Multi-Input Strictly Local Functions for Templatic Morphology

Hossep Dolatian
Jonathan Rawski

Dept. of Linguistics
Institute for Advanced Computational Science
Stony Brook University

Jan 4 2020

Table of Contents

Multi-Input Strictly Local Functions for Templatic Morphology

Explaining the title

1. **Semitic** templates
2. **Strict Locality** in templates
3. **Computing** templates as multi-string function

Link: multi-tape transducers

Introduction: Semitic templates

- Most languages have concatenative morphology
  - *hold* + ing

[1](McCarthy, 1981)

Introduction: Semitic templates

- Most languages have concatenative morphology
  - *hold* + ing
- Semitic has templatic morphology[1]

[1](McCarthy, 1981)

Introduction: Semitic templates

- Most languages have concatenative morphology
  - *hold* + ing
- Semitic has templatic morphology[1]

Active verbs
katab 'it wrote'

[1] (McCarthy, 1981)

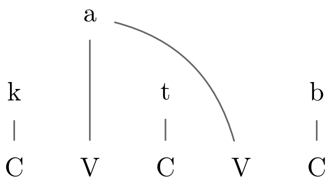
Introduction: Semitic templates

- Most languages have concatenative morphology
  - *hold* + ing
- Semitic has templatic morphology[1]

Active verbs

katab 'it wrote'

1. Inflectional V: *a*
2. Root C: *ktb*
3. Template T: *CV.CVC*



[1] (McCarthy, 1981)

Introduction: Semitic templates

- Most languages have concatenative morphology
  - *hold* + ing
- Semitic has templatic morphology[1]

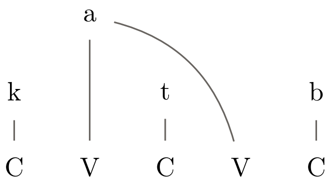
Active verbs

katab 'it wrote'

Passive verbs

kutib 'it was written'

1. Inflectional V: *a*
2. Root C: *ktb*
3. Template T: *CV.CVC*



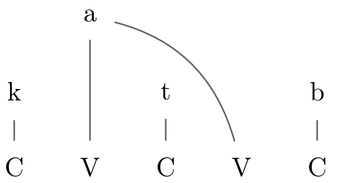
[1] (McCarthy, 1981)

Introduction: Semitic templates

- Most languages have concatenative morphology
  - *hold* + ing
- Semitic has templatic morphology[1]

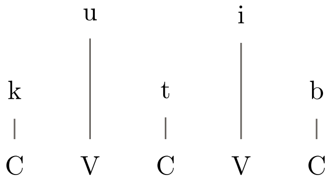
Active verbs
katab 'it wrote'

1. Inflectional V: *a*
2. Root C: *ktb*
3. Template T: *CV.CVC*



Passive verbs
kutib 'it was written'

1. Inflectional V: *ui*
2. Root C: *ktb*
3. Template T: *CV.CVC*



- Looks non-local..

[1] (McCarthy, 1981)

Introduction: locality

- Bulk of natural language processes are local

Introduction: locality

- Bulk of natural language processes are local
  - = uses *bounded* finite window

Introduction: locality

- Bulk of natural language processes are local
  - = uses *bounded* finite window
- Different domains and languages

Introduction: locality

- Bulk of natural language processes are local
  = uses *bounded* finite window
- Different domains and languages

	Cross-linguistically	Semitic
Allomorphy	✓ Embick (2010)	✓ Kastner (2016)

Introduction: locality

- Bulk of natural language processes are local
  = uses *bounded* finite window
- Different domains and languages

	Cross-linguistically		Semitic	
Allomorphy	✓	Embick (2010)	✓	Kastner (2016)
Morpho-semantics	✓	Marantz (2013)	✓	Arad (2003)

Introduction: locality

- Bulk of natural language processes are local
  - = uses *bounded* finite window
- Different domains and languages

	Cross-linguistically		Semitic	
Allomorphy	✓	Embick (2010)	✓	Kastner (2016)
Morpho-semantics	✓	Marantz (2013)	✓	Arad (2003)
Morpho-phonology	✓	Chandlee (2014)	?	

Introduction: locality

- Bulk of natural language processes are local
  - = uses *bounded* finite window
- Different domains and languages

	Cross-linguistically		Semitic	
Allomorphy	✓	Embick (2010)	✓	Kastner (2016)
Morpho-semantics	✓	Marantz (2013)	✓	Arad (2003)
Morpho-phonology	✓	Chandlee (2014)	?	us ☺

- Our questions:
  - How do you compute nonconcatenative morphology (=templates)

Introduction: computation

How do you compute morphology?

Concatenative morphology

- *hold* → *hold-ing*

- Easy to compute
  - Single-tape FST (1T-FST)

Introduction: computation

How do you compute morphology?

Concatenative morphology

- *hold* → *hold-ing*

- Easy to compute
    - Single-tape FST (1T-FST)
    - Computationally local

Introduction: computation

How do you compute morphology?

Concatenative morphology

- *hold* → *hold-ing*

- Easy to compute
  - Single-tape FST (1T-FST)
  - Computationally local

Templatic morphology

- ***kutib*** 'to be written'

- How to compute?
  - 1T-FSTs aren't for non-linearity
  - Unknown locality

Contribution

- Show template filling in Semitic is *computationally* local

[2](Bat-El, 2011; Ussishkin, 2011)

Contribution

- Show template filling in Semitic is *computationally* local
  - Locality depends on your computational machinery
  - use Multi-Tape FST

[2] (Bat-El, 2011; Ussishkin, 2011)

Contribution

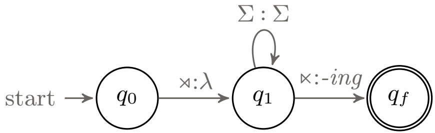
- Show template filling in Semitic is *computationally* local
  - Locality depends on your computational machinery
  - use Multi-Tape FST
- Locality in MT-FST regardless if
  1. Template is phonologically emergent, not a morphological primitive
  2. Words are derived from other words[2]
  3. Domain is infinite or finite language

[2] (Bat-El, 2011; Ussishkin, 2011)

Table of Contents

Computational formalisms

- Single-tape FST
  - read input as linear string



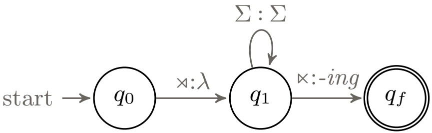
  - Most common formalism in CL + NLP[3]

---

[3](Mohri, 1997)
[4](Bird und Ellison, 1994; Beesley und Karttunen, 2003),...
[5](Kay, 1987; Kiraz, 2001)

Computational formalisms

- Single-tape FST
  - read input as linear string



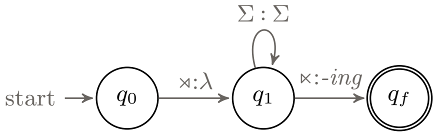
  - Most common formalism in CL + NLP[3]
  - Many computational formalisms exist for Semitic templates over single-tape FST[4]

---

[3](Mohri, 1997)
[4](Bird und Ellison, 1994; Beesley und Karttunen, 2003),...
[5](Kay, 1987; Kiraz, 2001)

Computational formalisms

- Single-tape FST
  - read input as linear string



  - Most common formalism in CL + NLP[3]
  - Many computational formalisms exist for Semitic templates over single-tape FST[4]
- Focus on Multi-Tape FSTs
  - Input is multiple items that are read together
  - Early and intuitive model for Semitic[5]

---

3 (Mohri, 1997)
4 (Bird und Ellison, 1994; Beesley und Karttunen, 2003),...
5 (Kay, 1987; Kiraz, 2001)

Multi-Tape transducers

What is a Multi-Tape transducer (MT FST)?

Multi-Tape transducers

What is a Multi-Tape transducer (MT FST)?

- *Multiple* input tapes, one output tape

Multi-Tape transducers

What is a Multi-Tape transducer (MT FST)?

- *Multiple* input tapes, one output tape
- Advance on every input tape either
  - **synchronously** at the same time *or*
  - **asynchronously** at different times

What does an Arabic MT-FST look like?

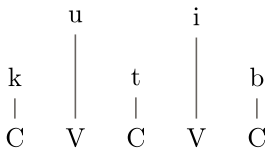
Multi-Tape transducers

What is a Multi-Tape transducer (MT FST)?

- *Multiple* input tapes, one output tape
- Advance on every input tape either
  - **synchronously** at the same time *or*
  - **asynchronously** at different times

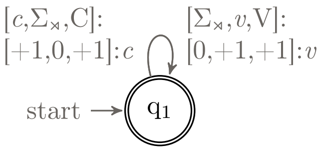
What does an Arabic MT-FST look like?

- Morphology has 3 input items
  1. Inflectional V: *ui*
  2. Root C: *ktb*
  3. Template T: *CV.CVC*



- MT-FST has 3 tapes
- Move over each tape
- Create only one output symbol

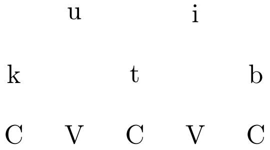
  [*input*]: [*direction*]: *output*



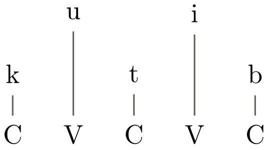
Illustrating MT FSTs

Working example:

Input: 3 tapes



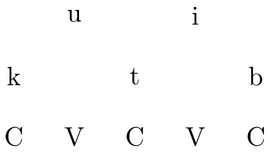
Output: filled template



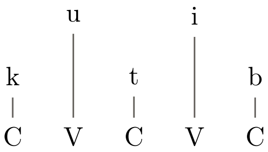
Illustrating MT FSTs

Working example:

Input: 3 tapes



Output: filled template

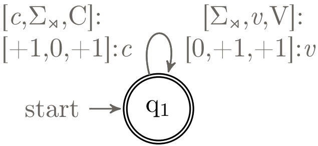


MT-FST implementation

Input:

C:	k	t	b		
V:	u	i			
T:	C	V	C	V	C

Output:



Illustrating MT FSTs

Working example:

Input: 3 tapes

	u		i	
k		t		b
C	V	C	V	C

Output: filled template

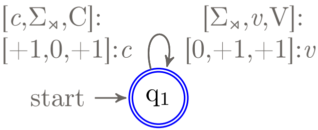
	u		i	
k		t		b
C	V	C	V	C

MT-FST implementation

Input:

C:	k	t	b		
V:	u	i			
T:	C	V	C	V	C

Output:



Illustrating MT FSTs

Working example:

Input: 3 tapes

	u		i	
k		t		b
C	V	C	V	C

Output: filled template

	u		i	
k		t		b
C	V	C	V	C

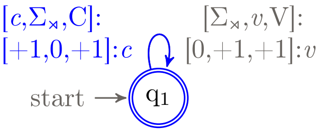
MT-FST implementation

Input:

C:	k	t	b		
V:	u	i			
T:	C	V	C	V	C

Output:

k



Illustrating MT FSTs

Working example:

Input: 3 tapes

	u		i	
k		t		b
C	V	C	V	C

Output: filled template

	u		i	
k		t		b
C	V	C	V	C

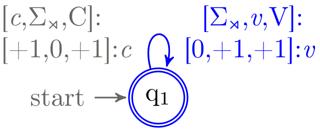
MT-FST implementation

Input:

C:	k	t	b		
V:	u	i			
T:	C	V	C	V	C

Output:

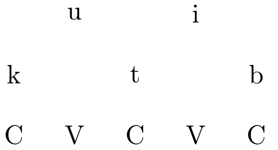
k u



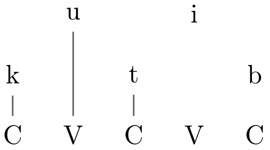
Illustrating MT FSTs

Working example:

Input: 3 tapes



Output: filled template



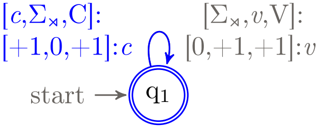
MT-FST implementation

Input:

C:	k	t	b		
V:	u	i			
T:	C	V	C	V	C

Output:

k u t

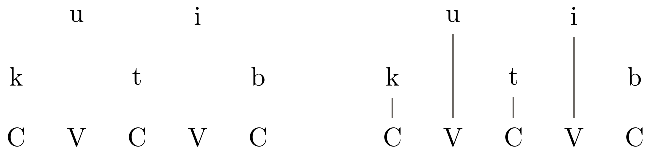


Illustrating MT FSTs

Working example:

Input: 3 tapes

Output: filled template



MT-FST implementation

Input:

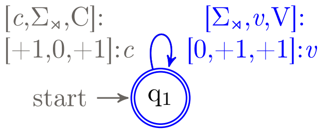
C: k t b

V: u i

T: C V C V C

Output:

k u t i



Illustrating MT FSTs

Working example:

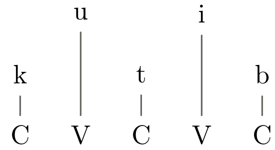
Input: 3 tapes

u i

k t b

C V C V C

Output: filled template



MT-FST implementation

Input:

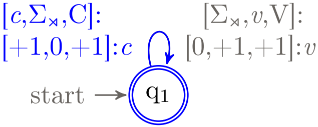
C: k t b

V: u i

T: C V C V C

Output:

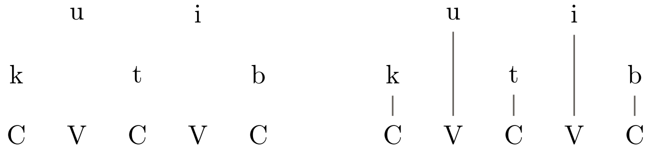
k u t i b



Illustrating MT FSTs

Working example:

Input: 3 tapes — Output: filled template



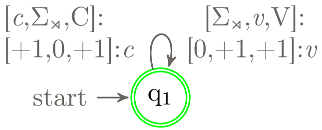
MT-FST implementation

Input:

C: k t b
V: u i
T: C V C V C

Output:

k u t i b



Subclasses vs. full powers of MT

- Multi-Tape FSTs for templates
  1. MT FSTs are an intuitive implementation
  2. Long history of use for Semitic

Subclasses vs. full powers of MT

- Multi-Tape FSTs for templates
  1. MT FSTs are an intuitive implementation
  2. Long history of use for Semitic
- But, do we need full power of MT FSTs? ...

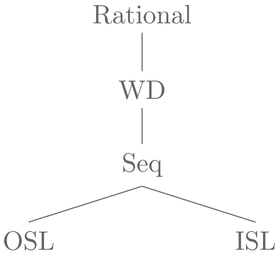
Subclasses vs. full powers of MT

- Multi-Tape FSTs for templates
  1. MT FSTs are an intuitive implementation
  2. Long history of use for Semitic
- But, do we need full power of MT FSTs? ... No!

Subclasses for FSTs

Single-tape FST:

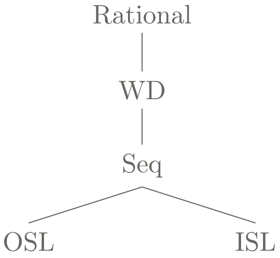
- Lot of work on subclasses! ☺



Subclasses for FSTs

Single-tape FST:

- Lot of work on subclasses! ☺

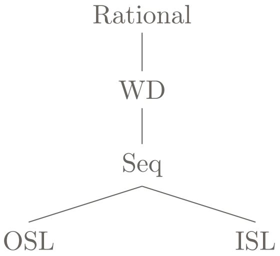


- Subclasses map to different patterns
- Concatenative morphology *mostly* needs ISL ☺

Subclasses for FSTs

Single-tape FST:

- Lot of work on subclasses! ☺



- Subclasses map to different patterns
- Concatenative morphology *mostly* needs ISL ☺

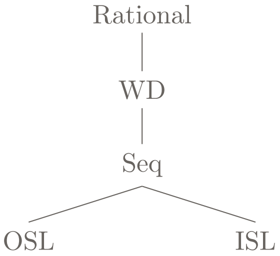
Multi-Tape FST

- Not much subclass work ☹

Subclasses for FSTs

Single-tape FST:

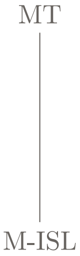
- Lot of work on subclasses! ☺



- Subclasses map to different patterns
- Concatenative morphology *mostly* needs ISL ☺

Multi-Tape FST

- Not much subclass work ☹



- Template need ISL over MT ! ☺

ISL over single-tape?

- Weak subclass for concatenative morphology is the k-Input Strictly Local (k-ISL) class

ISL over single-tape?

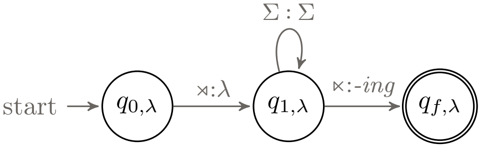
- Weak subclass for concatenative morphology is the k-Input Strictly Local (k-ISL) class
  - = keep track of **ONLY** last k segments in input

ISL morphology: suffixation

- English progressive: suffix *-ing*
  - *speak* *speak-ing*
  - *hold* *hold-ing*
- 1-ISL because *only* need to check if reached end boundary $\ltimes$

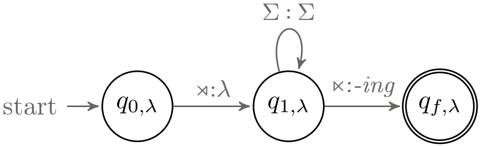
ISL morphology: suffixation

- Working example: *hold* → *hold-ing*
  Input: ⋊ h o l d ⋉
  Output:



ISL morphology: suffixation

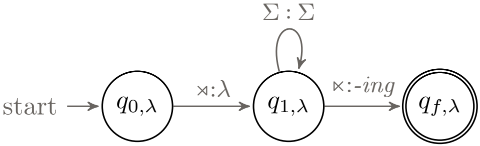
- Working example: *hold* → *hold-ing*
  Input: ⋊ h o l d ⋉
  Output:



- 1-ISL states keep track of last *k-1* (=0) seen input

ISL morphology: suffixation

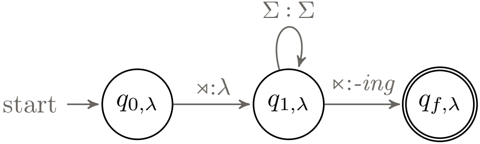
- Working example: *hold* → *hold-ing*
  Input: ⋊ h o l d ⋉
  Output:



- 1-ISL states keep track of last *k-1* (=0) seen input
  - Last seen is *empty string* λ

ISL morphology: suffixation

- Working example: *hold* → *hold-ing*
  Input: ⋊ h o l d ⋉
  Output:



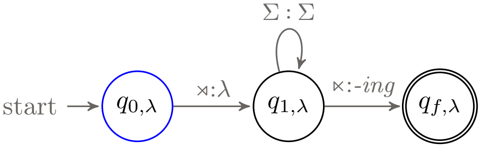
- 1-ISL states keep track of last *k-1* (=0) seen input
  - Last seen is *empty string* λ
- States with same *k-1* memorized string are the same
  - except initial and final states q_0, q_f

ISL morphology: suffixation

- Working example: *hold* → *hold-ing*

  Input: ⋊ h o l d ⋉

  Output:



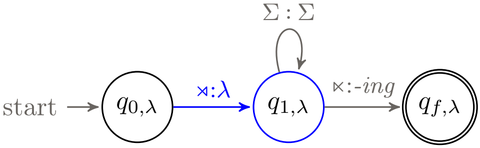
- 1-ISL states keep track of last *k-1* (=0) seen input
  - Last seen is *empty string* λ
- States with same *k-1* memorized string are the same
  - except initial and final states q_0, q_f

ISL morphology: suffixation

- Working example: *hold* → *hold-ing*

  Input: ⋊ h o l d ⋉

  Output:

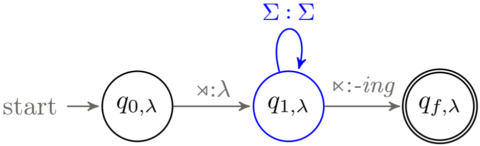


- 1-ISL states keep track of last *k-1* (=0) seen input
  - Last seen is *empty string* λ
- States with same *k-1* memorized string are the same
  - except initial and final states q_0, q_f

ISL morphology: suffixation

- Working example: *hold* → *hold-ing*

Input:	⋊	h	o	l	d	⋉
Output:		h				

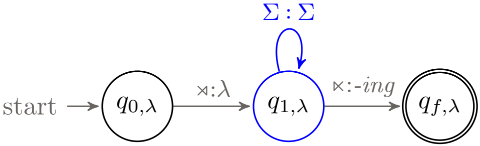


- 1-ISL states keep track of last *k-1* (=0) seen input
  - Last seen is *empty string* λ
- States with same *k-1* memorized string are the same
  - except initial and final states q_0, q_f

ISL morphology: suffixation

- Working example: *hold* → *hold-ing*

Input:	$\rtimes$	h	o	l	d	$\ltimes$
Output:		h	o			

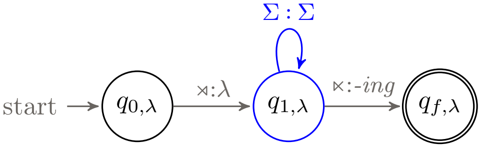


- 1-ISL states keep track of last *k-1* (=0) seen input
  - Last seen is *empty string* λ
- States with same *k-1* memorized string are the same
  - except initial and final states q_0, q_f

ISL MORPHOLOGY: SUFFIXATION

- Working example: *hold* → *hold-ing*

Input:	$\rtimes$	h	o	l	d	$\ltimes$
Output:		h	o	l		

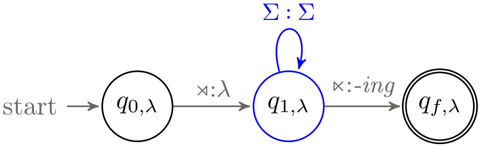


- 1-ISL states keep track of last *k-1* (=0) seen input
  - Last seen is *empty string* λ
- States with same *k-1* memorized string are the same
  - except initial and final states q_0, q_f

ISL morphology: suffixation

- Working example: *hold* → *hold-ing*

Input:	$\rtimes$	h	o	l	d	$\ltimes$
Output:		h	o	l	d	

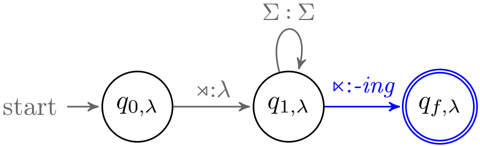


- 1-ISL states keep track of last *k-1* (=0) seen input
  - Last seen is *empty string* λ
- States with same *k-1* memorized string are the same
  - except initial and final states q_0, q_f

ISL morphology: suffixation

- Working example: *hold* → *hold-ing*

Input:	⋊	h	o	l	d	⋉		
Output:		h	o	l	d	i	n	g

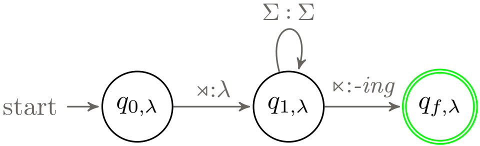


- 1-ISL states keep track of last *k-1* (=0) seen input
  - Last seen is *empty string* λ
- States with same *k-1* memorized string are the same
  - except initial and final states q_0, q_f

ISL morphology: suffixation

- Working example: *hold* → *hold-ing*

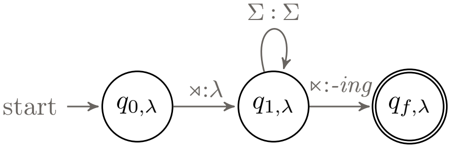
Input:	$\rtimes$	h	o	l	d	$\ltimes$			☺
Output:		h	o	l	d	i	n	g	



- 1-ISL states keep track of last *k-1* (=0) seen input
  - Last seen is *empty string* λ
- States with same *k-1* memorized string are the same
  - except initial and final states q_0, q_f

ISL over Multi-Tapes

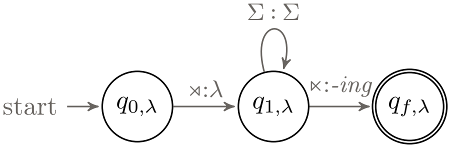
- k-ISL if check only the last k segments on...
  Single-tape FST
  the 1 input tape
  e.g. English suffixation



ISL over Multi-Tapes

- k-ISL if check only the last k segments on...

Single-tape FST	**Multi-Tape FST**
the 1 input tape	*every* input tape
e.g. English suffixation	



M-ISL over multi-tape FSTs

Working example:

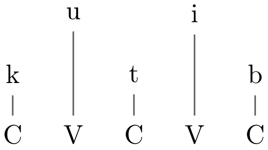
Input: 3 tapes

u i

k t b

C V C V C

Output: filled template



General MT-FST implementation

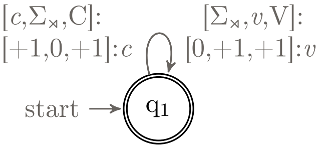
Input:

V: u i

C: k t b

T: C V C V C

Output:



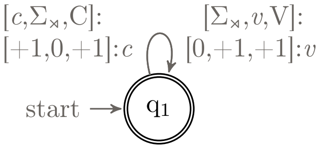
M-ISL over multi-tape FSTs

Working example: {ui},{ktb},{CV.CVC}→kutib

General MT-FST implementation

Input: Output:

V:	u	i			
C:	k	t	b		
T:	C	V	C	V	C



M-ISL over multi-tape FSTs

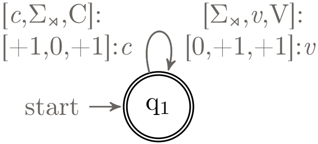
Working example: {ui},{ktb},{CV.CVC}→kutib

General MT-FST implementation

Add boundaries to input

Input: Output:

V:	⋊	u	i	⋉		
C:	⋊	k	t	b	⋉	
T:	⋊	C	V	C	V	C	⋉



M-ISL over multi-tape FSTs

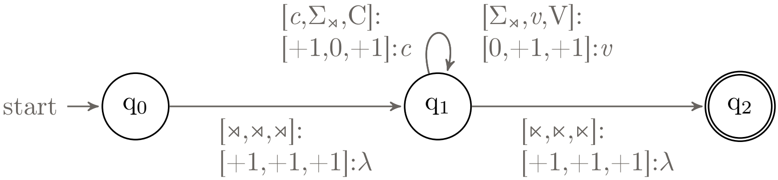
Working example: {ui},{ktb},{CV.CVC}→kutib

General MT-FST implementation
Add boundaries to MT-FST

Input:

V:	⋊	u	i	⋉			
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	V	C	⋉

Output:



M-ISL over multi-tape FSTs

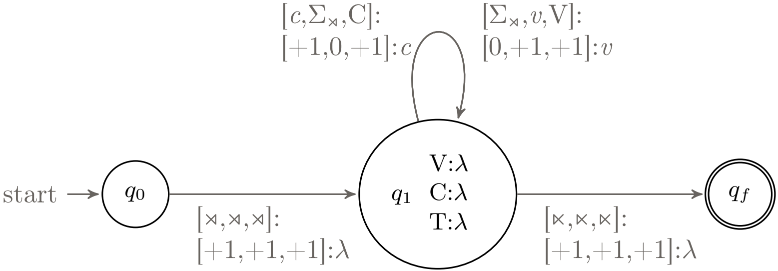
Working example: {ui},{ktb},{CV.CVC}→kutib

General MT-FST implementation

Make states remember last k-1 input (0) input

Input: Output:

V:	$\rtimes$	u	i	$\ltimes$			
C:	$\rtimes$	k	t	b	$\ltimes$		
T:	$\rtimes$	C	V	C	V	C	$\ltimes$



M-ISL over multi-tape FSTs

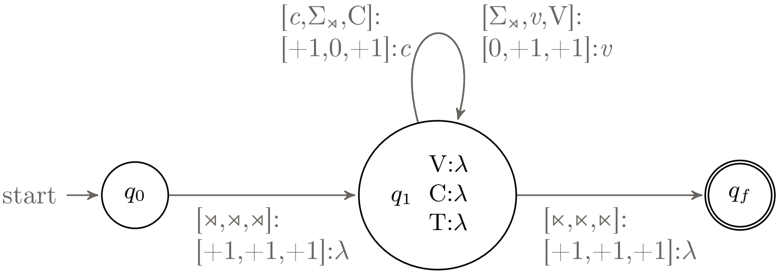
General MT-FST implementation

[1,1,1]-MISL because output depends on *only* the current input symbol

Input:

V:	$\rtimes$	u	i	$\ltimes$			
C:	$\rtimes$	k	t	b	$\ltimes$		
T:	$\rtimes$	C	V	C	V	C	$\ltimes$

Output:



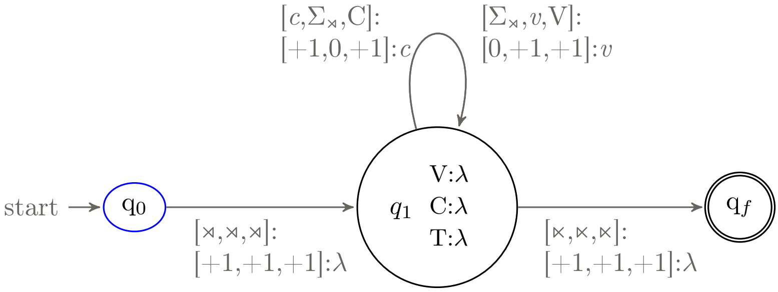
M-ISL over multi-tape FSTs

General MT-FST implementation

[1,1,1]-MISL because output depends on *only* the current input symbol

Input: Output:

V:	⋊	u	i	⋉			
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	V	C	⋉



M-ISL over multi-tape FSTs

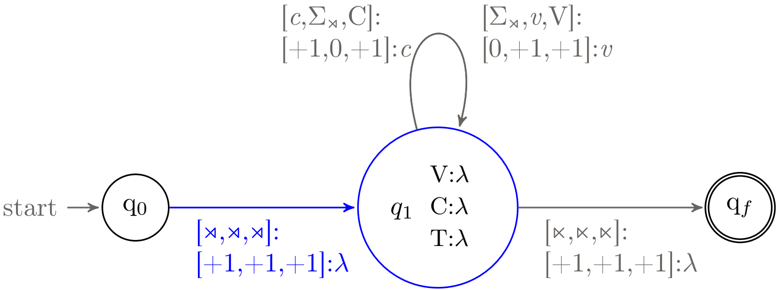
General MT-FST implementation

[1,1,1]-MISL because output depends on *only* the current input symbol

Input:

V:	⋊	u	i	⋉			
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	V	C	⋉

Output:

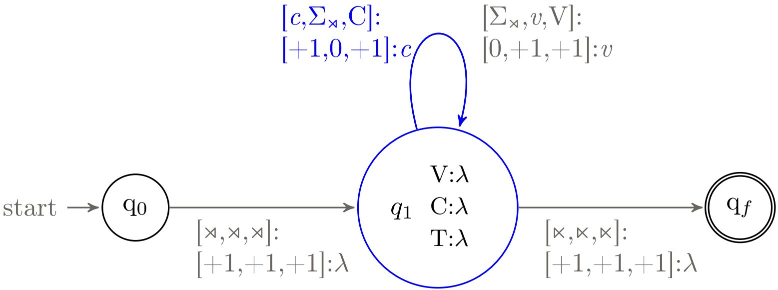


M-ISL over multi-tape FSTs

General MT-FST implementation

[1,1,1]-MISL because output depends on *only* the current input symbol

Input:								Output:
V:	⋊	u	i	⋉				
C:	⋊	k	t	b	⋉			
T:	⋊	C	V	C	V	C	⋉	k

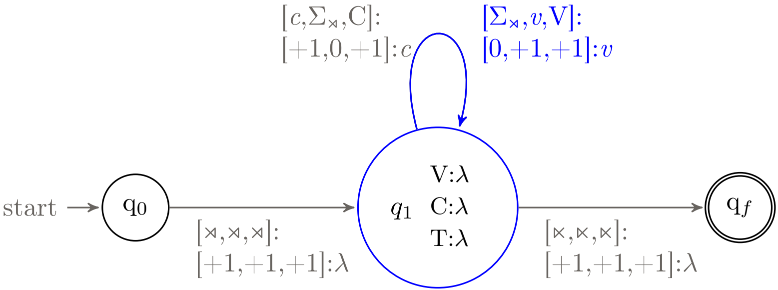


M-ISL over multi-tape FSTs

General MT-FST implementation

[1,1,1]-MISL because output depends on *only* the current input symbol

Input:								Output:	
V:	⋊	u	i	⋉					
C:	⋊	k	t	b	⋉				
T:	⋊	C	V	C	V	C	⋉	k	u

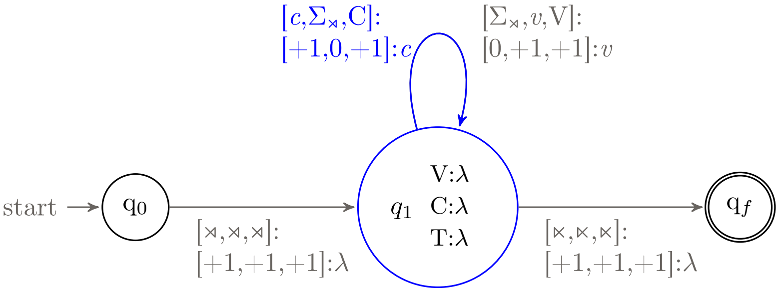


M-ISL over multi-tape FSTs

General MT-FST implementation

[1,1,1]-MISL because output depends on *only* the current input symbol

Input:								Output:		
V:	⋊	u	i	⋉						
C:	⋊	k	t	b	⋉					
T:	⋊	C	V	C	V	C	⋉	k	u	t

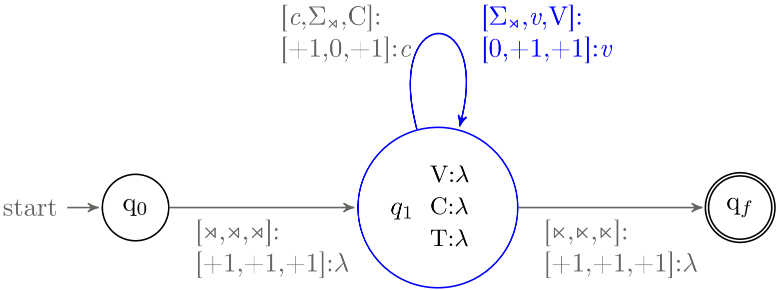


M-ISL over multi-tape FSTs

General MT-FST implementation

[1,1,1]-MISL because output depends on *only* the current input symbol

Input:								Output:			
V:	$\rtimes$	u	i	$\ltimes$							
C:	$\rtimes$	k	t	b	$\ltimes$						
T:	$\rtimes$	C	V	C	V	C	$\ltimes$	k	u	t	i



M-ISL over multi-tape FSTs

General MT-FST implementation

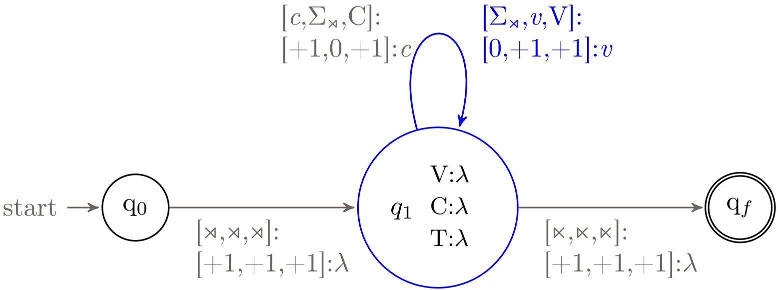
[1,1,1]-MISL because output depends on *only* the current input symbol

Input:

V:	⋊	u	i	⋉			
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	V	C	⋉

Output:

k u t i b

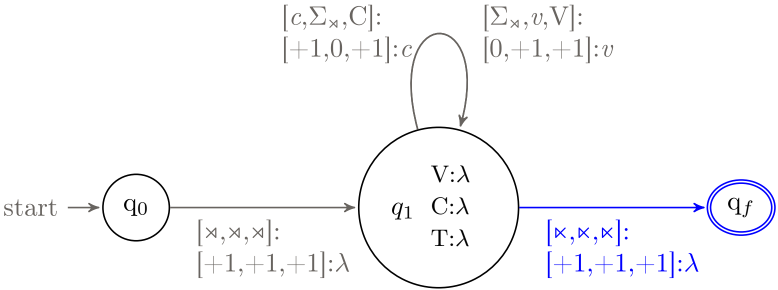


M-ISL over multi-tape FSTs

General MT-FST implementation

[1,1,1]-MISL because output depends on *only* the current input symbol

	Input:							Output:				
V:	⋊	u	i	⋉								
C:	⋊	k	t	b	⋉							
T:	⋊	C	V	C	V	C	⋉	k	u	t	i	b

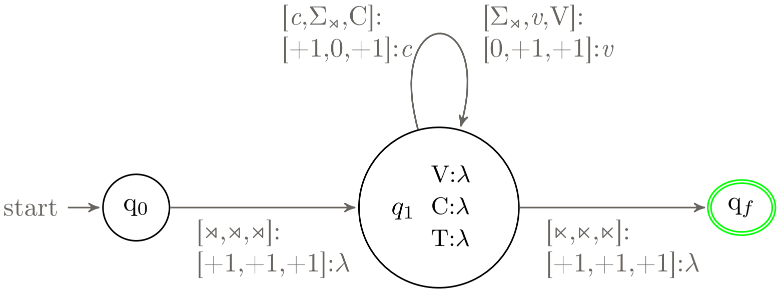


M-ISL over multi-tape FSTs

General MT-FST implementation

[1,1,1]-MISL because output depends on *only* the current input symbol

	Input:							Output:
V:	$\rtimes$	u	i	$\ltimes$				
C:	$\rtimes$	k	t	b	$\ltimes$			☺
T:	$\rtimes$	C	V	C	V	C	$\ltimes$	k u t i b



Why is it [1,1,1]-MISL

Why is it [1,1,1]-MISL

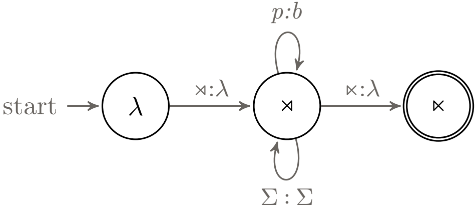
Consider an absolute neutralization rule:

- $p \to b \;/\; \_$
- p is voiced *regardless* of context

Why is it [1,1,1]-MISL

Consider an absolute neutralization rule:

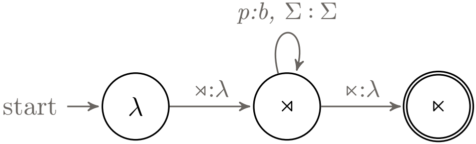
- $p \rightarrow b$ / _
- *p* is voiced *regardless* of context
- 1-ISL because only care about current input tape



Why is it [1,1,1]-MISL

Consider an absolute neutralization rule:

- 1-ISL because only care about current input tape

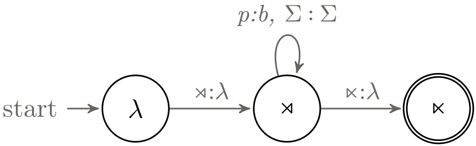


Template filling is [1,1,1]-MISL:

Why is it [1,1,1]-MISL

Consider an absolute neutralization rule:

- 1-ISL because only care about current input tape



Template filling is [1,1,1]-MISL:

- Change is based on *current* input symbol on *two* tapes

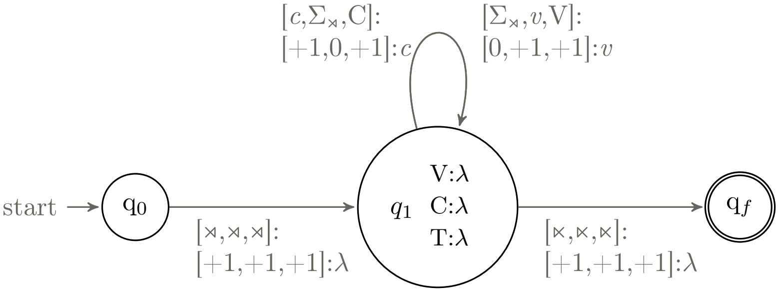


Table of Contents

More locality in Semitic templates

- Simple template matching is [1,1,1]-MISL

More locality in Semitic templates

- Simple template matching is [1,1,1]-MISL
- But what about the rest of Arabic?... A lot of MISL too!

More locality in Semitic templates

- Simple template matching is [1,1,1]-MISL
- But what about the rest of Arabic?... A lot of MISL too!

1-1 Matching	*ku.tib*	[1,1,1]-MISL

More locality in Semitic templates

- Simple template matching is [1,1,1]-MISL
- But what about the rest of Arabic?... A lot of MISL too!

1-1 Matching	*ku.tib*	[1,1,1]-MISL
Final spread	*k**a**.t**a**b*	[1,2,1]-MISL
Medial spread	*ka**t**.**t**ab*	[2,1,1]-MISL

More locality in Semitic templates

- Simple template matching is [1,1,1]-MISL
- But what about the rest of Arabic?... A lot of MISL too!

1-1 Matching	*ku.tib*	[1,1,1]-MISL
Final spread	*k**a**.t**a**b*	[1,2,1]-MISL
Medial spread	*ka**t.t**ab*	[2,1,1]-MISL
Pre-association	***ta-**ka.tab*	[1,1,1]-MISL
Edge-in effects		

More locality in Semitic templates

- Simple template matching is [1,1,1]-MISL
- But what about the rest of Arabic?... A lot of MISL too!

1-1 Matching	*ku.tib*	[1,1,1]-MISL
Final spread	*k**a**.t**a**b*	[1,2,1]-MISL
Medial spread	*ka**t.t**ab*	[2,1,1]-MISL
Pre-association	***ta-**ka.tab*	[1,1,1]-MISL
Edge-in effects		

- But locality depends on how you represent and derive these words

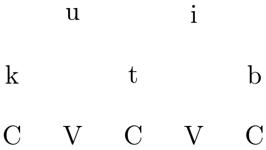
Final spreading

No spreading: 1-1 match for Vocalism and Template

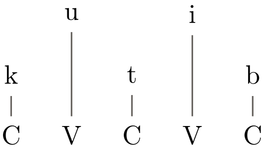
Final spreading

No spreading: 1-1 match for Vocalism and Template

Input: 2 vowels + 2 V slots



Output: 1-1 vocalism-V

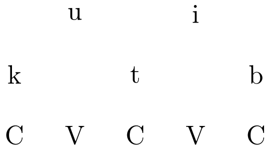


Final spreading: 1-many match for (final) Vocalism and Template

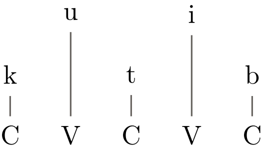
Final spreading

No spreading: 1-1 match for Vocalism and Template

Input: 2 vowels + 2 V slots

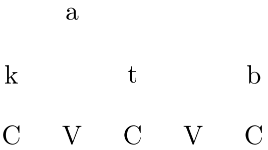


Output: 1-1 vocalism-V

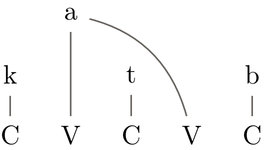


Final spreading: 1-many match for (final) Vocalism and Template

Input: 1 vowel + 2 V slots



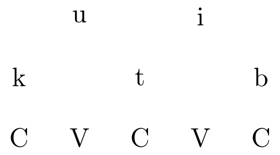
Output: 1-many vocalism-V



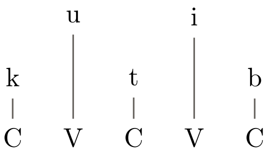
Final spreading

No spreading: 1-1 match for Vocalism and Template

Input: 2 vowels + 2 V slots

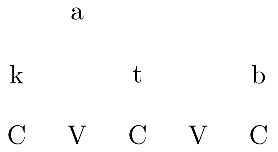


Output: 1-1 vocalism-V

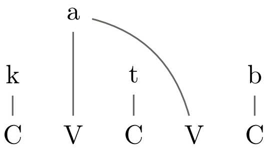


Final spreading: 1-many match for (final) Vocalism and Template

Input: 1 vowel + 2 V slots



Output: 1-many vocalism-V



Details

- *Why spreading?* OCP on Vocalism tier (and bigger words)
- *Locality*: non-local spread in single tape, local over multiple-tapes

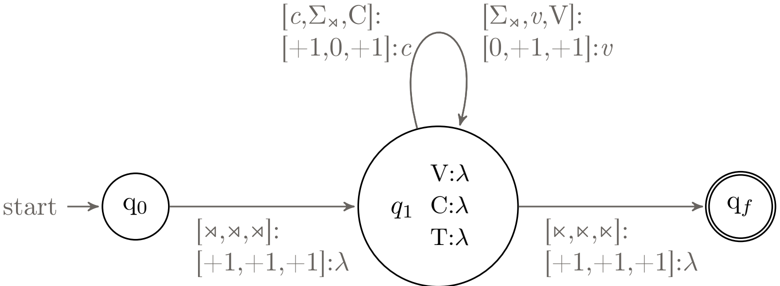
Final spreading is not [1,1,1]-MISL

Working example: {a},{ktb},{CV.CVC}→katab

General MT-FST implementation
MT-FST for final spreading is not [1,1,1]-MISL (needs *some* context)
[1,1,1]-MISL FST won't work ☹

Input: Output:

V:	⋊	a	⋉				
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	V	C	⋉



Final spreading is not [1,1,1]-MISL

Working example: {a},{ktb},{CV.CVC}→katab

General MT-FST implementation

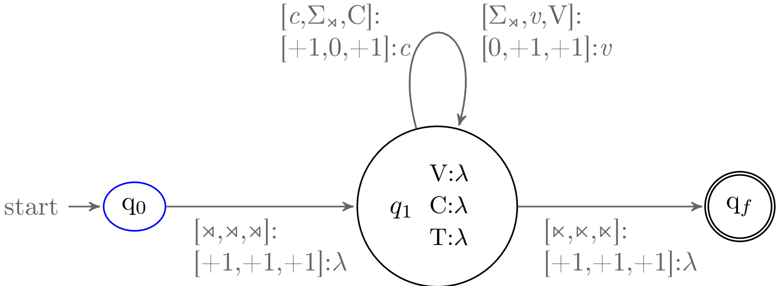
MT-FST for final spreading is not [1,1,1]-MISL (needs *some* context)

[1,1,1]-MISL FST won't work ☹

Input:

V:	$\rtimes$	a	$\ltimes$				
C:	$\rtimes$	k	t	b	$\ltimes$		
T:	$\rtimes$	C	V	C	V	C	$\ltimes$

Output:



Final spreading is not [1,1,1]-MISL

Working example: {a},{ktb},{CV.CVC}→katab

General MT-FST implementation

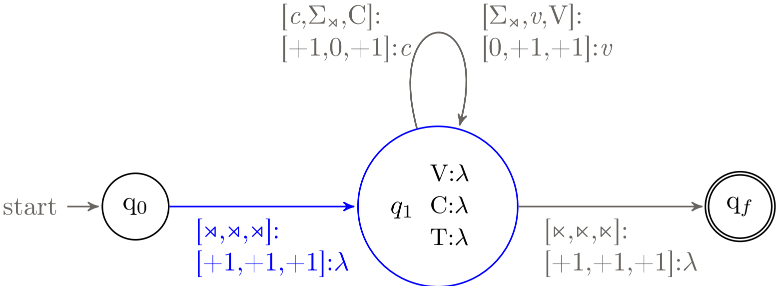
MT-FST for final spreading is not [1,1,1]-MISL (needs *some* context)

[1,1,1]-MISL FST won't work ☹

Input:

V:	⋊	a	⋉				
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	V	C	⋉

Output:



Final spreading is not [1,1,1]-MISL

Working example: {a},{ktb},{CV.CVC}→katab

General MT-FST implementation

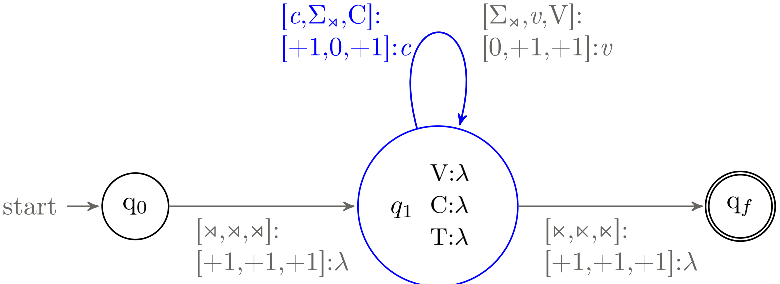
MT-FST for final spreading is not [1,1,1]-MISL (needs *some* context)

[1,1,1]-MISL FST won't work ☹

Input:

V:	⋊	a	⋉				
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	V	C	⋉

Output: k



Final spreading is not [1,1,1]-MISL

Working example: {a},{ktb},{CV.CVC}→katab

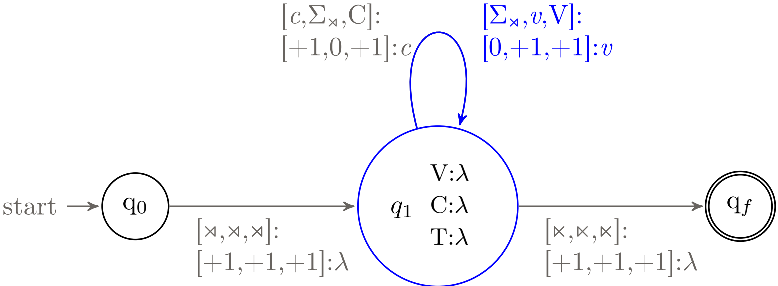
General MT-FST implementation

MT-FST for final spreading is not [1,1,1]-MISL (needs *some* context)

[1,1,1]-MISL FST won't work ☹

Input:

								Output:	
V:	$\rtimes$	a	$\ltimes$						
C:	$\rtimes$	k	t	b	$\ltimes$				
T:	$\rtimes$	C	V	C	V	C	$\ltimes$	k	a



Final spreading is not [1,1,1]-MISL

Working example: {a},{ktb},{CV.CVC}→katab

General MT-FST implementation

MT-FST for final spreading is not [1,1,1]-MISL (needs *some* context)

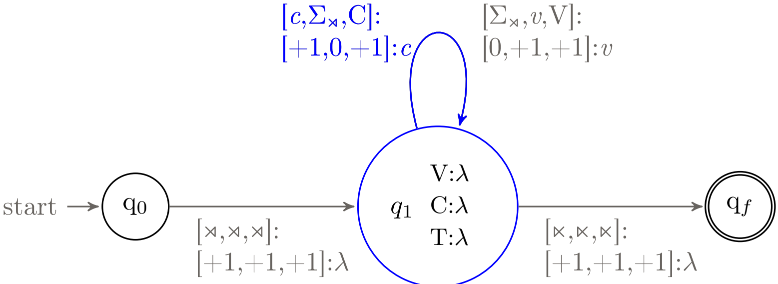
[1,1,1]-MISL FST won't work ☹

Input:

V:	⋊	a	⋉				
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	V	C	⋉

Output:

k a t

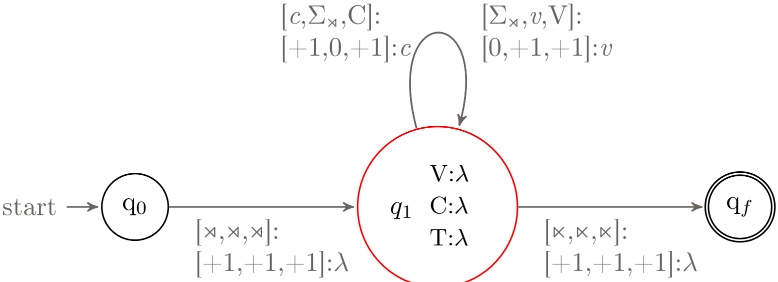


Final spreading is not [1,1,1]-MISL

Working example: {a},{ktb},{CV.CVC}→katab

General MT-FST implementation
MT-FST for final spreading is not [1,1,1]-MISL (needs *some* context)
[1,1,1]-MISL FST won't work ☹

Input:								Output:			
V:	⋊	a	⋉								
C:	⋊	k	t	b	⋉						☹
T:	⋊	C	V	C	V	C	⋉	k	a	t	



Finnal spreading is [1,2,1]-MISL

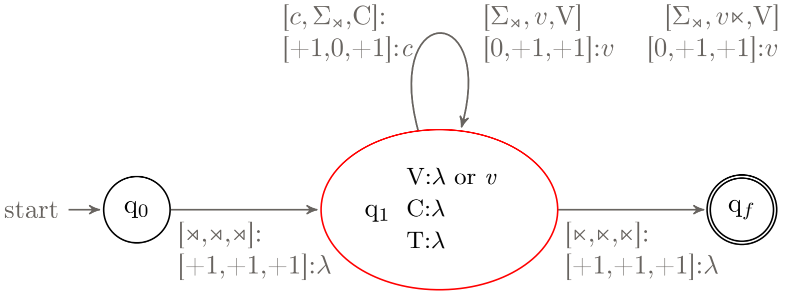
General MT-FST implementation

MT-FST for final spreading is [1,2,1]-MISL (remember final $v\ltimes$)

q_1 should be two separate states but slides have finite size

Input: Output:

V:	$\rtimes$	a	$\ltimes$			
C:	$\rtimes$	k	t	b	$\ltimes$	
T:	$\rtimes$	C	V	C	V	C	$\ltimes$



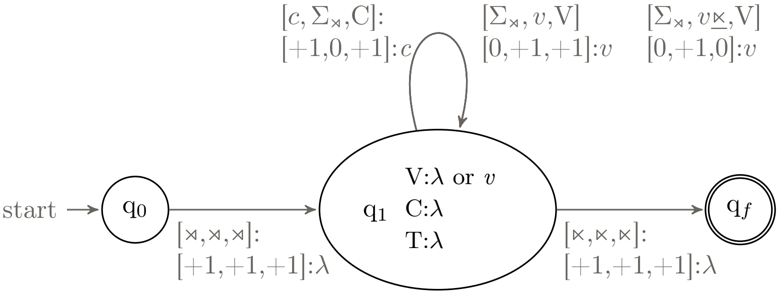
Final spreading is [1,2,1]-MISL

General MT-FST implementation

MT-FST for final spreading is [1,2,1]-MISL (remember final $v\ltimes$)

q_1 should be two separate states but slides have finite size

Input:							Output:
V:	⋊	a	⋉			
C:	⋊	k	t	b	⋉	
T:	⋊	C	V	C	V	C	⋉



Final spreading is [1,2,1]-MISL

General MT-FST implementation

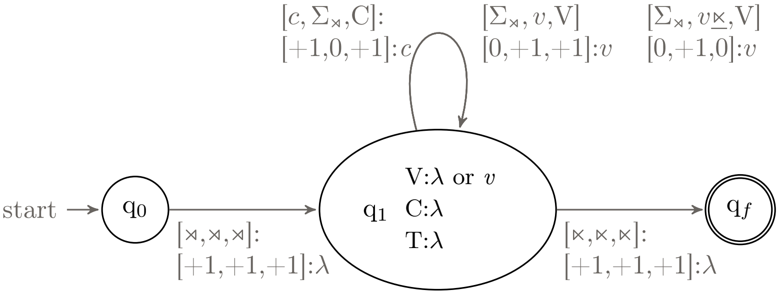
MT-FST for final spreading is [1,2,1]-MISL (remember final $v⋉$)

q_1 should be two separate states but slides have finite size

Input:

V:	⋊	a	⋉			
C:	⋊	k	t	b	⋉	
T:	⋊	C	V	C	V	C	⋉

Output:



Final spreading is [1,2,1]-MISL

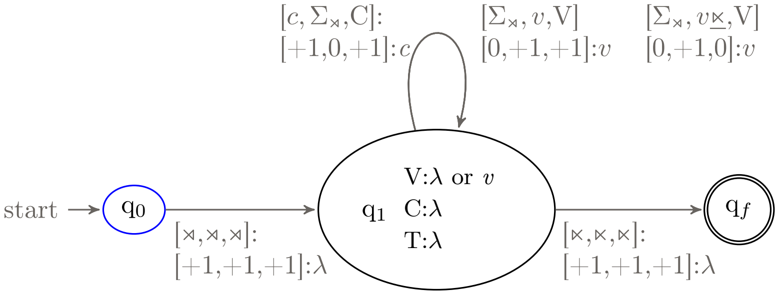
General MT-FST implementation

MT-FST for final spreading is [1,2,1]-MISL (remember final $v\ltimes$)

q_1 should be two separate states but slides have finite size

Input: | | | | | | | | Output:

V:	⋊	a	⋉				
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	V	C	⋉



Final spreading is [1,2,1]-MISL

General MT-FST implementation

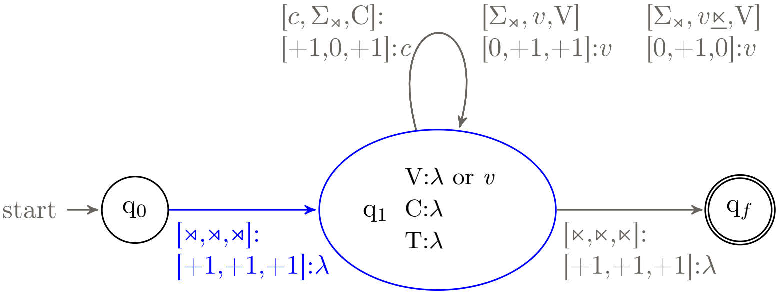
MT-FST for final spreading is [1,2,1]-MISL (remember final $v⋉$)

q_1 should be two separate states but slides have finite size

Input:

V:	⋊	a	⋉				
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	V	C	⋉

Output:



Final spreading is [1,2,1]-MISL

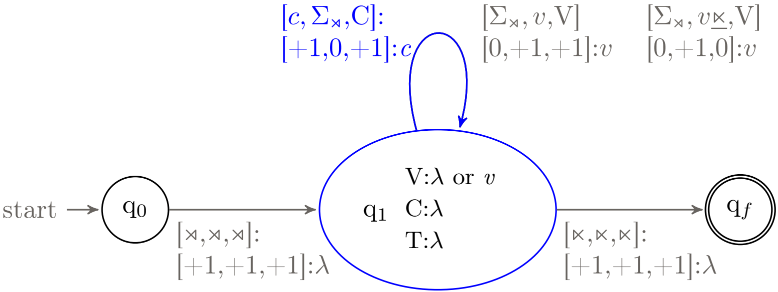
General MT-FST implementation

MT-FST for final spreading is [1,2,1]-MISL (remember final $v⋉$)

q_1 should be two separate states but slides have finite size

Input: Output:

V:	⋊	a	⋉					
C:	⋊	k	t	b	⋉			
T:	⋊	C	V	C	V	C	⋉	k

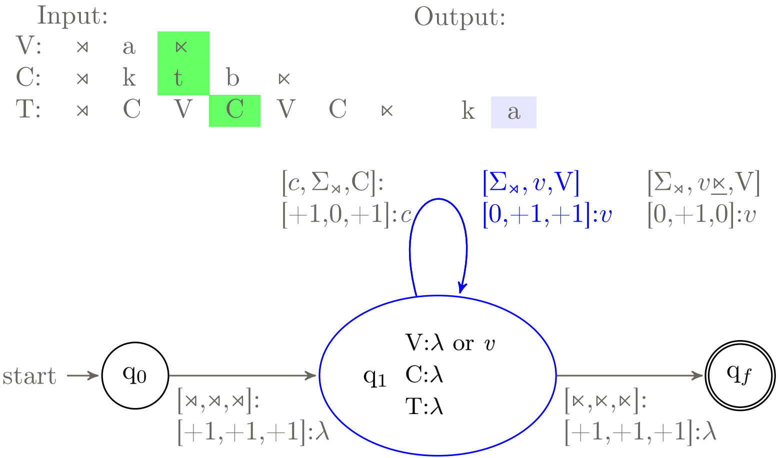


Final spreading is [1,2,1]-MISL

General MT-FST implementation

MT-FST for final spreading is [1,2,1]-MISL (remember final $v⋉$)

q_1 should be two separate states but slides have finite size

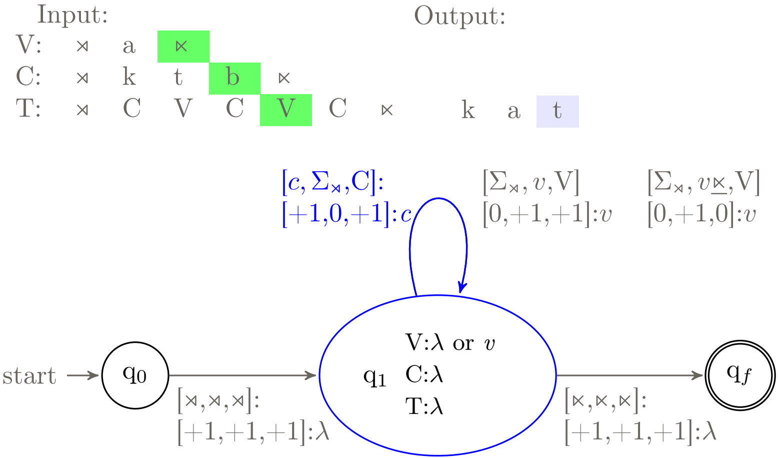


Final spreading is [1,2,1]-MISL

General MT-FST implementation

MT-FST for final spreading is [1,2,1]-MISL (remember final $v⋉$)

q_1 should be two separate states but slides have finite size

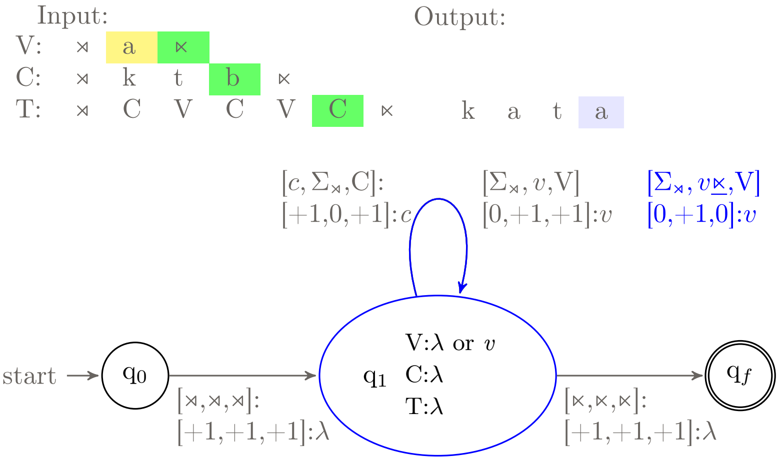


Final spreading is [1,2,1]-MISL

General MT-FST implementation

MT-FST for final spreading is [1,2,1]-MISL (remember final $v⋉$)

q_1 should be two separate states but slides have finite size



Final spreading is [1,2,1]-MISL

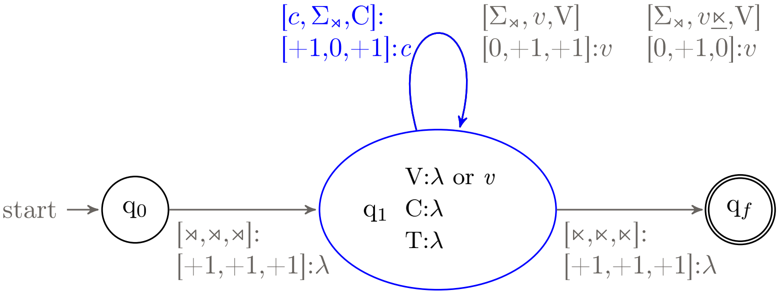
General MT-FST implementation

MT-FST for final spreading is [1,2,1]-MISL (remember final $v⋉$)

q_1 should be two separate states but slides have finite size

Input:												
V:	⋊	a	⋉									
C:	⋊	k	t	b	⋉							
T:	⋊	C	V	C	V	C	⋉					

Output:				
k	a	t	a	b



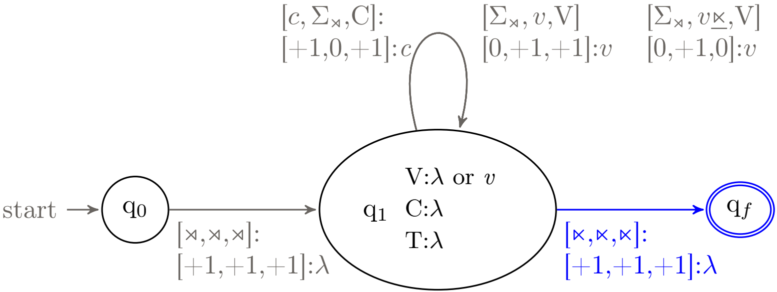
Final spreading is [1,2,1]-MISL

General MT-FST implementation

MT-FST for final spreading is [1,2,1]-MISL (remember final $v⋉$)

q_1 should be two separate states but slides have finite size

Input:								Output:				
V:	⋊	a	⋉									
C:	⋊	k	t	b	⋉							
T:	⋊	C	V	C	V	C	⋉	k	a	t	a	b



Final spreading is [1,2,1]-MISL

General MT-FST implementation

MT-FST for final spreading is [1,2,1]-MISL (remember final v⋉)

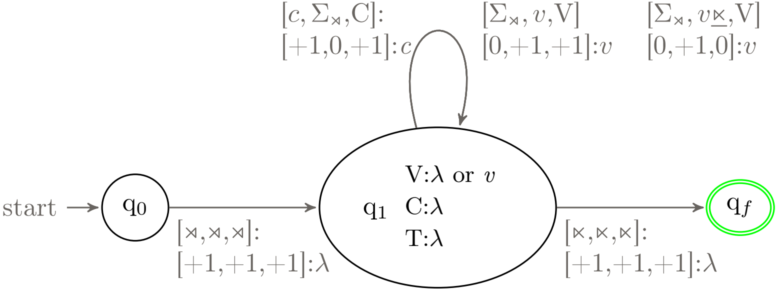
q_1 should be two separate states but slides have finite size

Input:

V:	⋊	a	⋉				
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	V	C	⋉

Output:

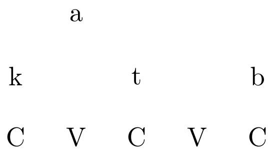
k a t a b ☺



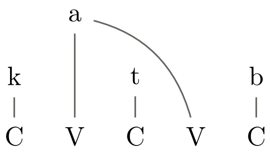
Medial spreading

- Simple (common) templates and final spreading is MISL

1-1 Match: *k**u**t**i**b*



Final spread: 1-many *k**a**t**a**b*

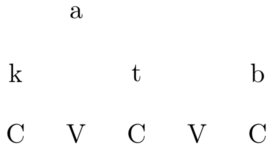


- What about <u>medial</u> spread: *ku**t.t**ib*?

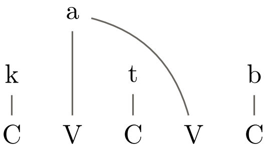
Medial spreading

- Simple (common) templates and final spreading is MISL

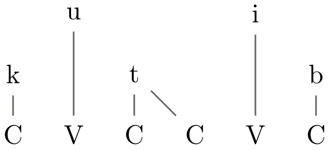
1-1 Match: *k**u**t**i**b*



Final spread: 1-many *k**a**t**a**b*



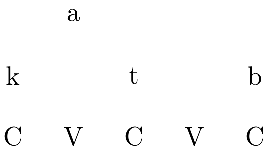
- What about <u>medial</u> spread: *ku**t.t**ib*?



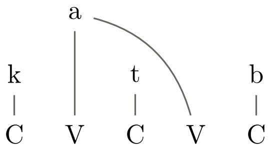
Medial spreading

- Simple (common) templates and final spreading is MISL

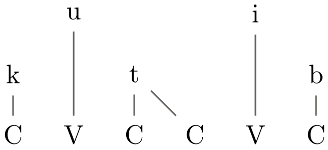
1-1 Match: *k**u**t**i**b*



Final spread: 1-many *k**a**t**a**b*



- What about <u>medial</u> spread: *ku**t.t**ib*?

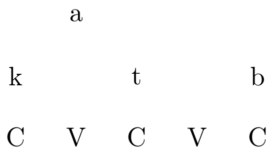


- Depends on computational *representation* and *derivation*

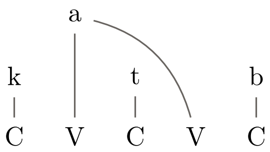
Medial spreading

- Simple (common) templates and final spreading is MISL

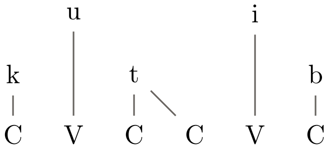
1-1 Match: *k**u**t**i**b*



Final spread: 1-many *k**a**t**a**b*



- What about <u>medial</u> spread: *ku**t.t**ib*?

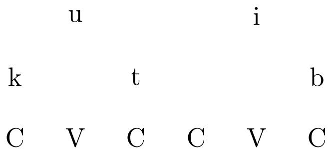


- Depends on computational *representation* and *derivation*
  - Representation: Template is *CVC.**G**VC*
  - Derivation: output *kutib* → *ku**t.t**ib*

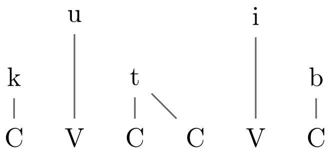
Medial spreading: representation

Working example: *kut.tib*

Input: medial geminate



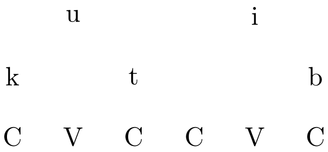
Output: filled template



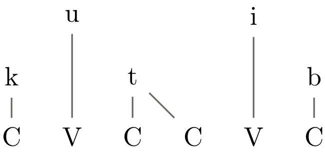
Medial spreading: representation

Working example: *ku**t.t**ib*

Input: medial geminate

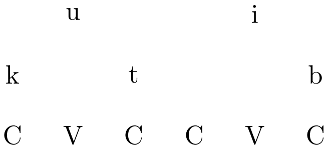


Output: filled template



But... Final spread predicts **ku**t.b**ib*

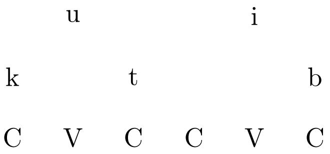
Input: medial geminate



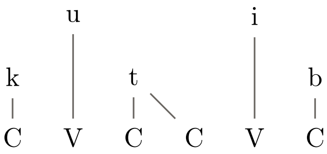
Medial spreading: representation

Working example: _ku**t.t**ib_

Input: medial geminate

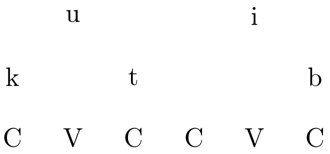


Output: filled template

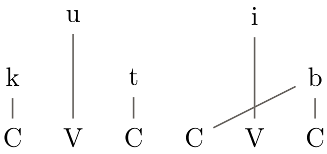


But... Final spread predicts _*ku**t.b**ib_

Input: medial geminate



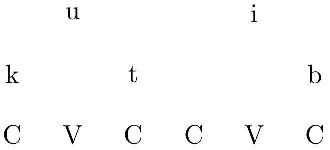
Output: filled template



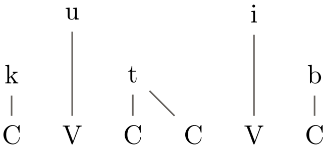
Medial spreading: representation

Working example: *ku**t**.**t**ib*

Input: medial geminate



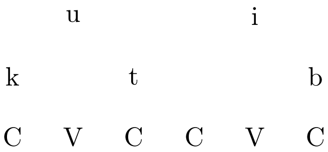
Output: filled template



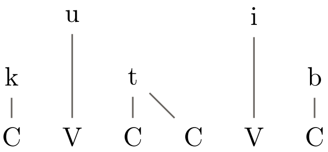
Medial spreading: representation

Working example: *kut.tib*

Input: medial geminate

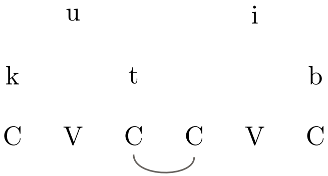


Output: filled template

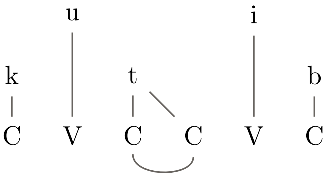


Trick: Template contains *geminated* structure

Input: medial geminate



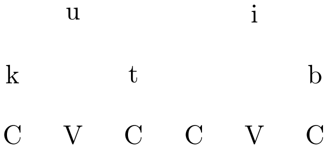
Output: filled template



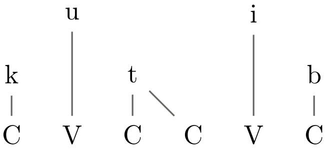
Medial spreading: representation

Working example: *kut.tib*

Input: medial geminate

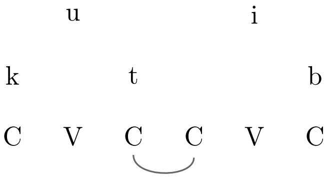


Output: filled template

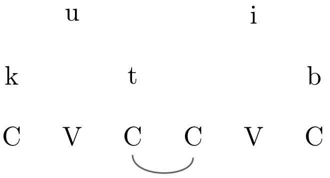


Trick: Template contains *geminated* structure

Input: medial geminate



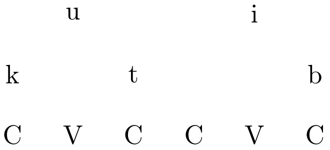
Output: filled template



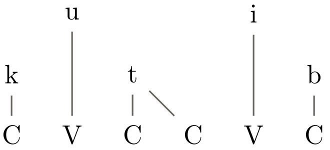
Medial spreading: representation

Working example: *kut.tib*

Input: medial geminate

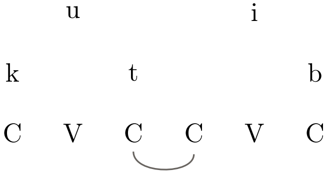


Output: filled template

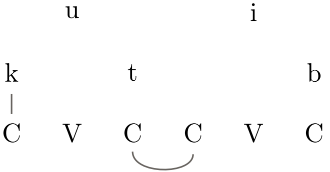


Trick: Template contains *geminated* structure

Input: medial geminate



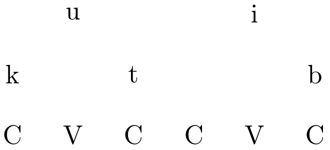
Output: filled template



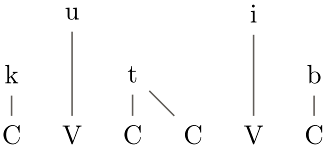
Medial spreading: representation

Working example: *kut.tib*

Input: medial geminate

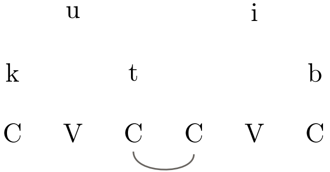


Output: filled template

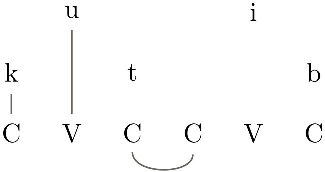


Trick: Template contains *geminated* structure

Input: medial geminate



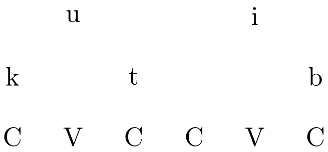
Output: filled template



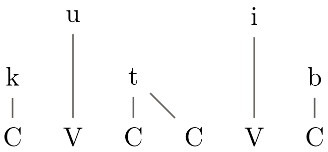
Medial spreading: representation

Working example: *kut.tib*

Input: medial geminate

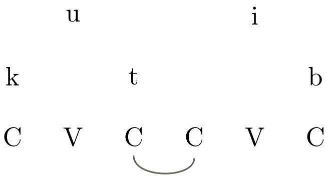


Output: filled template

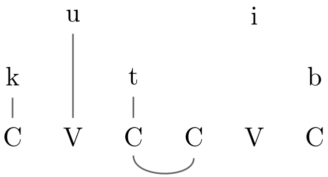


Trick: Template contains *geminated* structure

Input: medial geminate



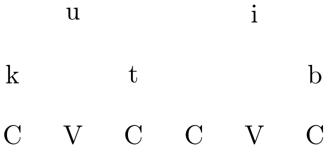
Output: filled template



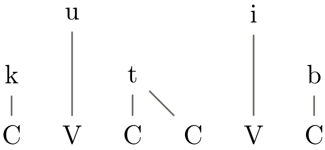
Medial spreading: representation

Working example: *kut.tib*

Input: medial geminate

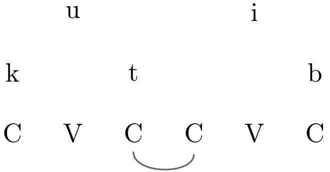


Output: filled template

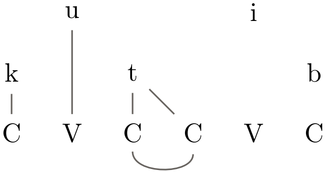


Trick: Template contains *geminated* structure

Input: medial geminate



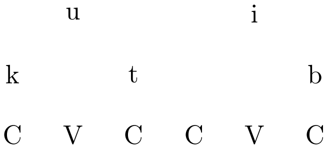
Output: filled template



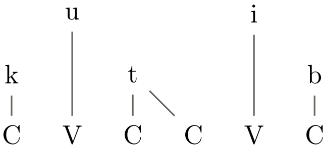
Medial spreading: representation

Working example: *kut.tib*

Input: medial geminate

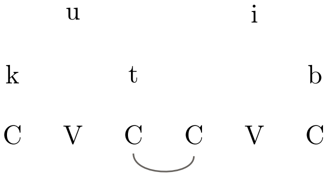


Output: filled template

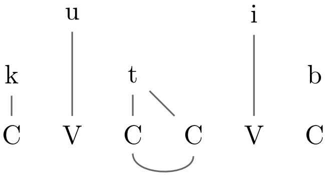


Trick: Template contains *geminated* structure

Input: medial geminate



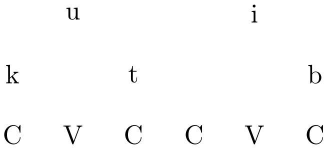
Output: filled template



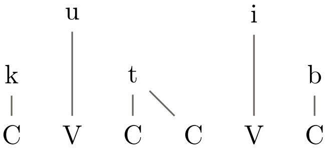
Medial spreading: representation

Working example: *kut.tib*

Input: medial geminate

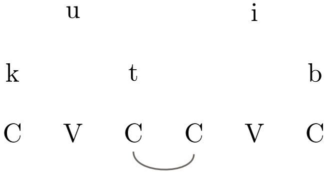


Output: filled template

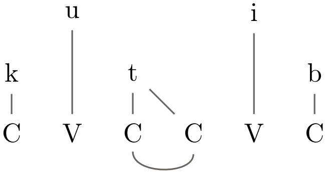


Trick: Template contains *geminated* structure

Input: medial geminate



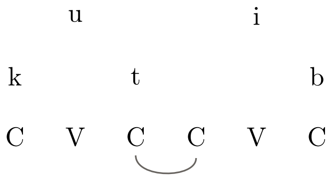
Output: filled template



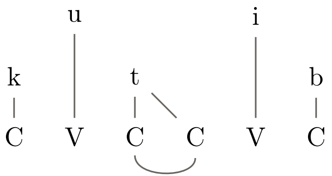
Medial spreading: representation

Trick: Template contains *geminated* structure

Input: medial geminate



Output: filled template

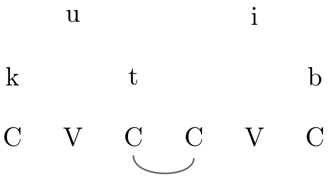


[1] (Kay, 1987)

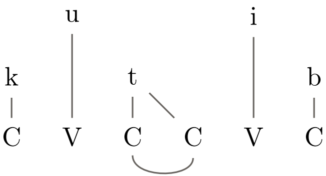
Medial spreading: representation

Trick: Template contains *geminated* structure

Input: medial geminate



Output: filled template



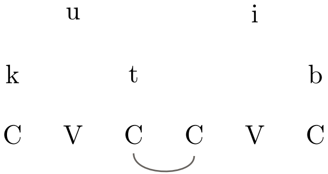
The trick works... but unclear how to insert C͡C into MT-FST

[1] (Kay, 1987)

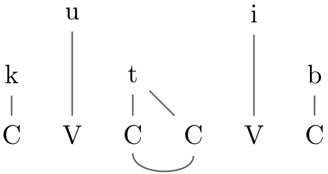
Medial spreading: representation

Trick: Template contains *geminated* structure

Input: medial geminate



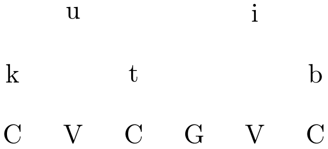
Output: filled template



The trick works... but unclear how to insert C͡C into MT-FST

Another trick: multi-linked C is changed to **C.G** (*geminate* node)[1]

Input: medial geminate

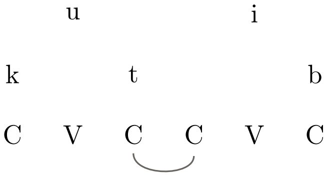


[1] (Kay, 1987)

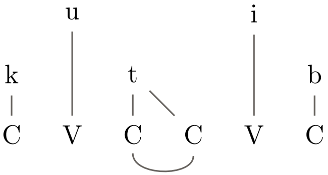
Medial spreading: representation

Trick: Template contains *geminated* structure

Input: medial geminate



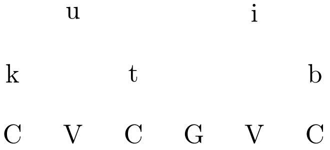
Output: filled template



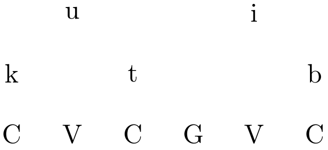
The trick works... but unclear how to insert C͡C into MT-FST

Another trick: multi-linked C is changed to **C.G** (*geminate* node)[1]

Input: medial geminate



Output: filled template

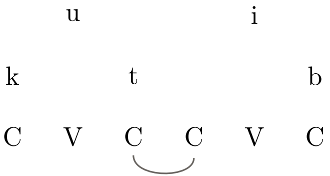


[1] (Kay, 1987)

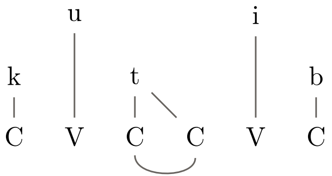
Medial spreading: representation

Trick: Template contains *geminated* structure

Input: medial geminate



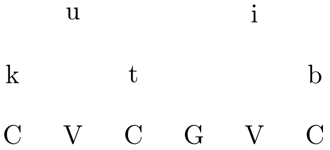
Output: filled template



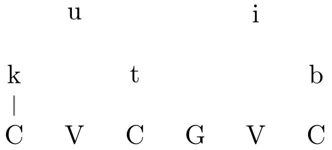
The trick works... but unclear how to insert C͡C into MT-FST

Another trick: multi-linked C is changed to **C.G** (*geminate* node)

Input: medial geminate



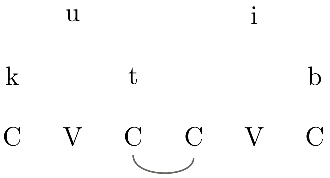
Output: filled template



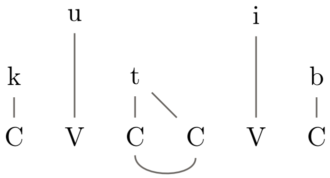
Medial spreading: representation

Trick: Template contains *geminated* structure

Input: medial geminate



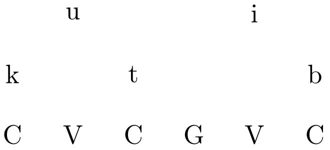
Output: filled template



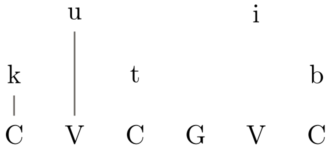
The trick works... but unclear how to insert C͡C into MT-FST

Another trick: multi-linked C is changed to **C.G** (*geminate* node)

Input: medial geminate



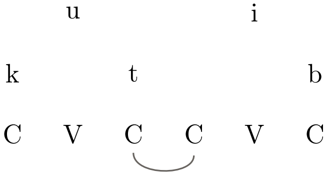
Output: filled template



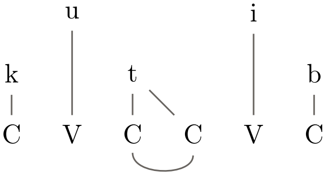
Medial spreading: representation

Trick: Template contains *geminated* structure

Input: medial geminate



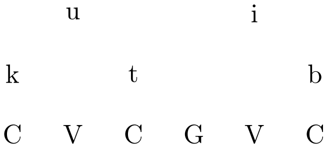
Output: filled template



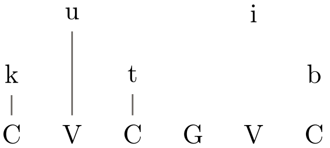
The trick works... but unclear how to insert $\widehat{\text{CC}}$ into MT-FST

Another trick: multi-linked C is changed to **C.G** (*geminate* node)

Input: medial geminate



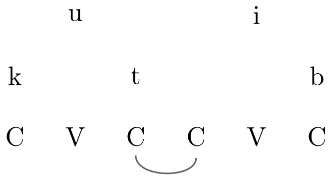
Output: filled template



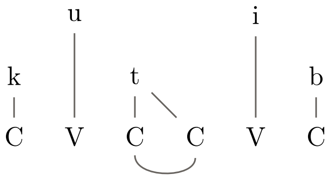
Medial spreading: representation

Trick: Template contains *geminated* structure

Input: medial geminate



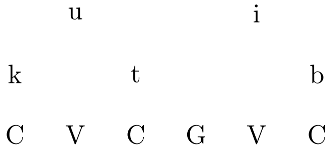
Output: filled template



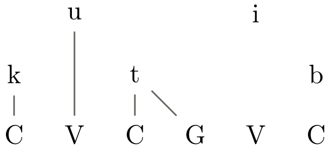
The trick works... but unclear how to insert C͡C into MT-FST

Another trick: multi-linked C is changed to **C.G** (*geminate* node)

Input: medial geminate



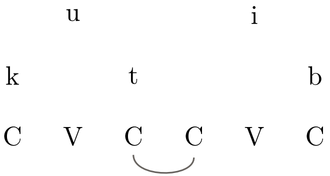
Output: filled template



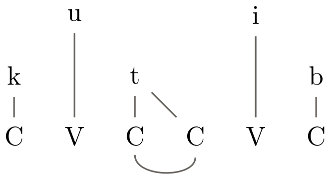
Medial spreading: representation

Trick: Template contains *geminated* structure

Input: medial geminate



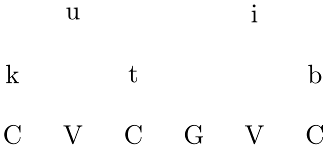
Output: filled template



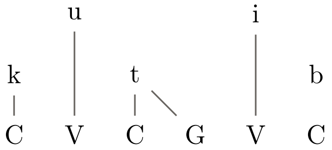
The trick works... but unclear how to insert C͡C into MT-FST

Another trick: multi-linked C is changed to **C.G** (*geminate* node)

Input: medial geminate



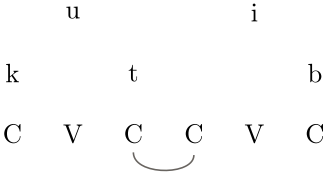
Output: filled template



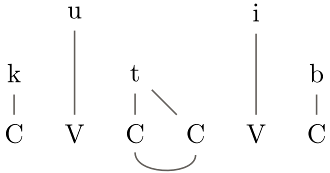
Medial spreading: representation

Trick: Template contains *geminated* structure

Input: medial geminate



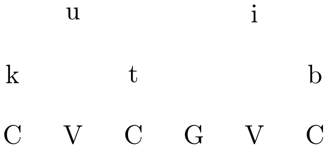
Output: filled template



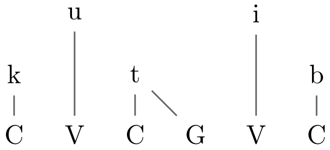
The trick works... but unclear how to insert C͡C into MT-FST

Another trick: multi-linked C is changed to **C.G** (*geminate* node)

Input: medial geminate



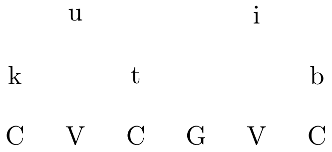
Output: filled template



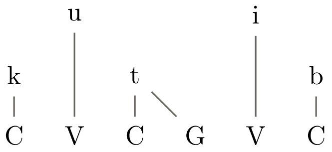
Medial spreading: representation

Representational trick: multi-linked C is changed to **C.G** (*geminate* node)

Input: medial geminate



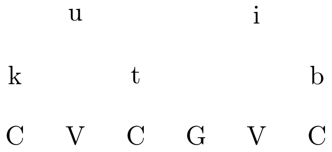
Output: filled template



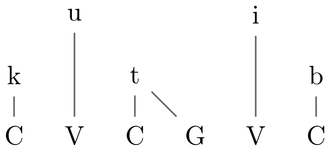
Medial spreading: representation

Representational trick: multi-linked C is changed to **C.G** (*geminate* node)

Input: medial geminate



Output: filled template



General MT-FST implementation

Representing geminates as unique **G** is easy for [2,1,1]-MISL!

Input:

V:	⋊	u	i	⋉				
C:	⋊	k	t	b	⋉			
T:	⋊	C	V	C	G	V	C	⋉

Output:

k a t t a b

Medial spreading: representation

General MT-FST implementation
Σ includes G as alphabet symbol

	Input:							Output:
V:	$\rtimes$	u	i	$\ltimes$				
C:	$\rtimes$	k	t	b	$\ltimes$			
T:	$\rtimes$	C	V	C	G	V	C	$\ltimes$

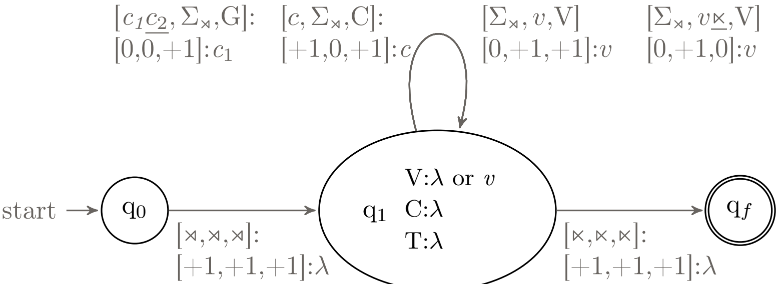
Medial spreading: representation

General MT-FST implementation

Σ includes G as alphabet symbol

Input: Output:

V:	⋊	u	i	⋉			
C:	⋊	k	t	b	⋉		
T:	⋊	C	V	C	G	V	C	⋉



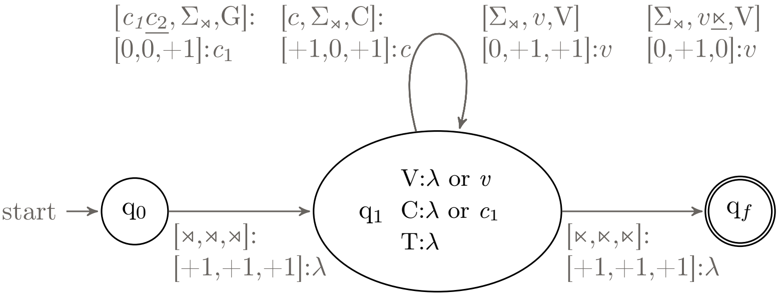
Medial spreading: representation

General MT-FST implementation

Σ includes G as alphabet symbol

Input: Output:

V:	⋊	u	i	⋉				
C:	⋊	k	t	b	⋉			
T:	⋊	C	V	C	G	V	C	⋉



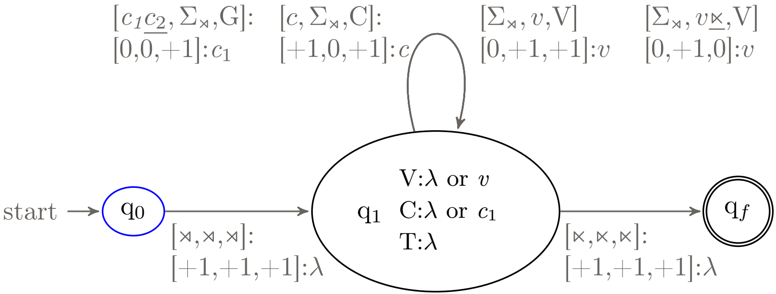
Medial spreading: representation

General MT-FST implementation

Σ includes G as alphabet symbol

Input: Output:

V:	⋊	u	i	⋉				
C:	⋊	k	t	b	⋉			
T:	⋊	C	V	C	G	V	C	⋉



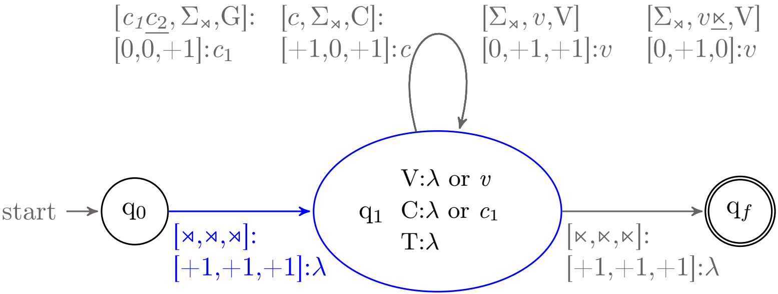
Medial spreading: representation

General MT-FST implementation

Σ includes G as alphabet symbol

Input: Output:

V:	⋊	u	i	⋉				
C:	⋊	k	t	b	⋉			
T:	⋊	C	V	C	G	V	C	⋉



Medial spreading: representation

General MT-FST implementation

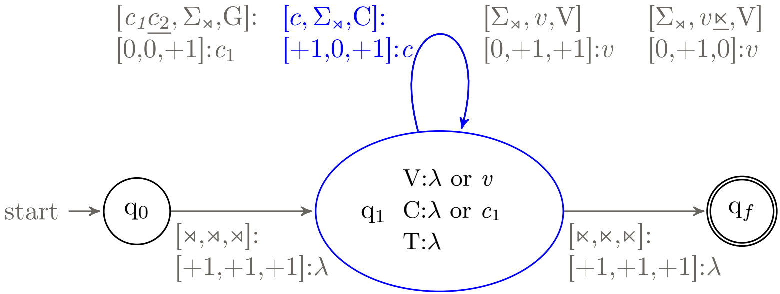
Σ includes G as alphabet symbol

Input:

V:	⋊	u	i	⋉				
C:	⋊	k	t	b	⋉			
T:	⋊	C	V	C	G	V	C	⋉

Output:

k



Medial spreading: representation

General MT-FST implementation

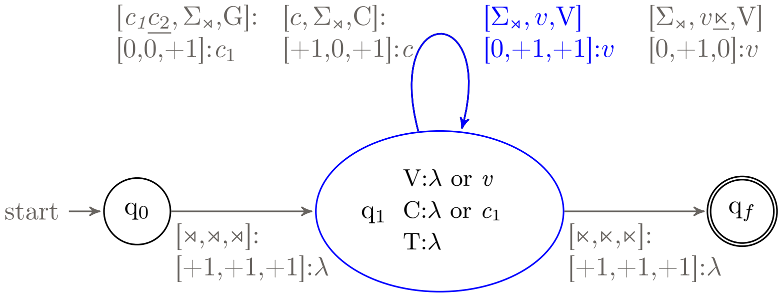
Σ includes G as alphabet symbol

Input:

V:	⋊	u	i	⋉				
C:	⋊	k	t	b	⋉			
T:	⋊	C	V	C	G	V	C	⋉

Output:

k u

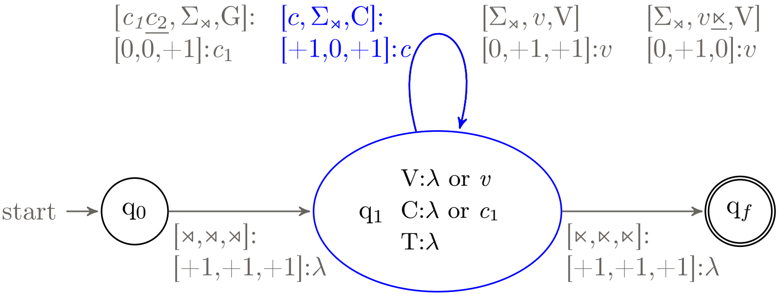


Medial spreading: representation

General MT-FST implementation

Σ includes G as alphabet symbol

	Input:								Output:		
V:	⋊	u	i	⋉							
C:	⋊	k	t	b	⋉						
T:	⋊	C	V	C	G	V	C	⋉	k	u	t



Medial spreading: representation

General MT-FST implementation

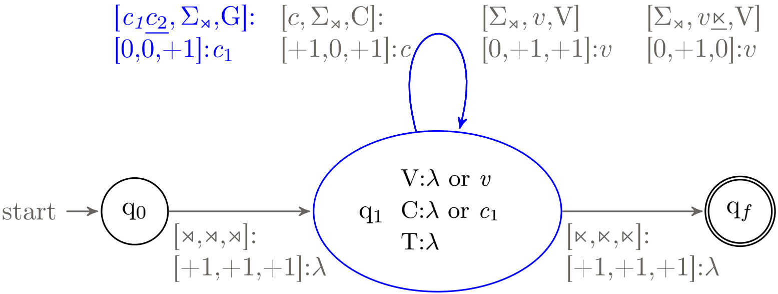
Σ includes G as alphabet symbol

Input:

V:	⋊	u	i	⋉				
C:	⋊	k	t	b	⋉			
T:	⋊	C	V	C	G	V	C	⋉

Output:

k u t t

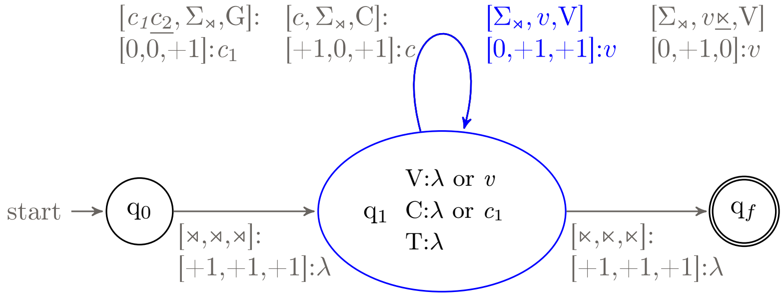


Medial spreading: representation

General MT-FST implementation

Σ includes G as alphabet symbol

	Input:								Output:				
V:	⋊	u	i	⋉									
C:	⋊	k	t	b	⋉								
T:	⋊	C	V	C	G	V	C	⋉	k	u	t	t	i

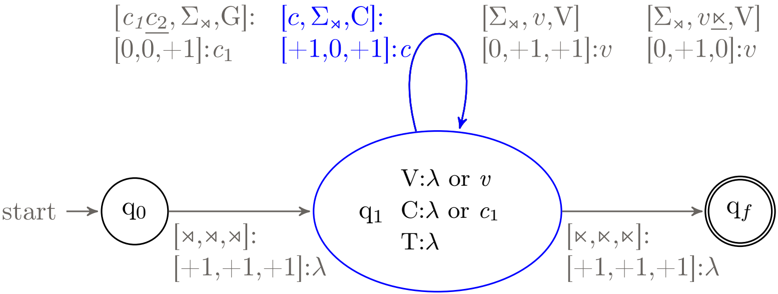


Medial spreading: representation

General MT-FST implementation

Σ includes G as alphabet symbol

	Input:								Output:					
V:	⋊	u	i	⋉										
C:	⋊	k	t	b	⋉									
T:	⋊	C	V	C	G	V	C	⋉	k	u	t	t	i	b

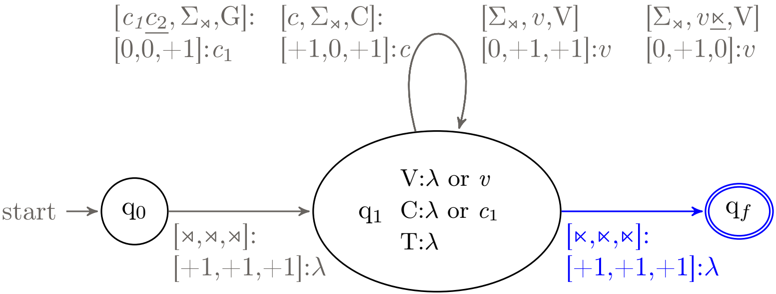


Medial spreading: representation

General MT-FST implementation

Σ includes G as alphabet symbol

	Input:								Output:					
V:	⋊	u	i	⋉										
C:	⋊	k	t	b	⋉									
T:	⋊	C	V	C	G	V	C	⋉	k	u	t	t	i	b



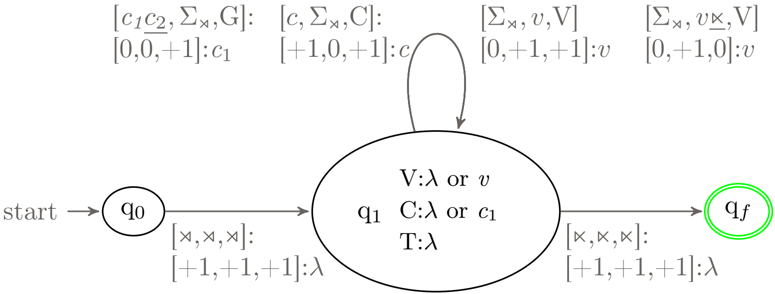
Medial spreading: representation

General MT-FST implementation

Σ includes G as alphabet symbol

	Input:								Output:
V:	⋊	u	i	⋉					
C:	⋊	k	t	b	⋉				
T:	⋊	C	V	C	G	V	C	⋉	k u t t i b

☺



Medial spreading: Derivation

- Computing medial spread *ka**t.t**ab* is tricky
- Depends on *representation* and *derivation*

Medial spreading: Derivation

- Computing medial spread *ka**t.t**ab* is tricky
- Depends on *representation* and *derivation*
  - Representing gemination with enriched template
    - Template = $CVC.\boldsymbol{G}VC$ [2,1,1]-MISL
  - Deriving gemination

Medial spreading: Derivation

- Computing medial spread *ka**t**.**t**ab* is tricky
- Depends on *representation* and *derivation*
  - Representing gemination with enriched template
    - Template = $CVC.\boldsymbol{G}VC$ [2,1,1]-MISL
  - Deriving gemination
    1. Input {ktb, ui, CV.CVC}

Medial spreading: Derivation

- Computing medial spread *ka**t.t**ab* is tricky
- Depends on *representation* and *derivation*
  - Representing gemination with enriched template
    - Template = *CVC.**G**VC* [2,1,1]-MISL
  - Deriving gemination

    1. Input {ktb, ui, CV.CVC}
    2. Intermediate kutib [1,1,1]-MISL

Medial spreading: Derivation

- Computing medial spread *ka**t**.**t**ab* is tricky
- Depends on *representation* and *derivation*
  - Representing gemination with enriched template
    - Template = *CVC.**G**VC* [2,1,1]-MISL
  - Deriving gemination

1.	Input	{ktb, ui, CV.CVC}	
2.	Intermediate	kutib	[1,1,1]-MISL
3.	Infix	kut.Gib	4-ISL
	... or mora	kut.μib	4-ISL

Medial spreading: Derivation

- Computing medial spread *ka**t.t**ab* is tricky
- Depends on *representation* and *derivation*
  - Representing gemination with enriched template
    - Template = *CVC.**G**VC* [2,1,1]-MISL
  - Deriving gemination

1.	Input	{ktb, ui, CV.CVC}	
2.	Intermediate	kutib	[1,1,1]-MISL
3.	Infix	kut.Gib	4-ISL
	... or mora	kut.μib	4-ISL
4.	Spread	kut.tib	2-ISL

Medial spreading: Derivation

- Computing medial spread *ka**t**.**t**ab* is tricky
- Depends on *representation* and *derivation*
  - Representing gemination with enriched template
    - Template = $CVC.\boldsymbol{G}VC$ [2,1,1]-MISL
  - Deriving gemination

1.	Input	{ktb, ui, CV.CVC}	
2.	Intermediate	kutib	[1,1,1]-MISL
3.	Infix	kut.Gib	4-ISL
	... or mora	kut.μib	4-ISL
4.	Spread	kut.tib	2-ISL

- 3&4 are ISL because (Chandlee, 2017)

Medial spreading: Derivation

- Computing medial spread *ka**t.t**ab* is tricky
- Depends on *representation* and *derivation*
  - Representing gemination with enriched template
    - Template = *CVC.**G**VC* [2,1,1]-MISL
  - Deriving gemination

1.	Input	{ktb, ui, CV.CVC}	
2.	Intermediate	kutib	[1,1,1]-MISL
3.	Infix	kut.Gib	4-ISL
	... or mora	kut.μib	4-ISL
4.	Spread	kut.tib	2-ISL

- 3&4 are ISL because (Chandlee, 2017)
- Take-away: prosodic representation *and* morphological derivation matter!
  - Representation is a composition of Derivation

More locality...

Depends on # + type of V, C, T

More locality...

Depends on # + type of V, C, T

Matching	Input			Output	Power
1-1 Matching	*ktb*	*ui*	*CVCVC*	*ku.tib*	[1,1,1]-MISL
Final spread	*ktb*	***a***	*CVCVC*	*k**a**.t**a**b*	[1,2,1]-MISL
Gemination	*ktb*	*ui*	*CVC.**G**VC*	*ka**t**.**t**ab*	[2,1,1]-MISL
Pre-association	*ksb*	*a*	*C**t**VCVC*	*k**t**a.sab*	[1,1,1]-MISL
Partial copying	*brd*	*a*	*CVC.**F**VC*	*bar.**b**ad*	2-way
Total copying	*zl*	*ia*	*CVC.CVC*	***zil.zal***	2-way
Edge-in	*ktb*	*ua**i***	*mV-tV-CVC.CVC*	*mu-ta-kas.s**i**b*	varies...
C-spreading	*trʒm*	*ui*	*CVC.CVC*	*tar.ʒam*	varies
	ktb	*ui*	*CVC.CVC*	*kut.tib*	varies

Table of Contents

Finiteness and grammar

- Focus is computing templates of Arabic verbs
  - *Result*: template-filling is largely MISL
  - *Fact*: All verbs stems (templates) are at most 2 syllables+prefix (8 segments)

[2] More because of template size

Finiteness and grammar

- Focus is computing templates of Arabic verbs
  - *Result*: template-filling is largely MISL
  - *Fact*: All verbs stems (templates) are at most 2 syllables+prefix (8 segments)
- **Counter**: why use MISL instead of using ISL over large window = 8 segments[2]
- E.g.

[2] More because of template size

Finiteness and grammar

- Focus is computing templates of Arabic verbs
  - *Result*: template-filling is largely MISL
  - *Fact*: All verbs stems (templates) are at most 2 syllables+prefix (8 segments)
- **Counter**: why use MISL instead of using ISL over large window = 8 segments[2]
- E.g.

Input	Output
ktb-a-CV.CVC	ka.tab

[2] More because of template size

Finiteness and grammar

- Focus is computing templates of Arabic verbs
  - *Result*: template-filling is largely MISL
  - *Fact*: All verbs stems (templates) are at most 2 syllables+prefix (8 segments)
- **Counter**: why use MISL instead of using ISL over large window = 8 segments[2]
- E.g.

Input	Output
ktb-a-CV.CVC	ka.tab
ktb-ui-CV.CVC	ku.tib

[2] More because of template size

Finiteness and grammar

- Focus is computing templates of Arabic verbs
  - *Result*: template-filling is largely MISL
  - *Fact*: All verbs stems (templates) are at most 2 syllables+prefix (8 segments)
- **Counter**: why use MISL instead of using ISL over large window = 8 segments[2]
- E.g.

Input	Output
ktb-a-CV.CVC	ka.tab
ktb-ui-CV.CVC	ku.tib
ktb-a-CVC.GVC	kat.tab

[2]More because of template size

Finiteness and grammar

- Focus is computing templates of Arabic verbs
  - *Result*: template-filling is largely MISL
  - *Fact*: All verbs stems (templates) are at most 2 syllables+prefix (8 segments)
- **Counter**: why use MISL instead of using ISL over large window = 8 segments[2]
- E.g.

Input	Output
ktb-a-CV.CVC	ka.tab
ktb-ui-CV.CVC	ku.tib
ktb-a-CVC.GVC	kat.tab
trʒm-a-CVC.CVC	tar.ʒam
...	

[2]More because of template size

Finiteness and grammar

- Focus is computing templates of Arabic verbs
  - *Result*: template-filling is largely MISL
  - *Fact*: All verbs stems (templates) are at most 2 syllables+prefix (8 segments)
- **Counter**: why use MISL instead of using ISL over large window = 8 segments[2]
- E.g.

Input	Output
ktb-a-CV.CVC	ka.tab
ktb-ui-CV.CVC	ku.tib
ktb-a-CVC.GVC	kat.tab
trʒm-a-CVC.CVC	tar.ʒam
...	

→ 1T-FST needs *all* possible combinations to be finite!

[2] More because of template size

Finiteness and grammar

- Focus is computing templates of Arabic verbs
- **Counter**: why use MISL instead of using ISL over large window $= 8^3$ segments

[3]More because of template size

Finiteness and grammar

- Focus is computing templates of Arabic verbs
- **Counter**: why use MISL instead of using ISL over large window $= 8^3$ segments
- Answer:
  - **Implementation**:

[3] More because of template size

Finiteness and grammar

- Focus is computing templates of Arabic verbs
- **Counter**: why use MISL instead of using ISL over large window $= 8^3$ segments
- Answer:
  - **Implementation**: trade-off between state explosion (single tape) and richer computational structure (MT)
  - **Scientific**:

[3] More because of template size

Finiteness and grammar

- Focus is computing templates of Arabic verbs
- **Counter**: why use MISL instead of using ISL over large window $= 8^3$ segments
- Answer:
  - **Implementation**: trade-off between state explosion (single tape) and richer computational structure (MT)
  - **Scientific**:
    - 1T-ISL reduces Arabic into a finite-language
    - Generalizations are lost

[3] More because of template size

Finiteness and grammar

- Focus is computing templates of Arabic verbs
- **Counter**: why use MISL instead of using ISL over large window $= 8^3$ segments
- Answer:
  - **Implementation**: trade-off between state explosion (single tape) and richer computational structure (MT)
  - **Scientific**:
    - 1T-ISL reduces Arabic into a finite-language
    - Generalizations are lost
- Teasing apart infiniteness and finiteness (Savitch, 1993)
  - **Grammars**: generalizations on infinite-ly lengthed strings and over finite-ly bounded strings
  - **Infinite**: can match any combination of Cs, Vs, Ts
  - **Finite**: only 2-syllable templates are allowed
  - **Composition:** Composition of infinite+finite is a finite language but we look at the infinite side of the equation

[3] More because of template size

Conclusion

- Semitic templates:
    1. Typologically rare
    2. Theoretically cool
    3. Computationally local

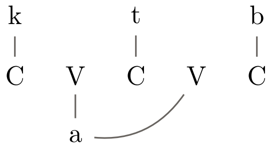
Conclusion

- Semitic templates:
    1. Typologically rare
    2. Theoretically cool
    3. Computationally local
- ... with the right (traditional) representation

Table of Contents

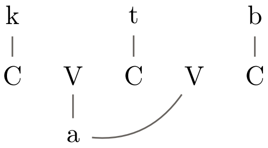
Final spreading for Consonants

- Most common final spread is for vowels: *katab*



Final spreading for Consonants

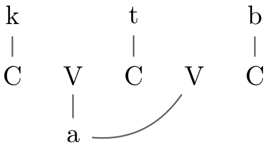
- Most common final spread is for vowels: *katab*



- For consonants, final spread can be caused by

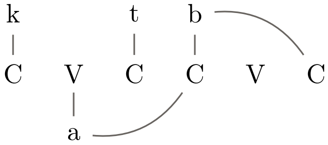
Final spreading for Consonants

- Most common final spread is for vowels: *katab*



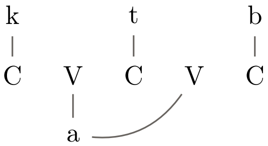
- For consonants, final spread can be caused by

1. >3C slots in template:
   *kta**b**a**b***



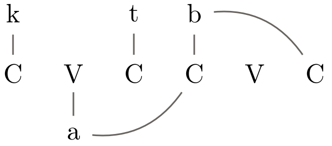
Final spreading for Consonants

- Most common final spread is for vowels: *katab*

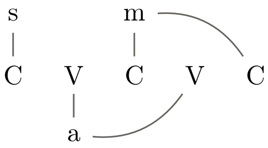


- For consonants, final spread can be caused by

1. >3C slots in template: *kta**b**a**b***

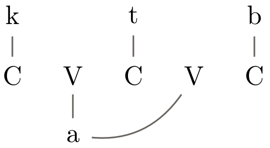


2. Subminimal roots: sa**m**a**m** (later *samm*)



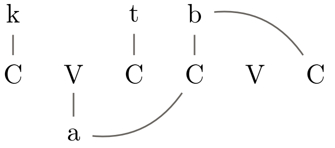
Final spreading for Consonants

- Most common final spread is for vowels: *katab*

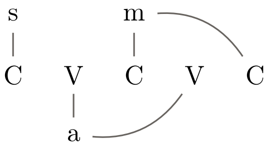


- For consonants, final spread can be caused by

1. >3C slots in template: *kta**b**a**b***



2. Subminimal roots: sa**m**a**m** (later *samm*)



- Both V and C final-spread are 2-MISL

Templates as primitive

- What our MISL FST takes as input:

Item		Role	Module
ktb	consonants	root or lexical content	morphology

Templates as primitive

- What our MISL FST takes as input:

Item		Role	Module
ktb	consonants	root or lexical content	morphology
ui	vocalism	inflection/theme	morphology

Templates as primitive

- What our MISL FST takes as input:

Item		Role	Module
ktb	consonants	root or lexical content	morphology
ui	vocalism	inflection/theme	morphology
$CV.CVC$	template	*basic verb*	morphology

Templates as primitive

- What our MISL FST takes as input:

Item		Role	Module
ktb	consonants	root or lexical content	morphology
ui	vocalism	inflection/theme	morphology
CV.CVC	template	*basic verb*	morphology
CVC.GVC	template	*causative verb*	

Templates as primitive

- What our MISL FST takes as input:

Item		Role	Module
ktb	consonants	root or lexical content	morphology
ui	vocalism	inflection/theme	morphology
$CV.CVC$	template	*basic verb*	morphology
$CVC.GVC$	template	*causative verb*	

- Classical idea is that template is a morphological primitive
- But controversial...

Templates as primitives

- Counter: templates are phonologically *emergent* and not primitives

Templates as primitives

- Counter: templates are phonologically *emergent* and not primitives

Classical input			Contemporary input
root	*ktb*	*ktb*	root

Templates as primitives

- Counter: templates are phonologically *emergent* and not primitives

Classical input			Contemporary input
root	*ktb*	*ktb*	root
vocalism	*ui*	*ui*	vocalism

Templates as primitives

- Counter: templates are phonologically *emergent* and not primitives

Classical input			Contemporary input
root	*ktb*	*ktb*	root
vocalism	*ui*	*ui*	vocalism
template	*CV.CVC*	CON	syllable optimization

Templates as primitives

- Counter: templates are phonologically *emergent* and not primitives

Classical input			Contemporary input
root	*ktb*	*ktb*	root
vocalism	*ui*	*ui*	vocalism
template	$CV.CVC$	CON	syllable optimization
template	$CVC.\boldsymbol{G}VC$	μ	+ autosegments

- Phonology determines optimal organization of segments based on

Templates as primitives

- Counter: templates are phonologically *emergent* and not primitives

Classical input			Contemporary input
root	*ktb*	*ktb*	root
vocalism	*ui*	*ui*	vocalism
template	$CV.CVC$	CON	syllable optimization
template	$CVC.\boldsymbol{G}VC$	μ	+ autosegments

- Phonology determines optimal organization of segments based on
    1. syllable structure
    2. morphological autosegments
    3. minimality/maximality needs

Templates as primitives

- Counter: templates are phonologically *emergent* and not primitives

Classical input			Contemporary input
root	*ktb*	*ktb*	root
vocalism	*ui*	*ui*	vocalism
template	$CV.CVC$	CON	syllable optimization
template	$CVC.\boldsymbol{G}VC$	μ	+ autosegments

- Phonology determines optimal organization of segments based on
  1. syllable structure
  2. morphological autosegments
  3. minimality/maximality needs
- Question: do we still need MT and MISL if the template is emergent?

Templates as primitives

- Counter: templates are phonologically *emergent* and not primitives

Classical input			Contemporary input
root	*ktb*	*ktb*	root
vocalism	*ui*	*ui*	vocalism
template	$CV.CVC$	CON	syllable optimization
template	$CVC.\boldsymbol{G}VC$	μ	+ autosegments

- Phonology determines optimal organization of segments based on
  1. syllable structure
  2. morphological autosegments
  3. minimality/maximality needs
- Question: do we still need MT and MISL if the template is emergent?
  - Yes

Templates as primitives

- How does template emerge?

Templates as primitives

- How does template emerge?
  - Optimizing syllable structure!

ktb + ui		*[CC	Onset	Contiguity
a. ☞	kutib			***
b.	ktbui	*!		
c.	uktib		*!	

Templates as primitives

- How does template emerge?
  - Optimizing syllable structure!

ktb + ui	*[CC	Onset	Contiguity
a. ☞ kutib			***
b. ktbui	*!		
c. uktib		*!	

- Phonological derivation has two parts
  1. Gen: organizes Cs and Vs
  2. Eval: evaluates which organization is phonologically optimal

Templates as primitives

- How does template emerge?
  - Optimizing syllable structure!

ktb + ui	*[CC	Onset	Contiguity
a. ☞ kutib			***
b. ktbui	*!		
c. uktib		*!	

- Phonological derivation has two parts
  1. Gen: organizes Cs and Vs
  2. Eval: evaluates which organization is phonologically optimal
- But Gen is a blackbox with little work on how its computationally modeled

Templates as primitives

- Candidates in Gen *imply* a template

Templates as primitives

- Candidates in Gen *imply* a template
  = manner of organizing C and V: katab

ktb + ui			*[CC	Onset	Contiguity
a.	☞	kutib CV.CVC			***
b.		ktbui CC.CVV	*!		
c.		uktib VC.CVC		*!	

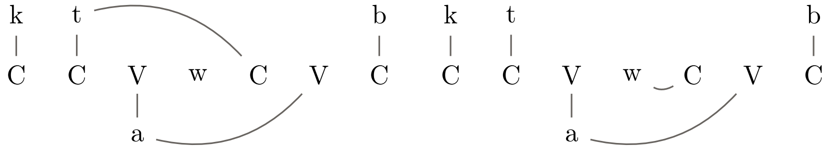
- MT models how Gen computes the phonologically emergent template
- **Conclusion**: whether emergent or primitive, the template is still there and needs to be computed

Minor wrinkles: vowel length

- Within a template, long vowels are always the same quality
  - V= a
  - T = CVV.CVC
  - *kaa.tab*
- Even if V>1, don't have two vowels in VV
  - V = ai
  - T = CVV.CVC
  - *kaa.tib*, not *kai.tib*
- Restriction is still MISL

Direction of copy G

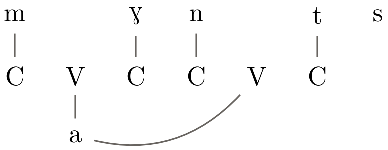
- In general case, spreading creates local copy from left-right
  - ka**t.t**ab vs. kat.**bab**
- Some patterns make contrasts between source of copied C:
  Form XII: *k.**t**aw.**t**ab* Form XIII: *k.taw.**w**ab*



- How can you compute this?
  - Two types of geminate G:
    1. G_C: Copy consonant from C tape: $ktaw.G_C ab \rightarrow k\boldsymbol{t}aw.\boldsymbol{t}ab$
    2. G_T: Copy consonant from T tape: $ktaw.G_T ab \rightarrow kta\boldsymbol{w}.\boldsymbol{w}ab$
- Both are MISL, but look at different tapes

Roots with too many Cs

- Verbs have 3 or 4 root consonants, never more
  - *katab* vs. *tarʒam*
- If a (borrowed) noun has >4 Cs, then a derived verb deletes the last C
  - *maɣnatis* → *maɣ.nat*



- Skipping final C is still MISL because the string is read left-right

Templates + Preassociated affixation

- Many templates consist of CV-template (for root) + affixes

Templates + Preassociated affixation

- Many templates consist of CV-template (for root) + affixes
  - Base: *katab* *kasab*

Templates + Preassociated affixation

- Many templates consist of CV-template (for root) + affixes
  - Base: *katab* *kasab*
  - Infix G/μ : *kat.**t**ab* *kas.**s**ab*

Base:	*katab*	*kasab*
Infix G/μ :	*kat.**t**ab*	*kas.**s**ab*

Templates + Preassociated affixation

- Many templates consist of CV-template (for root) + affixes

Base:	*katab*	*kasab*
Infix G/μ :	*kat.**t**ab*	*kas.**s**ab*
Prefix *ta-:*	ta-*kattab*	ta-*kassab*
Infix *t*:	*k*<t>*atab*	*k*<*t*>*asab*

- How do you compute?

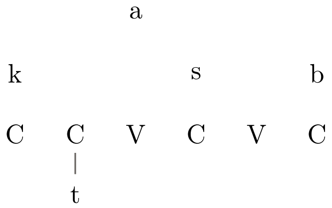
Templates + Preassociated affixation

- Many templates consist of CV-template (for root) + affixes
  - Base: *katab* *kasab*
  - Infix G/μ : *kat.**t**ab* *kas.**s**ab*
  - Prefix *ta-:* ta-*kattab* ta-*kassab*
  - Infix *t*: *k*<t>*atab* *k<t>asab*
- How do you compute?
  - Again, depends on *representation* vs *derivation*

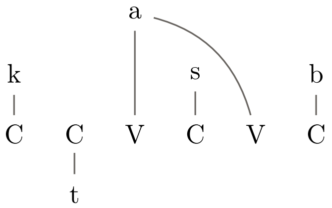
Pre-association: representation

Twist: Derive infixed *k*<t>*asab* via pre-associated infix *t* in template

Input: pre-associated infix *t*



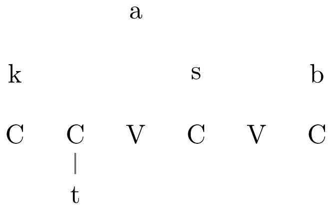
Output: C can't associate twice



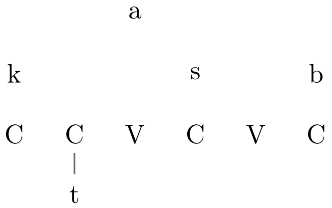
Pre-association: representation

Twist: Derive infixed *k*<t>*asab* via pre-associated infix *t* in template

Input: pre-associated infix *t*



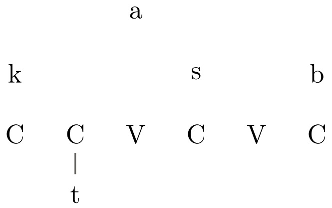
Output: C can't associate twice



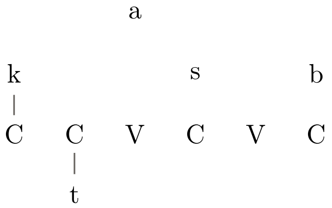
Pre-association: representation

Twist: Derive infixed *k*<t>*asab* via pre-associated infix *t* in template

Input: pre-associated infix *t*



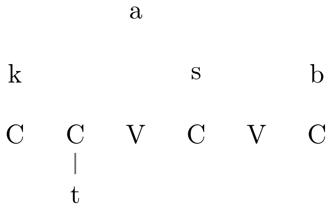
Output: C can't associate twice



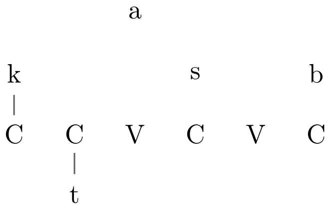
Pre-association: representation

Twist: Derive infixed *k*<t>*asab* via pre-associated infix *t* in template

Input: pre-associated infix *t*



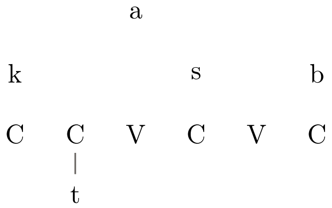
Output: C can't associate twice



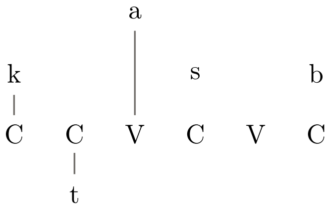
Pre-association: representation

Twist: Derive infixed *k*<t>*asab* via pre-associated infix t in template

Input: pre-associated infix t



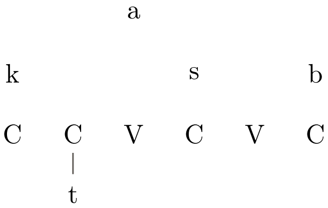
Output: C can't associate twice



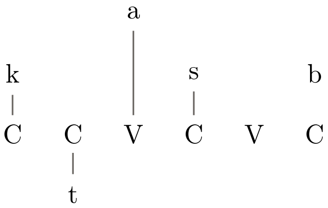
Pre-association: representation

Twist: Derive infixed *k*<t>*asab* via pre-associated infix *t* in template

Input: pre-associated infix *t*



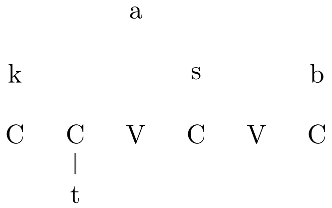
Output: C can't associate twice



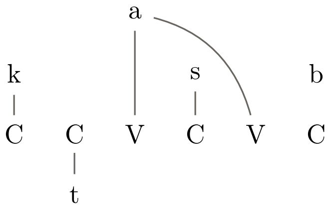
Pre-association: representation

Twist: Derive infixed *k*<t>*asab* via pre-associated infix *t* in template

Input: pre-associated infix *t*



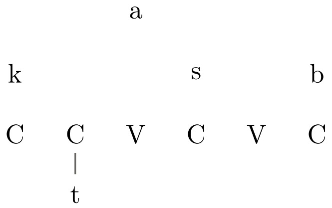
Output: C can't associate twice



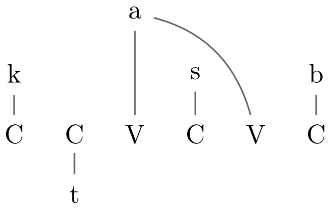
Pre-association: representation

Twist: Derive infixed *k*<t>*asab* via pre-associated infix *t* in template

Input: pre-associated infix *t*



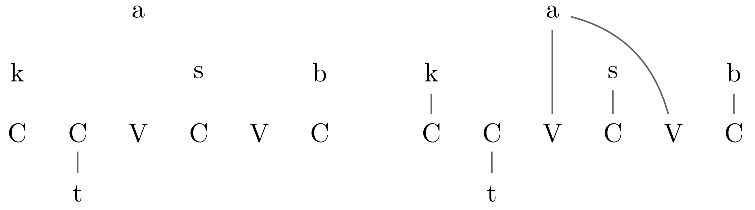
Output: C can’t associate twice



Pre-association: representation

Twist: Derive infixed *k*<t>*asab* via pre-associated infix *t* in template

Input: pre-associated infix *t* | Output: C can't associate twice



General MT-FST implementation

Each morpheme is its own tier (McCarthy, 1981)...

But can't 'clearly' enocde pre-associated edges in MT-FST

Input:

V: ⋊ a ⋉

C: ⋊ k s b ⋉

T: ⋊ C C V C V C ⋉

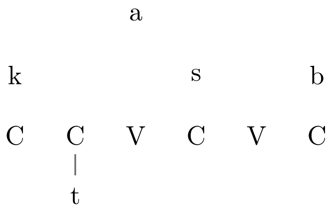
A: ⋊ t ⋉

Output:

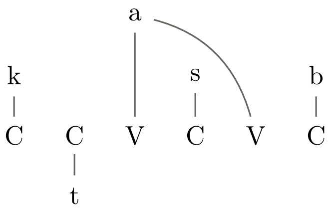
Pre-association: representation

Twist: Derive infixed *k*<t>*asab* via pre-associated infix *t* in template

Input: pre-associated infix *t*

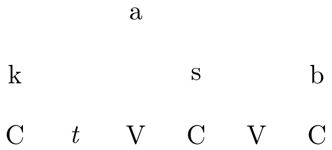


Output: C can't associate twice

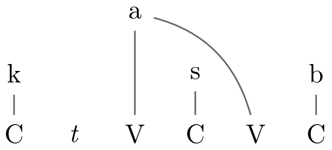


Solution: collapse pre-associated edges into Template

Input: pre-associated infix *t* inside T



Output: C can't associate twice

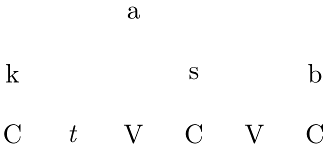


Pre-association: representation

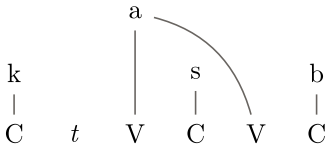
Pre-association: representation

Solution: collapse pre-associated edges into Template

Input: pre-associated infix *t* inside T



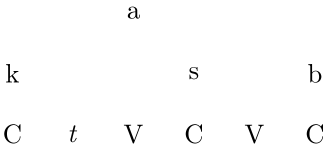
Output: C can't associate twice



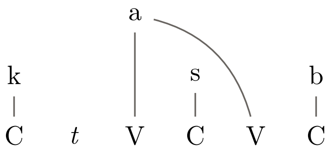
Pre-association: representation

Solution: collapse pre-associated edges into Template

Input: pre-associated infix *t* inside T



Output: C can't associate twice



General MT-FST implementation
Template alphabet now includes pre-associated segments

Input:

V:	$\rtimes$	a	$\ltimes$				
C:	$\rtimes$	k	s	b	$\ltimes$		
T:	$\rtimes$	C	t	V	C	V	C	$\ltimes$

Output:

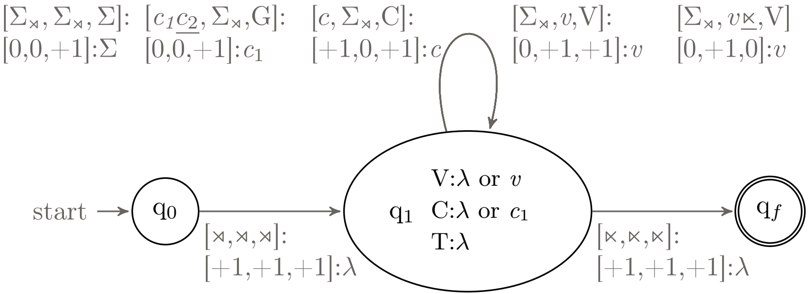
Pre-association: representation

General MT-FST implementation

Template alphabet now includes pre-associated segments (Σ)

Input: | | | | | | | | Output:

V:	⋊	a	⋉					
C:	⋊	k	s	b	⋉			
T:	⋊	C	t	V	C	V	C	⋉



Pre-association: representation

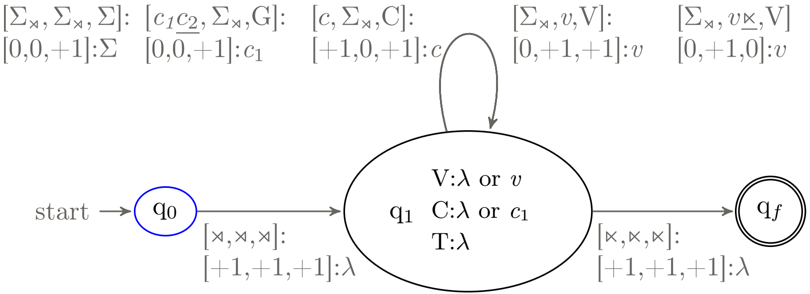
General MT-FST implementation

Template alphabet now includes pre-associated segments (Σ)

Input:

V:	⋊	a	⋉					
C:	⋊	k	s	b	⋉			
T:	⋊	C	t	V	C	V	C	⋉

Output:



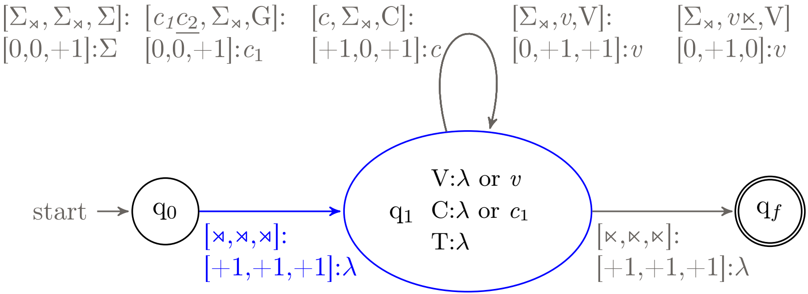
Pre-association: representation

General MT-FST implementation

Template alphabet now includes pre-associated segments (Σ)

Input: Output:

V:	⋊	a	⋉					
C:	⋊	k	s	b	⋉			
T:	⋊	C	t	V	C	V	C	⋉



Pre-association: representation

General MT-FST implementation

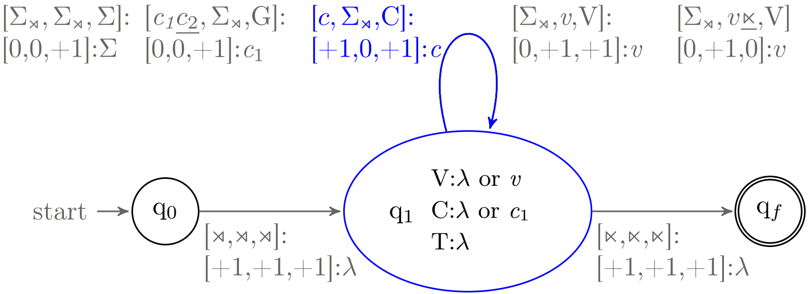
Template alphabet now includes pre-associated segments (Σ)

Input:

V:	⋊	a	⋉					
C:	⋊	k	s	b	⋉			
T:	⋊	C	t	V	C	V	C	⋉

Output:

k



Pre-association: representation

General MT-FST implementation

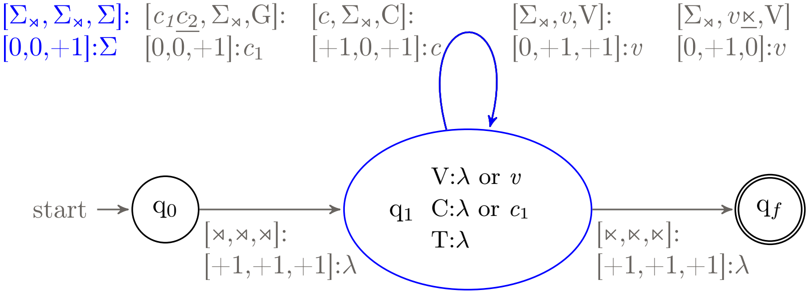
Template alphabet now includes pre-associated segments (Σ)

Input:

V:	⋊	a	⋉					
C:	⋊	k	s	b	⋉			
T:	⋊	C	t	V	C	V	C	⋉

Output:

k t

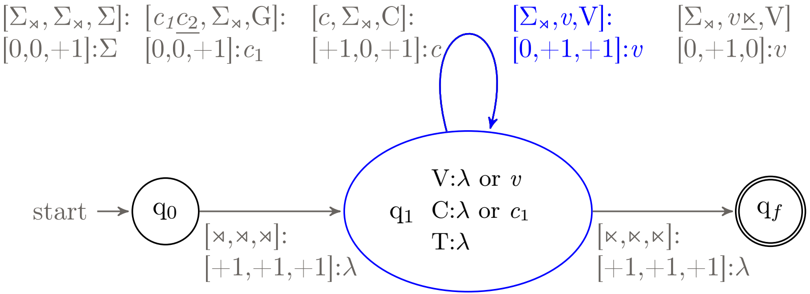


Pre-association: representation

General MT-FST implementation

Template alphabet now includes pre-associated segments (Σ)

	Input:								Output:		
V:	⋊	a	⋉								
C:	⋊	k	s	b	⋉						
T:	⋊	C	t	V	C	V	C	⋉	k	t	a



Pre-association: representation

General MT-FST implementation

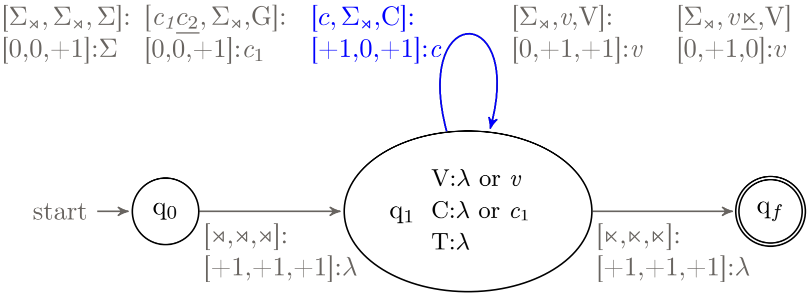
Template alphabet now includes pre-associated segments (Σ)

Input:

V:	⋊	a	⋉					
C:	⋊	k	s	b	⋉			
T:	⋊	C	t	V	C	V	C	⋉

Output:

k t a s



Pre-association: representation

General MT-FST implementation

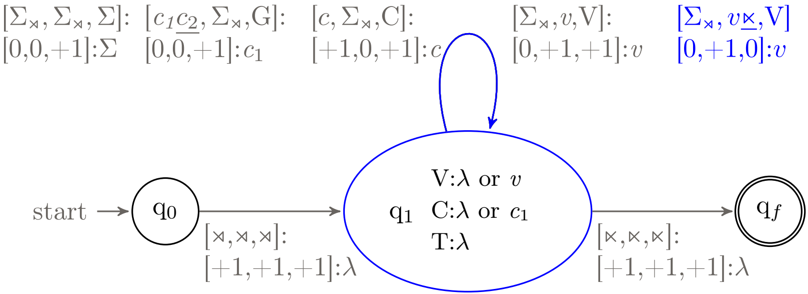
Template alphabet now includes pre-associated segments (Σ)

Input:

V:	⋊	a	⋉					
C:	⋊	k	s	b	⋉			
T:	⋊	C	t	V	C	V	C	⋉

Output:

k t a s a



Pre-association: representation

General MT-FST implementation

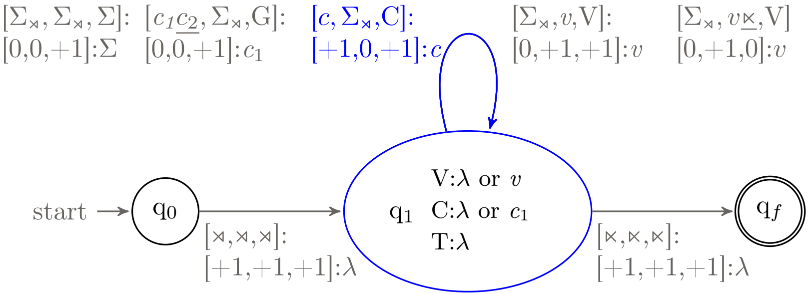
Template alphabet now includes pre-associated segments (Σ)

Input:

V:	$\rtimes$	a	$\ltimes$				
C:	$\rtimes$	k	s	b	$\ltimes$		
T:	$\rtimes$	C	t	V	C	V	C	$\ltimes$

Output:

k t a s a b

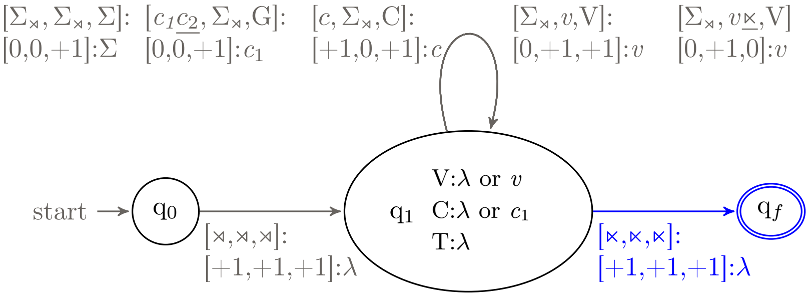


Pre-association: representation

General MT-FST implementation

Template alphabet now includes pre-associated segments (Σ)

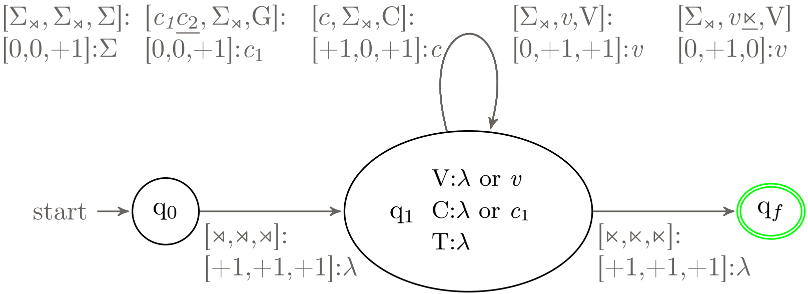
	Input:								Output:					
V:	⋊	a	⋉											
C:	⋊	k	s	b	⋉									
T:	⋊	C	t	V	C	V	C	⋉	k	t	a	s	a	b



Pre-association: representation

General MT-FST implementation
Template alphabet now includes pre-associated segments (Σ)

	Input:								Output:
V:	⋊	a	⋉						
C:	⋊	k	s	b	⋉				☺
T:	⋊	C	t	V	C	V	C	⋉	k t a s a b

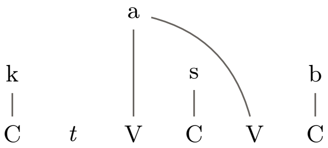


Preassociation: Derivation

- Model preassociation either representationally or derivationally

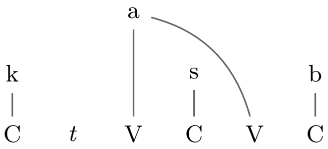
Preassociation: Derivation

- Model preassociation either representationally or derivationally
  1. Representation: segments as part of *Template* [1,1,1]-MISL



Preassociation: Derivation

- Model preassociation either representationally or derivationally
  1. Representation: segments as part of *Template* [1,1,1]-MISL



  2. Derivation

2.1	Input	{ksb, a, CV.CVC}	
2.2	Intermediate	ka.sab	[1,1,1]-MISL
2.3	Infix	k**t**a.sab	2-ISL

- Again, composition vs. sequential

Interim summary

Computing templates is local over MT-FST

Interim summary

Computing templates is local over MT-FST

Is everything about templaes local?

Interim summary

Computing templates is local over MT-FST

Is everything about templaes local?

- No ☺
- Appendix

Issues in pre-associated morphemes

- Pre-associated morphemes aren't part of "template filling'
- Only root C + inflectional vowels are part of template filling
  - *kasab* vs *k*t*asab*
- But root consonants are effected by morpheme-specific rules:

‣ Root:	*ksb*	*wʕd*
‣ Base	*kasab*	*waʕad*
‣ Infix <*t*>:	*k*<t>*asab*	**w*<t>*aʕad*
		ttaʕad

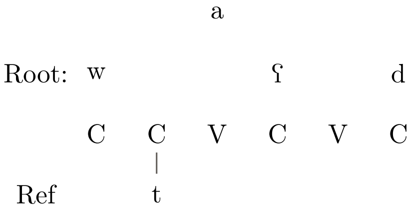
- $w_{\text{root}} \rightarrow t \; <t>_{\textit{refl}}$
  - Other morphemes don't trigger this: *gazaw-ta*

Issues in pre-associated morphemes

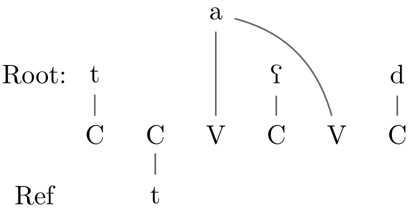
Rule references pre-associated morphemes in graph:

- $w_{\text{root}} \rightarrow t$ <t>$_{\mathit{refl}}$

Input: pre-associated infix *t*



Output: glide assimilates to reflexive



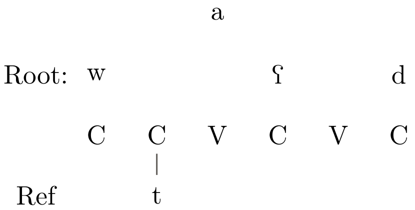
Morpheme-specific rules

- Again, representation vs. derivation...

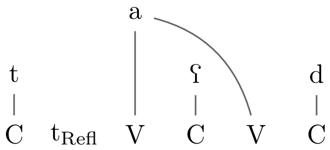
1 **Representation**

  - Flattened graph-associations lines with diacritics

Graph with pre-associations



Flattened pre-associations + diacritics



- Flattened pre-associations → assimilation is a local MISL rule
- MT-FST takes as input a template with a richer alphabet
  Input:

V:	⋊	a	⋉				
C:	⋊	w	ʕ	d	⋉		
T:	⋊	C	t_{Ref}	V	C	V	C	⋉

Morpheme-specific rules

- Again, representation vs. derivation...

General MT-FST implementation (Representation)

- Template alphabet now includes pre-associated segments (Σ)
- MT-FST takes as input a template with a richer alphabet

Input:

V:	$\rtimes$	a	$\ltimes$					
C:	$\rtimes$	w	ʕ	d	$\ltimes$			
T:	$\rtimes$	C	t_{Ref}	V	C	V	C	$\ltimes$

- Basic idea:
  - When you see a glide, don't output it
  - Wait to see the next item on T tape
  - If it's t_Ref then output *tt*
  - Else, output glide and continue normaly

Morpheme-specific rules

- Representational approach:
  - Template is $Ct_{Ref}V.CVC$ [1,1,2]-2-ISL
- As for derivational approach...

Root:	*wʕd*	
Base	*waʕad*	[1,1,1]-MISL
Infix <*t*>:	*w<t_{Refl}>aʕad	2-ISL
Assimilation	*ttaʕad*	2-ISL

- Composition vs. sequences

Handling pre-associations with "synchronity" and emptiness

- Previous MT FSTs were all *asynchronous*
  - Can move +1 on one tape but stay put on another
- Can encode morpho information as *synchronous* tapes
  - Each morpheme is its own tape
  - Must move in same direction on *some* (preassociated) tapes
  - Unassociated morphemes 'float' around with extra empty-string padding
  - Synchronous MT look like *elegant* encoding
  - But computationally equivalent to single tape FSTs

V:	$\rtimes$	a	$\ltimes$									
C:	$\rtimes$		w	ʕ	d	$\ltimes$						
T:	$\rtimes$					C	C	V	C	V	C	$\ltimes$
Refl:	$\rtimes$					t					$\ltimes$	

Whats not local?

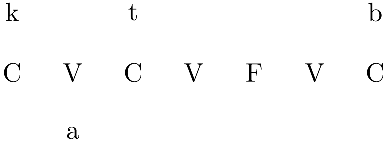
- A lot of templates can be computated locally
- Same for Semitic...
  - allomorphy (Kastner, 2016)
  - lexical semantics (Arad, 2003)
  - and phonology! (us)
- What are logically possible non-local patterns?
  - first-C copying: CV.CVC.FVC → ka.ta.kab
  - Root Reduplication
  - Edge-In Effects

Remember first consonant

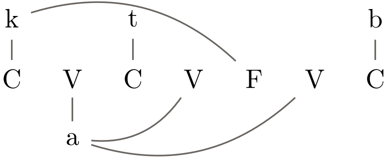
Hypothetical template – First-C copying

- Template: *CV.CV.FVC*
- F=output the first C again

Input: Copy-First template



Output: *ka.ta.**k**ab*

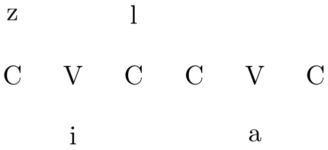


- Is MT but not MT ISL *if* no bound on 1^{st} C and F
- Arabic gets close to it
  - Base: *barad*
  - Derivative: *bar.**b**ad*
- Initial C reduplication is MT-3-ISL because
  - *b* is always bounded-ly close to (=1 segments apart from) F across the C tape

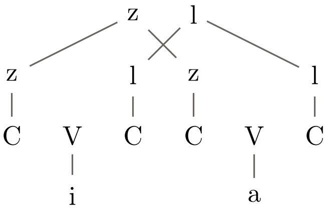
Root reduplication

Some words have root reduplication: ***zil.zal***

Input: single root *zl*



Output: ***zil.zal***



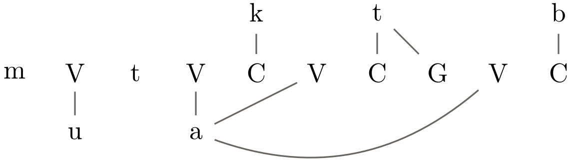
- Derivation:
  1. Root — *zl*
  2. Reduplicate root: — *zl-zl* — ?
  3. + template, vowel: — *zl-zl*, *ia*, *CVC.CVC*
  4. Fill: — *zil.zal* — [1,1,1]-MISL
- How powerful is reduplication?[1]
  - 1-ISL if reduplicant is *bounded* and *contiguous*
  - C-1-OSL over 2-way FSTs (matches reduplicative theory better)

---

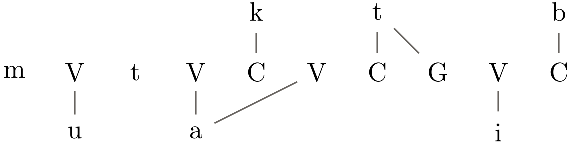
[1] (Chandlee, 2017; Dolatian, In press.)

Edge-in effects

- Left-right Vowel spread for *ua* → *m**u**.t**a**.k**a**t.t**a**b*



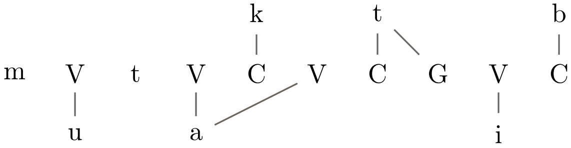
- But final *i* does not spread: *uai* → *m**u**.t**a**.k**a**t.t**i**b*



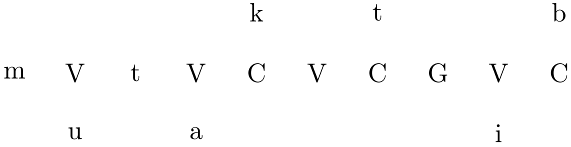
- How can you do that?
  - **Edge-in**: associate the edges first! (Hoberman, 1988)

Edge-in effects

- Desired output: *uai* → *m**u**.t**a**.k**a**t.t**i**b*

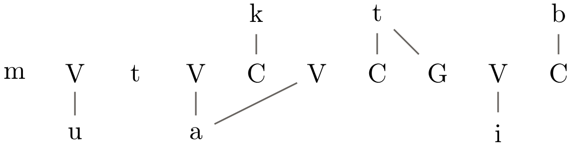


- Edge-in strategy: associate final *i* first + then left-right spread

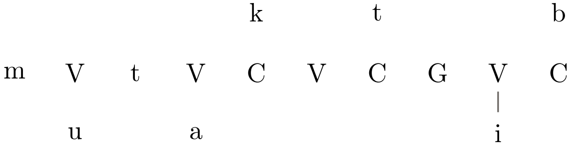


Edge-in effects

- Desired output: *uai* → *m**u**.t**a**.k**a**t.t**i**b*

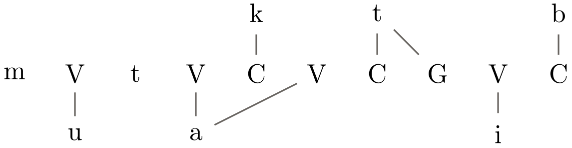


- Edge-in strategy: associate final *i* first + then left-right spread

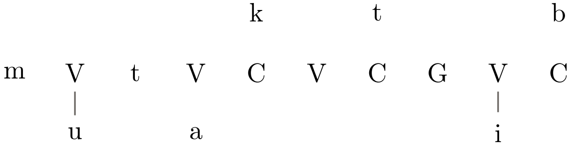


Edge-in effects

- Desired output: *uai* → _m**u**.t**a**.k**a**t.t**i**b_

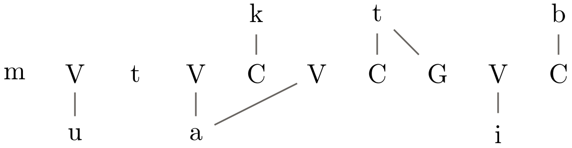


- Edge-in strategy: associate final *i* first + then left-right spread

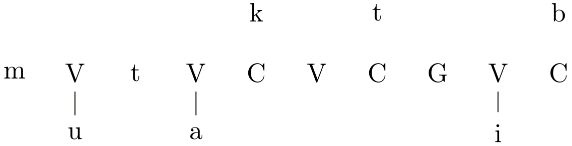


Edge-in effects

- Desired output: *uai* → *m**u**.t**a**.k**a**t.t**i**b*

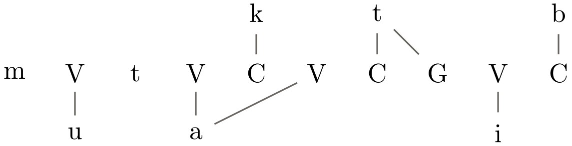


- Edge-in strategy: associate final *i* first + then left-right spread

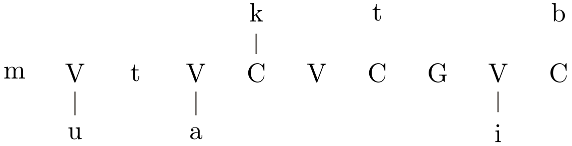


Edge-in effects

- Desired output: *uai* → *m**u**.t**a**.k**a**t.t**i**b*

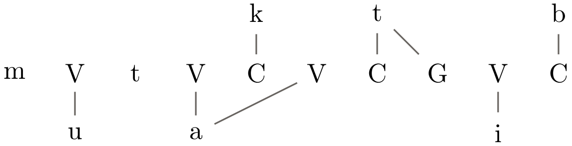


- Edge-in strategy: associate final *i* first + then left-right spread

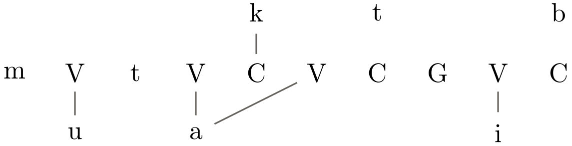


Edge-in effects

- Desired output: *uai* → *m*_**u**_*.t*_**a**_*.k*_**a**_*t.t*_**i**_*b*

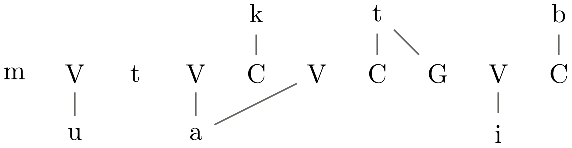


- Edge-in strategy: associate final *i* first + then left-right spread

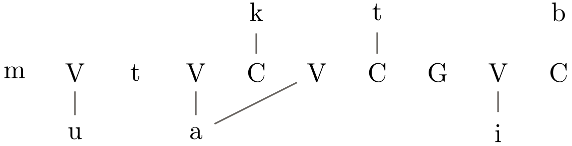


Edge-in effects

- Desired output: *uai* → *m**u**.t**a**.k**a**t.t**i**b*

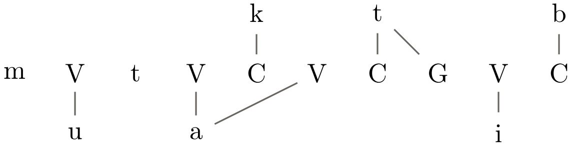


- Edge-in strategy: associate final *i* first + then left-right spread

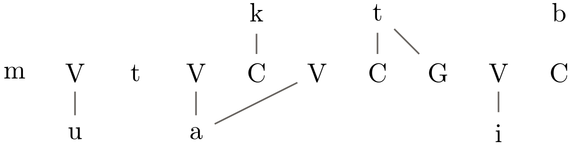


Edge-in effects

- Desired output: *uai* → *m**u**.t**a**.k**a**t.t**i**b*

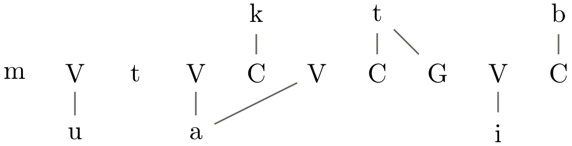


- Edge-in strategy: associate final *i* first + then left-right spread

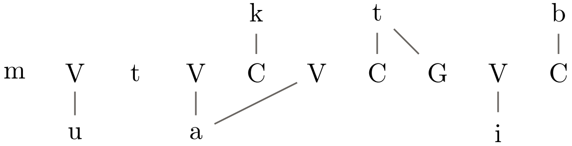


Edge-in effects

- Desired output: *uai* → *m**u**.t**a**.k**a**t.t**i**b*



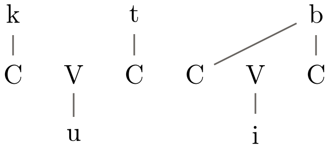
- Edge-in strategy: associate final *i* first + then left-right spread



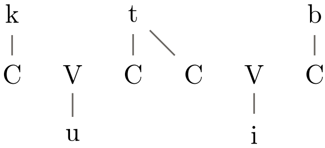
Edge-in effects

- Similar edge-in algorithms proposed for *ka**t.t**ab* vs. *kat.**b**a**b***

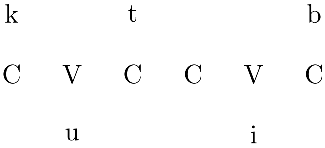
Final spread: *kut.**b**i**b***



Medial spread: *ku**t.t**ib*



- To trigger medial spread, not final spread...
  - McCarthy: derive *kut.tib* from *kut.bib* by reassociation
  - Representational trick: Geminate template CVC.GVC
  - But *ku**t.t**ib* is a common pattern while *kut.**b**i**b*** is rare!
- Edge-in alternative: Default is associate edges (C+V) first!



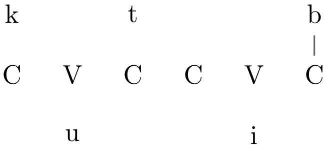
Edge-in effects

- Edge-in alternative: Default is associate edges (C+V) first!

k		t			b
C	V	C	C	V	C
	u			i	

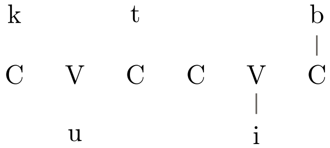
Edge-in effects

- Edge-in alternative: Default is associate edges (C+V) first!



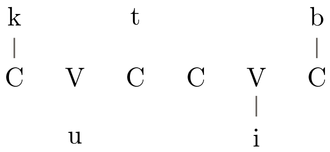
Edge-in effects

- Edge-in alternative: Default is associate edges (C+V) first!



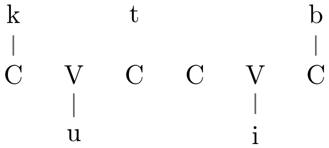
Edge-in effects

- Edge-in alternative: Default is associate edges (C+V) first!



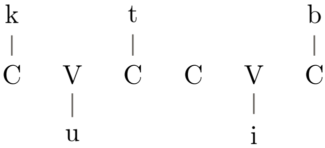
Edge-in effects

- Edge-in alternative: Default is associate edges (C+V) first!



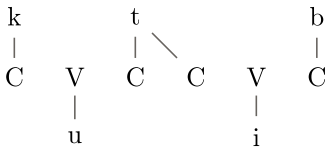
Edge-in effects

- Edge-in alternative: Default is associate edges (C+V) first!



Edge-in effects

- Edge-in alternative: Default is associate edges (C+V) first!



Edge-in effects: loanword adaption

- Verbs formed from loanwords have *CVC.CVC* template
- If loanword has 4 Cs:
  - **telefon** ‘telephone’
  - **talfan** ‘to telephone’
- If loanword has 3 Cs:
  - **ʃarʒ**-or ‘charger’
  - **ʃar.raʒ** ‘to charge’
- Edge-in effect again

1. Input:	tlfn	ʃrʒ
	CVC.CVC	*CVC.CVC*
2. Right edge:	CVC.CV**n**	CVC.CV**ʒ**
3. Left edge:	**t**VC.CV**n**	**ʃ**VC.CV**ʒ**
4. Left-right:	**t**V**l**.CV**n**	**ʃ**V**r**.CV**ʒ**
	tV**l**.fV**n**	**ʃ**V**r**.**r**V**ʒ**

Edge-in effects

- Intuition: simple and nice
- Computational: local too... we think
- What does the machine need to do?
    - Right-edge machine: read machine from right-edge (MISL)
    - Left-edge machine: do left-right MISL for leftovers in the template
- Alternative
    - if read C$\underline{\text{C}}$ (= on T), then check if on final consonant (=on C) and geminate
- Is edge-in effect MISL? To be determined

[Arad 2003] Arad, Maya: Locality constraints on the interpretation of roots: The case of Hebrew denominal verbs. In: Natural Language & Linguistic Theory 21 (2003), Nr. 4, S. 737–778

[Chandlee 2017] Chandlee, Jane: Computational locality in morphological maps. In: Morphology (2017), S. 1–43

[Dolatian In press.] Dolatian, Hossep: Armenian prosody: a case for prosodic stems. In: Proceedings of the 53rd Annual Meeting of the Chicago Linguistics Society. In press.

[Hoberman 1988] Hoberman, Robert D.: Local and long-distance spreading in Semitic morphology. In: Natural Language & Linguistic Theory 6 (1988), Nr. 4, S. 541–549

[Kastner 2016] Kastner, Itamar: Form and meaning in the Hebrew verb, New York University, Dissertation, 2016

[McCarthy 1981] McCarthy, John J.: A prosodic theory of nonconcatenative morphology. In: Linguistic inquiry 12 (1981), Nr. 3, S. 373–418