Computational Computational Linguistics

Thomas Graf
`mail@thomasgraf.net`
`http://thomasgraf.net`

Department of Linguistics

December 4 2013

Two Types of Computational Linguistics

- **Computational** Linguistics (NLP)
  computers solving natural language tasks
  - machine translation
  - text summarization
  - OCR
  - speech recognition
  - dialog-driven user interfaces

    ⋮

- Computational **Linguistics**
  linguistics with methods from theoretical computer science
  - What are the computational properties of natural language? (Marr Level 1 & 2)
  - What are the computational properties of linguistic theories?

Today's Topic

Questions

- How is natural language computed?
- In particular, what kind of memory is required?

Answer

- **"Naive" perspective**
  different subsystems use different memory systems
- **Linguistic perspective**
  different subsystems use same memory system,
  but different data structures

Today's Topic

Questions

- How is natural language computed?
- In particular, what kind of memory is required?

Answer

- **"Naive" perspective**
  different subsystems use different memory systems
- **Linguistic perspective**
  different subsystems use same memory system,
  but different data structures

Remark on Methodology

How these issues would be approached by

Experimentalists

- Design and run experiments
- Carry out statistical analysis
- For bonus points: Design model that replicates the statistical patterns

Computational linguists

- Look at linguistic patterns
- What type of grammar can generate them?
- What computational resources does the grammar need?

Outline

1. Linguistic Subsystems: Syntax and Phonology
2. Strings, Automata, and Memory
    - Formal Language Theory
    - Automata
    - Memory Requirements of Phonology and Syntax
3. A Linguistically Informed Look at Syntax
    - Minimalist Syntax
    - A Quick Example
    - Tree Structures and Memory

Linguistic Subsystems

Linguists distinguish several areas of language.

- Phonology: sounds and prosody
- Morphology: word forms
- Syntax: sentence structure
- Semantics: logical meaning
- Pragmatics: meaning in context

Computational linguists are mostly interested in structure rather than meaning ⇒ **phonology, morphology, syntax**

Phonological Patterns

- Only certain sound sequences are licit.
- Vowel systems show regularities.
  a-i-u, a-e-i-o-u, *e-o-i
- Sounds can be affected by their contexts,
  but only in specific ways.

intervocalic voicing	*nef+ið* → *nevið*	Icelandic
word-final devoicing	*rad* → *rat*	German
*intervocalic devoicing	*aba* → *apa*	unattested
umlaut	*mamm+u* → *mömmu*	Icelandic
dissimilation	*lun+alis* → *lunaris*	Latin
*anti-umlaut	*mömm+u* → *mammu*	unattested

Syntactic Patterns

- Island effects

(1) a. Which man did John say that Mary kissed?
b. * Which man did John cry because Mary kissed?

- Center-embedding

(2) a. The mouse that the cat that the dog chased ate is dead.
b. * The mouse that the cat that the dog chased ate is dead.

- Crossing dependencies

(3) a. The mouse, the cat, and the dog survived, slept, and chewed on a toy, respectively.
b. * The mouse, the cat, and the dog survived, slept, and chewed on a toy, respectively.

The General Issue

- Language is a harsh mistress, it's not "anything goes".
- In every language only certain patterns are allowed.
- Linguists devise models that account for those patterns while also ruling out unattested ones.
- But the kind of patterns differ between phonology and syntax.

Questions

- What kind of computational device generates all the correct patterns but none of the incorrect ones?
- Does this device work for phonology as well as syntax?

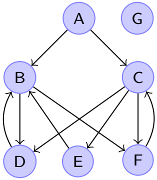
Outline

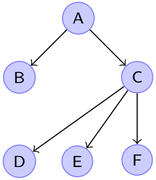
1. Linguistic Subsystems: Syntax and Phonology
2. Strings, Automata, and Memory
    - Formal Language Theory
    - Automata
    - Memory Requirements of Phonology and Syntax
3. A Linguistically Informed Look at Syntax
    - Minimalist Syntax
    - A Quick Example
    - Tree Structures and Memory

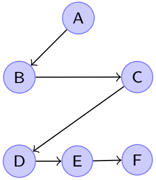
Language as Sets

In computer science, a language is simply **a set of objects of a specific type**:

- **graph**: structure of connected nodes
  flow chart, *street network*, *Wikipedia*, *internet*, *video game AI*
- **tree**: connected graph where every node is reachable from at most one node
  family tree, *hard drive layout*, *XML file*
- **string**: sequence of nodes
  telephone number, *Python program*, *human genome*, *Shakespeare's oeuvre*

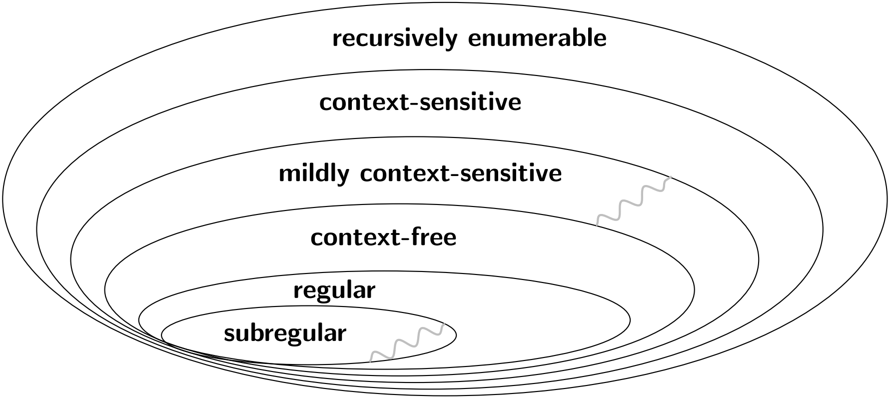






The Chomsky Hierarchy of String Languages

- The perceivable output of language is strings (sequences of sound waves, words, sentences).
- The complexity of string languages is measured by the (extended) **Chomsky hierarchy**. (Chomsky 1956, 1959)



Languages and Automata

- For every language class there is a computational model that can generate all languages in the class, and only those.
- Such a model is called an **automaton**.

	Example Language	**Automaton Model**
RE	theorems of first-order logic	Turing Machine
CS	all prime numbers	linear bounded automaton
MCS	crossing dependencies	embedded pushdown autom.
CF	center embedding	pushdown automaton
REG	all strings of even length	finite-state automaton
subREG	umlaut, voicing	

Finite-State Automata

A **finite-state automaton** (FSA) assigns every node in a string one of finitely many *states*, depending on

- the label of the node, and
- the state of the preceding node (if it exists).

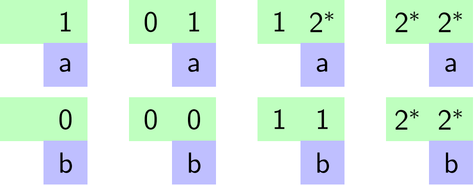
The FSA accepts the string if the last state is a *final state*.

Cognitive Intuition

- States are a metaphor for memory configurations.
- Every symbol in the input induces a change from one memory configuration into another.
- Only finitely many memory configurations are needed. Thus the amount of working memory used by the automaton is finitely bounded.

Example 1: Counting Symbols

FSA for strings over *a* and *b* where *a* occurs at least 2 times



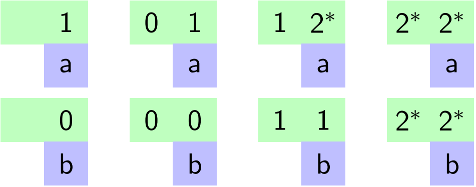
1 / a   0 1 / a   1 2^* / a 2^* 2^* / a

0 / b   0 0 / b   1 1 / b 2^* 2^* / b

b a b a a b

Example 1: Counting Symbols

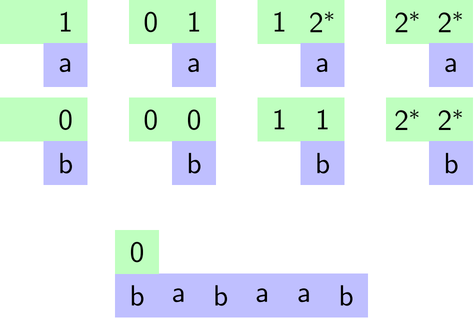
FSA for strings over *a* and *b* where *a* occurs at least 2 times



b a b a a b

Example 1: Counting Symbols

FSA for strings over *a* and *b* where *a* occurs at least 2 times

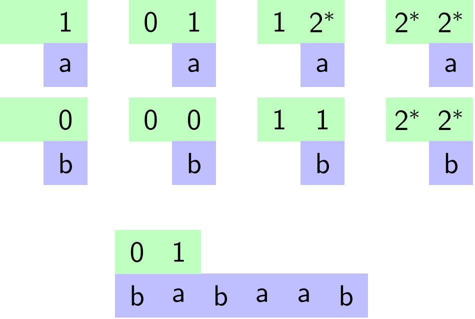


	1	0	1	1	2^*	2^*	2^*
	a		a		a		a
	0	0	0	1	1	2^*	2^*
	b		b		b		b

0					
b	a	b	a	a	b

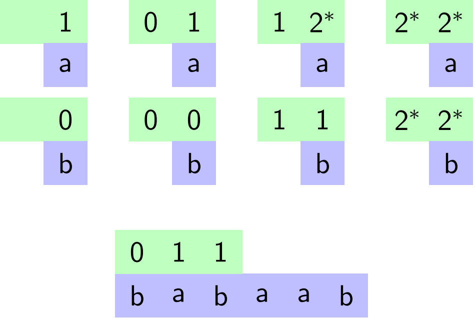
Example 1: Counting Symbols

FSA for strings over *a* and *b* where *a* occurs at least 2 times



Example 1: Counting Symbols

FSA for strings over *a* and *b* where *a* occurs at least 2 times



	1
	a

0	1
	a

1	2^*
	a

2^*	2^*
	a

	0
	b

0	0
	b

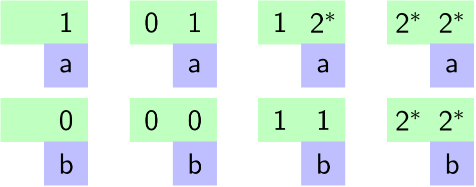
1	1
	b

2^*	2^*
	b

0	1	1			
b	a	b	a	a	b

Example 1: Counting Symbols

FSA for strings over *a* and *b* where *a* occurs at least 2 times

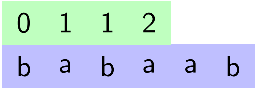


1   0 1   1 2^* 2^* 2^*

a   a   a   a

0   0 0   1 1 2^* 2^*

b   b   b   b

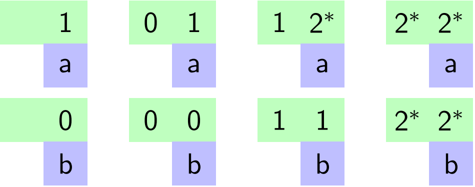


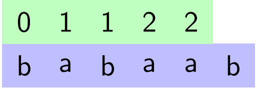
0 1 1 2

b a b a a b

Example 1: Counting Symbols

FSA for strings over *a* and *b* where *a* occurs at least 2 times





0	1	1	2	2	
b	a	b	a	a	b

Example 1: Counting Symbols

FSA for strings over *a* and *b* where *a* occurs at least 2 times

	1
	a

0	1
	a

1	2^*
	a

2^*	2^*
	a

	0
	b

0	0
	b

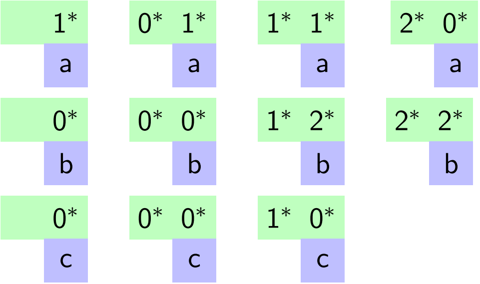
1	1
	b

2^*	2^*
	b

0	1	1	2	2	2
b	a	b	a	a	b

Example 2: Remembering Symbols

Strings over *a*, *b*, *c* where no *b* occurs between *a* and *c*

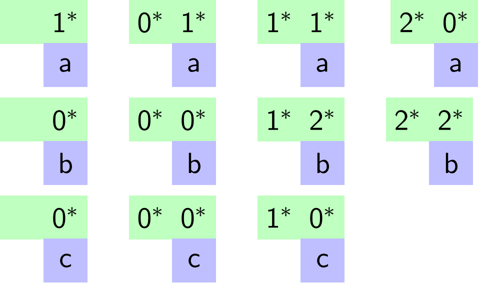


1^* a	0^* 1^* a	1^* 1^* a	2^* 0^* a
0^* b	0^* 0^* b	1^* 2^* b	2^* 2^* b
0^* c	0^* 0^* c	1^* 0^* c	

b c a b b a c

Example 2: Remembering Symbols

Strings over *a*, *b*, *c* where no *b* occurs between *a* and *c*



1^*
a

0^* 1^*
a

1^* 1^*
a

2^* 0^*
a

0^*
b

0^* 0^*
b

1^* 2^*
b

2^* 2^*
b

0^*
c

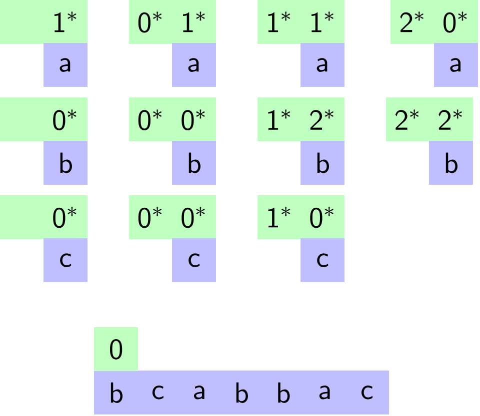
0^* 0^*
c

1^* 0^*
c

b c a b b a c

Example 2: Remembering Symbols

Strings over *a*, *b*, *c* where no *b* occurs between *a* and *c*



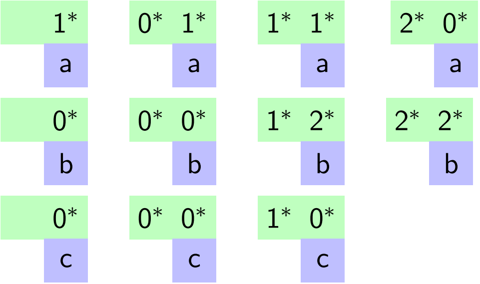
1* a	0* 1* a	1* 1* a	2* 0* a
0* b	0* 0* b	1* 2* b	2* 2* b
0* c	0* 0* c	1* 0* c	

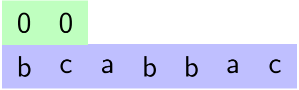
0

b c a b b a c

Example 2: Remembering Symbols

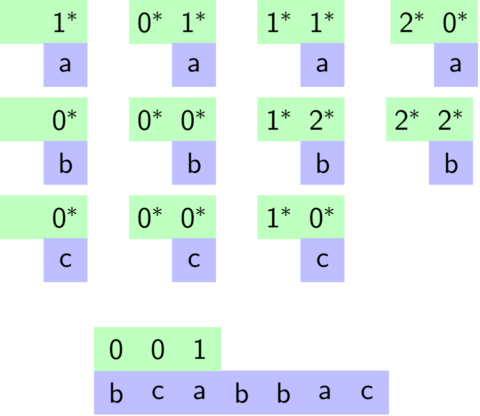
Strings over *a*, *b*, *c* where no *b* occurs between *a* and *c*





Example 2: Remembering Symbols

Strings over *a*, *b*, *c* where no *b* occurs between *a* and *c*



1^*
a

0^* 1^*
a

1^* 1^*
a

2^* 0^*
a

0^*
b

0^* 0^*
b

1^* 2^*
b

2^* 2^*
b

0^*
c

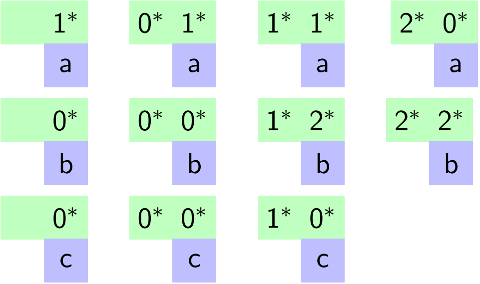
0^* 0^*
c

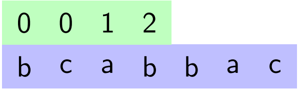
1^* 0^*
c

0 0 1
b c a b b a c

Example 2: Remembering Symbols

Strings over *a*, *b*, *c* where no *b* occurs between *a* and *c*

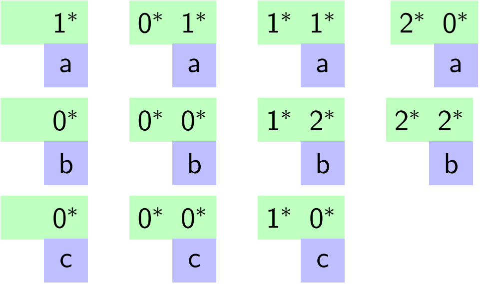


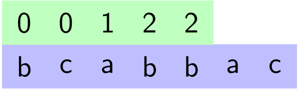


0	0	1	2			
b	c	a	b	b	a	c

Example 2: Remembering Symbols

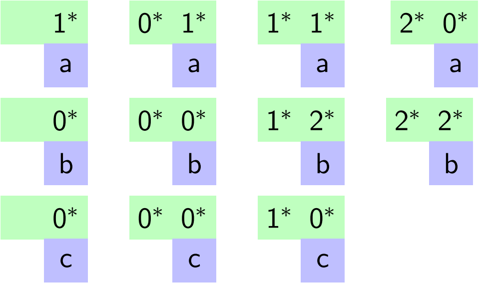
Strings over *a*, *b*, *c* where no *b* occurs between *a* and *c*





Example 2: Remembering Symbols

Strings over *a*, *b*, *c* where no *b* occurs between *a* and *c*



1^*
a

0^* 1^*
a

1^* 1^*
a

2^* 0^*
a

0^*
b

0^* 0^*
b

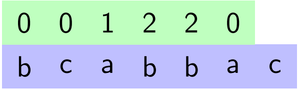
1^* 2^*
b

2^* 2^*
b

0^*
c

0^* 0^*
c

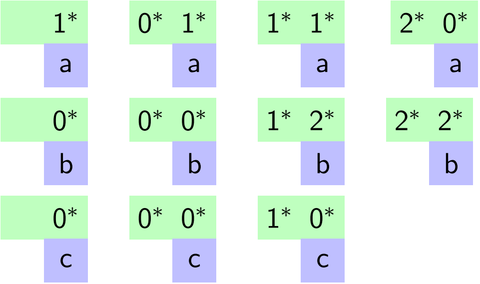
1^* 0^*
c



0 0 1 2 2 0
b c a b b a c

Example 2: Remembering Symbols

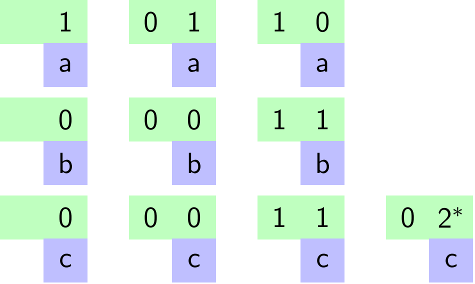
Strings over *a*, *b*, *c* where no *b* occurs between *a* and *c*



0	0	1	2	2	0	0
b	c	a	b	b	a	c

Example 3: More Sophisticated Counting

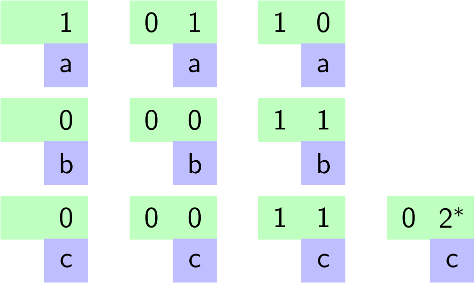
Strings over *a*, *b*, *c* with an even number of *a*s and a *c* at the end



b c a b b a c

Example 3: More Sophisticated Counting

Strings over a, b, c with an even number of as and a c at the end

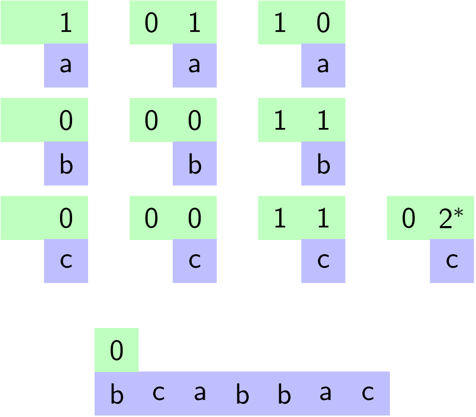


1 a	0 1 a	1 0 a	
0 b	0 0 b	1 1 b	
0 c	0 0 c	1 1 c	0 2* c

b c a b b a c

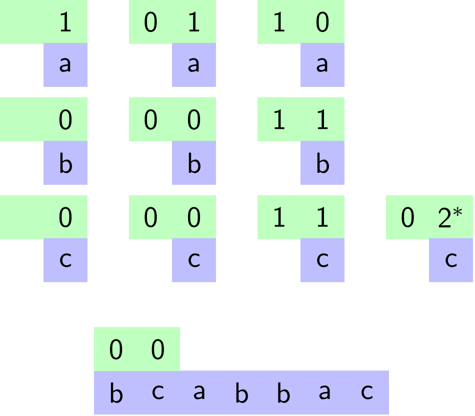
Example 3: More Sophisticated Counting

Strings over *a*, *b*, *c* with an even number of *a*s and a *c* at the end



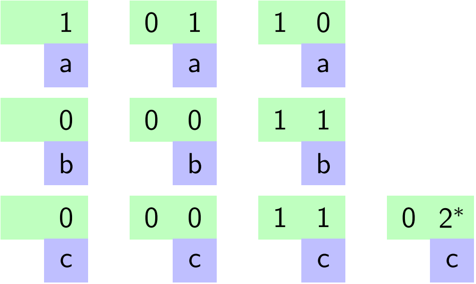
Example 3: More Sophisticated Counting

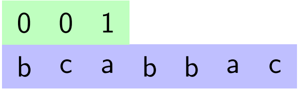
Strings over *a*, *b*, *c* with an even number of *a*s and a *c* at the end



Example 3: More Sophisticated Counting

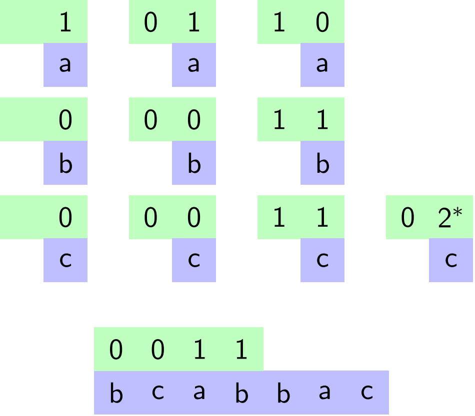
Strings over *a*, *b*, *c* with an even number of *a*s and a *c* at the end





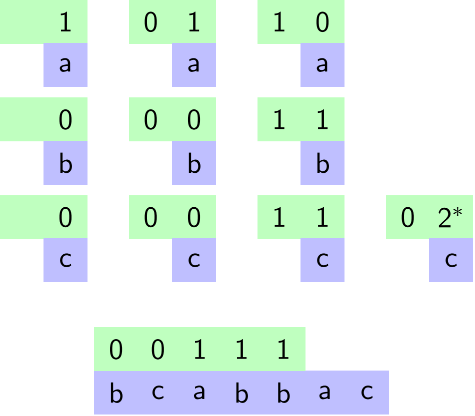
Example 3: More Sophisticated Counting

Strings over a, b, c with an even number of as and a c at the end



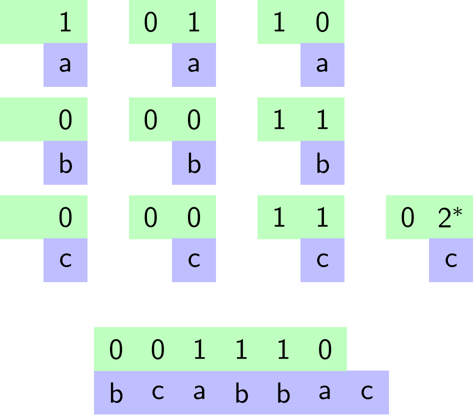
Example 3: More Sophisticated Counting

Strings over a, b, c with an even number of as and a c at the end



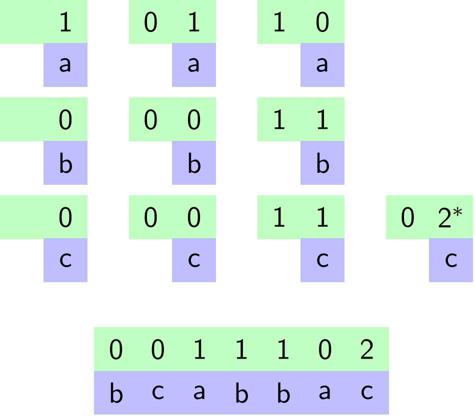
Example 3: More Sophisticated Counting

Strings over a, b, c with an even number of as and a c at the end



Example 3: More Sophisticated Counting

Strings over *a*, *b*, *c* with an even number of *a*s and a *c* at the end



	1	0	1	1	0		
	a		a		a		
	0	0	0	1	1		
	b		b		b		
	0	0	0	1	1	0	2^*
	c		c		c		c

0	0	1	1	1	0	2
b	c	a	b	b	a	c

(Embedded) Pushdown Automata

A **pushdown automaton** (PDA) is an FSA augmented with an unbounded stack of symbols. For every node in the string,

- the PDA assigns it a state depending on
  - the label of the node, and
  - the state of the preceding node (if it exists), and
  - the highest symbol on the stack (if it exists),
- depending on the state, the PDA may change the stack by
  - removing the top-most symbol, or
  - adding a new symbol on top of it.

The string is accepted if the last node is assigned a final state.

An **embedded pushdown automaton** is a PDA with a stack of stacks.

Cognitive Comparison

- **FSAs are simple**.
  - specification of how a memory configuration changes into another depending on input symbol
  - only use finitely bounded amount of working memory
- **PDAs are complex**.
  - finite memory (states) and infinite memory (stack)
  - configuration of finite and infinite memory are interlocked
  - infinite memory follows "first one in = last one out" principle

FSAs are cognitively a lot more plausible than PDAs.

Memory Requirements of Phonology and Syntax

- **Phonology**
  - Phonological patterns are regular. (Kaplan and Kay 1994)
  - A small number of patterns is not sub-regular. (Graf 2010)
  - Hence phonology can be computed by FSAs, but nothing weaker.
- **Syntax**
  - Syntax is not regular due to center embedding.
  - It is not context-free due to crossing dependencies. (Shieber 1985)
  - Computing syntactic dependencies over strings hence requires embedded pushdown automata, at the very least.

Interim Summary

- String languages can be classified according to their complexity and matched up with specific automata models.
- These automata give us some basic cognitive facts about memory usage and architecture.
- The string patterns we find in phonology and syntax differ significantly with respect to these parameters.

	Phonology	**Syntax**
Lang. Class	regular	$\geq$ mildly context-sensitive
Automaton	finite-state	embedded pushdown
Memory	finite	finite coupled with infinite

Interim Summary

- String languages can be classified according to their complexity and matched up with specific automata models.
- These automata give us some basic cognitive facts about memory usage and architecture.
- The string patterns we find in phonology and syntax differ significantly with respect to these parameters.

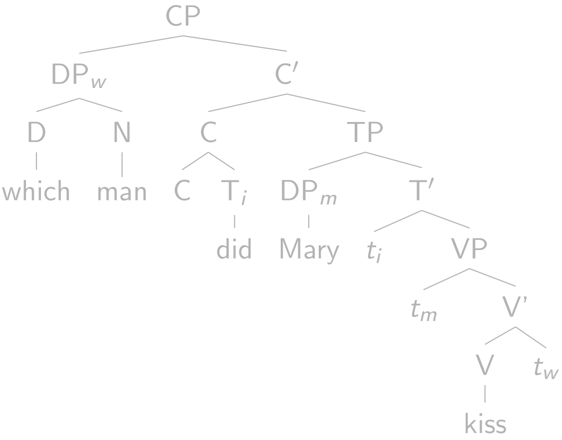
	Phonology	**Syntax**
Lang. Class	regular	$\geq$ mildly context-sensitive
Automaton	finite-state	embedded pushdown
Memory	finite	finite coupled with infinite

Outline

1. Linguistic Subsystems: Syntax and Phonology
2. Strings, Automata, and Memory
    - Formal Language Theory
    - Automata
    - Memory Requirements of Phonology and Syntax
3. A Linguistically Informed Look at Syntax
    - Minimalist Syntax
    - A Quick Example
    - Tree Structures and Memory

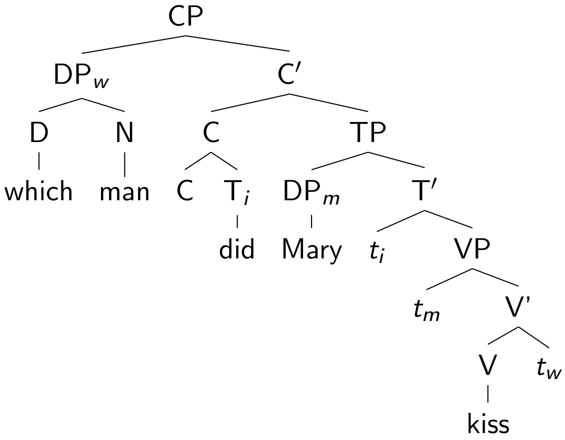
A Closer Look at Syntax

So far we have looked at syntactic patterns as string dependencies.
But **syntacticians work with trees**, not strings.



A Closer Look at Syntax

So far we have looked at syntactic patterns as string dependencies.
But **syntacticians work with trees**, not strings.



Minimalist Grammars

- **Minimalism** is the dominant syntactic theory. (Chomsky 1995)
- Can Minimalism change the computational picture of syntax? Maybe, but first we need a precise specification.
- **Minimalist grammars** are such a formalization, developed by Ed Stabler. (Stabler 1997)

Syntax as Chemistry of Language

Minimalist grammars treat syntax like chemistry.

Chemistry	**Syntax**
atoms	words
electrons	features
molecules	sentences
stable	grammatical
unstable	ungrammatical

- Every word is a collection of features.
- Every feature has either positive or negative polarity.
- Features of opposite polarity annihilate each other.
- Feature annihilation drives the structure-building operations **Merge** and **Move**.

Syntax as Chemistry of Language

Minimalist grammars treat syntax like chemistry.

Chemistry	**Syntax**
atoms	words
electrons	features
molecules	sentences
stable	grammatical
unstable	ungrammatical

- Every word is a collection of features.
- Every feature has either positive or negative polarity.
- Features of opposite polarity annihilate each other.
- Feature annihilation drives the structure-building operations **Merge** and **Move**.

Merge: Example 1

Assembling [$_{\text{DP}}$ the men]

$$\frac{\text{the}}{\mathrm{N}^{+}\ \mathrm{D}^{-}} \quad \frac{\text{men}}{\mathrm{N}^{-}}$$

- Features of opposite polarities annihilate
- Annihilation triggers Merge, which builds structure on top

Merge: Example 1

Assembling [$_{\text{DP}}$ the men]

$$\frac{\text{the}}{\mathrm{N}^{+}\ \mathrm{D}^{-}} \quad \frac{\text{men}}{\mathrm{N}^{-}}$$

- Features of opposite polarities annihilate
- Annihilation triggers Merge, which builds structure on top

Merge: Example 1

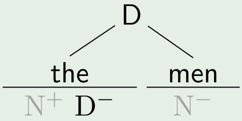
Assembling [$_{\text{DP}}$ the men]

$$\frac{\text{the}}{\text{N}^{+}\ \text{D}^{-}} \quad \frac{\text{men}}{\text{N}^{-}}$$

- Features of opposite polarities annihilate
- Annihilation triggers Merge, which builds structure on top

Merge: Example 1

Assembling $[_{\text{DP}}$ the men$]$



- Features of opposite polarities annihilate
- Annihilation triggers Merge, which builds structure on top

Merge: Example 1

Assembling [$_{\text{DP}}$ the men]

D

the $\text{N}^{+}\ \text{D}^{-}$ men N^{-}

Merge[N]

the $\text{N}^{+}\ \text{D}^{-}$ men N^{-}

- Features of opposite polarities annihilate
- Annihilation triggers Merge, which builds structure on top

Merge: Example 2

Assembling [$_{\text{VP}}$ the men like which men]

the	men	like	which	men
$N^{+}\ D^{-}$	N^{-}	$D^{+}\ D^{+}\ V^{-}$	$N^{+}\ D^{-}$	N^{-}

- *the* and *men* merged as before
- same steps for *which men*
- *like* merged with *which men*
- *like* merged with *the men*

Merge: Example 2

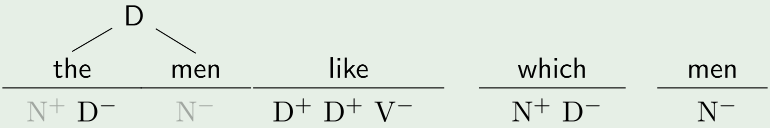
Assembling [$_{\text{VP}}$ the men like which men]

the	men	like	which	men
$N^+\ D^-$	N^-	$D^+\ D^+\ V^-$	$N^+\ D^-$	N^-

- *the* and *men* merged as before
- same steps for *which men*
- *like* merged with *which men*
- *like* merged with *the men*

Merge: Example 2

Assembling [$_{\text{VP}}$ the men like which men]



D

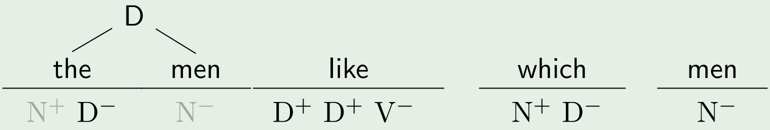
the | men | like | which | men

$\text{N}^{+}\ \text{D}^{-}$ | N^{-} | $\text{D}^{+}\ \text{D}^{+}\ \text{V}^{-}$ | $\text{N}^{+}\ \text{D}^{-}$ | N^{-}

- *the* and *men* merged as before
- same steps for *which men*
- *like* merged with *which men*
- *like* merged with *the men*

Merge: Example 2

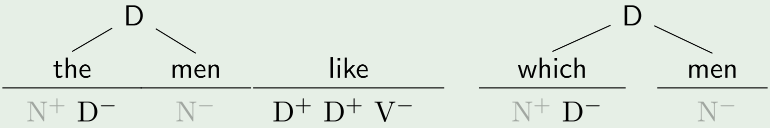
Assembling [$_{\text{VP}}$ the men like which men]



- *the* and *men* merged as before
- same steps for *which men*
- *like* merged with *which men*
- *like* merged with *the men*

Merge: Example 2

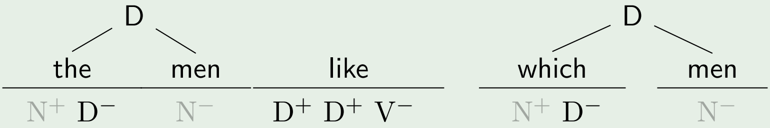
Assembling [$_{\text{VP}}$ the men like which men]



- *the* and *men* merged as before
- same steps for *which men*
- *like* merged with *which men*
- *like* merged with *the men*

Merge: Example 2

Assembling $[_{VP}$ the men like which men$]$



D D

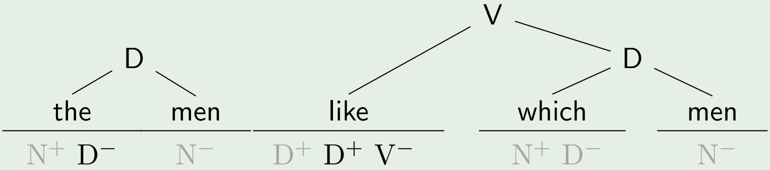
the men like which men

$N^+ \; D^-$ N^- $D^+ \; D^+ \; V^-$ $N^+ \; D^-$ N^-

- *the* and *men* merged as before
- same steps for *which men*
- *like* merged with *which men*
- *like* merged with *the men*

Merge: Example 2

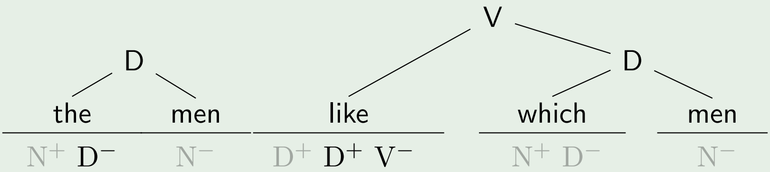
Assembling [$_{\text{VP}}$ the men like which men]



- *the* and *men* merged as before
- same steps for *which men*
- *like* merged with *which men*
- *like* merged with *the men*

Merge: Example 2

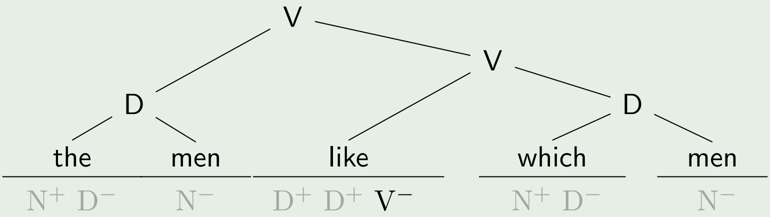
Assembling [$_{\text{VP}}$ the men like which men]



- *the* and *men* merged as before
- same steps for *which men*
- *like* merged with *which men*
- *like* merged with *the men*

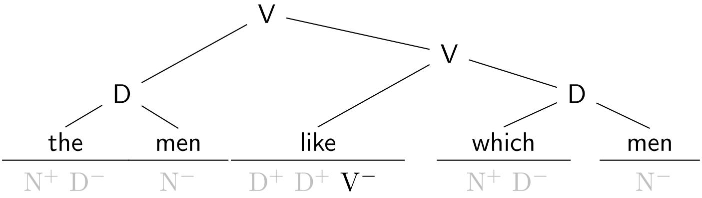
Merge: Example 2

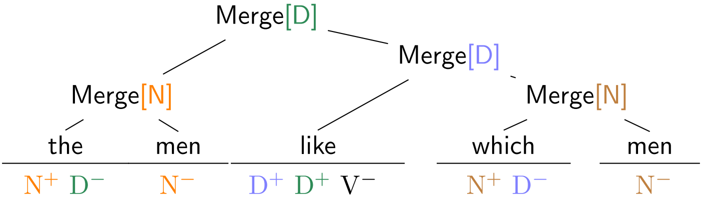
Assembling [$_{\text{VP}}$ the men like which men]



- *the* and *men* merged as before
- same steps for *which men*
- *like* merged with *which men*
- *like* merged with *the men*

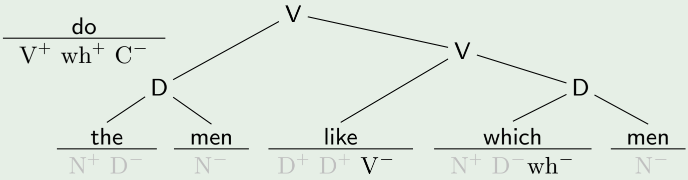
Merge: Example 2 [cont.]





Move

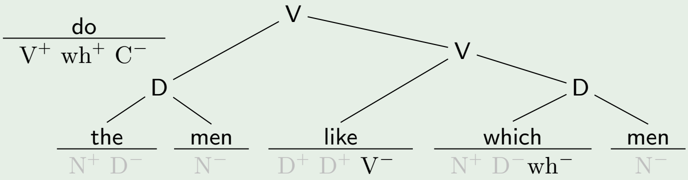
Assembling “which men do the men like?”



- Merge *do*
- Move triggered by features of opposite polarity

Move

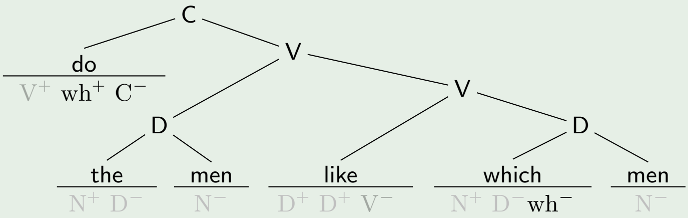
Assembling “which men do the men like?”



- Merge *do*
- Move triggered by features of opposite polarity

Move

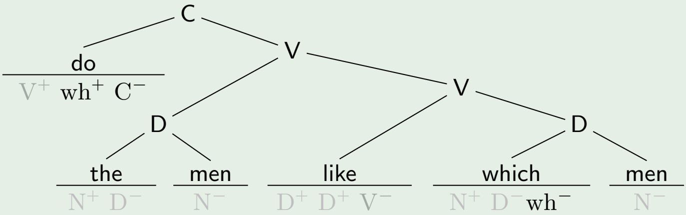
Assembling “which men do the men like?”



- Merge *do*
- Move triggered by features of opposite polarity

Move

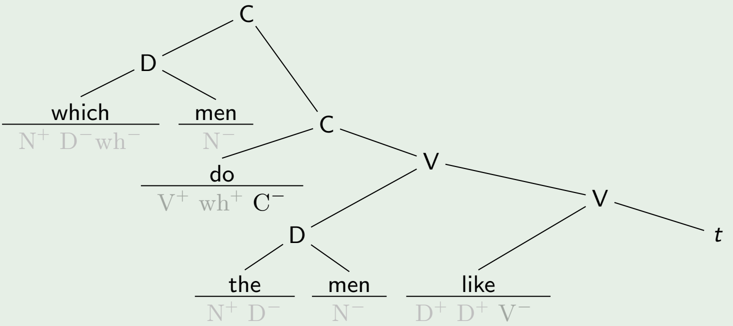
Assembling “which men do the men like?”



- Merge *do*
- Move triggered by features of opposite polarity

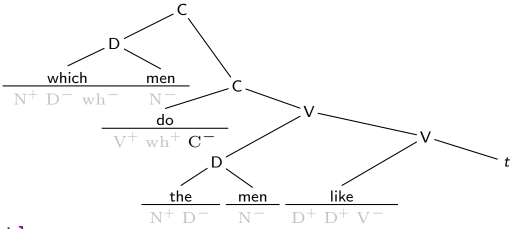
Move

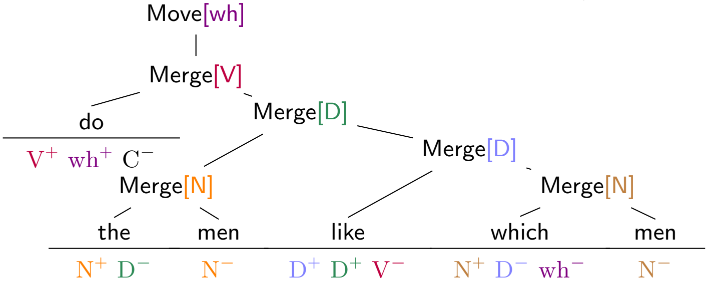
Assembling "which men do the men like?"



- Merge *do*
- Move triggered by features of opposite polarity

Derivation Trees with Move





What's the Point?

- Sentences aren't just strings, they contain hidden structure.
- Syntacticians usually look at the tree structure that is built by the operations Merge and Move.
- **But:** the history of how such a structure is built is also a tree ⇒ **phrase structure trees** and **derivation trees** as two possible views of tree-based syntax

Finite-State Tree Automata

A **finite-state tree automaton** (FSTA) assigns every node in a tree one of finitely many *states*, depending on

- the label of the node, and
- the states of the nodes immediately below it (if they exist).

The FSTA accepts the tree if the highest state is a *final state*.

Reminder: FSA Definition

A finite-state automaton (FSA) assigns every node in a **string** one of finitely many states, depending on

- the label of the node, and
- the state of the **preceding node** (if it exists).

The FSA accepts the string if the **last** state is a *final state*.

Finite-State Tree Automata

A **finite-state tree automaton** (FSTA) assigns every node in a tree one of finitely many *states*, depending on

- the label of the node, and
- the states of the nodes immediately below it (if they exist).

The FSTA accepts the tree if the highest state is a *final state*.

Reminder: FSA Definition

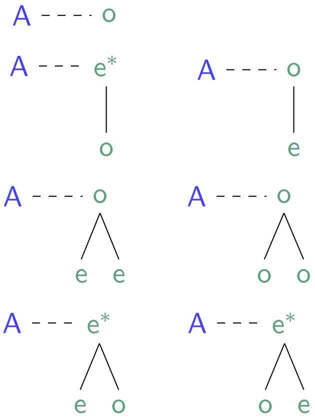
A finite-state automaton (FSA) assigns every node in a **string** one of finitely many states, depending on

- the label of the node, and
- the state of the **preceding node** (if it exists).

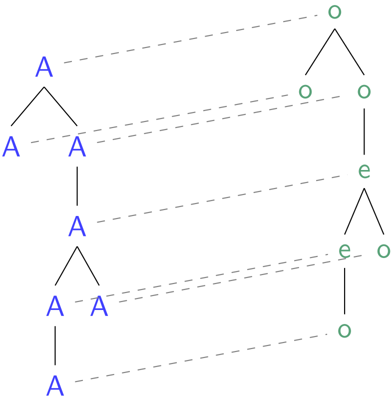
The FSA accepts the string if the **last** state is a *final state*.

Example of an FSTA

FSTA for binary trees over *a* with an even number of *a*s



Example State Assignment



Minimalism and FSTAs

- Phrase structure trees cannot be handled by FSTAs.
- But FSTAs are powerful enough for derivations trees.
  (Michaelis 2001; Kobele et al. 2007; Graf 2012)
- Since derivation trees are just a more abstract data structure for encoding syntactic dependencies, this means that **all syntactic dependencies can be computed with a finite amount of working memory**.

A New Perspective on Syntax and Phonology

Phonology	finite working memory computations over **strings**
Syntax	finite working memory computations over **trees**

Minimalism and FSTAs

- Phrase structure trees cannot be handled by FSTAs.
- But FSTAs are powerful enough for derivations trees. (Michaelis 2001; Kobele et al. 2007; Graf 2012)
- Since derivation trees are just a more abstract data structure for encoding syntactic dependencies, this means that **all syntactic dependencies can be computed with a finite amount of working memory**.

A New Perspective on Syntax and Phonology

Phonology	finite working memory computations over **strings**
Syntax	finite working memory computations over **trees**

Conclusion

- A computational perspective gives us a rough idea about memory usage.
- But it is important to look at the right data structure.
- Moving from strings to trees unearths a deep cognitive parallel between phonology and syntax, even though they involve very different dependencies.

References

Chomsky, Noam. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 2:113–124.

Chomsky, Noam. 1959. On certain formal properties of grammars. *Information and Control* 2:137–167.

Chomsky, Noam. 1995. *The minimalist program*. Cambridge, Mass.: MIT Press.

Graf, Thomas. 2010. *Logics of phonological reasoning*. Master's thesis, University of California, Los Angeles.

Graf, Thomas. 2012. Locality and the complexity of minimalist derivation tree languages. In *Formal Grammar 2010/2011*, ed. Philippe de Groot and Mark-Jan Nederhof, volume 7395 of *Lecture Notes in Computer Science*, 208–227. Heidelberg: Springer.

Kaplan, Ronald M., and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics* 20:331–378.

Kobele, Gregory M., Christian Retoré, and Sylvain Salvati. 2007. An automata-theoretic approach to minimalism. In *Model Theoretic Syntax at 10*, ed. James Rogers and Stephan Kepser, 71–80.

Michaelis, Jens. 2001. Transforming linear context-free rewriting systems into minimalist grammars. *Lecture Notes in Artificial Intelligence* 2099:228–244.

Shieber, Stuart M. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy* 8:333–345.

Stabler, Edward P. 1997. Derivational minimalism. In *Logical aspects of computational linguistics*, ed. Christian Retoré, volume 1328 of *Lecture Notes in Computer Science*, 68–95. Berlin: Springer.