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
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  \( nef+ið \rightarrow nevð \)  Icelandic
  * word-final devoicing  \( rad \rightarrow rat \)  German
  * intervocalic devoicing  \( aba \rightarrow apa \)  unattested
  * umlaut  \( mamm+u \rightarrow mömmu \)  Icelandic
  * dissimilation  \( lun+alis \rightarrow lunaris \)  Latin
  * anti-umlaut  \( mömm+u \rightarrow mammu \)  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.
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. Examples include flow chart, street network, Wikipedia, internet, video game AI
- **tree**: connected graph where every node is reachable from at most one node. Examples are family tree, hard drive layout, XML file
- **string**: sequence of nodes. Examples are telephone number, Python program, human genome, Shakespeare’s oeuvre
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)
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**.

<table>
<thead>
<tr>
<th>Example Language</th>
<th>Automaton Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>theorems of first-order logic</td>
</tr>
<tr>
<td>CS</td>
<td>all prime numbers</td>
</tr>
<tr>
<td>MCS</td>
<td>crossing dependencies</td>
</tr>
<tr>
<td>CF</td>
<td>center embedding</td>
</tr>
<tr>
<td>REG</td>
<td>all strings of even length</td>
</tr>
<tr>
<td>subREG</td>
<td>umlaut, voicing</td>
</tr>
<tr>
<td></td>
<td>Turing Machine</td>
</tr>
<tr>
<td></td>
<td>linear bounded automaton</td>
</tr>
<tr>
<td></td>
<td>embedded pushdown automaton</td>
</tr>
<tr>
<td></td>
<td>pushdown automaton</td>
</tr>
<tr>
<td></td>
<td>finite-state automaton</td>
</tr>
</tbody>
</table>
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 0 1 1 2* 2* 2* 2*
a a a a
0 0 1 1 2* 2* 2* 2*
b b b b
```

Example strings accepted by this FSA:

```
b a b a a a b
```
Example 1: Counting Symbols

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

\[
\begin{array}{c|c|c|c}
1 & 0 & 1 & 1 \\
\hline
a & a & a & a \\
0 & 0 & 0 & 0 \\
\hline
b & b & b & b \\
\end{array}
\]

$2^* 2^*$
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* 2*
a a a a
0 0 0 1 1 2* 2* 2*
b b b b
0 b a a a 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 a
0 b
```

```
0 1 a
0 0 b
1 1 b
```

```
1 2* a
2* 2* a
```

```
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
Example 1: Counting Symbols

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

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Symbol</th>
<th>State</th>
<th>Transition</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>a</td>
<td>1</td>
<td>2*</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>b</td>
<td>0</td>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>b</td>
<td>1</td>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>b</td>
<td>2</td>
<td>2*</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>b</td>
<td>2</td>
<td>2*</td>
<td>b</td>
</tr>
</tbody>
</table>

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

\[
\begin{array}{c|c|c|c|c}
1 & 0 & 1 & 1 & 2 \\
\text{a} & \text{a} & \text{a} & \text{a} & \text{a} \\
0 & 0 & 0 & 1 & 1 \\
\text{b} & \text{b} & \text{b} & \text{b} & \text{b} \\
\end{array}
\]
Example 2: Remembering Symbols

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

```
1* a
0* b
0* c
0* 0* b
0* 0* c
1* 1* a
2* 0*
1* 2* b
2* 2*
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 $c$

<table>
<thead>
<tr>
<th>1*</th>
<th>0* 1*</th>
<th>1* 1*</th>
<th>2* 0*</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$a$</td>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td>$b$</td>
<td>$b$</td>
<td>$b$</td>
<td>$b$</td>
</tr>
<tr>
<td>$c$</td>
<td>$c$</td>
<td>$c$</td>
<td>$c$</td>
</tr>
</tbody>
</table>

$b$ $c$ $a$ $b$ $b$ $a$ $c$
Example 2: Remembering Symbols

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

- $a$ can be followed by $a^*$ or $c^*$.
- $b$ can only be followed by $b^*$.
- $c$ can only be followed by $c^*$.

Some valid strings include:

- $a^*$
- $b^*$
- $c^*$
- $a^* a^*$
- $b^* b^*$
- $c^* c^*$
- $a^* b^* c^*$
- $b^* a^* b^* c^*$
- $c^* b^* a^* c^*$
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 $c$
Example 2: Remembering Symbols

Strings over \(a, b, c\) where no \(b\) occurs between \(a\) and \(c\)

\[
\begin{array}{c}
0^* 1^* \\
a \\
0^* b \\
0^* c \\
1^* 1^* \\
a \\
0^* 0^* b \\
0^* 0^* c \\
2^* 0^* \\
a \\
0^* 2^* b \\
0^* 2^* b \\
1^* 0^* c \\
1^* 1^* \\
a
\end{array}
\]
Example 2: Remembering Symbols

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

- $1^* 0^* 1^* 1^* 0^* 2^* 0^* a b c$
- $1^* 0^* 0^* 0^* 1^* 2^* 2^* b$
- $1^* 0^* 0^* 0^* 1^* c$
- $0^* 0^* 1^* 2^* 2^* a b b a c$
Example 2: Remembering Symbols

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

<table>
<thead>
<tr>
<th></th>
<th>1*</th>
<th>0* 1*</th>
<th>1* 1*</th>
<th>2* 0*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

| 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 $c$

$1^* a$
$0^* 1^* a$
$1^* 1^* a$
$2^* 0^* a$

$0^* b$
$0^* 0^* b$
$1^* 2^* b$
$2^* 2^* b$

$0^* c$
$0^* 0^* c$
$1^* 0^* c$

$0 0 1 2 2 0 0$
$b c a b b a c$
Example 3: More Sophisticated Counting

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

$$
\begin{array}{cccccc}
1 & 0 & 1 & 1 & 0 & 1 \\
a & a & a & 1 & 0 & 0 \\
0 & 0 & 0 & b & b & c \\
b & b & b & c & c & c \\
0 & 0 & 0 & 1 & 1 & 0 \\
c & c & c & 1 & 1 & 2^* \\
b & b & b & a & a & c \\
\end{array}
$$
Example 3: More Sophisticated Counting

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td></td>
<td>c</td>
</tr>
</tbody>
</table>

$2^*$
Example 3: More Sophisticated Counting

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2*</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>c</td>
</tr>
</tbody>
</table>
Example 3: More Sophisticated Counting

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

```
1 a
0 b
0 c
0 0 b c
0 0 c
0 0 0
0 0 0
0 0 0
```

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

0 1 0
a

0 0 0
b

0 0 0
b

1 1 1
b

1 1 0
b

1 0
a

0 1
a

0 0 1 1
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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
</tr>
</tbody>
</table>

0 2*
Example 3: More Sophisticated Counting

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0 1</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

Example string: `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

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

2*
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.

<table>
<thead>
<tr>
<th>Phonology</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lang. Class</td>
<td>Syntax</td>
</tr>
<tr>
<td>finite-state</td>
<td>regular</td>
</tr>
<tr>
<td>finite</td>
<td>≥ mildly context-sensitive</td>
</tr>
<tr>
<td>finite</td>
<td>embedded pushdown</td>
</tr>
<tr>
<td>finite</td>
<td>finite coupled with infinite</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Phonology</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lang. Class</strong></td>
<td>regular</td>
</tr>
<tr>
<td><strong>Automaton</strong></td>
<td>finite-state</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>finite</td>
</tr>
</tbody>
</table>
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.
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)
Minimalist grammars treat syntax like chemistry.

<table>
<thead>
<tr>
<th>Chemistry</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>atoms</td>
<td>words</td>
</tr>
<tr>
<td>electrons</td>
<td>features</td>
</tr>
<tr>
<td>molecules</td>
<td>sentences</td>
</tr>
<tr>
<td>stable</td>
<td>grammatical</td>
</tr>
<tr>
<td>unstable</td>
<td>ungrammatical</td>
</tr>
</tbody>
</table>

- 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.

<table>
<thead>
<tr>
<th>Chemistry</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>atoms</td>
<td>words</td>
</tr>
<tr>
<td>electrons</td>
<td>features</td>
</tr>
<tr>
<td>molecules</td>
<td>sentences</td>
</tr>
<tr>
<td>stable</td>
<td>grammatical</td>
</tr>
<tr>
<td>unstable</td>
<td>ungrammatical</td>
</tr>
</tbody>
</table>

- 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 \([_{DP} \text{ the men}]\)

\[
\begin{array}{c}
\text{the} \\
N^+ \ D^-
\end{array}
\quad
\begin{array}{c}
\text{men} \\
N^-
\end{array}
\]

- Features of opposite polarities annihilate
- Annihilation triggers Merge, which builds structure on top
Assembling $[\text{DP the men}]$

$\begin{array}{c}
\text{the} \\
N^+ D^-
\end{array}$

$\begin{array}{c}
\text{men} \\
N^-
\end{array}$

- Features of opposite polarities annihilate
- Annihilation triggers Merge, which builds structure on top
Assembling $[_{DP \text{ the men}}]$ 

\[
\begin{array}{c}
\text{the} \\
N^+ \quad D^- \\
\text{men} \\
N^- \\
\end{array}
\]

- 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 \([\text{DP the men}]\)

- Features of opposite polarities annihilate
- Annihilation triggers Merge, which builds structure on top
Assembling $[\text{VP the men like which men}]$

<table>
<thead>
<tr>
<th>the</th>
<th>men</th>
<th>like</th>
<th>which</th>
<th>men</th>
</tr>
</thead>
<tbody>
<tr>
<td>N⁺</td>
<td>D⁻</td>
<td>N⁻</td>
<td>D⁺</td>
<td>D⁺</td>
</tr>
<tr>
<td>N⁺</td>
<td>D⁻</td>
<td>N⁻</td>
<td>N⁻</td>
<td>N⁻</td>
</tr>
</tbody>
</table>

- *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 \([_{VP \ the \ men \ like \ which \ men}]\)

<table>
<thead>
<tr>
<th>the</th>
<th>men</th>
<th>like</th>
<th>which</th>
<th>men</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N^+)</td>
<td>(D^-)</td>
<td>(N^-)</td>
<td>(D^+)</td>
<td>(D^+)</td>
</tr>
</tbody>
</table>

- *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 \([VP \ the \ men \ like \ which \ men]\)

- 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 \([_{VP} \text{ the men like which men}]\)

- \(\text{the}\) and \(\text{men}\) merged as before
- same steps for \(\text{which men}\)
- \(\text{like}\) merged with \(\text{which men}\)
- \(\text{like}\) merged with \(\text{the men}\)
Assembling \( [VP \ the \ men \ like \ which \ men] \)

- \( \text{the} \) and \( \text{men} \) merged as before
- same steps for \( \text{which} \ \text{men} \)
- \( \text{like} \) merged with \( \text{which} \ \text{men} \)
- \( \text{like} \) merged with \( \text{the} \ \text{men} \)
Assembling \([VP \text{ the men like which men}]\)

- **the** and **men** merged as before
- same steps for **which men**
- **like** merged with **which men**
- **like** merged with **the men**
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**
Assembling \([v_{P\ \text{the men like which men}}]\)

- **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 \([_{VP} \text{ the men like which men}]\)

\[
\begin{array}{c}
\text{the} & \text{men} & \text{like} \\
N^+ & D^- & N^- & D^+ & D^+ & V^- \\
\text{which} & \text{men} \\
N^+ & D^- & N^- \\
\end{array}
\]

- *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.]

Diagram:

```
V
  /\  
D   V
  /\  |
the men like
N+ D- N- D+ D+ V-
```

```
V
  /\  
D   V
  /\  |
which men
N+ D- N-
```

```
Merge[D]
  /\
Merge[N]  Merge[D]
 /  
the men like
N+ D- N- D+ D+ V-
```

```
Merge[D]
  /\
Merge[N]  Merge[N]
 /  
which men
N+ D- N-
```
Move triggered by features of opposite polarity

Merge *do*
Assembling “which men do the men like?”

- **Merge** *do*
- Move triggered by features of opposite polarity
Assembling "which men do the men like?"

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

```
A ---- o

A ---- e*        A ---- o
    |              |    |
    o              e    e

A ---- o         A ---- o
    |     |     |
    e    o    e  o

A ---- e*        A ---- e*
    |    |    |    |    |
    e  o  e  o  e  o  e
```
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  |
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.