Finite State Language Processing

Gertjan van Noord

some parts based on joint work with:

Jan Daciuk
Dale Gerdemann

Motivation

- Efficiency
- Compactness
- Closure Properties

Sobering remark

- Not always applicable
- But if they are:
  - Practical
  - Elegant

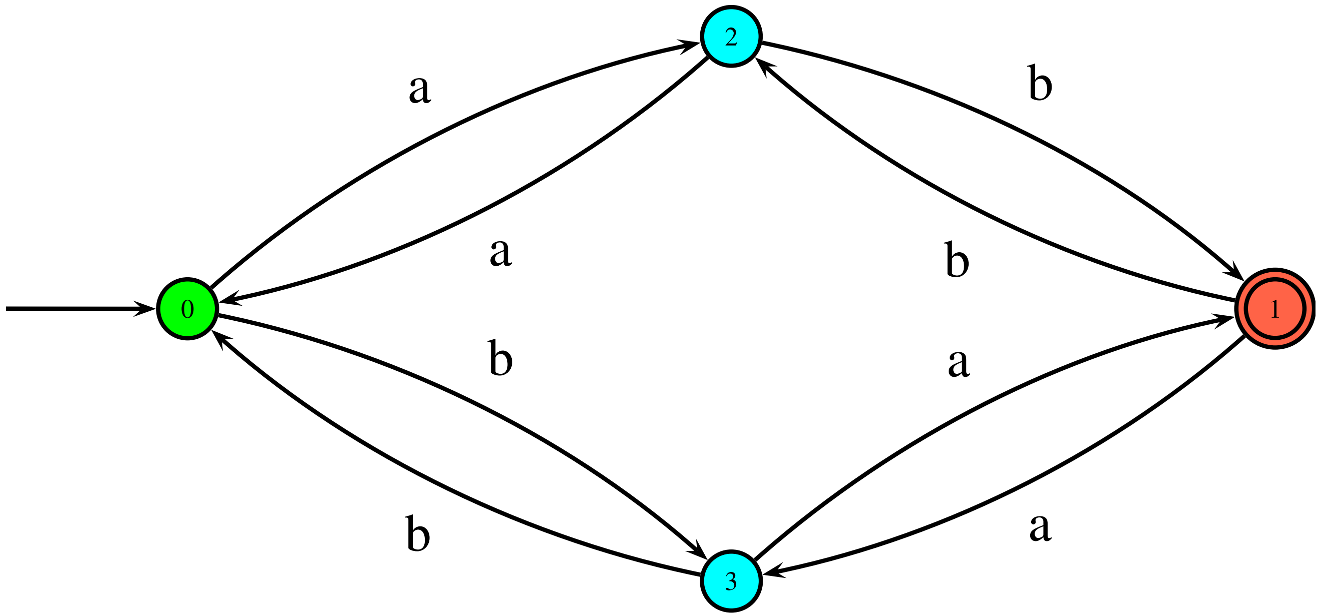
Overview

- Finite State Automata
- Dictionary Construction; Perfect Hash; Tuple Dictionaries
- Regular Expressions
- Finite State Optimality Phonology

PART 1: Finite State Automata

- Finite State Acceptors
- Finite State Transducers
- Weighted Finite State Automata

Example



Definition

A finite state acceptor $M = (Q, \Sigma, E, S, F)$:

- Q is a finite set of states
- Σ is a set of symbols
- $S \subseteq Q$ is a set of start states
- $F \subseteq Q$ is a set of final states
- E is a finite set of edges $Q \times (\Sigma \cup \{\epsilon\}) \times Q$.

Definition (2)

Paths:

1. for all $q \in Q, (q, \epsilon, q) \in \widehat{E})$
2. for all $(q_0, x, q) \in E$: $(q_0, x, q) \in \widehat{E}$
3. if (q_0, x_1, q_1) and (q_1, x_2, q) are both in $\widehat{E}$ then $(q_0, x_1x_2, q) \in \widehat{E}$

Definition (3)

- The language accepted by M:

$$L(M) = \{w | q_s \in S, q_f \in F, (q_s, w, q_f) \in \widehat{E}\}$$

- A language L is *regular* iff there is a finite state acceptor M such that $L = L(M)$.

Deterministic Finite State Acceptor

- Deterministic:
  - ⋆ Single start state
  - ⋆ No epsilon transitions
  - ⋆ For each state and each symbol there is at most one applicable transition
- For every M there is a deterministic automaton M' such that $L(M) = L(M')$.
- There is an algorithm which computes M' for any M.
- *Efficiency*!

Minimal Finite State Acceptor

- For every deterministic M there is a unique equivalent *minimal* M'
- There is an efficient algorithm which computes M' for any M.
- *Compactness*!

Some languages are not regular

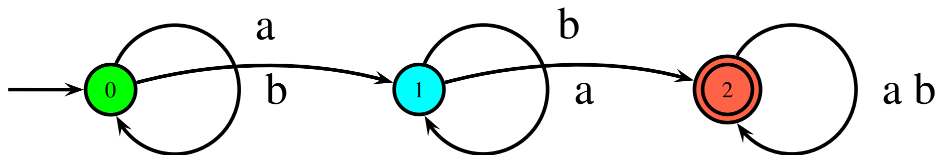
$L = a^n b^n$ is not a regular language.

- suppose L was regular
- then there is a finite automaton M for it. Suppose M has m states
- then what about the string $a^m b^m$. Since it is twice as long as m, there must be a state p in M which is traversed at least twice.
- now, while recognizing $a^m b^m$, at which point do we switch from a's to b's? Before the cycle? No. During the cycle? No. After the cycle? No.
- L cannot be regular

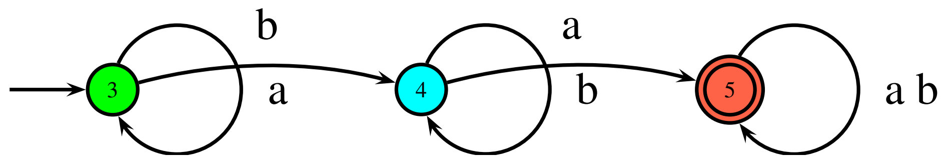
Closure Properties

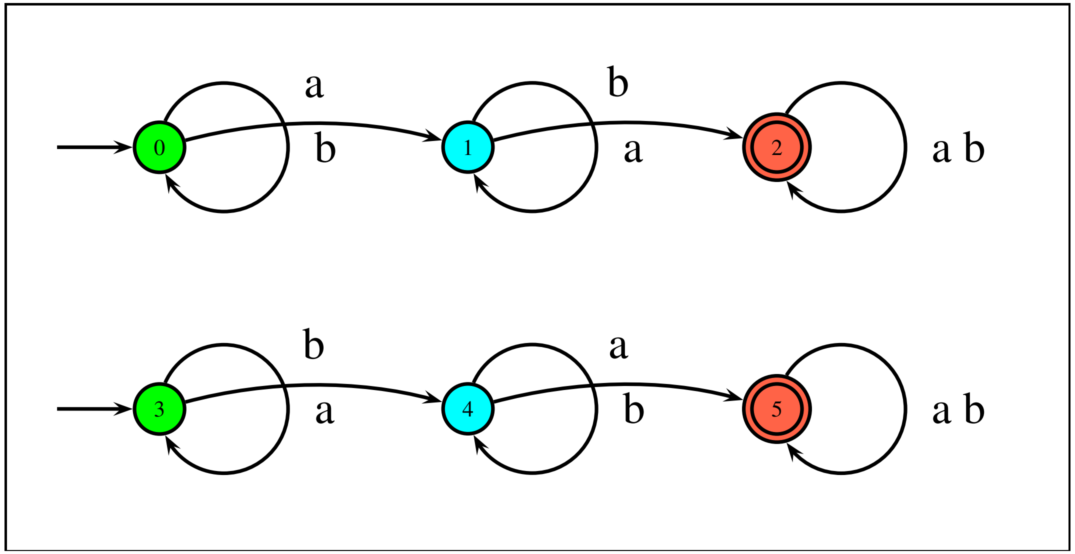
- union
- concatenation
- Kleene-closure
- complementation
- intersection
- . . .

Union

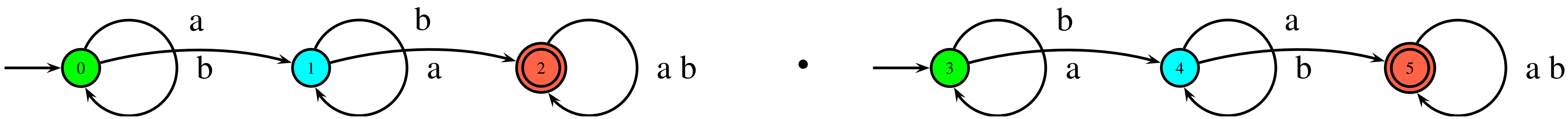


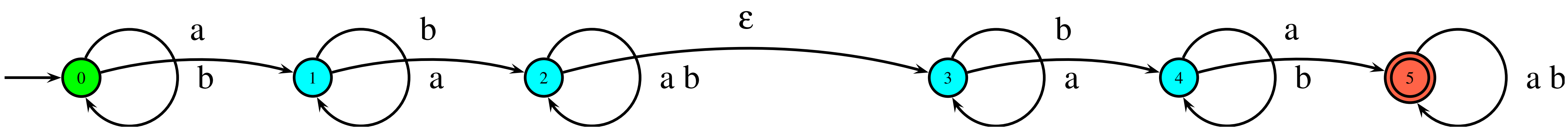
∪



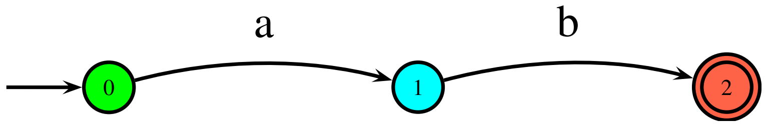


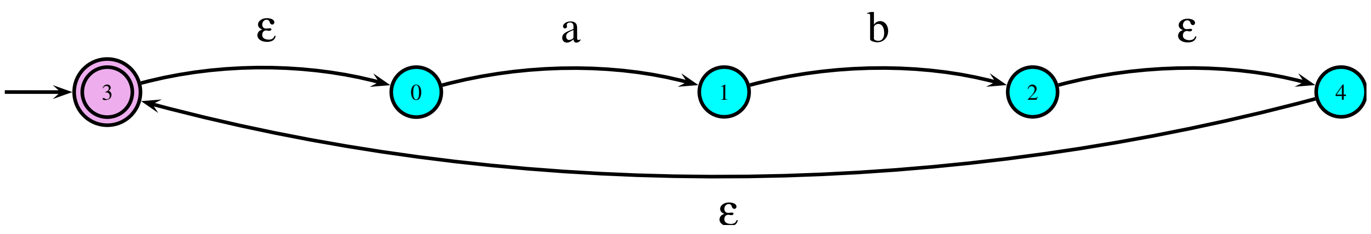
Concatenation



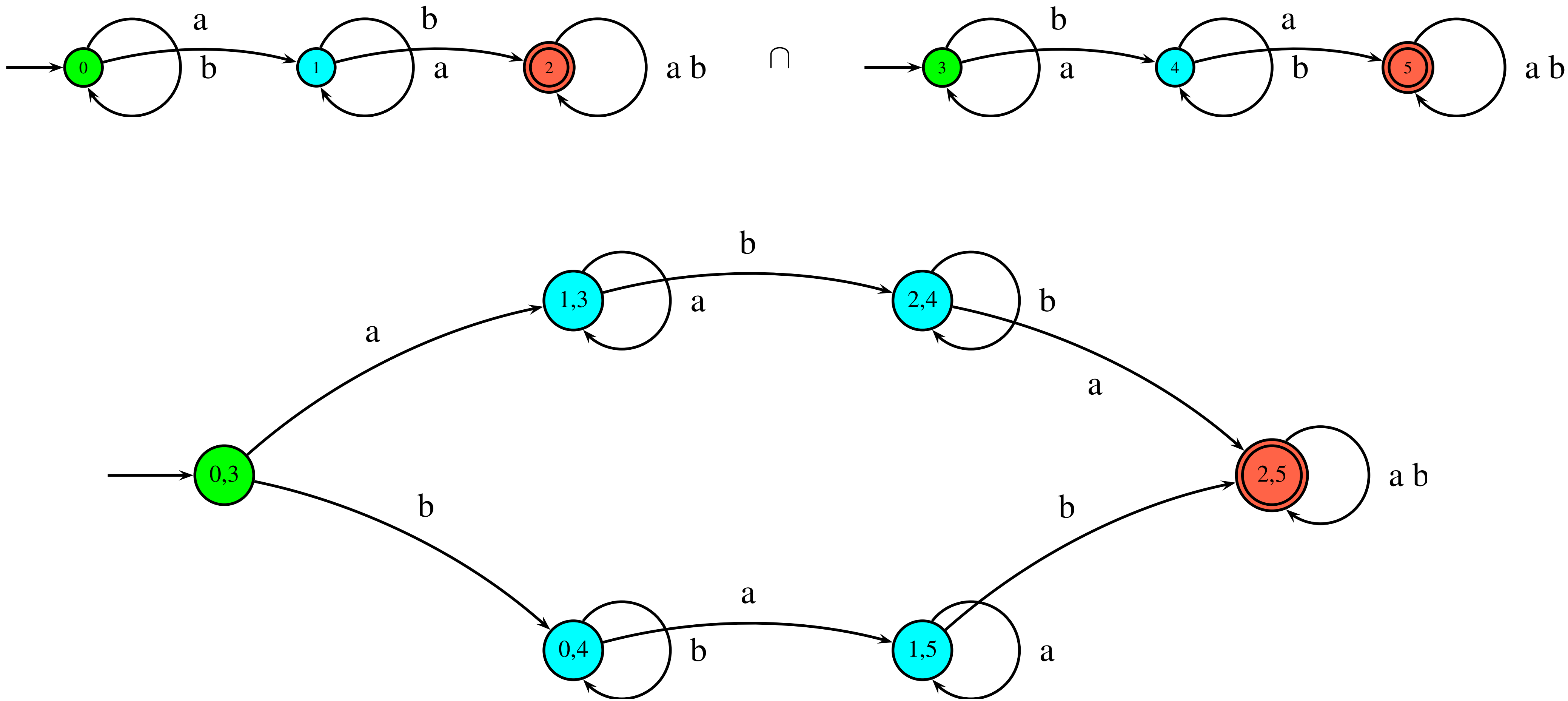


Kleene Closure

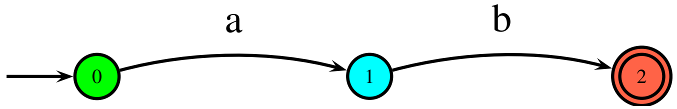


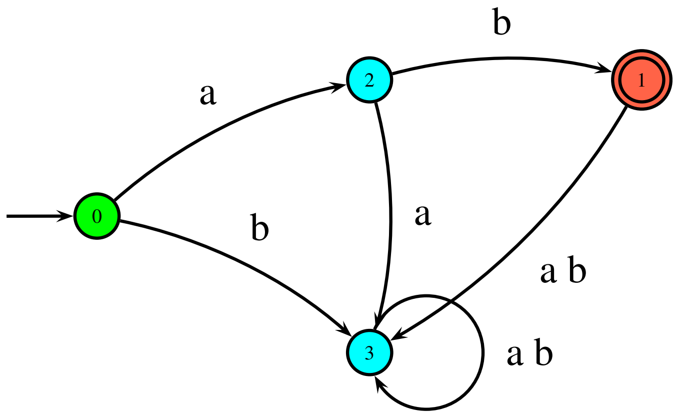


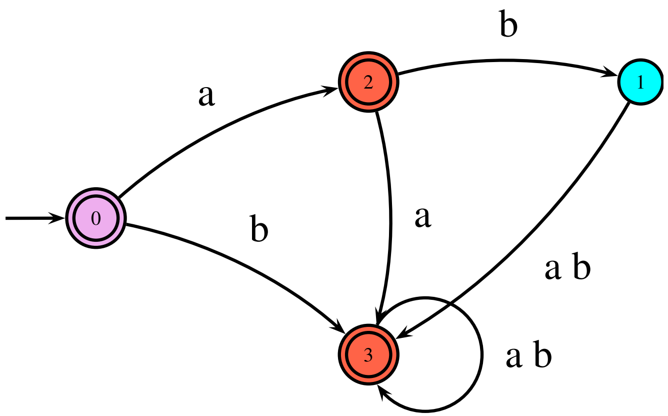
Intersection



Complement

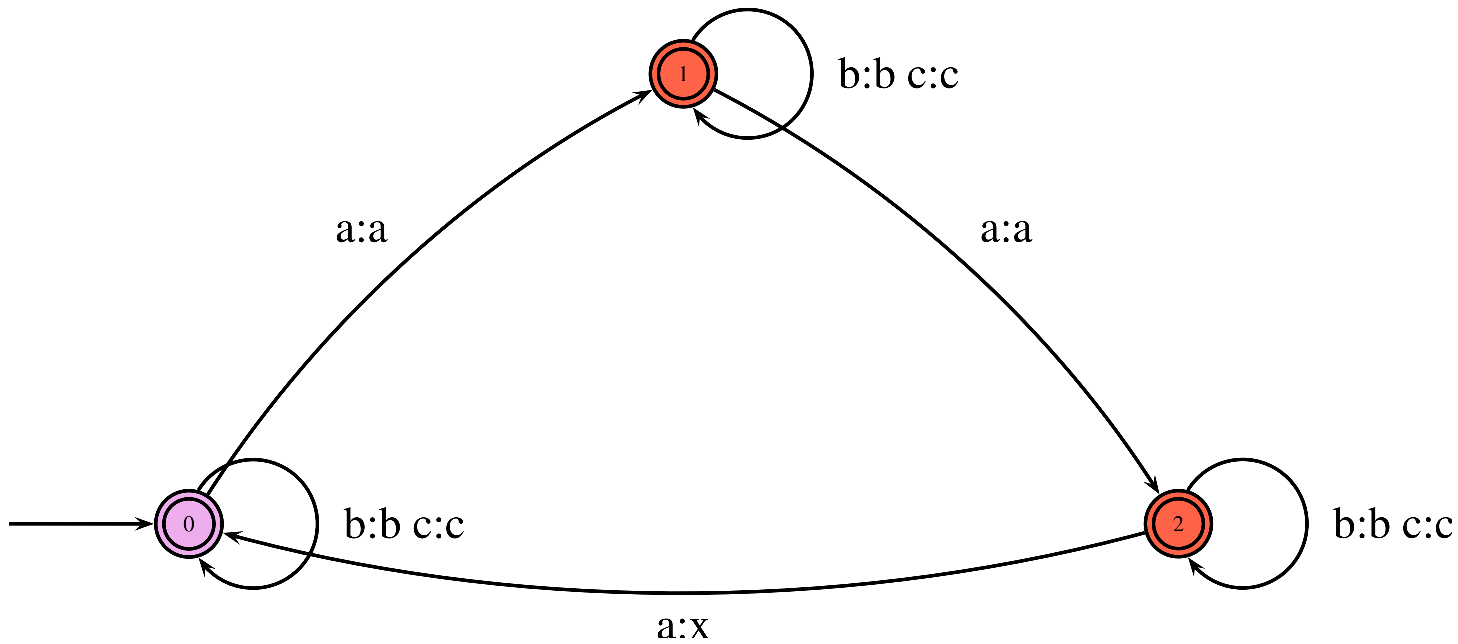






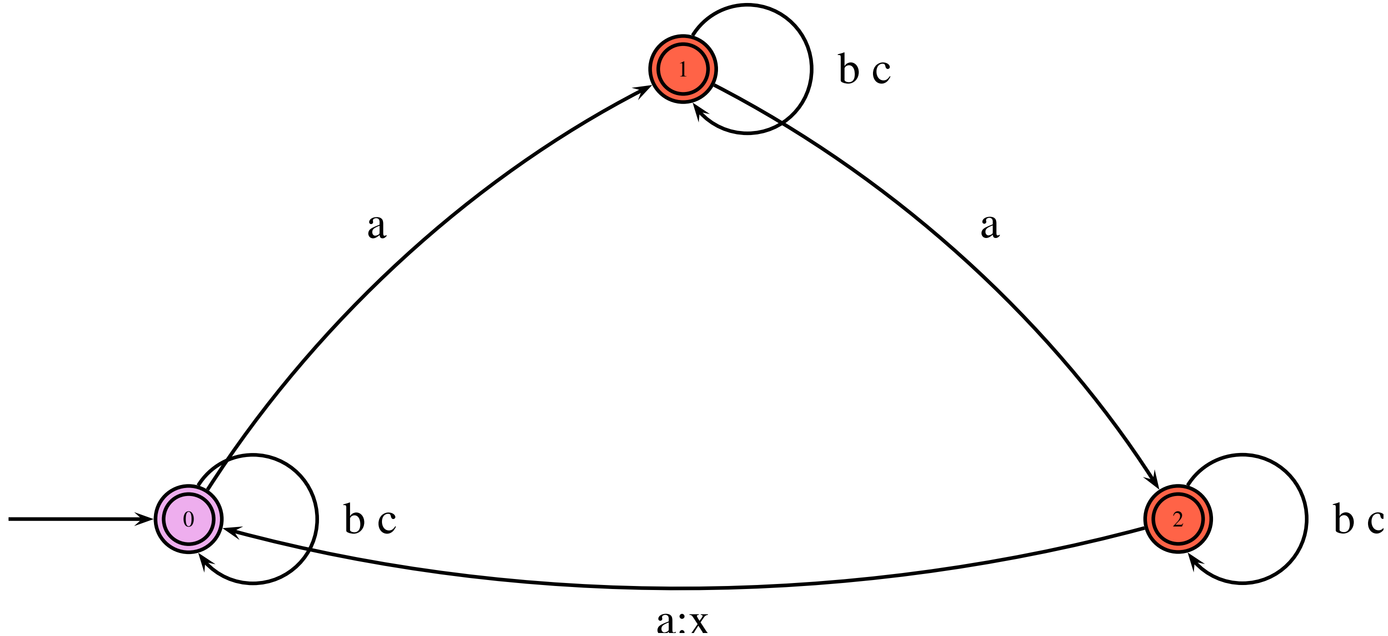
- input automaton must be deterministic

Finite State Transducers



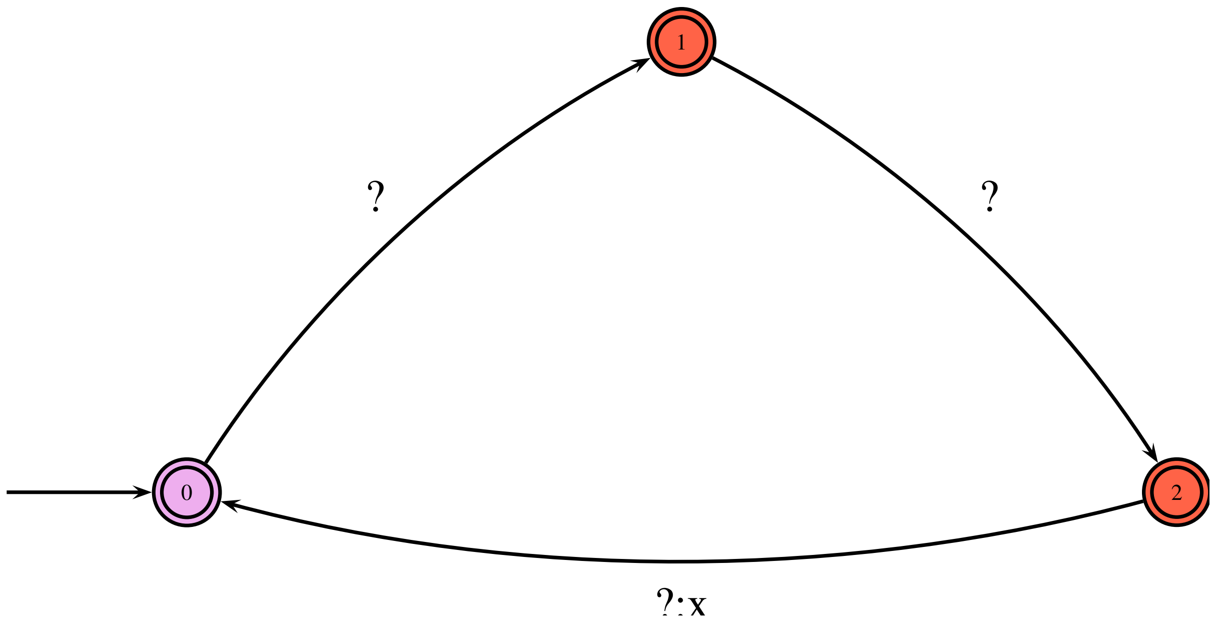
- every third a is mapped to x

Finite State Transducers



- identity pair is written as single symbol

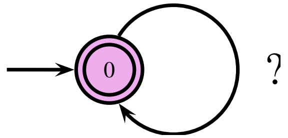
Finite State Transducers



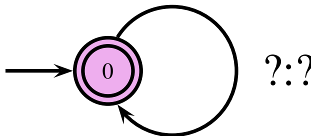
- question mark to refer to arbitrary symbol

Distinction

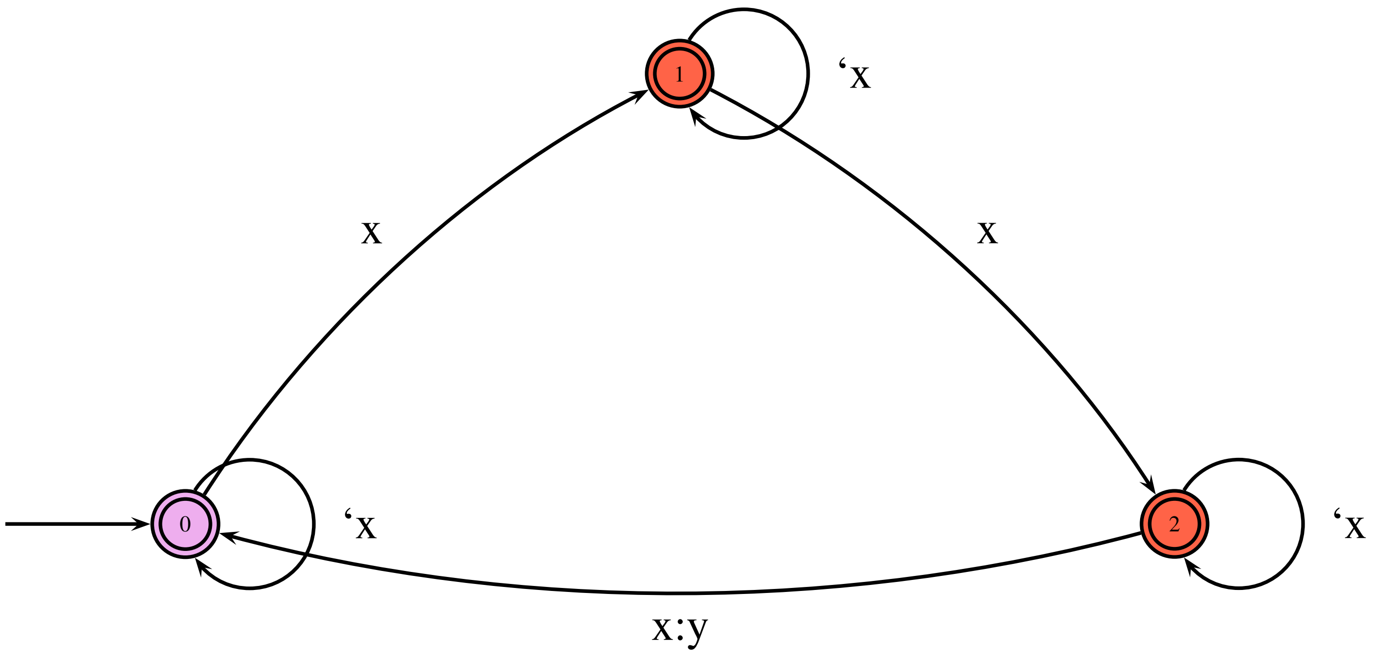
- Copy



- Garbage in, garbage out



Finite State Transducers



- term complement 'x to refer to an arbitrary symbol not equal to x.

Definition

A finite state transducer $M = (Q, \Sigma_d, \Sigma_r, E, S, F)$:

- Q is a finite set of states
- Σ_d, Σ_r are sets of symbols
- $S \subseteq Q$ is a set of start states
- $F \subseteq Q$ is a set of final states
- E is a finite set of edges $Q \times (\Sigma_d \cup \{\epsilon\}) \times \Sigma_r^* \times Q$.

Definition (2)

Paths:

1. for all $q \in Q, (q, \epsilon, \epsilon, q) \in \widehat{E})$
2. for all $(q_0, x, y, q) \in E$: $(q_0, x, y, q) \in \widehat{E}$
3. if (q_0, x_1, y_1, q_1) and (q_1, x_2, y_2, q) are both in $\widehat{E}$ then $(q_0, x_1x_2, y_1y_2, q) \in \widehat{E}$

Definition (3)

- The relation accepted by M:

$$R(M) = \{(x, y) | q_s \in S, q_f \in F, (q_s, x, y, q_f) \in \widehat{E}\}$$

- A relation R is regular iff there is a finite state transducer M such that $R = R(M)$.

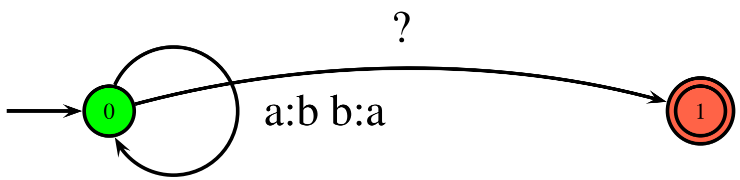
Closure

- regular relations are closed under *concatenation*, *Kleene-closure*, *union*
- *same length* regular relations are closed under *complementation*, *intersection*
- if R is a regular relation, then its domain and range are regular languages
- regular relations are closed under *inversion*!
- regular relations are closed under *composition*!

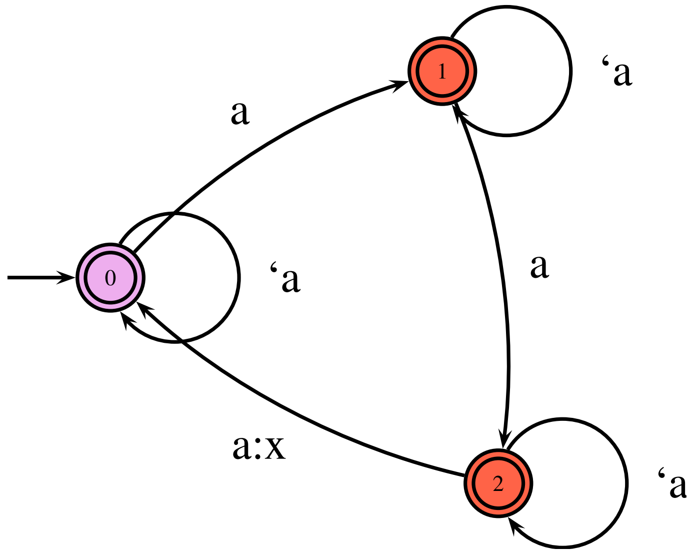
Composition

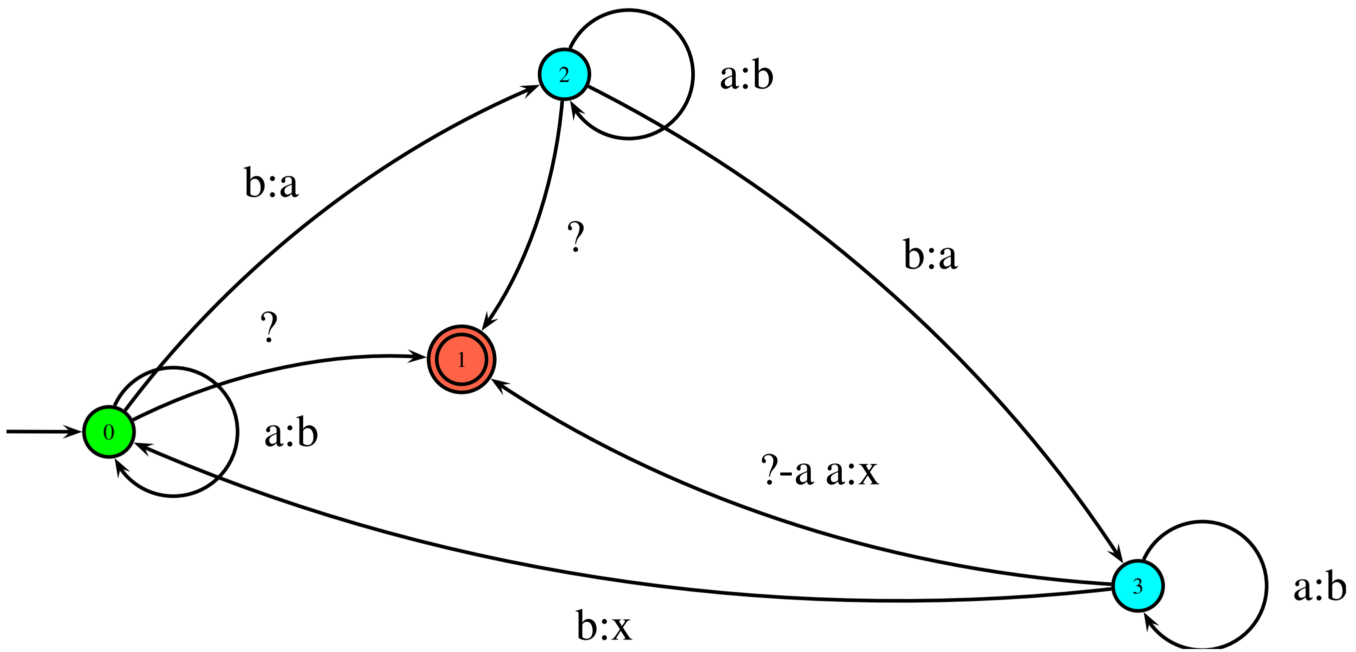
$$R_1 \circ R_2 : \{(x_1, x_3) | (x_1, x_2) \in R_1, (x_2, x_3) \in R_2\}$$

Composition: Example



$\circ$





Another example (Karttunen 1991)

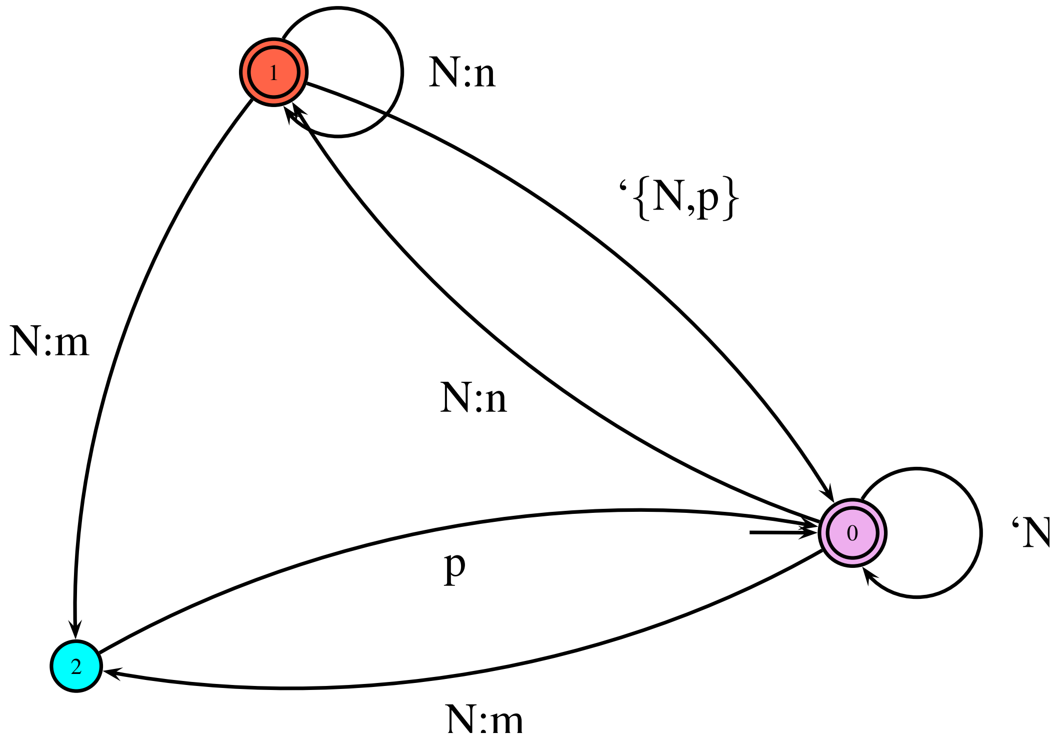
- Ordered application of context sensitive rules

```
N -> m / _ p; elsewhere n
p -> m / m _
```

- `kaNpan ==> kampan ==> kamman`
  `kaNton ==> kanton ==> kanton`

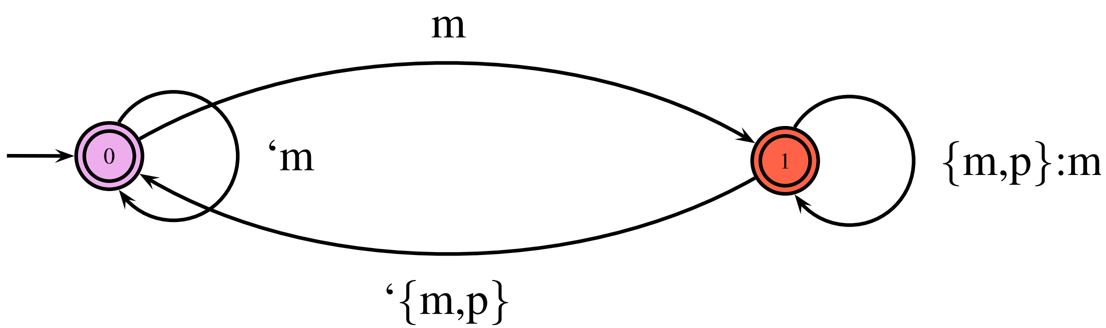
Another example (2)

- `N -> m / _ p; elsewhere n`

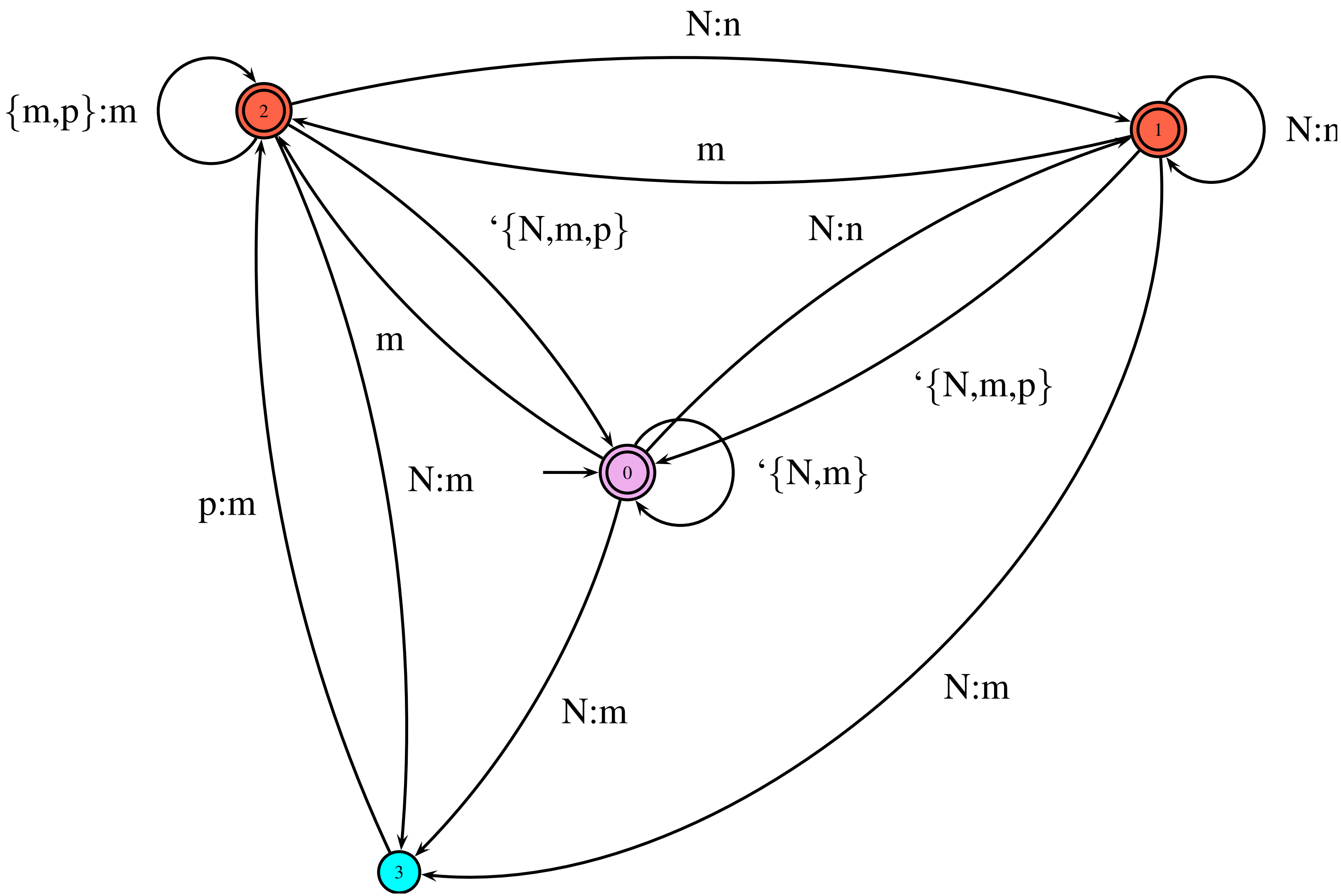


Another example (3)

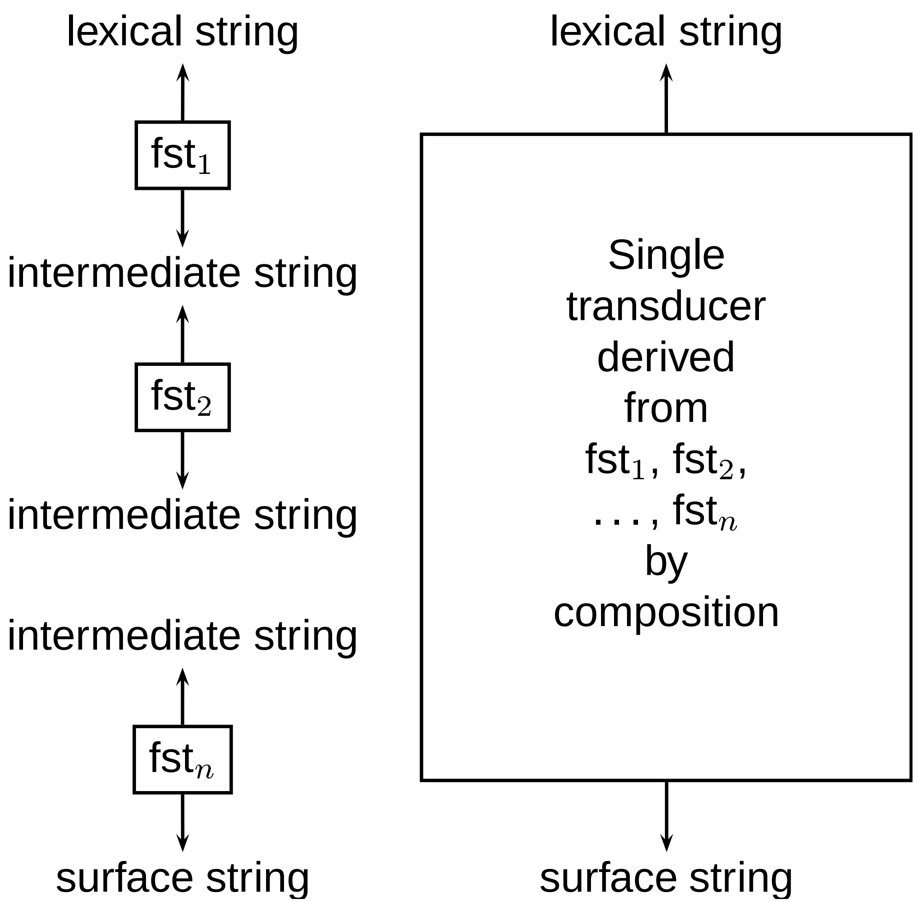
- `p -> m / m _`



Another example (4)



Cascades (Karttunen 1991)

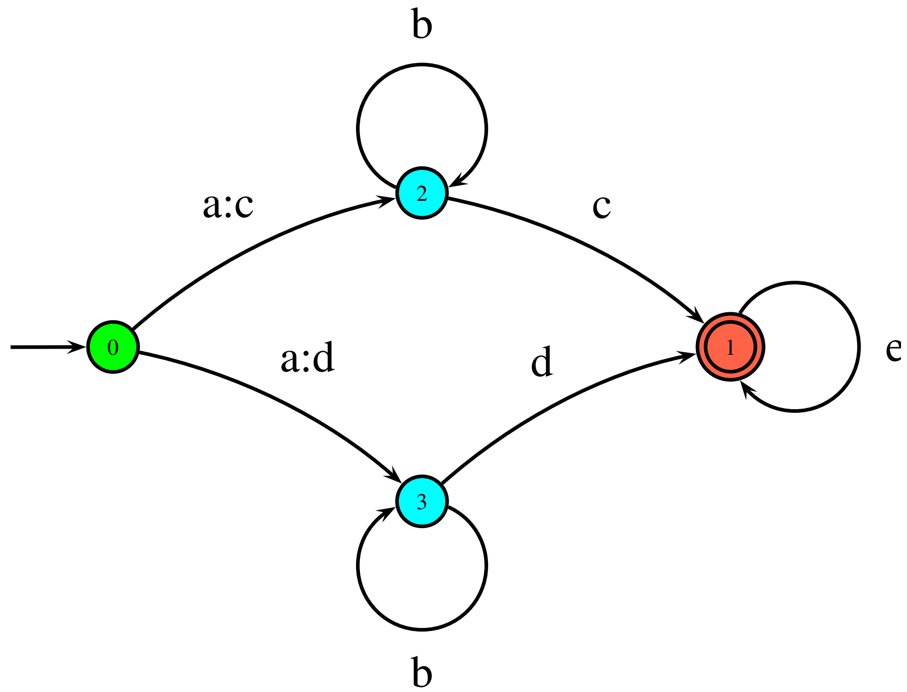


Transducers

- *functional* transducers
- *sequential* transducers: transducers which are deterministic for input
- *subsequential* transducers: additional output at final states

Example

- Some transducers are functional, but not sequential:



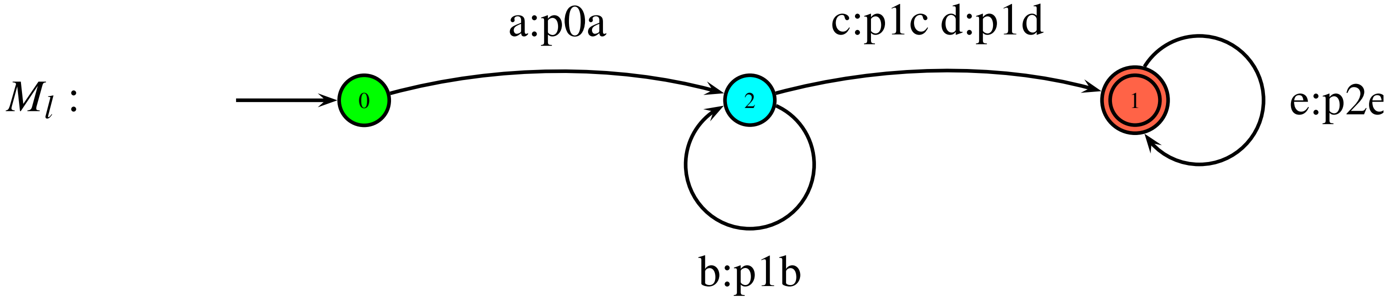
Algorithms

- Determine if a given transducer defines a *functional* relation.
- Determine if a given transducer defines a *subsequential* relation.
- Construct a subsequential transducer for a given transducer which defines a subsequential relation. *Determinization*
- Construct a minimal subsequential transducer for a given subsequential transducer. *Minimization*

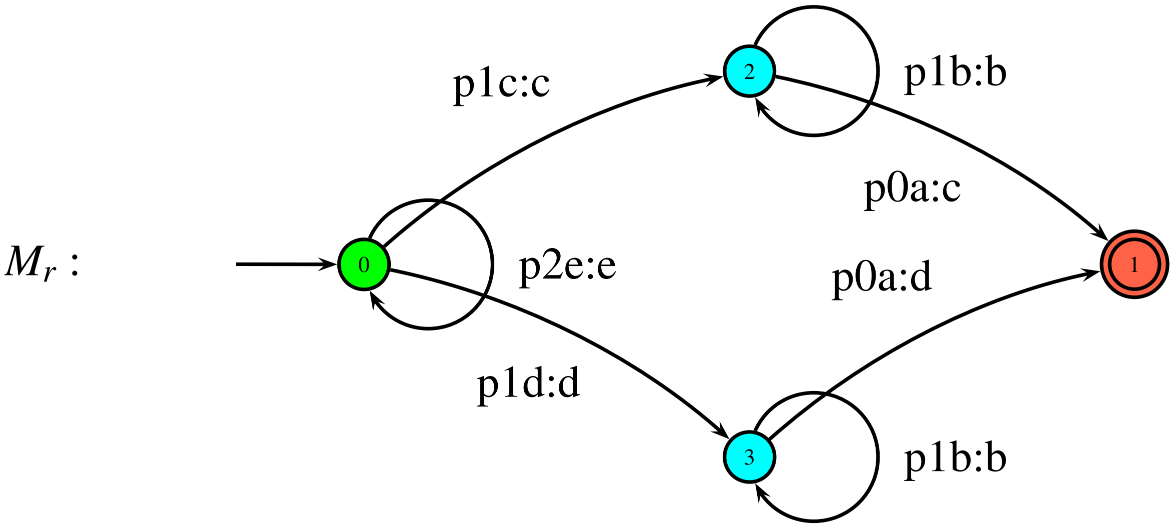
Bi-machines

- left-sequential transducer
- right-sequential transducer
- Every functional regular relation is the composition of a left-sequential transducer and a right-sequential transducer
- There is an algorithm which constructs for a given functional transducer the corresponding left- and right-sequential transducers.
- *Efficiency*

Example



M_l :
0
a:p0a
2
b:p1b
c:p1c d:p1d
1
e:p2e



M_r :
0
p2e:e
p1c:c
2
p1b:b
p0a:c
p1d:d
3
p1b:b
p0a:d
1

Example (2)

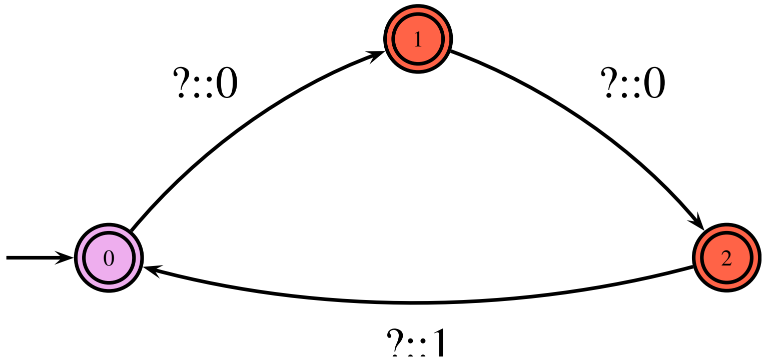
Input: `a b b b c e`

- Apply M_l: $\longrightarrow$ `p0a p1b p1b p1b p1c p2e`
- Reverse: $\longrightarrow$ `p2e p1c p1b p1b p1b p0a`
- Apply M_r: $\longrightarrow$ `e c b b b c`
- Reverse: $\longrightarrow$ `c b b b c e`

Weighted Finite Automata

- Weighted Finite State Acceptors
- Weighted Finite State Transducers

Example



$$\texttt{xxxxxxxxxxxxx} \Longrightarrow 4$$

Definition

A weighted finite state acceptor $M = (Q, \Sigma, W, E, S, F, \lambda)$:

- Q is a finite set of states
- Σ is a set of symbols
- W is set of weights
- $S \subseteq Q$ is a set of start states
- $F \subseteq Q$ is a set of final states
- E is a finite set of edges $Q \times (\Sigma \cup \{\epsilon\}) \times W \times Q$.
- λ is a function which assigns weights to each final state

Definition (2)

Paths:

1. for all $q \in Q, (q, \epsilon, 0, q) \in \widehat{E})$
2. for all $(q_0, x, w, q) \in E$: $(q_0, x, w, q) \in \widehat{E}$
3. if (q_0, x_1, w_1, q_1) and (q_1, x_2, w_2, q) are both in $\widehat{E}$ then $(q_0, x_1x_2, w_1 + w_2, q) \in \widehat{E}$

Definition (3)

- The weighted language accepted by M:

$$L(M) = \{(x, w + \lambda(q_f)) | q_s \in S, q_f \in F, (q_s, x, w, q_f) \in \widehat{E}\}$$

Weights (Mohri 1997)

- Various weight structures (*semirings*)
  - probabilities
  - negative logs of probabilities
  - strings
- Various algorithms and properties of transducers generalize

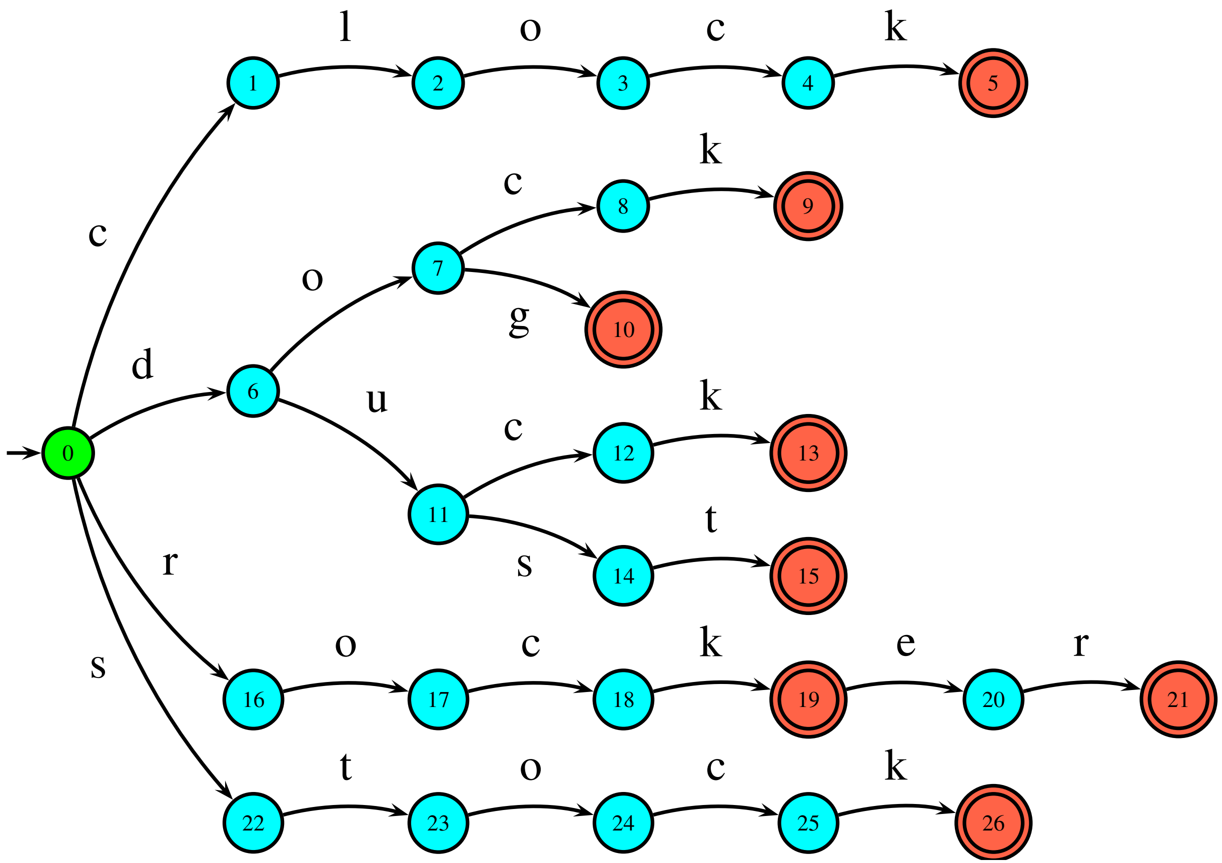
PART 2

- Dictionaries
- Perfect Hash FSA
- Tuple Dictionaries

List of words

```
clock
dock
stock
dog
duck
dust
rock
rocker
```

Trie

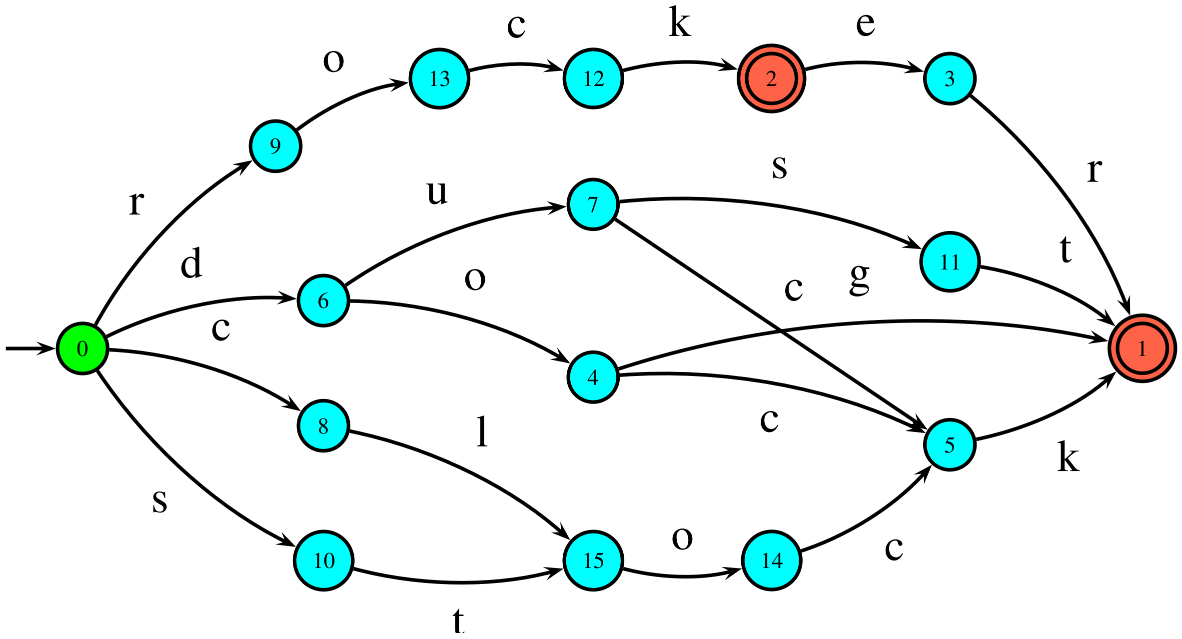


l
o
c
k
k
c
o
g
c
d
u
k
c
t
s
r
o
c
k
e
r
s
t
o
c
k
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

Tries

- Final states can be associated with lexicographic information
- Efficient
- Compact: sharing of identical prefixes
- Can we do better?

Minimize trie

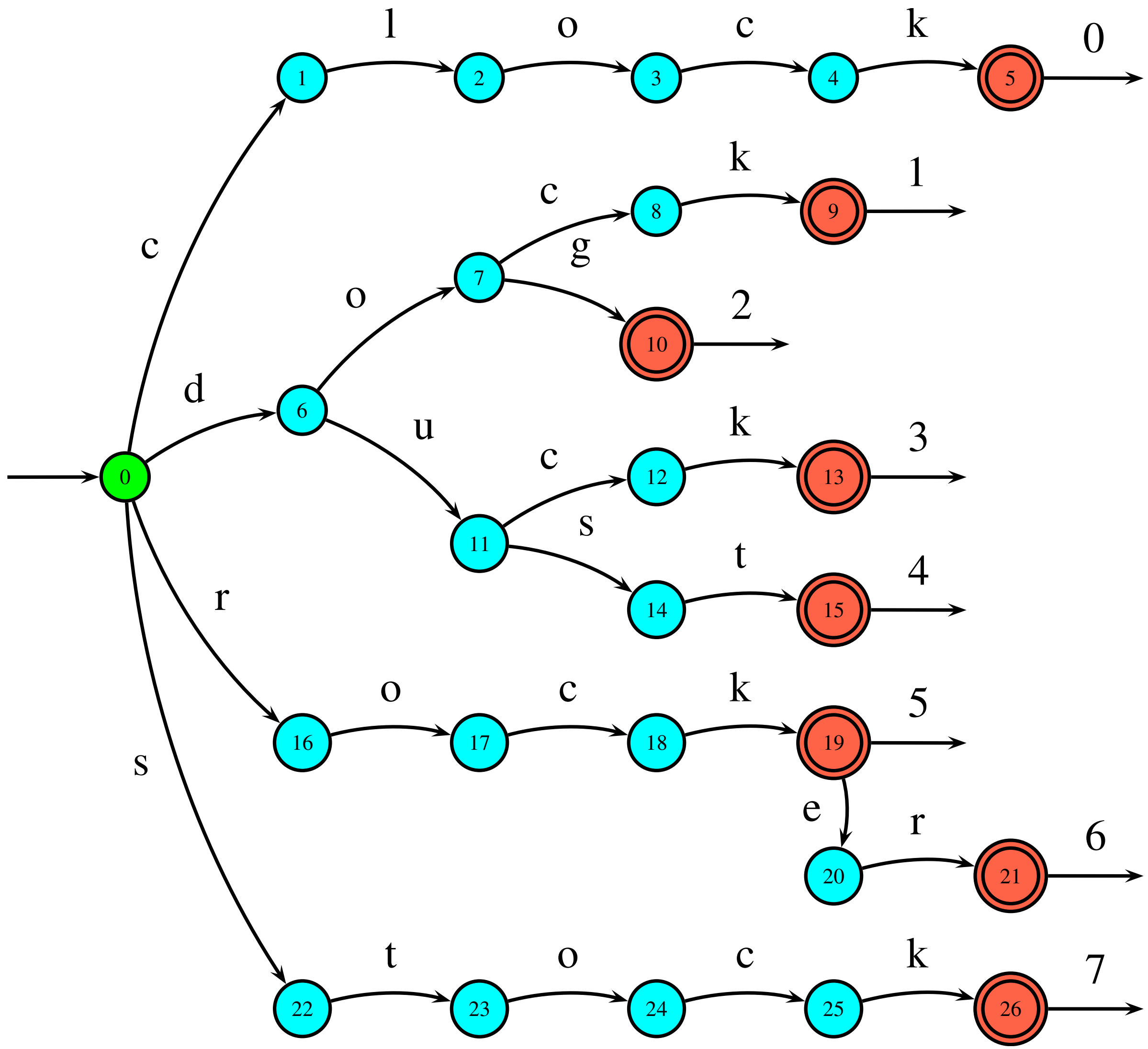


- Smaller
- How to associate lexicographic information?

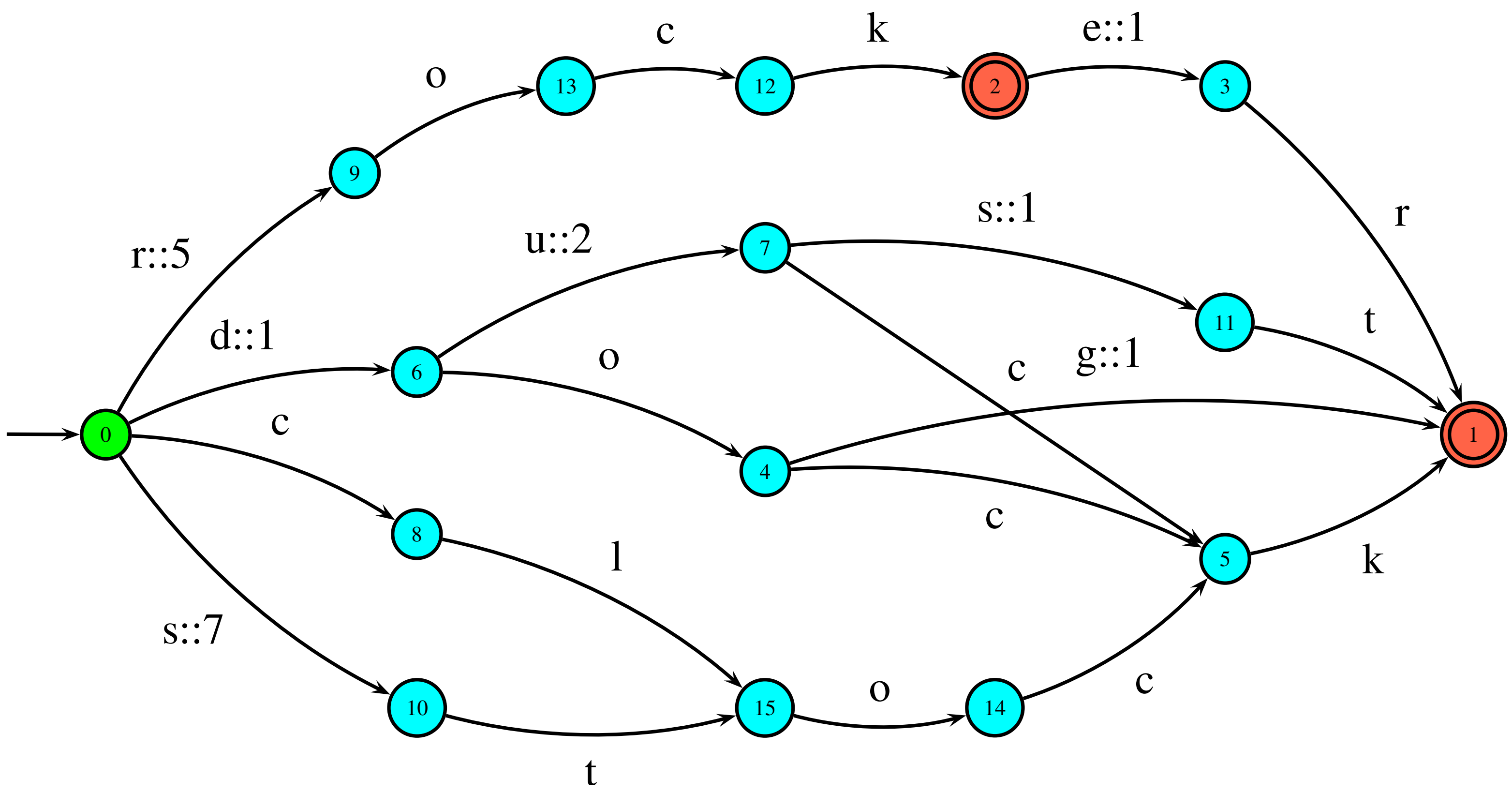
Perfect Hash Finite Automaton

- Assign unique number to each word
- Minimize weighted acceptor

Weighted Trie



Minimized Weighted Trie



Perfect Hash

Elegant way to construct an OPMPHF for a given set of keywords:

- Hash Function: map key to integer
- Perfect: every key is hashed to unique integer
- Minimal: n keys are mapped into range $0 \ldots n-1$
- Order Preserving: alphabetic order of keys is reflected in numeric order of integers

Advantages

- Efficient (optimal)
- Compact (in typical cases less than 10% of standard hashes)
- Order-preserving: application in suffix array construction on words

Incremental Construction

- Construct dictionary from *sorted* list of words
- Construct dictionary from *unsorted* list of words
- Add perfect hash weights directly to minimal automaton

Tuple Dictionaries

- map tuple of keys to some value
- e.g. Ngram language models
- compact representation using perfect hash automata

Motivation

- Collins 1999:
  - ⋆ loading hash table of bigram counts takes 8 minutes!
- Foster 2000:
  - ⋆ Maxent model with 35,000,000 features; each feature is a word pair
- . . .

Example

```
...
the    man    23
the    woman  15
their  man     4
...
```

Tuple Dictionary

- Construct a perfect hash automaton for the keys
- Replace each key with its perfect hash integer

Example

```
...
the    man    23
the    woman  15
their  man     4
...
```

$\Rightarrow$

```
 ...
4112  2008  23
4112  7023  15
4113  2008   4
 ...
```

Tuple Dictionary

- Construct a perfect hash automaton for the keys
- Replace each key with its perfect hash integer
- Determine the maximum integer per column
- Use per column minimal number of bytes (typically: 2, 3 or 4)

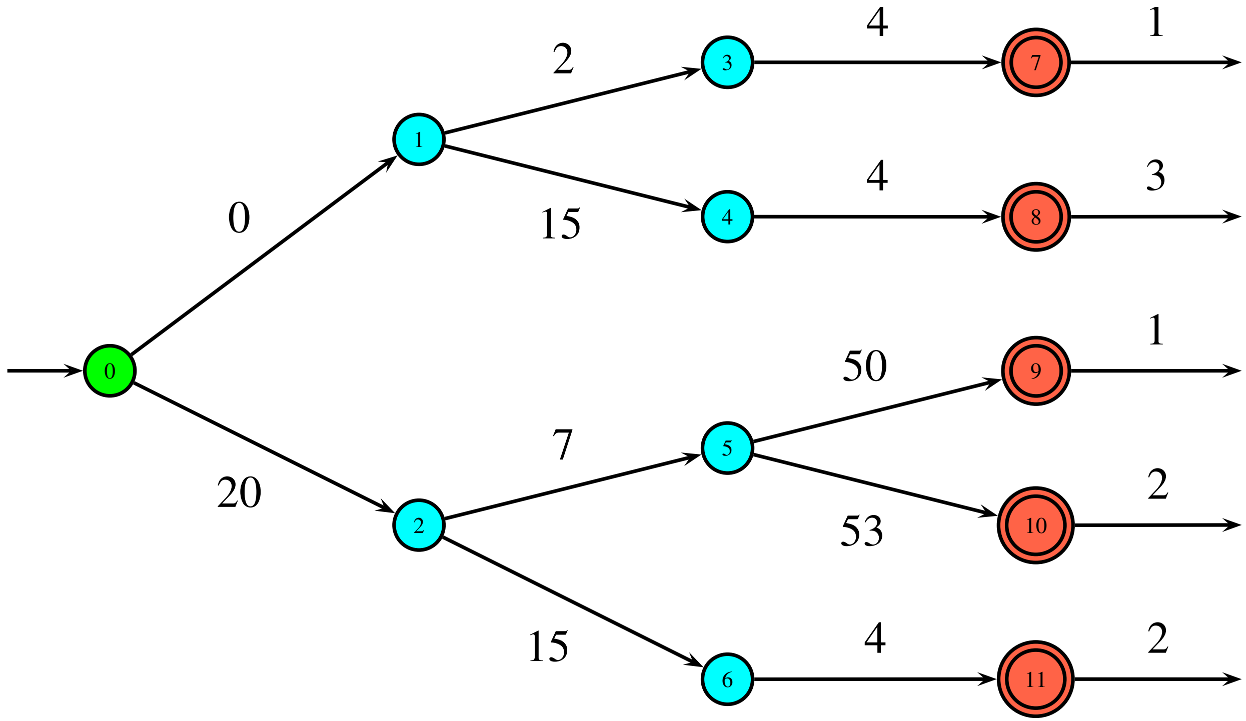
Usage

- For a given tuple: convert keys to integers
- Pack integers into key
- Binary search in tuple dictionary

Variants

- Daciuk and van Noord (2003).

0	2	4	1
0	15	4	3
20	7	50	1
20	7	53	2
20	15	4	2



Experiments

	Mbytes	in	out	elements
40K sents trigram counts	11.6	3	int	552462
40K sents fourgram counts	17.3	4	int	644886
POS-tagger bigram	11.9	2	int	350437
40K sents trigram prob	14.8	3	real	552462

Results (Mbytes)

test set	hash first el Prolog	hash concat C++	fsa concat	table	tree
trigram counts	60.3	52	11.1	4.9	4.3
fourgram counts	85.4	64	20.7	6.9	7.4
bigram POS-tagger	NA	37	4.0	4.2	3.2
fourgram prob	67.1	52	10.5	NA	8.7

Available

- http://www.eti.pg.gda.pl/~jandac/

PART 3: Regular Expressions

- Standard Regular Expressions
- Regular Expressions for Transducers
- Defining Regular Expression Operators

Regular Expressions

- Notation which describes regular languages
- More declarative than automata
- Regular expression compiler takes regular expression and computes corresponding automaton
- FSA Utilities

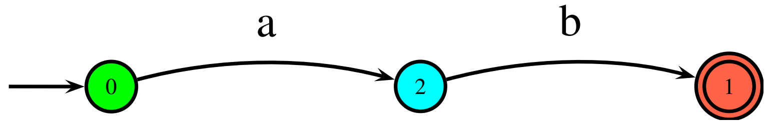
Regular Expression Operators (1)

- An atom a defines the language $\{a\}$.
- The expression {E1,E2} is the union of $L(E1)$ and $L(E2)$
- The expression [E1,E2] is the concatenation of $L(E1)$ and $L(E2)$
- The expression E1* is the Kleene closure of $L(E1)$
- Use (and) for grouping

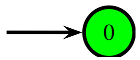
Regular Expression Operators (2)

- The expression `[]` is the language $\{\epsilon\}$
- The expression `{}` is the language $\emptyset$
- What is:

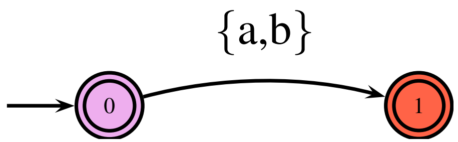
`[a,b,[]]`



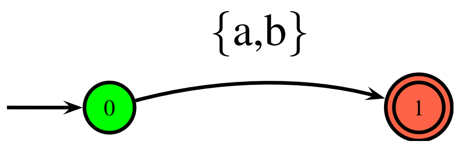
`[a,b,{}]`



`{a,b,[]}`



`{a,b,{}}`



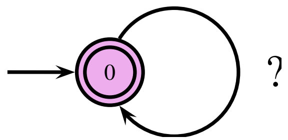
Regular Expression Operators (3)

- Optionality: `E1^`
- Intersection: `E1 & E2`
- Difference: `E1 - E2`
- Complement: `~E1`
- Term Complement: `` `E1 `` is a short-hand for `? - E1`

Regular Expression Operators (4)

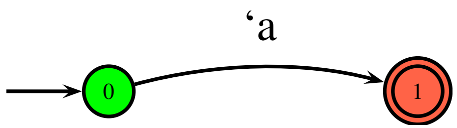
- Meta-symbol `?`: $\{x | x \in \Sigma\}$
- Interval `a..z`: $\{a, \ldots, z\}$
- What is:

`?*`

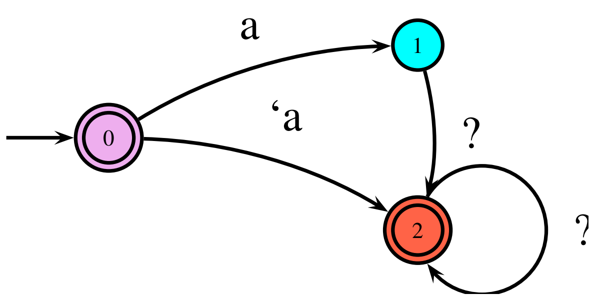


`? - a`

`‘a`



`~a`



What is:

- `~[~{},'a,~{}]`

Operators for Transductions

- cross-product: `E1:E2`
- composition: `E1 o E2`
- union, concatenation, Kleene-closure

Operators for Transductions (2)

- identity: `id(E1)`
- *coercion*
- `[a,b,c:[],d]` $\Longrightarrow$ `[id(a),id(b),c:[],id(d)]`
- What is: `? : ?`
- What is: `id(?)`
- Compare: `[?*,d:e]`

Operators for Transductions (3)

- `domain(E)`
- `range(E)`
- `inverse(E)`

Operators for Transductions (4)

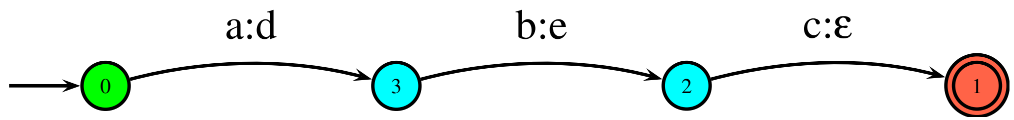
- `replace(T)`
- `replace(T,Left,Right)`

Replacement

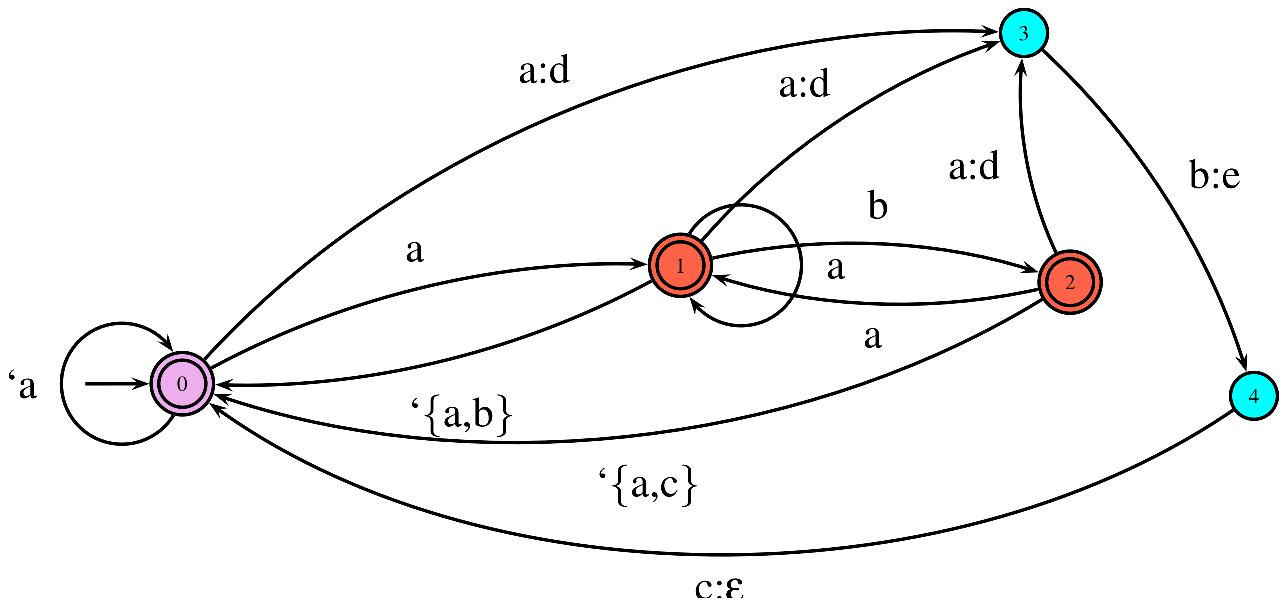
- Apply a given transduction everywhere (in context)
- Many variants possible
- Kaplan & Kay (1994); Karttunen (1995, 1996, 1997); Kempe & Karttunen (1996); Mohri & Sproat (1996); Gerdemann & van Noord (1999)
- implementation in FSA by Yael Cohen-Sygal `www.cl.haifa.ac.il`

Replacement (2)

- `[a,b,c]:[d,e]`



- `replace([a,b,c]:[d,e])`



Application: Soundex algorithm

- Soundex: algorithm to map proper names to codes
- Intention: similar names map to the same code
- Can be encoded by regular expression (Karttunen)

Soundex (2)

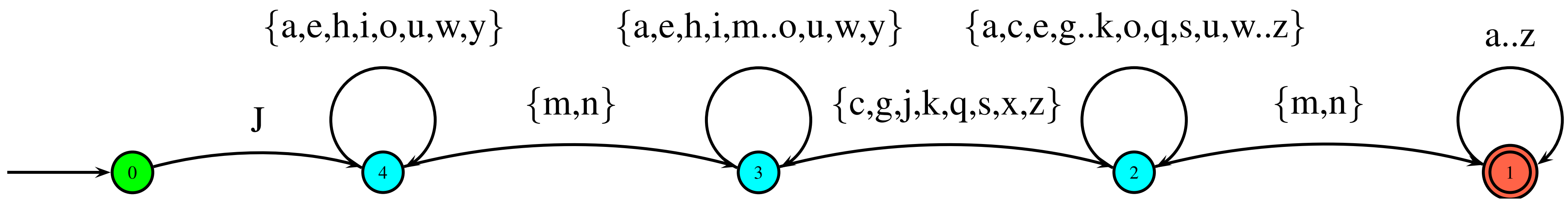
- retain the first letter
- drop all occurrences of a, e, h, i, o, u, w, y
- assign numbers to letters:
  - ⋆ b, f, p, v $\longrightarrow$ 1
  - ⋆ c, g, j, k, q, s, x, z $\longrightarrow$ 2
  - ⋆ d, t $\longrightarrow$ 3
  - ⋆ l $\longrightarrow$ 4
  - ⋆ m, n $\longrightarrow$ 5
  - ⋆ r $\longrightarrow$ 6
- map adjacent identical codes to single code
- convert to letter followed by three digits

Soundex (3)

```
[? ,  replace({a,e,h,i,o,u,w,y}:[])
                  o
      replace({     {b,f,p,v}+          : 1,
                 {c,g,j,k,q,s,x,z}+     : 2,
                       {d,t}+           : 3,
                        l+              : 4,
                      {m,n}+            : 5,
                        r+              : 6  })
                  o
             [?*, []:0*]
                  o
            [?,?,?,?:[]*]                  ]
```

Soundex (4)

- *Johnson* $\rightarrow$ J525; *Johanson* $\rightarrow$ J525; *Jackson* $\rightarrow$ J250
- But also: construct automaton recognizing all names that have code J525!



Defining Regular Expression Operators

- For patterns that occur over and over again, you can define your own operators.

```
macro(vowel,          {a,e,i,o,u}).

macro(contains(X),   [?*, X, ?*]).
```

- New operators can be used in the definition of additional operators

```
macro(free(X),        ~contains(X)).
```

Example: longest (Gerdemann)

- longest(A): the set of longest strings from A

```
macro(longest(A),      A - shorter(A)                      ).

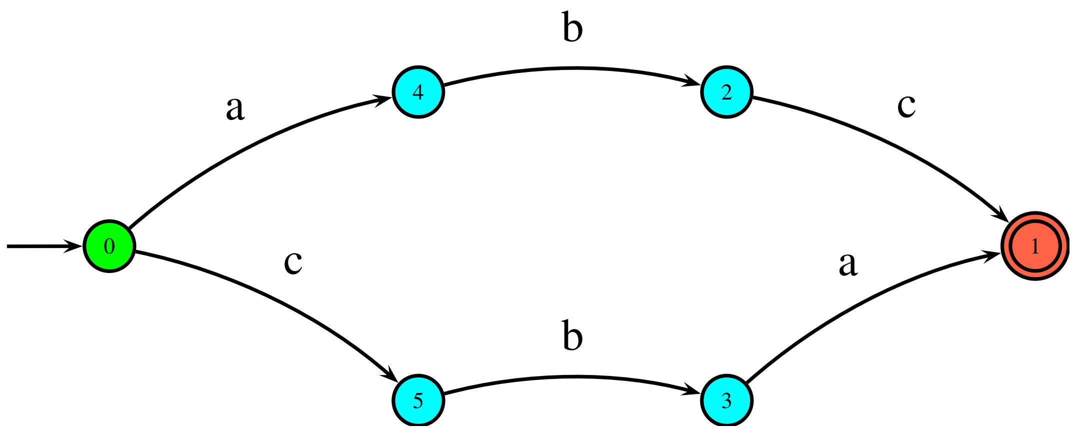
macro(shorter(A),      range(same_length(A) o shorten_t )).

macro(same_length(A), range(A o ?:?*)                     ).

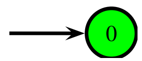
macro(shorten_t,       [?*, ?:[]+]                         ).
```

Example: longest (2)

- `longest({[a],[a,b],[b,a],[a,b,c],[c,b,a]})`



- `longest({a,b*,[c,d],[e,f]})`



Various Applications

- Bouma: Hyphenation
- Vaillette: Monadic Second Order Logic
- Malouf: Two-level Morphology
- Walther: One-level Morphology
- Malouf: tokenizer for WSJ
- Bouma: Grapheme to Phoneme Conversion
- Kiraz: multi-tape automata for Syriac and Arabic

Application

- Set of regular expression operator definitions
- Compile regular expression into automaton
- Compile automaton into efficient program (C, C++, Java, Prolog)

Application: Example

```
% fsa write=c -aux s2p.pl -r s2p > s2p.c

% cc s2p.c -o s2p
% echo "ik ga naar de blauwe schuit in leuven" | s2p
Ik xa nar d@ blMw@ sxLt In l|v@
%
```

PART 4

- Finite State Optimality Phonology
  - ⋆ Prince & Smolensky (1993)
  - ⋆ Frank & Satta (1998)
  - ⋆ Karttunen (1998)
  - ⋆ Gerdemann & van Noord (2000)
  - ⋆ Jäger (2001, 2003)
  - ⋆ Eisner (1997, 2000, 2002)

Optimality Theory

- Prince and Smolensky (1993)
- No rules
- Instead:

1. Universal function *Gen*
2. Set of ranked universal violable *constraints*

Syllabification: Gen

- *Input*: sequences of consonants and vowels
- *Gen*: assigns structure: sequence of syllables, such that
  - ⋆ optional onset, followed by nucleus, followed by optional coda
  - ⋆ onset and coda each contain an optional consonant
  - ⋆ nucleus contains an optional vowel
- Furthermore, certain consonants and vowels can be unparsed

Syllabification: Gen (2)

- *Gen(a)*:

`N[a]`	`N[a]N[]`
`N[a]D[]`	`N[]N[a]`
`N[]N[a]N[]`	`N[]N[a]D[]`
`N[]X[a]`	`N[]X[a]N[]`
`N[]X[a]D[]`	`O[]N[a]`
`O[]N[a]N[]`	`O[]N[a]D[]`
`O[]X[a]N[]`	`X[a]N[]`

Phonetic Realization

- Unparsed: not phonetically realized (*deletion*)
- Empty segment: phonetically realized by filling in default featural values (*epenthesis*)

Constraints

HaveOns Syllables must have onsets

NoCoda Syllables must not have codas

Parse Input segments must be parsed

FillNuc A nucleus position must be filled

FillOns An onset position must be filled

Constraints

- Universal
- Ranked
- Violable

Constraint Order

HaveOns $\gg$ NoCoda $\gg$ FillNuc $\gg$ Parse $\gg$ FillOns

OT Tableaux

	Candidate	HaveOns	NoCoda	FillNuc	Parse	FillOns
	N[a]	*!				
	N[a]N[]	*!				
	N[a]D[]	*!				
	N[]X[a]N[]	*!				
	N[]X[a]D[]	*!				
$\Longrightarrow$	O[]N[a]					*
	O[]N[a]N[]			*!		
	O[]N[a]D[]		*!			
	O[]X[a]N[]			*!		
	X[a]N[]	*!				

Finite-state Implementation

- Karttunen 1998
- *Gen* is a finite state transducer
- Each of the constraints is a finite state automaton
- *Lenient Composition*

Finite-state Implementation (2)

- Rewrite Rules $\Longrightarrow$ finite-state transducer
- Two-level Rules $\Longrightarrow$ finite-state transducer
- OT Constraints $\Longrightarrow$ finite-state transducer
- Constraint ranking vs. Rule ordering

Implementing Gen

```
macro(o_br,     'O['). % onset
macro(n_br,     'N['). % nucleus
macro(d_br,     'D['). % coda
macro(x_br,     'X['). % unparsed
macro(r_br,     ']').
macro(br, {o_br,n_br,d_br,x_br,r_br}).

macro(onset,    [o_br,cons^  ,r_br]).
macro(nucleus, [n_br,vowel^ ,r_br]).
macro(coda,     [d_br,cons^  ,r_br]).
macro(unparsed,[x_br,letter ,r_br]).
```

Implementing Gen (2)

```
macro(gen,        {cons,vowel}*
                        o
         insert_each_pos([{o_br,d_br,n_br},r_br]^)
                        o
                      parse
                        o
     ignore([onset^,nucleus,coda^],unparsed)*      ).

macro(parse, replace([[]:{o_br,d_br,x_br},cons, []:r_br])
                                  o
             replace([[]:{n_br,x_br},     vowel,[]:r_br])).

macro(insert_each_pos(E), [[ []:E, ?]*,[]:E]).
```

Implementing Constraints

```
macro(no_coda,  free(d_br)).

macro(parsed,   free(x_br)).

macro(fill_nuc, free([n_br, r_br])).

macro(fill_ons, free([o_br, r_br])).

macro(have_ons, ~[~[?*,onset],nucleus,?*] ).
```

Merciless Cascade

```
    gen
     o
have_ons
     o
 no_coda
     o
fill_nuc
     o
 parsed
     o
fill_ons
```

Lenient Composition!

```
macro(lenient_composition(I,C),

         { I o C, ~domain(I o C) o I } ).
```

I	C	I o C	~domain(I o C) o I	lc
a:b	b:b	a:b	d:d	a:b
b:b	e:e	b:b		b:b
c:d		c:e		c:e
c:e		e:e		e:e
d:d				d:d
e:e				

Putting it Together

```
   gen
    lc
have_ons
    lc
 no_coda
    lc
fill_nuc
    lc
 parsed
    lc
fill_ons
```

Problem: Constraints with Multiple Violations

```
O[b]N[e]O[b]N[o]X[p]
O[b]N[e]X[b]O[]N[o]X[p]
O[b]N[e]X[b]X[o]X[p]
X[b]O[]N[e]O[b]N[o]X[p]
X[b]O[]N[e]X[b]O[]N[o]X[p]
X[b]O[]N[e]X[b]X[o]X[p]
X[b]X[e]O[b]N[o]X[p]
X[b]X[e]X[b]O[]N[o]X[p]
```

Counting: separate constraint for each count

```
gen
lc
have_ons
lc
no_coda
lc
fill_nuc
lc
parsed2
lc
parsed1
lc
parsed0
lc
fill_ons
```

Counting (2)

- Is 2 good enough? Only for strings of length ≤ 6
- Is 5 good enough? Only for strings of length ≤ 9
- There is no bound to the length of a word . . .

Some OT analyses are not finite state

- Frank and Satta (due to Smolensky, after an idea by Hiller)
- Inputs: `[a*,b*]`
- Gen: map all `a`'s to `b`'s and all `b`'s to `a`'s; or map all `b`'s to `b`'s and all `a`'s to `a`'s
- Constraint: no `a`'s

Some OT analyses are not finite state (2)

- maps $a^n b^m$ to
  - ⋆ $\{b^n a^m\}$ if $n < m$
  - ⋆ $\{a^n b^m\}$ if $n > m$
  - ⋆ $\{b^n a^m, a^n b^m\}$ if $n = m$
- if we intersect range of this mapping with `[a*,b*]` then we have $\{a^n b^m\}$ where $n \geq m$.
- This language is known to be non-regular

Finite State OT: A New Approach

- counting
- matching

1. More Accurate
2. More Compact
3. More Efficient

Idea

- Candidates
- Alternatives is the set you can construct by *introducing* further constraint violations in Candidates
- Compose Candidates with complement(Alternatives)

More specifically

- Introduce a marker for each constrain violation
- Construct a filter which maps marked-up candidates to alternatives which have at least one marker more
- The range of this mapping is the Alternatives set
- Compose candidates with complement of Alternatives

Marking Constraints

- use @ to indicate a constraint violation

- `macro(mrk, @).`

```
macro(mark_v(parse),    replace([]:mrk,x_br,[]).
macro(mark_v(no_coda),  replace([]:mrk,d_br,[]).
macro(mark_v(fill_nuc), replace([]:mrk,[n_br,r_br],[])).
macro(mark_v(fill_ons), replace([]:mrk,[o_br,r_br],[])).

macro(mark_v(have_ons),
   replace([]:mrk,[],n_br)  o  replace(mrk:[],onset,[])).
```

Marking Constraints: Example

```
• c1:   O[ b ] N[ e ] X[ b ] X[ o ] X[ p ]
  c2:   O[ b ] N[ e ] O[ b ] N[ o ] X[ p ]
  c3:   X[ b ] X[ e ] O[ b ] N[ o ] X[ p ]

• c1: O[   b ] N[   e ] X[ @ b ] X[ @ o ] X[ @ p ]
  c2: O[   b ] N[   e ] O[   b ] N[   o ] X[ @ p ]
  c3: X[ @ b ] X[ @ e ] O[   b ] N[   o ] X[ @ p ]
```

Constructing Alternatives

- Ignore everything except *input* and *marker*

```
  b   e @ b @ o  @  p
  b   e   b   o  @  p
@ b @ e   b   o  @  p
```

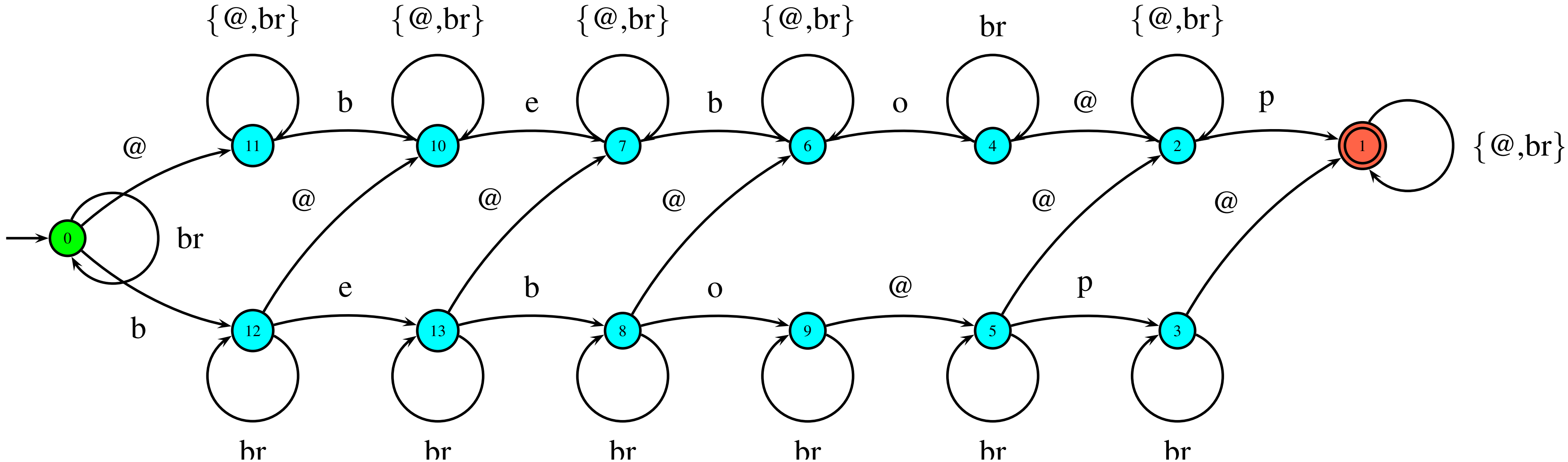
- Insert at least one additional marker:

```
[[?*,[]:mrk ]+, ?*]
```

- Insert brackets arbitrarily:

```
{[]:br, ‘br}*
```

Alternatives



Filter (2)

- candidates:

```
c1: O[   b ] N[   e ] X[ @ b ] X[ @ o ] X[ @ p ]
c2: O[   b ] N[   e ] O[   b ] N[   o ] X[ @ p ]
c3: X[ @ b ] X[ @ e ] O[   b ] N[   o ] X[ @ p ]
```

- note: c1 and c3 are in Alternatives

Optimality Operator

```
macro(Cands oo Constraint,
                  Cands
                    o
             mark_v(Constraint)
                    o
~ range( Cands o mark_v(Constraint) o add_violation  )
                    o
              {mrk:[],‘mrk}* ).
```

Example

```
macro(gen oo have_ons,
                    gen
                     o
              mark_v(have_ons)
                     o
~ range( gen o mark_v(have_ons) o add_violation  )
                     o
              {mrk:[],‘mrk}* ).
```

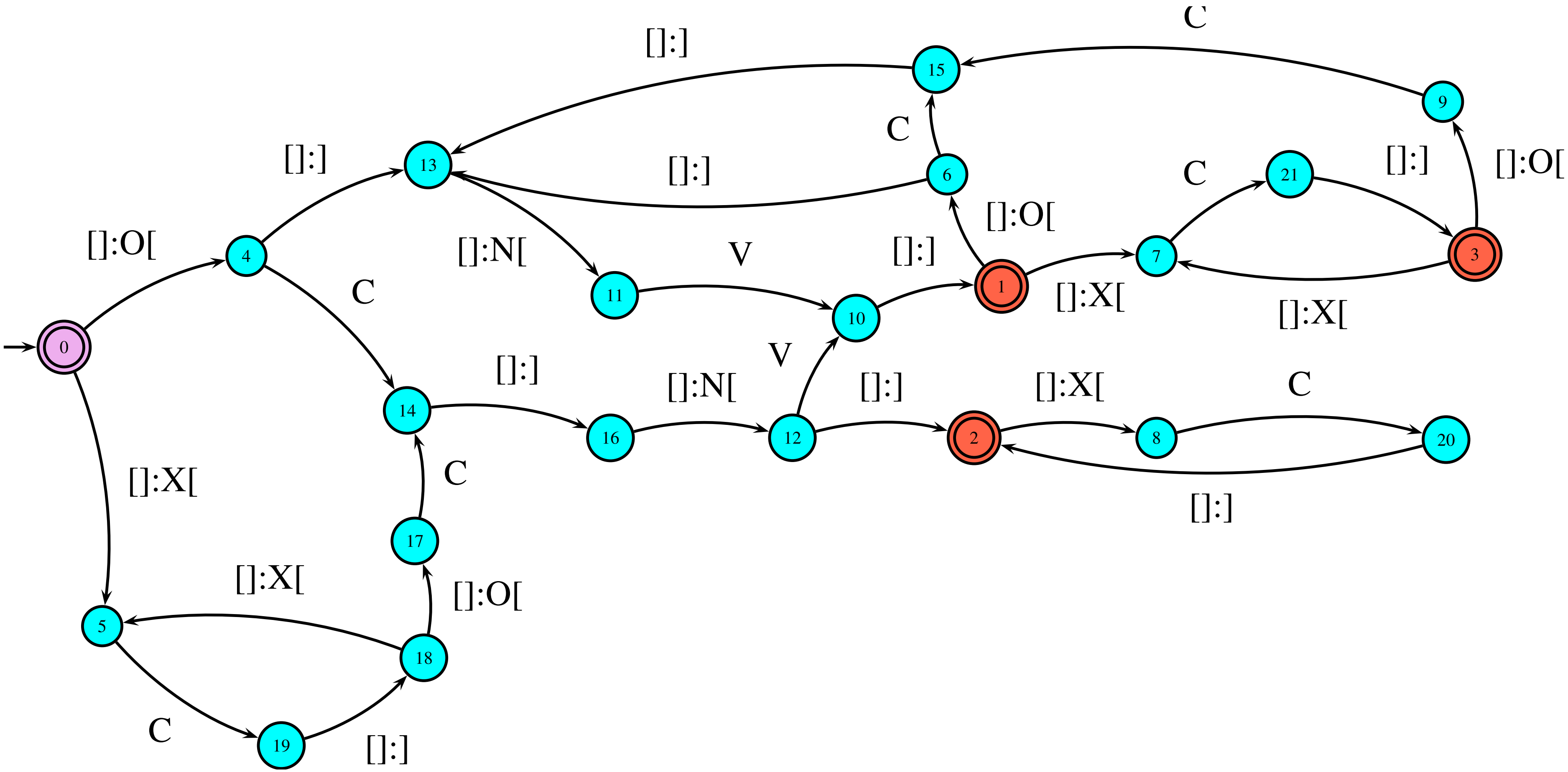
Add Violation

```
macro(add_violation,
      {br:[], `br}*         % delete brackets
             o
     [[?*,[]:mrk]+, ?*]   % add at least one @
             o
      {[]:br, `br}*         % reinsert brackets
     ).
```

Syllabification again

```
gen
oo
have_ons
oo
no_coda
oo
fill_nuc
oo
parsed
oo
fill_ons
```

Result



Properties

- 22 states!
- Exact!!
- 1 CPU second to compute

Not always exact

$$Parse \gg FillOns \gg HaveOns \gg FillNuc \gg NoCoda$$

`N[a]D[r]O[t]N[]D[s]` (art@s)

`N[a]O[r]N[]D[t]O[s]N[]` (ar@ts@)

Permutation

- Matching works as long as violations 'line up'
- Permutation in the filter to make them line up
- ```
  macro(permute_marker,
    [{[?*,mrk:[],?*,[]:mrk],
      [?*,[]:mrk,?*,mrk:[]]}*,?*]).
  ```
- More permutation for more precision
- Strictly more powerful than 'counting'
  ```

Optimality Operator (2)

```
macro(Cands oo Prec :: C),
                      Cands
                        o
                    mark_v(C)
                        o
~ range(Cands o mark_v(C) o add_violation(Prec) )
                        o
                  { mrk:[], `mrk }*  ).
```

Add Violations with Permutation

```
macro(add_violation(3),
                {br:[], `br}*
                      o
              [[?*,[]:mrk]+, ?*]
                      o
                 permute_marker
                      o
                 permute_marker
                      o
                 permute_marker
                      o
                {[]:br, `br}*  ).
```

Nine Constraint Orderings

id	constraint order
1	have_ons $\gg$ fill_ons $\gg$ no_coda $\gg$ fill_nuc $\gg$ parse
2	have_ons $\gg$ no_coda $\gg$ fill_nuc $\gg$ parse $\gg$ fill_ons
3	no_coda $\gg$ fill_nuc $\gg$ parse $\gg$ fill_ons $\gg$ have_ons
4	have_ons $\gg$ fill_ons $\gg$ no_coda $\gg$ parse $\gg$ fill_nuc
5	have_ons $\gg$ no_coda $\gg$ parse $\gg$ fill_nuc $\gg$ fill_ons
6	no_coda $\gg$ parse $\gg$ fill_nuc $\gg$ fill_ons $\gg$ have_ons
7	have_ons $\gg$ fill_ons $\gg$ parse $\gg$ fill_nuc $\gg$ no_coda
8	have_ons $\gg$ parse $\gg$ fill_ons $\gg$ fill_nuc $\gg$ no_coda
9	parse $\gg$ fill_ons $\gg$ have_ons $\gg$ fill_nuc $\gg$ no_coda

Experiments

- A permutation of at most 1 is required
- Compact automata
- Fast automata construction

Size of Automata

	Prec	Constraint order								
		1	2	3	4	5	6	7	8	9
match	exact	29	22	20	17	10	8	28	23	20
count	≤ 5	95	220	422	167	10	240	1169	2900	4567
count	≤ 10	280	470	1667	342	10	420	8269	13247	16777
count	≤ 15	465	720	3812	517	10	600	22634	43820	50502

Speed of Construction

	Prec	Constraint order								
		1	2	3	4	5	6	7	8	9
match	exact	1.0	0.9	0.9	0.9	0.8	0.7	1.5	1.3	1.1
count	≤ 5	0.9	1.7	4.8	1.6	0.5	1.9	10.6	18.0	30.8
count	≤ 10	2.8	4.7	28.6	4.0	0.5	4.2	83.2	112.7	160.7
count	≤ 15	6.8	10.1	99.9	8.6	0.5	8.2	336.1	569.1	757.2

Determining Exactness

- Assume T is a correct implementation of some OT analysis, except that it fails to distinguish different numbers of constraint violations for one or more constraints
- We can check this for each of the constraints

Determining Exactness (2)

- If T is not exact wrt to constraint C, then the following must be ambiguous:

```
      T
      o
   mark_v(C)
      o
 {‘mrk:[], mrk}*
```

- there is an algorithm to determine if a given transducer is functional

Harmony ordering

- A constraint imposes *harmony ordering* on the set of candidates
- In classical OT: *counting*
- Proposal: harmony ordering must be *regular relation*

Harmony ordering as a regular relation

- $>$ is the harmony ordering (partial order)
- harmony ordering should only order candidates with identical input
- $y > y'$ indicates that y is more harmonic than y'
- we require that there is a regular relation $R = \{(y, y')|y > y'\}$.
- if this condition is met, the resulting OT is regular (Jäger 2001, 2003; Eisner 2002)

Multiple Violations

- Some constraints are violated multiple times, i.e., at multiple locations. Typically, harmony ordering is regular.

- Some constraints are violated *gradiently*, i.e., different degrees of violation.

Gradient constraints

- constraints with bounded number of degrees of violation (can be thougt of as a series of non-gradient constraints)
- *horizontal* gradience: degree of violation proportional to some distance in strings
- McCarthy (2002) claims that the latter type of constraints should not be in OT
- Eisner (1997) and Birot (2003) show that the latter type of constraints might impose non-regular harmony ordering

Example: All-Feet-Left (Tesar and Smolensky (2000))

- Context: analysis of metrical stress
  - ⋆ some syllables are organized into *feet*
  - ⋆ prosodic word consists of those feet as well as other syllables
  - ⋆ each foot has a head syllable
  - ⋆ each word has a head foot
  - ⋆ head syllable of head foot receives primary stress
  - ⋆ other head syllables receive secondary stress
- $\sigma(\sigma\sigma 2)[\sigma 1\sigma]\sigma(\sigma 2)$

All-Feet-Left (2)

- various constraints which determine analysis of syllables into feet
- All-Feet-Left: assigns to each foot f as many violation marks as the number of syllables intervening between the left edge of the word and the left edge of f.

All-Feet-Left (3)

- $[\sigma\sigma](\sigma\sigma)(\sigma\sigma)$: 0+2+4 violations
- $[\sigma](\sigma)(\sigma)(\sigma)(\sigma)(\sigma)$: 0+1+2+3+4+5 violations
- In general, can assign a quadratic number of violations
- Birot 2003: such a harmony ordering cannot be described by regular relation

Finite State OT: Summary

- Phonological relations are (mostly) finite-state
- OT phonology is finite state provided:
  - ⋆ Gen is regular relation
  - ⋆ Each of the constraints is regular
  - ⋆ The harmony ordering is regular