

Lecture 15. Finite State Transducers and OT Issues

1. Finite state transducers.....	1
2. An Illustration	3
3. Ways of looking at a finite state transducers.....	4
4. Cascading finite state transducers	4
References	5

Reading for today and for Thursday's guest lecture by Rajesh Bhatt:

(Frank and Satta 1998) (Full references at end of handout)

<http://www.cog.jhu.edu/faculty/frank/papers/ot-revised.pdf>

(Karttunen 1998)

<http://www2.parc.com/istl/members/karttune/publications/pto/bilkent.pdf>

(Karttunen 2001)

<http://www2.parc.com/istl/members/karttune/publications/ciaa-2000/fst-in-nlp.pdf>

1. Finite state transducers.

What are finite state transducers and why are they interesting? First what they are.

Start from finite-state automata, and recall that they can be used either as recognizers or generators. It's simply a matter of how we interpret what we do with the symbols on the transition arrows. If we "read" the symbols from an input tape, then our fsa is an "acceptor".

Alternatively, if we run the machine non-deterministically, following any legal path from state to state from an initial state to a final state, and "output" the symbols that are on the transition arrows as we go, then our fsa is a "generator". The same machine can be used either way, and defines the same language in either case.

From these observations it's a small step to a finite-state transducer: we put PAIRS of symbols (or symbol strings) on our transition arrows, viewing one as input and the other as output. One can in principle allow triples and more, but we'll stick to ones with pairs, and call the members of the pairs inputs and outputs. Which is which is up to us, and in principle that's a free choice analogous to the choice of viewing a straight fsa as an acceptor or a generator.

We'll give the exact definition shortly.

Why are finite state transducers of interest? Because even though we know that English as a whole isn't finite-state (unless we put an upper bound on self-embedding), many subparts of English grammar, and possibly all of phonology and much or all of morphology, can be well described by finite-state means, especially with the help of finite-state transducers, and finite state machines are computationally very efficient. The fact that finite-state automata have such nice closure properties (Lectures 11-12) also makes fsa's much easier to work with than other more powerful formalisms.

Applications: Johnson (Johnson 1972) showed very early on that one could model some aspects of phonology with finite-state means, and Kaplan and Kay (1981) showed a technique for compiling multiple finite-state transducers into a single finite-state transducers for implementing Chomsky-Halle style rewrite rules, a technique explained in detail in (Kaplan and Kay 1994). In morphology, Koskenniemi (Koskenniemi 1983, Koskenniemi 1985, Koskenniemi 1986) used a similar technique with finite state transducers in his "two-level morphology".

In syntax, there are subsets of a natural language (e.g. English) which are finite-state, of course – there’s a nice non-trivial example of the language of ‘date expressions’ in (Karttunen 2001), for instance. Is it worthwhile using finite-state techniques where possible even when we know the whole language is not finite-state? It occurs to me that the recent tendency to answer ‘yes’ to such questions may not be of interest not only to computational linguists, where it’s certainly understandable, but perhaps also for all of us interested in realistic models of human language. It may be that syntax has also been implicitly operating on the principle of “Generalize to the Worst Case”, much as Montague did in semantics. And the advocacy of a mixture of finite-state grammars and richer grammars may be loosely analogous to the advocacy of using the simplest semantic types possible and invoking higher types only where needed.

In any case, there is a current explosion of interest in finite-state methods, particularly in finite-state transducers, in both theoretical and computational linguistics, in phonology, morphology, and subparts of syntax.

Definitions.

See: **2.2 Finite State Transducers** (Blackburn and Striegnitz) at <http://www.coli.uni-sb.de/~kris/nlp-with-prolog/html/node13.html>

Also: [Finite State Transducers](http://jugernaut.eti.pg.gda.pl/~jandac/thesis/node13.html) (Jan Daciuk) at <http://jugernaut.eti.pg.gda.pl/~jandac/thesis/node13.html>

Also: [Finite State Transducers](http://www.spectrum.uni-bielefeld.de/Classes/Winter97/IntroCompPhon/compphon/node73.html) (Dafydd Gibbon) at <http://www.spectrum.uni-bielefeld.de/Classes/Winter97/IntroCompPhon/compphon/node73.html>

Good illustrations without a definition can be found in the Karttunen (2001) article. The differences among the different definitions are quite minor, mainly affecting details of what counts as deterministic vs. non-deterministic, treatment of the empty symbol on transitions, etc.

I’ll follow Gibbon: it’s very explicit, and I think it agrees on all important points with the others.

A finite state transducer is a finite state automaton in which the members of Σ (the symbols labeling the arcs) are pairs, triples, etc., rather than simple symbols. Traditionally the members of Σ in a transducer are just pairs, of which the left-hand member is the ‘input symbol’ and the right-hand member is the ‘output symbol’.

A deterministic finite state transducer (DFST) is defined as a septuple $\langle q_0, Q, F, \Sigma, \delta, \lambda, \Lambda \rangle$, where:

$q_0 \in Q$ -- the initial state

Q -- a finite set of states

$F \subseteq Q$ -- the final states, a subset of Q .

Σ -- a finite set of input symbols (the input alphabet)

Λ -- a finite set of output symbols (the output alphabet)

δ is a function in $Q \times \Sigma \rightarrow Q$ -- δ is the set of *transitions*, exactly as for a deterministic fsa, mapping a pair of a state and an input symbol to a state.

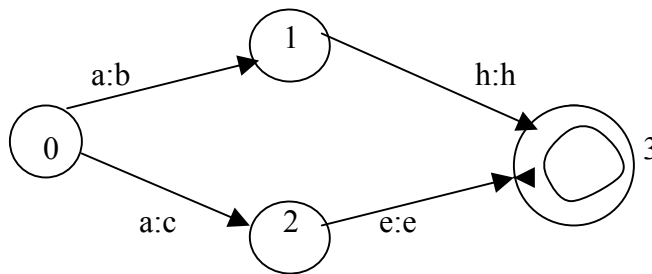
λ is a function in $Q \times \Sigma \rightarrow \Lambda$ – the *transduction function* (*emission function*), mapping a state and an input symbol to an output symbol.

Frequently the transition function and the transduction function are combined into a single *transition-transduction function*, which may also be called δ , in $Q \times \Sigma \rightarrow Q \times \Lambda$, mapping a pair of a state and an input symbol onto a pair of a state and an output symbol.

A finite-state transducer is *non-deterministic* if either the transition mapping or the transduction mapping fails to be a function, i.e. if there is more than one possible transition or more than one possible output symbol for a given pair of a state and an input symbol.

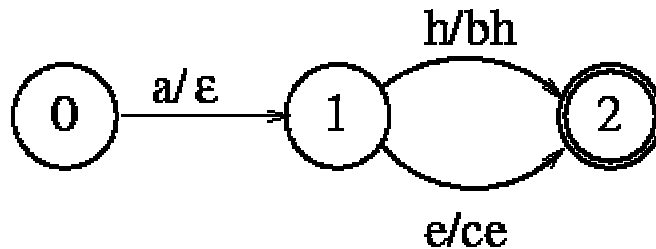
2. An Illustration

A very simple deterministic finite-state transducer.



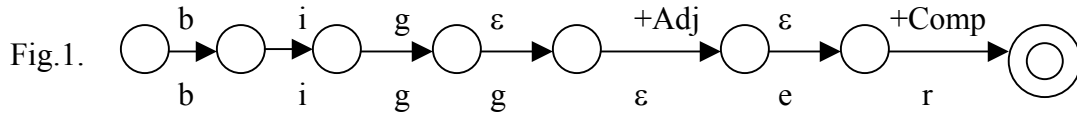
This very primitive transducer accepts just two input strings, *ah* and *ae*, and maps them (transduces them, translates them) onto *bh* and *ce* respectively. (One can imagine it as representing the pair of rules “*a* becomes *b* before *h* and becomes *c* before *e*.”)

This transducer is not deterministic, because there are two paths out of state 0 with input *a*. Allowing strings of symbols in either input or output (with no loss of generality, I’m pretty sure), and using ϵ as the empty string, we can replace the transducer above with the following equivalent deterministic finite state transducer.



Sometimes a deterministic finite state transducer will need many more states than an equivalent non-deterministic one, but they run very fast because the choice of paths is locally deterministic.

For a second example, take Fig 1 in Karttunen's 2001 paper.



This automaton represents the information that the comparative form of the adjective *big* is *bigger*; ϵ here is the empty string. The automaton is inherently bidirectional: the same transducer can be used for analysis (surface input, “upward” application) or for generation (lexical input, “downward” application.)

3. Ways of looking at a finite state transducers

- We can think of the input symbols as appearing on one tape (usually called the input tape) and the output symbols on another tape (usually called the output tape.) In that case the transducer is in *translating* mode, from ‘input tape’ to ‘output tape’, or from the first tape to the second tape. (For the first automaton above, it translates *ah* to *bh* and *ae* to *ce*.)
- Reverse translation: We can just as well use the same machine to translate from the second tape to the first. So the first machine in section 2 would then translate *bh* to *ah* and *ce* to *ae*.
- Generation mode. The transducer works like a finite state automaton run in generation mode, but writes a pair of output strings rather than just one. (Our automaton generates the pairs of output strings $\langle ah, bh \rangle$ and $\langle ae, ce \rangle$.)
- Acceptance mode. (Recognition mode). Both tapes are input tapes; the two current input symbols have to match the two symbols on the transition arc in order to make a transition. (Our automaton accepts the pairs of input strings $\langle ah, bh \rangle$ and $\langle ae, ce \rangle$.)

4. Cascading finite state transducers

A finite state transducer defines a *regular relation*. “Cascading finite state transducers” corresponds to performing a **composition of relations** to produce a new relation.

From Karttunen (2001):

As is well-known, phonological rewrite rules and two-level constraints can be implemented as finite-state transducers (Johnson 1972, Kaplan and Kay 1981, Karttunen et al. 1987).

The application of a system of rewrite rules to an input string can be modeled as a cascade of transductions, that is, a sequence of compositions that yields a relation mapping the input string to one or more surface realizations.

Karttunen illustrates with a system of three ordered rules for vowel alternation in Yokuts; see his paper.

5. Closure properties for regular relations

I am taking this from Karttunen (2001) p.5.

- Regular relations, like regular expressions, are closed under
 - Union
 - Concatenation
 - Iteration A^+ (one or more concatenations of members of A)
 - Kleene star A^* (union of empty string or relation and A^+)
 - Optionality (A) : union of A with the empty string or empty relation
- Regular relations, unlike regular expressions, are not in general closed under
 - Complement
 - Intersection
- Regular relations are closed under composition $A \circ B$

Karttunen discusses a number of operators that may be added to the syntax of regular expressions that are very useful in linguistic applications, making linguistically common constructions much easier to represent without expanding the formal power beyond that of regular expressions. Three that he focuses on are *restriction*, *replacement*, and *marking*. He discusses their usefulness in morphology and in sub-areas of syntax. He argues for the high value of working with regular expressions in computational linguistics. He makes no claims regarding their value in theoretical linguistics or models of human processing.

References

- Frank, Robert, and Satta, Giorgio. 1998. Optimality theory and the generative complexity of constraint violability. *Computational Linguistics* 24:307-315.
- Johnson, C. Douglas. 1972. *Formal Aspects of Phonological Description*. The Hague: Mouton.
- Kaplan, Ronald M., and Kay, Martin. 1981. Phonological rules and finite-state transducers: Abstract. Paper presented at *Linguistic Society of America Meeting Handbook, Fifty-sixth Annual Meeting*, New York. Reprint.
- Kaplan, Ronald M., and Kay, Martin. 1994. Regular models of phonological rule systems. *Computational Linguistics* 20:331-378.
- Karttunen, Lauri, Koskeniemi, Kimmo, and Kaplan, Ronald M. 1987. A compiler for two-level phonological rules. Stanford: CSLI, Stanford University.
- Karttunen, Lauri. 1998. The proper treatment of Optimality Theory in computational phonology. Paper presented at *FSMNL'98. International Workshop on Finite-State Methods in Natural Language Processing, June 29-July 1, 1998*, Bilkent University, Ankara, Turkey. Reprint.
- Karttunen, Lauri. 2001. Applications of Finite-State Transducers in Natural Language Processing. In *Implementation and Application of Automata. Lecture Notes in Computer Science Volume 2088*, eds. S. Yu and A. Paun, 34-46. Heidelberg: Springer Verlag.
- Koskeniemi, Kimmo. 1983. *Two-level Morphology: a general computational model for word-form recognition and production*. Helsinki: University of Helsinki.
- Koskeniemi, Kimmo. 1985. A general two-level computational model for word-form recognition and production. In *Computational Morphosyntax: Report on Research 1981-1984*, ed. Fred Karlsson, 1-18. Helsinki: University of Helsinki.
- Koskeniemi, Kimmo. 1986. Compilation of automata from morphological two-level rules. In *Papers from the Fifth Scandinavian Conference of Computational Linguistics*, ed. Fred Karlsson, 143-150. Helsinki: University of Helsinki.