MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
Department of Brain & Cognitive Sciences
6.863/9.611J Spring 2004, Natural Language Processing

Laboratory 1, Component I: Word Parsing – Computational & Linguistic Background

1. General overview & purpose of this document

Since this is the first laboratory of the course, it would be useful to set out the structure that all the laboratories will follow. Our basic approach to understanding natural language processing will be to divide the subject matter into two parts: *what* knowledge of language is – its representation – and *how* that knowledge is put to use – its computation. Roughly, one can think of this in the usual computer science terms as the distinction between *data structures* and *algorithms,* though the distinction is not hard and fast. For example, given the sentence "the dogs like ice-cream," linguists have uncovered many different kinds of representations that seem to be involved. To take just two, we know that 'dogs' has the plural marker 's' following 'dog' and not the reverse, 'sdog'. Further, in English we have to abide by the order 'the dogs' and not 'dogs the', and 'the dogs' forms a kind of unit in its own right. We know this because of sentences like the following, where "the dogs" evidently gets carried around *en masse*: "The dogs, John never liked" ; I liked the cats, and Mary, the dogs. (For more on levels of representation in linguistics and computation, refer to the first notes installment on the course web site, and to the references at the end of this document; a classic source is N. Chomsky, *Logical Structure of Linguistic Theory*, 1955).

Given this distinction, our approach will be to carve up all laboratory assignments into three components: first, the linguistic representations and the computational methods used – the *what* and the *how*; second, a warm-up component that introduces you to the computational tools; and third, a more extensive and creative component that will exercise your skill in tackling a problem in the particular area addressed – perhaps a novel language, or an extension of an existing system. You may find that components 2 and 3 are a lot like learning a new programming language: first, basic syntax and simple programs, and then, writing a more original program on your own. You may also find that you have to learn new terminology, like the keywords or schemas of a programming language – oftentimes, unfamiliar linguistic terms like *morpheme* or *X-bar theory.* There is no need to be alarmed. Learning this new vocabulary will be no different from, say, physics or any other science. All the vocabulary will be introduced along the way. (And if you don't understand some piece of terminology, by all means please just ask.) You should, however, beware of one common pitfall: beginning students quite naturally carry into this domain – their baggage from elementary grammar school. Everyday notions like *grammar* and what's a 'grammatical' sentence don't carry their usual everyday meanings. 'Grammatical' doesn't mean what your teacher told you was proper English (or Hindi, or Chinese). For the most part, you'll have to check this conceptual baggage at the door, just as you have to abandon your notions about Newtonian mechanics to get to relativity theory (or abandon naïve physics to get to Newtonian mechanics).

This, document then, is the first of three written components that make up laboratory 1. It serves as computational and linguistic background and guidance to the laboratory itself. Its purpose is to familiarize you with the formal computational machinery for 'parsing' words, and introduce you to the basic computational engines used today for this – namely, finite-state transducers. This is the 'how' of word parsing, also loosely called *stemming.* Additionally, we

provide a brief introduction to the 'what' of word parsing – what linguists call *morphological analysis*, *morphotactics*, or *morphophonology.* Both computational and linguistic aspects of morphological analysis have a large literature; for additional readings on all the matters discussed here, please see the reference pointers at the end of this document. (If you are already comfortable with the linguistic terminology for talking about words, as well as the technology for finite-state machines, you might want to skip ahead to section 4 of this document, which describes the particular implementation for word parsing that we shall use, Kimmo, but I would suggest you not to.)

So much for general introduction. Here's a roadmap to the remainder of this document. First, we'll see what representations and computational machinery we need to chop words into parts, so as to look them up in a dictionary while adjusting for spelling changes, a method applicable to many (but not all) languages. Here we'll introduce *finite-transition networks* and *finite-state transducers* as the computational engines of choice, along with the formal operations on networks/transducers of use in language modeling, such as composition and concatenation. Second, we shall see how to use this technology to build rule systems that can model the mapping between the word forms we see in text, such as *cats*, and their corresponding underlying, internal forms, of more use to a computer, such as *cat* + Plural. Third, we'll show how to account for spelling changes, as when *y* changes to *i* before an *e,* as in *tries.* Finally, we describe the computational implementation we shall use, Kimmo. By the time you've finished, you should be well prepared for the introductory exercises in lab component II.

2. Introduction: Why Word Parsing?

We begin our study of natural language processing with what might at first blush seem like the most mundane and innocent of all topics: words. We all 'know' what words are. So did Hamlet (Polonius: What do you read, my lord? Hamlet: Words. Words. Words. *Hamlet,* Act II, Scene ii). All human languages use words – lots of them (By recent count, as of 2001, there were 76,598,718,000 English words on the whole of the Web – many repetitive repetitive, of course!) In truth, all human languages we know of have the potential to have an indefinitely large numbers of words (just think of a simple example such as, *missile, antimissile, anti-anti-missile,…*). Yet as we shall see, this seemingly innocuous notion of a 'word' highlights all the key issues of natural language processing. So that is one important reason we start here.

Secondly, handling words is an engineering necessity in its own right: the words are outside the computer, and we have to get information about them inside. For example, we need to label words as nouns and verbs, and know that both *cat* and *cats* come from the same root, if only to build the most trivial kind of web search engine. Likewise, if the computer is to answer, we will have to get the words out again, so word parsing is an integral part of nearly all NLP systems. However, we must mention one big caveat. In this course we shall put to one side the fascinating and important question of the usual input/output modality involved in human language: speech (and its modality-specific relatives such as signing). Computer speech recognition (and generation) is a very challenging problem in its own right. But we shall side-step it, and instead make the (perhaps improper) idealization that some other method has provided us with a textual, or *orthographic* input representation, noting where this idealization might lead us astray, or how to accommodate certain aspects of speech, such as intonation or stress. In these laboratories, for the most part we'll be doing natural language *text* processing.

Even with this idealization we've still got to map from the *external* form of words on paper, Ascii, Unicode, or whatever into an *internal* format that a computer can manipulate for further language processing. In so doing, we underscore our two central issues of 'what' and 'how': what should the external and internal formats look like, and how do we carry out this mapping, in the best case, efficiently? (And finally, for those with a cognitive science bent, we can ask whether these procedures connect in any way to what people actually do.)

In particular, the information gleaned from the words captured in an internal representation is used further down the natural language processing pipeline – for example, a system must figure out that 'dogs' is a Noun, with a plural marker on the end, if only to look up in its internal dictionary some representation of the meaning of 'dog.' So word parsing implements a particular kind of divide-and-conquer approach to the whole problem of language processing. But why do we call it parsing?

We call the problem of mapping from an *external* language representation to an *internal* one (or the reverse) the problem of *parsing*. That is why we call this laboratory 'word parsing.' Note that while people often think of parsing as 'chunking' just sentences or expressions of a human or computer language into parts, we shall apply this notion in an extended way, to any external-internal form mapping. Note further that since parsing is a *mapping*, in general one internal (or external) form might map to more than one internal form. We call this situation *ambiguity* and it is a very important part of all natural languages, which embrace ambiguity, as opposed to formal or artificial languages, which try to avoid it. For example, the word 'flies' is either the plural form of the noun 'fly' (more than one of them), or it is a present tense of the verb 'fly' – in isolation, we don't know which. How we deal with ambiguity will prove to be a central computational issue in natural language processing.

So what should we take as the external and internal representations of words? For the external representation, we have already assumed a string of characters, e.g., 'dogs' is 'd o g s'. What about 'the dogs'? Ah, we've already uncovered a hidden assumption: does 'the dogs' get fed in as 't h e d o g s' or as 't h e **B** d o g s **#**', where **B** is a special 'blank' character and # an end-of-word marker? That's the familiar task of *tokenization* as computer scientists (and linguists) call it. Again, for now we shall defer this problem, and for the purposes of this laboratory assume that words come to us delimited by some end-of-word marker #, and that word strings come separated by one or more blank spaces, and that contractions like 'we'll' are expanded into their full forms 'we will.' (We'll not brush the blanks under the rug forever – the laboratory tools have special procedures for tokenization, a challenge in its own right. After all, Latin was neverwrittenwithanyspaces, possibly to save valuable parchment area. If you want, you can use Google to search for 'tokenization' and you'll discover many programs, from Perl to C, do to this job.)

2.1 The first step: breaking down words into parts

OK, so far we've just been dodging data. Time to wrestle with some. What is word parsing about? We shall approach this by first figuring out what parts words can be chopped into, and then how those parts can be put together – in short, building a dictionary for words and their components. We should remark that this is not the only possible approach, nor even the fastest when implemented. It is, however, the most principled and broadly applicable. Another well-known strategy works without a dictionary, using language-particular rules to break

words down. (One of the best-known methods here is called the *Porter algorithm,* after its author (1980). It is a purely algorithmic approach to stripping the endings off of words, and then patching these in certain cases with simple repairs. For example, this method will take the word *hoped* and strip off the *ed,* yielding 'hop'. If a word is too short, it adds an 'e', which applies in this case, to give the correct result, 'hope.' A typical mistake is to take *wander* and chop off the *er,* leaving the (incorrect) *wand.* Since it does not use a dictionary, it can be fast; but also, as just shown can make mistakes. We'll discuss this below. See your text and http://www.tartarus.org/~martin/PorterStemmer for more description.) We shall take a more basic approach, one that is more widely applicable across all languages; there are engineering trade-offs.

What are a language's possible word forms? It is pretty easy to see that words can be broken down into indivisible parts – like atoms because these parts cannot be further divided 'by ordinary chemical means' – in this case, without destroying their meaning. Linguists call these minimal units of meaning **morphemes**. For example, in the word form 'cats', 'cat' is a morpheme, and so is the plural marker 's' – it doesn't make any sense to chop a cat up further into 'ca' because that no longer means cat; nor can 's' be further taken apart, at least from the standpoint of meaning (both can of course be taken apart in some other sense, say their sounds). Linguists call this the analysis *morphology*, after the Greek *morphos,* shape, form; and *logos,* word – the study of word forms. For those unfamiliar with this terminology, an excellent one-page summary can be found at the following site, and we urge you to consult that now.

http://pandora.cii.wwu.edu/vajda/ling201/test1materials/Morphologyoverhead.htm

Further, the morphemes in a language can be arranged together in certain linear orders but not others. Consider for example:

- Fruit, Fruitless, fruitlessness; great, greatness but not:
- *Fruitnessless, *Greatless, *Fruitness
- Cool, cooler, coolest but not: fruit, *fruiter, *fruitest

Such examples demonstrate that not only must the morphemes *less* and *ness* appear in a certain order, but also that they can follow only certain other morphemes and not others. Other combinations are ill-formed. We shall by convention mark with an asterisk (*) such ill-formed combinations. Note again that this mark does not carry any 'prescriptive' weight – it does not mean 'good' or 'bad.' Later on we shall see how to accommodate such things as dialectical variation. (You say *potato,* and I say *pohtahto*.)

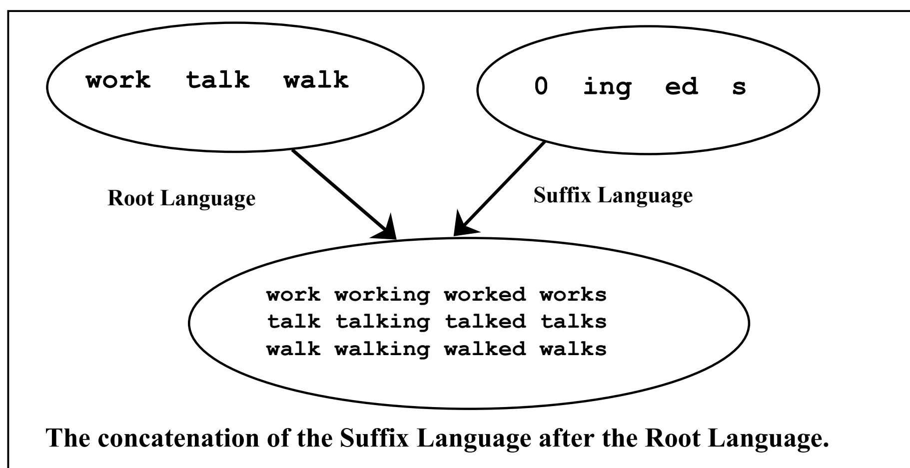
How does one figure out what the morphemes in a language are? How does one determine which morphemes can get glued together with others? There's no way but to look at the data – studying sequences like 'fruit' above. By examining which groups of morphemes can appear in the same places as others (i.e. are substitutable for one another), we can in fact formally define equivalence classes of morphemes. For instance, if *er* can appear after an English word whenever *est* can appear, then *er* would be in the same morpheme class as *est* (is it?). (In Box 2 we sketch one way to formally compute these equivalence classes.) Determining a language's morpheme classes and their possible arrangements is the study of *morphosyntax,* from *morphos* and the Greek *syntaxis*, σψνταξισ, 'to arrange together'. So we see that just

as the words in a sentence have their own possible arrangements, their syntax, so do the morphemes.

Traditionally, a linguist might write the morphemic decomposition of 'cooler' as cool+er, where the plus sign denotes concatenation. The 'cool' part that can stand alone as a word form is dubbed the *root* or *stem.* The 'er' ending, which cannot appear in isolation unless it's glued in some way to a stem, is called an *affix.* (Well, er, maybe not). A *prefix* appears at the beginning of a word (*un*clear); a *suffix* at the end (hard*ly*); and an *infix* somewhere word internally (Tagalog: um+hinigi → h*um*ingi, 'to borrow' – hmm, can you think of any familiar infixes in English?). Finally, to round out our terminology, consider the plural morpheme 's' that appears after 'cat' as in 'cats', forming the plural. But what about 'sheep'? Where's the beef? 's' now? (Note that *sheep* is ambiguously singular or plural.) Examples like *sheep* prompt linguists to posit for the sake of regularity a *zero morpheme* – an unpronounced element on the surface, but one that carries plural information just like the 's' that does appear in *cats*. More accurately, a linguist would say that 'cats' decomposes as 'cat'+Pl, where 'Pl' stands for a plural morpheme which in English *surfaces* as 's' in some contexts, but nothing at all in others, and *z* in still others (as in 'dogs') – but this is getting a bit ahead of our story. We'll go into depth about the different *alternations* of morphemes below.

Summarizing so far then, we have a system that will pair an 'external' word form, such as 'flies' – let us call this a **surface form** – with one or more corresponding 'internal' or *underlying* decompositions – let us call these the corresponding **lexical forms**. Thus, what we really are doing with word parsing is computing *pairs* of (lexical, surface) word forms.

We can now see that one way to factor apart the possible words in a language is to describe, separately, the language of *roots* and the language of *suffixes*, and then simply concatenate these two languages together, as shown below. Now we can walk the walk and talk the talk.



The concatenation of the Suffix Language after the Root Language.

Note how this factoring approach trades off memory for time: it can generate the crossproduct of the root and suffix languages, but at a cost of two lookups instead of one. This is a point to which we'll return below: if there are n_1 root forms and n_2 suffix forms, under free combination we can produce $n_1 \times n_2$ word forms using only $n_1 + n_2$ space, a great space savings for reasonably sized n_1 and n_2.

Summarizing so far, at least two kinds of knowledge seem to be required word parsing, and are, to a first approximation, *all* that we have to know:

- The root forms, because we simply can't take a surface form like *duckling* and strip of the *ing*. We know this because *duckl* is not a root
- The ordering relations, namely that only certain morphemes can be concatenated together, because we can't simply take a surface form like *beer* and strip off the *er* (as in 'doer') – that would take all the fizz out of beer

All this tells us *what* we must represent. *How* should we represent and use that knowledge computationally? Since what we have to represent is (1) equivalence classes and (2) whether one class can precede or follow another, then a minimal model would capture just this knowledge and nothing more. That is, we have a linear string of a finite number of classes. The most direct and simplest model for this is a *finite-state automaton* – which is itself just a linear concatenation of a finite number of states, with a distinguished start state and one or more final states. We expect that you have already encountered the basic definitions of terminology of finite-state automata (fsa's) and their mere mention should bring some kind of Pavlovian response to mind; if not, and you're not salivating, please do refer to the definition below.

We will first need some basic terminology for talking about strings of symbols.
An *alphabet* Σ, is a (nonempty finite) set of atomic symbols, denoted Σ.
A *string* s is a (finite or infinite) concatenation of alphabet elements, perhaps null (the empty string)
The *epsilon* or *null* character ε denotes the empty string, of zero length. (Also called λ in the literature)
The asterisk * denotes 0 or more occurrences of a symbol, e.g., $a^*=\{\varepsilon, a, aa, aaa, \ldots\}$
A *language, L,* is the set of all possible strings over an alphabet, denoted Σ^*
A *relation* R is a set of ordered pairs (x, y) over strings of some language L.

Definition A finite-state automaton (FSA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a (nonempty) finite set of states or equivalence classes, the *states*;
- Σ is a (nonempty) finite set of symbols, the *alphabet*. We will allow the existence of the empty symbol, ε, as a possible alphabet symbol.
- $q_0 \subseteq Q$ is a distinguished *start state*
- $F \subseteq Q$ is the set of *final states*
- $\delta \subseteq Q \times \Sigma \times Q$ is a mapping from $Q \times \Sigma \rightarrow 2^Q$, the *transition mapping* that takes the fsa from a state and a symbol to some set of next states, a subset of Q. If δ is a function, then there is always just one possible next state and the fsa is *deterministic*, otherwise, *nondeterministic.* We denote by $\delta(w)$ the operation of δ on some alphabet symbol w, the next state relation.

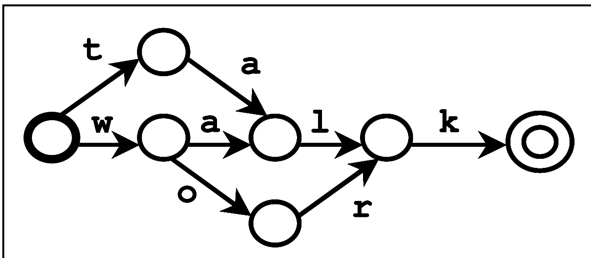
Note that we have defined FSA's so that every transition is labeled with an alphabet symbol. This lets us define an equivalent version of FSA's in the form of *finite-transition networks* (FTNs), directed graphs where the nodes are FSA states, there is a distinguished start state marked by an entering arrow that comes from no preceding state; the final states are double circles; and there is a directed edge labeled with the alphabet symbol w, between two nodes

(states) q_i and q_j iff $\delta(w)= q_j$. Informally, as we trace out paths from start state to final state(s), the sequence of graph edge labels spell out the possible strings that the fsa recognizes (accepts, generates). The FTN formulation was first proposed by Claude Shannon and Warren Weaver in *The Mathematical Theory of Communication* (1949).

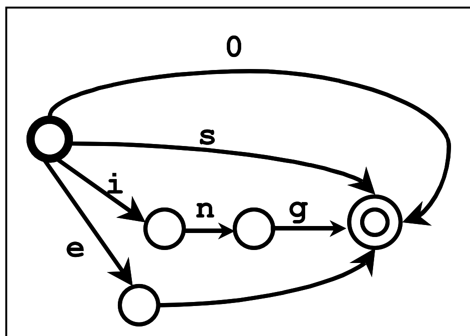
Given this formalism, we can state precisely several notions we will use later on in natural language processing. Let *G* be an FTN. The set of all such strings *G* recognizes is called the *language* recognized (generated) by *G* (or its equivalent FSA), denoted *L*(*G*). The language defined by the class of FTNs is known as the *regular languages*. Given some string ('sentence') *s,* the path through an FTN *G* from start to finish is called a *parse* of *s* with respect to *G*. A string (sentence) *s* is *ambiguous* with respect to an FTN *G* if there exists more than one distinct path through *G* such that *s* is recognized by *G,* i.e., is in *L*(*G*). An FTN is ambiguous if at least one of its sentences is ambiguous.

Note that parsing and the notion of a parse are *always* defined with respect to a particular machine (grammar), as is ambiguity. The intuition is that each distinct accepting path (parse) of a string *s,* running from start state to a final state provides a different 'meaning' of *s* under some sense of 'meaning'. For example, if we have an FTN to parse the word 'flies' then there is one path that would correspond to 'fly+Plural' (fly as a noun), and another distinct path corresponding to 'fly+present -ense' (fly as a verb). You will see this for yourself in the first laboratory.

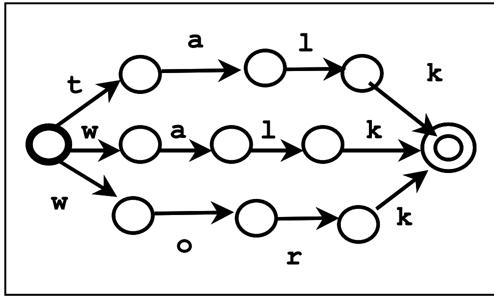
The FTN graph representation provides a rather obvious, direct picture of the valid sequence of morpheme classes (the states in the automaton or network), concatenated in a linear way. Here for example is a finite-state network for the simple walk-the-walk root language given earlier:



And a finite-state network for the simple suffix language class described earlier:

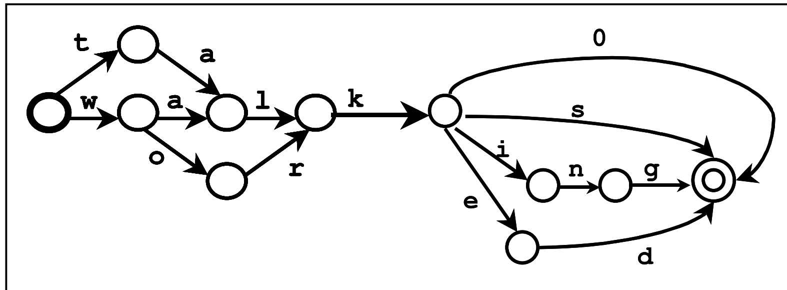


Of course, we could have written other FTNs that would recognize exactly the same strings. Here's one – the following FTN also recognizes the 'walk' language:



You'll note that we have deliberately 'compressed' our original FTNs by turning them into *prefix tree acceptors* - 'alk', 'ork', 'k' have a common prefixes, so we combine these into one edge. You can see how this saves on duplicate nodes and edges. We shall commonly use this prefix tree form for FTNs since it provides a compact (and efficient) representation for spelling possibilities.

The FTN format is particularly vivid for demonstrating set operations in an intuitive way. Using our original root and affix networks, we can *concatenate* them together by overlaying the double-circle 'final' state of the root network on top of the start state of the affix network as follows, to form a new network that spells out roots plus possible endings:



The picture should make clear that the concatenation of any two (or more) FTNs is also an FTN, and so FTNs are closed under concatenation. We leave the demonstration for a very bored reader. (What about other set operations on FTNs, such as union, intersection, or complementation? A picture tells a story…)

Let's recap. To build a root plus affix FTN to model words, we:

1. Figure out what the roots are & write them as a prefix tree FTN;
2. Determine the possible morpheme classes, corresponding to the states of an FTN, and which classes follows which other classes or roots, and write them as prefix tree FTNs;
3. We concatenate the two FTNs to form an FTN that can spell out all the root-plus-affix forms. (Question: What about prefixes, like 'un' in 'unhappy'? What would the network look like?)

Where do the morpheme classes and their order come from? We mentioned earlier that we could induce this information from a corpus of data via the relation of *substitutability*. Here we shall say just a bit more about this method, leaving formal details for later notes. Informally,

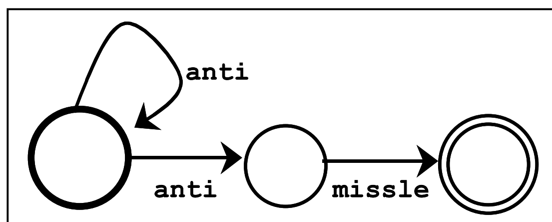
note that the set of words {*walk, talk, work*} can all be followed by the string 's' – *walks, talks, works*. However, words like *happy, sad, glad*, etc. cannot. In other words, {*walk, talk, work*} are intersubstitutable in the context ___s, where the underscore ___ indicates the place where we substitute our test strings: if '*walk*s' is in the language, then so is '*talk*s'. The substitutability relation in fact defines an equivalence class (why? If *w* is in the substitutability relation, and so is *t,* then we say that *w* and *t* are in the same equivalence class. Check that this does in fact define an equivalence class.) So, {*work, talk, work*} all get tossed into the same bucket, at least with respect to the suffix morpheme 's'.

If we continue this approach with many other words, we'll find that 'ness' has to follow 'less', as we announced at the very beginning of this document. If there turn out to be a finite number of classes, this method yields both classes (corresponding to the states of an FTN) and linear ordering (corresponding to the transition arcs of an FTN). Indeed, this approach, dubbed *distributional analysis* and pioneered by the structural linguists of the 1940s, finds succinct statement in a passage by one of their scholars, Rulon Wells of Yale:

> "[Words] are assigned to classes on the basis of the environments in which they occur. Each environment determines one and only one class, namely the class of all [words] occurring in that environment… A word *A* belongs to the class determined by the environment ____X if X is either an utterance or occurs as part of some utterance." Wells, 1947, pp. 81-82.

Sound familiar? What Wells did not know – because the technology had not been invented – was that his definition amounts to no more and no less than the definition of an FTN – that is, if his method works, then it determines an FTN. (This is easy to prove, via the Myhill-Nerode theorem for characterizing finite-state automata (1956) – see Hopcroft & Ullman (1976). If you are eager to learn more about this induction method, which is learning procedure that must work from complete data, i.e., all the examples in the language must be fairly presented, we can refer you to Berwick & Pilato, *J. Machine Learning*, 1985, who implemented this method to automatically induce FTNs from linguistic data, in another context. Do you think that this is the way that children might learn this part of language? Think about what the time complexity of this approach might be in terms of the # of equivalence classes that are 'learned' – what do you think it is? Why?)

Before proceeding, it is important to recall that not *all* languages can be described by FTNs – not even all human words (language root and affix systems). Of course, FTNs *can* recognize arbitrarily long words like *anti-anti…missile* quite easily – that is, they can generate *infinite languages*, simply by using a loop in their state network. This example also depicts a *nondeterministic* FTN, since in the start state there are *two* possible next states on the transition *anti*:



However, human languages can be more complex than this, and form patterns that are *not* describable by any FTN. Consider Bambarra, an African language spoken in Mali (example

from Culy, 1986). In Bambarra, words can be in the form *Noun+o+Noun*, as in *wulu o wulu* ('whichever dog'), and where there can be any Noun duplicate on the left and the right . Also, we find Noun forms such as *wulu+nyini+la*='dog searcher', which can also be arbitrarily extended to the right (sort of the reverse of 'antimissile', as in:

wulunyinila= 'dog searcher'
wulunyinina+nyini+la = 'one who searches for dog searchers'
wulunyininanyinila+nyini+la = 'one who searches for one who searches for dog searchers'
… (and so on…dogged search, no less)

The *coup de grace* for FTNs comes from combining these ever-longer Nouns with the *Noun o Noun* form, yielding possible Bambarra words such as:

*wulunyininanyinila+**o**+wulunyininanyinila* = 'whichever one who searches for dog searchers'
*wulunyininanyinilanyinila **o** wulunyininanyinilanyinila*,
etc.

Such a language is in the form *wow*, where *w* is a string that is arbitrarily long, a copy on both sides of the *o*. It is easy to see that such languages cannot be recognized by any FTN. We use the pigeonhole principle, and the fact that an FTN has a *finite* number of states (equivalence classes for strings), and so must 'bin' every string into one of a finite number of bins – such strings are intersubstitutable. The intuition is that to recognize such strings, a machine must 'remember' the number of *nyinini* copies the left of the *o*, and then check that the number matches up with the sequence after the *o.* But this must mean that the machine must have an arbitrarily large number of states, one to separately 'remember' each possible such sequence. Since an FTN does not have an arbitrarily large number of states, it must fail on some case where it places two different length copies of *nyinini* in the same equivalence class.

(Sketch. To firm up this argument, assume that we have such an FTN, *FTN* that correctly recognizes this language, call it *L(copy).* Note that the correct strings are in the form $a^n\ o\ a^n$, where a^n denotes the number of copies of *nyini*. This number must match up on the left and the right. Call the language the FTN accepts *L(FTN).* We assume that *L(FTN)=L(copy)*, and derive a contradiction. Since any FTN has a finite number of states, say *m,* and since *n* can be arbitrarily large, by the pigeonhole principle there must exist *k,l,* with $k \neq l$, such that a^k and a^l are in the same equivalence class. But then $a^k\ o\ a^l \in L(FTN)$ *iff* $a^k o\ a^k \in L(FTN)$, a contradiction.)

Putting to one side such examples for the moment, a language's morpheme classes and their possible linear arrangements as an FTN is one kind of knowledge that we must represent for word parsing.[1] But does this make sense from an engineering point of view? Surely we can just store all the legitimate possibilities! Surely yes, given today's cheap memory (or even a person's). But this brute-force approach fails on at least three grounds. First: the sheer number of word forms can boggle even large memory stores. While we have given only the simplest examples above, some human languages (like Inukituit above, or Turkish) have very rich and

[1] You might ask how a child can figure out the morpheme classes just by exposure to their language. An excellent question; you might think about whether children would have to go through a kind of equivalence class computation – and what its problems might be. See C. deMarcken (1995) for one approach.

complex concatenative affix systems (they are called 'agglutinative' languages). For example, it is estimated that the number of Turkish word forms easily exceeds 600 billion. Or similarly, in a language like Inukititut; see the figure below. Worse still, as we saw above, the number of words in a language actually grows without bound, and this is neither uncommon nor unnatural. And impractical: in point of fact, as far as I know nearly every hand-held thesaurus (e.g., "Franklin Ace" incorporates a version of the lookup methods we shall study in this laboratory). Second, simply listing all the words completely misses the open-ended, generative aspect of the system. Listing does not make explicit that 'cooler' and 'coolest' are closely related – in a list, there is no logical implication between such forms. Third: explicit lists don't reflect what people seem to do. Psychological tests indicate that we really do decompose words into their parts as sugested above. The classic experiment, by Berko (1958) is known as the *wug* test. Individuals given pseudowords to recognize react slower when the word form has a seemingly legitmate decomposition, a possible suffix, presumably the effects of lookup and processing – e.g., *juvenate* is rejected more slowly than *pretoire.* (Good experimentalists should note that we've omitted much proper detail here about controls.)

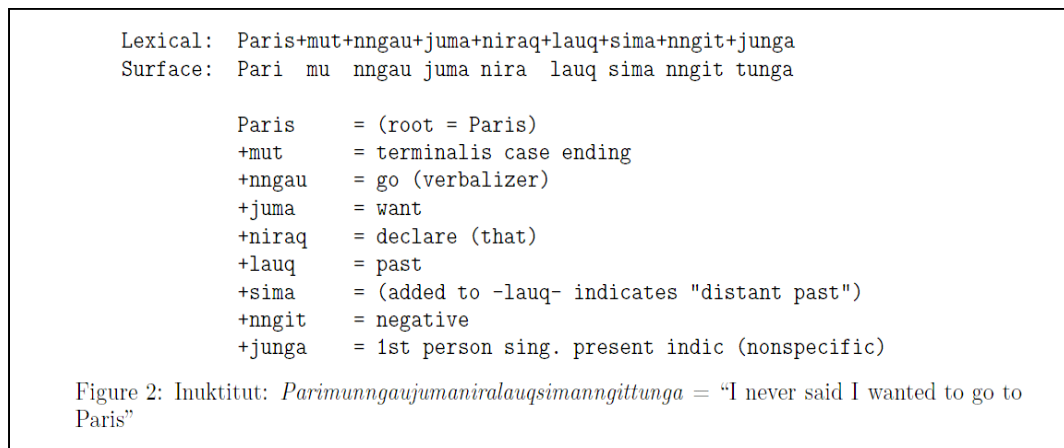
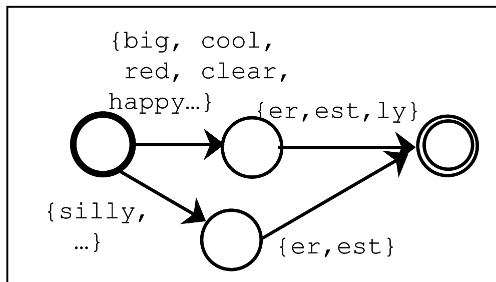


Figure 2: Inuktitut: *Parimunngaujumaniralauqsimanngittunga* = "I never said I wanted to go to Paris"

However, before moving on, we should try our hand at a very small case study, to illustrate how one builds up an FTN for parsing words into root and morpheme classes. Suppose we wanted to build an FTN to parse the following English adjectives into their proper chunks. We'll see how to successively approximate this data. (By the way – how would you *find* this data to begin with?)

- Big, bigger, biggest
- Cool, cooler, coolest, coolly
- Red, redder, reddest
- Clear, clearer, clearest, clearly, unclear, unclearer, unclearest, unclearly
- Happy, happier, happiest, happily
- Unhappy, unhappier, unhappiest, unhappily
- Real, unreal, silly, sillier, silliest

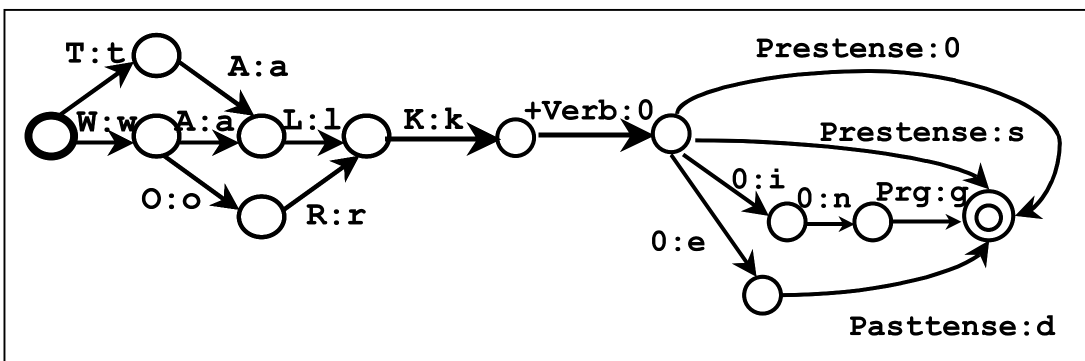
OK, at first glance it seems that there's a class of adjectives like *big, cool, red, clear, silly* that can take the comparative endings, *er, est;* these also take *ly* - no wait, *silly* does not. So we could construct the following FTN:



Now, what about taking care of the prefix *un*? We could simply place this in front of the roots It can appear before *happy,* but there are no such English words as *unsilly,* nor *unred.* You should try to carry out some surgery on our network above to see if you can come up with an FTN that accepts all and only the valid sequences we have given above, and none of the invalid ones (i.e., ones not listed). Note that simply adding 'un' to the front of the network won't work (since that would allow *unsilly*). The important lesson here is that the FTN states are used to distinguish *different* possible morpheme sequences.

So, by modeling possible morpheme strings, are we done? Not quite! In our haste we seem to have forgotten entirely about surface and lexical representations. Recall that if we have an input like 'cats', we really want to pair it with a lexical (underlying) form 'cat + Pl', and vice-versa. So we need some way to have our FTNs cough up output.

This is easy to do. We extend FTNs by adding an *output* for every transition. As the FTN traverses an edge from one state to another, it emits an output symbol. To do this, we must add an <u>*output alphabet*</u> to our definition of a finite-state machine. This yields a <u>*finite-state transducer*</u> (FST): a six-tuple (Q, Σ_1, Σ_2, δ, q_0 F), where δ is now extended from a map $Q \times \Sigma \times Q$ to a map $Q \times \Sigma \times \Sigma \times Q$. In other words: the transition mapping goes from a state and a *pair* of symbols to some possible next states. It's easier to draw than say: we simply put *pairs* of symbols *x:y* on each arc, one denoting the input character, the other the output character. (Another convention that is commonly used is to list one element of the pair *above* the transition arc and the other *below* it.) For example, now we can redesign our root and suffix machine to return the lexical form with features attached:



So as to make the distinction between the two alphabet pairs clear, we have taken the liberty to write the 'surface' symbols in lowercase, and the lexical or underlying symbols beginning in Uppercase. Note that '+Verb' is all one atomic symbol, as is 'Prestense', etc. Obviously, these are arbitrary (but meaningful) labels supplied by the linguist, to be later manipulated by computer.

Now for instance, given a surface form 'talk', we can trace the path through the network following the second symbol of each colon pair, and noting the output at each transition. This will yield the output form 'TALK+VerbPrestense' – i.e., a verb in the present tense. To get this far, our network illustrates one other notational change that is conventional: we have replaced the empty symbol (string) ε with the symbol 0, and so allow a transition from the state following K:k via the pair +Verb:0 - that is, with *nothing* matching in the input string. Empty symbols – 'zeroes' are similarly used in the rest of the suffix, both on the left and the right. As we shall see, this will introduce computational complexity into the resulting system, often in subtle ways. Still, it all works: 'walking' will yield 'WALK+Verb+Prg' – i.e., a verb in the progressive tense.

It is also very <u>important</u> to note that the transducer works <u>both</u> ways – for either the surface or the lexical form can be output. We can we take the lexical form 'WALK+VerbPasttense' and trace this through our machine. That will yield the pairing with the lexical form for the output 'worked'. Pay attention to how the zeroes are consumed here – we could say that the output was 'work0ed' (Remember our notion of a zero morpheme.) In short, for every pair of strings *x,y* over the two alphabets, that is, for two languages, an FST defines a <u>relation</u> (*x,y*) where *x,y* are related just in case *x* can be paired with *y* by running it through an FST. Neither member of the pair has 'priority' – the machine runs both ways. So, we can think of the FST as either 'recognizing' (or parsing) a surface form, recovering its lexical form; or we can think of the FST as 'generating' or producing a surface form from an underlying lexical (dictionary) form. That is very convenient, and is what our computational machine will do.

Because the underlying FST is finite-state, we call this a <u>*regular relation*</u>. Intuitively, we may think of the direction from 'surface' form to dictionary or lexical form as 'recognition' and the direction from lexical to surface form as 'generation', but the machine really is neutral between the two. Since finite-state transducers may have loops, we see that they can relate two infinite (regular/finite-state) languages.

Transducers and the relations they determine have some subtly different properties from the more familiar finite-state networks, and so deserve further mention. In particular, whereas the intersection of two finite-state transition networks is always another finite-state transition network (that is, FTNs are <u>closed</u> under intersection; this simple result is easily verified by considering the intersection machine defined by the cross product of the states of the two initial finite-state machines and a joint transition mapping – see Hopcroft and Ullman, 1979 if you are not sure.) The same is <u>not</u> true for FSTs. To see this, note that we can easily define an FST M_1 that maps $a^n \rightarrow b^n c*$ and vice-versa, and a second FST M_2 that maps $a^n \rightarrow b^* c^n$. The intersection of the two transducers, $M_1 \cap M_2$ is the mapping $a^n \rightarrow b^n c^n$, and this cannot be described by an FTN, by the same argument as in our Bambarra example. This negative result will become important is when we try to define several transduction constraints operating in parallel, so that they all apply simultaneously.

A second major difference between FTNs and FSTs is that while any nondeterministic FTN can be converted into a deterministic FTN that recognizes the same language as before, the same is not true for FTNs. For FSTs, the conversion trick is to see that the transition mapping is defined as $\delta: q \rightarrow 2^Q$. That is, it maps to some subset of next states, the power set of *Q*. We can make this mapping a function, so that each transition on a particular symbol from a state can only have one output, by constructing a new FTN whose states denote possible subsets of the original machine. Intuitively, if a transition on symbol *a* takes a nondeterministic machine to more than one next-state, say state 1 or state 2, then the corresponding deterministic machine can be in a 'superstate' that represents the state of being in either 'state 1 or state 2' – that is, both. So for example, if we take the nondeterministic *antimissile* FTN from above, we see that on encountering *anti* the FTN can go to either state 1 or state 2. One way to write this is to say that the transition mapping takes the machine from state 1 to the union of states 1 and 2, i.e., $1 \rightarrow \{1, 2\}$, that is, after processing *anti* the machine could be in *either* state 1 or state 2. That is certainly an expression of nondeterminism. If the machine is an FTN, then we can convert this into a deterministic machine by creating a new superstate $q' = \{q_1,\} \cup, \{q_2\}$.

However, with FSTs we are not so lucky. There are certain FSTs that are inherently nondeterministic – they cannot be turned into equivalent deterministic machines. Intuitively, since the machine is spitting out symbols as it goes along, it cannot 'take them back' once they have been output. Similarly, if we try the subset construction trick, and the transition to one state requires us to spit out an *a,* and another a *b,* we cannot output *both* at the same time. For example, the following FTN, from Barton, Berwick, and Ristad (1987) cannot be made deterministic:

What is the implication? We don't have the same guarantees.

3. You say potatoe, I saw pohtatoe.

OK. We now have our morpheme transducer. Is there anything else we have to know? Well, yes – there is another complication you may have already noticed. The way a morpheme 'looks' or is spelled out depends on its context. This is called *morphological alternation.* Consider adding the plural morpheme 's' to a root. We get different results, depending on the plural morpheme context:

- We have cat+s→ cats but:
- Fox+s→ foxes (an 'e' added in before the 'x') and
- Quiz+s→ quizzes (two things – the 'z' is doubled, and an 'e' added)

What's going on here? These spelling changes are the written (orthographic) reflexes of sound changes (phonological rules), particular to each language, often occurring at morpheme boundaries (like the *s* that is added here). The upshot is that we can't just break surface into morpheme chunks and look up each chunk individually in a dictionary. Instead we'll have to consider the 'impedance match' between the underlying or lexical form – quiz+Pl – and the surface form we see – quizzes.

We will carry out these spelling change adjustments by means of a second set of finite-state rules, represented as finite-state transducers. We will write one rule for each spelling change, each corresponding to a different transducer. The idea is that each transducer will act as a filter on the set of possible lexical:surface pairs, a tollgate that only admits valid pairings. Then

to make sure all the rules apply, we run them in parallel against the word string we are given. This will allow only *possible* valid spellings (which might include some impossible words, since the rules do not look at any dictionary). As it happens, in English there are only about 5 or so spelling change rules that we need.

Here is how the transducer system works. Let us take a particular example – *fox+s*→ *foxes*. The insertion of an *e* in this context is called epthenthesis. If we study related English examples, we see that the examples break down in the following way:

- *Cat/cats; miss/misses; fox/foxes; buzz/buzzes; church/churches; fish/fishes.*
- From this we can conclude that one can write the following rule as a constraint: If an *e* appears, then it must be in the following context: the immediately preceding character is a morpheme boundary + preceded by a 'sound' like *s, x, z, ch, sh,* and the immediately following context is an *s* followed by a word boundary, #.
- We see that this constraint includes reference to *both* surface context (i.e, *s, x, ch*) and lexical context (i.e., morpheme boundary +). So in fact, we want to write the rule in terms of *both* lexical, surface pairs and specify this via a transducer that can include lexical:surface labels on its transitions. The transducer will be designed so that it accepts *only* the lexical/surface string pairs that obey the constraint. For example, it should rule out the possibility *cat+s/cates* (instead of *cats*).
- To do this, we pair these *surface* forms with their underlying *lexical* counterparts. We use the empty, null character epsilon, denoted 0, to 'pad out' surface and lexical strings, always matching a lexical + with a 0 (this keeps the morpheme boundaries straight and also solves a certain technical problem that we discuss later):

Lexical: c a t + s # f o x + 0 s # b u z z + 0 s # c h u r c h + s f i s h + s
Surface: c a t 0 s # f o x 0 e s # b u z z 0 e s #… etc.

Using the abbreviation Csib for the class of *sibilants* = {*s, x, z*}, we can write the pairing for the *fox, buzz, misses* examples as a linear constraint pattern like the following, where a single form like *Csib* means the lexical:surface pair *Csib:Csib*; the underscore indicates where the pairing on the lefthand side of the arrow must appear:

0:e → Csib:Csib +:0___s:s #:#

We build up our pattern by adding the disjunctions implied by our other examples:

0:e → c:c h:h +:0___s:s #:#
0:e → s:s h:h+:0___s:s #:#

To express the disjunctions more compactly, we shall write | as 'or' in square brackets and also abbreviate *X:X* as *X*, for any character *X* (this pairing comprising an identity relation). Now our pattern is:

0:e → [Csib| sh |ch] +:0___s #

Now let us build the actual FST that will accept exactly this transduction. The intuition is that we want the states of the epenthesis machine to 'remember' the left context, then allow a transition on the pair *e:*0 exactly when it is in the state of having remembered the sequence of character pairs that comprises the left context; then move to a state from the machine will check the right context, again by a sequence of state transitions that admit only the right

context character pairs. We have thus reduced the problem to a familiar one from computer science of string matching, which may be implemented efficiently by any number of means. Let us always denote the start state of the associated FST by a circle labeled 1. Then our intuition means that we should at least have a straight-line accepting path that consists of the following character sequence transitions from state to state:

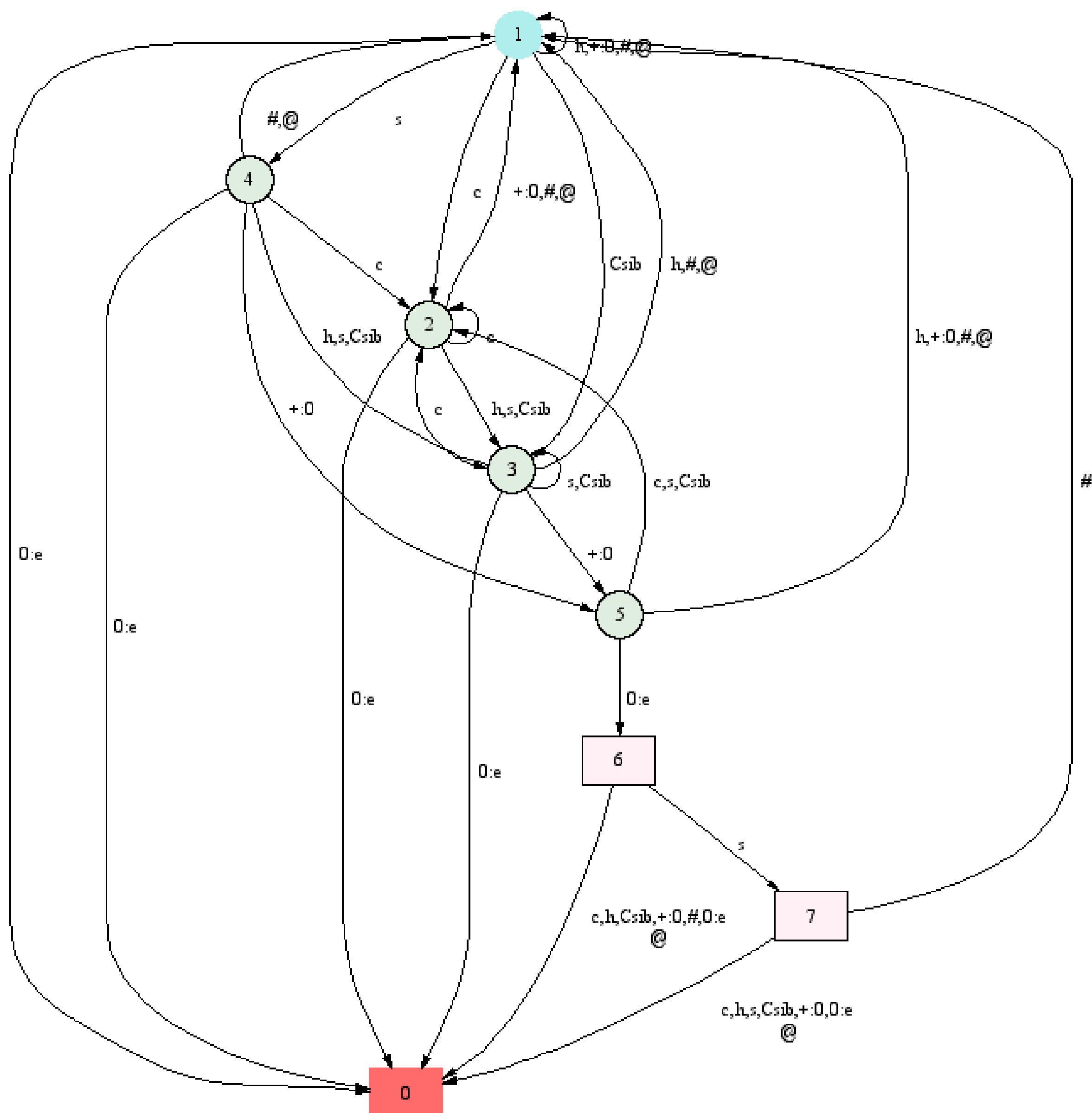
Csib:Csib +:0 *0:e* *s:s* #:#
State 0→ →State 1→ State 2 →State 3→ State 4→ Accept state

Since in general once we have found a valid *0:e* pairing we might find another (consider longer words), we loop back to the Accept state by making State 1 an accepting state. Further, recall that we want to *reject* any string where *0:e* occurs that does not fit this pattern. That means, for example, that at each point along the line after *0:e* is found, then if the correct next character pair is not found, we should reject the string. To this end, we must add transitions from State 3 to a special failure state, call it State 0, on any pair except *s:s*. – that is, *Csib*, *c, h, +:0, 0:e,* and, finally, our abbreviation @:@ which stands for all the other characters except those anywhere in the pattern itself (the set difference between the Alphabet and *Csib, c, h, +:0,* and *0:e*). We do the same for all the states after the *0:e* pair.

Continuing, we must also add new states (renumbering the old ones) to 'remember' the additional left-context patterns *sh, ch,* and any other combinations of these, e.g., *schs,* etc., since in general the strings to check can be arbitrarily long. Finally, we must label the states after the start state, but before the *0:e* to be checked as accepting (final) states. (Why? Because if string ends before we have found an *0:*e, that is OK.) All in all, our final FTN looks like this: (Quick question: what is the purpose of the loop from State 1 to itself labeled with transition @?)

As usual, we say that the machine accepts the string iff it is in a final state when the lexical:surface pairs it is presented are all used up. The final machine has 7 states.
We will also write it in the following tabular format, where the rows are the states (7) and the columns are the character pair labels on the transitions (8 distinct pairs), in the form lexical/surface. The entries in the table are the next states. Accepting states are indicated by colons, and nonfinal states by a period. The zero (0) reject state is given in the table entries, but not otherwise listed. State 1 is the start state.

```
RULE "3 Epenthesis, 0:e => [Csib|ch|sh|] +:0___s [#]" 7 8
   c h s Csib + # 0 @
   c h s Csib 0 # e @
1: 2 1 4 3    1 1 0 1
2: 2 3 3 3    1 1 0 1
3: 2 1 3 3    5 1 0 1
4: 2 3 3 3    5 1 0 1
5: 2 1 2 2    1 1 6 1
6. 0 0 7 0    0 0 0 0
7. 0 0 0 0    0 1 0 0
```

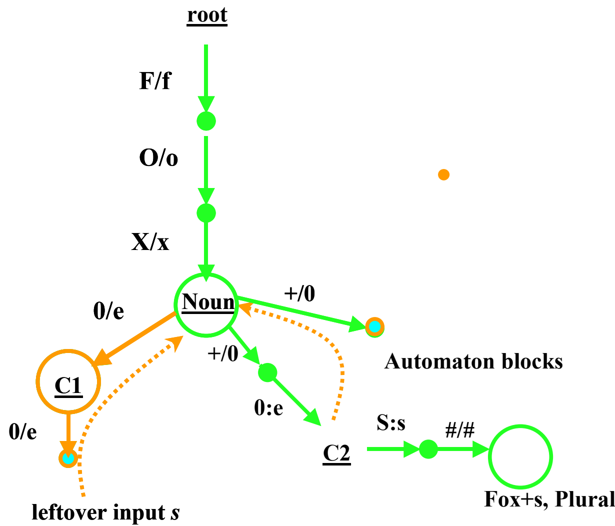


FTN for (most of) an epenthesis rule.

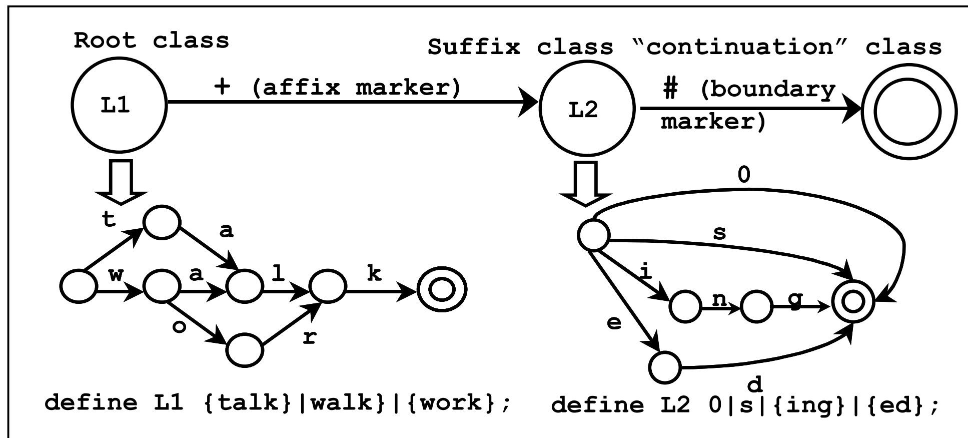
Given our epenthesis machine, we can give it an (arbitrarily long) string and it will accept only those strings that include the *0:e*. You can check this yourself by using the Laboratory implementation of this approach, which is called Kimmo.

We are now in a position to characterize our entire word parser. It consists of two finite state machines operating in tandem: one that checks for the correct morpheme sequences, and one that checks for the correct lexical:surface pairs. We run this together, so that at each point the collective set of transducers and morpheme automaton make transitions based on each character that is seen. In the case of a pair like *fox+s/foxes*, we arrive at a picture like the following. Here, the large circles represent morpheme states, e.g., C1 and C2 are two different <u>states</u> in a morpheme automaton for English, corresponding to two different possible suffix

forms, while the small circles denote the spelling change automata states. We have capitalized the lexical string for clarity. The 'root' stands for the start state of the morpheme tree.



Another view:

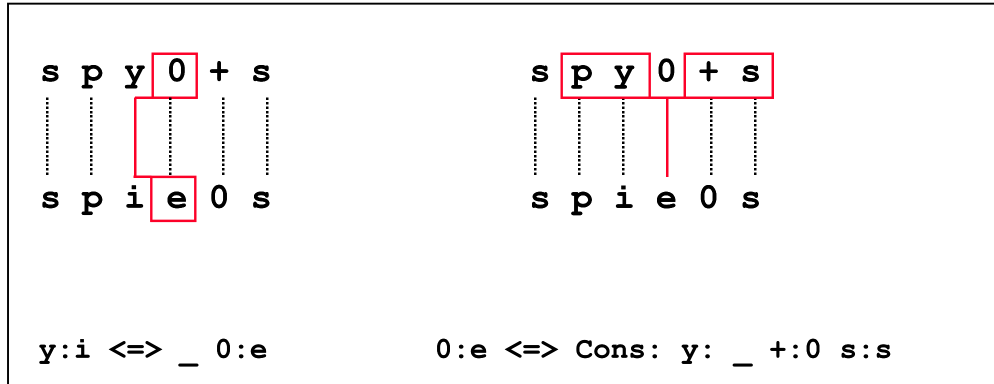


However, we know that a language will, in general, have more than one spelling change rule. In that case, we must supply FTNs for each. In English, for example, there are these five rules that do most of the work:

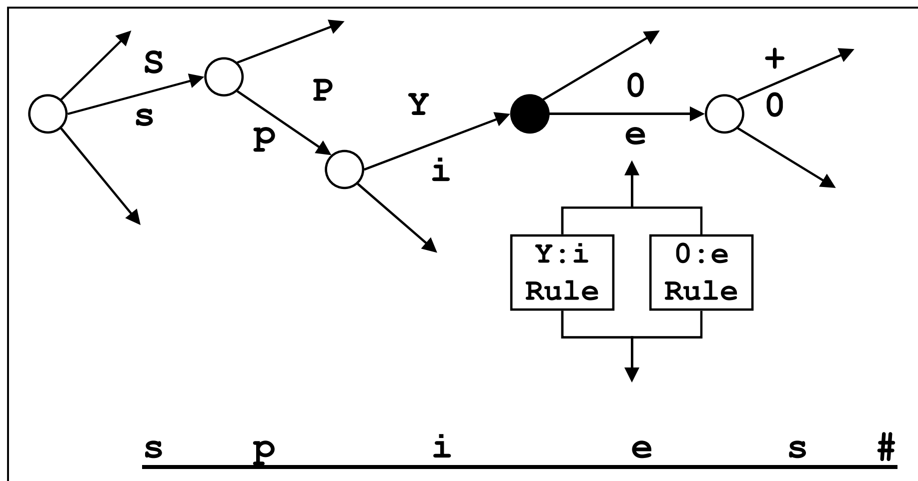
1. Epenthesis (we know about this one)
2. y-i spelling (*happy+ly:happily*)
3. Reduplication, or germination (doubling of consonants, e.g., *big+er bigger*)
4. Elision (erasing an *e*, e.g., *move+ing*:*moving*)
5. i-y spelling (*tie+ing*:*tying*)

Let's see what happens when we have to apply several rules. For instance, consider the lexical:surface pairing *spy+s:spies,* or rather *spy0+s:spie0s.* Here we see that there are <u>two</u> rules that apply: (1) our familiar epenthesis rule (modified to take into account the *y* as left

context); (2) y-i spelling that pairs y:*i* if before a plural ending. Clearly, we must see to it that both rules apply to get the right lexical:surface form. We block off the two contextual conditions in the figure. If we apply the two rules as declarative constraints *in parallel* then the rules will work (we can imagine this as two templates simultaneously imposed on the lexical:surface pair):



With five or six rules, we apply all of these at once. Thus the picture looks like this:



How can we be sure that this will work? While the notion of parallel rule application seems clear enough, this is actually quite a challenge to work out. Laboratory 1a points out that there are certain difficulties that arise when one rule provides the context for another, and asks you to propose a solution.

In the next section of these notes, we shall ask what the computational complexity of this machinery is – what sort of device it characterizes, and whether this is sufficient to naturally describe what we see in natural language. The bottom line is that in the end, while such systems have proved quite useful – indeed, they are the most successful machines we know of for implementing morphological processors to handle a wide variety of languages from

Hungarian to Japanese to Czech to English, they are both too weak and too strong. They are too *weak* in that they have difficulty in representing infixation, common in Semitic languages such as Arabic and Hebrew. This is exactly what one might expect from a system that is based on linear concatenation. In Arabic, vowels are interpolated *inside* of a consonant framework such as *ktb.* Systems like Kimmo are too *strong* in that they can characterize computationally intractable problems that reflect processes never seen in human languages – they are at least NP-hard, if not worse. (If you don't know what these terms mean, we'll explicate them in the next installment.)