NLP and IR

Formal beginnings
Regular expressions and Finite State Automata

Alexandre Rademaker[1]

FGV/EMAp

April 4, 2019

[1] Olga Zamaraeva, University of Washington

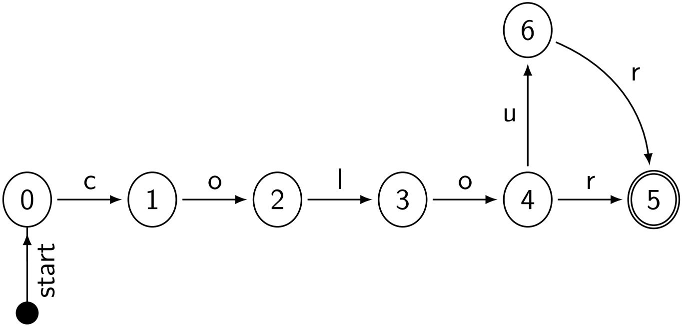
So, we will talk about: Regular Expressions

- sequence of characters that defines a (search) pattern
- \([0-9]{3}\)[0-9]{3}-[0-9]{4}
  - matches phone numbers
- You can play with regexes here:
  - `https://regex101.com/`
- RegEx as a tool:
- RegEx are equivalent to Finite State Automata
- Fun fact: Kleene developed RegEx to describe the capabilities of early neural nets

Which are directly related to: Finite State Automata

colou?r

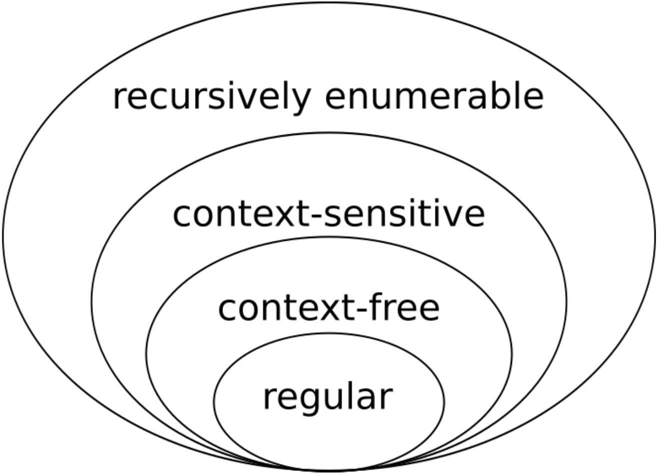
{color; colour}



The two things are equivalent in that they "generate" the same set of strings

Which describe: Regular Languages

...which are part of the Chomsky Hierarchy:



picture from Wikipedia.

What are formal languages?

Informally, a formal language is a set of strings (over some alphabet) and a set of rules.

What about just enumerating all possible strings? Is this a language, formally?

Formal languages

$\sum$ ={a,b,c} (alphabet)
$L = (a|b)c$ (regular language)

- Formally, a *grammar* (informally, set of rules) *generate*s or *accepts* (describes) a *language* (set of strings defined over an alphabet).
- One grammar has greater *generative power* than the other if can define a language that the other cannot.
- This is achieved by placing more or less constraints on how grammar rules can be written

Formal languages: Definitions

terminal specific word

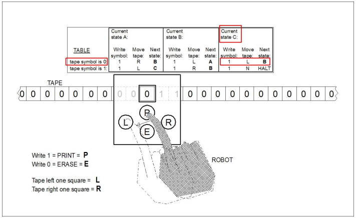
non-terminal generalization over a word, a class of words

rewrite rule (production) in what way can the symbols be grouped together?

NP $\rightarrow$Det Nominal
NP $\rightarrow$ ProperNoun
Nominal $\rightarrow$ Noun|Nominal Noun
Det $\rightarrow$ the
Det $\rightarrow$ a
Noun $\rightarrow$ flight

The Turing Machine (1936; Alan Turing (1912-1954))

- A reading/writing head is moving along the tape, executing instructions (program).
- "Turing equivalent" (or "Turing complete"): no restrictions as to what you can write to the tape.
- The most expressive language (can **run any program**, including self)
- Not actually a machine, it is a mental construct
- In fact, engineers often try to avoid Turing-complete solutions (why?)



Why need subTuring languages?

- Turing-complete languages are unbounded in power
- Why bother with SubTuring languages, then?
- "Accidentally Turing-complete": `http://beza1e1.tuxen.de/articles/accidentally_turing_complete.html`

Natural language complexity

This is the malt that the rat that the cat that the dog worried killed ate. – Victor H. Yngve (1960)

- *The Republicans who the senator who she voted for chastised were trying to cut all benefits for veterans. (J&M, p. 529)*
- What level of complexity occurs in natural language?
- *Models* of natural language differ in their *power* with respect to levels of *complexity*
- What type of models do we end up using *in practice*?

Natural language complexity

Swiss-German:

...mer em Hans es huss hälfed aastriiche

English:

...we helped Hans paint the house

"It is generally agreed that natural languages ... are not regular, although most attempted proofs of this are widely known to be incorrect."

Regular?

- No, because recursion.
- Recall the definition: a regular language is equivalent to a finite state automaton, while recursion requires potentially infinite stack

Natural language complexity

Swiss-German:

…mer em Hans es huss hälfed aastriiche

English:

…we helped Hans paint the house

"It is generally agreed that natural languages … are not regular, although most attempted proofs of this are widely known to be incorrect."

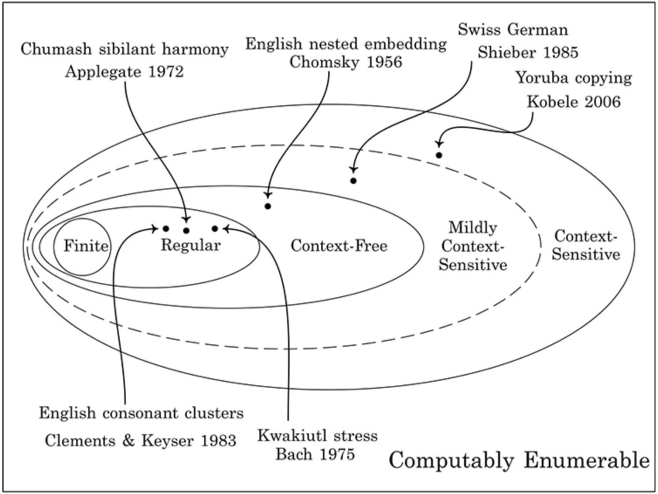
- Context-free?
  - Maybe, but maybe not, because Swiss German
- Context-sensitive?
  - $a^m b^n c^m d^n \ldots$
  - realistically, how big are m and n?

Why try to determine language complexity?

- We know the expressive power of our models
- How adequate is model X for language Y?
  - You want your program to react to certain input (language) in a certain way;
  - If the model's capabilities wrt this input are **limited**, better know that upfront
- Are natural languages *really* context-sensitive?
  - ...most programming languages are actually Turing-complete
  - what might this mean for natural languages?

Chomsky hierarchy and human language

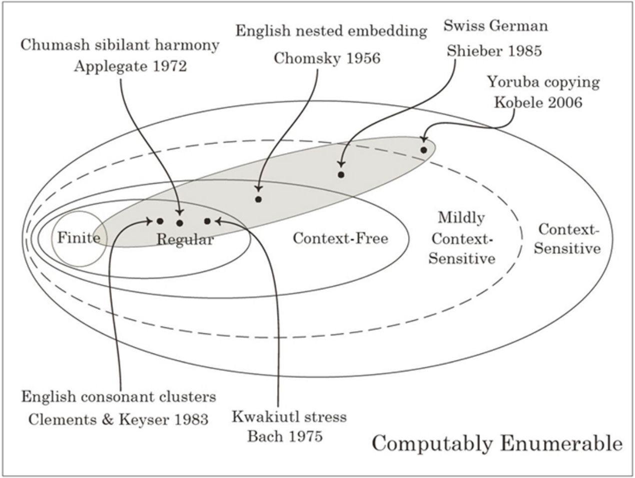
What should the learner acquire?



picture from Rawski & Heinz. Language, vol. 95 no. 1, 2019, pp. e125-e135.

Chomsky hierarchy and learnability

What should the learner acquire?



picture from Rawski & Heinz. Language, vol. 95 no. 1, 2019, pp. e125-e135.

Language complexity roadmap

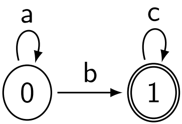
- We will start with regular languages by looking at regular expressions as search patterns and FSA for morphology
- We will continue with context free grammars by looking at probabilistic context free parsing
- We can work with a Turing-complete[2] language formalism (HPSG) in Assignment 5.

[2] We won't really use the full power of the complexity.

Regular languages

- (Right-linear:)
- At most one non-terminal on the LHS (left-hand side) and at most one terminal followed by one non-termial on the RHS.
- $A \rightarrow xB$ or $A \rightarrow x$ where x is a terminal and A,B are non-terminals

a*bc*



$S \rightarrow aS$
$S \rightarrow bA$
$A \rightarrow \epsilon$
$A \rightarrow cA$

Three views of the same object

Regular languages can be described by regular expressions or by finite state automata

- Regular language: a set of strings
- Regular expressions compactly describes a regular language
- Finite State Automaton (FSA) accepts or generates all the strings from the language (and no other strings)
- In this sense, the FSA and the regular expression are "equivalent" to each other

**Regular languages: Formal definition

Required symbols:

- ϵ is the empty string
- $\emptyset$ is the empty set
- $\sum$ is an alphabet (e.g. $\sum$ ={a,b,c,d})

The class of regular languages over $\sum$ is formally defined as:

- $\emptyset$ is a regular language
- $\forall$a $\in \sum \cup \epsilon$, {a} is a regular language
- If L1 and L2 are regular languages, then so are:
  - L1 $\cdot$ L2 $=$ {xy $|$ x $\in$ L1, y $\in$ L2} (concatenation)
  - L1 $\cup$ L2 (union or disjunction)
  - L1* (Kleene closure)

The Pumping Lemma (intuition)

- Why are languages of the form $a^n b^n$ not regular?
  - aka: why it is not possible to model syntax with FSA?

Regular expressions: Tools that use them

- A variety of unix tools (grep, sed, tr, awk . . .), editors (emacs, jEdit, . . .), and programming languages (perl, python, Java, . . .) incorporate regular expressions.
- Implementations are fairly efficient, but regex are still slow as far as computers go
- The various tools and languages differ w.r.t. the exact syntax of the regular expressions they allow but all are similar.

The syntax of regular expressions (I)

Regular expressions consist of

- strings of literal characters: `c`, `A100`, `natural language`, `30 years!`
- disjunction:
  - ordinary disjunction: `famil(y|ies)`
  - character classes: `[Tt]he`, `bec[oa]me`
  - ranges: `[A-Z]` (any capital letter)
- negation: `[^a]` (any symbol but a)
  `[^A-Z0-9]` (not an uppercase letter or number)

The syntax of regular expressions (II)

- counters
  - optionality: `?`
    `colou?r`
  - any number of occurrences: `*` (Kleene star)
    `[0-9]* years`
  - at least one occurrence: `+`
    `[0-9]+ dollars`
- wildcard for any character: `.`
  `beg.n` for any character in between `beg` and `n`

The syntax of regular expressions (III)

- Escaped characters: to specify a character with a special meaning (`*`, `+`, `?`, `(`, `)`, `|`, `[`, `]`) it is preceded by a backslash (\)
  e.g., a period is expressed as `\.`
- Operator precedence, from highest to lowest:

  parentheses `()`
  counters `* + ?`
  character sequences
  disjunction `|`

Regular expressions in python programming language

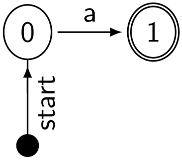
- `https://docs.python.org/3/library/re.html`
- (do study the above docs whenever you have a problem, rather than quickly looking at them and then trying to guess why you have a problem)
- Your best friend: `https://regex101.com/`

RegEx example: Single character

- Alphabet: $\sum = \{a\}$
- Language (set of strings): $L = \{a\}$
- Regular expression: a
- FSA:

RegEx example: Single character

- Alphabet: $\sum = \{a\}$
- Language (set of strings): $L = \{a\}$
- Regular expression: a
- FSA:

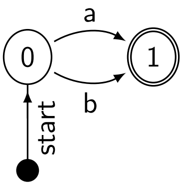


RegEx example: Disjunction/Union

- Alphabet: $\sum = \{a,b\}$
- Language (set of strings): $L = \{a,b\}$
- Regular expression: $a|b$
- FSA:

RegEx example: Disjunction/Union

- Alphabet: $\sum = \{a,b\}$
- Language (set of strings): $L = \{a,b\}$
- Regular expression: $a|b$
- FSA:

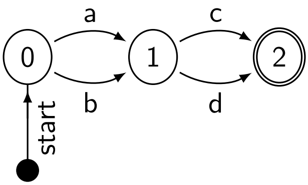


RegEx example: Concatenation

- Alphabet: $\sum = \{a,b,c,d\}$
- Language (set of strings): L1 $= \{a,b\}$, L2 $= \{c,d\}$
- Regular expression: $(a|b) \cdot (c|d)$
- FSA:

RegEx example: Concatenation

- Alphabet: $\sum = \{a,b,c,d\}$
- Language (set of strings): $L1 = \{a,b\}$, $L2 = \{c,d\}$
- Regular expression: $(a|b)\cdot(c|d)$
- FSA:



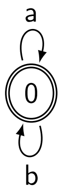
what about: $(ab)|(cd)$?

RegEx example: Kleene closure

- Alphabet: $\sum = \{$a,b$\}$
- Language (set of strings): L1 $= \{\epsilon$,a,b,aa,bb,ab,ba,bb...$\}$
- Regular expression: (a$|$b)*
- FSA:

RegEx example: Kleene closure

- Alphabet: $\sum = \{a,b\}$
- Language (set of strings): L1 $= \{\epsilon$,a,b,aa,bb,ab,ba,bb...$\}$
- Regular expression: (a|b)*
- FSA:

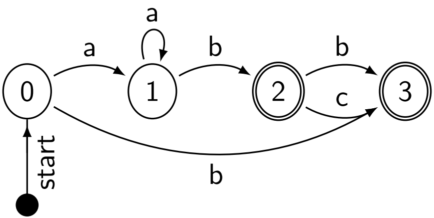


FSA as transition table

State	a	b	c
$>$0	1	3	-
1	1	2	-
2:	-	3	3
3:	-	-	-

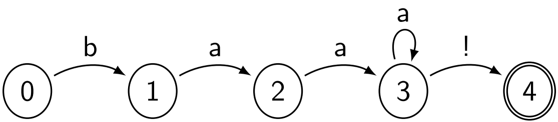
FSA as transition table

State	a	b	c
$>$0	1	3	-
1	1	2	-
2:	-	3	3
3:	-	-	-

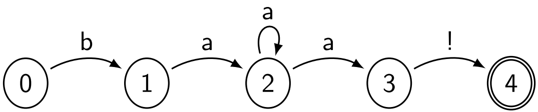


DFSA and NFSA

- **Deterministic Finite State Automaton (DFSA, DFA)**
  - Does not have "free" (ϵ, empty string) transitions
  - (Meaning: State changes only after reading an input)
  - Given an input sequence, you can predict uniquely the next state



- **Nondeterministic Finite State Automaton (NFSA, NFA)**
  - Can have "free" (ϵ, empty string) transitions
  - Given an input sequence, you can't always predict the next state
  - Can be convenient, e.g. to convert regex to a FSA

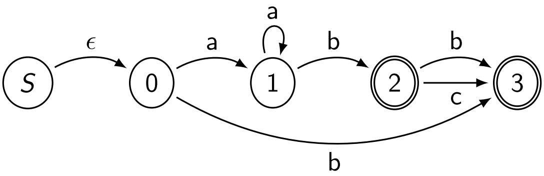


Converting FSA to regex

E.g. The State Removal method

- If transitions out of Start state:
  - Add a new Start state
  - Old Start state no longer start
  - Add an ϵ transition from new Start to old
- If more that one Accepting state or transitions out of Accepting state:
  - Add a new Accepting State
  - Add ϵ transitions from old Accepting states to new
  - Old Accepting states no longer accept
- Now, remove intermediate states one by one, combining simple inputs on the edges into more complex inputs.
- (Choose the simplest nodes to remove)
- This algorithm is easy to visualize but harder to apply systematically
- Look up e.g. *transition closure method* if you want systematic

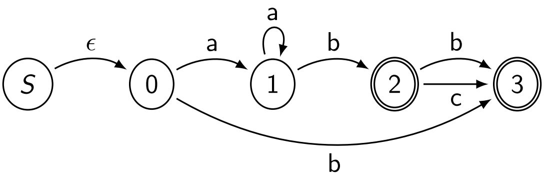
The State Removal Method



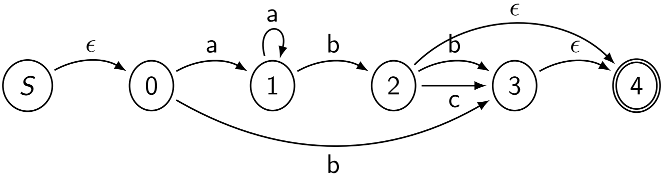
Getting rid of multiple accepting states:

Getting rid of State 1:

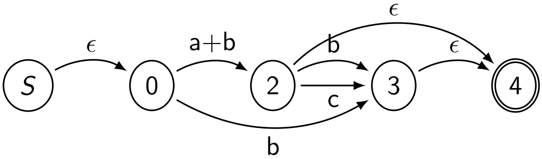
The State Removal Method



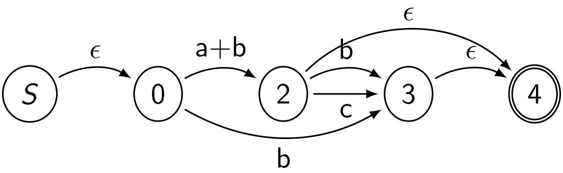
Getting rid of multiple accepting states:



Getting rid of State 1:



The State Removal Method - 2

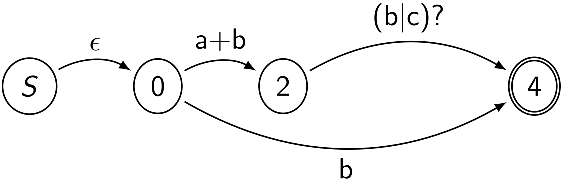


Getting rid of State 3:

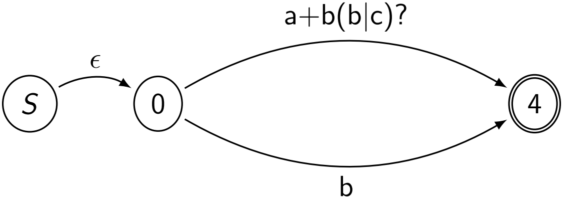
Getting rid of State 2:

The State Removal Method - 2

Getting rid of State 3:



Getting rid of State 2:



The final picture yields the regular expression:

(a+b(b|c)?)|b

Converting regex to FSA (Thompson's construction)

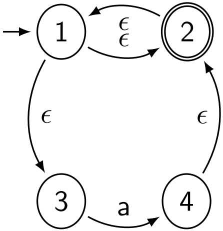
- Basically reversing the process above (with a bit more ϵ involved, if you want to follow the general case).
- (Why so many epsilon-transitions? Thompson construction assumes strictly 1 entry and 1 exit point for each state)
- you are operating not just on nodes in a graph, but on separate *automata*, so you need to preserve their start and accepting states
- A tutorial: `https://www.itu.dk/courses/BPRD/E2012/regex-to-nfa.pdf`
- Another tutorial: `http://web.cecs.pdx.edu/~harry/compilers/slides/LexicalPart3.pdf`

Converting regex to FSA: regex to NFA

a*(b|c)

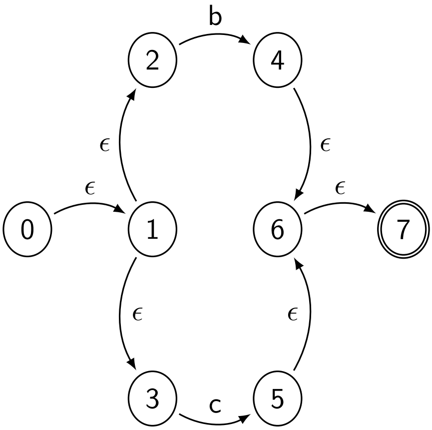
Converting regex to FSA: regex to NFA

a*(b|c)
a* (NFA, Kleene star of a):



Converting regex to FSA: regex to NFA

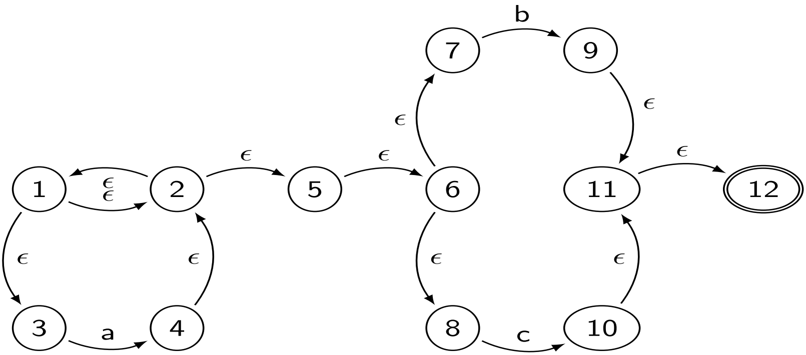
a*(b|c)
b|c (NFA, disjunction of b and c):



Converting regex to FSA: regex to NFA

a*(b|c)
a*(b|c) (NFA, concatenation of a* and b|c):

Note: we can get from 0 to 4 by *a*; from 4 to 4 by *a*; from 4 to 10 by *b*; from 4 to 11 by *c*. From 10 and 11, we can just halt.



Converting NFA to DFA

ϵ **closure**: sets of NFA states corresponding to each State to which you can get from that state by any # of ϵ arcs. (e.g. 1: 1,2,3,5,6,7,8)

State	NFA states
1	1,2,3,5,6,7,8
2	2,5,6,7,8
3	3
4	4,2,1,3,5,6,7,8
5	5,6,7,8
6	6,7,8
7	7
8	8
9	9,11,12
10	10,11,12
11	11,12
12	12

a*(b|c)

Converting NFA to DFA

All states to which you can get from State1 to State2 by some input and any *epsilon* transitions:

State	a	b	c
1	1,2,3,4,5,6,7,8	9,11,12	10,11,12
2	1,2,3,4,5,6,7,8	9,11,12	10,11,12
3	1,2,3,4,5,6,7,8	-	-
4	1,2,3,4,5,6,7,8	9,11,12	10,11,12
5	-	9,11,12	10,11,12
6	-	9,11,12	10,11,12
7	-	9,11,12	-
8	-	-	-
9	-	-	-
10	-	-	-
11	-	-	-
12	-	-	-
13	-	-	-

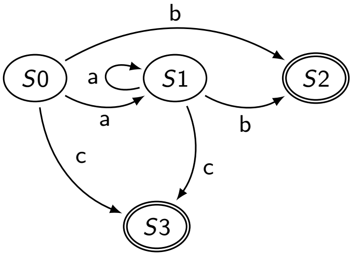
a*(b|c)

Converting NFA to DFA

Define new DFA states by sets of NFA states reachable by each input in the alphabet. Set Start State S0 to *epsilon*-closure of the NFA start state. S0 = {1,2,3,5,6,7,8}

State	a	b	c
S0	S1 {1,2,3,4,5,6,7,8}	S2 {9,11,12}	S3 {10,11,12}
S1	S1 {1,2,3,4,5,6,7,8}	S2 {9,11,12}	S3 {10,11,12}
S2	-	-	-
S3	-	-	-

a*(b|c) (DFA: could be minimized by combining S0 and S1)



What you need to know

- Regular expressions and FSA are “equivalent” (accept/generate the same set of strings (language))
- Regular expressions can be used to search for patterns
- Regular languages are the least expressive in the Chomsky hierarchy
- Models of language differ in their expressive power
- A formal grammar generates/accepts a language (a set of strings)
- An FSA accepts all strings from the regular language that it describes
- FSA operations: Disjunction, Concatenation, Kleene closure
- Algorithm to convert FSA to RegEx (optionally: and vice versa)
- Syntax of RegEx and how to use them in python