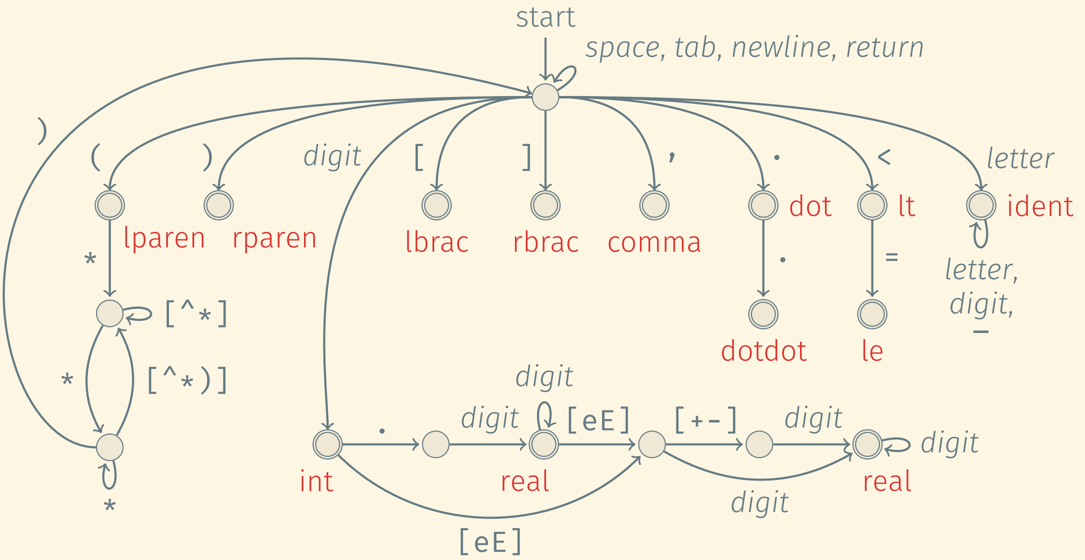
EXTENDED EXAMPLE: AN INCOMPLETE SCANNER FOR PASCAL (2)

Construction of the DFA:

- Turn each regular expression into an NFA,
  label each accepting state with the token represented by this expression.
- Add an NFA that consumes spaces and comments.
- Join the NFA using ϵ-edges from a new start state to their start states.
- Add ϵ-transitions from the accepting states of the spaces/comments NFA to the start state.
- Turn the NFA into a DFA:
  - If the tokens are unambiguous, each accepting state of the DFA, viewed as a set, includes accepting states from only one of the regular expression NFA.
  - Label the DFA accepting state with this token.

Construction of the DFA:

- Turn each regular expression into an NFA,
  label each accepting state with the token represented by this expression.
- Add an NFA that consumes spaces and comments.
- Join the NFA using ϵ-edges from a new start state to their start states.
- Add ϵ-transitions from the accepting states of the spaces/comments NFA to the start state.
- Turn the NFA into a DFA:
  - If the tokens are unambiguous, each accepting state of the DFA, viewed as a set, includes accepting states from only one of the regular expression NFA.
  - Label the DFA accepting state with this token.
- Minimize the DFA.


start
space, tab, newline, return
)
(
)
digit
[
]
,
.
<
letter
lparen
rparen
lbrac
rbrac
comma
dot
lt
ident
*
.
=
letter,
digit,
_
[^*]
dotdot
le
*
[^*)]
digit
digit
[eE]
[+-]
digit
.
digit
int
real
real
*
digit
[eE]

EXTENDED EXAMPLE: AN INCOMPLETE SCANNER FOR PASCAL (4)

Driver code:

- Whenever the scan reaches an accepting state of the spaces/comments NFA, set a start marker.
- Whenever the scan reaches an accepting state of any other NFA, set an end marker and remember the token.
- Whenever the scan reaches state $\emptyset$,
  - Go back to the end marker.
  - Report the remembered token.
  - Turn the text between start and end marker into a representation of the scanned token (integer, identifier string, ...).
  - Set the start marker to be equal to the end marker.

Goal

Given a DFA D, produce a DFA D' with the minimum number of states and such that $\mathcal{L}(D) = \mathcal{L}(D')$.

MINIMIZING THE DFA (1)

Goal

Given a DFA D, produce a DFA D' with the minimum number of states and such that $\mathcal{L}(D) = \mathcal{L}(D')$.

Idea

Group states of D into classes of equivalent states (accepting/ non-accepting, same transitions).

MINIMIZING THE DFA (1)

Goal

Given a DFA D, produce a DFA D' with the minimum number of states and such that $\mathcal{L}(D) = \mathcal{L}(D')$.

Idea

Group states of D into classes of equivalent states (accepting/ non-accepting, same transitions).

Procedure

MINIMIZING THE DFA (1)

Goal

Given a DFA D, produce a DFA D' with the minimum number of states and such that $\mathcal{L}(D) = \mathcal{L}(D')$.

Idea

Group states of D into classes of equivalent states (accepting/ non-accepting, same transitions).

Procedure

- Start with two equivalence classes: accepting and non-accepting

MINIMIZING THE DFA (1)

Goal

Given a DFA D, produce a DFA D' with the minimum number of states and such that $\mathcal{L}(D) = \mathcal{L}(D')$.

Idea

Group states of D into classes of equivalent states (accepting/ non-accepting, same transitions).

Procedure

- Start with two equivalence classes: accepting and non-accepting
- Find an equivalence class C and a letter a such that, upon reading a, the states in C transition to $k > 1$ equivalence classes $C'_1, C'_2, \ldots, C'_k$. Partition C into subclasses $C_1, C_2, \ldots, C_k$ such that, upon reading a, the states in C_i transition to states in C'_i.

MINIMIZING THE DFA (1)

Goal

Given a DFA D, produce a DFA D' with the minimum number of states and such that $\mathcal{L}(D) = \mathcal{L}(D')$.

Idea

Group states of D into classes of equivalent states (accepting/non-accepting, same transitions).

Procedure

- Start with two equivalence classes: accepting and non-accepting
- Find an equivalence class C and a letter a such that, upon reading a, the states in C transition to $k > 1$ equivalence classes $C'_1, C'_2, \ldots, C'_k$. Partition C into subclasses $C_1, C_2, \ldots, C_k$ such that, upon reading a, the states in C_i transition to states in C'_i.
- Repeat until no such "partitionable" equivalence class C can be found.

MINIMIZING THE DFA (1)

Goal

Given a DFA D, produce a DFA D' with the minimum number of states and such that $\mathcal{L}(D) = \mathcal{L}(D')$.

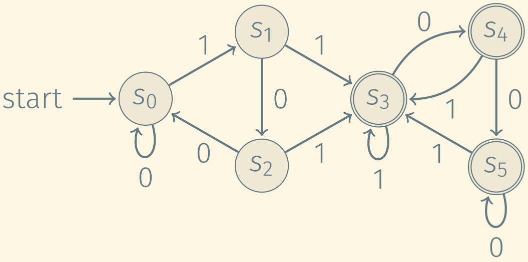
Idea

Group states of D into classes of equivalent states (accepting/ non-accepting, same transitions).

Procedure

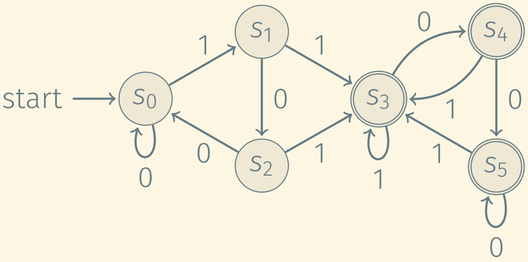
- Start with two equivalence classes: accepting and non-accepting
- Find an equivalence class C and a letter a such that, upon reading a, the states in C transition to $k > 1$ equivalence classes $C'_1, C'_2, \ldots, C'_k$. Partition C into subclasses $C_1, C_2, \ldots, C_k$ such that, upon reading a, the states in C_i transition to states in C'_i.
- Repeat until no such "partitionable" equivalence class C can be found.
- The final set of equivalence classes is the set of states of the minimized DFA.

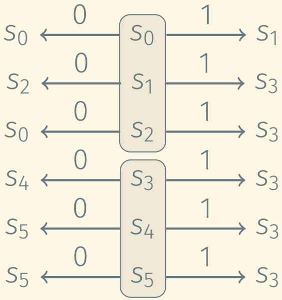
MINIMIZING THE DFA: EXAMPLE

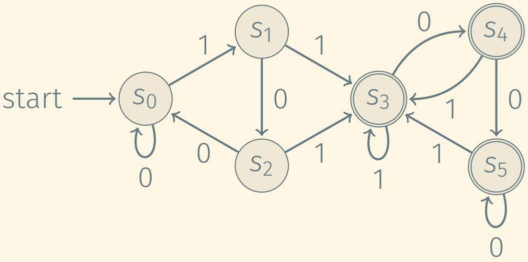


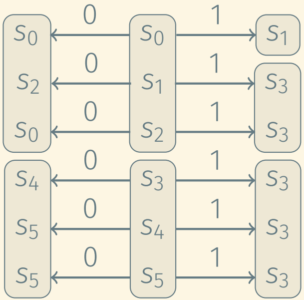
$$S_0 \xleftarrow{0} S_0 \xrightarrow{1} S_1$$
$$S_2 \xleftarrow{0} S_1 \xrightarrow{1} S_3$$
$$S_0 \xleftarrow{0} S_2 \xrightarrow{1} S_3$$
$$S_4 \xleftarrow{0} S_3 \xrightarrow{1} S_3$$
$$S_5 \xleftarrow{0} S_4 \xrightarrow{1} S_3$$
$$S_5 \xleftarrow{0} S_5 \xrightarrow{1} S_3$$

MINIMIZING THE DFA: EXAMPLE

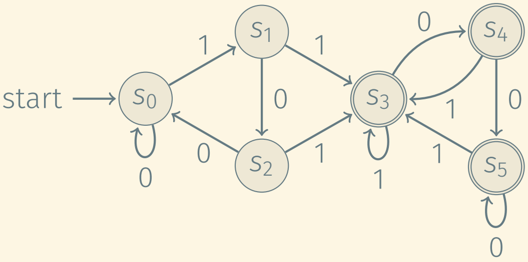


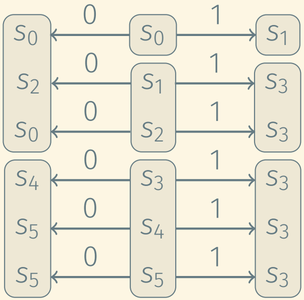


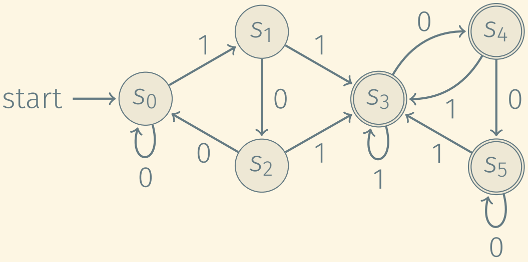

start
s_0
s_1
s_2
s_3
s_4
s_5
0
1

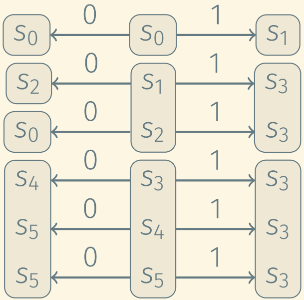

s_0
s_2
s_0
s_4
s_5
s_5
s_0
s_1
s_2
s_3
s_4
s_5
s_1
s_3
s_3
s_3
s_3
s_3
0
1

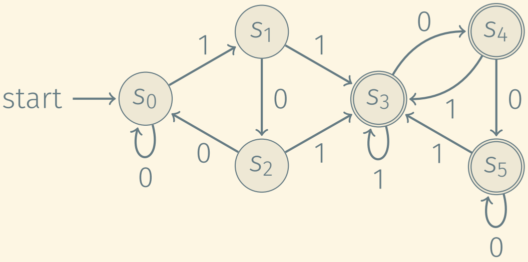
MINIMIZING THE DFA: EXAMPLE

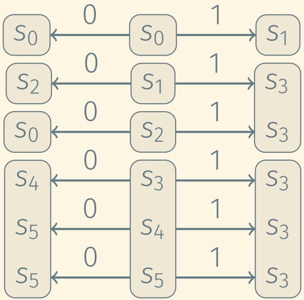




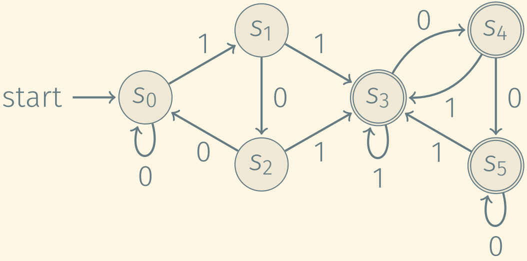

start
s_0
s_1
s_2
s_3
s_4
s_5
0
1

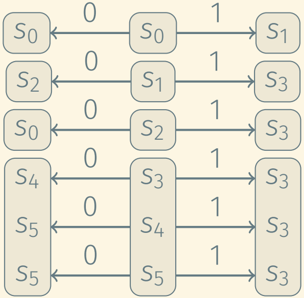

$s_0 \xleftarrow{0} s_0 \xrightarrow{1} s_1$
$s_2 \xleftarrow{0} s_1 \xrightarrow{1} s_3$
$s_0 \xleftarrow{0} s_2 \xrightarrow{1} s_3$
$s_4 \xleftarrow{0} s_3 \xrightarrow{1} s_3$
$s_5 \xleftarrow{0} s_4 \xrightarrow{1} s_3$
$s_5 \xleftarrow{0} s_5 \xrightarrow{1} s_3$


start
s_0
s_1
s_2
s_3
s_4
s_5
0
1

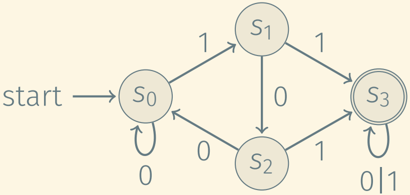

s_0 $\xleftarrow{0}$ s_0 $\xrightarrow{1}$ s_1
s_2 $\xleftarrow{0}$ s_1 $\xrightarrow{1}$ s_3
s_0 $\xleftarrow{0}$ s_2 $\xrightarrow{1}$ s_3
s_4 $\xleftarrow{0}$ s_3 $\xrightarrow{1}$ s_3
s_5 $\xleftarrow{0}$ s_4 $\xrightarrow{1}$ s_3
s_5 $\xleftarrow{0}$ s_5 $\xrightarrow{1}$ s_3

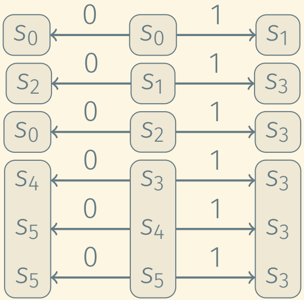
MINIMIZING THE DFA: EXAMPLE





MINIMIZING THE DFA: EXAMPLE





MINIMIZING THE DFA (2)

The described procedure ensures that $\mathcal{L}(D) = \mathcal{L}(D')$ but does not distinguish between different types of accepting states (corresponding to tokens).

To distinguish between different types of accepting states, start with one equivalence class per type of accepting state.

ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA

- Building a scanner
    - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
    - Minimizing the DFA

- Limitations of regular languages

ROAD MAP

- Regular languages
- Regular expressions
- Deterministic finite automata (DFA)
- Non-deterministic finite automata (NFA)
- Expressive power of DFA and NFA
- Equivalence of regular expressions, DFA, and NFA

- Building a scanner
  - Regular expression $\longrightarrow$ NFA $\longrightarrow$ DFA
  - Minimizing the DFA

- Limitations of regular languages

If R and S are regular languages, then so are

HOW GENERAL ARE REGULAR LANGUAGES?

If R and S are regular languages, then so are

- RS, $R \cup S$, R^*

If R and S are regular languages, then so are

- RS, $R \cup S$, R^*

  By definition

If R and S are regular languages, then so are

- RS, $R \cup S$, R^*

  By definition

- $\Sigma^* \setminus R$ (the complement of R)

HOW GENERAL ARE REGULAR LANGUAGES?

If R and S are regular languages, then so are

- RS, $R \cup S$, R^*

  By definition

- $\Sigma^* \setminus R$ (the complement of R)

  Build a DFA for $\Sigma^* \setminus R$ from a DFA for R by making accepting states non-accepting and vice versa.

If R and S are regular languages, then so are

- RS, $R \cup S$, R^*

  By definition

- $\Sigma^* \setminus R$ (the complement of R)

  Build a DFA for $\Sigma^* \setminus R$ from a DFA for R by making accepting states non-accepting and vice versa.

- $\overleftarrow{R} = \{\overleftarrow{\sigma} \mid \sigma \in R\}$, where $\overleftarrow{\sigma}$ is σ written backwards

HOW GENERAL ARE REGULAR LANGUAGES?

If R and S are regular languages, then so are

- RS, $R \cup S$, R^*

  By definition

- $\Sigma^* \setminus R$ (the complement of R)

  Build a DFA for $\Sigma^* \setminus R$ from a DFA for R by making accepting states non-accepting and vice versa.

- $\overleftarrow{R} = \{\overleftarrow{\sigma} \mid \sigma \in R\}$, where $\overleftarrow{\sigma}$ is σ written backwards

  A regular expression for R "written backwards" is a regular expression for $\overleftarrow{R}$.

HOW GENERAL ARE REGULAR LANGUAGES?

If R and S are regular languages, then so are

- RS, $R \cup S$, R^*

  By definition

- $R \cap S$

- $\Sigma^* \setminus R$ (the complement of R)

  Build a DFA for $\Sigma^* \setminus R$ from a DFA for R by making accepting states non-accepting and vice versa.

- $\overleftarrow{R} = \{\overleftarrow{\sigma} \mid \sigma \in R\}$, where $\overleftarrow{\sigma}$ is σ written backwards

  A regular expression for R "written backwards" is a regular expression for $\overleftarrow{R}$.

HOW GENERAL ARE REGULAR LANGUAGES?

If R and S are regular languages, then so are

- RS, $R \cup S$, R^*

  By definition

- $R \cap S$

  $R \cap S = \Sigma^* \setminus ((\Sigma^* \setminus R) \cup (\Sigma^* \setminus S))$

- $\Sigma^* \setminus R$ (the complement of R)

  Build a DFA for $\Sigma^* \setminus R$ from a DFA for R by making accepting states non-accepting and vice versa.

- $\overleftarrow{R} = \{\overleftarrow{\sigma} \mid \sigma \in R\}$, where $\overleftarrow{\sigma}$ is σ written backwards

  A regular expression for R "written backwards" is a regular expression for $\overleftarrow{R}$.

HOW GENERAL ARE REGULAR LANGUAGES?

If R and S are regular languages, then so are

- RS, $R \cup S$, R^*

  By definition

- $R \cap S$

  $R \cap S = \Sigma^* \setminus ((\Sigma^* \setminus R) \cup (\Sigma^* \setminus S))$

- $R \setminus S$

- $\Sigma^* \setminus R$ (the complement of R)

  Build a DFA for $\Sigma^* \setminus R$ from a DFA for R by making accepting states non-accepting and vice versa.

- $\overleftarrow{R} = \{\overleftarrow{\sigma} \mid \sigma \in R\}$, where $\overleftarrow{\sigma}$ is σ written backwards

  A regular expression for R "written backwards" is a regular expression for $\overleftarrow{R}$.

HOW GENERAL ARE REGULAR LANGUAGES?

If R and S are regular languages, then so are

- RS, $R \cup S$, R^*

  By definition

- $R \cap S$

  $R \cap S = \Sigma^* \setminus ((\Sigma^* \setminus R) \cup (\Sigma^* \setminus S))$

- $R \setminus S$

  $R \setminus S = R \cap (\Sigma^* \setminus S)$

- $\Sigma^* \setminus R$ (the complement of R)

  Build a DFA for $\Sigma^* \setminus R$ from a DFA for R by making accepting states non-accepting and vice versa.

- $\overleftarrow{R} = \{\overleftarrow{\sigma} \mid \sigma \in R\}$, where $\overleftarrow{\sigma}$ is σ written backwards

  A regular expression for R "written backwards" is a regular expression for $\overleftarrow{R}$.

NOT ALL LANGUAGES ARE REGULAR

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

$\Rightarrow$ The language $\mathcal{L} = \{0^n1^n \mid n \geq 0\}$ is not regular!

NOT ALL LANGUAGES ARE REGULAR

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

$\Rightarrow$ **The language $\mathcal{L} = \{0^n1^n \mid n \geq 0\}$ is not regular!**

- Assume $\mathcal{L}$ is regular and let $n_{\mathcal{L}}$ be as in the Pumping Lemma.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

$\Rightarrow$ The language $\mathcal{L} = \{0^n1^n \mid n \geq 0\}$ is not regular!

- Assume $\mathcal{L}$ is regular and let $n_{\mathcal{L}}$ be as in the Pumping Lemma.
- Let $\sigma = 0^{n_{\mathcal{L}}}1^{n_{\mathcal{L}}} \in \mathcal{L}$.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_\mathcal{L}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_\mathcal{L}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_\mathcal{L}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

$\Rightarrow$ **The language $\mathcal{L} = \{0^n1^n \mid n \geq 0\}$ is not regular!**

- Assume $\mathcal{L}$ is regular and let $n_\mathcal{L}$ be as in the Pumping Lemma.
- Let $\sigma = 0^{n_\mathcal{L}}1^{n_\mathcal{L}} \in \mathcal{L}$.
- Then $\sigma = \alpha\beta\gamma$ with $|\alpha\beta| \leq n_\mathcal{L}$ and $|\beta| > 0$ and $\alpha\beta\beta\gamma \in \mathcal{L}$.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

$\Rightarrow$ **The language $\mathcal{L} = \{0^n1^n \mid n \geq 0\}$ is not regular!**

- Assume $\mathcal{L}$ is regular and let $n_{\mathcal{L}}$ be as in the Pumping Lemma.
- Let $\sigma = 0^{n_{\mathcal{L}}}1^{n_{\mathcal{L}}} \in \mathcal{L}$.
- Then $\sigma = \alpha\beta\gamma$ with $|\alpha\beta| \leq n_{\mathcal{L}}$ and $|\beta| > 0$ and $\alpha\beta\beta\gamma \in \mathcal{L}$.
- Since $|\alpha\beta| \leq n_{\mathcal{L}}$, we have $\alpha = 0^k$ and $\beta = 0^m$, where $m = |\beta| > 0$.

NOT ALL LANGUAGES ARE REGULAR

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

$\Rightarrow$ **The language $\mathcal{L} = \{0^n1^n \mid n \geq 0\}$ is not regular!**

- Assume $\mathcal{L}$ is regular and let $n_{\mathcal{L}}$ be as in the Pumping Lemma.
- Let $\sigma = 0^{n_{\mathcal{L}}}1^{n_{\mathcal{L}}} \in \mathcal{L}$.
- Then $\sigma = \alpha\beta\gamma$ with $|\alpha\beta| \leq n_{\mathcal{L}}$ and $|\beta| > 0$ and $\alpha\beta\beta\gamma \in \mathcal{L}$.
- Since $|\alpha\beta| \leq n_{\mathcal{L}}$, we have $\alpha = 0^k$ and $\beta = 0^m$, where $m = |\beta| > 0$.
- Thus, $\alpha\beta\beta\gamma = 0^{m+n_{\mathcal{L}}}1^{n_{\mathcal{L}}} \notin \mathcal{L}$, a contradiction.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

Let $D = (S, \Sigma, \delta, s_0, F)$ be a DFA such that $\mathcal{L} = \mathcal{L}(D)$.

PROOF OF THE PUMPING LEMMA

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

Let $D = (S, \Sigma, \delta, s_0, F)$ be a DFA such that $\mathcal{L} = \mathcal{L}(D)$.

Let $n_{\mathcal{L}} = |S| + 1$.

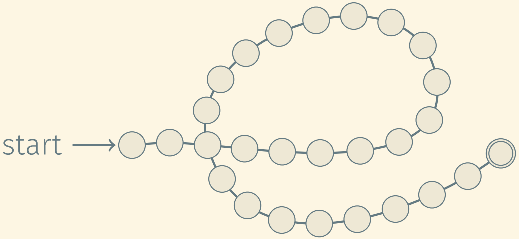
Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

Let $D = (S, \Sigma, \delta, s_0, F)$ be a DFA such that $\mathcal{L} = \mathcal{L}(D)$.

Let $n_{\mathcal{L}} = |S| + 1$.



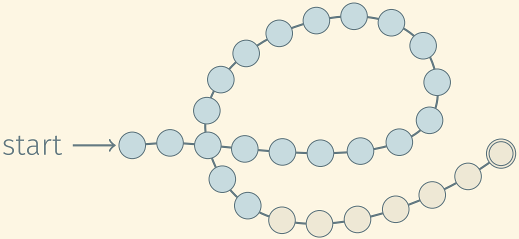
Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

Let $D = (S, \Sigma, \delta, s_0, F)$ be a DFA such that $\mathcal{L} = \mathcal{L}(D)$.

Let $n_{\mathcal{L}} = |S| + 1$.



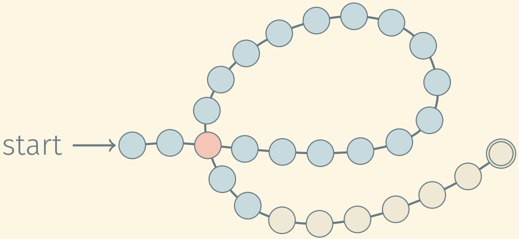
Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

Let $D = (S, \Sigma, \delta, s_0, F)$ be a DFA such that $\mathcal{L} = \mathcal{L}(D)$.

Let $n_{\mathcal{L}} = |S| + 1$.



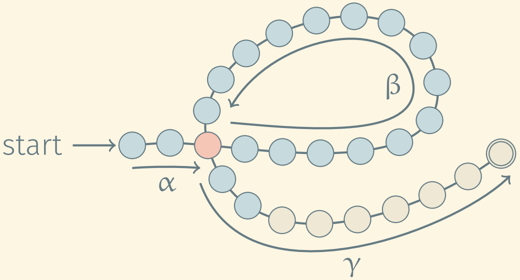
Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

Let $D = (S, \Sigma, \delta, s_0, F)$ be a DFA such that $\mathcal{L} = \mathcal{L}(D)$.

Let $n_{\mathcal{L}} = |S| + 1$.



Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

- $\mathcal{L} = \{(^m)^m \mid m \geq 0\}$ is not regular.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

- $\mathcal{L} = \{(^m)^m \mid m \geq 0\}$ **is not regular.**

  Same structure as $\mathcal{L}' = \{0^n1^n \mid n \geq 0\}$.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

- **$\mathcal{L} = \{(^m)^m \mid m \geq 0\}$ is not regular.**

  Same structure as $\mathcal{L}' = \{0^n1^n \mid n \geq 0\}$.

- **$\mathcal{L} = \{a^p \mid p$ is a prime number$\}$ is not regular.**

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

- **$\mathcal{L} = \{(^m)^m \mid m \geq 0\}$ is not regular.**

  Same structure as $\mathcal{L}' = \{0^n1^n \mid n \geq 0\}$.

- **$\mathcal{L} = \{a^p \mid p$ is a prime number$\}$ is not regular.**
  - Assume $\mathcal{L}$ is regular.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

- **$\mathcal{L} = \{(^m)^m \mid m \geq 0\}$ is not regular.**

  Same structure as $\mathcal{L}' = \{0^n1^n \mid n \geq 0\}$.

- **$\mathcal{L} = \{a^p \mid p$ is a prime number$\}$ is not regular.**
  - Assume $\mathcal{L}$ is regular.
  - Choose prime number $p \geq n_{\mathcal{L}} + 2$
    $\Rightarrow \sigma = a^p \in \mathcal{L}$.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

- **$\mathcal{L} = \{(^m)^m \mid m \geq 0\}$ is not regular.**

  Same structure as $\mathcal{L}' = \{0^n1^n \mid n \geq 0\}$.

- **$\mathcal{L} = \{\mathrm{a}^p \mid p \text{ is a prime number}\}$ is not regular.**
  - Assume $\mathcal{L}$ is regular.
  - Choose prime number $p \geq n_{\mathcal{L}} + 2$ $\Rightarrow \sigma = \mathrm{a}^p \in \mathcal{L}$.
  - $\sigma = \alpha\beta\gamma$, where $\alpha = \mathrm{a}^a$, $\beta = \mathrm{a}^b$, $a + b \leq n_{\mathcal{L}}$ and $b > 0$.

PUMPING LEMMA: MORE APPLICATIONS

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

- **$\mathcal{L} = \{(^m)^m \mid m \geq 0\}$ is not regular.**

  Same structure as $\mathcal{L}' = \{0^n1^n \mid n \geq 0\}$.

- **$\mathcal{L} = \{a^p \mid p$ is a prime number$\}$ is not regular.**
  - Assume $\mathcal{L}$ is regular.
  - Choose prime number $p \geq n_{\mathcal{L}} + 2$ $\Rightarrow \sigma = a^p \in \mathcal{L}$.
  - $\sigma = \alpha\beta\gamma$, where $\alpha = a^a$, $\beta = a^b$, $a + b \leq n_{\mathcal{L}}$ and $b > 0$.
  - $\alpha\beta^c\gamma \in \mathcal{L}$, where $c = |\alpha\gamma| = p - b \geq 2$.

Pumping Lemma

For every regular language $\mathcal{L}$, there exists a constant $n_{\mathcal{L}}$ such that every $\sigma \in \mathcal{L}$ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as $\sigma = \alpha\beta\gamma$ with

- $|\alpha\beta| \leq n_{\mathcal{L}}$,
- $|\beta| > 0$, and
- $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

- **$\mathcal{L} = \{(^m)^m \mid m \geq 0\}$ is not regular.**

  Same structure as $\mathcal{L}' = \{0^n1^n \mid n \geq 0\}$.

- **$\mathcal{L} = \{a^p \mid p$ is a prime number$\}$ is not regular.**
  - Assume $\mathcal{L}$ is regular.
  - Choose prime number $p \geq n_{\mathcal{L}} + 2$ $\Rightarrow \sigma = a^p \in \mathcal{L}$.
  - $\sigma = \alpha\beta\gamma$, where $\alpha = a^a$, $\beta = a^b$, $a + b \leq n_{\mathcal{L}}$ and $b > 0$.
  - $\alpha\beta^c\gamma \in \mathcal{L}$, where $c = |\alpha\gamma| = p - b \geq 2$.
  - However, $|\alpha\beta^c\gamma| = (b + 1)c$, which is not prime because $b + 1 \geq 2$ and $c \geq 2$. Contradiction.

SUMMARY

- Parsing is complex $\Rightarrow$ Apply to a token stream rather than a character stream.

SUMMARY

- Parsing is complex $\Rightarrow$ Apply to a token stream rather than a character stream.
- Lexical analysis turns character stream into more compact token stream.

SUMMARY

- Parsing is complex ⇒ Apply to a token stream rather than a character stream.
- Lexical analysis turns character stream into more compact token stream.
- Regular languages are general enough to capture the structure of tokens but not general enough to capture the structure of programming languages.

SUMMARY

- Parsing is complex $\Rightarrow$ Apply to a token stream rather than a character stream.
- Lexical analysis turns character stream into more compact token stream.
- Regular languages are general enough to capture the structure of tokens but not general enough to capture the structure of programming languages.
- There exist languages that are not regular.

SUMMARY

- Parsing is complex $\Rightarrow$ Apply to a token stream rather than a character stream.
- Lexical analysis turns character stream into more compact token stream.
- Regular languages are general enough to capture the structure of tokens but not general enough to capture the structure of programming languages.
- There exist languages that are not regular.
- Regular languages are described using regular expressions and recognized using DFA.

SUMMARY

- Parsing is complex $\Rightarrow$ Apply to a token stream rather than a character stream.
- Lexical analysis turns character stream into more compact token stream.

- Regular languages are general enough to capture the structure of tokens but not general enough to capture the structure of programming languages.
- There exist languages that are not regular.

- Regular languages are described using regular expressions and recognized using DFA.
- DFA are very simple machines that can be implemented very efficiently.

SUMMARY

- Parsing is complex $\Rightarrow$ Apply to a token stream rather than a character stream.
- Lexical analysis turns character stream into more compact token stream.

- Regular languages are general enough to capture the structure of tokens but not general enough to capture the structure of programming languages.
- There exist languages that are not regular.

- Regular languages are described using regular expressions and recognized using DFA.
- DFA are very simple machines that can be implemented very efficiently.
- NFA are mainly a tool for translating regular expressions to DFA.

SUMMARY

- Parsing is complex $\Rightarrow$ Apply to a token stream rather than a character stream.
- Lexical analysis turns character stream into more compact token stream.

- Regular languages are general enough to capture the structure of tokens but not general enough to capture the structure of programming languages.
- There exist languages that are not regular.

- Regular languages are described using regular expressions and recognized using DFA.
- DFA are very simple machines that can be implemented very efficiently.
- NFA are mainly a tool for translating regular expressions to DFA.
- Lexical analysis requires some simple extensions to DFA because we need to know which token wa accepted and we need to support greediness/backtracking.