Regular expressions and automata

Introduction

Finite State Automaton (FSA)

Finite State Transducers (FST)

Regular expressions(I) (Res)

Standard notation for characterizing text sequences

Specifying text strings:

- Web search: **woodchuck**

  (with an optional final **s**) (lower/upper case)
- Computation of frequencies
- Word-processing (Word, Emacs,Perl)

Regular expressions (II)

A RE formula is a special language (an algebraic notation) to specify simple classes of strings: a sequence of symbols (i.e, alphanumeric characters).

woodchucks, a, song,!,Mary says

REs are used to

- Specify search strings - to define a pattern to search through a corpus
- Define a language

Regular expressions (III)

- Basically they are combinations of simple units (character or strings) with connectives as concatenation, disjunction, option, kleene star, etc.
- Used in languages as Perl or Python and Unix commands as grep, replace,...

Regular expressions (IV)

- Case sensitive: **woodchucks** different from **Woodchucks**

- [] means disjuntion

[Ww]oodchucks

[1234567890] (any digit)

[A-Z] an uppercase letter

- [^] means **cannot be**

  [^A-Z] not an uppercase letter

  [^Ss] neither 'S' nor 's'

Regular expressions (V)

- ? means preceding character or nothing

Woodchucks? means Woodchucks or Woodchuck

colou?r color or colour

- * (kleene star)- zero or more occurrences of the immediately previous character

  a* any string or zero or more as (a,aa, hello)

[0-9][0-9]* - any integer

- + one or more occurrences

[0-9]+

Regular expressions (VI)

- Disjunction operator | cat|dog
- There are other more complex operators
- Operator precedence hierarchy
- Very useful in substitutions (i.e. Dialogue)

Regular expressions (VII)

Useful to write patterns:

Examples of substitutions in dialogue

User: Men are all alike

ELIZA: IN WHAT WAY

*s/.*all.*/ IN WHAT WAY*

User: They're always bugging us about something

ELIZA: CAN YOU THINK OF A SPECIFIC EXAMPLE

*s/*always.*/ CAN YOU THINK OF A SPECIFIC EXAMPLE*

Regular expressions (VIII)

- Acronym detection

patterns acrophile

```
acro1 = re.compile('^([A-Z][,\.-/_])+$')
acro2 = re.compile('^([A-Z])+$')
acro3 = re.compile('^\d*[A-Z](\d[A-Z])*$')
acro4 = re.compile('^[A-Z][A-Z][A-Z]+[A-Za-z]+$')
acro5 = re.compile('^[A-Z][A-Z]+[A-Za-z]+[A-Z]+$')
acro6 = re.compile('^([A-Z][,\.-/_]){2,9}(\'s|s)?$')
acro7 = re.compile('^[A-Z]{2,9}(\'s|s)?$')
acro8 = re.compile('^[A-Z]*\d[-_]?[A-Z]+$')
acro9 = re.compile('^[A-Z]+[A-Za-z]+[A-Z]+$')
acro10 = re.compile('^[A-Z]+[/-][A-Z]+$')
```

Some readings

Kenneth R. Beesley and Lauri Karttunen, Finite State Morphology, CSLI Publications, 2003

Roche and Schabes 1997
Finite-State Language Processing. 1997. MIT Press, Cambridge, Massachusetts.

References to Finite-State Methods in Natural Language Processing
http://www.cis.upenn.edu/~cis639/docs/fsrefs.html

Some toolbox

ATT FSM tools
http://www2.research.att.com/~fsmtools/fsm/

Beesley, Kartunnen book
http://www.stanford.edu/~laurik/fsmbook/home.html

Carmel
http://www.isi.edu/licensed-sw/carmel/

Dan Colish's PyFSA (Python FSA)
https: //github.com/dcolish/PyFSA

Regular expressions and automata

- Regular expressions can be implemented by the finite-state automaton.
- Finite State Automaton (FSA) a significant tool of computational lingusitics. They are related to other computational tools:

  - Finite State Transducers (FST)

  - N-gram

  - Hidden Markov Models

Equivalence

Regular Expressions

Regular Languages

Finite State Automaton

Regular Languages (RL)

- Alphabet (vocabulary) Σ
- Concatenation operation
- Σ^* strings over Σ (free monoid)
- Language $L \subseteq \Sigma^*$
- Languages and grammars

L, L_1 y L_2 are languages

operations

concatenation $L_1 \cdot L_2 = \{u \cdot v \mid u \in L_1 \wedge v \in L_2\}$

union $L_1 \cup L_2 = \{u \mid u \in L_1 \vee u \in L_2\}$

intersection $L_1 \cap L_2 = \{u \mid u \in L_1 \wedge u \in L_2\}$

difference $L_1 - L_2 = \{u \mid u \in L_1 \wedge u \notin L_2\}$

complement $\overline{L} = \Sigma - L$

Finite State Automata (FSA)

$<\Sigma, Q, i, F, E>$

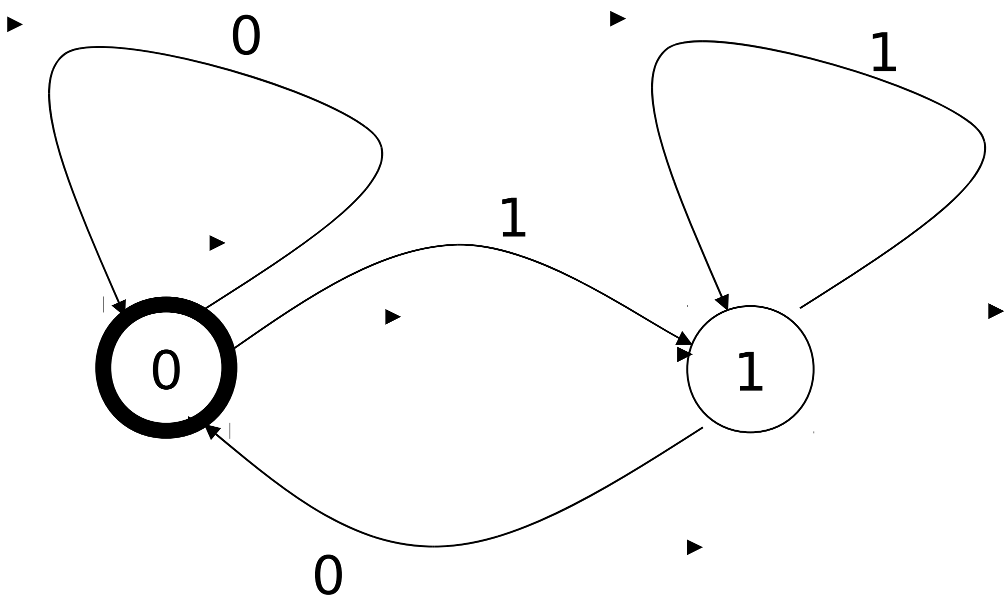
Σ	alphabet
Q	**finite** set of states
$i \in Q$	initial state
$F \subseteq Q$	final states set
$E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$	arc set
$E: \{d \mid d: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q\}$	transitions set

Example 1: Recognizes multiple of 2 codified in binary

Examples of numbers recognized

0

10 (2 in decimal)
100 (4 in decimal)
110 (6 in decimal)

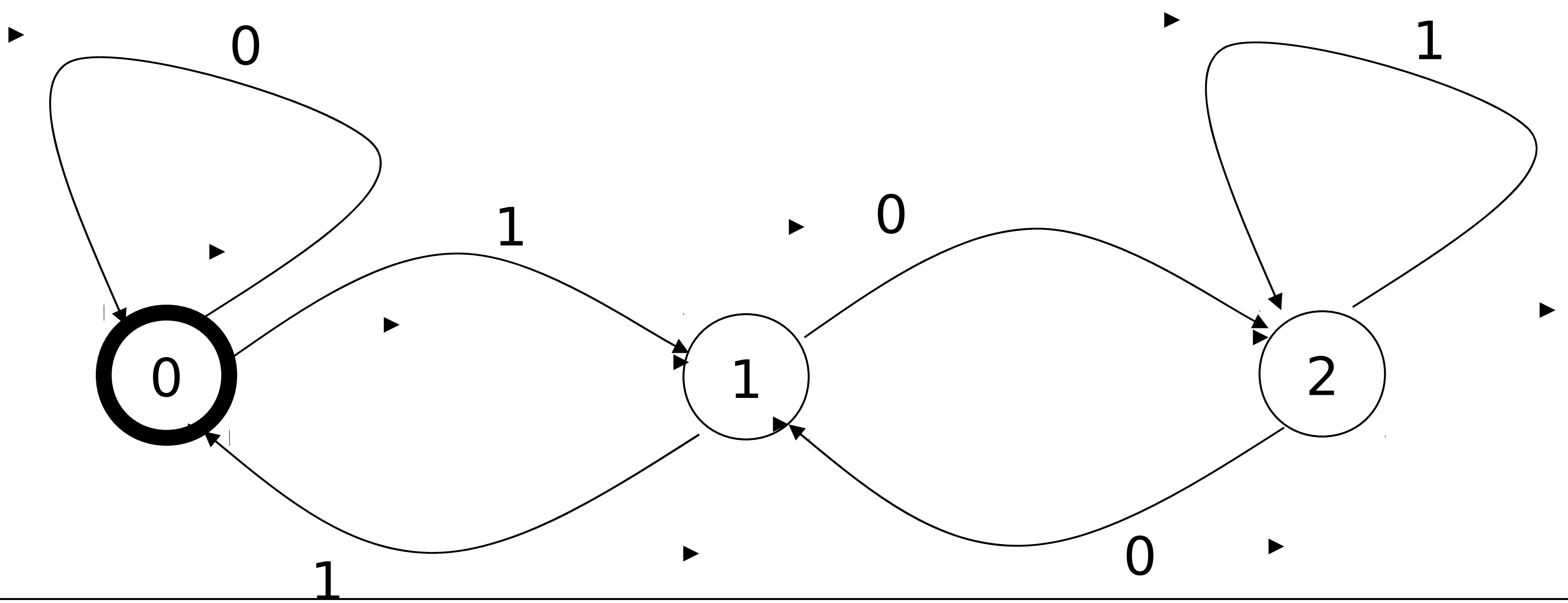


State 0:

The string recognized till now ends with 0

State 1:
The string recognized till now ends with 1

Example 2: Recognizes multiple of 3 codified in binary



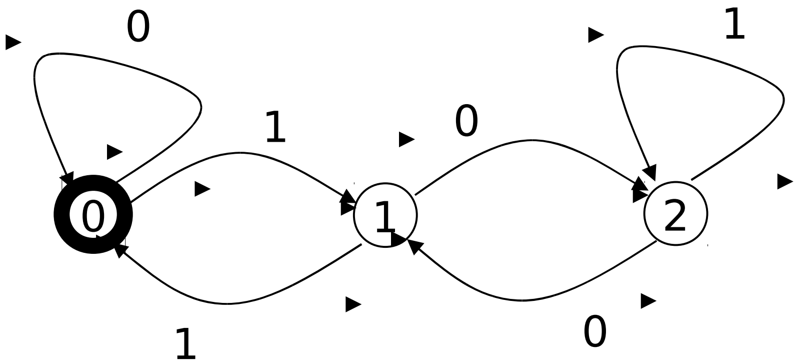
State 0: The string recognized till now is multiple of 3

State 1: The string recognized till now is multiple of 3 + 1

State 2: The string recognized till now is multiple of 3 + 2

The transition from a state to the following multiplies by 2 the current string and adds to it the current tag

Tabular representation of the FSA



	0	1
0	0	1
1	2	0
2	1	2

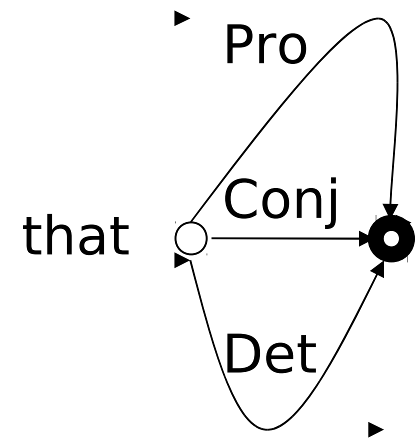
Recognizes multiple of 3 codified in binary

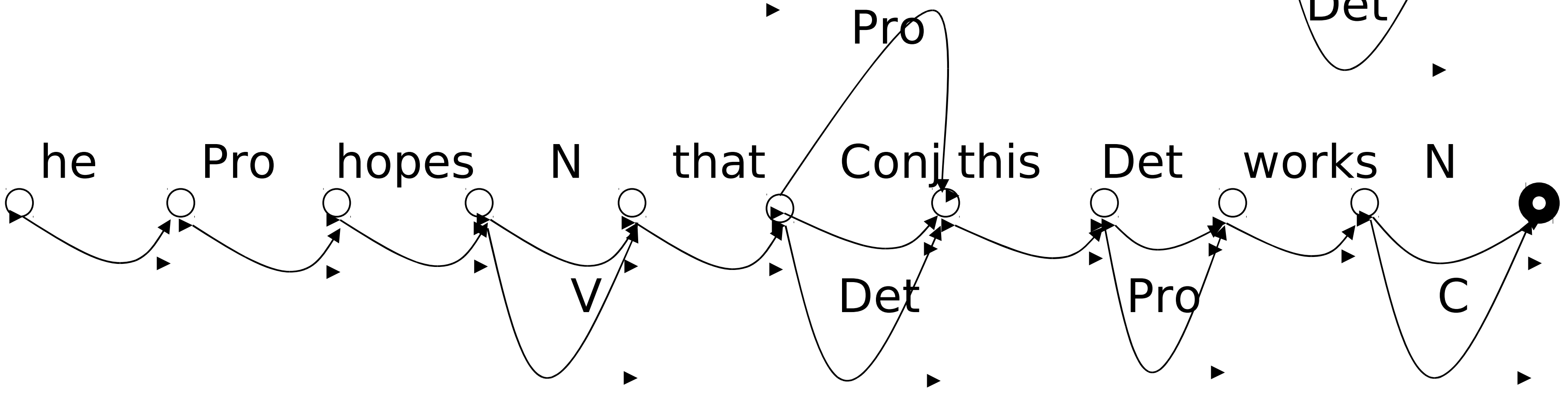
Properties of regular languages(RL) and FSA

- Let A a FSA
- L(A) is the language generated (recognized) by A
- The class of RL (o FSA) is closed under
  - union
  - intersection
  - concatenation
  - complement
  - Kleene star(A*)
- FSA can be determined
- FSA can be minimized

Example of the use of closure properties

Representation of the Lexicon



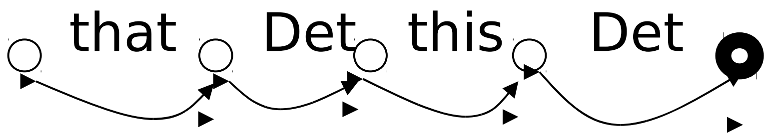


Let S the FSA:

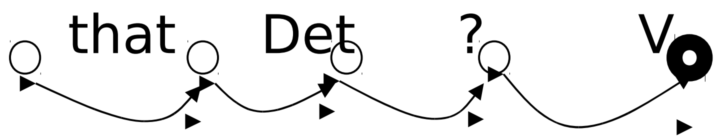
Representation of the sentence with POS tags

Restrictions (negative rules)

FSA C_1



FSA C_2

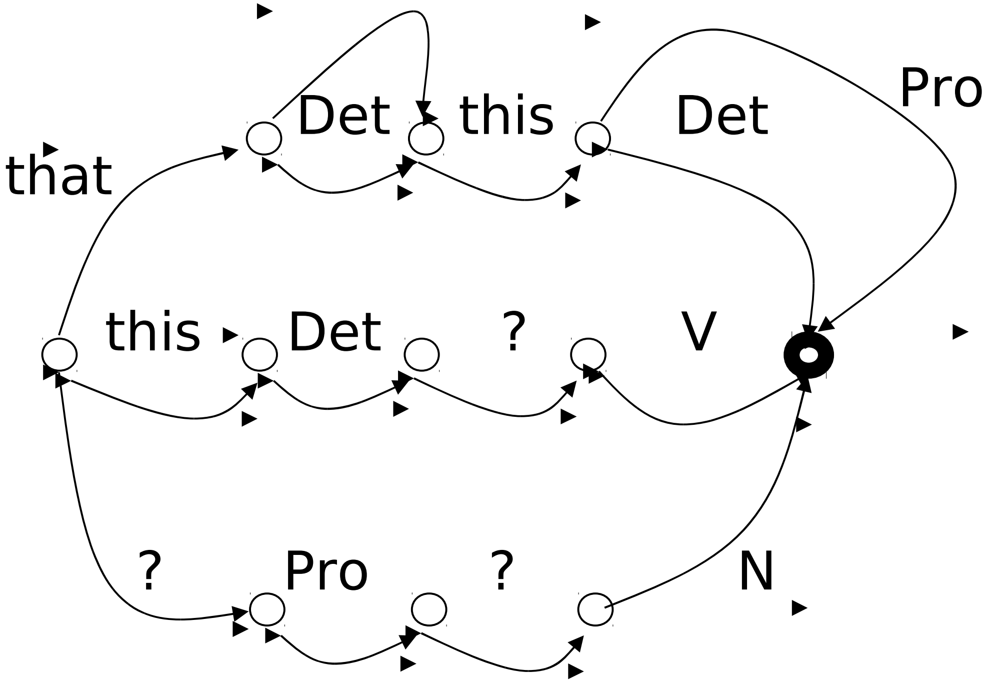


We are interested on

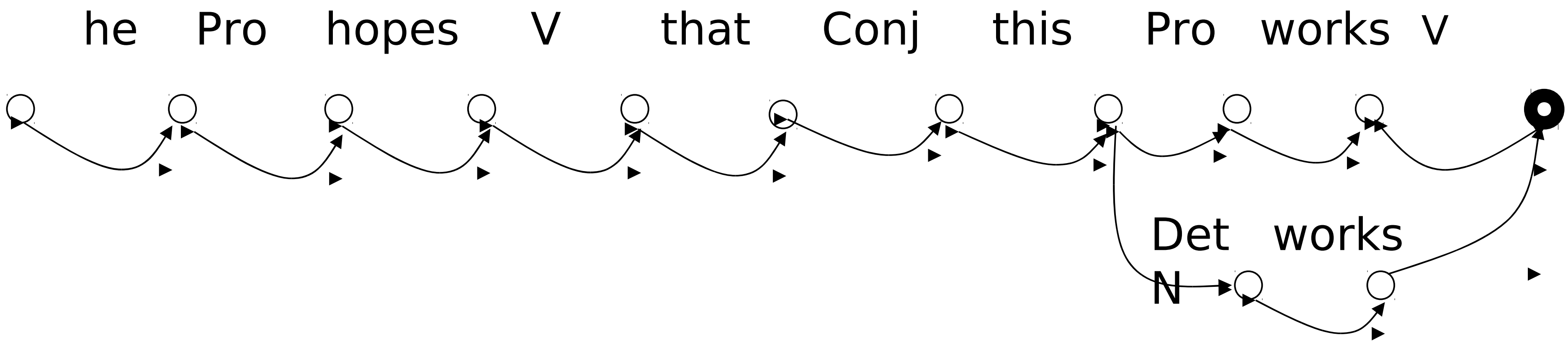
$$S - (\Sigma^* \bullet C1 \bullet \Sigma^*) - (\Sigma^* \bullet C2 \bullet \Sigma^*) =$$

$$S - (\Sigma^* \bullet (C1 \cup C2) \bullet \Sigma^*)$$

From the union of negative rules we can build a Negative grammar G = $\Sigma^* \bullet (C1 \cup C2 \cup \ldots \cup Cn) \bullet \Sigma^*)$



The difference between the two FSA S -G will result on:



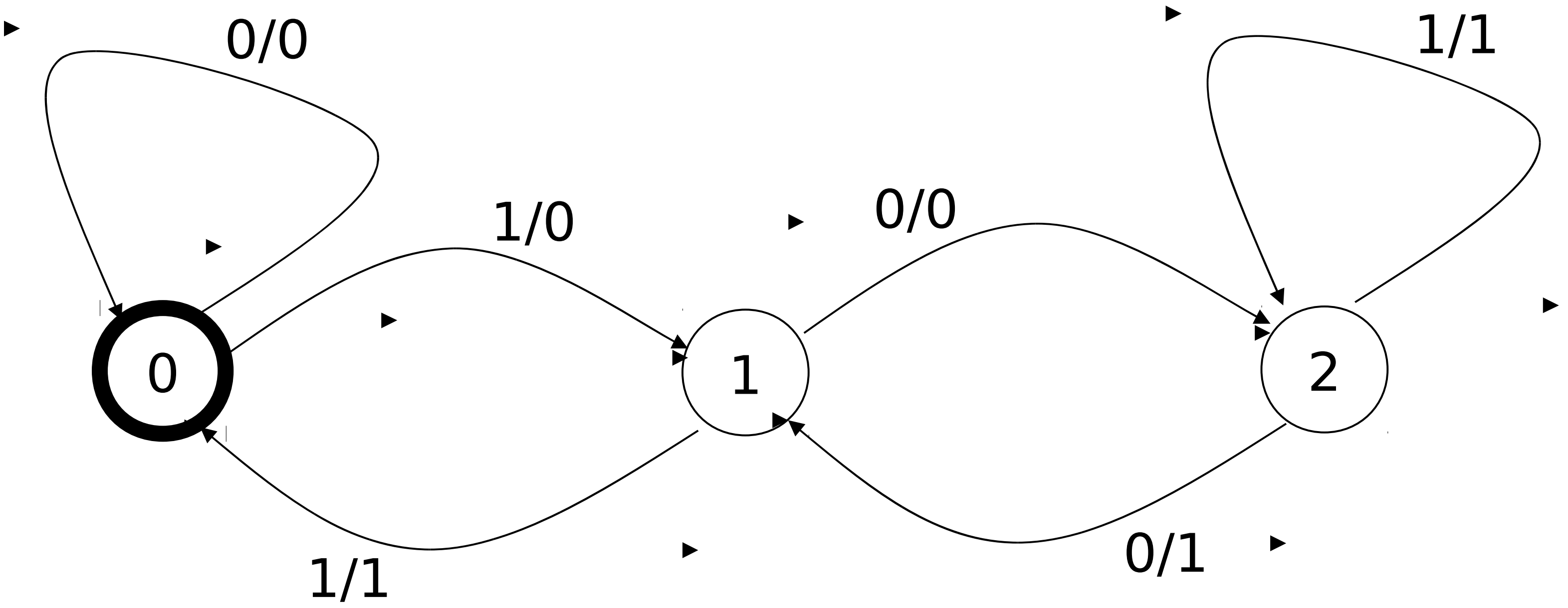
Most of the ambiguities have been solved

Finite State Transducers (FST)

$<\Sigma_1, \Sigma_2, Q, i, F, E>$

Σ_1	input alphabet
Σ_2	output alphabet
frequently $\Sigma_1 = \Sigma_2 = \Sigma$	
Q	**finite** states set
$i \in Q$	initial state
$F \subseteq Q$	final states set
$E \subseteq Q \times (\Sigma_1^* \times \Sigma_2^*) \times Q$	arcs set

Example 3

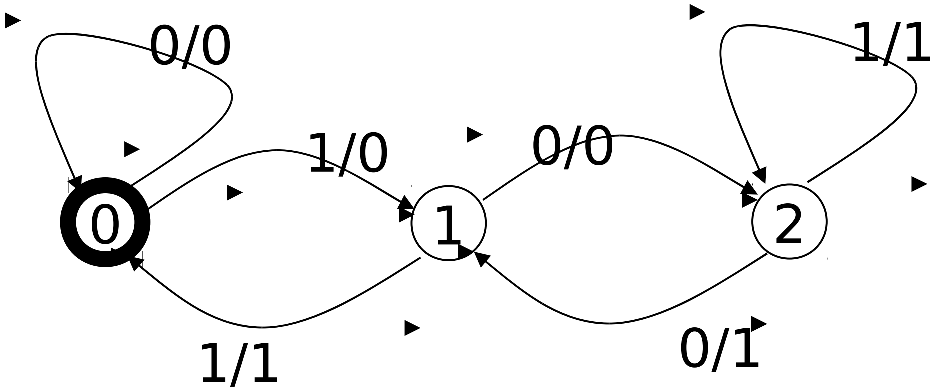


Td3: division by 3 of a binary string

$\Sigma_1 = \Sigma_2 = \Sigma = \{0,1\}$

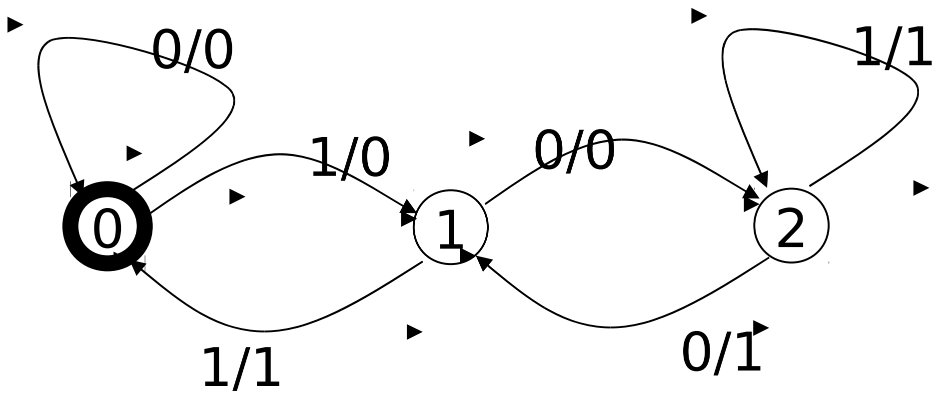
Example 3

input	output
0	0
11	01
110	010
1001	0011
1100	0100
1111	0101
10010	00110



Td3: division by 3 of a binary string

$\Sigma_1 = \Sigma_2 = \Sigma = \{0,1\}$



State 0:

Recognized: 3k

Emited: k

State 1:

Recognized : 3k+1

Emited : k

State 2:

Recognized : 3k+2

Emited : k

invariant:

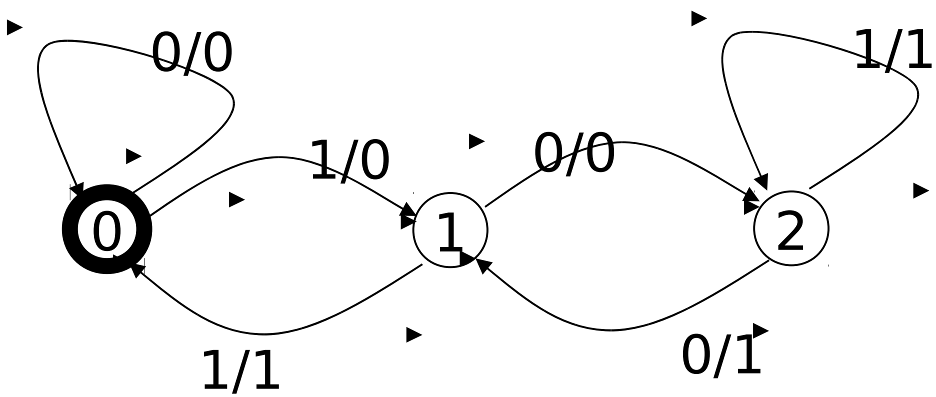
emited * 3 = Recognized

invariant:

emited * 3 + 1 = Recognized

invariant:

emited * 3 + 2 = Recognized



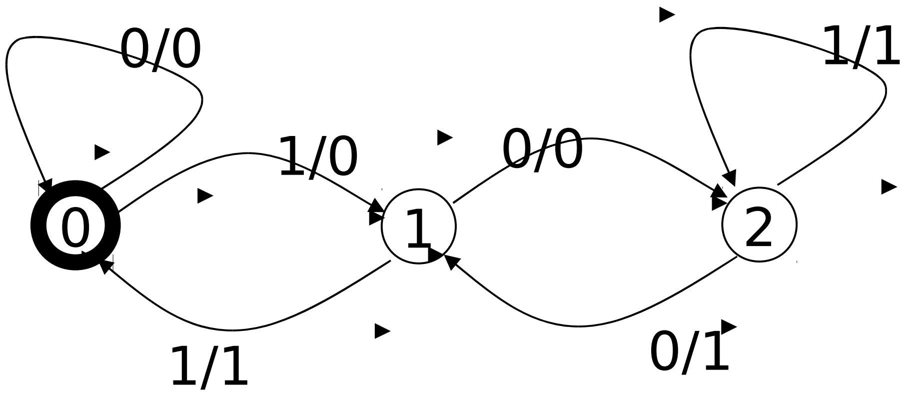
state 0:	
Recognized:	3k
Emited:	k

consums:	0
emits:	0
recognized:	3*k*2 = 6k
emited:	k*2 = 2k

State 0
satisfies invariant

consums:	1
emits:	0
recognized:	3*k*2 + 1= 6k + 1
emited:	k*2 = 2k

State 1
satisfies invariant



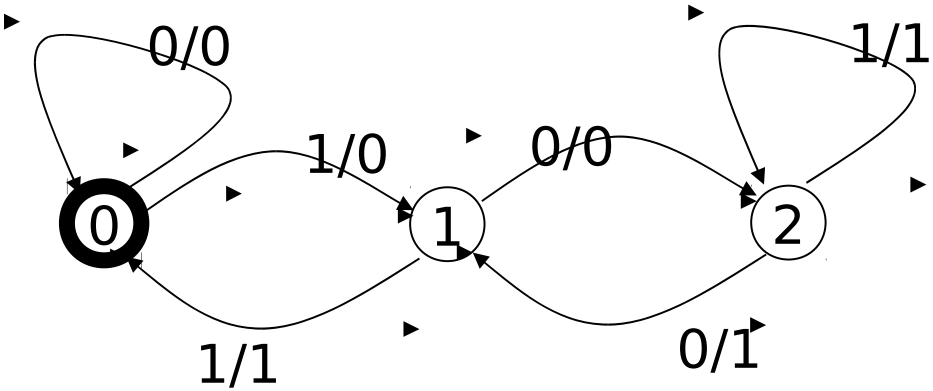
state 1:	
recognized:	3k+1
emited:	k

consums:	0
emits:	0
recognized:	(3k+1)*2 = 6k + 2
Emited:	k*2 = 2k

State 2
satisfies invariant

consums:	1
emits:	1
recognized:	(3k+1)*2 + 1= 6k + 3
emited:	k*2 + 1 = 2k + 1

State 0
satisfies invariant



state 2:	
recognized:	3k+2
emited:	k

consums:	0
emits:	1
recognized:	(3k+2)*2 = 6k + 4
emited:	k*2 + 1 = 2k + 1

State 1
satisfies invariant

consums:	1
emits:	1
recognized:	(3k+2)*2 + 1= 6k + 5
emited:	k*2 + 1 = 2k + 1

State 2
satisfies invariant

FSA associated with a FST

FST $<\Sigma_1, \Sigma_2, Q, i, F, E>$

FSA $<\Sigma, Q, i, F, E'>$

$\Sigma = \Sigma_1 \times \Sigma_2$

$(q_1, (a,b), q_2) \in E' \Leftrightarrow (q_1, a, b, q_2) \in E$

Application of a FST

Traverse the FST in all forms compatible with the input (using backtracking if needed) until reaching a final state and generate the corresponding output

Consider input as a FSA and compute the intersection of the FSA and the FST

Aplications of FSA(and FST)

- Increasing use in NLP
  - Morphology
  - Phonology
  - Lexical generation
  - ASR (Automatic Speech Recognition)
  - POS tagging
  - Simplification of Grammars
  - Information Extraction

- Why FSA (and FST)?
  - Temporal and spatial efficiency
  - Some FSA can be determined and optimized for leading to more compact representations
  - Possibility to be used in cascade form

Determinization of a FST

Not all FST are determinizable, if it is the case they are named **subsequential**

The <u>non deterministic FST</u> is equivalent to the <u>deterministic one</u>