Finite State Automata and Search

Doing NLP with FST

Trevor Sullivan

October 27, 2016

University of Arizona

Table of contents

Introduction to FSM

Nomenclature

Nomenclature

Finite State Machine $\iff$ **Finite State Automaton**

Nomenclature

Finite State Machine $\iff$ **Finite State Automaton**

Two types of FSM:

Nomenclature

Finite State Machine $\iff$ **Finite State Automaton**

Two types of FSM:

1. **Finite State Acceptor** (usually what is meant by Finite State Automaton)

Nomenclature

Finite State Machine $\iff$ **Finite State Automaton**

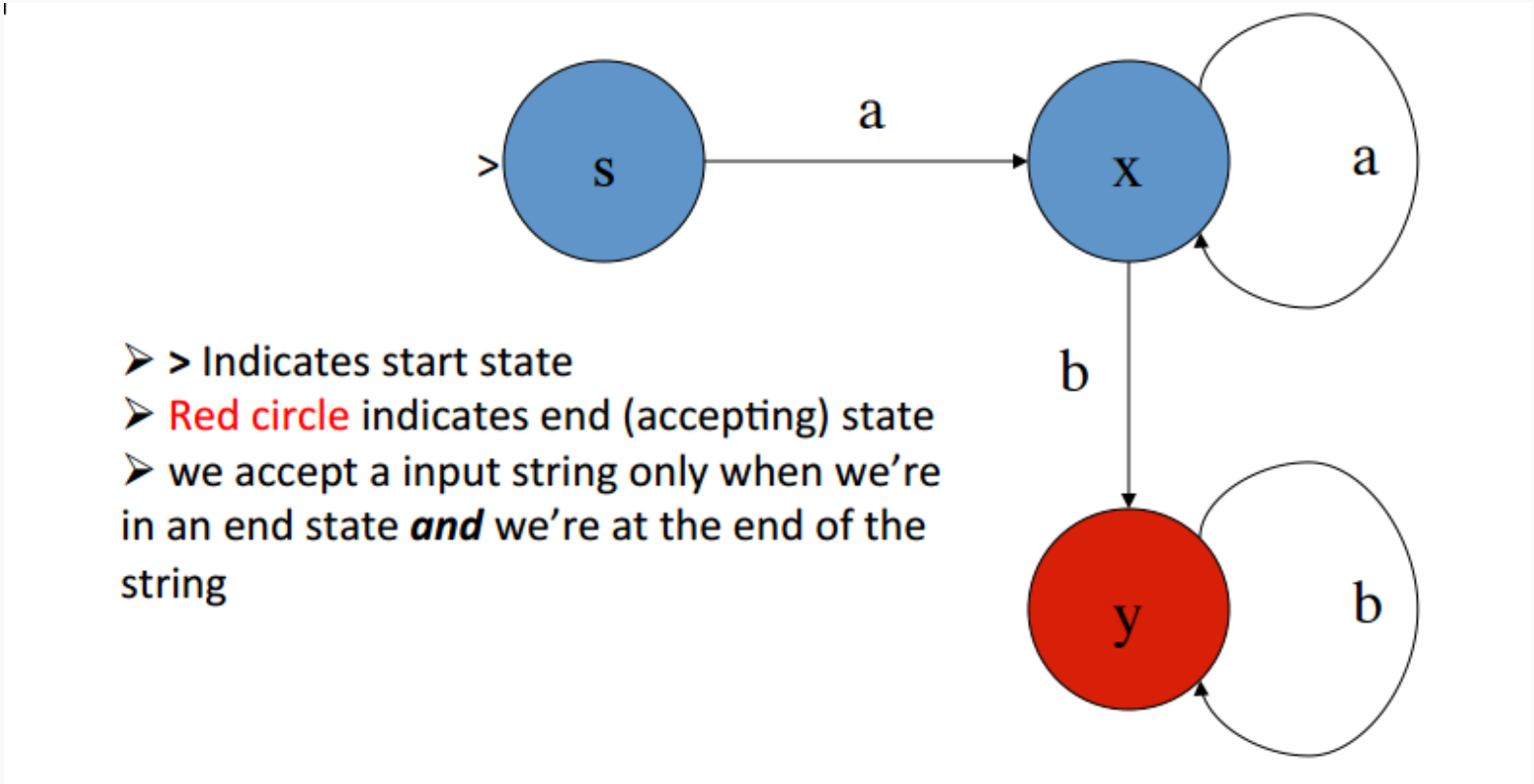
Two types of FSM:

1. **Finite State Acceptor** (usually what is meant by Finite State Automaton)
2. **Finite State Transducer**

What is a FSM?

What is a FSM?

A directed graph



- > Indicates start state
- Red circle indicates end (accepting) state
- we accept a input string only when we're in an end state ***and*** we're at the end of the string

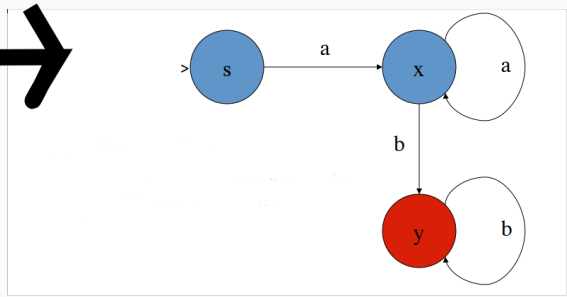
$L = \{a^+ b^+\}$

What is a FSM?

REGEX

{a+b+}

FSA

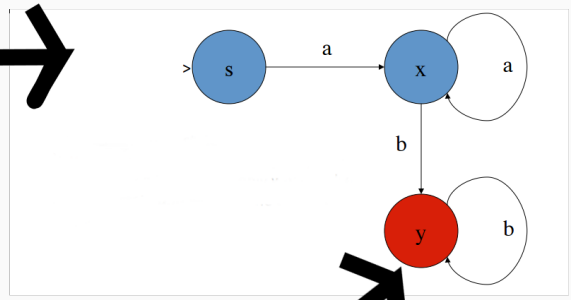


What is a FSM?

REGEX

{a+b+}

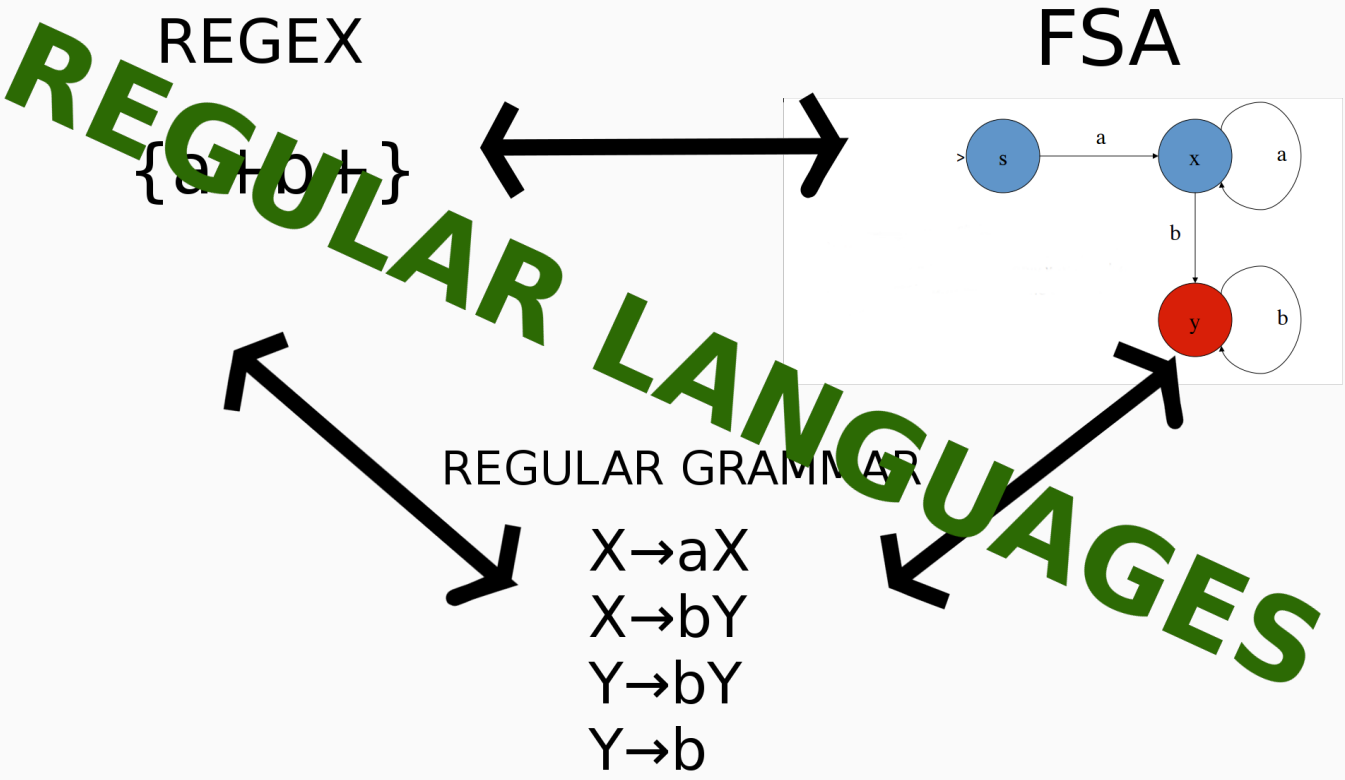
FSA



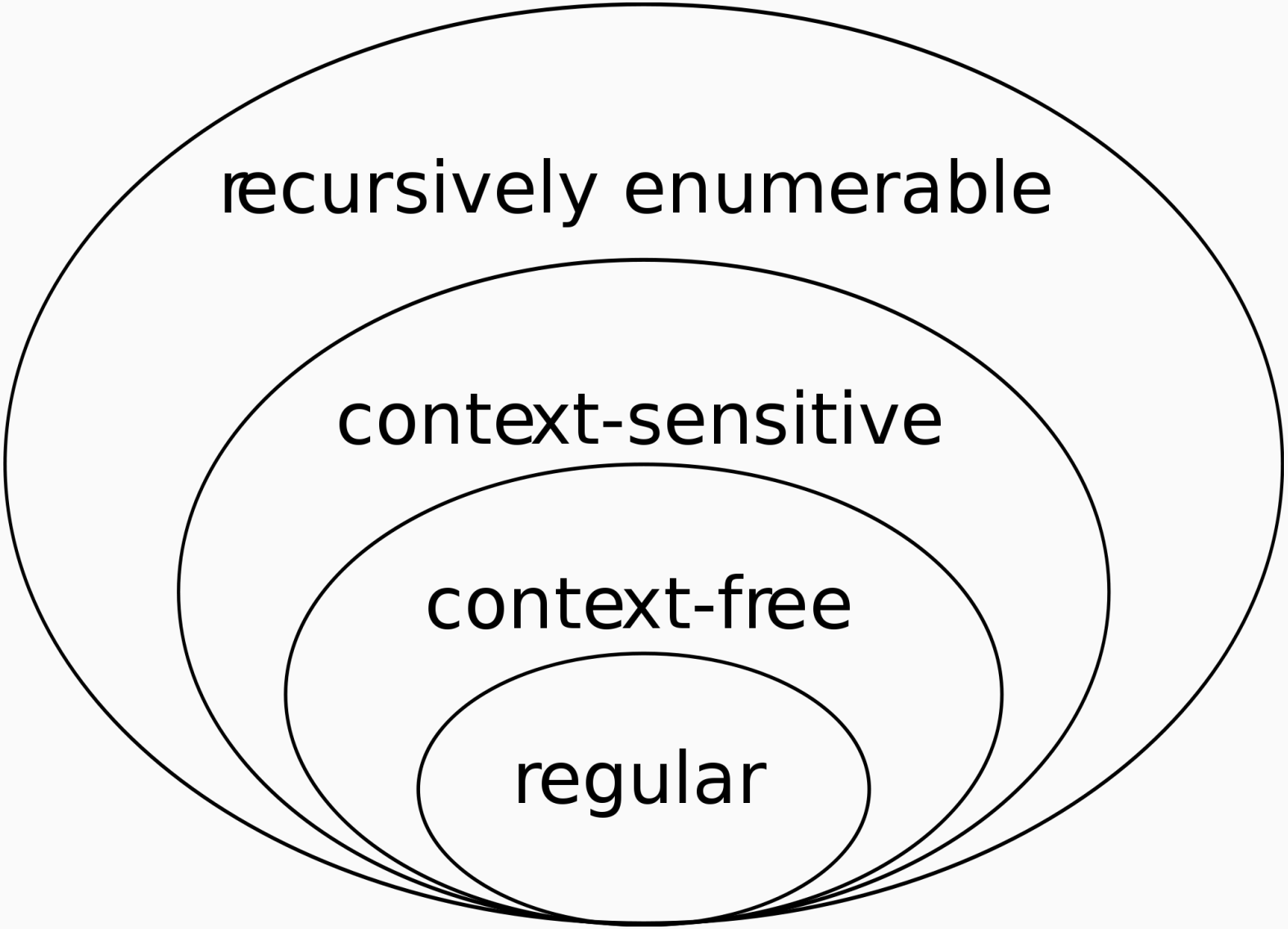
REGULAR GRAMMAR

X→aX
X→bY
Y→bY
Y→b

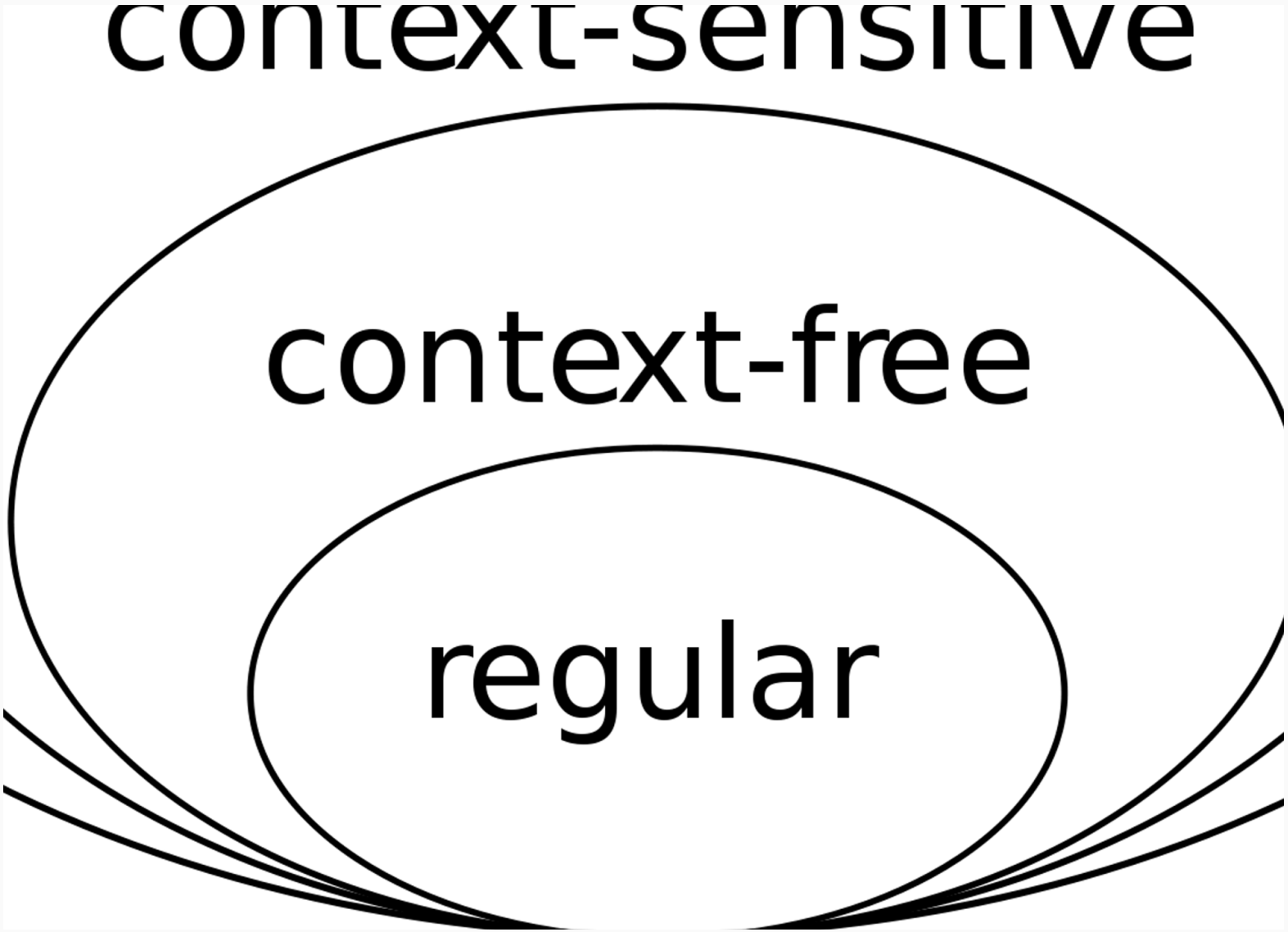
What is a FSM?


REGEX
{a+b+}
FSA
s
x
y
a
a
b
b
REGULAR GRAMMAR
X→aX
X→bY
Y→bY
Y→b
REGULAR LANGUAGES

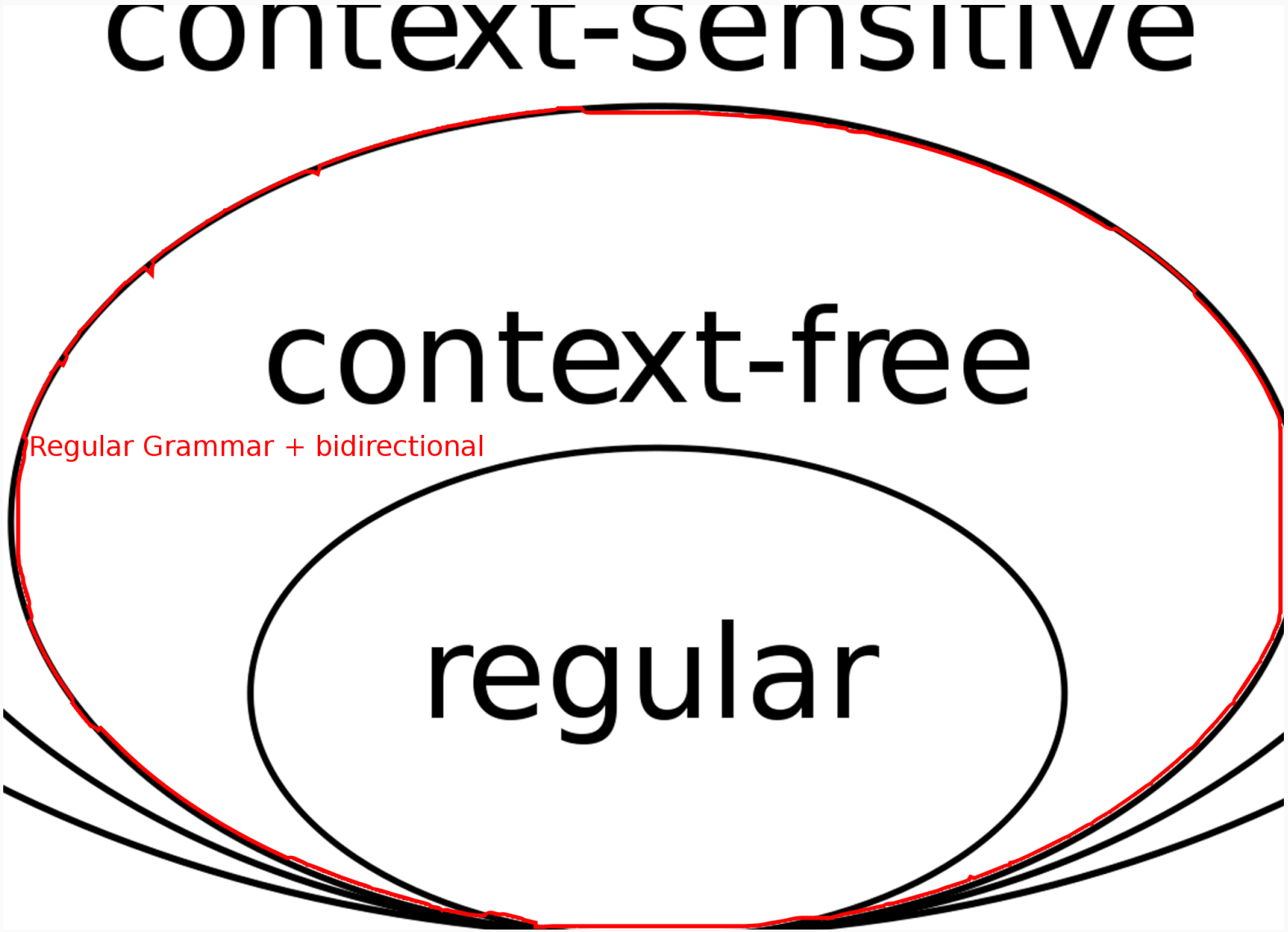
Chomsky Hierarchy



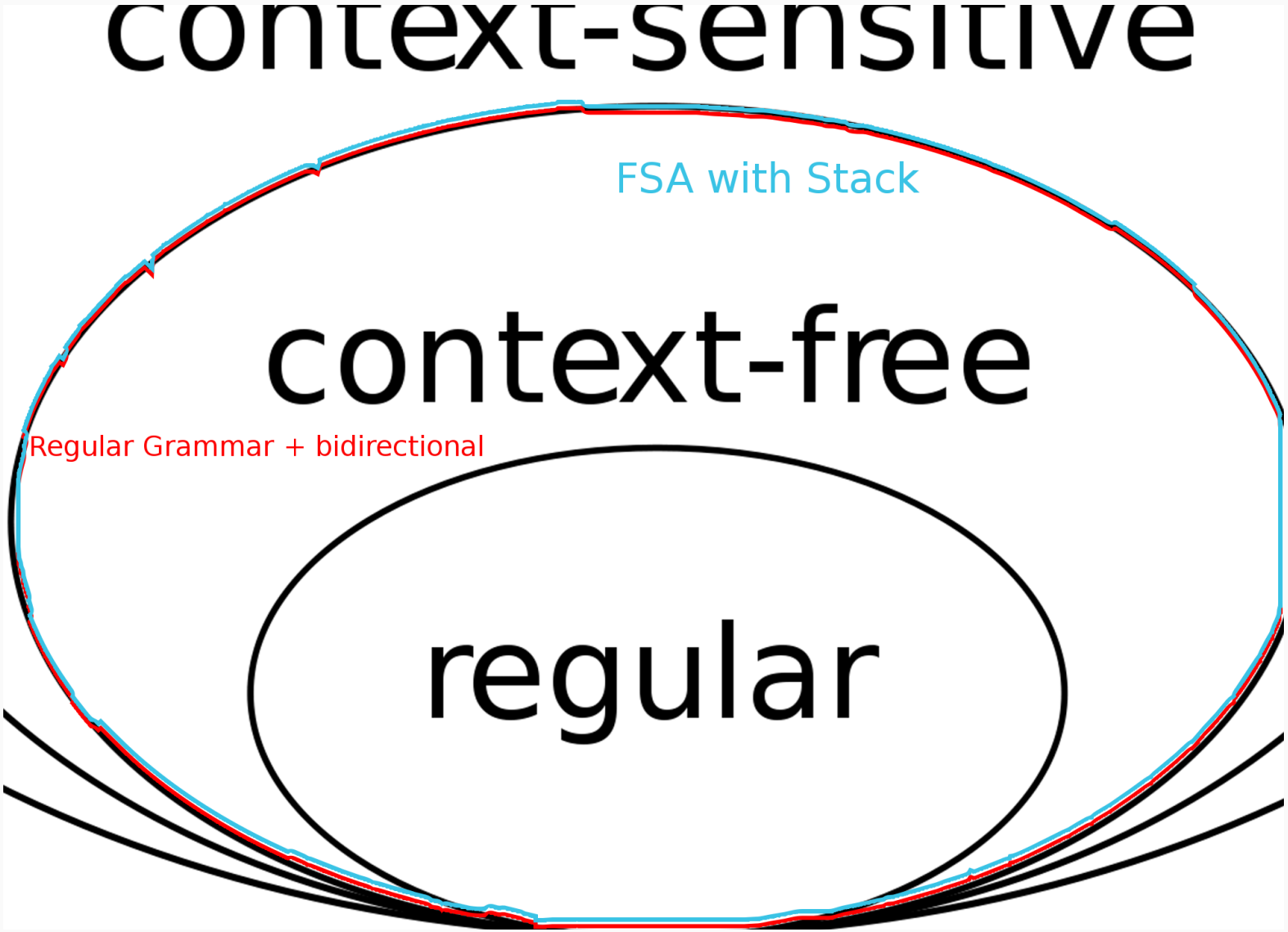
Chomsky Hierarchy



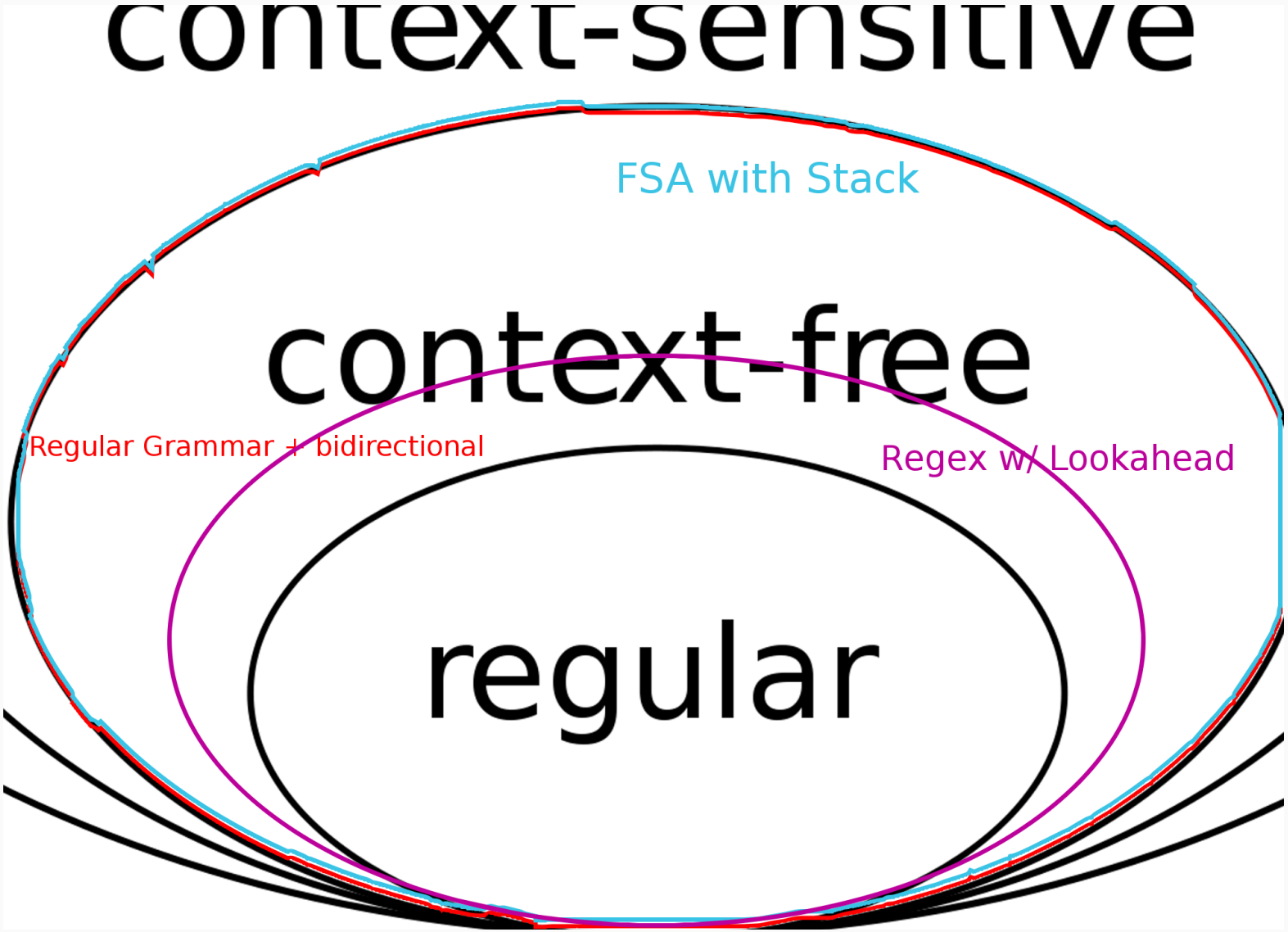
Chomsky Hierarchy



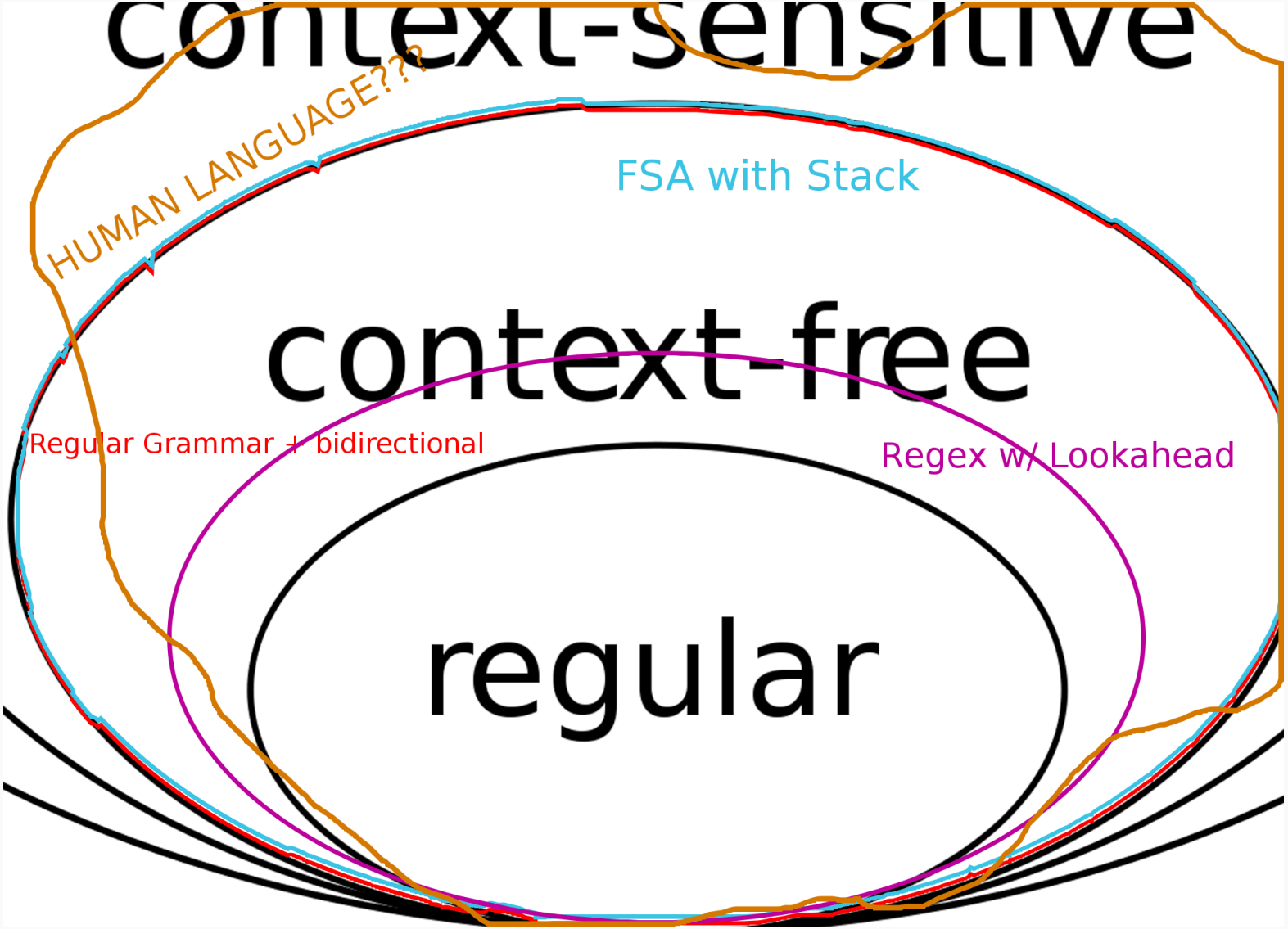
Chomsky Hierarchy



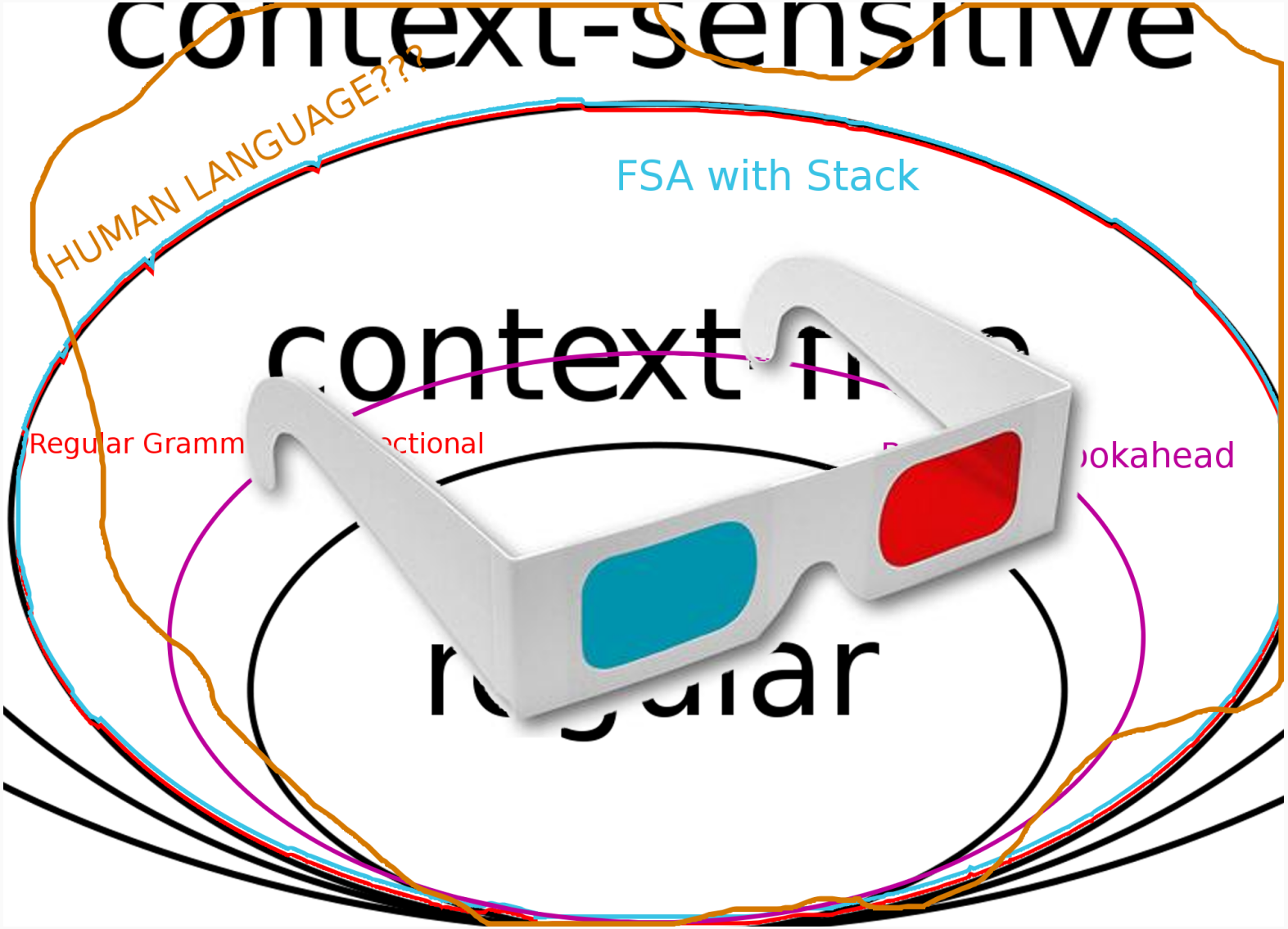
Chomsky Hierarchy



Chomsky Hierarchy



Chomsky Hierarchy

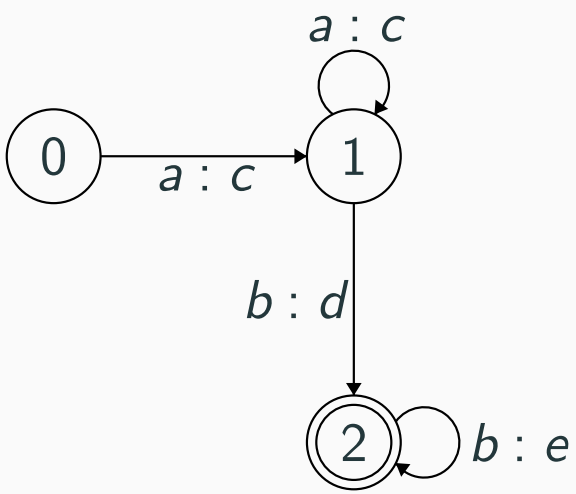


FST

A finite state machine that turns one regular language into another regular language.

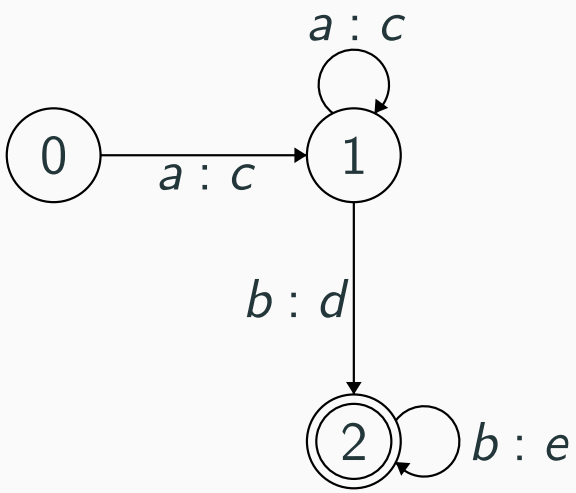
FST

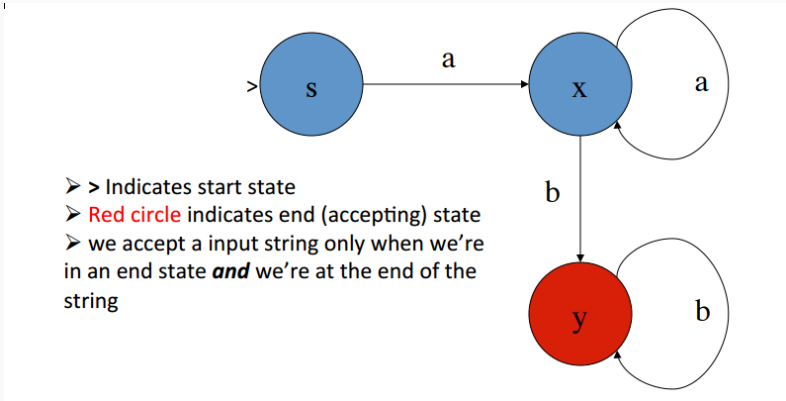
A finite state machine that turns one regular language into another regular language.



FST

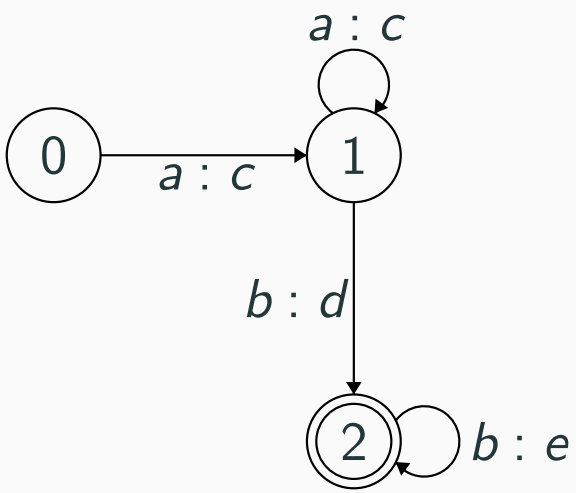
A finite state machine that turns one regular language into another regular language.

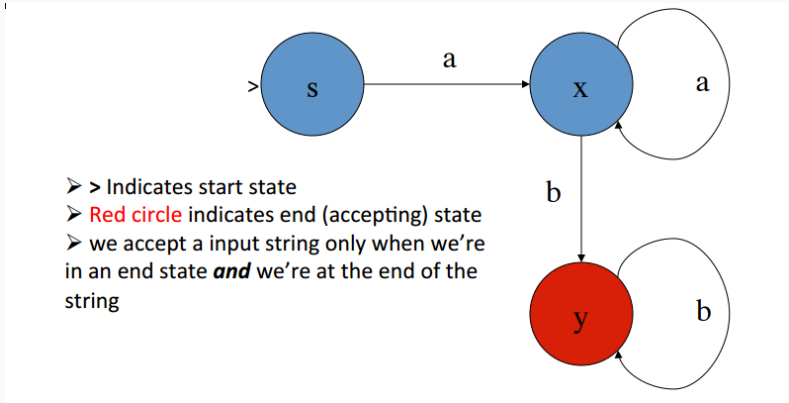




FST

A finite state machine that turns one regular language into another regular language.





FST that transforms $\{a + b+\}$ to $\{c + de*\}$

Formal Definition

Formal Definition

$\langle \Sigma, S, s_0, \delta, F \rangle$

Σ alphabet

S states

$s_0 \in S$ start state

δ transition function
$\delta : S, \Sigma \to S$
(Can be partial or total)

$\mathbf{F} \subset S$ final states
(can be $\emptyset$)

Uses in Computational Linguistics

Morphology!

There are dictionaries of derived forms of words, but these can never be fully complete.

Cortois and Laporte (1991) demonstrated that most terms found missing from LADL[1]s DELAS[2] by comparing against natural language text are derived forms that were missed.

[1] Labratorie d'Automatique Documentaire et Linguistique
[2] Dictionairie Electronique du LADL S

SDICOS, the morphological transducer

Problem:

Present in DICOS	Missing from DICOS
activier	activable
	activabilite
	suractivation
	suractivable

Attempts to add these to the dictionary inevitably miss many and dramatically increase the dictionary size and reduce search speed without much benefit.

Solution: FST

Rather than storing a word list, make a transducer that changes an input word to output dictionary information.

Solution: FST

- inculperont $\Longleftrightarrow$ inculper, V+f3p
- aimerant $\Longleftrightarrow$ aimer, V+f3p
- chevaux $\Longleftrightarrow$ cheval, N+mp
- carnavals $\Longleftrightarrow$ carnaval, N+mp
- pommes $\Longleftrightarrow$ pomme, N+fp
- pommes $\Longleftrightarrow$ pommer, V+P2s
- pommes $\Longleftrightarrow$ pommer, V+S2s

Solution: FST

- inculperont $\Longleftrightarrow$ inculper, V+f3p
- aimerant $\Longleftrightarrow$ aimer, V+f3p
- chevaux $\Longleftrightarrow$ cheval, N+mp
- carnavals $\Longleftrightarrow$ carnaval, N+mp
- pommes $\Longleftrightarrow$ pomme, N+fp
- pommes $\Longleftrightarrow$ pommer, V+P2s
- pommes $\Longleftrightarrow$ pommer, V+S2s

(disambiguation problem: we don't care)

SDICOF

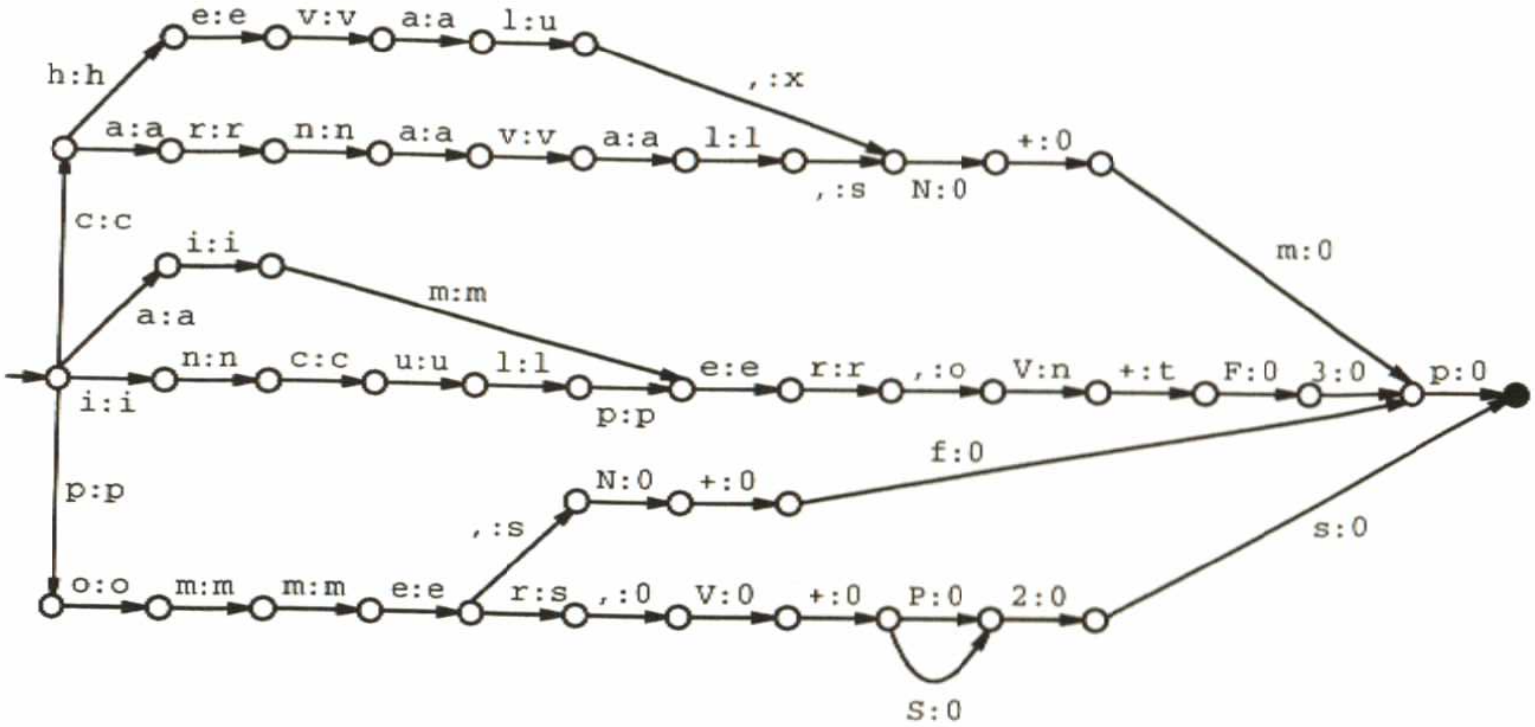


Figure 2.4: A sample of the DICOF transducer

Performance

	NUMBER OF ENTRIES	SIZE (Ko)	SPEED (words/second)
DICOF TRANSDUCER	612,848	948	1,100
NEW DICOF TRANSDUCER	1,301,976	1,014	1,100

Issues

Not very convenient because you cannot link a derivitive in a text to the root of it's derivational tree.

We need a system that given a derivitive returns the trees that contain it.

New FST

- C1 + activier + Vable.32RA/S $\Longleftrightarrow$ coactivable
- I3 + C1 + activier + Vable.32RA/S $\Longleftrightarrow$ incoactivable
- O1 + activier + Vable.32RA/S $\Longleftrightarrow$ coactivable
- I3 + O1 + activier + Vable32RA/S $\Longleftrightarrow$ incoactivable

New FST

- C1 + activier + Vable.32RA/S $\iff$ coactivable
- I3 + C1 + activier + Vable.32RA/S $\iff$ incoactivable
- O1 + activier + Vable.32RA/S $\iff$ coactivable
- I3 + O1 + activier + Vable32RA/S $\iff$ incoactivable

(C = symmetry of subjects, O = symmetry of objects, eg

"IBM and Microsoft codeveloped OS/2" vs "The alarm and the lock are always coactivated")

SDICOS

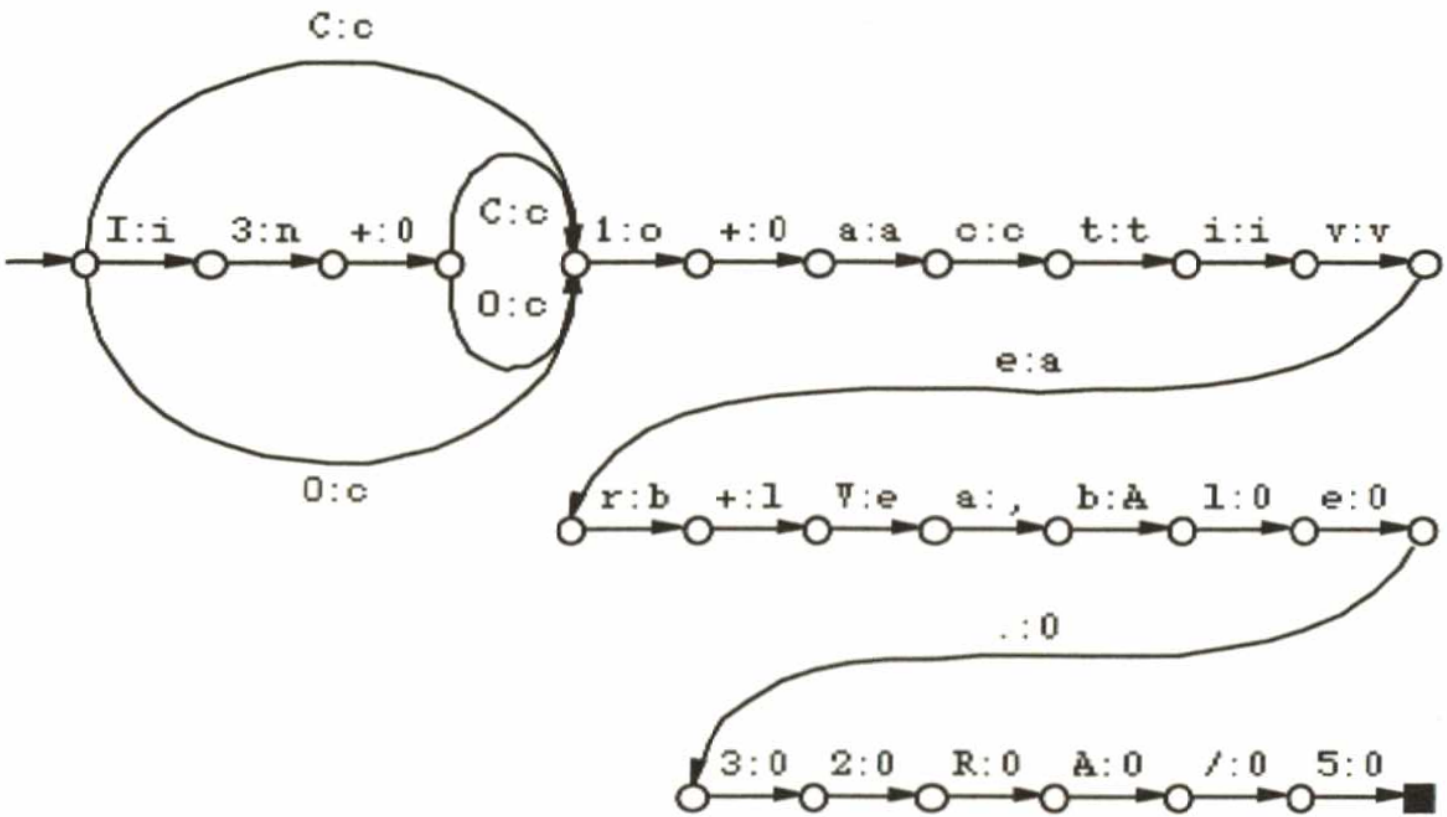


Figure 2.5: SDICOS, the Structured DICOS transducer

SDICOS

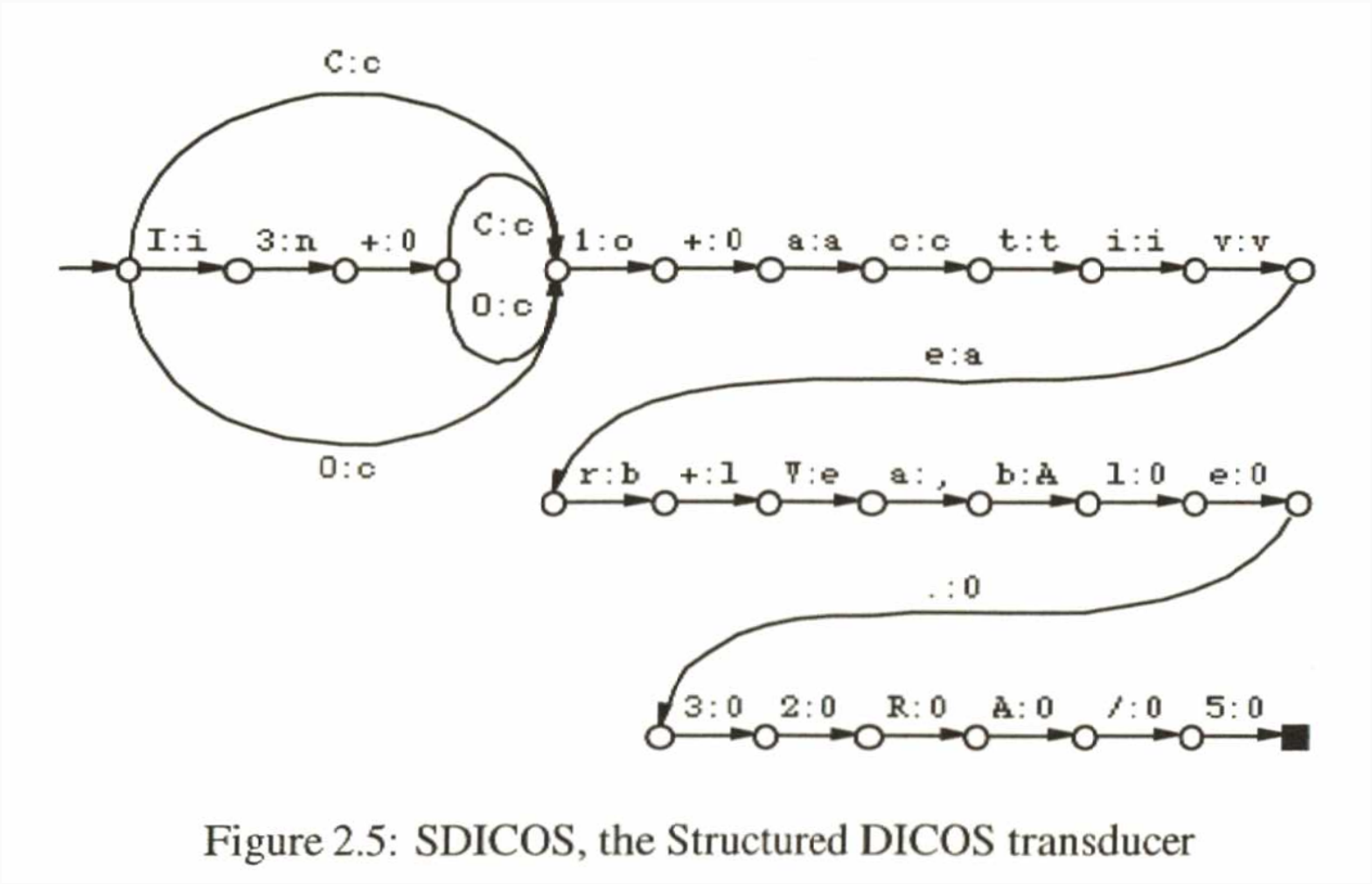


Figure 2.5: SDICOS, the Structured DICOS transducer

Not as fast (519 w/s), significantly smaller memory imprint (97Kb)

More Generic

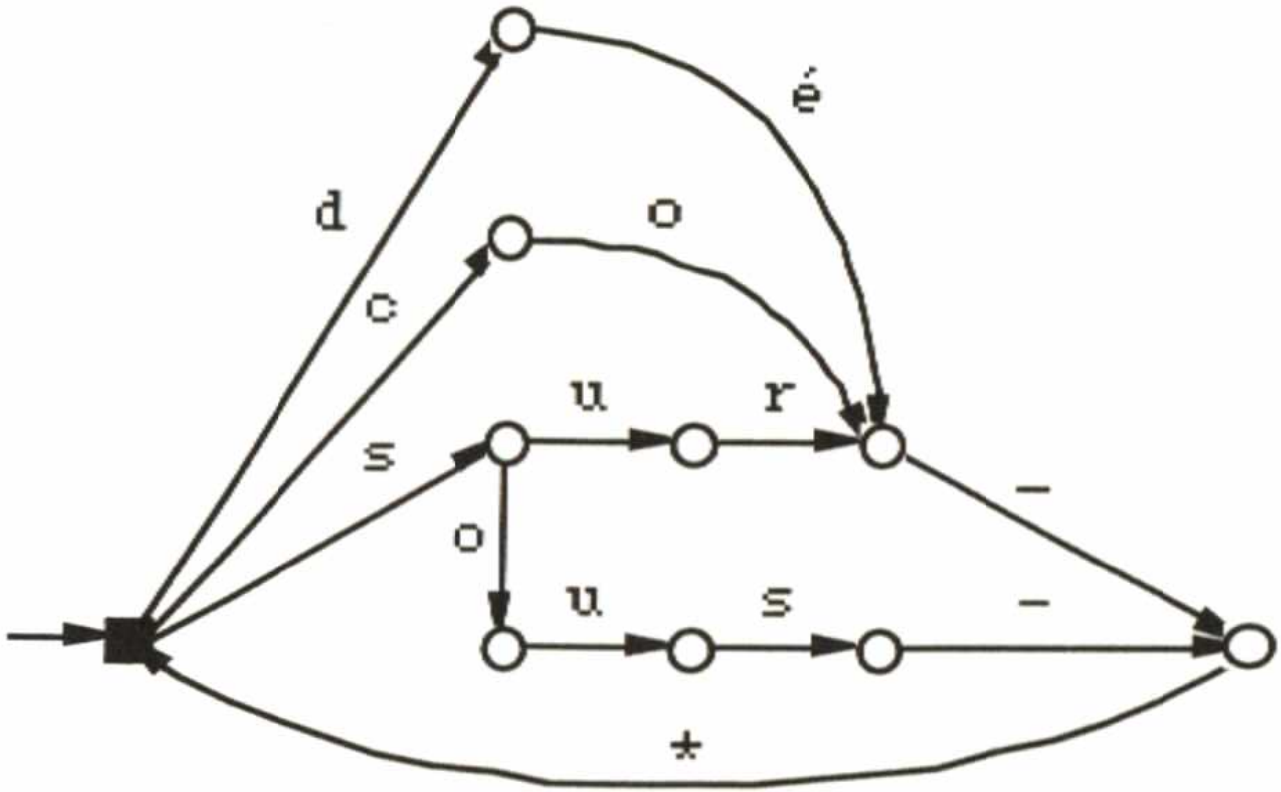


Figure 2.6: A sample of PREF(A)

More Generic

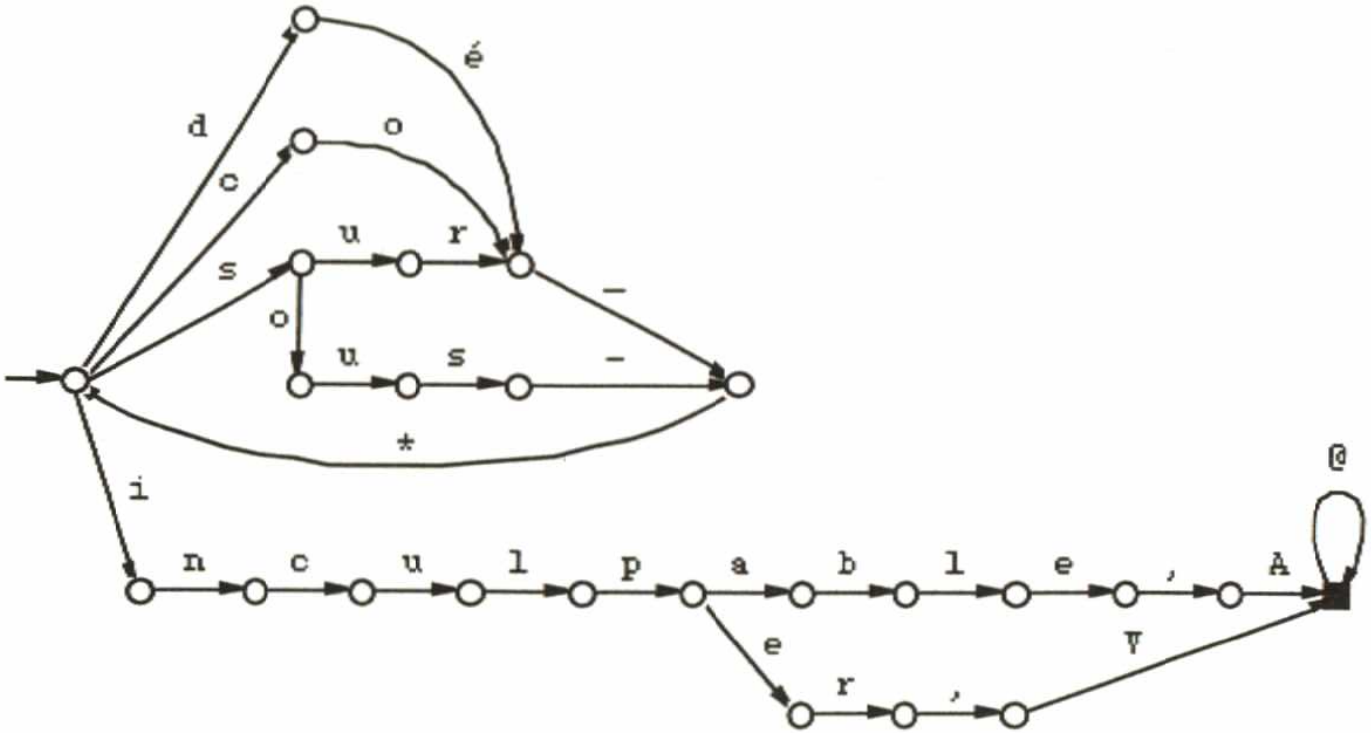


Figure 2.7: A sample of PREF(A) • DICOS(A)

More Generic

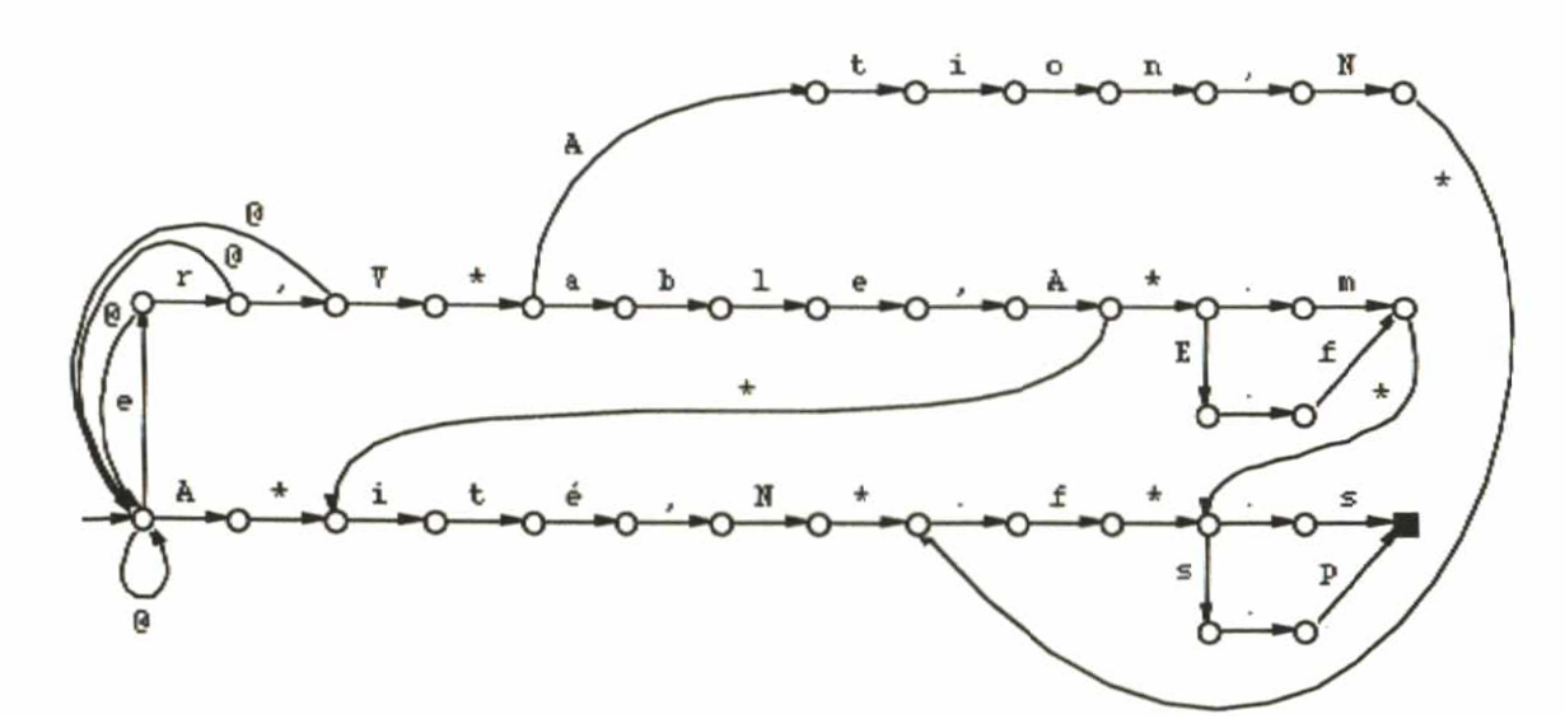


Figure 2.8: A sample of SUF(A)

More Generic

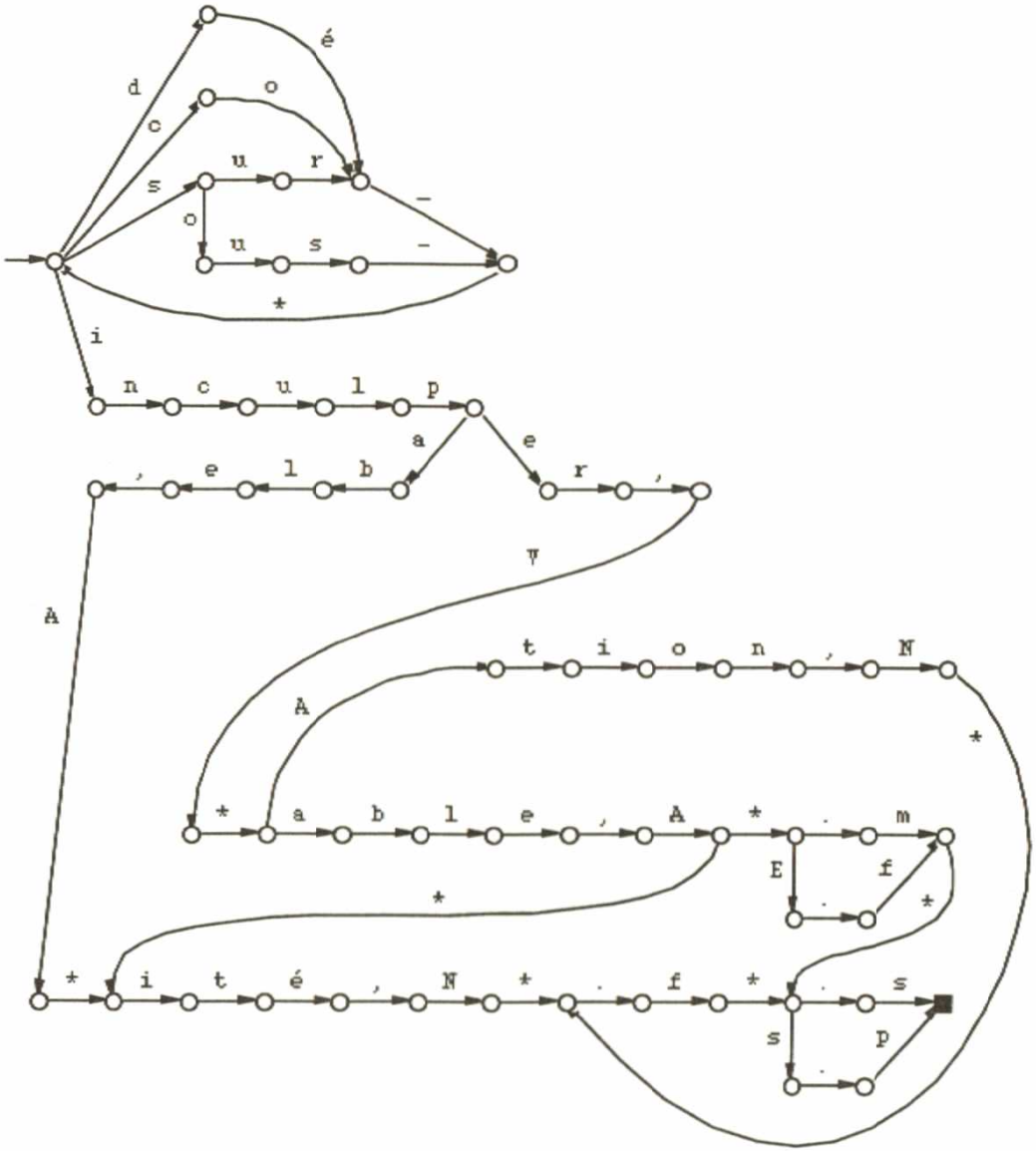


Figure 2.9: A sample of (PREF(A) • DICOS(A)) ∩ SUF(A)

Parsing?!

CFGs are generally considered the best descriptor of natural languages

Parsing?!

CFGs are generally considered the best descriptor of natural languages but they're big and slow. FSMs can help alleviate that problem.

Parsing?!

CFGs are generally considered the best descriptor of natural languages

but they're big and slow. FSMs can help alleviate that problem.

The Earley algorithm converts a deterministic CFG into a FSA by using dot notation (LR(0), LR(1))

This technique involves using FSMs to model grammars by combining many grammar rules into finite states.

Advantages

- Compact

Reconceptualize parsing as a string transformation.

[3] Roche 1993

Advantages

- Compact
- Determinizable

Reconceptualize parsing as a string transformation.

[3] Roche 1993

Advantages

- Compact
- Determinizable
- Symmetrical: Parsing and Production are the same process

Reconceptualize parsing as a string transformation.

[3] Roche 1993

Advantages

- Compact
- Determinizable
- Symmetrical: Parsing and Production are the same process
- CFG $\rightarrow$ FST is a transduction[3]

Reconceptualize parsing as a string transformation.

[3] Roche 1993

Advantages

- Compact
- Determinizable
- Symmetrical: Parsing and Production are the same process
- CFG $\rightarrow$ FST is a transduction[3]

Reconceptualize parsing as a string transformation.

[3] Roche 1993

Advantages

- Compact
- Determinizable
- Symmetrical: Parsing and Production are the same process
- CFG $\rightarrow$ FST is a transduction[3]

Reconceptualize parsing as a string transformation.

Parser: $\Sigma_w^* \implies 2^{(\Sigma_g^* \cdot \Sigma_w^* \cdot \Sigma_g^*)}$

"Bob left this morning" $\iff$ "(S(N Bob N)(V left (N this morning N)V)S)"

[3] Roche 1993

CFG

N *thinks that* S	S
N *kept* N	S
John	N
Peter	N
the book	N

Table 8.1: syntactic dictionary for *John thinks that Peter kept the book*

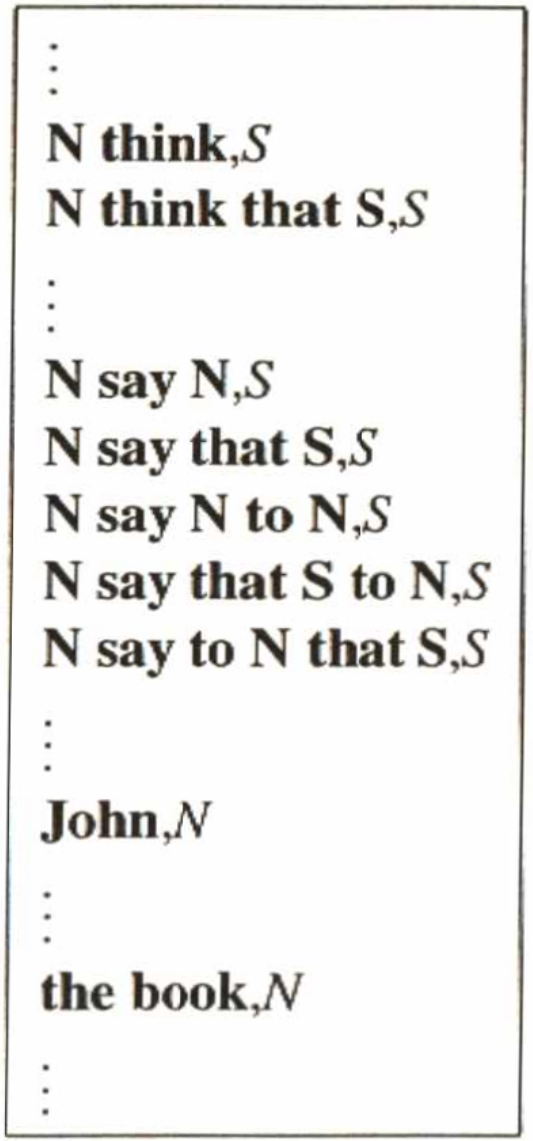


Figure 8.1: Sample of the Syntactic Dictionary

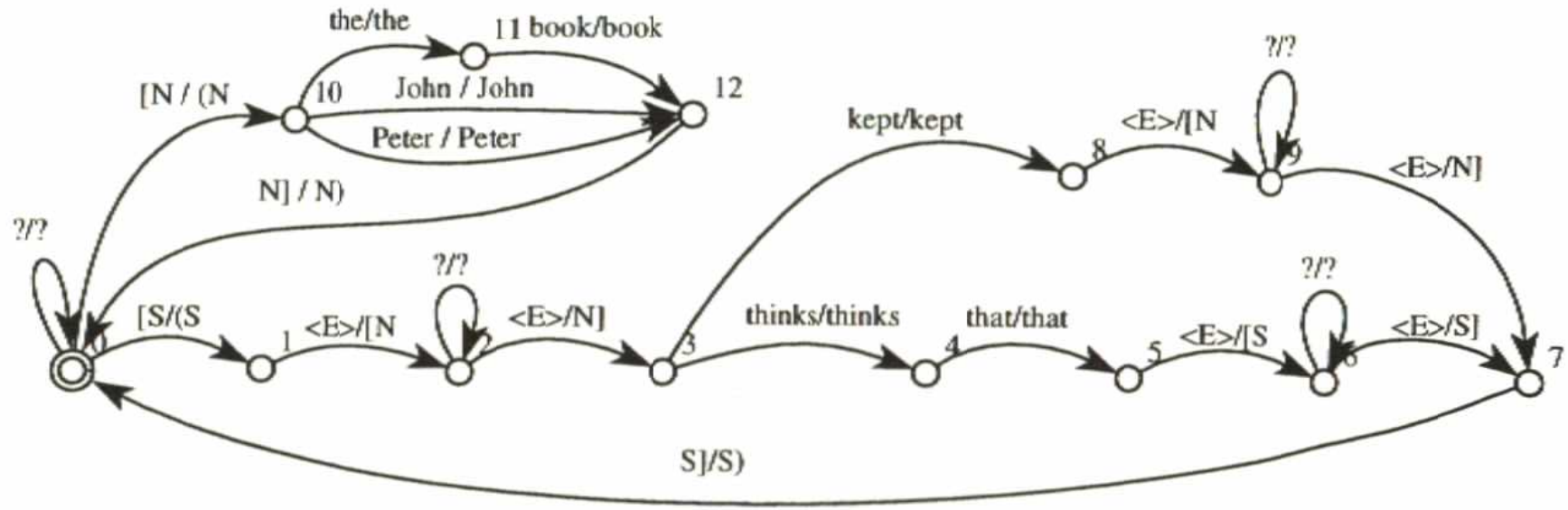


Figure 8.2: Transducer T_{dic_1} representing the syntactic dictionary dic_1

IFSP

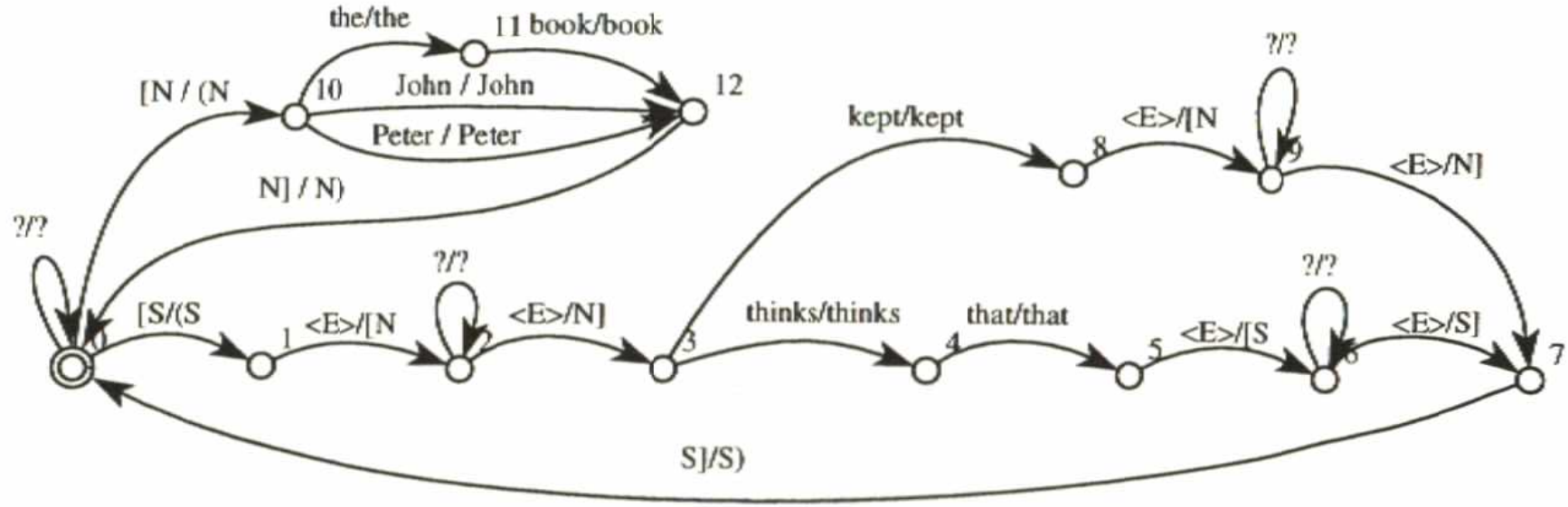


Figure 8.2: Transducer T_{dic_1} representing the syntactic dictionary dic_1

[S Mike thinks that Robert kept the book S]

IFSP

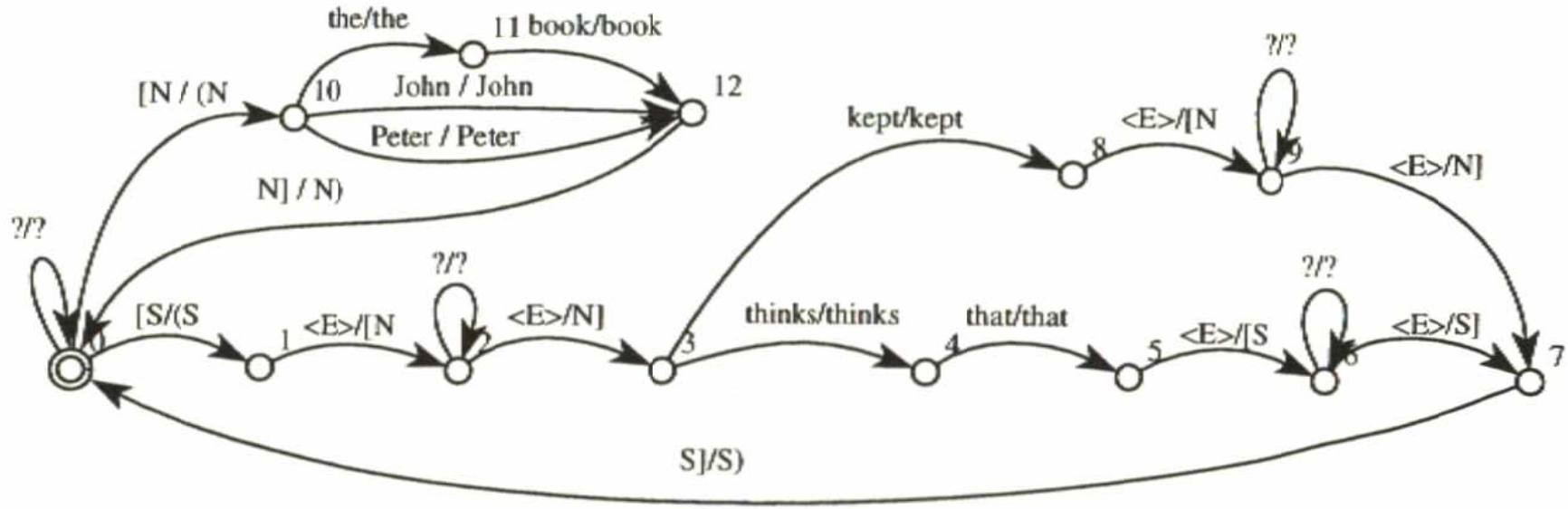


Figure 8.2: Transducer T_{dic_1} representing the syntactic dictionary dic_1

(S [N Mike thinks that Robert N] kept [S the book S]S)

(S [N Mike N] thinks that [S Robert kept the book S]S)

IFSP

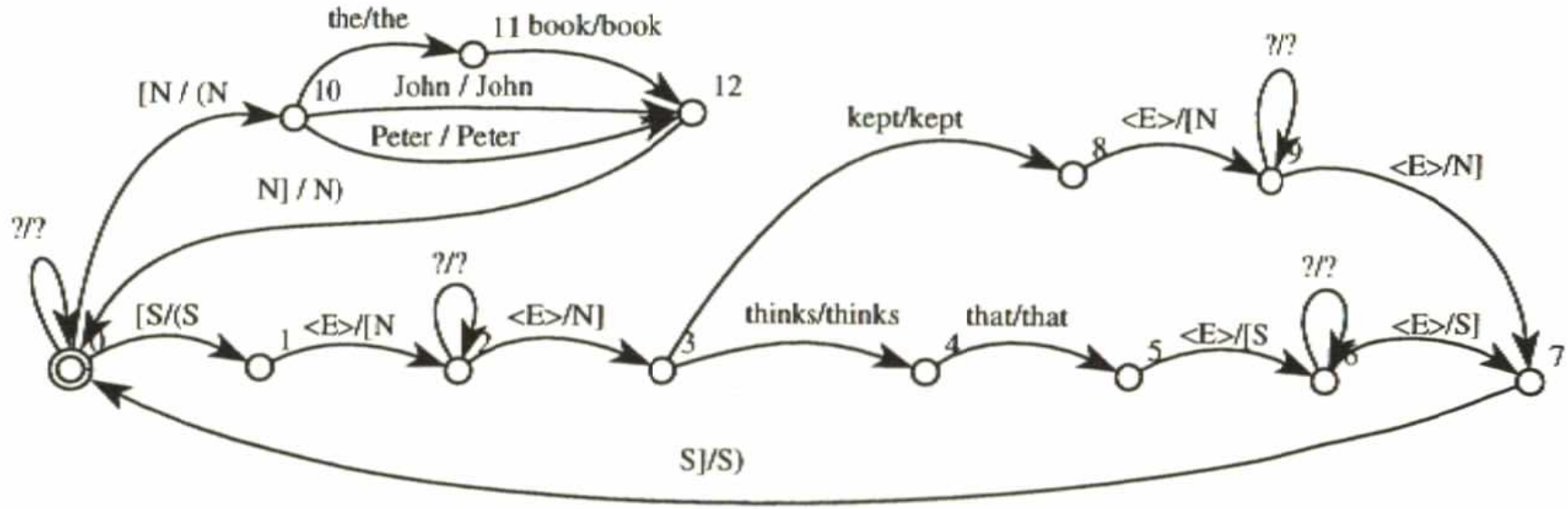


Figure 8.2: Transducer T_{dic_1} representing the syntactic dictionary dic_1

~~(S [N Mike thinks that Robert N] kept [S the book S]S)~~

(S [N Mike N] thinks that [S Robert kept the book S]S)

IFSP

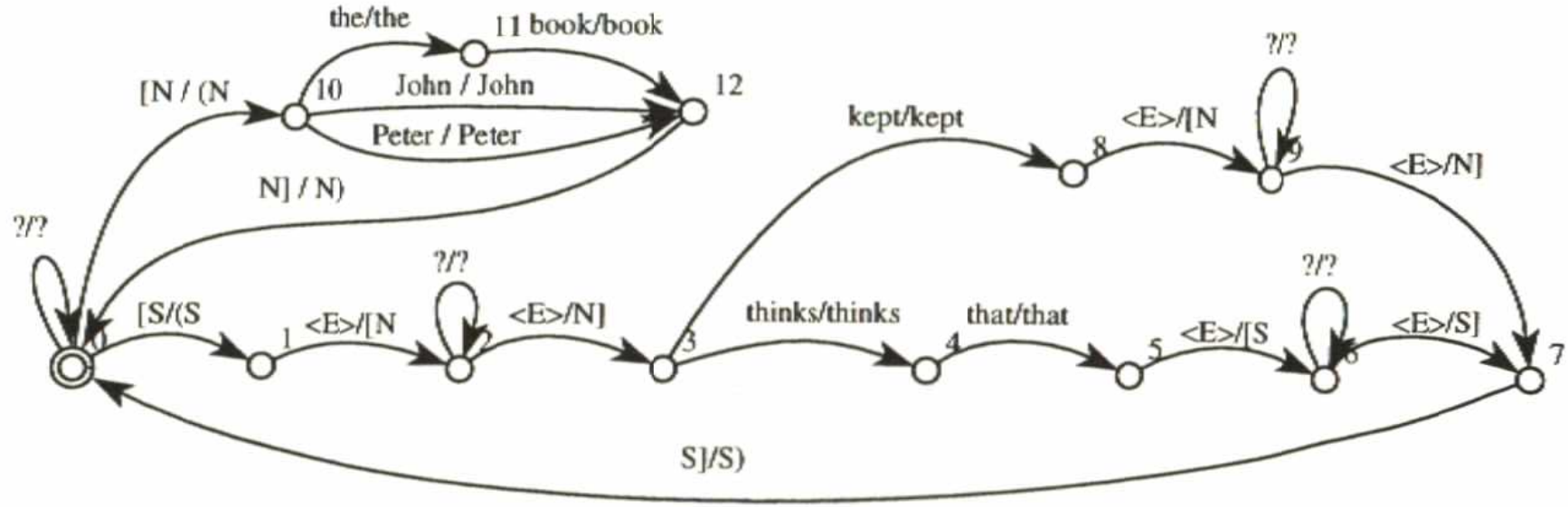


Figure 8.2: Transducer T_{dic_1} representing the syntactic dictionary dic_1

(S (N Mike N) thinks that (S [N Robert N] kept [N the book N]S)S)

IFSP

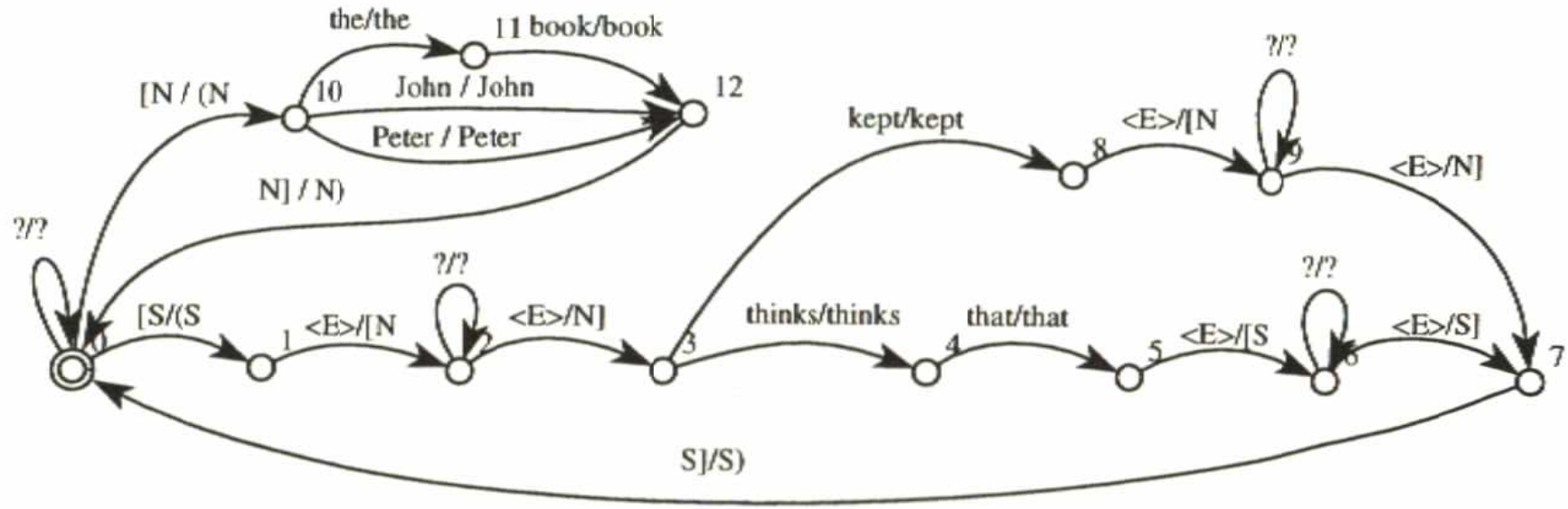


Figure 8.2: Transducer T_{dic_1} representing the syntactic dictionary dic_1

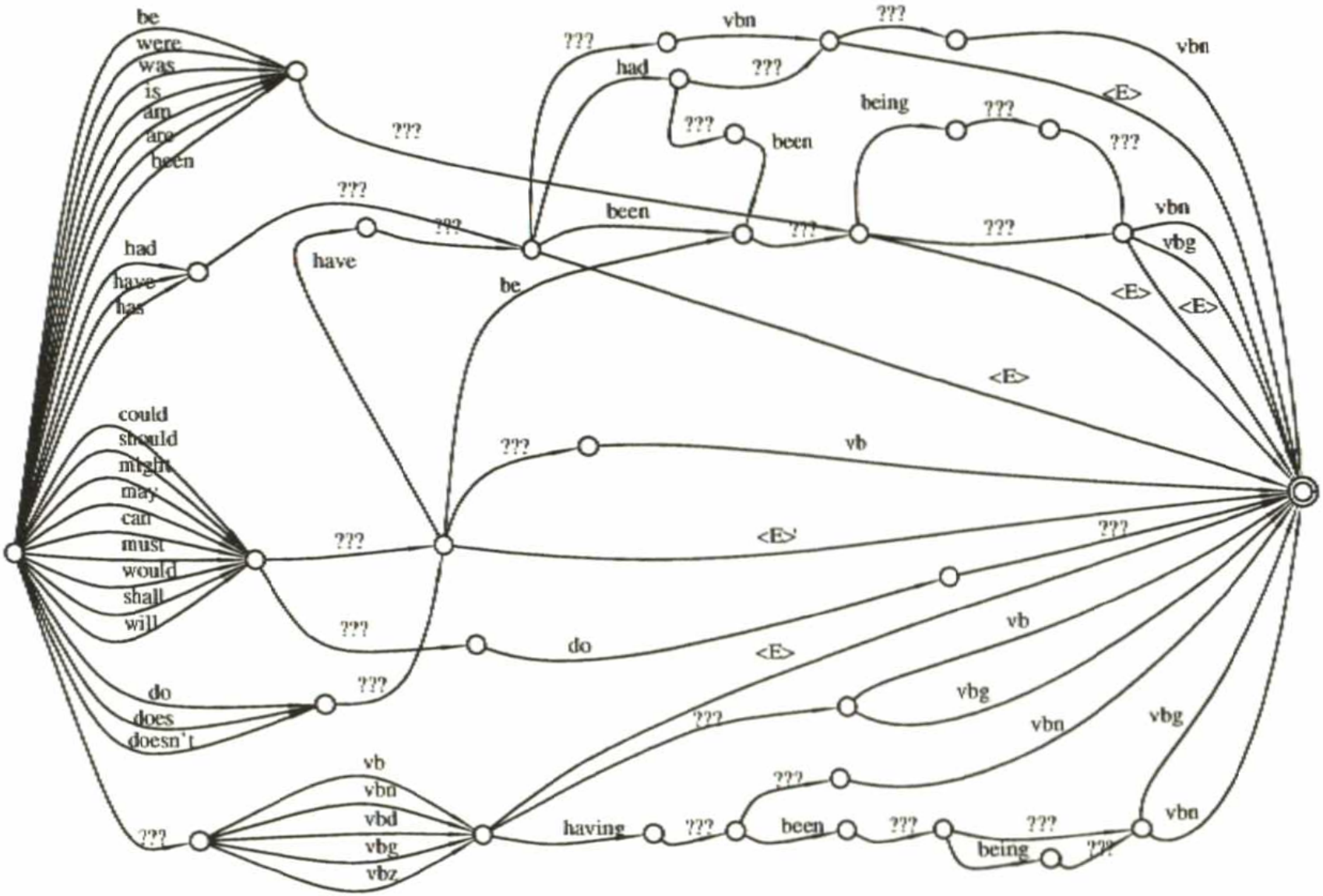
(S (N Mike N) thinks that (S (N Robert N) kept (N the book N)S)S)

Algorithm

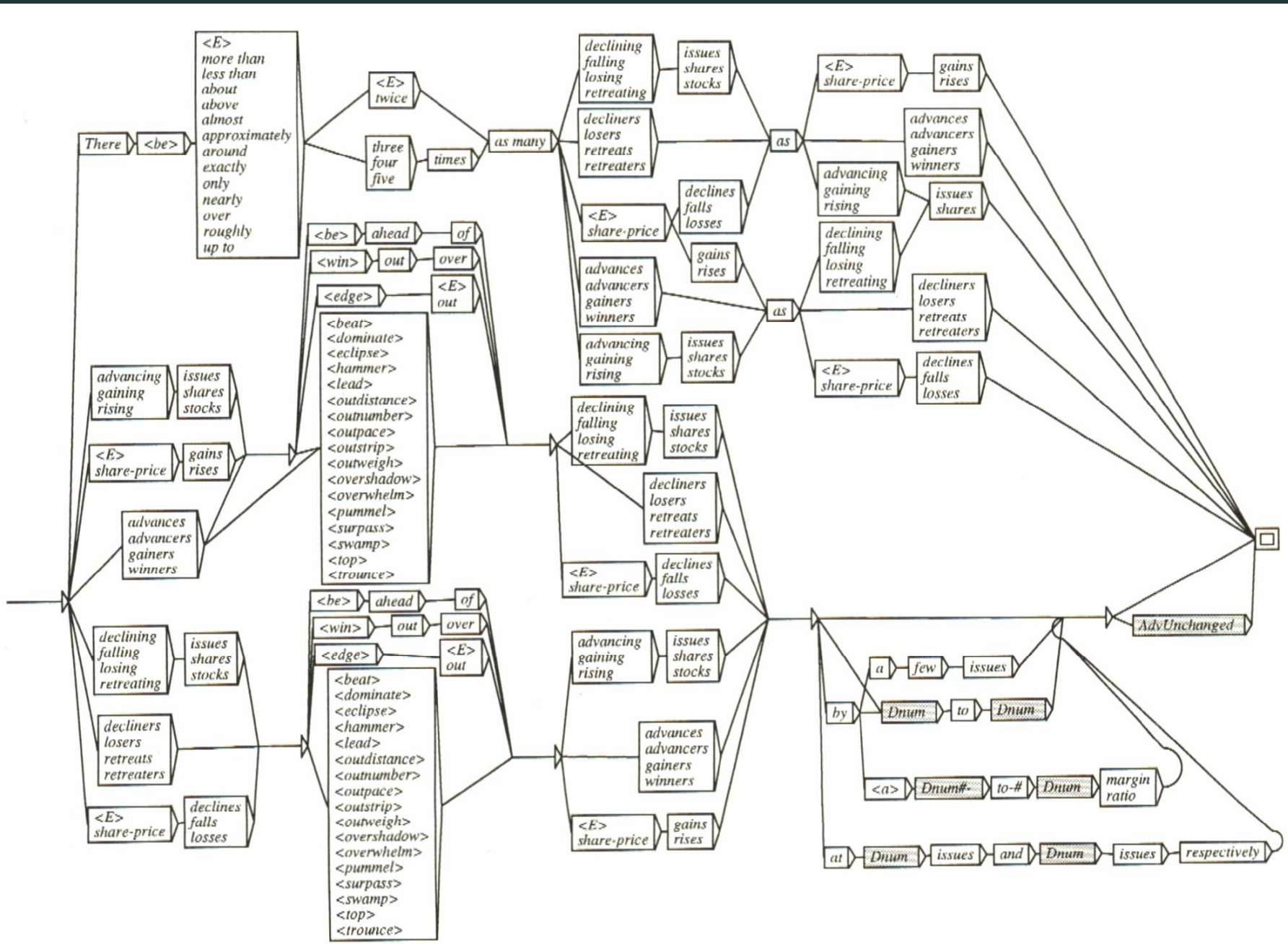
```
TransducerParse(T, sentence):
  sent1 = sentence

  while(sent2 = Apply_Transducer(T, sent1) != sent1):
    sent1 = sent2

  return sent1
```

note: I haven't worked out the math, but I'm pretty sure this algorithm actually has the full power of a turing machine and can describe any Chomsky type 0 language.

Auxilliary Verb Complex



MORE


There
<be>
<E>
more than
less than
about
above
almost
approximately
around
exactly
only
nearly
over
roughly
up to
<E>
twice
three
four
five
times
as many
declining
falling
losing
retreating
issues
shares
stocks
decliners
losers
retreats
retreaters
<E>
share-price
declines
falls
losses
gains
rises
advances
advancers
gainers
winners
advancing
gaining
rising
as
<E>
share-price
gains
rises
advances
advancers
gainers
winners
advancing
gaining
rising
issues
shares
declining
falling
losing
retreating
decliners
losers
retreats
retreaters
<E>
share-price
declines
falls
losses
<be>
ahead
of
<win>
out
over
<edge>
<E>
out
<beat>
<dominate>
<eclipse>
<hammer>
<lead>
<outdistance>
<outnumber>
<outpace>
<outstrip>
<outweigh>
<overshadow>
<overwhelm>
<pummel>
<surpass>
<swamp>
<top>
<trounce>
AdvUnchanged
a
few
issues
by
Dnum
to
Dnum
<a>
Dnum#
to-#
Dnum
margin
ratio
at
Dnum
issues
and
Dnum
issues
respectively

Great, but can I use it to do real work?

Uses in NLP (search)

Search

We have some data indexed with words (ie, natural language), and we want to find the correct stuff based on a query word.

Search

We have some data indexed with words (ie, natural language), and we want to find the correct stuff based on a query word.

How do??

Trie

Trie

- Directed (Deteriministic)

Trie

- Directed (Deteriministic)
- Acyclic

Trie

- Directed (Deteriministic)
- Acyclic
- Word

Trie

- Directed (Deteriministic)
- Acyclic
- Word
- Graph

Trie

- Directed (Deteriministic)
- Acyclic
- Word
- Graph

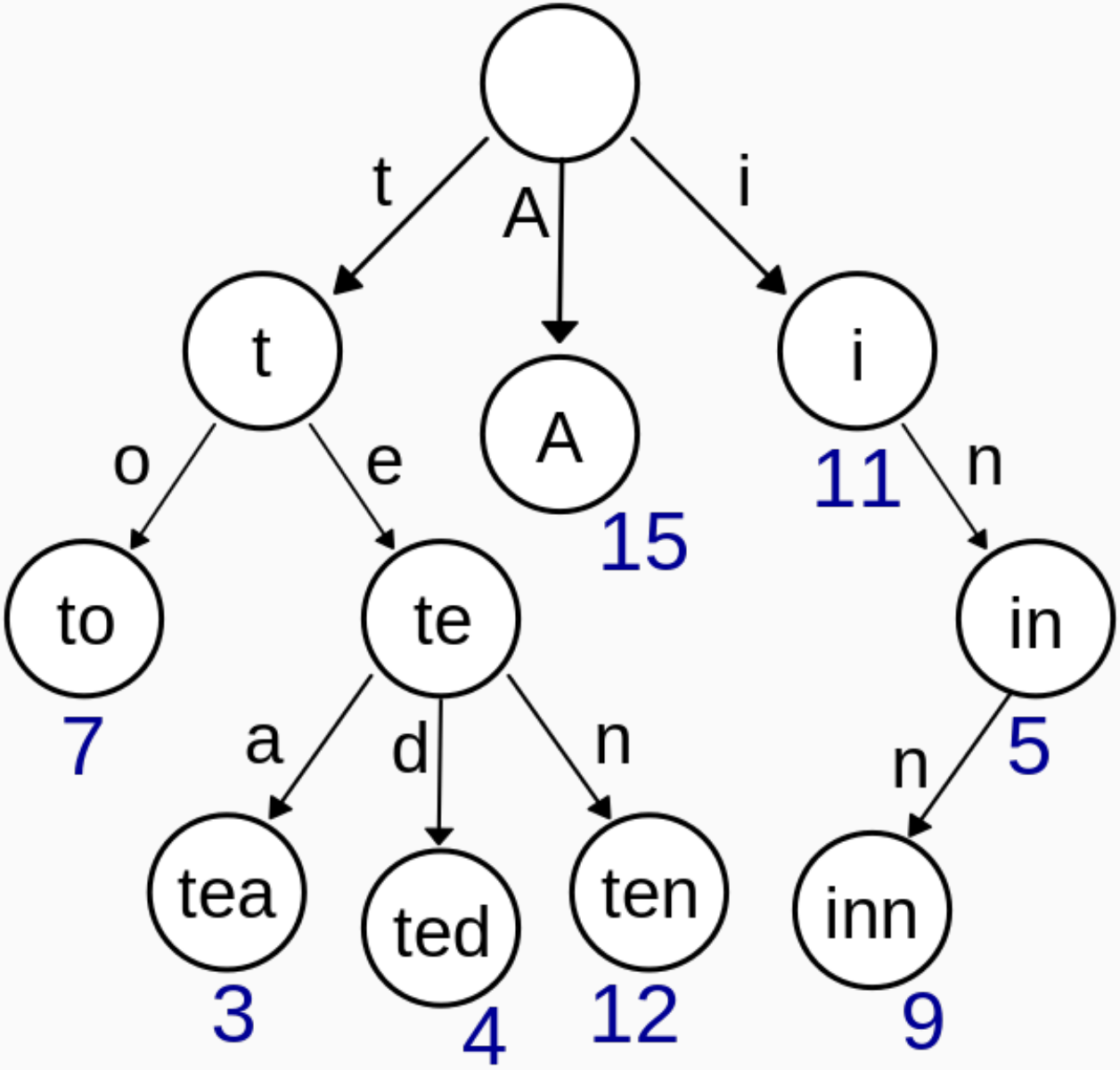
Trie

- Directed (Deteriministic)
- Acyclic
- Word
- Graph

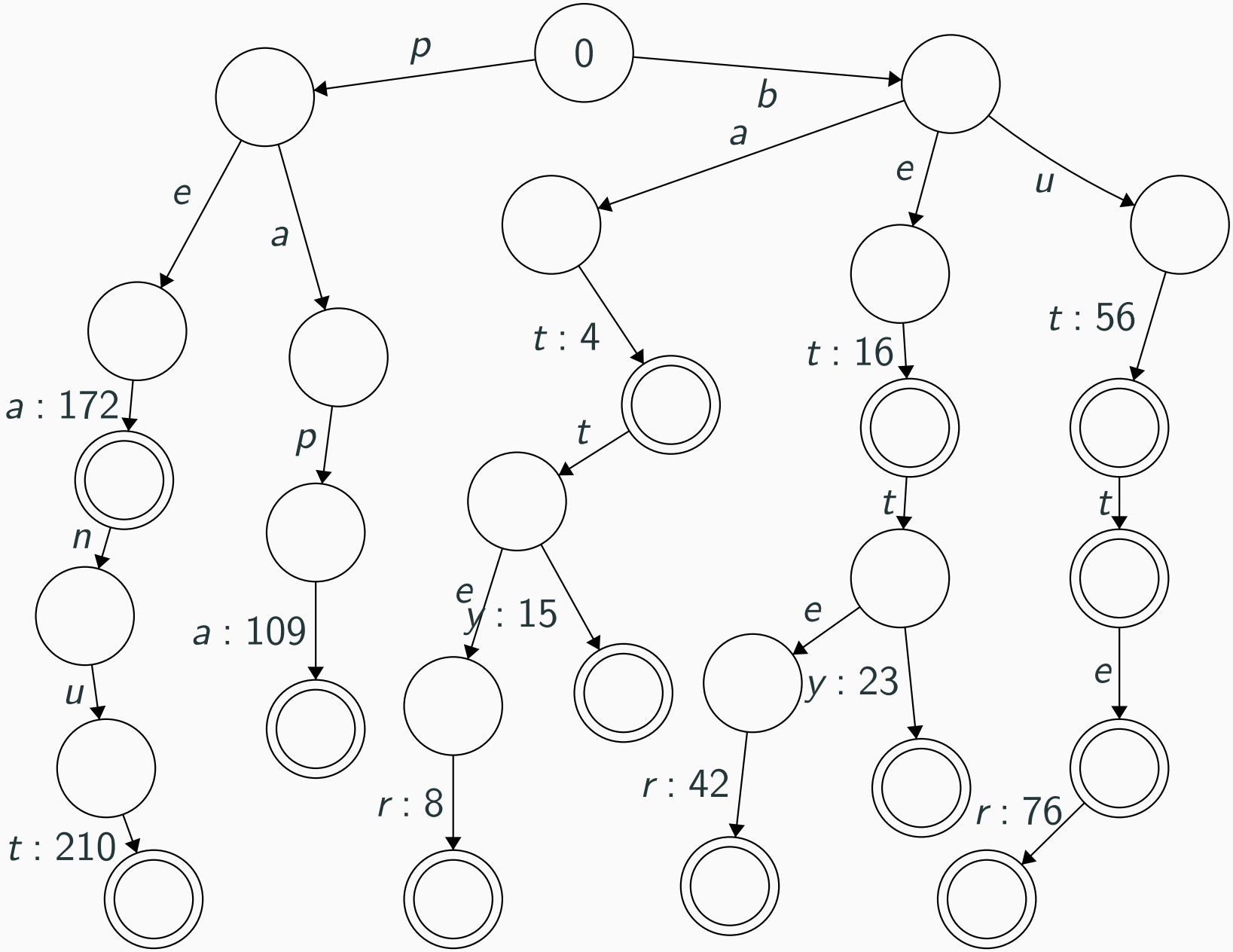
Trie/DAWG

A finite state transducer that contains a dictionary by storing words as paths through the FSM

Trie/DAWG



Trie/DAWG


0
p
b
e
a
a
e
u
t : 4
t : 16
t : 56
a : 172
p
t
t
t
n
e
y : 15
e
e
a : 109
y : 23
u
r : 8
r : 42
r : 76
t : 210

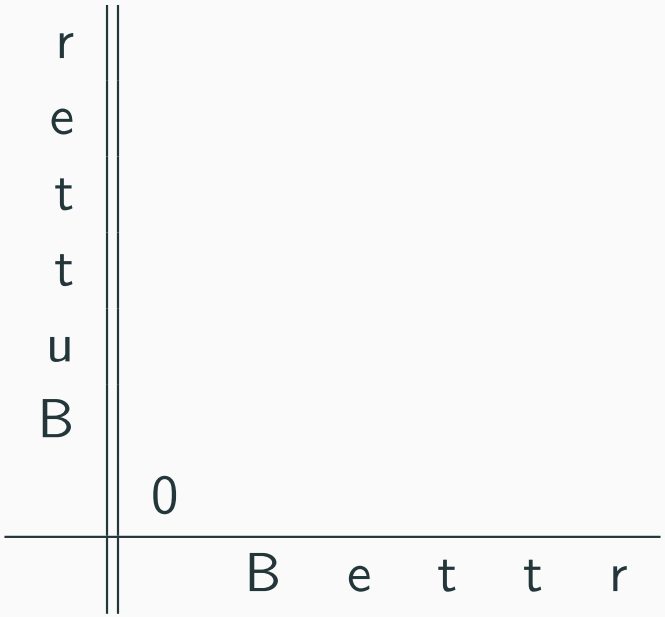
Cool, so what?

Cool, so what?

If only there was a way to take steps through the trie that are close to but not exactly the same as the query...

Levenshtein

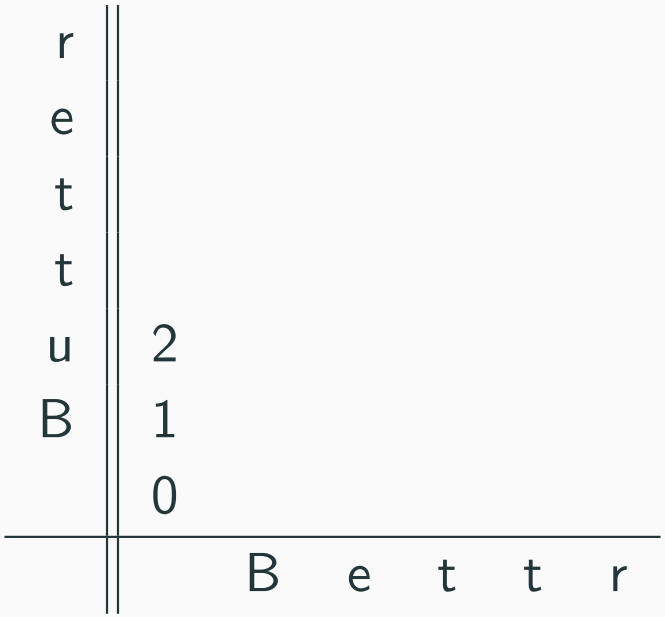
Levenshtein



Levenshtein

r						
e						
t						
t						
u						
B	1					
	0					
		B	e	t	t	r

Levenshtein



Levenshtein

r	6					
e	5					
t	4					
t	3					
u	2					
B	1					
	0					
		B	e	t	t	r

Levenshtein

r	6	5				
e	5	4				
t	4	3				
t	3	2				
u	2	1				
B	1	0				
	0	1				
		B	e	t	t	r

Levenshtein

r	6	5	4			
e	5	4	3			
t	4	3	3			
t	3	2	2			
u	2	1	1			
B	1	0	1			
	0	1	2			
		B	e	t	t	r

Levenshtein

r	6	5	4	3		
e	5	4	3	2		
t	4	3	3	1		
t	3	2	2	1		
u	2	1	1	2		
B	1	0	1	2		
	0	1	2	3		
		B	e	t	t	r

Levenshtein

r	6	5	4	3	3	
e	5	4	3	2	2	
t	4	3	3	1	1	
t	3	2	2	1	1	
u	2	1	1	2	3	
B	1	0	1	2	3	
	0	1	2	3	4	
		B	e	t	t	r

Levenshtein

r	6	5	4	3	3	2
e	5	4	3	2	2	2
t	4	3	3	1	1	2
t	3	2	2	1	1	2
u	2	1	1	2	3	4
B	1	0	1	2	3	4
	0	1	2	3	4	5
		B	e	t	t	r

Levenshtein distance: 2

Levenshtein

r	6	5	4	3	3	2
e	5	4	3	2	2	2
t	4	3	3	1	1	2
t	3	2	2	1	1	2
u	2	1	1	2	3	4
B	1	0	1	2	3	4
	0	1	2	3	4	5
		B	e	t	t	r

Levenshtein distance: 2

Way too expensive to do for every string

Levenshtein

r	6	5	4	3	3	2
e	5	4	3	2	2	2
t	4	3	3	1	1	2
t	3	2	2	1	1	2
u	2	1	1	2	3	4
B	1	0	1	2	3	4
	0	1	2	3	4	5
		B	e	t	t	r

Levenshtein distance: 2

Way too expensive to do for every string

but...

Levenshtein Automata

Levenshtein Automata

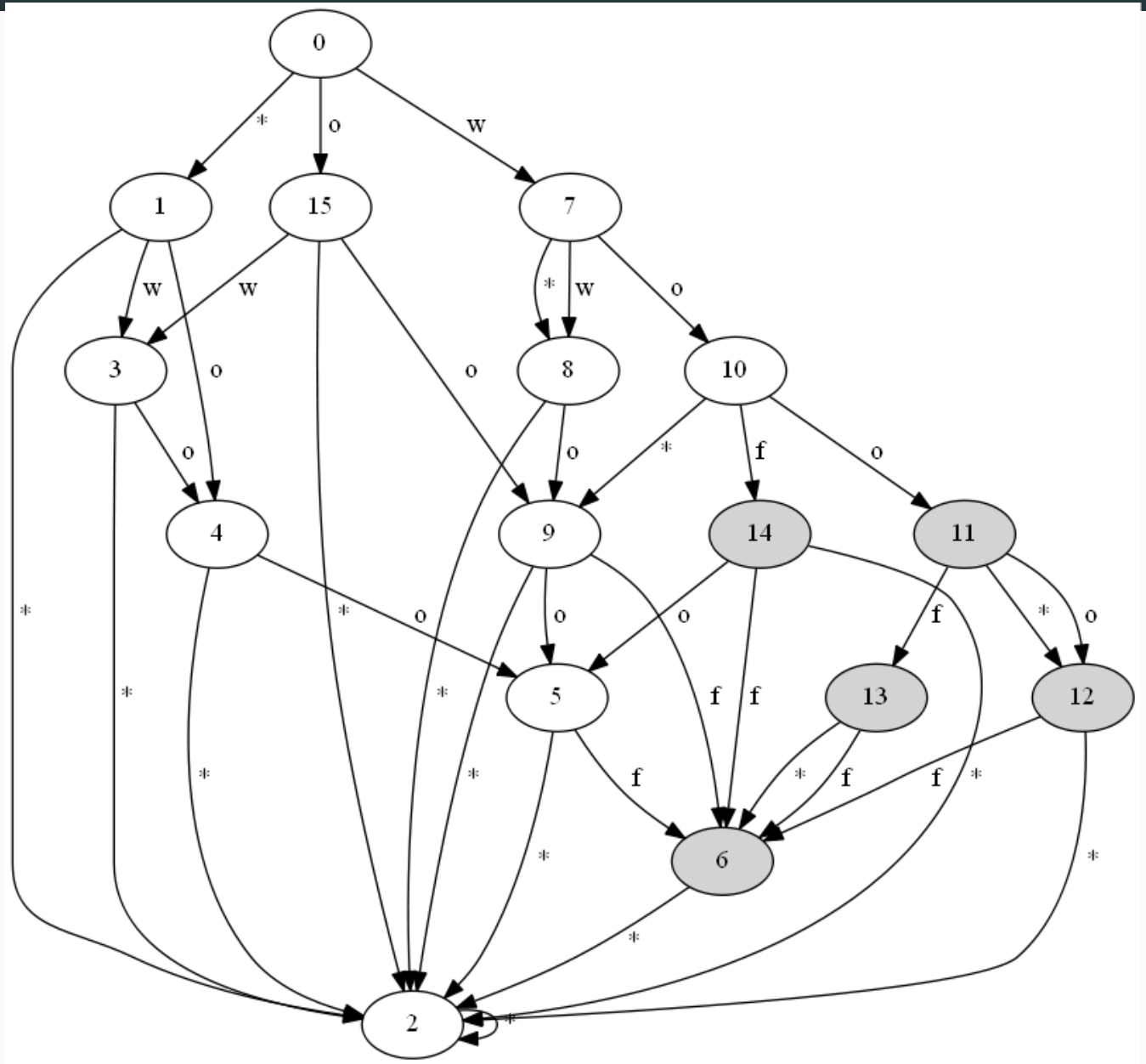
r	6	5	4	3	3	2
e	5	4	3	2	2	2
t	4	3	3	1	1	2
t	3	2	2	1	1	2
u	2	1	1	2	3	4
B	1	0	1	2	3	4
	0	1	2	3	4	5
		B	e	t	t	r

Mihov and Schultz (2002)

Python implementation

```
class LevenshteinAutomaton:
  def __init__(self, string, n):
      self.string = string
      self.max_edits = n
  def start(self):
      return range(len(self.string)+1)
  def step(self, state, c):
      new_state = [state[0]+1]
      for i in range(len(state)-1):
          cost = 0 if self.string[i] == c else 1
          new_state.append(min(new_state[i]+1, state[i]+cost,
            state[i+1]+1))
      return [min(x,self.max_edits+1) for x in new_state]
  def is_match(self, state):
      return state[-1] <= self.max_edits
  def can_match(self, state):
      return min(state) <= self.max_edits
```

Levenshtein Automaton (credit to Jules Jacobs)



Properties

Properties

Numbers of states is linearly bounded by the length of the query string

Properties

Numbers of states is linearly bounded by the length of the query string

builds in O(n) time

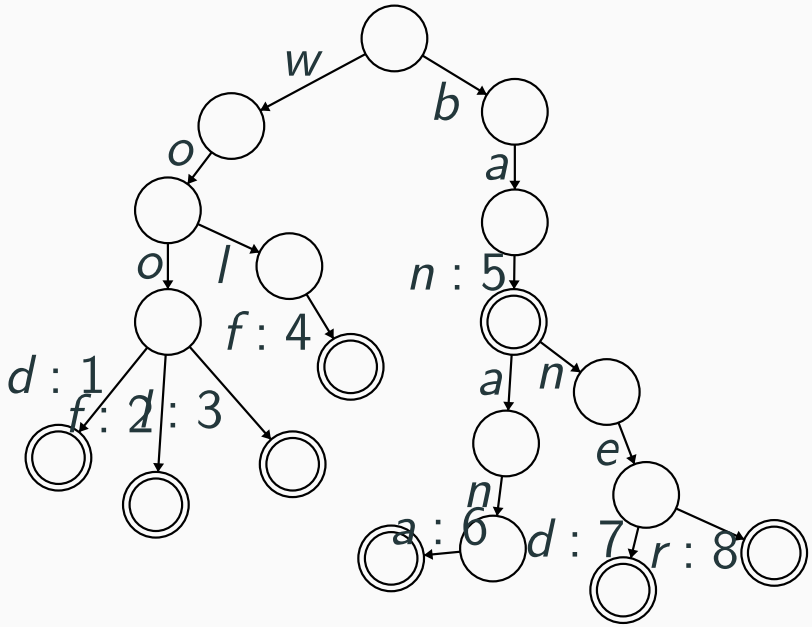
Properties

Numbers of states is linearly bounded by the length of the query string

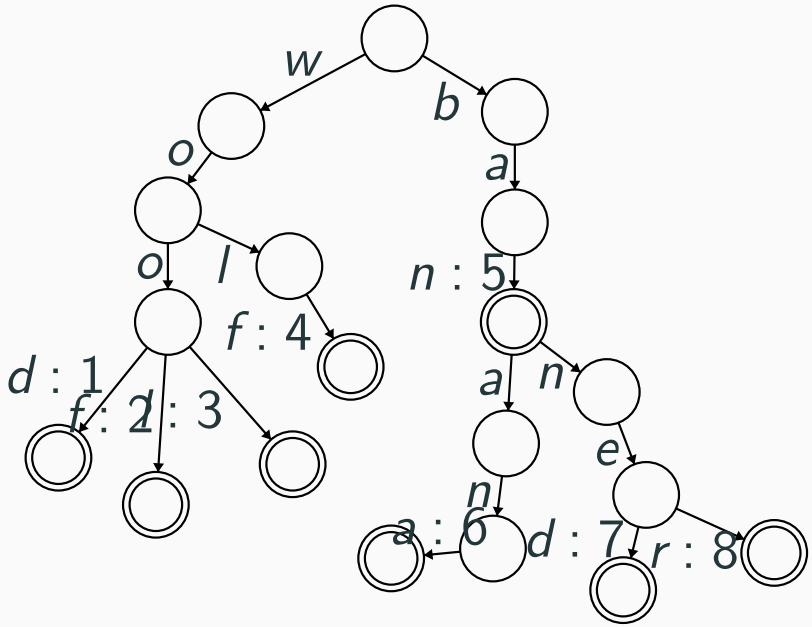
builds in O(n) time

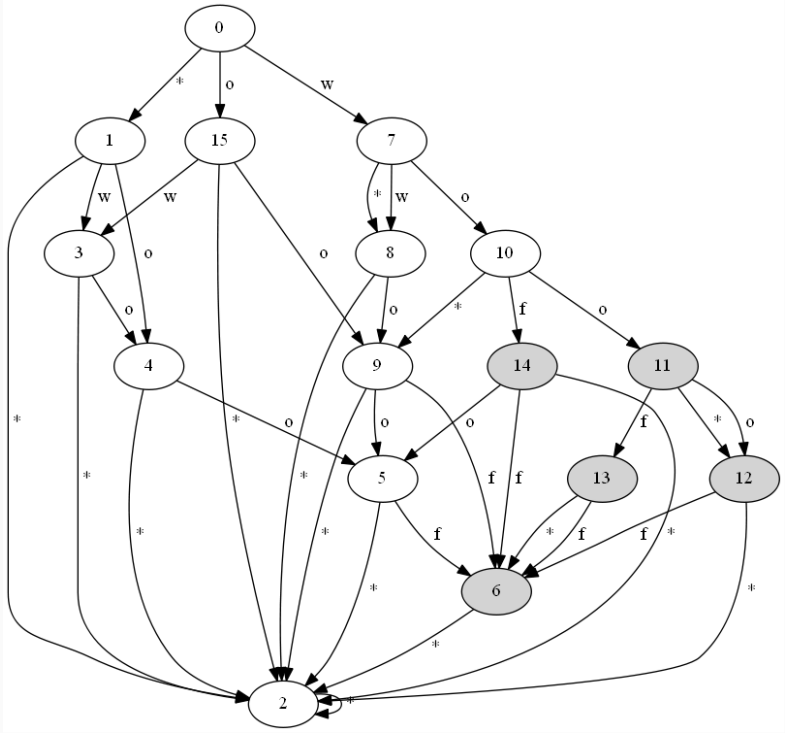
(can cut that down to O(δ) by using sparse vectors)

Use



Use





Wrap-up

Summary

We can do better.

Summary

We can do better.

What if we don't have the whole search term from the user to make a Levenshtein Automaton with?

Summary

We can do better.

What if we don't have the whole search term from the user to make a Levenshtein Automaton with?

What if it's only a prefix?

Summary

We can do better.

What if we don't have the whole search term from the user to make a Levenshtein Automaton with?

What if it's only a prefix?

There still isn't a super efficient way to do fuzzy prefix searching.

Questions?

Tools and resources

Apache Lucene full stack FOSS search solution

Marisa_trie Python trie, shares interface with DAWG

DAWG another Python trie

Roche and Schabes 1997 primary source on FS comp ling

Mihov and Schultz 2002 levenshtein automata

References I