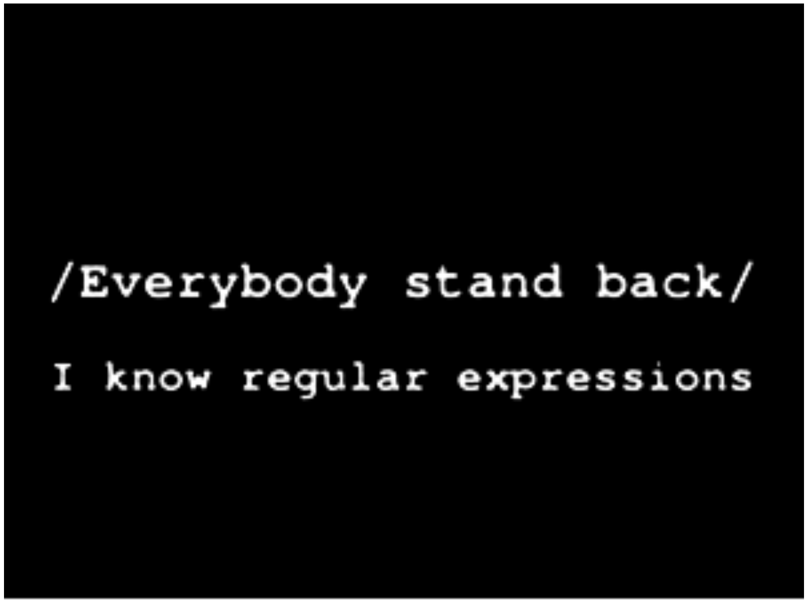
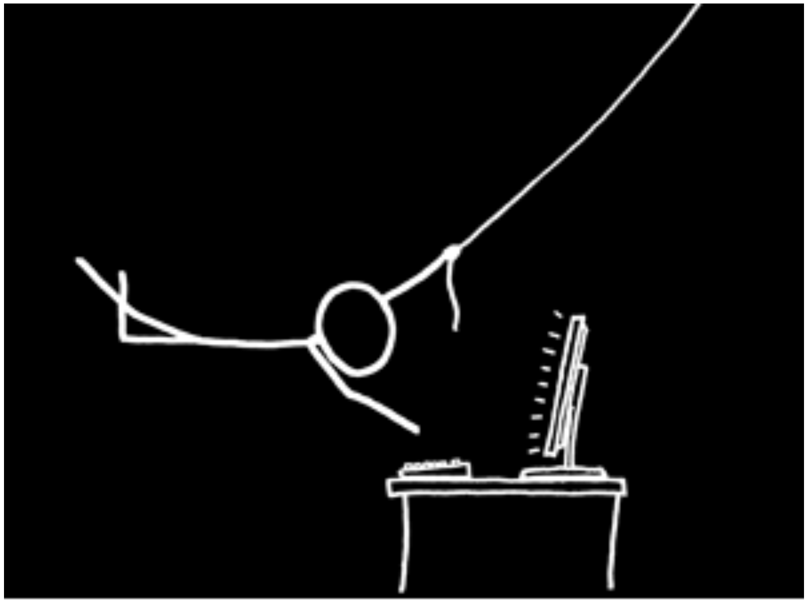
Regular Expressions
Finite State Automata

http://xkcd.com/208/



(Front)



(Back)

Overview

- Regular expressions are essentially a tiny, highly specialized programming language (embedded inside Python and other languages)
- Can use this little language 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?"‘
- Regular expressions can also be used as a language generator; regular expression languages are the first in the Chomsky hierarchy

Introduction

- Regular Expression a.k.a. regex, regexp or RE
  - A language for specifying text search strings
  - An example (matches names like “Jane Q. Public”):
    - Perl or Python: `/\b[A-Z][a-z]+ +[A-Z]\. +[A-Z][a-z]+\b/`
  - Applications/Tools using Regular Expressions:
    - All modern programming languages (most notably Perl), but also Python, Java, php, etc.
      - **In this talk, we use the (Perl) convention that regular expressions are surrounded by / - Python uses “**

Introduction

- Helpful applications of ‘regex’
  - Recognizing all email addresses

    …....@.......edu

    …....@.......gov

    .……@.......com
  - Recognizing all URLs
    - A fairly predictable set of characters & symbols selected from a finite set (e.g. a-z, www, http, ~, /)
  - What other ones?

Introduction

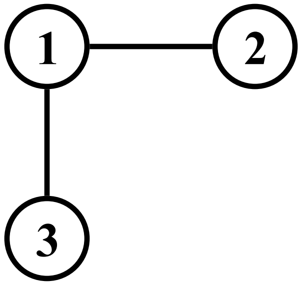
- In language theory, Regular Expressions specify a language that can be recognized by Finite State Automata

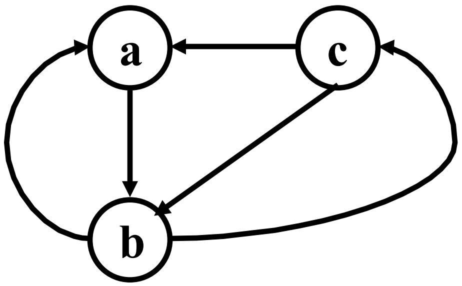
  a.k.a. Finite Automaton, Finite State Machine, FSA or FSM
  - An abstract machine which can be used to implement regular expressions (etc.).
  - Has a finite number of states, and a finite amount of memory (i.e., the current state).
  - Can be represented by directed graphs or transition tables

Automata Theory: 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.

- Graph: A set of nodes (or vertices), some or all of which may be connected by edges.
  - An example:
  - A directed graph example:





Regular Languages

- The first class of languages covered in Automata Theory.
- A regular expression defines a regular language over an alphabet ∑:
  - The empty set is a regular language: `//`
  - Any symbol from ∑ is a regular language:

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

    ∑ = { a, b, c} `/ab/` `/bc/` `/ca/`

Regular Languages

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

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

    $\sum = \{ a, b, c\}$ `/a*/` `/(ab|ca)*/`
  - Parentheses group a sub-language to override operator precedence
- The regular languages are the first in the Chomsky hierarchy (context-free languages and context-sensitive languages are the next)
- Regular languages are exactly the set of languages recognized by finite automata

Regular Expressions for Matching

- Perl’s (and Python’s) regular expression syntax is a superset of the notation required to express a regular language.
  - Makes more useful and succinct for matching expressions
  - 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. `/\n|\r/` Special escapes (newline, return)
    4. `/./` Wildcard matches any character
    5. `/[^b-e]/` Complement of character class
    6. `/a*/ /[af]*/ /(abc)*/` Kleene star: zero or more
    7. `/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 match 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

- Greediness
  - Regular expression counters/quantifiers which allow for a regular language to match a variable number of times (i.e., the Kleene star, the Kleene plus, “`?`”, “`{min,max}`”, and “`{min,}`”) are inherently *greedy:*
    - That is, when they are applied, they will match as many times as possible, up to `max` times in the case of “`{min,max}`”, at most once in the “`?`” case, and infinitely many times in the other cases.
    - Each of these quantifiers may be applied non-greedily, by placing a question mark after it. Non-greedy quantifiers will at first match the **minimum** number of times.
    - For example, against the string “From each according to his abilities”:
      - `/\b\w+.*\b\w+/` matches the entire string, and
      - `/\b\w+.*?\b\w+/` matches just “From each”

Using Regular Expressions

- In Perl, a regular expression can just be used directly for matching, the following is true if the string matches:

  string =~ m/ <regular expr> /
- But in many other languages, including Python (and Java), the regular expression is first defined with the compile function

  pattern = re.compile(‘<regular expr>’)
- Then the pattern can be used to match strings

  m = pattern.search(string)

  where m will be true if the pattern matches anywhere in the string

More Regular Expression Functions

- Python includes other useful functions
  - pattern.match – true if matches the beginning of the string
  - pattern.search – scans through the string and is true if the match occurs in any position
    These functions return a "MatchObject" or None if no match found
  - pattern.findall – finds all occurrences that match and returns them in a list
- MatchObjects also have useful functions
  - match.group( ) – returns the string(s) matched by the RE
  - match.start( ) – returns the starting position of the match
  - match.end( ) – returns the ending position of the match
  - match.span( ) – returns a tuple containing the start, end
  - And note that using the MatchObject as a condition in, for example, an If statement will be true, while if the match failed, None will be false.

Substitution with Regular Expressions

- Once a regular expression has matched in a string, the matching sequence may be replaced with another sequence of zero or more characters:
  - Convert “red” to “blue”
    - Perl: `$string =~ s/red/blue/g;`
    - **Python:** `p = re.compile(“red”)    string = p.sub(“blue”, string)`
  - Convert leading and/or trailing whitespace to an ‘=’ sign:
    - Python:
      ```
      p = re.compile(″^\s+|\s+$″)
      string = p.sub(″=″,string)
      ```
  - Remove all numbers from string: “These 16 cows produced 1,156 gallons of milk in the last 14 days.”
    - Python:
      ```
      p = re.compile(″ \d{1,3}(,\d{3})*″)
       string = p.sub(″″,string)
      ```
    - The result: “These cows produced gallons of milk in the last days.”

Extensions to 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: `p = re.compile(”(\w+)\s+\1\b)”)`
      `p.search(“Paris in the the spring”).group()`
      `returns ‘the the’`
  - 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 (equivalent to using | on single characters)

[A-Z] matches G or W or E (a single capital letter)

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 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

Helpful Regular Expression Websites

1. Tutorials:
   1.a. The Python Regular Expression HOWTO:

   http://docs.python.org/howto/regex.html

   A good introduction to the topic, and assumes that you will be using Python.

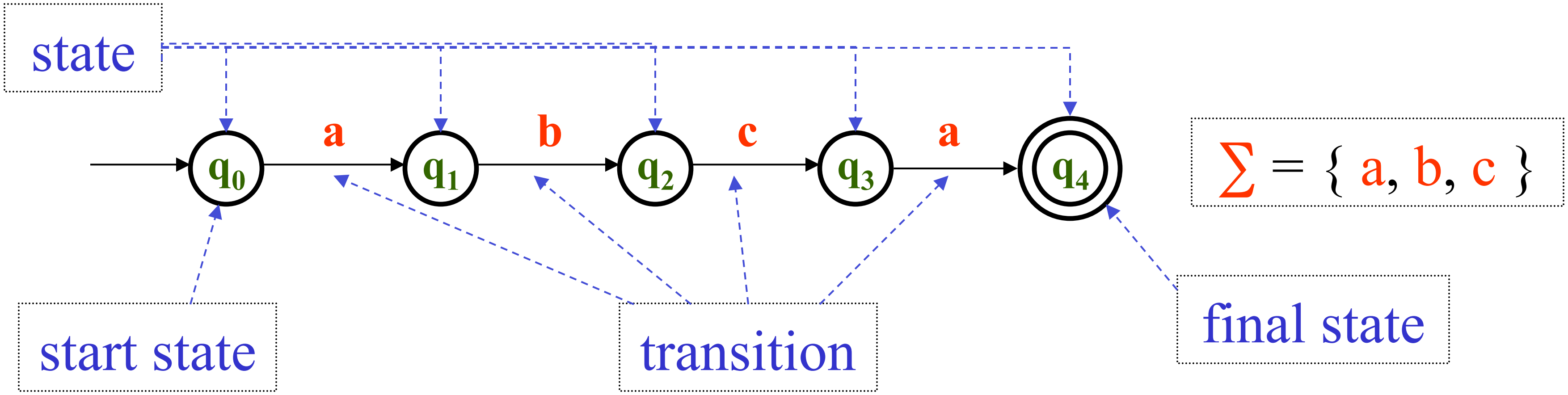
2. Free interactive testing/learning/exploration tools:
   2.a. Regular Expression tester:
   http://www.roblocher.com/technotes/regexp.aspx

3. Regular expression summary pages
   3.a. Dave Child’s Regular Expression Cheat Sheet from addedbytes.com
   http://www.addedbytes.com/download/regular-expressions-cheat-sheet-v2/pdf/

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 ε, 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)



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

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

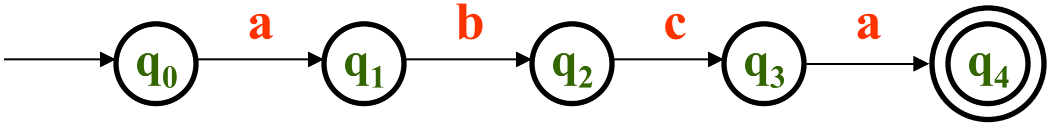
	Input		
State	**a**	**b**	**c**
0	1	∅	∅
1	∅	2	∅
2	∅	∅	3
3	4	∅	∅
4	∅	∅	∅

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)

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



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

IS_1:

a	b	c	a

IS_2:

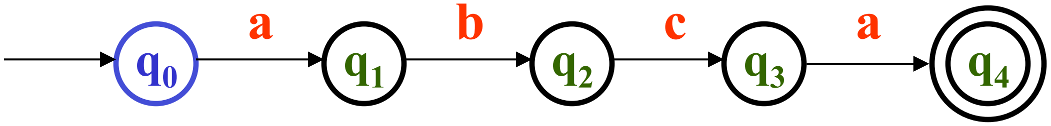
c	c	b	a

IS_3:

a	b	c	a	c

Finite-state Automata (4/23)

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

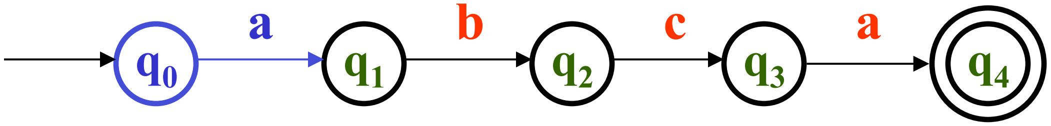


IS$_1$:	a	b	c	a	
IS$_2$:	c	c	b	a	
IS$_3$:	a	b	c	a	c

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

Finite-state Automata (5/23)

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



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

IS_1:

a	b	c	a

IS_2:

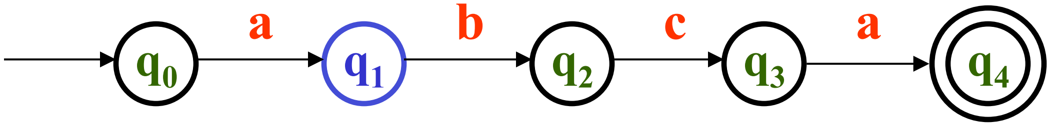
c	c	b	a

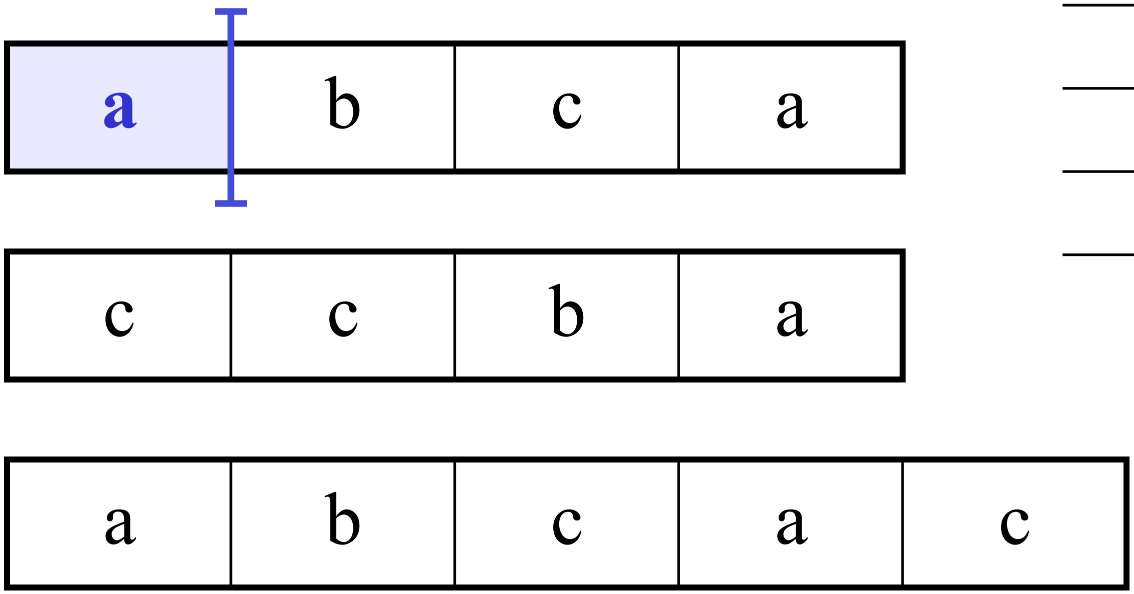
IS_3:

a	b	c	a	c

Finite-state Automata (6/23)

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





IS_1:

a	b	c	a

IS_2:

c	c	b	a

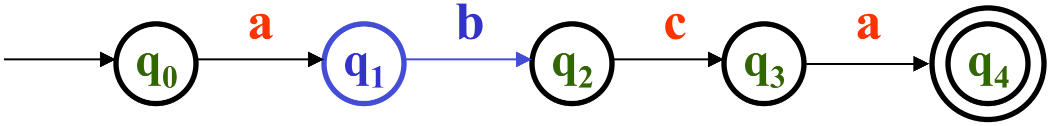
IS_3:

a	b	c	a	c

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

Finite-state Automata (7/23)

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

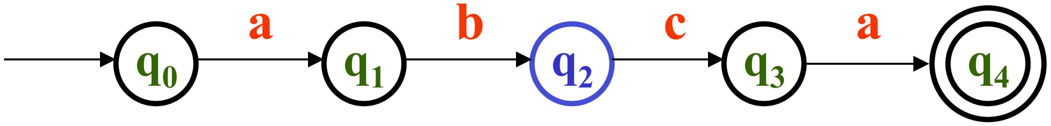


IS$_1$:	**a**	**b**	c	a	
IS$_2$:	c	c	b	a	
IS$_3$:	a	b	c	a	c

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

Finite-state Automata (8/23)

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



IS$_1$:

a	**b**	c	a

IS$_2$:

c	c	b	a

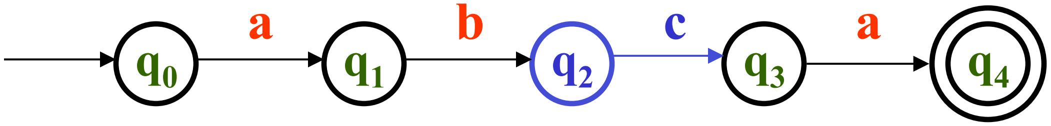
IS$_3$:

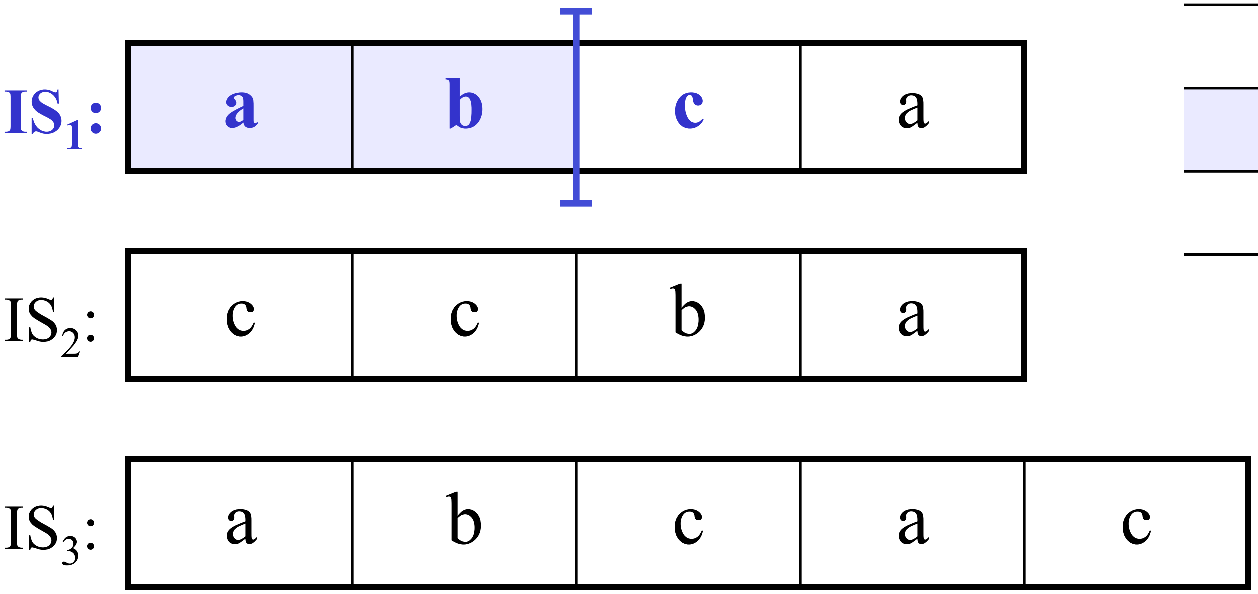
a	b	c	a	c

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

Finite-state Automata (9/23)

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





IS$_1$: a b c a

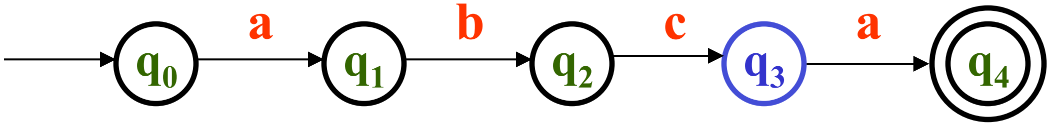
IS$_2$: c c b a

IS$_3$: a b c a c

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

Finite-state Automata (10/23)

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



IS$_1$:

a	**b**	**c**	a

IS$_2$:

c	c	b	a

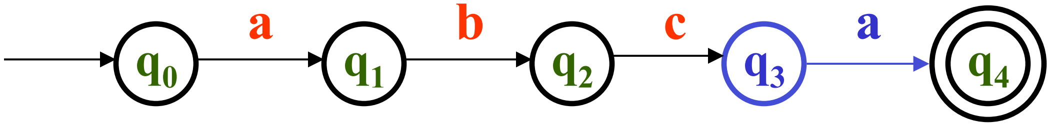
IS$_3$:

a	b	c	a	c

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

Finite-state Automata (11/23)

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



IS$_1$:

a	**b**	**c**	**a**

IS$_2$:

c	c	b	a

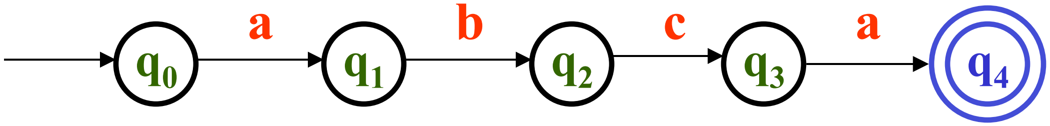
IS$_3$:

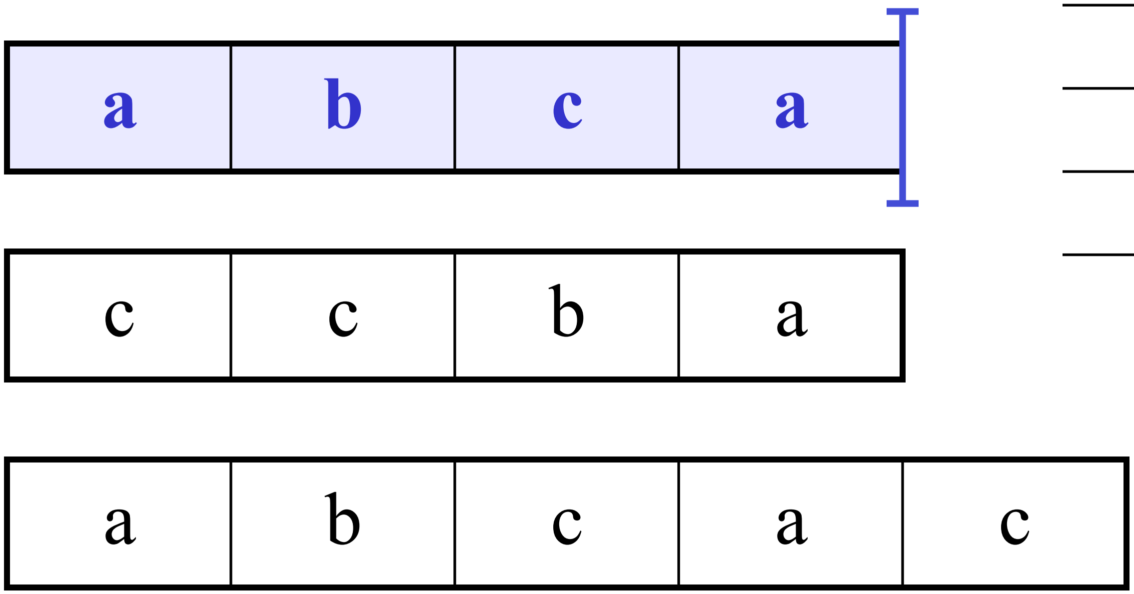
a	b	c	a	c

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

Finite-state Automata (12/23)

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





IS$_1$: **a** **b** **c** **a**

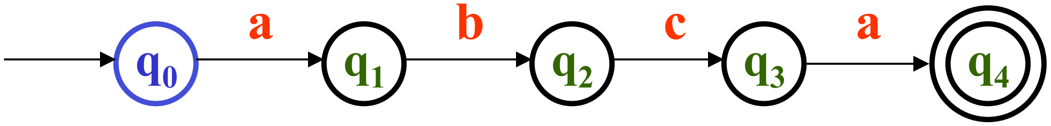
IS$_2$: c c b a

IS$_3$: a b c a c

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

Finite-state Automata (13/23)

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

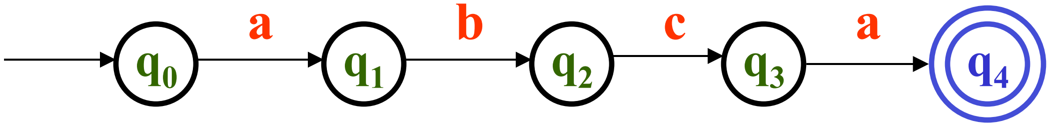


State	Input a	b	c
0	1	∅	∅
1	∅	2	∅
2	∅	∅	3
3	4	∅	∅
4	∅	∅	∅

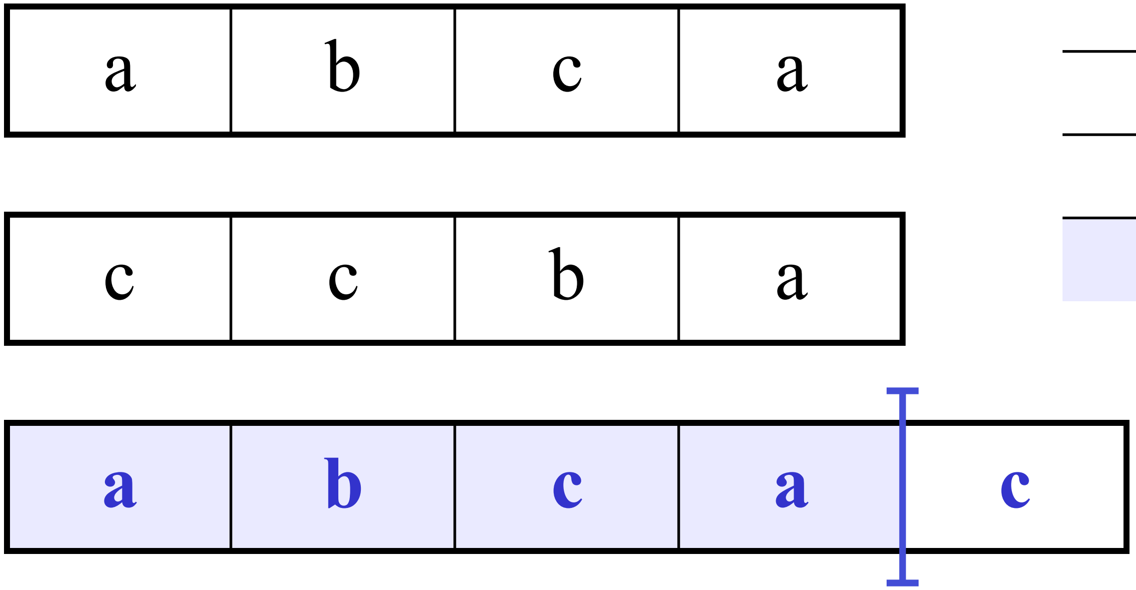
IS_1:	a	b	c	a	
IS_2:	**c**	c	b	a	
IS_3:	a	b	c	a	c

Finite-state Automata (14/23)

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



	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

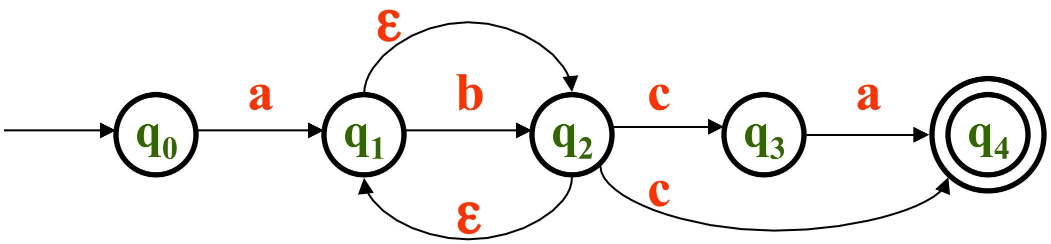
Finite-state Automata (15/23)

- 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 ε-transitions, i.e. transitions which consume the empty string (no symbols).

Finite-state Automata (16/23)

- An example NFA:

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



	Input			
State	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?/`.

Properties of REs and RSA

- 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.