

Scanner: Lexical Analysis

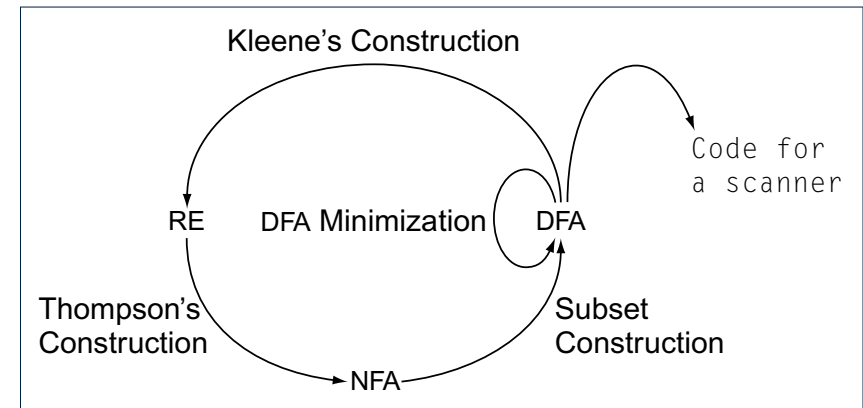
Readings: EAC2 Chapter 2



EECS4302 A:
Compilers and Interpreters
Fall 2026

CHEN-WEI WANG

Scanner: Formulation & Implementation

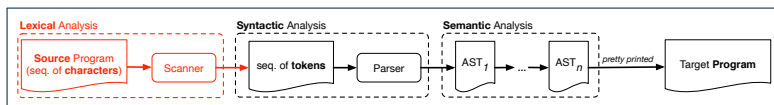


3 of 68

Scanner in Context



Recall:



- Treats the input programs as a **a sequence of characters**
- Applies rules **recognizing** character sequences as **tokens** [**lexical** analysis]
- Upon termination:
 - Reports character sequences not recognizable as tokens
 - Produces a **a sequence of tokens**
- Only part of compiler touching **every character** in input program.
- Tokens **recognizable** by scanner constitute a **regular language**.

2 of 68

Alphabets



An **alphabet** is a **finite, nonempty** set of symbols.

- The convention is to write Σ , possibly with a informative subscript, to denote the alphabet in question.
- Use either a **set enumeration** or a **set comprehension** to define your own alphabet.
 - e.g., $\Sigma_{eng} = \{a, b, \dots, z, A, B, \dots, Z\}$ [the English alphabet]
 - e.g., $\Sigma_{bin} = \{0, 1\}$ [the binary alphabet]
 - e.g., $\Sigma_{dec} = \{d \mid 0 \leq d \leq 9\}$ [the decimal alphabet]
 - e.g., Σ_{key} [the keyboard alphabet]

4 of 68

Strings (1)

- A **string** or a **word** is **finite** sequence of symbols chosen from some **alphabet**.
 e.g., Oxford is a string over the English alphabet Σ_{eng}
 e.g., 01010 is a string over the binary alphabet Σ_{bin}
 e.g., 01010.01 is **not** a string over Σ_{bin}
 e.g., 57 is a string over the decimal alphabet Σ_{dec}
- It is **not** correct to say, e.g., $01010 \in \Sigma_{bin}$ [Why?]
- The **length** of a string w , denoted as $|w|$, is the number of characters it contains.
 - e.g., $|Oxford| = 6$
 - ϵ is the **empty string** ($|\epsilon| = 0$) that may be from any alphabet.
- Given two strings x and y , their **concatenation**, denoted as xy , is a new string formed by a copy of x followed by a copy of y .
 - e.g., Let $x = 01101$ and $y = 110$, then $xy = 01101110$
 - The empty string ϵ is the **identity for concatenation**:
 $\epsilon w = w = w\epsilon$ for any string w

5 of 68

Strings (2)

- Given an **alphabet** Σ , we write Σ^k , where $k \in \mathbb{N}$, to denote the **set of strings of length k from Σ**

$$\Sigma^k = \{w \mid \underbrace{w \text{ is a string over } \Sigma \wedge |w| = k}_{\text{more formal?}}\}$$

more formal?

- e.g., $\{0, 1\}^2 = \{00, 01, 10, 11\}$
- Given Σ , Σ^0 is $\{\epsilon\}$
- Given Σ , Σ^+ is the **set of nonempty strings**.

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots = \{w \mid w \in \Sigma^k \wedge k > 0\} = \bigcup_{k>0} \Sigma^k$$

- Given Σ , Σ^* is the **set of strings of all possible lengths**.

$$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$$

6 of 68

Review Exercises: Strings

- What is $|\{a, b, \dots, z\}^5|$?
- Enumerate, in a systematic manner, the set $\{a, b, c\}^4$.
- Explain the difference between Σ and Σ^1 .
- Prove or disprove: $\Sigma_1 \subseteq \Sigma_2 \Rightarrow \Sigma_1^* \subseteq \Sigma_2^*$

7 of 68

Languages

- A **language L over Σ** (where $|\Sigma|$ is finite) is a set of strings s.t.
 $L \subseteq \Sigma^*$

- When useful, include an informative subscript to denote the **language L** in question.
 - e.g., The language of **compilable** Java programs
 $L_{Java} = \{prog \mid prog \in \Sigma_{key}^* \wedge prog \text{ compiles in Eclipse}\}$
Note. $prog$ compiling means **no lexical**, **syntactical**, or **type** errors.
 - e.g., The language of strings with n 0's followed by n 1's ($n \geq 0$)
 $\{\epsilon, 01, 0011, 000111, \dots\} = \{0^n 1^n \mid n \geq 0\}$
 - e.g., The language of strings with an equal number of 0's and 1's
 $\{\epsilon, 01, 10, 0011, 0101, 0110, 1100, 1010, 1001, \dots\}$
 $= \{w \mid \# \text{ of } 0\text{'s in } w = \# \text{ of } 1\text{'s in } w\}$

8 of 68

Review Exercises: Languages



1. Use **set comprehensions** to define the following **languages**.
Be as **formal** as possible.
 - A language over $\{0, 1\}$ consisting of strings beginning with some 0's (possibly none) followed by at least as many 1's.
 - A language over $\{a, b, c\}$ consisting of strings beginning with some a's (possibly none), followed by some b's and then some c's, s.t. the # of a's is at least as many as the sum of #'s of b's and c's.
2. Explain the difference between the two languages $\{\epsilon\}$ and $\emptyset$.
3. Justify that Σ^* , $\emptyset$, and $\{\epsilon\}$ are all languages over Σ .
4. Prove or disprove: If L is a language over Σ , and $\Sigma_2 \supseteq \Sigma$, then L is also a language over Σ_2 .
Hint: Prove that $\Sigma \subseteq \Sigma_2 \wedge L \subseteq \Sigma^* \Rightarrow L \subseteq \Sigma_2^*$
5. Prove or disprove: If L is a language over Σ , and $\Sigma_2 \subseteq \Sigma$, then L is also a language over Σ_2 .
Hint: Prove that $\Sigma_2 \subseteq \Sigma \wedge L \subseteq \Sigma^* \Rightarrow L \subseteq \Sigma_2^*$

9 of 68

Problems



- Given a **language** L over some **alphabet** Σ , a **problem** is the **decision** on whether or not a given **string** w is a member of L .

$$w \in L$$

Is this equivalent to deciding $w \in \Sigma^*$? [**No**]

$w \in \Sigma^* \Rightarrow w \in L$ is **not** necessarily true.

- e.g., The Java compiler solves the problem of **deciding** if a user-supplied **string of symbols** is a **member** of L_{Java} .

10 of 68

Regular Expressions (RE): Introduction



- **Regular expressions** (RegExp's) are:
 - A type of language-defining notation
 - This is **similar** to the **equally-expressive** **DFA**, **NFA**, and **ϵ -NFA**.
 - **Textual** and look just like a programming language
 - e.g., Set of strings denoted by $01^* + 10^*$? [specify formally]
 $L = \{0x \mid x \in \{1\}^*\} \cup \{1x \mid x \in \{0\}^*\}$
 - e.g., Set of strings denoted by $(0^*10^*10^*)^*10^*$?
 $L = \{w \mid w \text{ has odd \# of } 1\text{'s}\}$
 - This is **dissimilar** to the diagrammatic **DFA**, **NFA**, and **ϵ -NFA**.
 - RegExp's can be considered as a "user-friendly" alternative to **NFA** for describing software components. [e.g., text search]
 - Writing a RegExp is like writing an **algebraic** expression, using the defined operators, e.g., $((4 + 3) * 5) \% 6$
- Despite the programming convenience they provide, **RegExp's**, **DFA**, **NFA**, and **ϵ -NFA** are all **provably equivalent**.
 - They are capable of defining **all** and **only** regular languages.

11 of 68

RE: Language Operations (1)



- Given Σ of input alphabets, the **simplest** RegExp is? [$s \in \Sigma^1$]
 - e.g., Given $\Sigma = \{a, b, c\}$, expression **a** denotes the language $\{a\}$ consisting of a single string **a**.
- Given two languages $L, M \in \Sigma^*$, there are 3 operators for building a **larger language** out of them:

1. Union

$$L \cup M = \{w \mid w \in L \vee w \in M\}$$

In the textual form, we write $+$ for union.

2. Concatenation

$$LM = \{xy \mid x \in L \wedge y \in M\}$$

In the textual form, we write either $.$ or nothing at all for concatenation.

12 of 68

RE: Language Operations (2)



3. Kleene Closure (or Kleene Star)

$$L^* = \bigcup_{i \geq 0} L^i$$

where

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^1 &= L \\ L^2 &= \{x_1 x_2 \mid x_1 \in L \wedge x_2 \in L\} \\ &\vdots \\ L^i &= \{ \underbrace{x_1 x_2 \dots x_i}_{i \text{ concatenations}} \mid x_j \in L \wedge 1 \leq j \leq i \} \\ &\dots \end{aligned}$$

In the textual form, we write $*$ for closure.

Question: What is $|L^i|$ ($i \in \mathbb{N}$)?

$$\begin{bmatrix} |L^i| \\ |L| \end{bmatrix}$$

Question: Given that $L = \{0\}^*$, what is L^* ?

13 of 68

RE: Construction (1)



We may build **regular expressions** *recursively*:

- Each (**basic** or **recursive**) form of regular expressions denotes a **language** (i.e., a set of strings that it accepts).

- Base Case:**

- Constants ϵ and $\emptyset$ are regular expressions.

$$\begin{aligned} L(\epsilon) &= \{\epsilon\} \\ L(\emptyset) &= \emptyset \end{aligned}$$

- An input symbol $a \in \Sigma$ is a regular expression.

$$L(a) = \{a\}$$

If we want a regular expression for the language consisting of only the string $w \in \Sigma^*$, we write w as the regular expression.

- Variables such as **L**, **M**, etc., might also denote languages.

14 of 68

RE: Construction (2)



- Recursive Case:** Given that E and F are regular expressions:
 - The union $E + F$ is a regular expression.

$$L(E + F) = L(E) \cup L(F)$$

- The concatenation EF is a regular expression.

$$L(EF) = L(E)L(F)$$

- Kleene closure of E is a regular expression.

$$L(E^*) = (L(E))^*$$

- A parenthesized E is a regular expression.

$$L(E) = L(E)$$

15 of 68

RE: Construction (3)



Exercises:

- $\emptyset + L$
- $\emptyset L$
- $\emptyset^*$

$$[\emptyset + L = L = \emptyset + L]$$

$$[\emptyset L = \emptyset = L\emptyset]$$

$$\begin{aligned} \emptyset^* &= \emptyset^0 \cup \emptyset^1 \cup \emptyset^2 \cup \dots \\ &= \{\epsilon\} \cup \emptyset \cup \emptyset \cup \dots \\ &= \{\epsilon\} \end{aligned}$$

- $\emptyset^* L$

$$[\emptyset^* L = L = L\emptyset^*]$$

16 of 68

RE: Construction (4)



Write a regular expression for the following language

$\{ w \mid w \text{ has alternating } 0\text{'s and } 1\text{'s} \}$

- Would $(01)^*$ work? [alternating 10's?]
- Would $(01)^* + (10)^*$ work? [starting and ending with 1?]
- $0(10)^* + (01)^* + (10)^* + 1(01)^*$
- It seems that:
 - 1st and 3rd terms have $(10)^*$ as the common factor.
 - 2nd and 4th terms have $(01)^*$ as the common factor.
- Can we simplify the above regular expression?
- $(\epsilon + 0)(10)^* + (\epsilon + 1)(01)^*$

17 of 68

RE: Review Exercises



Write the regular expressions to describe the following languages:

- $\{ w \mid w \text{ ends with } 01 \}$
- $\{ w \mid w \text{ contains } 01 \text{ as a substring} \}$
- $\{ w \mid w \text{ contains no more than three consecutive } 1\text{'s} \}$
- $\{ w \mid w \text{ ends with } 01 \vee w \text{ has an odd \# of } 0\text{'s} \}$
-

$$\left\{ sx.y \mid \begin{array}{l} s \in \{+, -, \epsilon\} \\ x \in \Sigma_{dec}^* \\ y \in \Sigma_{dec}^* \\ \wedge \neg(x = \epsilon \wedge y = \epsilon) \end{array} \right\}$$

-
- $\left\{ xy \mid \begin{array}{l} x \in \{0,1\}^* \wedge y \in \{0,1\}^* \\ x \text{ has alternating } 0\text{'s and } 1\text{'s} \\ \wedge y \text{ has an odd \# } 0\text{'s and an odd \# } 1\text{'s} \end{array} \right\}$

18 of 68

RE: Operator Precedence



- In an order of **decreasing precedence**:
 - Kleene star operator
 - Concatenation operator
 - Union operator
- When necessary, use **parentheses** to force the intended order of evaluation.
- e.g.,
 - 10^* vs. $(10)^*$ [10^* is equivalent to $1(0^*)$]
 - $01^* + 1$ vs. $0(1^* + 1)$ [$01^* + 1$ is equivalent to $(0(1^*)) + (1)$]
 - $0 + 1^*$ vs. $(0 + 1)^*$ [$0 + 1^*$ is equivalent to $(0) + (1^*)$]

19 of 68

DFA: Deterministic Finite Automata (1.1)



- A **deterministic finite automata (DFA)** is a **finite state machine (FSM)** that **accepts** (or **recognizes**) a pattern of behaviour.
 - For **lexical** analysis, we study patterns of **strings** (i.e., how **alphabet** symbols are ordered).
 - Unless otherwise specified, we consider strings in $\{0,1\}^*$
 - Each pattern contains the set of satisfying strings.
 - We describe the patterns of strings using **set comprehensions**:
 - $\{ w \mid w \text{ has an odd number of } 0\text{'s} \}$
 - $\{ w \mid w \text{ has an even number of } 1\text{'s} \}$
 - $\left\{ w \mid \begin{array}{l} w \neq \epsilon \\ \wedge w \text{ has equal \# of alternating } 0\text{'s and } 1\text{'s} \end{array} \right\}$
 - $\{ w \mid w \text{ contains } 01 \text{ as a substring} \}$
 - $\left\{ w \mid \begin{array}{l} w \text{ has an even number of } 0\text{'s} \\ \wedge w \text{ has an odd number of } 1\text{'s} \end{array} \right\}$
- Given a pattern description, we design a **DFA** that accepts it.
 - The resulting **DFA** can be transformed into an **executable program**.

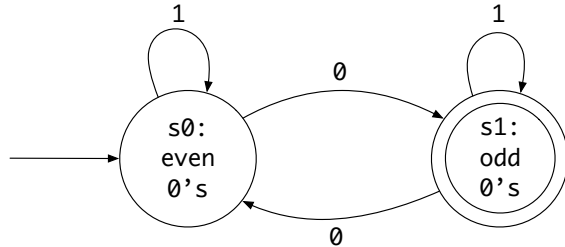
20 of 68

DFA: Deterministic Finite Automata (1.2)



- The **transition diagram** below defines a DFA which **accepts/recognizes** exactly the language

$\{ w \mid w \text{ has an odd number of } 0\text{'s} \}$



- Each **incoming** or **outgoing** arc (called a **transition**) corresponds to an input alphabet symbol.
- s_0 with an unlabelled **incoming** transition is the **start state**.
- s_1 drawn as a double circle is a **final state**.
- All states have **outgoing** transitions covering $\{0, 1\}$.

21 of 68

Review Exercises: Drawing DFAs



Draw the transition diagrams for DFAs which accept other example string patterns:

- $\{ w \mid w \text{ has an even number of } 1\text{'s} \}$
- $\{ w \mid w \text{ contains } 01 \text{ as a substring} \}$
- $\left\{ w \mid \begin{array}{l} w \text{ has an even number of } 0\text{'s} \\ \wedge \text{ } w \text{ has an odd number of } 1\text{'s} \end{array} \right\}$

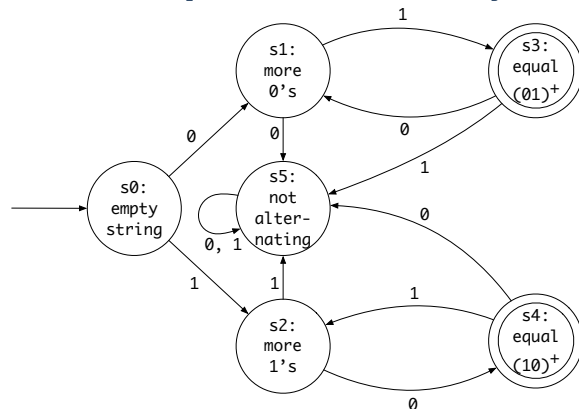
23 of 68

DFA: Deterministic Finite Automata (1.3)



The **transition diagram** below defines a DFA which **accepts/recognizes** exactly the language

$\{ w \mid \begin{array}{l} w \neq \epsilon \\ \wedge \text{ } w \text{ has equal \# of alternating } 0\text{'s and } 1\text{'s} \end{array} \}$



22 of 68

DFA: Deterministic Finite Automata (2.1)



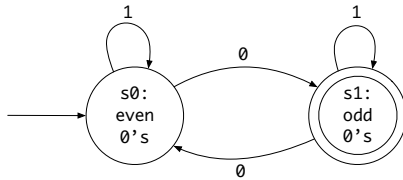
A **deterministic finite automata (DFA)** is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

- Q is a finite set of **states**.
- Σ is a finite set of **input symbols** (i.e., the **alphabet**).
- $\delta: (Q \times \Sigma) \rightarrow Q$ is a **transition function**
 δ takes as arguments a state and an input symbol and returns a state.
- $q_0 \in Q$ is the **start state**.
- $F \subseteq Q$ is a set of **final** or **accepting states**.

24 of 68

DFA: Deterministic Finite Automata (2.2)



We formalize the above DFA as $M = (Q, \Sigma, \delta, q_0, F)$, where

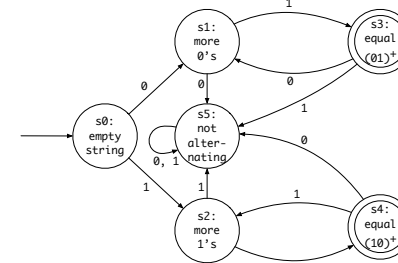
- $Q = \{s_0, s_1\}$
- $\Sigma = \{0, 1\}$
- $\delta = \{((s_0, 0), s_1), ((s_0, 1), s_0), ((s_1, 0), s_0), ((s_1, 1), s_1)\}$

state \ input	0	1
s ₀	s ₁	s ₀
s ₁	s ₀	s ₁

- $q_0 = s_0$
- $F = \{s_1\}$

25 of 68

DFA: Deterministic Finite Automata (2.3.2)

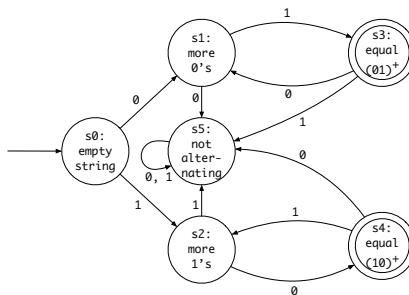


- $\delta =$

state \ input	0	1
s ₀	s ₁	s ₂
s ₁	s ₅	s ₃
s ₂	s ₄	s ₅
s ₃	s ₁	s ₅
s ₄	s ₅	s ₂
s ₅	s ₅	s ₅

27 of 68

DFA: Deterministic Finite Automata (2.3.1)



We formalize the above DFA as $M = (Q, \Sigma, \delta, q_0, F)$, where

- $Q = \{s_0, s_1, s_2, s_3, s_4, s_5\}$
- $\Sigma = \{0, 1\}$
- $q_0 = s_0$
- $F = \{s_3, s_4\}$

26 of 68

DFA: Deterministic Finite Automata (2.4)



- Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$:
 - We write $L(M)$ to denote the **language of M** : the set of strings that M **accepts**.
 - A string is **accepted** if it results in a sequence of transitions: beginning from the **start** state and ending in a **final** state.

$$L(M) = \left\{ a_1 a_2 \dots a_n \mid \begin{array}{l} 1 \leq i \leq n \wedge a_i \in \Sigma \wedge \delta(q_{i-1}, a_i) = q_i \wedge q_n \in F \end{array} \right\}$$

- M **rejects** any string $w \notin L(M)$.
- We may also consider $L(M)$ as **concatenations of labels** from the set of all valid **paths** of M 's transition diagram; each such path starts with q_0 and ends in a state in F .

28 of 68

DFA: Deterministic Finite Automata (2.5)



- Given a **DFA** $M = (Q, \Sigma, \delta, q_0, F)$, we may simplify the definition of $L(M)$ by extending δ (which takes an input symbol) to $\hat{\delta}$ (which takes an input string).

$$\hat{\delta} : (Q \times \Sigma^*) \rightarrow Q$$

We may define $\hat{\delta}$ recursively, using δ !

$$\begin{aligned}\hat{\delta}(q, \epsilon) &= q \\ \hat{\delta}(q, xa) &= \delta(\hat{\delta}(q, x), a)\end{aligned}$$

where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$

- A neater definition of $L(M)$: the set of strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ is an **accepting state**.

$$L(M) = \{w \mid w \in \Sigma^* \wedge \hat{\delta}(q_0, w) \in F\}$$

- A language L is said to be a **regular language**, if there is some **DFA** M such that $L = L(M)$.

29 of 68

Review Exercises: Formalizing DFAs



Formalize DFAs (as 5-tuples) for the other example string patterns mentioned:

- $\{w \mid w \text{ has an even number of 0's}\}$
- $\{w \mid w \text{ contains 01 as a substring}\}$
- $\left\{w \mid \begin{array}{l} w \text{ has an even number of 0's} \\ \wedge w \text{ has an odd number of 1's} \end{array}\right\}$

30 of 68

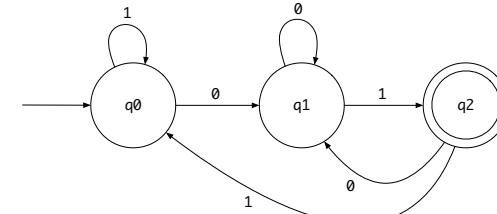
NFA: Nondeterministic Finite Automata (1.1)



Problem: Design a DFA that accepts the following language:

$$L = \{x01 \mid x \in \{0, 1\}^*\}$$

That is, L is the set of strings of 0s and 1s ending with 01.



Given an input string w , we may simplify the above DFA by:

- nondeterministically** treating state q_0 as both:
 - a state **ready** to read the last two input symbols from w
 - a state **not yet ready** to read the last two input symbols from w
- substantially reducing the outgoing transitions from q_1 and q_2

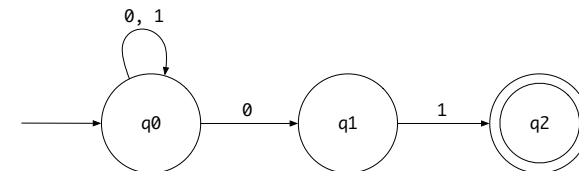
31 of 68

Compare the above DFA with the DFA in slide 39.

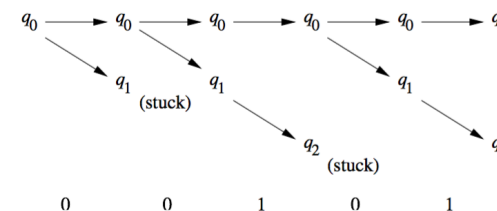
NFA: Nondeterministic Finite Automata (1.2)



- A **non-deterministic finite automata (NFA)** that accepts the same language:



- How an NFA determines if an input **00101** should be processed:



32 of 68

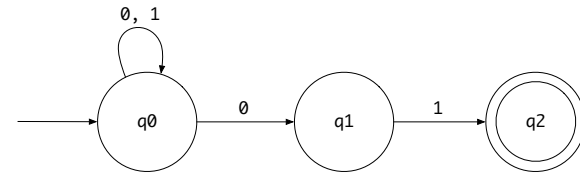
NFA: Nondeterministic Finite Automata (2)



- A **nondeterministic finite automata (NFA)**, like a **DFA**, is a **FSM** that **accepts** (or **recognizes**) a pattern of behaviour.
- An **NFA** being **nondeterministic** means that from a given state, the **same input label** might correspond to **multiple transitions** that lead to **distinct states**.
 - Each such transition offers an **alternative path**.
 - Each alternative path is explored **in parallel**.
 - If **there exists** an alternative path that **succeeds** in processing the input string, then we say the **NFA accepts** that input string.
 - If **all** alternative paths get stuck at some point and **fail** to process the input string, then we say the **NFA rejects** that input string.
- NFAs** are often more succinct (i.e., fewer states) and easier to design than **DFAs**.
- However, **NFAs** are just as **expressive** as are **DFAs**.
 - We can **always** convert an **NFA** to a **DFA**.

33 of 68

NFA: Nondeterministic Finite Automata (3.2)



Given an input string 00101:

- Read 0:** $\delta(q_0, 0) = \{q_0, q_1\}$
- Read 0:** $\delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
- Read 1:** $\delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
- Read 0:** $\delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
- Read 1:** $\delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0, q_1\} \cup \{q_2\} = \{q_0, q_1, q_2\}$
 $\therefore \{q_0, q_1, q_2\} \cap \{q_2\} \neq \emptyset \therefore 00101$ is **accepted**

35 of 68

NFA: Nondeterministic Finite Automata (3.1)



- A **nondeterministic finite automata (NFA)** is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$
 - Q is a finite set of **states**.
 - Σ is a finite set of **input symbols** (i.e., the **alphabet**).
 - $\delta : (Q \times \Sigma) \rightarrow \mathbb{P}(Q)$ is a **transition function**
 - Given a state and an input symbol, δ returns a set of states.
 - Equivalently, we can write: $\delta : (Q \times \Sigma) \rightarrow Q$ [a **partial function**]
 - $q_0 \in Q$ is the **start state**.
 - $F \subseteq Q$ is a set of **final** or **accepting states**.
- What is the difference between a **DFA** and an **NFA**?
 - δ of a **DFA** returns a **single state**.
 - δ of an **NFA** returns a (possibly empty) **set** of states.

34 of 68

NFA: Nondeterministic Finite Automata (3.3)



- Given a **NFA** $M = (Q, \Sigma, \delta, q_0, F)$, we may simplify the definition of $L(M)$ by extending δ (which takes an input symbol) to $\hat{\delta}$ (which takes an input string).

$$\hat{\delta} : (Q \times \Sigma^*) \rightarrow \mathbb{P}(Q)$$

We may define $\hat{\delta}$ recursively, using δ !

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= \{q\} \\ \hat{\delta}(q, xa) &= \bigcup \{\delta(q', a) \mid q' \in \hat{\delta}(q, x)\} \end{aligned}$$

where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$

- A neater definition of $L(M)$: the set of strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ contains **at least one accepting state**.

$$L(M) = \{w \mid w \in \Sigma^* \wedge \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

36 of 68

DFA $\equiv$ NFA (1)



- For many languages, constructing an accepting **NFA** is easier than a **DFA**.
- From each state of an **NFA**:
 - Outgoing transitions need **not** cover the entire Σ .
 - From a given state, the same symbol may **non-deterministically** lead to multiple states.
- In practice:
 - An **NFA** has just as many states as its equivalent DFA does.
 - An **NFA** often has fewer transitions than its equivalent **DFA** does.
- In the **worst** case:
 - While an **NFA** has n states, its equivalent **DFA** has 2^n states.
- Nonetheless, an **NFA** is still just as **expressive** as a **DFA**.
 - A **language** accepted by some **NFA** is accepted by some **DFA**:

$$\forall N \bullet N \in \text{NFA} \Rightarrow (\exists D \bullet D \in \text{DFA} \wedge L(D) = L(N))$$
 - And vice versa, trivially?

$$\forall D \bullet D \in \text{DFA} \Rightarrow (\exists N \bullet N \in \text{NFA} \wedge L(D) = L(N))$$

37 of 68

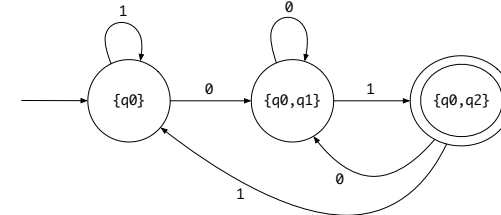
DFA $\equiv$ NFA (2.2): Lazy Evaluation (2)



Applying **subset construction** (with **lazy evaluation**), we arrive in a **DFA** transition table:

state \ input	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

We then draw the **DFA** accordingly:



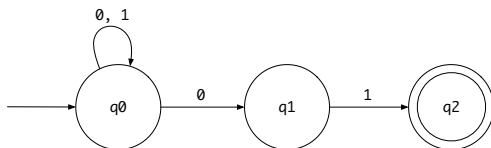
Compare the above DFA with the DFA in slide 31

39 of 68

DFA $\equiv$ NFA (2.2): Lazy Evaluation (1)



Given an **NFA**:



Subset construction (with **lazy evaluation**) produces a **DFA** with δ as:

state \ input	0	1
$\{q_0\}$	$\delta(q_0, 0) = \{q_0, q_1\}$	$\delta(q_0, 1) = \{q_0\}$
$\{q_0, q_1\}$	$\delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$	$\delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
$\{q_0, q_2\}$	$\delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$	$\delta(q_0, 1) \cup \delta(q_2, 1) = \{q_0\} \cup \emptyset = \{q_0\}$

38 of 68

DFA $\equiv$ NFA (2.2): Lazy Evaluation (3)



- Given an **NFA** $N = (Q_N, \Sigma_N, \delta_N, q_0, F_N)$:

ALGORITHM: *ReachableSubsetStates*

INPUT: $q_0 : Q_N$; **OUTPUT:** **Reachable** $\subseteq \mathbb{P}(Q_N)$

PROCEDURE:

```

Reachable := { {q0} }
ToDiscover := { {q0} }
while (ToDiscover  $\neq \emptyset$ ) {
    choose S :  $\mathbb{P}(Q_N)$  such that  $S \in \text{ToDiscover}$ 
    remove S from ToDiscover
    NotYetDiscovered :=
        ( { { $\delta_N(s, 0)$  |  $s \in S$ } }  $\cup$  { { $\delta_N(s, 1)$  |  $s \in S$ } } )  $\setminus$  Reachable
    Reachable := Reachable  $\cup$  NotYetDiscovered
    ToDiscover := ToDiscover  $\cup$  NotYetDiscovered
}
return Reachable
    
```

- RT of *ReachableSubsetStates*? [$O(2^{|Q_N|})$]
- Often only a small portion of the $|\mathbb{P}(Q_N)|$ **subset states** is **reachable** from $\{q_0\} \Rightarrow$ **Lazy Evaluation** efficient in practice!

40 of 68

ε-NFA: Examples (1)

Draw the NFA for the following two languages:

1.

$$\left\{ xy \mid \begin{array}{l} x \in \{0,1\}^* \\ \wedge y \in \{0,1\}^* \\ \wedge x \text{ has alternating } 0\text{'s and } 1\text{'s} \\ \wedge y \text{ has an odd \# } 0\text{'s and an odd \# } 1\text{'s} \end{array} \right\}$$

2.

$$\left\{ w : \{0,1\}^* \mid \begin{array}{l} w \text{ has alternating } 0\text{'s and } 1\text{'s} \\ \vee w \text{ has an odd \# } 0\text{'s and an odd \# } 1\text{'s} \end{array} \right\}$$

3.

$$\left\{ sx.y \mid \begin{array}{l} s \in \{+, -, \epsilon\} \\ \wedge x \in \Sigma_{dec}^* \\ \wedge y \in \Sigma_{dec}^* \\ \wedge \neg(x = \epsilon \wedge y = \epsilon) \end{array} \right\}$$

41 of 68

ε-NFA: Formalization (1)

An **ε-NFA** is a 5-tuple

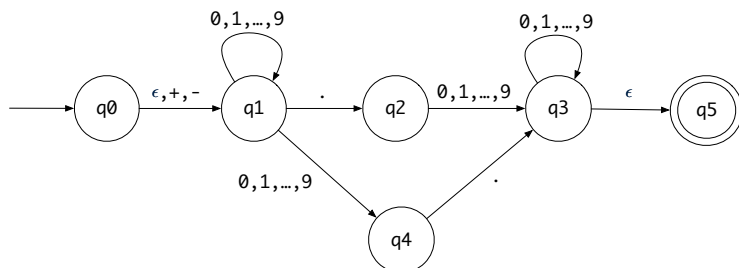
$$M = (Q, \Sigma, \delta, q_0, F)$$

- Q is a finite set of **states**.
- Σ is a finite set of **input symbols** (i.e., the **alphabet**).
- $\delta : (Q \times (\Sigma \cup \{\epsilon\})) \rightarrow \mathbb{P}(Q)$ is a **transition function**
 δ takes as arguments a state and an input symbol, or an **empty string** ϵ , and returns a set of states.
- $q_0 \in Q$ is the **start state**.
- $F \subseteq Q$ is a set of **final** or **accepting states**.

43 of 68

ε-NFA: Examples (2)

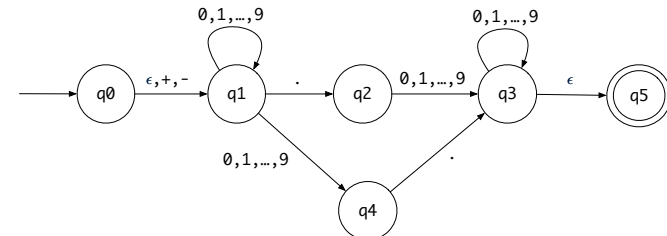
$$\left\{ sx.y \mid \begin{array}{l} s \in \{+, -, \epsilon\} \\ \wedge x \in \Sigma_{dec}^* \\ \wedge y \in \Sigma_{dec}^* \\ \wedge \neg(x = \epsilon \wedge y = \epsilon) \end{array} \right\}$$



From q_0 to q_1 , reading a sign is **optional**: a **plus** or a **minus**, or **nothing at all** (i.e., ϵ).

42 of 68

ε-NFA: Formalization (2)



Draw a transition table for the above NFA's δ function:

	ε	+, -	.	0 .. 9
q_0	{ q_1 }	{ q_1 }	∅	∅
q_1	∅	∅	{ q_2 }	{ q_1, q_4 }
q_2	∅	∅	∅	{ q_3 }
q_3	{ q_5 }	∅	∅	{ q_3 }
q_4	∅	∅	{ q_3 }	∅
q_5	∅	∅	∅	∅

44 of 68

ε-NFA: Epsilon-Closures (1)



- Given ε-NFA N

$$N = (Q, \Sigma, \delta, q_0, F)$$

we define the **epsilon closure** (or **ε-closure**) as a function

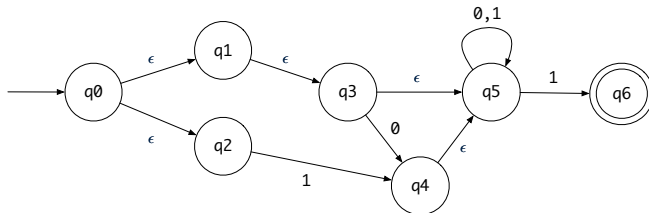
$$\text{ECLOSE} : Q \rightarrow \mathbb{P}(Q)$$

- For any state $q \in Q$

$$\text{ECLOSE}(q) = \{q\} \cup \bigcup_{p \in \delta(q, \epsilon)} \text{ECLOSE}(p)$$

45 of 68

ε-NFA: Epsilon-Closures (2)



$$\begin{aligned} \text{ECLOSE}(q_0) &= \{ \delta(q_0, \epsilon) = \{q_1, q_2\} \} \\ &= \{q_0\} \cup \text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_2) \\ &= \{ \text{ECLOSE}(q_1), \delta(q_1, \epsilon) = \{q_3\}, \text{ECLOSE}(q_2), \delta(q_2, \epsilon) = \emptyset \} \\ &= \{q_0\} \cup (\{q_1\} \cup \text{ECLOSE}(q_3)) \cup (\{q_2\} \cup \emptyset) \\ &= \{ \text{ECLOSE}(q_3), \delta(q_3, \epsilon) = \{q_5\} \} \\ &= \{q_0\} \cup (\{q_1\} \cup (\{q_3\} \cup \text{ECLOSE}(q_5))) \cup (\{q_2\} \cup \emptyset) \\ &= \{ \text{ECLOSE}(q_5), \delta(q_5, \epsilon) = \emptyset \} \\ &= \{q_0\} \cup (\{q_1\} \cup (\{q_3\} \cup (\{q_5\} \cup \emptyset))) \cup (\{q_2\} \cup \emptyset) \end{aligned}$$

46 of 68

ε-NFA: Formalization (3)



- Given a ε-NFA $M = (Q, \Sigma, \delta, q_0, F)$, we may simplify the definition of $L(M)$ by extending δ (which takes an input symbol) to $\hat{\delta}$ (which takes an input string).

$$\hat{\delta} : (Q \times \Sigma^*) \rightarrow \mathbb{P}(Q)$$

We may define $\hat{\delta}$ recursively, using δ !

$$\hat{\delta}(q, \epsilon) = \text{ECLOSE}(q)$$

$$\hat{\delta}(q, xa) = \bigcup \{ \text{ECLOSE}(q'') \mid q'' \in \delta(q', a) \wedge q' \in \hat{\delta}(q, x) \}$$

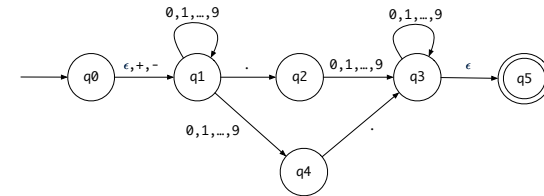
where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$

- Then we define $L(M)$ as the set of strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ contains **at least one accepting state**.

$$L(M) = \{w \mid w \in \Sigma^* \wedge \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

47 of 68

ε-NFA: Formalization (4)



Given an input string 5.6:

$$\hat{\delta}(q_0, \epsilon) = \text{ECLOSE}(q_0) = \{q_0, q_1\}$$

- Read 5:** $\delta(q_0, 5) \cup \delta(q_1, 5) = \emptyset \cup \{q_1, q_4\} = \{q_1, q_4\}$

$$\hat{\delta}(q_0, 5) = \text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$$

- Read .:** $\delta(q_1, .) \cup \delta(q_4, .) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$

$$\hat{\delta}(q_0, 5.) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$$

- Read 6:** $\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \emptyset = \{q_3\}$

$$\hat{\delta}(q_0, 5.6) = \text{ECLOSE}(q_3) = \{q_3, q_5\}$$

[5.6 is **accepted**]

48 of 68

DFA $\equiv$ ϵ -NFA: Extended Subset Const. (1)



Subset construction (with *lazy evaluation* and *epsilon closures*) produces a **DFA** transition table.

	$d \in 0 \dots 9$	$s \in \{+, -\}$	.
$\{q_0, q_1\}$	$\{q_1, q_4\}$	$\{q_1\}$	$\{q_2\}$
$\{q_1, q_4\}$	$\{q_1, q_4\}$	$\emptyset$	$\{q_2, q_3, q_5\}$
$\{q_1\}$	$\{q_1, q_4\}$	$\emptyset$	$\{q_2\}$
$\{q_2\}$	$\{q_3, q_5\}$	$\emptyset$	$\emptyset$
$\{q_2, q_3, q_5\}$	$\{q_3, q_5\}$	$\emptyset$	$\emptyset$
$\{q_3, q_5\}$	$\{q_3, q_5\}$	$\emptyset$	$\emptyset$

For example, $\delta(\{q_0, q_1\}, d)$ is calculated as follows: $[d \in 0 \dots 9]$

$$\begin{aligned}
 & \cup \{ \text{ECLOSE}(q) \mid q \in \delta(q_0, d) \cup \delta(q_1, d) \} \\
 = & \cup \{ \text{ECLOSE}(q) \mid q \in \emptyset \cup \{q_1, q_4\} \} \\
 = & \cup \{ \text{ECLOSE}(q) \mid q \in \{q_1, q_4\} \} \\
 = & \text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) \\
 = & \{q_1\} \cup \{q_4\} \\
 = & \{q_1, q_4\}
 \end{aligned}$$

49 of 68

Regular Expression to ϵ -NFA



- Just as we construct each complex *regular expression* recursively, we define its equivalent ϵ -NFA *recursively*.
- Given a regular expression R , we construct an ϵ -NFA E , such that $L(R) = L(E)$, with
 - Exactly **one** accept state.
 - No incoming arc to the start state.
 - No outgoing arc from the accept state.

51 of 68

DFA $\equiv$ ϵ -NFA: Extended Subset Const. (2)



Given an ϵ -NFA $N = (Q_N, \Sigma_N, \delta_N, q_0, F_N)$, by applying the **extended subset construction** to it, the resulting **DFA** $D = (Q_D, \Sigma_D, \delta_D, q_{D_{start}}, F_D)$ is such that:

$$\begin{aligned}
 \Sigma_D &= \Sigma_N \\
 q_{D_{start}} &= \text{ECLOSE}(q_0) \\
 F_D &= \{ S \mid S \subseteq Q_N \wedge S \cap F_N \neq \emptyset \} \\
 Q_D &= \{ S \mid S \subseteq Q_N \wedge (\exists w \bullet w \in \Sigma^* \Rightarrow S = \hat{\delta}_N(q_0, w)) \} \\
 \delta_D(S, a) &= \cup \{ \text{ECLOSE}(s') \mid s \in S \wedge s' \in \delta_N(s, a) \}
 \end{aligned}$$

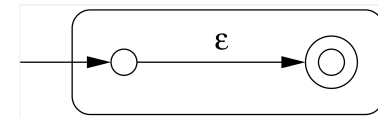
50 of 68

Regular Expression to ϵ -NFA

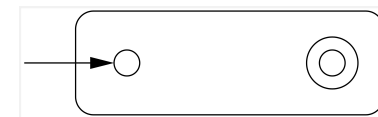


Base Cases:

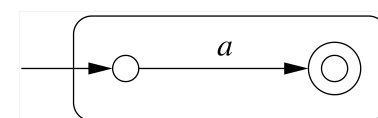
- ϵ



- $\emptyset$



- a



$[a \in \Sigma]$

52 of 68

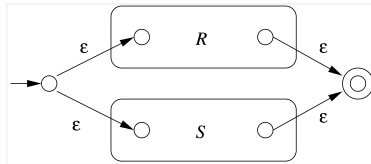
Regular Expression to ϵ -NFA



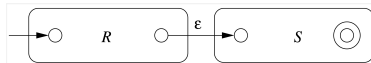
Recursive Cases:

[R and S are RE's]

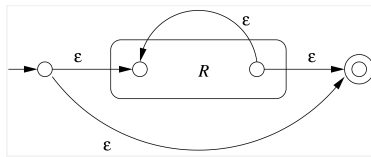
- $R + S$



- RS



- R^*

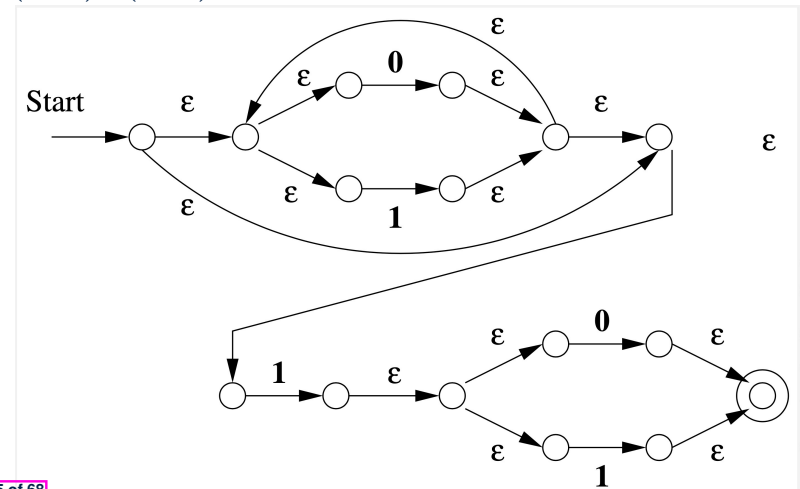


53 of 68

Regular Expression to ϵ -NFA: Examples (1.2)



- $(0 + 1)^* 1(0 + 1)$

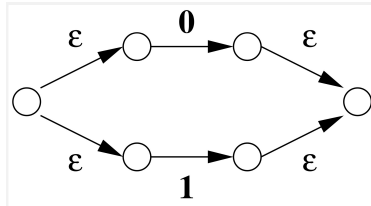


55 of 68

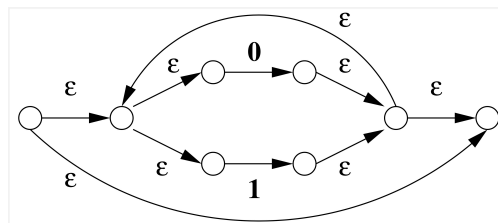
Regular Expression to ϵ -NFA: Examples (1.1)



- $0 + 1$



- $(0 + 1)^*$



54 of 68

Minimizing DFA: Motivation



- Recall: Regular Expression $\rightarrow$ ϵ -NFA $\rightarrow$ DFA
- DFA produced by the **extended subset construction** (with **lazy evaluation**) may **not** be **minimum** on its size of state.
- When the required size of memory is sensitive (e.g., processor's cache memory), the fewer number of DFA states, the better.

56 of 68

Minimizing DFA: Algorithm



```

ALGORITHM: MinimizeDFAStates
INPUT: DFA  $M = (Q, \Sigma, \delta, q_0, F)$ 
OUTPUT:  $M'$  s.t. minimum  $|Q|$  and equivalent behaviour as  $M$ 
PROCEDURE:
   $P := \emptyset$  /* refined partition so far */
   $T := \{ F, Q - F \}$  /* last refined partition */
  while ( $P \neq T$ ):
     $P := T$ 
     $T := \emptyset$ 
    for ( $p \in P$ ):
      find the maximal  $S \subset p$  s.t. splittable( $p, S$ )
      if  $S \neq \emptyset$  then
         $T := T \cup \{S, p - S\}$ 
      else
         $T := T \cup \{p\}$ 
    end
  end
  
```

splittable(p, S) holds iff there is $c \in \Sigma$ s.t.

1. $S \subset p$ (or equivalently: $p - S \neq \emptyset$)
2. Transitions via c lead all $s \in S$ to states in **same partition** p_1 ($p_1 \neq p$).

57 of 68

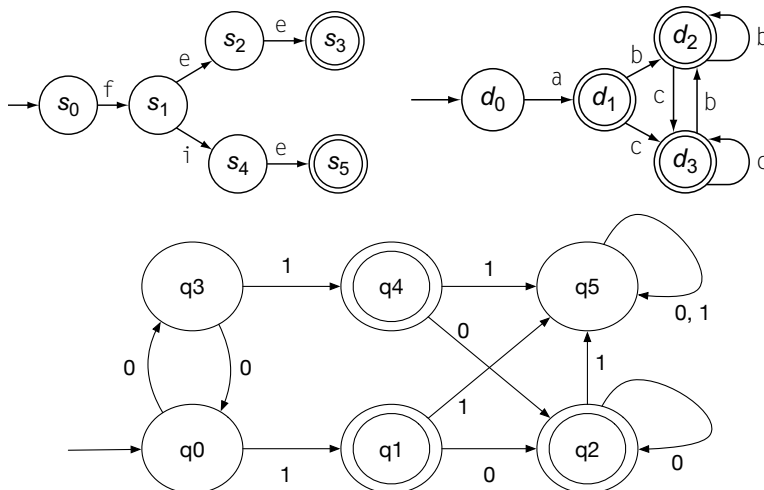
Exercise: Regular Expression to Minimized DFA



Given regular expression $r[0..9]^+$ which specifies the pattern of a register name, derive the equivalent DFA with the minimum number of states. Show all steps.

59 of 68

Minimizing DFA: Examples



58 of 68

Exercises: Minimize the DFA from here: Q1 & Q2, p59, EAC2.

Implementing DFA as Scanner



- The source language has a list of **syntactic categories**:
e.g., keyword while [while]
e.g., identifiers [[a-zA-Z][a-zA-Z0-9_]*]
e.g., white spaces [[\t\r]+]
- A compiler's **scanner** must recognize **words** from all syntactic categories of the source language.
 - Each syntactic category is specified via a **regular expression**.

$$\underbrace{r_1}_{\text{syn. cat. 1}} + \underbrace{r_2}_{\text{syn. cat. 2}} + \dots + \underbrace{r_n}_{\text{syn. cat. n}}$$

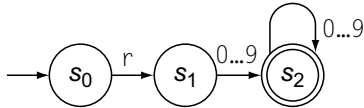
- Overall, a scanner should be implemented based on the **minimized DFA** accommodating all syntactic categories.
- Principles of a scanner:
 - Returns one **word** at a time
 - Each returned word is the **longest possible** that matches a **pattern**
 - A **priority** may be specified among patterns (e.g., **new** is a keyword, not identifier)

60 of 68

Implementing DFA: Table-Driven Scanner (1)



- Consider the **syntactic category** of register names.
- Specified as a **regular expression**: $r[0..9]^+$
- Afer conversion to ϵ -NFA, then to DFA, then to **minimized DFA**:



- The following tables encode knowledge about the above DFA:

Classifier (CharCat)					Transition (δ)				Token Type (Type)			
r	0,1,2,...,9	EOF	Other		Register	Digit	Other		s ₀	s ₁	s ₂	s _e
Register	Digit	Other	Other		s ₀	s ₁	s _e	s _e	invalid	invalid	register	invalid
					s ₁	s _e	s ₂	s _e				
					s ₂	s _e	s ₂	s _e				
					s _e	s _e	s _e	s _e				

61 of 68

Index (1)



Scanner in Context

Scanner: Formulation & Implementation

Alphabets

Strings (1)

Strings (2)

Review Exercises: Strings

Languages

Review Exercises: Languages

Problems

Regular Expressions (RE): Introduction

RE: Language Operations (1)

63 of 68

Implementing DFA: Table-Driven Scanner (2)



The scanner then is implemented via a 4-stage skeleton:

```

NextWord()
-- Stage 1: Initialization
state := s0 ; word := ε
initialize an empty stack s ; s.push(bad)
-- Stage 2: Scanning Loop
while (state ≠ se)
  NextChar(char) ; word := word + char
  if state ∈ F then reset stack s end
  s.push(state)
  cat := CharCat[char]
  state := δ[state, cat]
-- Stage 3: Rollback Loop
while (state ∉ F ∧ state ≠ bad)
  state := s.pop()
  truncate word
-- Stage 4: Interpret and Report
if state ∈ F then return Type[state]
else return invalid
end
  
```

62 of 68

Index (2)



RE: Language Operations (2)

RE: Construction (1)

RE: Construction (2)

RE: Construction (3)

RE: Construction (4)

RE: Review Exercises

RE: Operator Precedence

DFA: Deterministic Finite Automata (1.1)

DFA: Deterministic Finite Automata (1.2)

DFA: Deterministic Finite Automata (1.3)

Review Exercises: Drawing DFAs

64 of 68

Index (3)

DFA: Deterministic Finite Automata (2.1)
DFA: Deterministic Finite Automata (2.2)
DFA: Deterministic Finite Automata (2.3.1)
DFA: Deterministic Finite Automata (2.3.2)
DFA: Deterministic Finite Automata (2.4)
DFA: Deterministic Finite Automata (2.5)
Review Exercises: Formalizing DFAs
NFA: Nondeterministic Finite Automata (1.1)
NFA: Nondeterministic Finite Automata (1.2)
NFA: Nondeterministic Finite Automata (2)
NFA: Nondeterministic Finite Automata (3.1)

65 of 68

Index (4)

NFA: Nondeterministic Finite Automata (3.2)
NFA: Nondeterministic Finite Automata (3.3)
DFA $\equiv$ NFA (1)
DFA $\equiv$ NFA (2.2): Lazy Evaluation (1)
DFA $\equiv$ NFA (2.2): Lazy Evaluation (2)
DFA $\equiv$ NFA (2.2): Lazy Evaluation (3)
 ϵ -NFA: Examples (1)
 ϵ -NFA: Examples (2)
 ϵ -NFA: Formalization (1)
 ϵ -NFA: Formalization (2)
 ϵ -NFA: Epsilon-Closures (1)

66 of 68

Index (5)

ϵ -NFA: Epsilon-Closures (2)
 ϵ -NFA: Formalization (3)
 ϵ -NFA: Formalization (4)
DFA $\equiv$ ϵ -NFA: Extended Subset Const. (1)
DFA $\equiv$ ϵ -NFA: Extended Subset Const. (2)
Regular Expression to ϵ -NFA
Regular Expression to ϵ -NFA
Regular Expression to ϵ -NFA
Regular Expression to ϵ -NFA: Examples (1.1)
Regular Expression to ϵ -NFA: Examples (1.2)
Minimizing DFA: Motivation

67 of 68

Index (6)

Minimizing DFA: Algorithm
Minimizing DFA: Examples
Exercise:
Regular Expression to Minimized DFA
Implementing DFA as Scanner
Implementing DFA: Table-Driven Scanner (1)
Implementing DFA: Table-Driven Scanner (2)

68 of 68