Reg(ular expression|exp?)

Zaki Mughal

University of Houston:
CougarCS

2013 Feb 28

What is good for?

oblig. xkcd


WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Theory

formal language theory

[1] Huffman coding

Theory

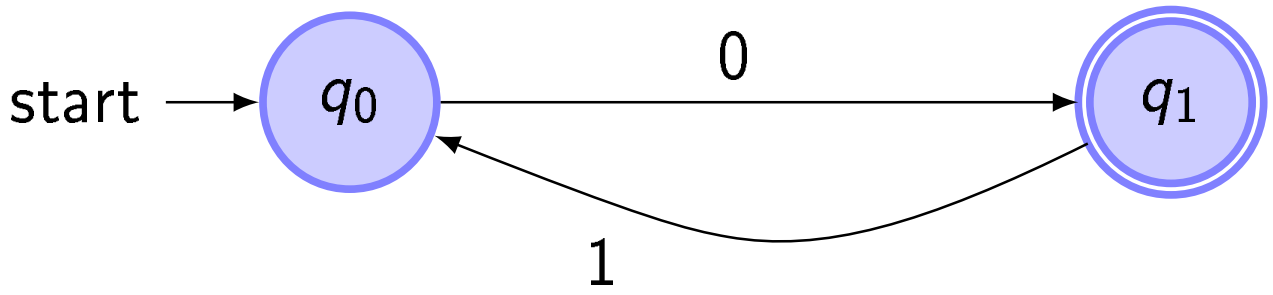
formal language theory

finite-state systems

- neural networks (McCulloch and Pitts, 1943 !!!)
- sequential circuits (Huffman[1], 1954 !!!)

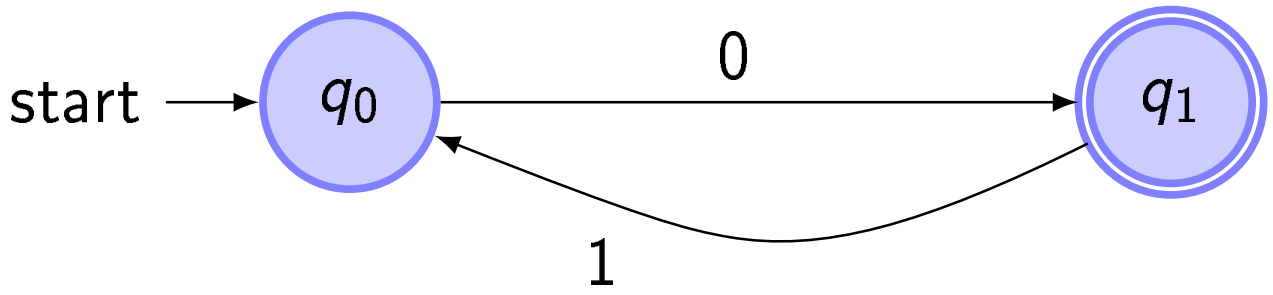
[1] Huffman coding

Finite state automata (FSA)



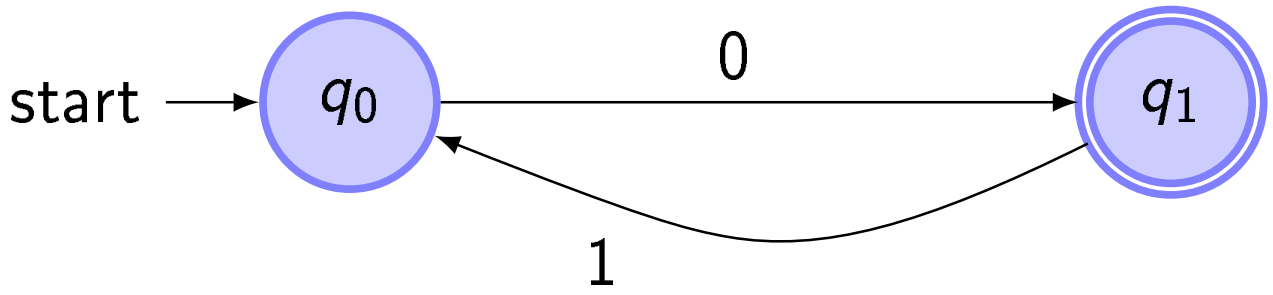
- q_0 is a start state (there can only be one!)

Finite state automata (FSA)



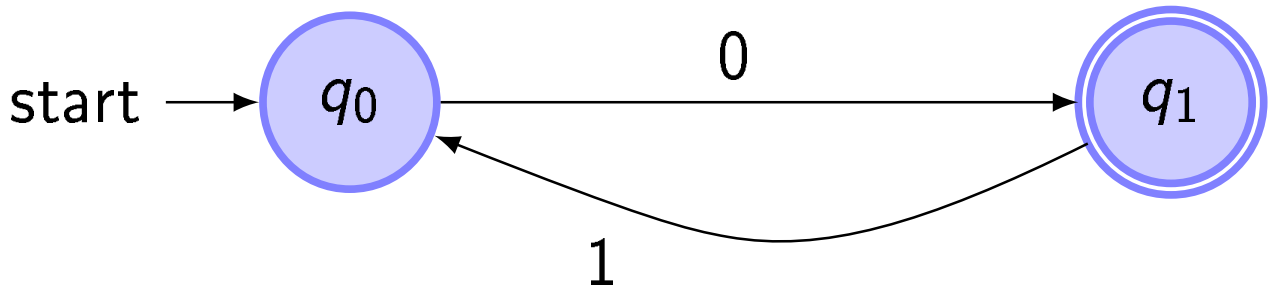
- q_0 is a start state (there can only be one!)
- q_1 is an accepting/final state (can be one of many)

Finite state automata (FSA)



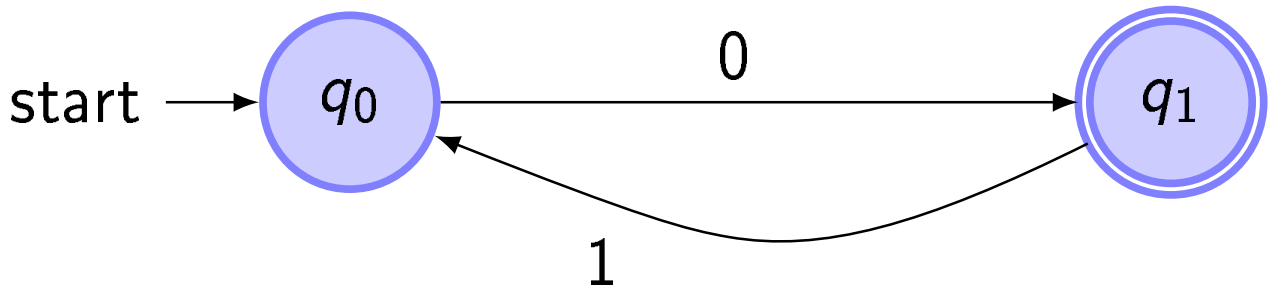
- q_0 is a start state (there can only be one!)
- q_1 is an accepting/final state (can be one of many)
- follow the symbols on each path

Finite state automata (FSA)



- q_0 is a start state (there can only be one!)
- q_1 is an accepting/final state (can be one of many)
- follow the symbols on each path
- all strings that end in the accepting state at end of the input are accepted (matched)

Finite state automata (FSA)



- q_0 is a start state (there can only be one!)
- q_1 is an accepting/final state (can be one of many)
- follow the symbols on each path
- all strings that end in the accepting state at end of the input are accepted (matched)

Matches the language that contains strings: 0, 010, 01010, 0101010, . . .

Regular expressions are equivalent to FSA.
Operations:

symbol the symbol ‘a’ matches the character a.

Regular expressions are equivalent to FSA.
Operations:

symbol the symbol ‘a’ matches the character a.

concatenation ‘ab’ matches ‘a’ followed by ‘b’

Regular expressions are equivalent to FSA.
Operations:

symbol the symbol ‘a’ matches the character a.

concatenation ‘ab’ matches ‘a’ followed by ‘b’

alternation ‘a+b’ matches either ‘a’ or ‘b’

Regular expressions are equivalent to FSA.
Operations:

symbol	the symbol ‘a’ matches the character a.
concatenation	‘ab’ matches ‘a’ followed by ‘b’
alternation	‘a+b’ matches either ‘a’ or ‘b’
Kleene star	‘a*’ matches the empty string, ‘a’, ‘aa’, ‘aaa’ (‘a’, 0 or more times)

Regular expressions are equivalent to FSA.
Operations:

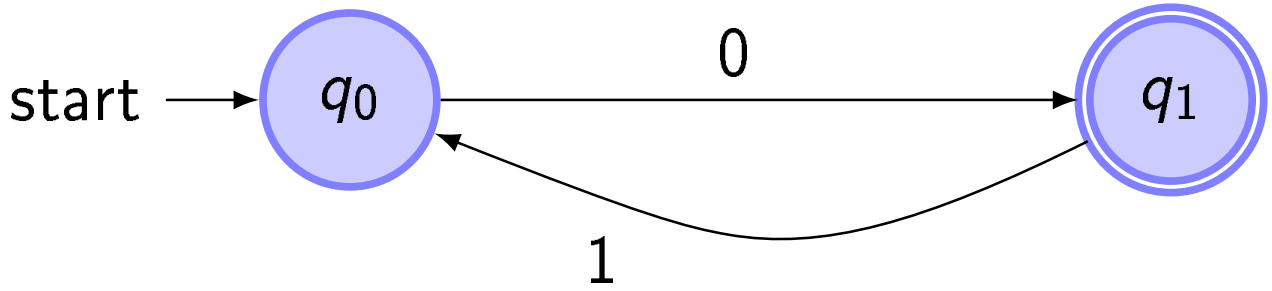
symbol	the symbol ‘a’ matches the character a.
concatenation	‘ab’ matches ‘a’ followed by ‘b’
alternation	‘a+b’ matches either ‘a’ or ‘b’
Kleene star	‘a*’ matches the empty string, ‘a’, ‘aa’, ‘aaa’ (‘a’, 0 or more times)
parentheses	can be used to group things

- ‘ab’ : ab

- ‘ab’ : ab
- ‘ab*’ : a, ab, abb, abbb, abbbb, . . .

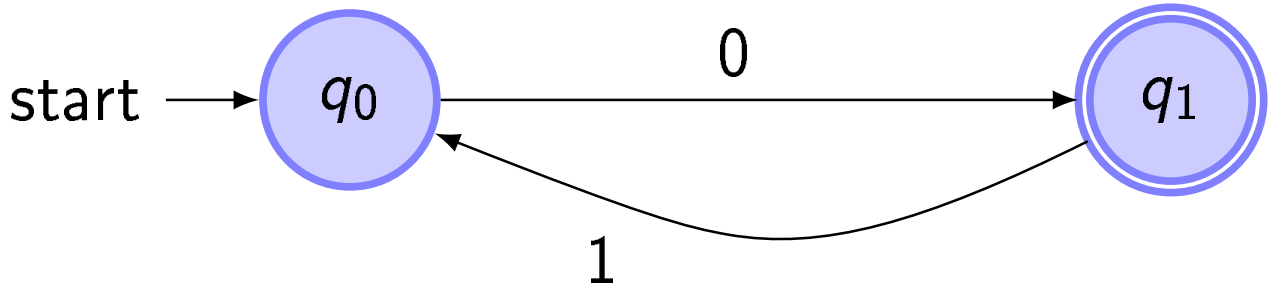
- ‘ab’ : ab
- ‘ab*’ : a, ab, abb, abbb, abbbb, . . .
- ‘(ab)*’ : a, ab, abab, ababab, abababab, . . .

Finite state automata (FSA)



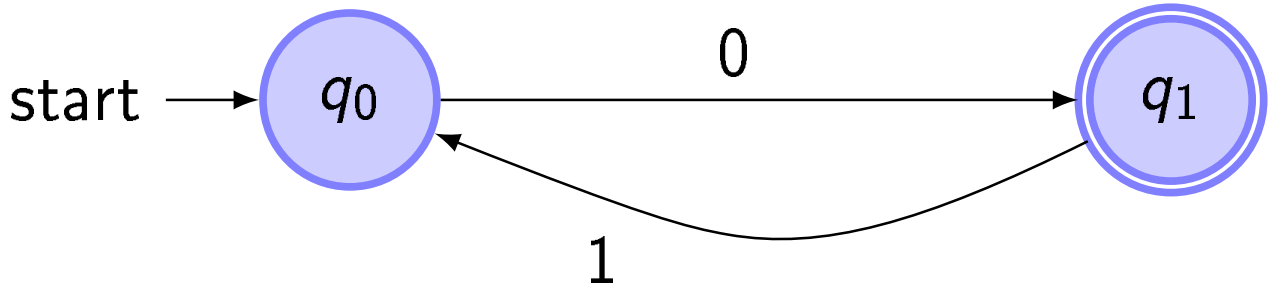
- $0(10)^*$

Finite state automata (FSA)



- $0(10)^*$
- $(01)^*0$

Finite state automata (FSA)



- $0(10)^*$
- $(01)^*0$
- $0 + (01)^*0$

  redundant

We're done...

We're done...

with the theory

Regular expressions ON COMPUTERS!

- Ken Thompson's (of Unix fame) work on text editors in late 60s.
- concatenation, parentheses, and Kleene star are the same
- alternation uses a pipe (|) instead of +

Regular expressions ON COMPUTERS!

- ab $\rightarrow$ ab
- $(ab)^*$ $\rightarrow$ (ab)*
- a+b $\rightarrow$ a|b

Quantifiers

- $(|a) \rightarrow a?$

  zero or one times: empty string or 'a'

Quantifiers

- (|a) $\rightarrow$ a?

  zero or one times: empty string or 'a'

- aa* $\rightarrow$ a+

  one or more times: a, aa, aaa, ...

Quantifiers

- (|a) $\rightarrow$ a?

  zero or one times: empty string or 'a'

- aa* $\rightarrow$ a+

  one or more times: a, aa, aaa, ...

And just for "fun":

- aaa+ $\rightarrow$ a{3,}

  5 or more times: aaa, aaaa, aaaaa, ...

Quantifiers

- (|a) $\rightarrow$ a?

  zero or one times: empty string or 'a'

- aa* $\rightarrow$ a+

  one or more times: a, aa, aaa, ...

And just for "fun":

- aaa+ $\rightarrow$ a{3,}

  5 or more times: aaa, aaaa, aaaaa, ...

- (aa|aaa) $\rightarrow$ a{2,3}

  2 or 3 times: aa, aaa

Quantifiers

- (|a) → a?

  zero or one times: empty string or 'a'

- aa* → a+

  one or more times: a, aa, aaa, ...

And just for "fun":

- aaa+ → a{3,}

  5 or more times: aaa, aaaa, aaaaa, ...

- (aa|aaa) → a{2,3}

  2 or 3 times: aa, aaa

- aa → a{2}

  exactly 2 times: aa

Character classes

- [0123456789] : any digit (0|1|2|3|4|5|6|7|8|9)
- [0-9] : same thing using a range
- default character classes:

  [[:digit:]], (\d)

  [[:word:]] (\w)

  [[:space:]] (\s)
- any character: . (dot)

Match a number: [1-9][[:digit:]]*

Anchors

- ^ (caret): beginning of line
- $ (dollar sign): end of line
- \b (dollar sign): word boundary

Match all lines ending with spaces: [[:space:]]+$

snore

- it's just syntax
- different for different engines
- grep (global regular expression print)
- `grep -color=auto`
- `alias grep='grep -color=auto'`

Languages

Let's look at some different languages.
Java, C, Perl, Python

Tools

- `txt2regex` - build regex for multiple languages/tools
- <http://www.regexper.com/> : railroad diagrams
- <http://www.regular-expressions.info/
- Friedl's "Mastering Regular Expressions" from O'Reilly books