Computational Morphology: Xerox finite state tool

Yulia Zinova

15 February 2016 – 19 February 2016

Overview

XFST and foma

- XFST (Xerox finite state tool) is a commercial tool, the main book (includes a CD with software) is Karttunen (2003)
- foma is the open-source analog (Hulden, 2009)

Languages

- FST converts surface string language into analysis string language (both directions).
- The surface language is given.
- The analysis language has to be designed by the linguist.
- Xerox convention: each analysis string consists of the traditional dictionary base form followed by tags
  cantar+Verb+PInd+2P+Pl
  alto+Adj+Fem+Sg

Challenges

- Morphotactics:
  Words are composed of smaller elements that must be combined in a certain order:
  piti-less-ness is English
  piti-ness-less is not English
- Phonological alternations
  The shape of an element may vary depending on the context
  pity is realized as *piti* in *pitilessness*
  die becomes *dy* in *dying*

Regular relations

- The relation between the surface forms of a language and the corresponding lexical forms can be described as a regular relation.
- A regular relation consists of ordered pairs of strings.
  leaf+N+Pl : leaves
  hang+V+Past : hung
- Any finite collection of such pairs is a regular relation.
- Regular relations are closed under operations such as concatenation, iteration, union, and composition.
- Complex regular relations can be derived from simple relations.

Let's start

- Go to `http://www.fsmbook.com`, accept the agreement, download software.
- Run xfst .
- The xfst[0]: prompt indicates that the xfst application is waiting for a command. The number 0 indicates that the network stack is empty.
- 2 types of XFST commands:
    1. adding networks to the stack, replacing some or all of the stack by the result of some operation, and saving the stack into a file;
    2. working with the network that was most recently added to the stack.

Making and saving a network (1)

- To load a network you should:
  - load a previously compiled network from a binary file or
  - compiling a new network from some text source.
- In either case, the network becomes the topmost one on the stack.

Making and saving a network (2)

- In this example, we compile a network from a regular expression using the command 'read regex.' We type
  xfst[0] : *read regex* [%0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9];
- What does this regular expression denote?

Making and saving a network (2)

- In this example, we compile a network from a regular expression using the command 'read regex.' We type
  xfst[0] : *read regex* [%0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9];
- What does this regular expression denote?
- This expression denotes the language that consists of the ten decimal digits.
- Because 0 is a special symbol (epsilon) in a regular expression, it is necessary to prefix it here with %, the escape character, to have it interpreted as a digit.
- The semicolon at the end of the line closes the regular expression.

Making and saving a network (3)

- When the command is terminated with a carriage return, XFST responds...

Making and saving a network (3)

- When the command is terminated with a carriage return, XFST responds...
  2 states, 10 arcs, 10 words.
  xfst[1]:
  showing that the network representing this ten-word language consists of 2 states and 10 arcs.
- The new prompt, xfst[1]: shows that we now have one network on the stack.
- The command 'print net' displays the structure of the network on the screen.

Making and saving a network (4)

- The 'print net' command displays the states of the network: *s*0 (a non-final state), *fs*1 (a final state)
- and the labeled arcs leading from s0 to fs1.
- In addition, we see the symbol alphabet of the network (Sigma), the regular expression it was compiled from, and some characteristics of the network (Flags, Arity).
- It is often convenient to give a network a name that can be used in a regular expression to refer to it.
- The command for that assignment is 'define':
  xfst[1]: define Digit
  xfst[0]:
- The 'define' command requires at least one argument: the symbol that is being defined, here 'Digit'.
- If no further specification is given, the network on the top of the

Making and saving a network (5)

- The 'define' command can take the second argument: a regular expression that denotes the desired language or relation.
- Try
  xfst[0]: define Digit [%0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9] ;
- What is the state of the stack after the comand?

Making and saving a network (5)

- The 'define' command can take the second argument: a regular expression that denotes the desired language or relation.
- Try
  xfst[0]: define Digit [%0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9] ;
- What is the state of the stack after the comand?
- The stack remains empty.
- Note the closing semicolon that marks the end of the regular expression.
- Once defined, the name 'Digit' can be used in regular expressions to represent the language in question.

Making and saving a network (6)

- Let us construct a transducer that converts US numerals to the European format.
- In US numerals the comma is used as a separator, the period marks the beginning of the decimal part.
- In Europe the convention is the opposite.
- Thus "1,000.00" in the US corresponds to "1.000,00" in Europe.
- How should such transducer be defined?

Making and saving a network (7)

- A transducer that does this conversion can be defined as follows, using the defined 'Digit' symbol:
  xfst[0]: read regex %. -> %, , %, -> %. || Digit _ Digit ;
- How many arcs does the automaton have?

Making and saving a network (7)

- A transducer that does this conversion can be defined as follows, using the defined 'Digit' symbol:
  xfst[0]: read regex %. -> %, , %, -> %. || Digit _ Digit ;
- How many arcs does the automaton have?
- 41
- This transducer represents the parallel replacement of "." by "," and "," by "." between two digits.

Testing the network

- To verify that the transducer does what it is supposed to do we can use the 'apply' command.
- Because transducers are bidirectional, we must specify the direction of application.
- In this case, it is 'down'; that is, the US representation is on the "upper" side of the transducer:
  apply down 1,234.99
- The 'apply' command may also be used to take the input strings from a file instead of typing them directly.

Testing the network

- Create a file US-num.txt with several lines with numbers (terminate the last line!)
- Try
  apply down $<$ **US-num.txt**
- How is the file processed?
- What will happen if you add .5 to the list of numbers? And 10,00,00? 5,0,0?

Saving the network

- In order to have the transducer available in the future, we can save it to a file.
- The command 'save' writes all the networks currently on the stack into a single file.
- In this case, the file will contain just one network:
  save stack US-to-EU-num.fst

Plan

- Load the network we just created from a file to the stack.
- Add another network on the top of the first one.
- Perform an operation to replace both of them with the result of that operation.

Loading and using a stored network (1)

- Clear the stack:
  clear stack
- Load the network back from the file:
  load US-to-EU-num.fst
- Create another network by compiling a simple network from the same little text file we already used above: **read text** < **US-num.txt**
- The 'read text' command expects as its argument a name of a file containing a list of words, one entry per line. It compiles the word list into a network.

Loading and using a stored network (2)

- The command 'print words' displays the content of the compiled word list:
  print words
- How many networks are there in the stack at the moment?
- Try the **print stack** command.
- Note: unary commands such as **print net** and **print words** apply to the top network on the stack.
- Try the *print net* command. How do you interpret the result? Draw it!

Composing a network

- The **compose net** operation replaces the two networks on the stack by the result of the operation. Do it!
- Thus we now have just one network left.
- View its contents using the same **print words** command as before.
- How do you interpret what you see?
- Draw the resulting transducer.

Resulting transducer

- The result of the composition is a transducer.
- It denotes a relation, a mapping from one regular language into another one.
- On its "upper side", the transducer has the three original US-style numbers, each mapped to a corresponding European-style on the "lower side" of the transducer.
- For the most part, the mapping is an identity relation because each digit is mapped to itself.
- The only difference is that periods are mapped to commas, and vice versa.

Inspecting the transducer

- We can view the upper and lower languages of the relation independently. **print upper-words** displays the three US numbers; print lower-words shows what they have been transduced into.
- The **apply** command maps strings on one side of the transducer to the corresponding strings on the other side. Try **apply up 0,5**. Try also **apply up 0.5**.
- We can also extract one of the languages from the relation. The command **lower-side net** extracts from the transducer a simple automaton that contains just the three European numbers.

- It is more convenient, for many purposes, to write a list of commands to be run in batch mode without any user interaction.
- Let us write a script that compiles the US-to-European transducer and uses it to produce a file of European-style numbers from a file of US-style numbers.
- A script is an ordinary text file that can be prepared with any text editor, such as Emacs (see xfst.script).
- To run a script, tell xfst **source xfst.script**

Defining aliases (1)

- XFST allows the user to create simple names for more complex commands.
- For example,
  alias dir system ls -l *.txt
  creates a new XFST command 'dir' that has the same effect as 'system ls -l *.txt'
- The chosen alias must be a single word with no hyphens, underscores, or other special characters.
- The command **print alias** lists all the current aliases and their definitions.

Defining aliases (2)

- An alias can represent an arbitrary sequence of commands. To create such an alias, the user first types only the name to be defined.
  alias ConvertAndShow
- XFST responds by prompting the user for commands.
- The list can be terminated by a special symbol, **END;**, with no extra whitespace around it (alias.txt)
- now try **ConvertAndShow**

Command Syntax

- XFST commands are in general of the form '<command> <type or object>'
- <command> specifies the operation to be performed
- the second term, if any, gives some additional specification about the type of the operation or the object it applies to.
- For example, there are several variants of the 'print' command: 'print net', 'print sigma', 'print words', etc.
- All display commands and all unary operations, such as 'lower-side net', apply to the network on the top of the stack.
- Some commands, such as 'print net' and 'print words', can be followed by a name of network which has been given a name with the 'define' command

Short names

- Virtually all XFST commands can be abbreviated to a single word command.
- For example, the 'print' part of all print commands can be dropped.
- Thus 'sigma' as a command has the same effect as 'print sigma'.
- Similarly, 'regex' and 'read regex' are equivalent.
- Short command names are convenient when one is working in an interactive mode.
- It is better to use the long commands for scripts for readability.

Command Classes

- The FST commands can be grouped into five classes:
  1. Input/Output and Stack Commands
  2. Display commands
  3. Tests of network properties
  4. Operations on networks
  5. System commands
- The list of commands: commands.txt

Exercise

- Exercise on the Brazilian Portuguese Pronunciation (portuguese exercise.pdf)

References:

Hulden, M. (2009). Foma: a finite-state compiler and library. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, pages 29–32. Association for Computational Linguistics.

Karttunen, L. (2003). Finite-state morphology.