Finite State Morphology Assignment
Finite State Machine
Consume input; move to next state

Improve the machine?
• Eliminate one state (more elegant)
• Add error message if you push the button too soon
• What to do if someone keeps putting money in after the final state.
Finite State Transducer
Consume input; produce output: move to next state

Improve the machine?
• Eliminate one state (more elegant)
• Add error message if you push the button too soon
• What to do if someone keeps putting money in after the final state.
An English Word

• From David Crystal (Cambridge Encyclopedia of English)

• Grace (noun): graces
  – Graceful
    • Ungraceful
      – Ungracefully
      – Ungracefulness
    • Gracefully
    • Gracefulness
  – Grace (verb): graces, graced, gracing

• Disgrace (noun): disgraces
  – Disgraceful
    • Disgracefully
    • Dis gracefulness
  – Disgrace (verb): disgraces, disgraced, disgracing

• Graceless
  • Gracelessly
  • Gracelessness

• Gracious
  • Graciously
  • Graciousness
  • Ungracious
    – Ungraciously
    – Ungraciousness
1. Trace all the words based on “grace” to make sure that they all get through the machine.

2. What is wrong with the machine? (Overgenerate? Undergenerate? Redundant? Doesn’t capture structure and ambiguity?)
1. input: dis nstem Ø pst
2. output: disgraceed (we will fix it later)

dis nstem Ø pst    underlying form or    upper form
disgraceed         surface form         or     lower form
Underlying forms and surface forms

• Phonology:
  – Underlying: \[\text{ɪNgəmplɪt} \text{ (incomplete)}\]
  – Surface: \[\text{ɪŋkəmplɪt}\]

• Orthography
  – Underlying: happy+ness
  – Surface: happiness
A terrible twist of terminology in XFST

• The underlying form is called the upper form.
• The surface form is called the lower form.
• Think of it this way:
  – in phonology class, if you are going to derive a word using a sequence of rules, you write the underlying form at the top of the page/board and write the derivation under it.
Underlying and surface forms

• Underlying forms can be glosses of morphemes.

• Underlying: happy[AdjCmpr]
  • Surface: happier

• Underlying: happy[NDer][Npl]
  • Surface: happinesses
Where do those things in square brackets come from?

• You make them up.
• I decided to call –er [AdjCmpr] for “adjective, comparative”
• You could call it something else.
You have to make up labels for your morphemes
It will look like code instead of circles and arrows

<table>
<thead>
<tr>
<th>Multichar_Symbols</th>
<th>LEXICON NRoot</th>
<th>NSuffs-s</th>
</tr>
</thead>
<tbody>
<tr>
<td>[NSg] [NPl] [Adj] [AdjCmpr] [AdjSupr] [NDer]</td>
<td>book:book</td>
<td>NSuffs-s</td>
</tr>
<tr>
<td></td>
<td>pencil:pencil</td>
<td>NSuffs-s</td>
</tr>
<tr>
<td></td>
<td>child[NSg]:child</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>child[NPl]:children</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>rash:rash</td>
<td>NSuffs-es</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEXICON Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:0 NRoot ;</td>
</tr>
<tr>
<td>0:0 AdjRoot ;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEXICON AdjRoot</th>
</tr>
</thead>
<tbody>
<tr>
<td>happy[Adj]:happy # ;</td>
</tr>
<tr>
<td>happy:happy AdjSuffs ;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEXICON AdjSuffs</th>
</tr>
</thead>
<tbody>
<tr>
<td>[AdjCmpr]:er # ;</td>
</tr>
<tr>
<td>[AdjSupr]:est # ;</td>
</tr>
<tr>
<td>[NDer]:ness NSuffs-es ;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEXICON NSuffs-s</th>
</tr>
</thead>
<tbody>
<tr>
<td>[NSg]:0 # ;</td>
</tr>
<tr>
<td>[NPI]:s # ;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEXICON NSuffs-es</th>
</tr>
</thead>
<tbody>
<tr>
<td>[NSg]:0 # ;</td>
</tr>
<tr>
<td>[NPI]:es # ;</td>
</tr>
</tbody>
</table>
Another way

LEXICON NRoot
book:book NSuffs;
pencil:pencil NSuffs;
child[NSg]:child #;
child[NPl]:children #;
rash:rash NSuffs;

LEXICON NSuffs
[NSg]:0 # ;
[NPl]:s # ;

s -> e s || s h _ .#. ;
What will you do for this assignment?

• You will use a program called XFST (Xerox Finite State Transducer).
• If you give it an underlying form, it will give you the surface form.
• If you give it a surface form, it will give you the underlying form.

Xerox Finite-State Tool, version 2.10.23 (cfsm 2.12.18) (svn 25211)

Type "help" to list all commands available or "help help" for further help.

xfst[0]: read lexc lex-without-rules.t
xfst[0]: read lexc lex-without-rules.txt
Opening input file 'lex-without-rules.txt'
February 13, 2014 04:56:33 GMT
Reading UTF-8 text from 'lex-without-rules.txt'
Root...2, AdjRoot...2, AdjSuffs...3, NRoot...5, NSuffs-s...2, NSuffs-es...2
Building lexicon...Minimizing...Done!
3.4 Kb. 35 states, 44 arcs, 13 paths.
Closing 'lex-without-rules.txt'
xfst[1]: apply up children
cild[NP]
xfst[1]: apply up books
book[NP]
xfst[1]:

xfst[1]: apply down child[NP]
children
xfst[1]:

Opening input file 'three-eng-plural-script.xfst'
March 17, 2015 04:23:34 GMT
Opening file eng-sibilant-plural.regex...
Last line is not terminated.
2.4 Kb. 6 states, 18 arcs, Circular.
Closing file eng-sibilant-plural.regex...
Defined 'R': 2.4 Kb. 6 states, 18 arcs, Circular.
Opening input file 'three-eng-plurals-lexc.txt'
March 03, 2015 04:31:01 GMT
Reading UTF-8 text from 'three-eng-plurals-lexc.txt'
NRoot...5, NSuffs...2
Building lexicon...Minimizing...Done!
2.8 Kb. 21 states, 25 arcs, 8 paths.
Closing 'three-eng-plurals-lexc.txt'
Defined 'L': 2.8 Kb. 21 states, 25 arcs, 8 paths.
2.9 Kb. 23 states, 28 arcs, 8 paths.
Closing file three-eng-plural-script.xfst...
Copyright © Palo Alto Research Center 2001-2015
Xerox Finite-State Tool, version 2.10.23 (cfsm 2.12.18) (svn 25211)

Type "help" to list all commands available or "help help" for further help.

xfst[1]: up rashes
rash[NPI]
xfst[1]: down rash[NPI]
rashes
xfst[1]: up children
child[NPI]
xfst[1]: up pencil
pencil[NSg]
xfst[1]: down child[NSg]
child
xfst[1]: down child[NPI]
children
xfst[1]:
Break the job down into small pieces

- Start with your hundred sentences and hundred noun phrases
- Make a regression test suite (upper and lower forms)
  - Include ill-formed words to be sure they fail
- Give a square bracket name to each morpheme
- Install XFST and learn to run it.
- Write a file called a lexc file.
- Optional: write a rule file.
- Test:
  - If you input a word that is well-formed in your language, the output is the underlying form.
  - If the word is not ambiguous, you should get one underlying form.
  - If the word is ambiguous (e.g., [unlock]able, un[lockable]), you should get one underlying form for each meaning.
  - If you input an ill-formed word (e.g., happyness), there should be no underlying form and the FST will output three question marks (???).
Input/Output behavior

• Input: well-formed upper
  Output: corresponding lower form

• Input: lower (orthographic) form
  Output: upper (lemma plus names of morphological categories that are present)

• Input: illformed or not covered yet
  – *disful
  – *ungrace
  – cat

• Output: nothing
The regression test suite

- Order of morphemes (morphotactics)
  - Include infixing
- Morpho-phonological alternations
  - Insertions, deletions, assimilations
- The test suite shows the language your machine accepts (in computer science terms)
  - E.g., your machine accepts \((ab)^n\)
  - Test suite includes ab, abab, ababab, etc.
  - Test suite includes aa, aba, etc.
Review of Karttunen and Beesley tutorial on lexc and fst
Look for more examples at

The tutorial by Beesley and Karttunen is divided

- Monday morning
- Monday afternoon
- Tuesday morning
- etc.
- I referenced them as Xday { morning | afternoon }, slide N
Computational morphology

Karttunen, LSA 2005, day 1, slide 10

Analysis

leaf N Pl  leave N Pl  leave V Sg3

leaves

Generation

hang V Past

hanged  hung
Four notations

• lexc: lexicon compiler
  – declarative

• FST: finite state transducer
  – declarative

• XFST command line
  – Xerox finite state transducer
  – procedural: something happens every time you type a command

• Linux command line
  – procedural
Declarative vs procedural

• Your LEXC file is not a program.
• It is data.
• But you have to understand what the program will do with it.

• Monday morning, slide 13
Tuesday afternoon: Lexc (lexicon compiler)

• Make a lexc file
• at the xfst prompt (xfst command line)
• read lexc < filename
• apply up word
Format of the LEXC file
Multichar_Symbols declaration

Multichar_Symbols  [Noun] [Verb] [Adj] [Adv]

[Sg] [Pl]

[1P] [2P] [3P] ^FEAT1 ^FEAT2

The Multichar_Symbols statement is formally optional and is
placed at the top of your lexc source file. You can declare as many
multicharacter symbols as you find necessary or useful. The
compiler uses this declaration to separate the strings of your lexc
program into symbols. You are strongly encouraged to include a
non-alphabetic character in the spelling of each multicharacter
symbol to help them stand out visually.
Multichar_Symbols declaration

Multichar_Symbols

[Noun]
[Verb]
[Adj]
[Adv]

[Sg] [Pl] ! Number tags

[1P] [2P] [3P] ! Person tags

^FEAT1 ^FEAT2 ! “Feature” tags, used internally

The Multichar_Symbols declaration can extend over multiple lines and has no terminator.
The Body of your lexc Program

- The body of a lexc program is composed of LEXICONs.

- There should always be one LEXICON named Root. It corresponds to the Start State in the resulting Network.

  LEXICON Root
  
  dog       N ;
  cat       N ;
  bird      N ;

- If you don’t define a LEXICON Root, lexc will try to use the first LEXICON in the file as the Start State.

- Done confuse “LEXICON Root” with the roots of your natural language.
Entries in a LEXICON

- Each defined LEXICON must have at least one entry.

- An entry consists of two parts and is terminated with a semicolon

  data continuation-class ;

- The data part has to fit one of four formats:
  - string
  - upper:lower
  - < regular-expression >
  - empty

  e.g.
  - dog
  - swim:swam
  - < a b* c >
  - e.g.
The upper:lower entries are the simplest way to specify portions of the network where the upper-side and lower-side differ. They are especially useful for irregularities/suppletions.

Multichar_Symbols +Verb +Past +Noun +Sg +Pl

LEXICON Root

swim+Verb+Past:swam # ;
go+Verb+Past:went # ;
child+Noun+Pl:children # ;
ox+Noun+Pl:oxen # ;

Tuesday afternoon, slide 11
Continuation Classes

- The Continuation Class is just the name of a defined LEXICON or #, indicating end-of-word (a final state).

  Multichar_Symbols [Noun] [Sg] [Pl]

  LEXICON Root
  dog   N ;
  cat   N ;

  LEXICON N
  [Noun][Sg]:0 # ;
  [Noun][Pl]:s # ;

Slide 14, Tuesday afternoon
Formally Speaking . . .

- Lexc syntax is a kind of right-recursive context-free phrase-structure grammar.

- Context-free phrase-structure grammars can *in general* describe context-free languages (i.e. languages beyond finite-state power), including languages with balanced parentheses.

- But with the right-recursive limitation, a phrase-structure grammar can define only finite-state languages.

- Lexc can describe only finite-state languages.

- Lexc descriptions compile into finite-state networks.
Design and software engineering issues: keep it clean and modular

Thinking About lexic LEXICONS

- A LEXICON should hold a coherent class of morphemes

- The entries in a lexic LEXICON are unioned together by the compiler; the order of the entries in a LEXICON is *not* significant.

- Think of each LEXICONs as a potential “target”
  - Entries “point at” a LEXICON via the ContinuationClass
  - But each entry in a LEXICON could itself point to a different ContinuationClass

- During development, you may have to subdivide lexicons
  - Avoid having copies of the same material (if possible)
  - You may change an entry in one place and forget to change the copy
Some common morphotactic operations

• Skip a morpheme
• Loop
Optional Morphemes via By-Pass

LEXICON Root
  Vroot ;
LEXICON Vroot
  kant  V ;
  dir   V ;
  don   V ;
  pens  V ;

LEXICON V
  AdLex ;
  Vend ; ! bypass

LEXICON AdLex
  ad   Vend ;

LEXICON Vend
  as    # ;
  is    # ;
  os    # ;
  us    # ;
  u     # ;
  i     # ;
**Lexc Idiom: Optional Morphemes via “Escape” Entries**

<table>
<thead>
<tr>
<th>LEXICON Vroot</th>
<th>LEXICON Vend</th>
</tr>
</thead>
<tbody>
<tr>
<td>kant</td>
<td>as</td>
</tr>
<tr>
<td>dir</td>
<td>is</td>
</tr>
<tr>
<td>don</td>
<td>os</td>
</tr>
<tr>
<td>pens</td>
<td>us</td>
</tr>
<tr>
<td></td>
<td>u</td>
</tr>
<tr>
<td></td>
<td>i</td>
</tr>
</tbody>
</table>

LEXICON AdLex

<table>
<thead>
<tr>
<th>ad</th>
<th>Vend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vend</td>
<td>! escape</td>
</tr>
<tr>
<td>LEXICON</td>
<td>Nroot</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>kat</td>
<td>N</td>
</tr>
<tr>
<td>hund</td>
<td>N</td>
</tr>
<tr>
<td>elefant</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LEXICON</td>
<td>N</td>
</tr>
<tr>
<td>eg</td>
<td>N</td>
</tr>
<tr>
<td>et</td>
<td>N</td>
</tr>
<tr>
<td>in</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Nend</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LEXICON</td>
<td>Nend</td>
</tr>
<tr>
<td>o</td>
<td>Plur</td>
</tr>
</tbody>
</table>

Tuesday afternoon, slide 20
Stem compounding (loops) in lexc

---

**LEXICON** Nroot

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>kat</td>
<td>N ;</td>
</tr>
<tr>
<td>hund</td>
<td>N ;</td>
</tr>
<tr>
<td>elefant</td>
<td>N ;</td>
</tr>
</tbody>
</table>

---

**LEXICON** N

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nroot ;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nend ;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**LEXICON** Plur

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>Case ;</td>
<td>! Opt. plural ending</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Case ;</td>
</tr>
</tbody>
</table>

---

**LEXICON** Case

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td># ;</td>
</tr>
<tr>
<td></td>
<td># ;</td>
</tr>
</tbody>
</table>

---

**LEXICON** Nend

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>Plur ;</td>
</tr>
</tbody>
</table>

---

Tuesday afternoon, slide 21
**Special Characters in lexc**

Overall, there are far fewer special characters in lexc than in regular expressions. In lexc, the following are special:

<table>
<thead>
<tr>
<th>Special</th>
<th>Literalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>used in upper:lower notation</td>
</tr>
<tr>
<td>;</td>
<td>terminates an entry</td>
</tr>
<tr>
<td>&lt;</td>
<td>begins a regular expression</td>
</tr>
<tr>
<td>&gt;</td>
<td>ends a regular expression</td>
</tr>
<tr>
<td>0</td>
<td>denotes the empty string (epsilon)</td>
</tr>
<tr>
<td>!</td>
<td>introduces a comment line</td>
</tr>
<tr>
<td>#</td>
<td>continuation-class for end-of-word</td>
</tr>
<tr>
<td>%</td>
<td>literalizing prefix</td>
</tr>
</tbody>
</table>

Tuesday afternoon, slide 22
Monday afternoon, 39-42 replace rules
Adding morpho-phonological rules

Using lexc and xfst together

- Write a lexc source file (e.g. mysrc-lexc.txt) using xemacs or a similar text editor
- Write suitable alternation rules (e.g. a regex file mysrc-rul.regex)
- Then, from the xfst interface
  - xfst[]: read regex < mysrc-rul.regex
  - xfst[]: define Rules
  - xfst[]: read lexc < mysrc-lexc.txt
  - xfst[]: define Lexicon
  - xfst[]: read regex Lexicon .o. Rules ;
  - xfst[]: apply up testword
Rule Schema:  
upper -> lower  
upper -> lower || leftcontext _ rightcontext

E.g.

xfst> read regex s -> z || [a|e|i|o|u]_[a|e|i|o|u];

xfst> apply down casa

What is this rule intended to do? What comes out?
xfst[0]: read regex s -> z || [a|e|i|o|u] _ [a|e|i|o|u];
2.5 Kb. 4 states, 25 arcs, Circular.
xfst[1]: apply down casa;
caza
xfst[1]: apply up caza
caza
caza
caza
xfst[1]: \[\]
multiple results

\[ a \ b \ | \ b \ | \ b \ a \ | \ a \ b \ a \rightarrow x \]

applied in a downward direction to the string “aba”

\[
\begin{array}{cccc}
  \underline{a \ b \ a} & \underline{a \ b \ a} & \underline{a \ b \ a} & \underline{a \ b \ a} \\
  \underline{a \ x \ a} & \underline{a \ x} & \underline{x \ a} & \underline{x} \\
\end{array}
\]

Four “factorizations” of the input string.
@-\rightarrow  \text{left-to-right, longest match replacement}

\begin{align*}
\text{a b | b | b a | a b a @-\rightarrow x}
\end{align*}

applied to “aba”, effectively prefers the longest match, 
as if the matching were being done left-to-right

\[
\begin{array}{cccc}
\text{aba} & \text{aba} & \text{aba} & \text{aba} \\
\text{axa} & \text{ax} & \underline{xa} & \underline{x} \\
\end{array}
\]
xfst[0]: read regex a b | b a | a b a --> x;
2.3 Kb. 4 states, 14 arcs, Circular.
xfst[1]: apply down aba;
x
xfst[1]: apply up x;
ab
aba
b
ba
x
xfst[1]: apply up axa;
axa
aabaa
xfst[1]:  

conditional replacement

A -> B

Replacement

L _ R

Context

The relation that replaces A by B between L and R leaving everything else unchanged.

Sources of complexity:

- Replacements and contexts may overlap
- Alternative ways of interpreting “between left and right.”
both contexts on the input side, || operator

\[ A \rightarrow B \quad || \quad L \_ R \]

\[
\begin{array}{c}
abababa \\
\text{o.} \\
ab \rightarrow x \quad || \quad abba
\end{array}
\]

yields

\[
\begin{array}{c}
abababa \\
\text{o.} \\
abx\text{x}a
\end{array}
\]

In practice, this is the most-used type of Replace Rule

The string “ab” is replaced by the string “x” when it is preceded by “ab” and followed by “a”.

Wednesday morning, slide 31
xfst[0]: read regex a b -> x || a b _ a ;
2.5 Kb. 7 states, 23 arcs, Circular.
xfst[1]: apply down abababa ;
abxxa
xfst[1]: 

L on input side, R on the output side, // operator

A -> B // L _ R

\[ \begin{align*}
ababa \\
.o.
\end{align*} \]

yields

\[ \begin{align*}
ababa \\
abxaba
\end{align*} \]

// rules can be useful for handling vowel harmony

Wednesday morning, slide 32
xfst[0]: read regex a b -> x // a b _ a;
2.5 Kb. 7 states, 22 arcs, Circular.
xfst[1]: apply down abababa;
abxaba
Ordering of multiple morpho-phonological rules

**kaNpat example**

- Assume a language that joins morpheme *kaN* (with an underspecified nasal N) and morpheme *pat* into the underlying or morphophonemic form *kaNpat*. This language then has “alternation” rules that dictate that N, when followed by p, gets realized as m. And p, when preceded by m, gets realized as m. The derivation looks like

- **Underlying input:** kaNpat
- **Rule1:** N -> m || _ p
- **Output of Rule1:** kampat
- **Rule2:** p -> m || m _
- **Output of Rule2:** kammat

- The composition operation (.o.) reduces the derivational cascade of transducer networks into a single transducer network.
Sequential application in detail

Karttunen, LSA 2005, day 1, slide 24

N becomes m before p

p becomes m after m
Composition

Karttunen, LSA 2005, day 1, slide 25
xfst[0]: read regex N -> m || _ p
;
2.3 Kb. 3 states, 10 arcs, Circular.
xfst[1]: apply down kaNpat

xfst[0]: read regex p -> m || m _ ;
2.2 Kb. 2 states, 6 arcs, Circular.
xfst[1]: apply down kaNpat
kaNpat
xfst[1]: apply down kampat
kammat
xfst[1]:
Your first cascade of rules

xfst> define Rule1  N -> m || _ p ;

xfst> define Rule2  p -> m || m _ ;

xfst> read regex Rule1 .o. Rule2 ;

xfst> apply down kaNpat

- What is the output?

- Now restart (with ‘clear stack’), define the two Rules as shown above, push them on the stack in the right order, and perform the composition on the stack using ‘compose net’. What is your result? (Remember that the networks must be pushed in the right order.)
Rule Abbreviations

- Multiple left-hand sides, separated by commas:
  
  \[ b \rightarrow p, \ d \rightarrow t, \ g \rightarrow k \mid \_ \mid .\#. \]

- Multiple right-hand sides, separated by commas:
  
  \[ e \rightarrow i \mid \_ (s) \mid .\#. \mid .\#. \mid p \mid r \]

- Use .\#. to refer to either the very beginning or the very end of a word.
Palatalization and Vowel Raising

Karttunen, LSA 2005, day 2, slide 22

- Palatalization
  - tim --> cim

- Vowel Raising
  - memi --> mimi

- Interaction
  - temi --> cimi
  - tememem --> cimimim
Vowel Raising & Palatalization

define C [ b | c | d | f | g | h | j | k | l |
m | n | p | q | r | s | t | v |
x | y | z ];

define Raising e -> i \_ C* i ;
define Palatalization t -> c \_ i;

regex Raising .o. Palatalization;

down memi
mimi
down tim
cim
down temi
cimi
down tememi
cimimi
tememi
timimi
cimimi
Two languages with vowel shortening

Left context on the input side  \( \mathbf{V}: \rightarrow \mathbf{V} \ | \ | \mathbf{V}: C^* \ _\)

Slovak
\[ \text{vol} + [a:] \mathbf{v} + [a:\_m\_e:] \]
\[ \text{vol} + [a:] \mathbf{v} + [a\_m\_e] \]

we call often

Left context on the output side  \( \mathbf{V}: \rightarrow \mathbf{V} \ // \mathbf{V}: C^* \ _\)

Gidabal
\[ \text{gunuU:} \mathbf{m} + [a:] \mathbf{d a: ng} + [b\_e:] \]
\[ \text{gunuU:} \mathbf{m} + [a\_d a: ng} + [b\_e] \]

is certainly right on the stump
define V [ a | e | i | o | u | a ];
define C [ b | c | d | f | g | h | j | k | l |
    m | n | p | q | r | s | t | v |
    x | y | z ];

define SlovakShortening %: -> 0 || V %: C* V _ ;

define GidabalShortening %: -> 0 // V %: C* V _ ;

push SlovakShortening
down vola:va:me:
vola:vame

push GidabalShortening
down gunu:mba:da:ngbe:
gunu:mbada:ngbe
Stop

• Extra slides follow
xfst[0]: read regex a b | b | b a | a b a \rightarrow x ;
768 bytes. 3 states, 12 arcs, Circular.
xfst[1]: apply down aba ;
   x
   xa
   ax
   axa
xfst[1]: apply up x ;
   ab
   aba
   b
   ba
   x
xfst[1]: apply up axa ;
   aabaa
   aaba
   abaa
   aba
   axa
xfst[1]: 

@-> left-to-right, longest match replacement

```
ab | b | ba | aba @-> x
```

applied to “aba”, effectively *prefers* the longest match, 
as if the matching were being done left-to-right
Making a lexical transducer

Karttunen, LSA 2005, day 2, slide 24

Morphotactics

Lexicon
Regular Expression

Rules
Regular Expressions

Alternations

Compiler

composition

Lexicon FST

Rule FSTs

Lexical Transducer
(a single FST)
The language that the machine accepts

• Our pop machine accepts a language consisting of the strings:
  – nickle nickle button-press
  – dime button-press

• Our “grace” machine accepts the language consisting of the strings:
  – grace, graces, graced, gracing, graceful, gracefully, gracefulness, etc.

• Monday morning, slide 23
Monday morning, slide 24

"Apply"

Either accepts or rejects
Monday morning, slide 25

"Apply Down"  mesa+Noun+Fem+Pl  A Transducer

"Apply Up"