# Regular Expressions and Finite State Automata

# Introduction

- Regular expressions are equivalent to Finite State Automata in recognizing regular languages, the first step in the Chomsky hierarchy of formal languages
- The term regular expressions is also used to mean the extended set of string matching expressions used in many modern languages
  - Some people use the term regexp to distinguish this use
- Some parts of regexps are just syntactic extensions of regular expressions and can be implemented as a regular expression – other parts are significant extensions of the power of the language and are not equivalent to finite automata

# Concepts and Notations

- **Set:** An unordered collection of unique elements

  $S_1 = \{ a, b, c \}$ $S_2 = \{ 0, 1, \ldots, 19 \}$ *empty set:* $\emptyset$

  *membership:* $x \in S$ *union:* $S_1 \cup S_2 = \{ a, b, c, 0, 1, \ldots, 19 \}$

  *universe of discourse:* $U$ *subset:* $S_1 \subset U$

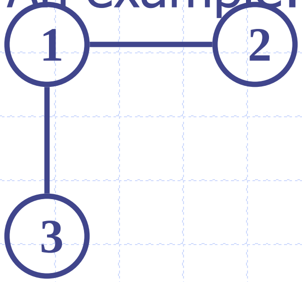
  *complement:* if $U = \{ a, b, \ldots, z \}$, then $S_1' = \{ d, e, \ldots, z \} = U - S_1$

- **Alphabet:** A finite set of symbols
  - Examples:
    - Character sets: ASCII, ISO-8859-1, Unicode
    - $\Sigma_1 = \{ a, b \}$ $\Sigma_2 = \{ \text{Spring, Summer, Autumn, Winter} \}$

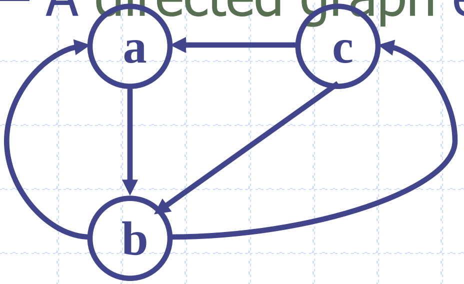
- **String:** A sequence of zero or more symbols from an alphabet
  - The empty string: $\varepsilon$

# Concepts and Notations

- **Language**: A set of strings over an alphabet
  - Also known as a formal language; may not bear any resemblance to a natural language, but could model a subset of one.
  - The language comprising **all** strings over an alphabet $\Sigma$ is written as: $\Sigma^*$
- **Graph**: A set of nodes (or vertices), some or all of which may be connected by edges.
  - An example:




    – A directed graph example:



# Regular Expressions

- A regular expression defines a regular language over an alphabet $\Sigma$:
  - $\varnothing$ is a regular language: `//`
  - Any symbol from $\Sigma$ is a regular language:

    $\Sigma = \{ a, b, c\}$ `/a/` `/b/` `/c/`
  - Two concatenated regular languages is a regular language:

    $\Sigma = \{ a, b, c\}$ `/ab/` `/bc/` `/ca/`

# Regular Expressions

- Regular language (continued):
  - The union (or disjunction) of two regular languages is a regular language:

    $\Sigma = \{ a, b, c\}$ `/ab|bc/` `/ca|bb/`
  - The Kleene closure (denoted by the Kleene star: *) of a regular language is a regular language:

    $\Sigma = \{ a, b, c\}$ `/a*/` `/(ab|ca)*/`
  - Parentheses group a sub-language to override operator precedence (and, we'll see later, for "memory").

# Finite Automata

- Finite State Automaton

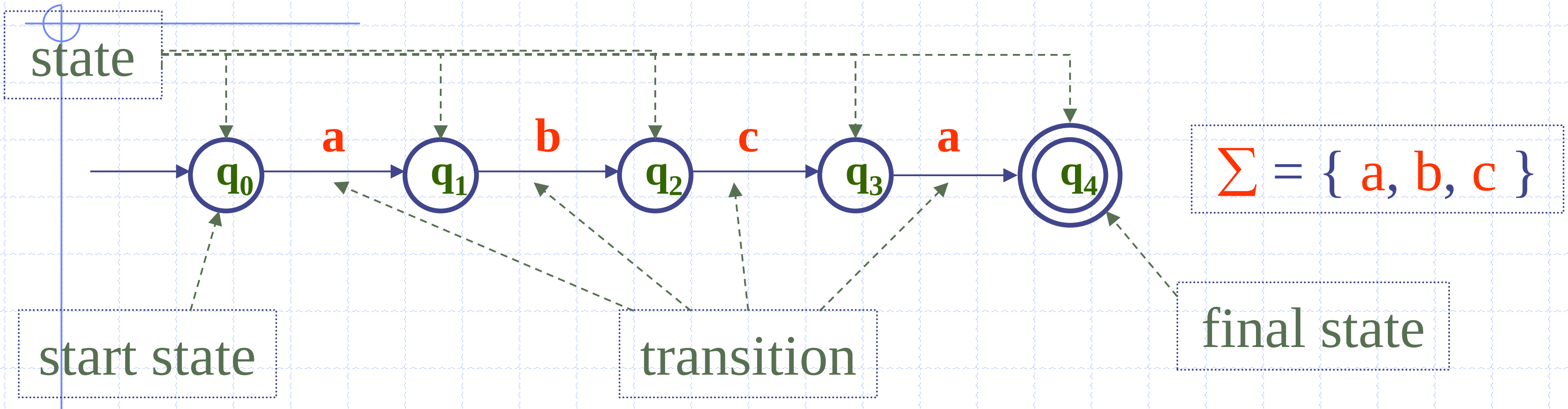
  a.k.a. Finite Automaton, Finite State Machine, FSA or FSM

  - An abstract machine which can be used to implement regular expressions (etc.).
  - Has a finite number of states, and a finite amount of memory (i.e., the current state).
  - Can be represented by directed graphs or transition tables

# Finite-state Automata (1/23)

- Representation
  - An FSA may be represented as a directed graph; each node (or vertex) represents a state, and the edges (or arcs) connecting the nodes represent transitions.
  - Each state is labelled.
  - Each transition is labelled with a symbol from the alphabet over which the regular language represented by the FSA is defined, or with $\varepsilon$, the empty string.
  - Among the FSA's states, there is a start state and at least one final state (or accepting state).

# Finite-state Automata (2/23)




- Representation (continued)
  - An FSA may also be represented with a state-transition table. The table for the above FSA:

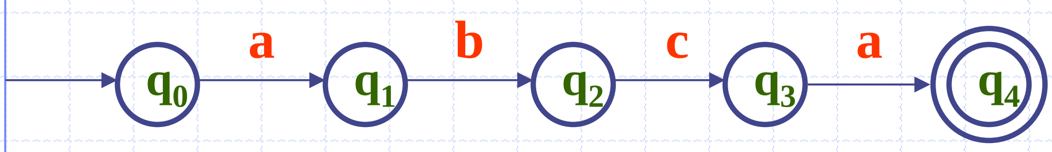
| | Input | | |
|---|---|---|---|
| State | a | b | c |
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

# Finite-state Automata (3/23)

- Given an input string, an FSA will either accept or reject the input.
  - If the FSA is in a final (or accepting) state after all input symbols have been consumed, then the string is accepted (or recognized).
  - Otherwise (including the case in which an input symbol cannot be consumed), the string is rejected.

# Finite-state Automata (3/23)

$\Sigma = \{ a, b, c \}$




| State | Input a | Input b | Input c |
|---|---|---|---|
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

$IS_1$:

| | | | |
|---|---|---|---|
| a | b | c | a |

$IS_2$:

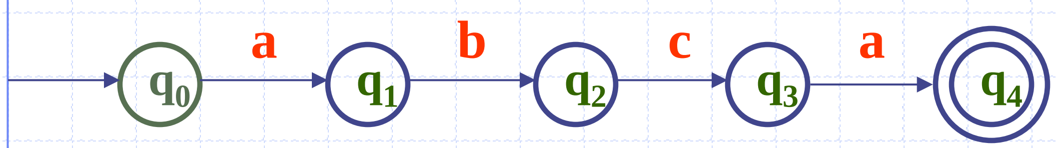
| | | | |
|---|---|---|---|
| c | c | b | a |

$IS_3$:

| | | | | |
|---|---|---|---|---|
| a | b | c | a | c |

# Finite-state Automata (4/23)

$\Sigma = \{ a, b, c \}$




| | Input | | |
|---|---|---|---|
| **State** | **a** | **b** | **c** |
| **0** | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

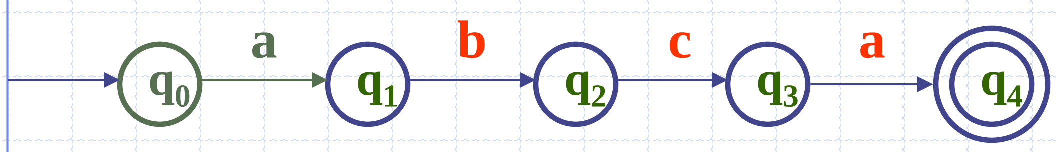
| | | | | |
|---|---|---|---|---|
| **$IS_1$:** | a | b | c | a |

| | | | | |
|---|---|---|---|---|
| $IS_2$: | c | c | b | a |

| | | | | | |
|---|---|---|---|---|---|
| $IS_3$: | a | b | c | a | c |

# Finite-state Automata (5/23)

$\Sigma = \{ a, b, c \}$

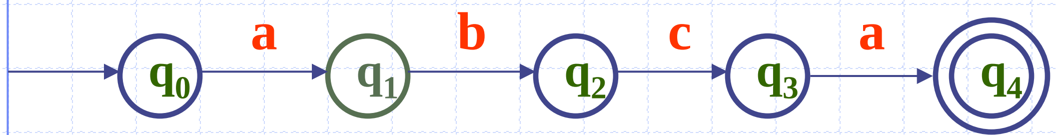



| State | Input a | b | c |
|---|---|---|---|
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

| | | | | | |
|---|---|---|---|---|---|
| $IS_1$: | a | b | c | a | |
| $IS_2$: | c | c | b | a | |
| $IS_3$: | a | b | c | a | c |

# Finite-state Automata (6/23)

$\Sigma = \{ a, b, c \}$




| State | a | b | c |
|---|---|---|---|
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

(Input: a, b, c)

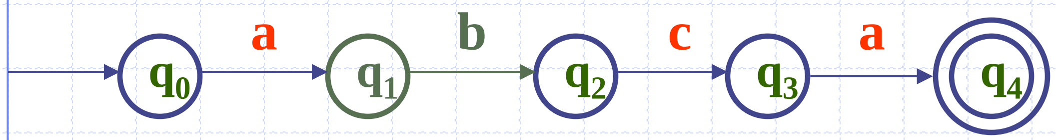
$IS_1$: | a | b | c | a |

$IS_2$: | c | c | b | a |

$IS_3$: | a | b | c | a | c |

# Finite-state Automata (7/23)

$\Sigma = \{ a, b, c \}$




| State | Input a | Input b | Input c |
|---|---|---|---|
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

$IS_1$:

| | | | |
|---|---|---|---|
| a | b | c | a |

$IS_2$:

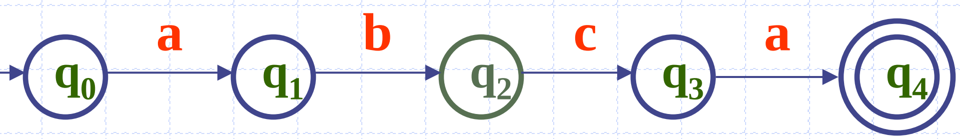
| | | | |
|---|---|---|---|
| c | c | b | a |

$IS_3$:

| | | | | |
|---|---|---|---|---|
| a | b | c | a | c |

# Finite-state Automata (8/23)

$\Sigma = \{ a, b, c \}$




| State | Input a | b | c |
|---|---|---|---|
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

$IS_1$:

| | | | |
|---|---|---|---|
| a | b | c | a |

$IS_2$:

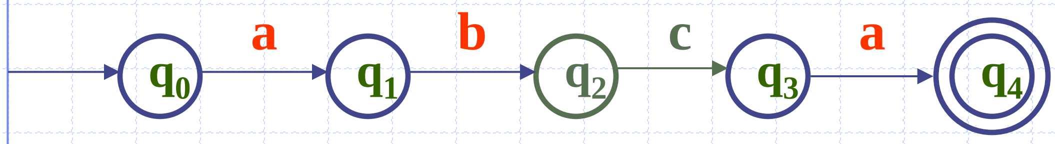
| | | | |
|---|---|---|---|
| c | c | b | a |

$IS_3$:

| | | | | |
|---|---|---|---|---|
| a | b | c | a | c |

# Finite-state Automata (9/23)

$\Sigma = \{ a, b, c \}$




| State | Input a | Input b | Input c |
|---|---|---|---|
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

$IS_1$:

| | | | |
|---|---|---|---|
| a | b | c | a |

$IS_2$:

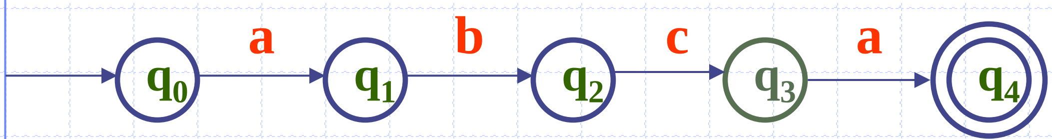
| | | | |
|---|---|---|---|
| c | c | b | a |

$IS_3$:

| | | | | |
|---|---|---|---|---|
| a | b | c | a | c |

# Finite-state Automata (10/23)

$\Sigma = \{ a, b, c \}$




| | Input | | |
|---|---|---|---|
| **State** | **a** | **b** | **c** |
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

**$IS_1$:**

| | | | |
|---|---|---|---|
| **a** | **b** | **c** | a |

$IS_2$:

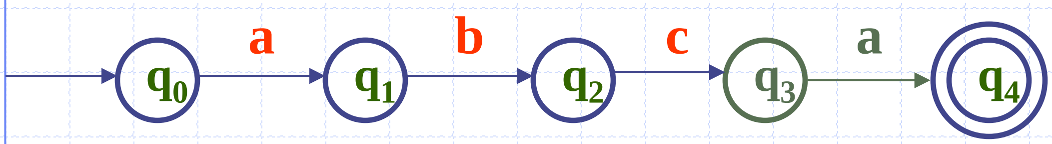
| | | | |
|---|---|---|---|
| c | c | b | a |

$IS_3$:

| | | | | |
|---|---|---|---|---|
| a | b | c | a | c |

# Finite-state Automata (11/23)

$\sum = \{ a, b, c \}$




| State | Input: a | b | c |
|---|---|---|---|
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

$IS_1$:

| | | | |
|---|---|---|---|
| a | b | c | a |

$IS_2$:

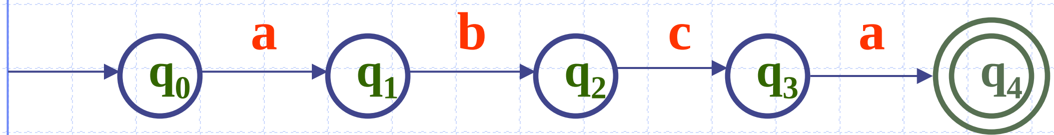
| | | | |
|---|---|---|---|
| c | c | b | a |

$IS_3$:

| | | | | |
|---|---|---|---|---|
| a | b | c | a | c |

# Finite-state Automata (12/23)

$\Sigma = \{ a, b, c \}$




| State | Input a | b | c |
|---|---|---|---|
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

$IS_1$:

| | | | |
|---|---|---|---|
| **a** | **b** | **c** | **a** |

$IS_2$:

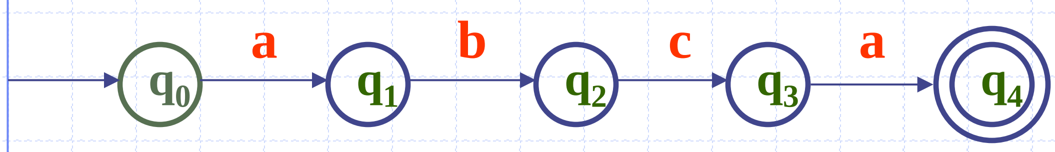
| | | | |
|---|---|---|---|
| c | c | b | a |

$IS_3$:

| | | | | |
|---|---|---|---|---|
| a | b | c | a | c |

# Finite-state Automata (13/23)

$\Sigma = \{ a, b, c \}$




| State | Input a | Input b | Input c |
|---|---|---|---|
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

$IS_1$:

| | | | |
|---|---|---|---|
| a | b | c | a |

$IS_2$:

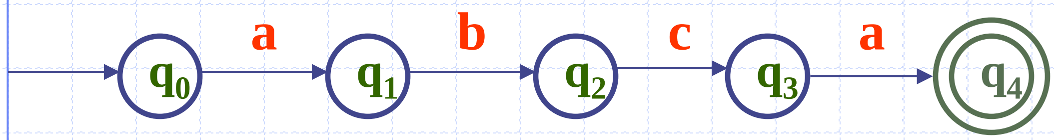
| | | | |
|---|---|---|---|
| c | c | b | a |

$IS_3$:

| | | | | |
|---|---|---|---|---|
| a | b | c | a | c |

# Finite-state Automata (14/23)

$\Sigma = \{ a, b, c \}$




| | Input | | |
|---|---|---|---|
| State | a | b | c |
| 0 | 1 | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ |
| 2 | ∅ | ∅ | 3 |
| 3 | 4 | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ |

$IS_1$:

| | | | |
|---|---|---|---|
| a | b | c | a |

$IS_2$:

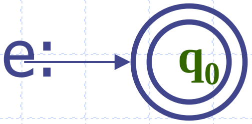
| | | | |
|---|---|---|---|
| c | c | b | a |

**$IS_3$:**

| | | | | |
|---|---|---|---|---|
| **a** | **b** | **c** | **a** | **c** |

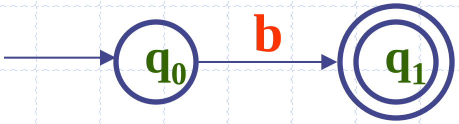
# Finite-state Automata (22/23)

- An FSA defines a regular language over an alphabet $\Sigma$:
  - $\varnothing$ is a regular language:



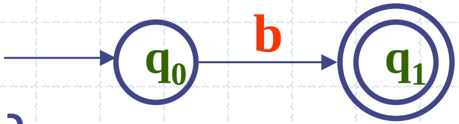

  - Any symbol from $\Sigma$ is a regular language:

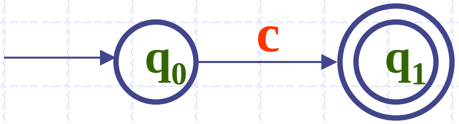
$\Sigma = \{ a, b, c\}$

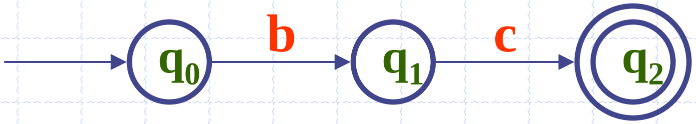



  - Two concatenated regular languages is a regular language:

$\Sigma = \{ a, b, c\}$

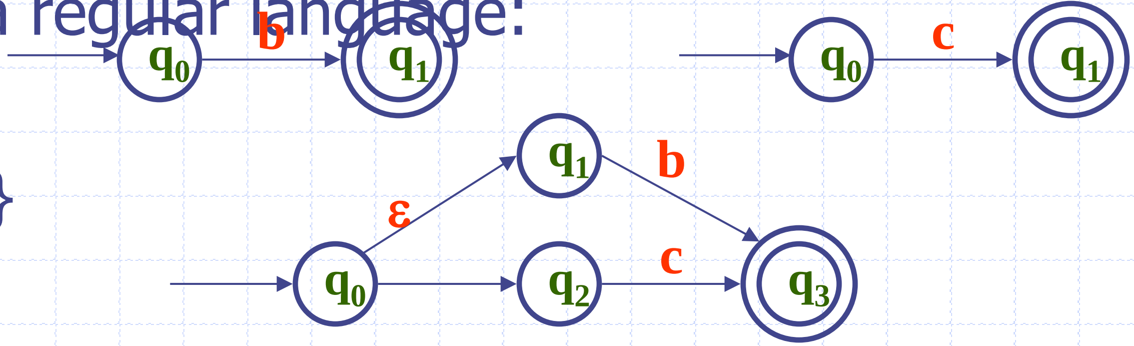


# Finite-state Automata (23/23)

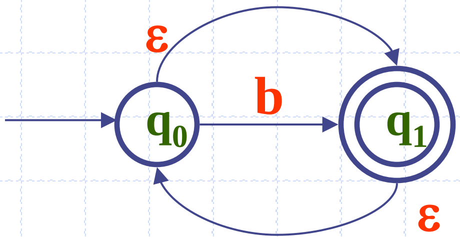
- regular language (continued):
  - The union (or disjunction) of two regular languages is a regular language:




$\Sigma$ = { a, b, c}

  - The Kleene closure (denoted by the Kleene star: *) of a regular language is a regular language:

$\Sigma$ = { a, b, c}



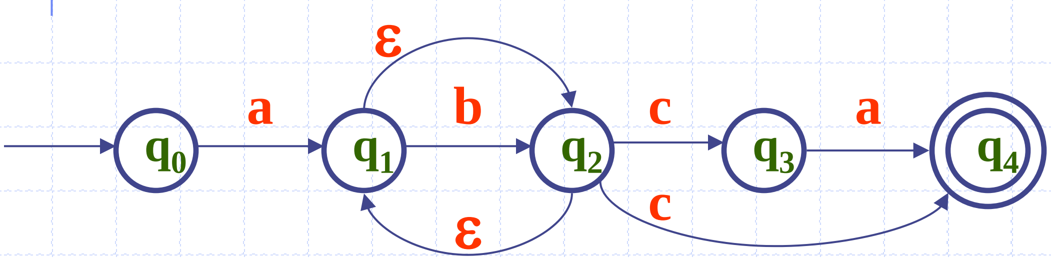
# Finite-state Automata (15/23)

- Determinism
  - An FSA may be either deterministic (DFSA or DFA) or non-deterministic (NFSA or NFA).
    - An FSA is deterministic if its behavior during recognition is fully determined by the state it is in and the symbol to be consumed.
      - I.e., given an input string, only one path may be taken through the FSA.
    - Conversely, an FSA is non-deterministic if, given an input string, more than one path may be taken through the FSA.
      - One type of non-determinism is $\varepsilon$-transitions, i.e. transitions which consume the empty string (no symbols).

# Finite-state Automata (16/23)

- An example NFA:

$\Sigma = \{ a, b, c \}$




| State | Input a | b | c | ε |
|---|---|---|---|---|
| 0 | 1 | ∅ | ∅ | ∅ |
| 1 | ∅ | 2 | ∅ | 2 |
| 2 | ∅ | ∅ | 3,4 | 1 |
| 3 | 4 | ∅ | ∅ | ∅ |
| 4 | ∅ | ∅ | ∅ | ∅ |

– The above NFA is equivalent to the regular expression `/ab*ca?/`.

# Finite-state Automata (17/23)

- String recognition with an NFA:
  - Backup (or backtracking): remember choice points and revisit choices upon failure
  - Look-ahead: choose path based on foreknowlege about the input string and available paths
  - Parallelism: examine all choices simultaneously

# Finite-state Automata (18/23)

- Recognition as search
  - Recognition can be viewed as selection of the correct path from all possible paths through an NFA (this set of paths is called the state-space)
  - Search strategy can affect efficiency: in what order should the paths be searched?
    - Depth-first (LIFO [last in, first out]; stack)
    - Breadth-first (FIFO [first in, first out]; queue)
    - Depth-first uses memory more efficiently, but may enter into an infinite loop under some circumstances

# RegExps

- The extended use of regular expressions is in many modern languages:
  - Perl, php, Java, python, …
- Can use regexps to specify the rules for any set of possible strings you want to match
  - Sentences, e-mail addresses, ads, dialogs, etc
- "Does this string match the pattern?", or "Is there a match for the pattern anywhere in this string?"
- Can also define operations to do something with the matched string, such as extract the text or substitute for it
- Regular expression patterns are compiled into a executable code within the language

# Regular Expressions

- Regexp syntax is a superset of the notation required to express a regular language.
  - Some examples and shortcuts:
    1. `/[abc]/ = /a|b|c/` Character class; disjunction
    2. `/[b-e]/ = /b|c|d|e/` Range in a character class
    3. `/[\012\015]/ = /\n|\r/` Octal characters; special escapes
    4. `/./ = /[\x00-\xFF]/` Wildcard; hexadecimal characters
    5. `/[^b-e]/ = /[\x00-af-\xFF]/` Complement of character class
    6. `/a*/ /[af]*/ /(abc)*/` Kleene star: zero or more
    7. `/a?/ = /a|/ /(ab|ca)?/` Zero or one
    8. `/a+/ /([a-zA-Z]1|ca)+/` Kleene plus: one or more
    9. `/a{8}/ /b{1,2}/ /c{3,}/` Counters: exact repeat quantification

# Regular Expressions

- Anchors
  - Constrain the position(s) at which a pattern may match
  - Think of them as "extra" alphabet symbols, though they actually consume ε (the zero-length string):
  - `/^a/` Pattern must match at beginning of string
  - `/a$/` Pattern must match at end of string
  - `/\bword23\b/` "Word" boundary: `/[a-zA-Z0-9_][^a-zA-Z0-9_]/` or `/[^a-zA-Z0-9_][a-zA-Z0-9_]/`
  - `/\B23\B/` "Word" non-boundary

# Regular Expressions

- Escapes
  - A backslash "\" placed before a character is said to "escape" (or "quote") the character. There are six classes of escapes:
    1. **Numeric character representation**: the octal or hexadecimal position in a character set: "`\012`" = "`\xA`"
    2. **Meta-characters**: The characters which are syntactically meaningful to regular expressions, and therefore must be escaped in order to represent themselves in the alphabet of the regular expression: "`[](){}|^$.?+*\`" (note the inclusion of the backslash).
    3. **"Special" escapes** (from the "C" language):

| | | | |
|---|---|---|---|
| newline: | "`\n`" = "`\xA`" | carriage return: | "`\r`" = "`\xD`" |
| tab: | "`\t`" = "`\x9`" | formfeed: | "`\f`" = "`\xC`" |

# Regular Expressions

- **Escapes** (continued)
  - Classes of escapes (continued):

4. **Aliases**: shortcuts for commonly used character classes. (Note that the capitalized version of these aliases refer to the complement of the alias's character class):
   - whitespace: "`\s`" = "`[ \t\r\n\f\v]`"
   - digit: "`\d`" = "`[0-9]`"
   - word: "`\w`" = "`[a-zA-Z0-9_]`"
   - non-whitespace: "`\S`" = "`[^ \t\r\n\f]`"
   - non-digit: "`\D`" = "`[^0-9]`"
   - non-word: "`\W`" = "`[^a-zA-Z0-9_]`"
5. **Memory/registers/back-references**: "`\1`", "`\2`", etc.
6. **Self-escapes**: any character other than those which have special meaning can be escaped, but the escaping has no effect: the character still represents the regular language of the character itself.

# Regular Expressions

- Memory/Registers/Back-references
  - Many regular expression languages include a memory/register/back-reference feature, in which sub-matches may be referred to later in the regular expression, and/or when performing replacement, in the replacement string:
    - Perl: `/(\w+)\s+\1\b/` matches a repeated word
    - Python: `re.sub(”(the\s+)the(\s+|\b)”,”\1”,string)` removes the second of a pair of ‘the’s
  - **Note: finite automata cannot be used to implement the memory feature.**

## Regular Expression Examples

Character classes and Kleene symbols

[A-Z] = one capital letter

[0-9] = one numerical digit

[st@!9] = s, t, @, ! or 9

[A-Z] = matches G or W or E

does not match GW or FA or h or fun

[A-Z]+ = one or more consecutive capital letters

matches GW or FA or CRASH

[A-Z]? = zero or one capital letter

[A-Z]* = zero, one or more consecutive capital letters

matches on eat or EAT or I

so, [A-Z]ate

matches Gate, Late, Pate, Fate, but not GATE or gate

and [A-Z]+ate

matches: Gate, GRate, HEate, but not Grate or grate or STATE

and [A-Z]*ate

matches: Gate, GRate, and ate, but not STATE, grate or Plate

## Regular Expression Examples (cont'd)

[A-Za-z]  =  any single letter
so   [A-Za-z]+
matches on any word composed of only letters,
but will not match on "words":  bi-weekly ,  yes@SU  or IBM325

they will match on bi, weekly, yes, SU and IBM

a shortcut for [A-Za-z]  is \w, which in Perl also includes  _

so (\w)+   will match  on  Information,  ZANY, rattskellar and jeuvbaew

\s  will match whitespace
so   (\w)+(\s)(\w+)   will match     real estate    or   Gen Xers

## Regular Expression Examples (cont'd)

Some longer examples:

([A-Z][a-z]+)\s([a-z0-9]+)

matches: Intel c09yt745 but not IBM series5000

[A-Z]\w+\s\w+\s\w+[!]

matches: The dog died!

It also matches that portion of " he said, " The dog died! "

[A-Z]\w+\s\w+\s\w+[!]$

matches: The dog died!

But does not match "he said, " The dog died! " because the $ indicates end of Line, and there is a quotation mark before the end of the line

(\w+ats?\s)+

parentheses define a pattern as a unit, so the above expression will match:

Fat cats eat Bats that Splat

## **Regular Expression Examples** (cont'd)

To match on part of speech tagged data:

(\w+[-]?\w+\|[A-Z]+) will match on:

- bi-weekly|RB
- camera|NN
- announced|VBD

(\w+\|V[A-Z]+) will match on:

- ruined|VBD
- singing|VBG
- Plant|VB
- says|VBZ

(\w+\|VB[DN]) will match on:

- coddled|VBN
- Rained|VBD
- But not changing|VBG

## **Regular Expression Examples** (cont'd)

Phrase matching:

a\|DT ([a-z]+\|JJ[SR]?) (\w+\|N[NPS]+)

matches: a|DT loud|JJ noise|NN
a|DT better|JJR Cheerios|NNPS

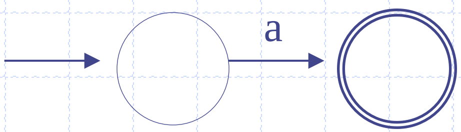
(\w+\|DT) (\w+\|VB[DNG])* (\w+\|N[NPS]+)+

matches: the|DT singing|VBG elephant|NN seals|NNS
an|DT apple|NN
an|DT IBM|NP computer|NN
the|DT outdated|VBD aging|VBG Commodore|NNNP computer|NN hardware|NN

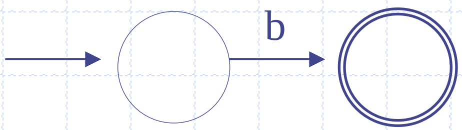
# RE to ε-NFA Example

- Convert R= (ab+a)* to an NFA
  - We proceed in stages, starting from simple elements and working our way up

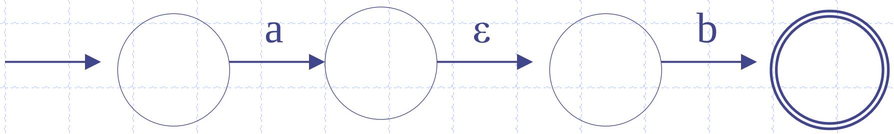
a




b

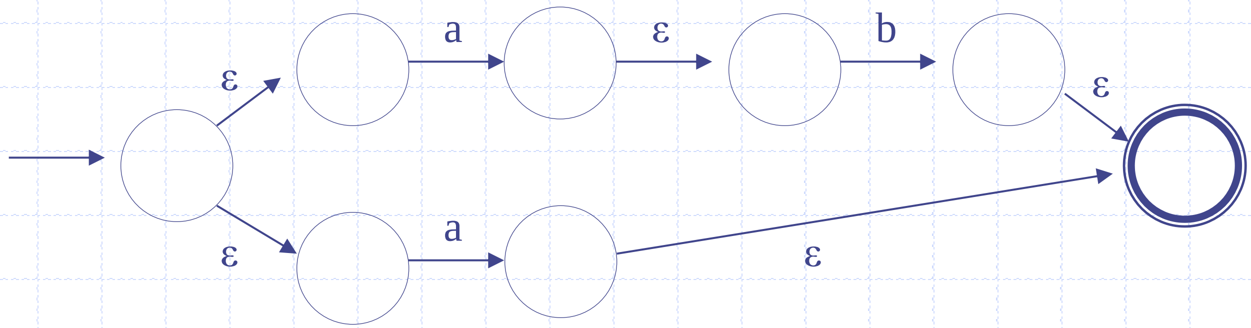



ab

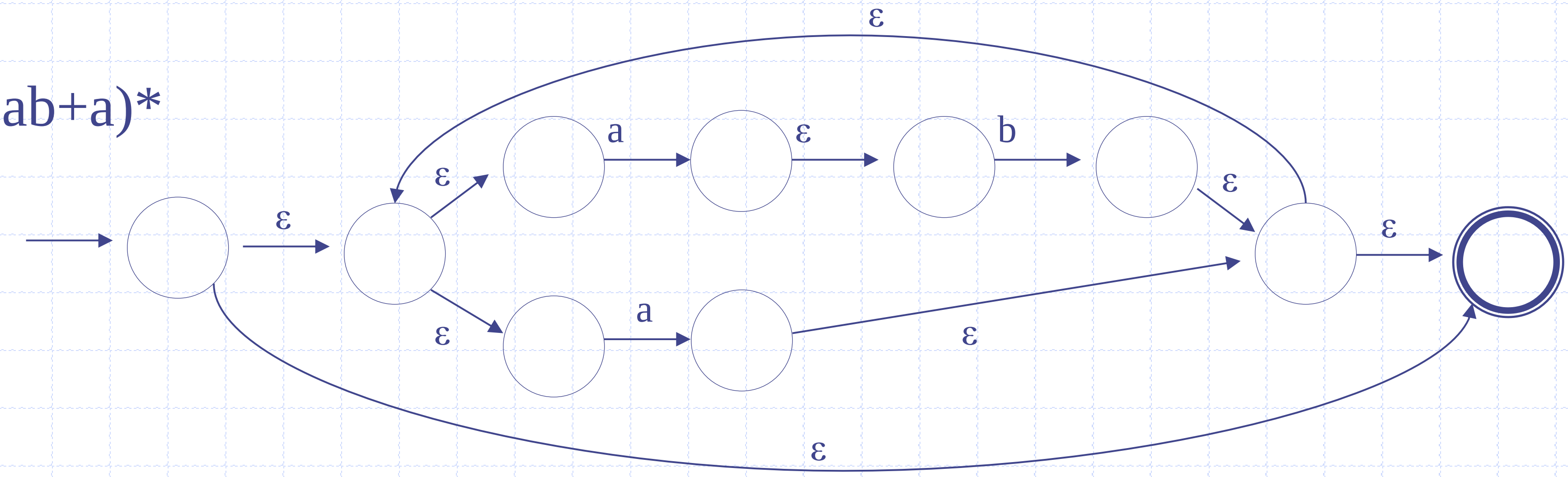


# RE to ε-NFA Example (2)

ab+a




(ab+a)*



# NFA to DFA

- See additional PDF for more examples…

# Conclusion

- Both regular expressions and finite-state automata represent regular languages.
- The basic regular expression operations are: concatenation, union/disjunction, and Kleene closure.
- The regular expression language is a powerful pattern-matching tool.
- Any regular expression can be automatically compiled into an NFA, to a DFA, and to a unique minimum-state DFA.
- An FSA can use any set of symbols for its alphabet, including letters and words.