Massachusetts Institute of Technology
6.863J/9.611J Natural Language Processing, Spring, 2003
Department of Electrical Engineering and Computer Science
Department of Brain and Cognitive Sciences

Handout 2: Notes #2: FSAs, Morphological Processing, and KIMMO

Agenda:

1. Finite transition networks, machines, transducers
2. Morphological processing: basic principles
3. The Kimmo system: an outline of how it works
4. A spelling change automaton
5. Racing through an example
6. The complexity of Kimmo; what is to be done?

1 Basic definitions—finite transition networks

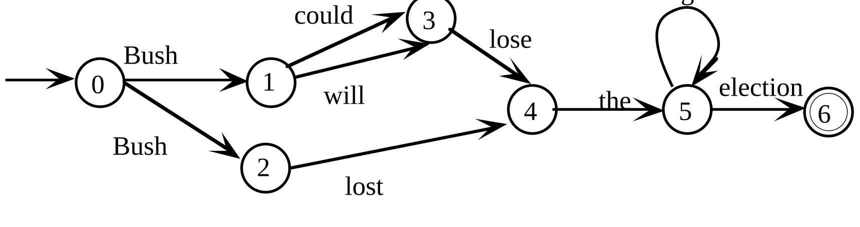
(Shannon and Weaver, *The Mathematical Theory of Communication*, 1949)

The simplest case: base all grammatical relations on left-to-right (binary) precedence relation ρ. This models pure concatenation only; therefore, assumes the only structure is left-to-right order.

Note properties of this relation: $x_1 \rho x_2$ if x_1 precedes x_2. Transitive, antisymmetric (if $x_1 \rho x_2$ then $\neg x_2 \rho x_1$). Since precedence is a binary relation, we can obviously represent it as a 2-D matrix, or a graph.

Three computational models capture exactly this relation:

- finite-state automata
- finite transition networks (FTNs)
- right/left linear grammars



Definition 1: A *finite graph network* G is a finite set of triples (p, w, q) where $w \in \Sigma$ = a (finite) set of *labels*, or *alphabet* (*vocabulary*); $p, q \in Q$= a (finite) set of *nodes* or *states*.

Definition 2: A *path* p through a finite graph network G is an ordered set of elements of G in the form $(q_0, w_1, q_1), (q_1, w_2, q_2), \ldots, (q_{k-1}, w_k, q_k)$. The *sentence* or *string* associated with a path p is the sequence of labels $w_1 w_2 \cdots w_k$.

Notation: ϵ (sometimes λ) denotes the empty string. Given a finite alphabet Σ, $w^* \in \Sigma^*$ denotes the set of all strings over Σ. Strings are formed via concatenation, and obey the following properties, $a, b \in \Sigma^*$:

(Associativity) $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

(Left and right identity) $a \cdot \epsilon = \epsilon \cdot a = a$

(Thus algebraically the set of strings is a *monoid*.)

Definition 3: A *finite transition network* (FTN) G is a graph network plus:

1. A distinguished start state$\in Q$, marked with an entering arrow that comes from no preceding state;
2. A set of final states $F \subseteq Q$, marked with a double circle.

Definition 4: A sentence (string) $w = w_1 w_2 \cdots w_k$ is *accepted* (*recognized*, *generated*) by an FTN G iff there exists a path in G with initial element in the form (q_0, w_1, q_1) and final element in the form (q_{k-1}, w_k, q_k), s.t. q_0 is the start state, and $q_k \in F$.

Definition 5: The *language* L accepted (recognized, generated) by FTN G, denoted $L(G)$ or simply L is defined as $\{w | w \text{ accepted by } G\}$. G is a *generative grammar* for the language L. L is G's *weak generative capacity*.

Definition 6: A *parse* of w with respect to FTN G is a (representation of) the path(s) such that $w \in L(G)$. (What happens when $w \notin L(G)$?) The set of paths (parses) generated by G is *its strong generative capacity*.

Definition 7: A sentence w is *ambiguous* if there exists more than one distinct path such that $w \in L(G)$. An FTN G is ambiguous if at least one of its sentences is ambiguous.

language can have more than one grammar. Ambiguity and parsing are relative to particular FTNs (grammars). An ambiguous sentence has two or more structural descriptions in terms of paths. The intent is that each path has a different meaning (under some sense of "meaning"). Note that this is just our compositionality (Fregean) assumption again. Example: *fruit flies like a banana*. Note that only lexical (word category) ambiguity is possible for FTNs.

Note. Parsing is harder than recognition. Sometimes, it is *much* harder.

Definition 8: An FTN G is *nondeterministic* if there exists some state q such that there is more than one edge labeled w_i leaving q, and *deterministic* otherwise.

Note. Ambiguity$\rightarrow$nondeterminism; but nondeterminism$\not\rightarrow$ambiguity.

Definition 9: A *finite-state automaton* (fsa) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite nonempty set of *states*, Σ a finite, nonempty *vocabulary*, or *alphabet*, q_0 a (unique) *start state*, $F \subseteq Q$ the set of *final states*, and δ, the *transition mapping*, a mapping that goes from (state, alphabet symbol) pairs to subsets of new states that is, from $Q \times \Sigma \mapsto 2^Q$.

finite-state automaton is simply a mathematical version of an FTN, under an obvious 1-1 correspondence between graph nodes and states, labels and alphabet symbols, and graph edges and the transition mapping. An fsa is deterministic if δ is a function, and nondeterministic if it is a general relation, mapping to any subset of states.

trong generative capacity and FTNs. Even a cursory glance shows that FTNs are insufficient for describing natural languages, because of insufficient strong generative capacity. Consider ambiguity again: *a dark blue sky*. FTNs must obey associativity, under concatenation. What does this say about phrases and this example?

eak generative capacity and FTNs. FTNs are even insufficient, it appears, to describe the possible *words* in some languages, like Bambarra (Mali), where words can be in the form *Noun+o+Noun*, as in *wulu o wulu* ('whichever dog'). Because: can form patterns of the form $a^n cb^n$. Example: *wulu+nyini+la*='dog searcher'. Then:

wulunyinina+nyini+la = 'one who searches for dog searchers'

wulunyininanyinila+nyini+la = 'one who searches for one who searches for dog searchers', . . .

Now, the coup de grace: combine this with the *Noun o Noun* pattern:

wulunyininanyinila+o+wulunyininanyinila = 'whichever one who searches for dog searchers'

wulunyininanyinilanyinila o wulunyininanyinilanyinila= etc!

Is this analysis correct? Can one really extend this forever?

2 Morphological processing

Morphological (word) parsing demonstrates *all* the problems of full-blown parsing. Plus, it's a key part of all NLP systems.

We can define *morphological processing* as retrieving the information carried by individual words, as they contribute to the meaning and the structure of sentences.

There are several different kinds of morphological information:

- *Inflectional* morphology refers to the way in which a surface word form relates to the pure syntax of a sentence, e.g., adding an *s* to a verb to reflect "agreement" with the verb and also a particular tense, as in English with *he hits the ball* vs. *we hit the ball* (null marker in the second case). In particular, inflectional morphology does *not* change a word's syntactic category.

- *Derivational morphology* refers to the relation of one word to another; classically, it is said to change a word's syntactic category. For example: *revolt–revolution* (Verb to a Noun); *transmit—transmission*. Note that the last case shows that derivationally related forms may have noncompositional meanings (that is, meanings that are not the sum of their parts): *transmission* can be the transmission of a car. (In this way they behave much like idioms.) Inflectional forms are purely compositional. It's common in other languages to have diminuative forms, e.g., *libro–librito* in Spanish.

- *Clitics* like the *n't* in *don't* or as in *l'article* in French, are part of the phonology of a host word, but don't specify the properties of that word. They are of two sorts: (i) simple clitics that have their own syntactic category (like *n't*) but need to be glued onto another word to get pronounced; and (ii) others that are part of the syntax, like the *'s* (the *genitive* or *possessive* marker) that is glued onto a phrase, as in *the guy from Boston's name*. (Note where the *'s* occurs and what it modifies; we don't have *the guy's from Boston name*.)

Why not just use a dictionary to look up this information? There are two general reasons why you shouldn't or can't:

1. Some languages associate a single meaning with an astronomically large number of distinct surface word forms. Example: Finnish, where the verb categories multiply out to tens of thousands of potential surface forms for every such verb. (Hankamer estimate: 600 billion forms for Turkish)

2. Speakers don't represent what they know about words with a list. They can recognize new inflected forms and nonsense words; this argues for some kind of rule-based regularity, not a memorized list. (Though as we will see *access* may be based in part on memorization.)

Conclusion (Anderson): we look up the pieces of the word, as in,

mis	+	interpret	+	ation	+	s
'mis'		'interpret'		noun form		plural

But, we might not find all the pieces in the dictionary,

karahka	+	i	+	ta
'stick'		plural		partitive

Here obviously several spelling-change rules must be undone. Starting from the decomposed form, there is a rule that changes the *a* to an *o* before an *i*; a second rule that eliminate the *t* between short vowels like *i* and *a*; and finally a rule that turns the *i* into a gliding sound *j* between the two vowels *o* and *a*.

This leads to the following general plan for morphological analysis:

- Starting from the surface form, *undo* the phonology by inverting the rules. Retrieve a list of the possible representations that could underlie this form.

- Paste these together and return the composition of their individual meanings.

Let's review this approach and its problems, summarizing some of the material from Anderson's (1989) paper.
Step (1) is possible only if the phonological rules are truly invertible. C. Douglas Johnson (1972) showed that the spelling change rules could be described as finite-state automata, and be inverted, if one assumed that they were simultaneous (applied as a group, like equations).

What is the problem with invertibility? Consider this example. Here are spelling changes that add *s* to English words. These rules convert *try* to *tries*, *fox* to *foxes*, *book* to *books*, and so forth. The CAPITAL letters are the underlying forms in the dictionary, while the small roman letters are the surface forms.

1.	S	→	es/*Sibilant*_____
2.	S	→	es/Y_____
3.	S	→	s
4.	Y	→	y/_____i
5.	Y	→	i/_____*Vowel*
6.	Y	→	y
7.	T	→	t
8.	R	→	r

Sibilant={ch, s, sh, x, z}
Vowel={a, e, i, o, u}

For example, the dictionary form *TRYS* is just *try* plus the suffix *s*. The mapping rules change it to the spelling form that we all know as *tries*.

How do the mapping rules work? The interpretation of a rule such as $Y \rightarrow y$ is that the dictionary letter *Y* may be rewritten as the output form *y*. That is, no matter where we see a *Y* we can *rewrite* it with the new symbol *y*. Since we carry out this replacement anywhere, regardless of the surrounding letters, this is a *context-free* rewrite rule. Rules 1, 2, 4, and 5 are different. They also specify a rewriting directive, but the notation after the slash means that the rewriting can *only be carried out in the context specified after the slash*, where the underbar stands for the symbol on the lefthand side of the rule that is being rewritten. Therefore, these are *context-sensitive* rules. For example, rule 2 says that *S* can be rewritten as *as if* that *S* has a *Y* to its immediate left. Remember that the matching here is strict: it must be a capital *Y* that is to the left, not *y*. The symbols *sibilant* and *vowel* stand for classes of letters, to save writing.

To actually apply the set of rules to a given word, we try to match them to the supplied dictionary word *in the order given above*, top to bottom.

Let's try an example to see how this works. Start with *TRYS*. We now run through the list top to bottom. Rule 1 cannot apply because the *S* is not followed by a sibilant (*ch*, *s*, etc.). Rule 2 can trigger, because *S* is preceded by *Y*. So we replace *TRYS* with *TRYes*. Now rule 5 can fire: the *Y* is before a Vowel, namely, *e*. So we replace *Y* with *i*, yielding *TRies*. Finally, rules 7 and 8 replace the *T* and the *R* to give *tries*.

Step	Form	Rule applied
0	*TRYS*	2 $S \rightarrow es/Y$_____
1	*TRYes*	5 $Y \rightarrow i/$_____*Vowel*
2	*TRies*	7 $T \rightarrow t$
3	*tRies*	8 $R \rightarrow r$
4	*tries*	None

In this particular example, the rules are arranged so that we don't have to rescan the entire list of rules each time. Why are they ordered in this way?

The mapping procedure gives us the spelling if we know the dictionary entry, but our lookup routines need to go in the other direction. Given a spelled-out word, we need to find the dictionary form. To reverse the process, one simple idea is to reverse the arrows:

$$\begin{array}{llll} 6' & y & \rightarrow & Y \\ 5' & i & \rightarrow & Y/\underline{\qquad}\,\mathit{Vowel} \end{array}$$

What about the ordering of the reversed rules? How should we start with *tries* and go backwards to get TRYS?

While this one example works fine, it is hand-tailored. In fact, the ordered reversed-rule system is peculiar because it won't be able to recover some potential dictionary forms. For example, suppose there was an (underlying) dictionary entry *TRIES.* The forward-working spelling change rules would produce *tries* (assuming two new rules mapping S to s and E to e). Therefore, we would now have an ambiguous spelling on the surface. Just as in parsing, our goal is to map from the spelling form to the dictionary entry, but our rule system would recover only one possibility. This problem pops up for many words where there are two or more underlying dictionary forms: *ranged* could be either *RANG+ED* or *RANGE+ED.* Thus the recovery of dictionary entries seems to be nondeterministic.

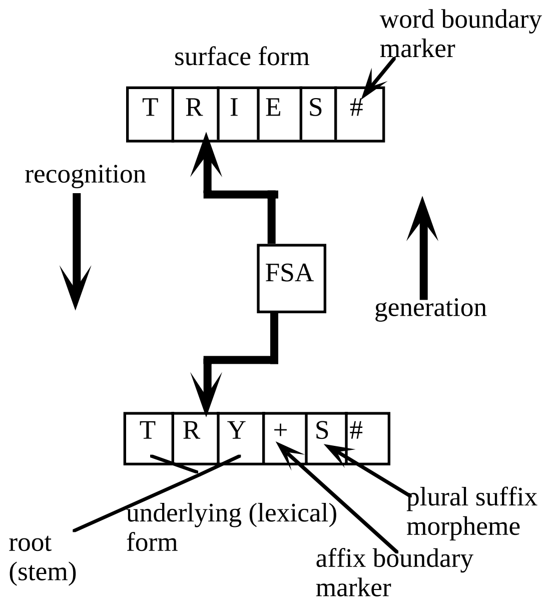
Besides nondeterminism, a second problem with the reversal method is that the rules must be carefully ordered to work. While rule interactions seem easy to work out for small sets of rules, in a full-scale system of spelling changes it may not be so easy to see what to do.

To fix this problems, the KIMMO system assumes that all rules are *simultaneously* applicable. It is fairly easy to show (as C.D. Johnson did in 1972) that *if* this assumption is made then the resulting system can generate only finite-state languages. Since it is the relation between a surface and underlying form that is finite-state, we might call it a *finite-state relation.* However, if we allow ordered rewrite rules of any sort, then it is likewise easy to show that we can simulate any Turing machine computation i.e., any computable procedure at all. Since an arbitrary computable procedure can't be inverted—it might not even terminate—the nondeterminism in reversing things could be deadly. Hence KIMMO. It was also argued by the original authors of KIMMO that such as system would be *inherently* fast because the finite-state character would guarantee linear time parsability. We shall see that those claims are formally incorrect, though they seem to hold on the average. We will return to the complexity issue and these other problems later.

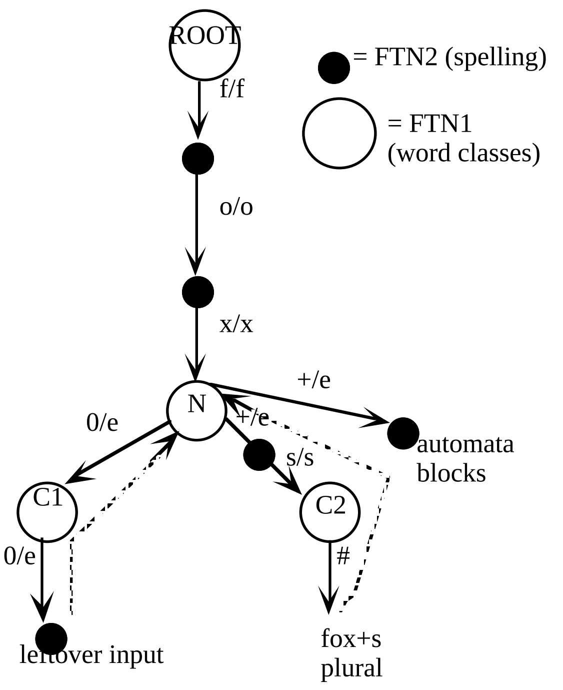
There are many other problems with the KIMMO view. There is a second problem with what it means to look up the pieces of a form in the dictionary, since for a complex word the pieces might not correspond to any distinct substring: for example, the way that *sit* and PAST are combined to form *sat* (what is called ablaut, umlaut, and other types of word mutation).

3 The KIMMO system

KIMMO consists of two sets of fsa's, run together:



- FSA1: Encodes spelling changes as a finite-state transducer
- FSA2: Encodes root+suffix combinations as series of root, suffix (lexicon) classes. Example classes for English: noun, verb, noun-singular, noun-plural, progressive, agentive, comparative. Each state of this automaton encodes a different root+suffix class, like plural nouns (C2 below) vs. singular nouns (C1), which could allow different endings.



Notation: The KIMMO system uses the following terminology.

morpheme= a minimal unit of meaning

affix= morpheme added to *root* or *stem* of a word, e.g., *s* to form plural. *Suffix*= added at end; *prefix*= at beginning; *infix*=inside word (e.g., *sit–sat*)

surface form = how a word is normally spelled

underlying form, *lexical form* = spelling form used to look up word in dictionary

lexicon= when referring to the KIMMO system, a state in FSA2 corresponding to a root or suffix class.

alternation class= a label made up of one or more *lexicon classes*, used to summarize what lexicon (FSA2) states might follow an actual dictionary word.

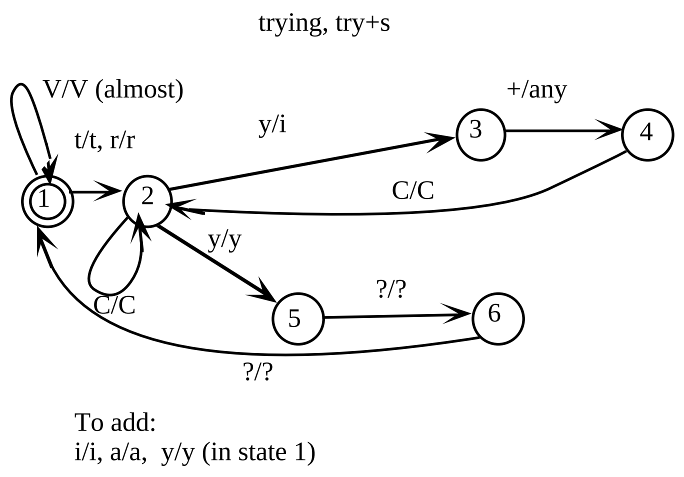
a *continuation class*= a label for FSA2 not covered as an alternation class

+ = root boundary marker

ϵ= null (empty) character

# = end of input marker

4 A spelling change example



Handling multiple spelling changes

Apply all changes in parallel (or used merged fsa), so as to enforce the simultaneous applicability principle discovered by Johnson (1972). (FTN intersection) Merger can sometimes require the cross-product of automaton states, but usually not. (Why is it the cross-product?)

Spelling rules for English. 5 rules plus 1 identity rule (required for implementation). A "full" rule set might include 50+ rules, but some systems have used up to 350 rules.

Rule	Example	# states
1. [Insert an *e* if sibilant] Epenthesis (EP)	fox–foxes; cat–cats	6
2. [doubling] Gemination (G)	cool–cooler; big–bigger	16
3. Y-spelling (Y)	toy–toys; try–tries	6
4. [drop *e*] Elision (EL)	large–larger	15
5. I-spelling (I)	lie–lying	7

In English a regular verb takes the following endings:

Category	Ending	Example	Abbreviation
First person, *I*	∅	*I like*	
Second person, *You*	∅	*You like*	
Third person, *it*	$+s$	*It likes*	P3
First person, plural *We*	∅	*We like*	
Past	$+ed$	*liked*	PS
Past Participle	$+ed$	*were liked*	PR
Progressive	$+ing$	*liking*	PP
Agentive	$+er$	*liker*	AG
Able	$+able$	*likable*	AB

The abbreviations P3, PS, PR, PP, AG, and AB are dictionary divisions that we need because, for example, a past tense verb *liked* takes a different ending from a progressive *liking*. Besides these divisions, we need others because certain verbs take some of these endings but not others. First of all, there are two main classes of irregular verbs: those that can just take the regular progressive, agentive, and able endings, but is otherwise irregular (an example is *do*, since we have *doing* and *doable* but not *doed*); and those verbs that take the third person singular, progressive, and agentive and able endings (an example is *bite*, since we have *bites* and *biting* but not *bited*).

Call the first *i*rregular *v*erb class IV1, and the second IV2. Further, there are verbs that appear just in third person, like *is* (class IP3); verbs that are irregular past forms, like *kept* (class IPS); and irregular past participles, like *bitten* (Class IPP). Adding these up, we have 6 verb dictionary divisions: regular verbs, V; IV1; IV2; IP3, IPS, and IPP. In addition, the class V leads to 7 possible choices, because a verb could have one of 7 possible endings: P3, PS, PP, PR, I(irregular), AG, and AB. Each of these acts like the state in a finite-state device, to prepare the lookup procedure for what might come next.

5 An example

In the trace that follows, we'll keep track of the transduction letter-by-letter. Indentation will indicate progress through the string, so when we backup the trace will reflect this by unindenting that line. The number in the left-hand column will mirror this indentation. The spelling change automata are traced in parallel, via a 6-element vector. Each number in the vector is the current state of that automaton. Below each part of the 6-element vector we'll indicate the abbreviation of the relevant automaton, EP, G, Y, EL, or I. The first element of the vector is always the identity (default) automaton. Each step will show the pair of characters being processed in (underlying, surface) form. The symbol 0 is used to denote the empty string. We'll now step through the recognition of *races'*, one letter at a time.

```
Recognizing surface form "races'".
0 (r.r) --> (1 1  1 2 1  1)
               EP G Y EL I

1   (a.a) --> (1 1  4 1 2  1)
                 EP G Y EL I

2     (c.c) --> (1 2 16 2 11 1)

3       (e.0) --> (1 1 16 1 12 1)
                     EP G Y EL I

4         Entry |race| ends --> new lexicon N, config (1 1 16 1 12 1)
                                                         EP G Y EL I
```

In the first 4 steps the recognizer processes the root *race* up to its end in the root lexicon, and starts searching the sublexicon N. This is entirely fortuitous. It is just because we have listed *race* as a noun first. The N sublexicon will now be searched, with the automata starting off in the states indicated. All may seem to be fine, but there's one problem: The machine assumed that an underlying *e* was paired with a surface empty character 0. That is, it is assuming that it is processing a form like *racing*. The elision transducer winds up in state 12; we'll need to keep track of this later on. This is a bad guess.

Next the N sublexicon points to two others: the first, C1, calls for the end of the word; the second, C2, calls for +*s* (the plural ending). Both of these fail, since the next surface character is an *e*. In the first case, there is either leftover input or the guess that there is an *'s* ending fails; in the second sublexicon, the epenthesis automaton blocks, because in English the combination + underlying and *e* on the surface is allowed only after strings like *fox*, not *rac*. The machine is restarted in the state configuration that we had following the recognition of *race* as a root.

```
5           Entry /0 ends --> new lexicon C1, config (1 1 16 1 12 1)
                                                        EP G Y EL I
6             Entry /0 is word-final --> path rejected (leftover input).
5           (+.0) --> (1 1 16 1 13 1)
                         EP G Y EL I
6             Nothing to do.
5           (+.e) --> automaton Epenthesis blocks from state 1.
4         Entry |race| ends --> new lexicon P3, config (1 1 16 1 12 1)
                                                          EP G Y EL I
```

We now start searching the verb sublexicons. P3 tries to find a verb with underlying form +*s*, but this fails (in two ways). Again note the nondeterminism introduced by an empty character. The machine backtracks to consider the PS (past tense) sublexicon. This choice plows a little further into the string, since this sublexicon looks for underlying +*ed* forms, and hence the underlying *e* matches with the surface *e* of *races'*, but past that things don't fare so well. Without going into the details, the searches through sublexicons PP, PR, I, AG, and AB—all without success, since none of these allow the ending *es'*. Finally, having exhausted all these choices, we back up to the really wrong guess: the one that stuck an empty character after *c* but before *e*.

```
5           (+.0) --> (1 1 16 1 13 1)
                       EP G Y EL I
6             Nothing to do.
5           (+.e) --> automaton Epenthesis blocks from state 1.
4         Entry |race| ends --> new lexicon PS, (1 1 16 1 12 1)
                                                 EP G Y EL I
5           (+.0) --> (1 1 16 1 13 1)
6             (e.0) --> automaton Elision blocks from state 13.
6             (e.e) --> (1 1 16 1 1 1)
7               Nothing to do.
5           (+.e) --> automaton Epenthesis blocks from state 1.
4         Entry |race| ends --> new lexicon PP,  (1 1 16 1 12 1)
5           (+.0) --> (1 1 16 1 13 1)
6             (e.0) --> automaton Elision blocks from state 13.
6             (e.e) --> (1 1 16 1 1 1)
7               Nothing to do.
5           (+.e) --> automaton Epenthesis blocks from state 1.
4         Entry |race| ends --> new lexicon PR, (1 1 16 1 12 1)
5           (+.0) --> (1 1 16 1 13 1)
6             Nothing to do.
5           (+.e) --> automaton Epenthesis blocks from state 1.
4         Entry |race| ends --> new lexicon I, (1 1 16 1 12 1)
5           Entry /0 is word-final -->rejected (leftover input)
4         Entry |race| ends --> new lexicon AG, (1 1 16 1 12 1)
5           (+.0) --> (1 1 16 1 13 1)
6             (e.0) --> automaton Elision blocks from state 13.
6             (e.e) --> (1 1 16 1 1 1)
7               Nothing to do.
5           (+.e) --> automaton Epenthesis blocks from state 1.
4         Entry |race| ends --> new lexicon AB, (1 1 16 1 12 1)
5           (+.0) --> (1 1 16 1 13 1)
6             Nothing to do.
5           (+.e) --> automaton Epenthesis blocks from state 1.
```

Now instead of inserting the empty character, we map *e* to an underlying *e*. This finishes off *race* in the root lexicon, and we start searching the N sublexicon. Note that this time the e-elision state is 14 rather than 12. We're still not done yet. We first try searching sublexicon C1, but this fails because C1 allows only no ending or *'s*. We turn to C2 (plural Noun endings). First we try a zero ending, but of course this is a deadend. Finally, we try the other possibility, *'s*, and this succeeds—at last. The parser returns *races'* as a genitive, plural, noun.

```
3       (e.e) --> (1 1 16 1 14 1)
                    EP G Y EL I
4        Entry |race| ends --> new lexicon N, (1 1 16 1 14 1)
                                                EP G Y EL I
5         Entry /0 ends --> new lexicon C1, config (1 1 16 1 14 1)
6          Entry /0 is word-final -->rejected (leftover input)
5           (+.0) --> (1 1 16 1 15 1)
6            (s.s) --> (1 4 16 2 1 1)
7             Entry +/s ends--> new lexicon C2,  (1 4 16 2 1 1)
8              Entry /0 is word-final -->rejected(leftover input)
8               ('.') --> (1 1 16 1 1 1)
9                End --> lexical form ("race+s'" (N PL GEN))
```

In fact, we're not quite done. If we wanted to retrieve *all* possible entries, we would have to go on, since the word might be ambiguous. In this case, all the other possibilities fail (we search all the verb sublexicons as before, but now with the e-elision automaton starting from state 14 rather than state 12).

6 The complexity of KIMMO

Kimmo does backtracking for *both* recognition and generation. There are two sources of non-determinism, because there are *two* finite-state transducers in KIMMO: the spelling change automaton and the dictionary automaton. In addition, the spelling change automaton can posit null characters on the surface.
Example:

```
Generating from lexical form "fox+s"
Setq count: 5
Enqueue count: 1

 1   f     1,1,1,2,1,1     11 +   foxg    XXX Gemination
 2   fo    1,1,4,1,2,1     12 +   foxf    XXX Gemination
 3   fox   1,3,16,2,1,1    13 +   foxd    XXX Gemination
 4   foxt  XXX Gemination  14 +   foxb    XXX Gemination
 5 + foxs  XXX Gemination  15 +   fox0    XXX 1,6,16,1,1,1
 6 + foxr  XXX Gemination  16     fox0s   XXX Epenthesis
 7 + foxp  XXX Gemination  17 (4) foxe    1,5,16,1,1,1
 8 + foxn  XXX Gemination  18     foxes   1,1,16,2,1,1
 9 + foxm  XXX Gemination  19     "foxes" ***result
10 + foxl  XXX Gemination

("foxes")
```

The formal computational complexity of KIMMO

We can quickly reduce any instance of a known intractable problem, 3-satisfiability (3-SAT) to an instance of KIMMO recognition (or generation). Thus, if KIMMO recognition or generation were computationally tractable, so would be 3-SAT. But this is highly unlikely (under the hypotheses below).

The worst case for recognition occurs when the surface form gives *no* information about what its underlying form should be, just as the variables in a 3-SAT formula give no indication as to whether they should be *true* or *false*. Unlimited global harmony forces the constraint that variable value assignments are consistent, while the dictionary only admits "words" with 3 morphemes in a row with at least one having the feature t (for *true*).

Definition 10: $\mathcal{P}$= the class of problems solvable in time n^j for some integer j (polynomial time), on a *deterministic* Turing machine, problem size parameter n.
$\mathcal{NP}$= the class of problems solvable in time n^j, for some integer j, on a *nondeterministic* Turing machine, problem size parameter n.
We assume that $\mathcal{P} \neq \mathcal{NP}$. Problems in $\mathcal{NP}$ are easy to *verify* (check their answers) in polynomial time but difficult to discover solutions for in polynomial time (no known nonexponential algorithm).

Definition 11: A problem T is *NP-hard* if it is at least as hard computationally as any problem in $\mathcal{NP}$, i.e., there exists a polynomial-time reduction from any problem in $\mathcal{NP}$ to T.
A problem T is *NP-complete* if it is in $\mathcal{NP}$ and is NP-hard.

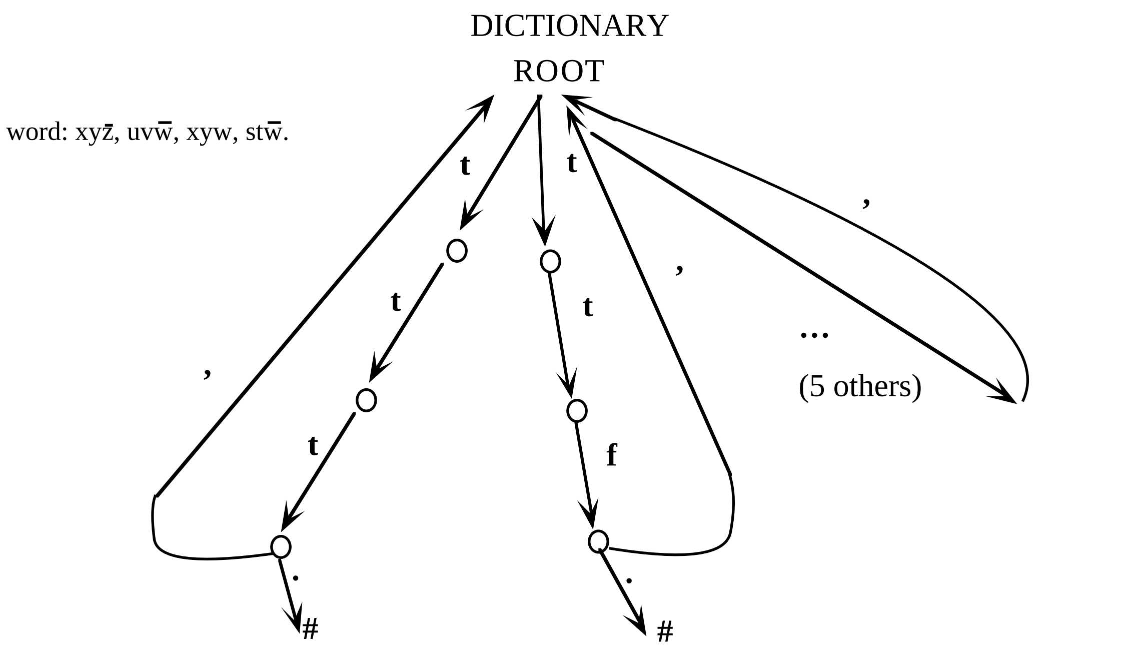
Fact: 3-SAT is NP-complete.

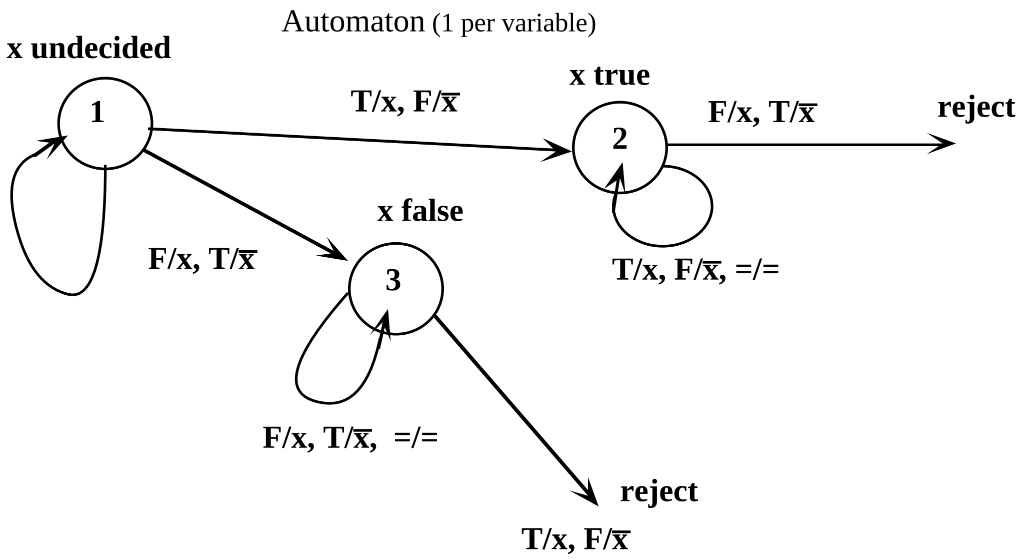
Definition 12: The KIMMO *word recognition problem* is: given an arbitrary KIMMO dictionary D, automata, word w, is $w \in D$?

Theorem 1: KIMMO word recognition is NP-complete.
Proof. By reduction from 3-SAT.

Given input formula φ, construct the word form shown below, construct 1 automaton per variable and use the fixed dictionary shown below.

Claim: Word w is in the dictionary iff φ is satisfiable. The construction takes polynomial time. ∎





What are the restrictions that might be imposed to repair this problem?

1. Limit global harmony processes. In fact: languages of the world don't use more than two or three global harmony processes.

2. Assume unambiguous dictionary continuations so that parsing is unambiguous (i.e., surface cues tell us directly what the underlying form should be). (This explains away the complexity directly.)

3. Rely on average case.

A good test language is Warlpiri, which exhibits significant harmony and reduplication effects, but certainly fewer than two independent harmony processes. The results are as expected. For example, here is the trace of the system recognizing the surface form *pinyi* (the hyphens are inserted for readability only). Recall that the system consists of two sets of finite-state automata, one that checks for each surface—underlying form pairing type, and other that checks for possible co-occurrences of stems and roots. In the trace below, each backtracking point is numbered in square brackets and refers to that state to which the system must return and proceed from. In the example that follows, there are 9 backtracks.

```
Recognizing surface form "pi-nyi".
1.(0)(y.0) --> automaton NYJ Orthography blocks from state 1.
2.(0) + (p.p) --> (1 1 1 1 1 1 1).
3.(1)(i.i) --> (1 1 1 1 1 1 2).
4.(2) Nothing to do.
5.(1) [3](<u1>.i) --> (1 1 1 2 1 1 2).
6.(2)Entry |p<u1>| ends --> new lexicon ]V3STEM, config (1 1 1 2 1 1 2)
7.(3)(-.0) --> (1 3 1 2 1 1 2).
8.(4) Nothing to do.
9.(3) [7](-.-) --> (1 2 1 2 1 1 2).
10.(4)(n.n) --> (1 2 2 3 1 1 2).
11.(5)(y.0) --> automaton <u1> Assimilation blocks from state 3.
12.(5) + (y.y) --> (1 2 1 1 1 1 2).
13.(6)(i.i) --> (1 2 1 1 1 1 2).
14.(7)Entry |-nyi| ends --> new lexicon ]END, config (1 2 1 1 1 1 2)
15.(8)End --> result is ("p<u1>-nyi" (@DAMAGE V NPST)).
16.(7) [14]Entry |-nyi| ends --> new lexicon DIRENCLITIC, config
           (1 2 1 1 1 1 2)17.(8)(-.0) --> automaton Hyphen Realization
        blocks from state 2.
18.(7) [14] Entry |-nyi| ends --> new lexicon ]WORD, config (1 2 1 1 1 1 2)
19.(8)(#.0) --> automaton Hyphen Realization blocks from state 2.
20.(7) [14]Entry |-nyi| ends --> new lexicon SUBJ, config (1 2 1 1 1 1 2)
21.(8)(-.0) --> automaton Hyphen Realization blocks from state 2.
22.(7) [14]Entry |-nyi| ends --> new lexicon OBJ, config (1 2 1 1 1 1 2)
23.(8)(-.0) --> automaton Hyphen Realization blocks from state 2.
24.(7) [14]Entry |-nyi| ends --> new lexicon RLA, config (1 2 1 1 1 1 2)
25.(8)(-.0) --> automaton Hyphen Realization blocks from state 2.
26.(7) [14]Entry |-nyi| ends --> new lexicon ], config (1 2 1 1 1 1 2)
27.(8) (-0.) --> automaton Hyphen Realization blocks from state 2.
28.(1) [3](I.i) --> (1 1 1 1 1 2 2).
29.(2)Nothing to do.
"pi-nyi" ==> ("p<u1>-nyi" (@DAMAGE V NPST))
```

The figure displays the resulting graph of word length vs. backtracking for a distribution of over 80 Warlpiri words. Note that the two-level system does substantial backtracking: the amount of backtracking grows linearly with input length.

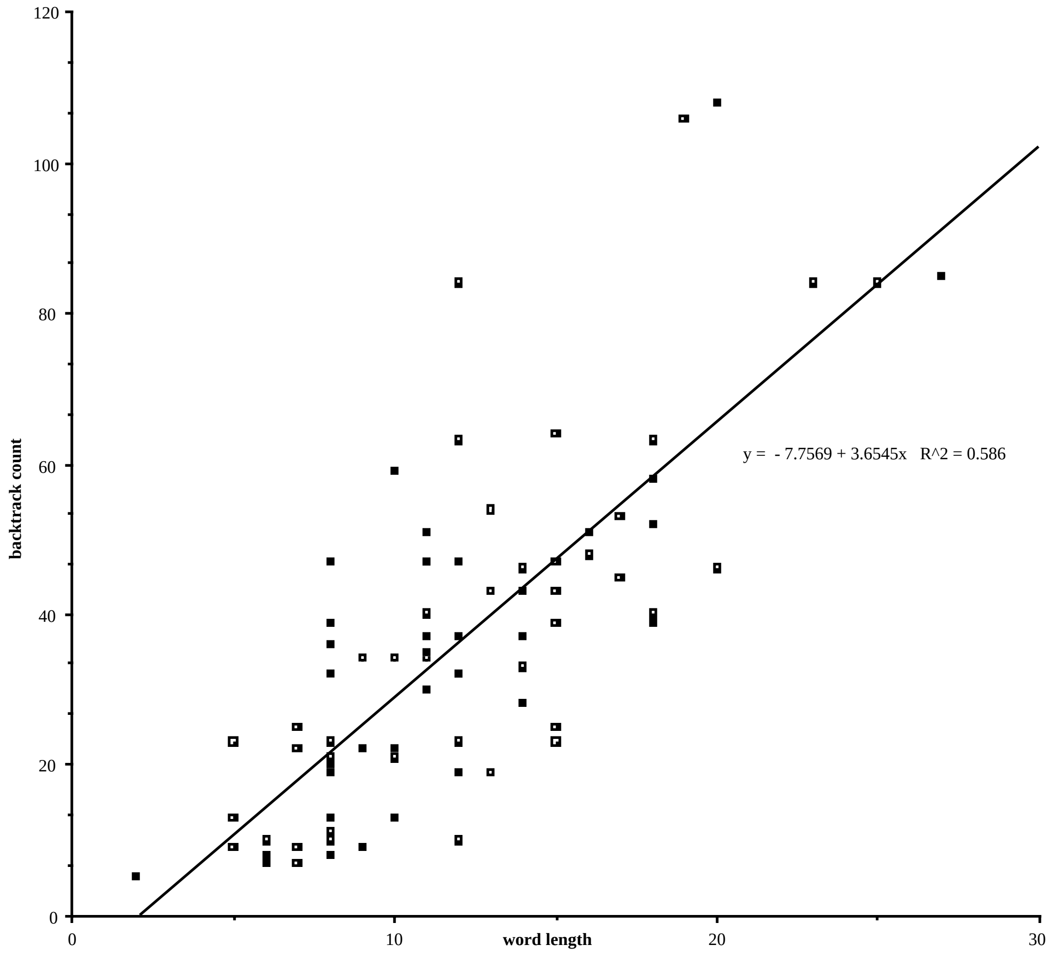


Figure 1: A graph of the backtracking done by the two-level morphological analyzer for 81 Warlpiri words. The expected backtracking is linear in word length, reflecting the relative sparsity of "hard" SAT problems along with the nonambiguity of some underlying word decompositions.