Review of Karttunen and Beesley tutorial on lexc and fst

Feb 26, 2015
The tutorial by Beesley and Karttunen is divided

• Monday morning
• Monday afternoon
• Tuesday morning
• etc.
• I referenced them as Xday { morning | afternoon }, slide N
Finite State Machine
Consume input; move to next state

Start

- nickel
  - nickel
    - Push button
      - Got selection
      - got 10 cents
    - dime
      - nickel
        - Got 5 cents
      - nickel
        - Start

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

Start

Got 5 cents

Got 10 cents

End

nickle: cha-ching

dime: cha-ching

Push-button: dispense

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
    - Disgracefulness
  - 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?)
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: ill-formed or not covered yet
  • *disful
  • *ungrace
  • cat
  • Output: nothing
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
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.
• If you give it an illformed surface form, it will fail.

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
It will look like code instead of circles and arrows

LEXICON Root
0:0  NRoot ;
0:0  AdjRoot ;

LEXICON AdjRoot
happy[Adj]:happy  # ;
happy:happi     AdjSuffs ;

LEXICON AdjSuffs
[AdjCmpr]:er     # ;
[AdjSupr]:est     # ;
[NDer]:ness     NSuffs-es ;

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

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

LEXICON NSuffs-es
[NSg]:0  # ;
[NPl]:es  # ;
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
Tuesday afternoon: Lexc (lexicon compiler)

• Make a lexc file
• at the xfst prompt (xfst command line)
• read lexc < filename
• apply up word
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
  
  \textit{data} \hspace{1cm} \textit{continuation-class} ;

- The data part has to fit one of four formats:
  - string
  - upper:lower
  - \textless{} \textit{regular-expression} \textgreater{}
  - empty

  e.g. dog
  e.g. swim:swam
  e.g. \textless{} a b* c \textgreater{}
  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 # ;
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 # ;
Thinking About lexc LEXICONS

- A LEXICON should hold a coherent class of morphemes

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

- Think of each LEXICON 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

---

Tuesday afternoon, slide 15
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.
Optional Morphemes via By-Pass

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

LEXICON V
  AdLex ;
  Vend ; ! bypass

LEXICON Vend
  as    # ;
  is    # ;
  os    # ;
  us    # ;
  u     # ;
  i     # ;

Tuesday afternoon, slide 18
Lexc Idiom: Optional Morphemes via “Escape” Entries

LEXICON Vroot
kant AdLex ;
dir AdLex ;
don AdLex ;
pens AdLex ;

LEXICON AdLex
ad Vend ;
Vend ; ! escape

LEXICON Vend
as # ;
is # ;
os # ;
us # ;
u # ;
i # ;

Tuesday afternoon, slide 19
## Lexc Idiom: Loops

<table>
<thead>
<tr>
<th>LEXICON</th>
<th>Nroot</th>
<th>LEXICON</th>
<th>Plur</th>
</tr>
</thead>
<tbody>
<tr>
<td>kat</td>
<td>N ;</td>
<td>j</td>
<td>Case ; ! Opt. plural ending</td>
</tr>
<tr>
<td>hund</td>
<td>N ;</td>
<td></td>
<td>Case ;</td>
</tr>
<tr>
<td>elefant</td>
<td>N ;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEXICON</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>eg</td>
<td>N ; ! loop</td>
</tr>
<tr>
<td>et</td>
<td>N ; ! loop</td>
</tr>
<tr>
<td>in</td>
<td>N ; ! loop</td>
</tr>
<tr>
<td>Nend</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEXICON</th>
<th>Nend</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>Plur ;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEXICON</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td># ; ! Opt. case ending</td>
</tr>
</tbody>
</table>

---

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

LEXICON Nroot
kat N ;
hund N ;
elefant N ;

LEXICON N
Nroot ;
Nend ;

LEXICON Nend
o Plur ;

LEXICON Plur
j Case ; ! Opt. plural ending
            Case ;

LEXICON Case
n # ;
            # ;
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>
Monday afternoon, 39-42 replace rules
FST notation
Another way

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

LEXICON NSuffs
[NSg]:0 # ;
[NPI]:s # ;
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
xfst[1]:
multiple results

\[ ab \mid b \mid ba \mid aba \rightarrow x \]

applied in a downward direction to the string “aba”

\[
\begin{align*}
abba & \quad abba & \quad abba & \quad abba \\
axa & \quad ax & \quad xa & \quad x
\end{align*}
\]

Four “factorizations” of the input string.
@→ left-to-right, longest match replacement

\[ a \ b \ | \ b \ | \ b \ a \ | \ a \ b \ a \ @→ \ x \]

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

\[ \require{cancel}
\begin{array}{cccc}
\cancel{a \ b \ a} & \cancel{a \ b \ a} & \cancel{a \ b \ a} & a \ b \ a \\
\cancel{a \ x \ a} & \cancel{a \ x} & \cancel{x \ a} & \underline{x}
\end{array}
\]
xfst[0]: read regex a b | 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**

\[ A \rightarrow B \quad \text{Replacement} \quad L \_ R \quad \text{Context} \]

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 -> B || L _ R

a b a b a b a  
. o.  

 yields  

a b a b a b a

a b x x a

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 \rightarrow B // L _ R

\[ \begin{array}{c}
  a \\ b \\ a \\ b \\ a \\ b \\ a \\
  \cdot o \end{array} \quad \text{yields} \quad \begin{array}{c}
  a \\ b \\ a \\ b \\ a \\ b \\ a \\
  a \\ b \\ x \\ a \\ b \\ a
\end{array} \]

// rules can be useful for handling vowel harmony
xfst[0]: read regex a b -> x // a b _ a ;
2.5 Kb. 7 states, 22 arcs, Circular.
xfst[1]: apply down abababa ;
abxaba
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.
xfst[0]: read regex N -> m || _ p
;
2.3 Kb. 3 states, 10 arcs, Circular.
xfst[1]: apply down kaNpat
kampat

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:
  \[ b \rightarrow p, \quad d \rightarrow t, \quad g \rightarrow k \quad \| \quad _\_ \quad .\# . \]

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

- Use .#. to refer to either the very beginning or the very end of a word.
Two languages with vowel shortening

Left context on the input side

\[ V: \rightarrow V \quad || \quad V: C^* \_ \]

Slovak

\[ \text{vol} + \underline{a:} \; v + \underline{a: m} \; e: \]

we call often

Left context on the output side

\[ V: \rightarrow V \quad \// \quad V: C^* \_ \]

Gidabal

\[ \text{gunu:} \; m + b \underline{a:} + d \underline{a:} \; ng + b \underline{e:} + \]

is certainly right on the stump
xfst[0]: read regex  a b | b | b a | a b a  ->  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
aba axa
xfst[1]:

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

\[ a \ b \ | \ b \ | \ b \ a \ | \ a \ b \ a \ @-> \ x \]

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

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