KIMMO: A GENERAL MORPHOLOGICAL PROCESSOR

Lauri Karttunen

1. General Features

KIMMO is an Interlisp-10 implementation of Kimmo Koskenniemi's "Two-level" model for morphological analysis and synthesis. The most important feature of the program is that it is language independent. It makes a clear distinction between declarative descriptions of particular languages and the process by which words are generated or recognized using such descriptions. Because of its modular structure, one can use the program for different languages, even concurrently, by simply referring to alternative sets of rules and lexicons. It is possible to make changes in the rules and to edit the lexicon without affecting other parts of the program. The diagram below describes the overall organization of KIMMO.

```
GRAMMAR
    |     |
    |     |
    |     |
    |     |
    |     |
LEXICON
    |     |
    |     |
    |     |
    |     |
    |     |
```

tries............
ANALYZER
    |     |
    |     |
    |     |
    |     |
    |     |
GENERATOR
    |     |
    |     |
    |     |
    |     |
    |     |
try+s V Sg 3rd

Another characteristic feature of the model is reversibility. The same description and lexicon are used both for recognition and for generation. In fact, the analyzer and the synthesizer share the same basic process whose task is to ensure that surface strings get correctly paired with lexical representations. Because of this, the processing time is not significantly affected by the direction. It hardly takes any longer to recognize an inflected form of a word than it does to generate the same form from its lexical representation. This is an interesting feature because people also seem to be able to recognize words about as

Texas Linguistic Forum 22, 1983
fast as they can produce them. Although the program does not attempt to model human performance, it at least has some of the same characteristics.

The reversability of the program stems largely from the fact that the type of description used is not as biased towards generation as morphological and phonological descriptions typically are. Two-level rules do not describe transformations of sequences of segments as rules of generative phonology do. They are simply descriptions of correspondences between lexical and surface forms. In that respect the model is similar to old-fashioned structural phonology although it also differs from it in important ways. Just as in structural phonology, in the Two-level model there are no rule interactions, no relationships like bleeding or feeding that result from rules being applied in a sequence so that later rules apply to the output of earlier ones.

The program is quite compact and efficient in its operation at least in part because it is set up to use only relatively "shallow" morphological descriptions. Aside from theoretical considerations, it remains to be seen whether there actually exist natural languages which could not be adequately described within the limitations of the current model. Because the model nevertheless does allow a wide range of descriptive devices, it is possible to describe languages that are morphologically quite complex, as Koskenniemi's own treatment of Finnish shows. The possible devices include, for example, deletions and insertions of segments in phonologically or morphologically conditioned environments, the use of archiphonemes in the lexicon to represent partially specified segments, the use of morphophonemes to handle suppletive alternation patterns, simultaneous reference to both phonological and lexical environments, optional and obligatory rules, etc.

The most important technical feature of Koskenniemi's and our implementation of the Two-level model is that morphological rules are represented in the processor as automata, more specifically, as finite state transducers. There is a one-to-one correspondence between the rules and the automata. The idea of compiling rules into finite state machines comes originally from a yet unpublished paper by Martin Kay and Ronald Kaplan on word recognition. Conversations with Kay and Kaplan have been the main source of inspiration both for Koskenniemi and for us.

One important consequence of compiling the grammar in this way is that the complexity of the linguistic description of a language
has no significant effect on the speed at which the forms of that language can be recognized or generated. This is due to the fact that finite state machines are very fast to operate because of their simplicity. Thus the number of such machines produced by the conversion and the number of states in each machine has only a minimal effect on processing time. Although Finnish, for example, is morphologically a much more complicated language than English, there is no difference of the same magnitude in the processing times for the two languages. This is a very desirable characteristic for practical purposes. It also has some psycholinguistic interest because of the common sense observation that we talk about "simple" and "complex" languages but not about "fast" and "slow" ones. The speed at which we recognize words does not seem to reflect the complexity of morphological rules that are used to describe their form.

We tried to design KIMMO in such a way that it could be used as a part of some larger system for parsing, semantic interpretation, sentence generation, and the like. Because of that, the output from the recognizer consists of a string of morphological features associated with the word. With our English dictionary and rules, the program works as follows:

Recognizer
Input: tries
Output: try+s V PRES SG 3RD

Here try+s is the corresponding lexical string and the features are picked up from the root and suffix lexicons as the form is processed.

The generator currently takes as its input an actual sequence of morphemes:

Generator
Input: big+er
Output: bigger

It would be a simple matter to change the generator so that it would start from a lexical representation of a word coupled with a list of inflectional features, e.g. "big Compar".

In its current form the program does not distinguish between inflectional and derivational suffixes. All features of a word are simply collected to a single list. This incremental treatment seems appropriate for inflectional markers such as the plural +s in English but it is inadequate for derivational suffixes like the
In words like harden. The program could easily be made more sophisticated in that respect. Derivational suffixes could be viewed as functions that apply to stem features transforming them in appropriate ways. If the program were used to provide input for a syntactic parser, this would be necessary for languages like Japanese and Finnish, which have a large set of productive suffixes for verb derivation.

2. Rule Formalism for Two-level Rules

The sample descriptions that we wrote for English, Rumanian, Japanese, and French were actually first encoded directly as automata and only later written down as rules in Koskenniemi's format. This is because it turned out to be a fairly simple matter to convert informally stated rules of the permitted sort to finite state transducers. By the time we saw Koskenniemi's rule formalism, most of that work had already been completed. The rules that appear in the later sections have been translated from already existing automata although the conversion actually should take place in the other direction. We think that the algorithms that Kay and Kaplan have designed for compiling phonological rules to transducers can easily be applied to Koskenniemi's Two-level rules but we have not yet pursued the matter.

All Two-level rules express correspondences between lexical and surface forms. Typically they also involve a surrounding context which can be specified by referring either to lexical or to surface environment or to some combination thereof. Each such rule is a statement indicating whether the correspondence in question is restricted to, required, or prohibited in the particular environment. The general format of Two-level rules is given below. Here CP refers to a lexical/surface correspondence, LC and RC refer to the left and right context, respectively, and *OP* is a logical operator that indicates what status CP has in the given context.

Two-level rule: CP *OP* LC ___ RC

The most common operator in our rules is (--) which says that the stated correspondence is obligatory in the given environment and possible only there.

It is important to keep in mind that CP, LC, and RC all consist of sequences of pairs. The first member of each pair is a character of the alphabet in which the lexicon is written, the
second member is a character from the surface alphabet. Of course, the two alphabets generally overlap and the surface representation of a word may turn out to be identical to its lexical representation. For example, \((n \cdot m), (p \cdot p), (p \cdot 0)\) are all possible pairs. The symbol \(0\) stands for the null string. Thus \((p \cdot 0)\) is a case where lexical \(p\) gets deleted, i.e. has only the null string as its surface realization. Whenever the two members of a pair are identical, a single character can be used to represent the pair, thus \(p\) could be used as an abbreviation for \((p \cdot p)\).

The fact that rules in Two-level phonology can refer to both lexical and surface context in part makes up for the lack of rule ordering. Any simple interaction between two rules A and B that in standard phonology results from A being ordered before B can be mimicked in Two-level phonology by making A refer to lexical and B to surface environments. In fact the model even allows simultaneous reference to both kinds of context, which is not permitted in standard theory. No systematic comparison of the two frameworks exists so far.

A simple example of a Two-level rule is given below. Here the correspondence part and the right context both consist of a single pair. No left context is specified.

\[(n \cdot m) \leftrightarrow \_ \_ \_ p\]

In words, the rule says that lexical \(n\) always corresponds to surface \(m\) when followed by a lexical and surface \(p\), and only then. The symbol \(\leftrightarrow\) here is equivalent to "if-and-only-if".

Koskenniemi's notation is actually slightly misleading as far as the correspondence part of the rule is concerned. It does not indicate that the above rule focuses on the surface realization of lexical \(n\), not on the possible lexical counterparts of surface \(m\). It specifies the context where lexical \(n\) can and must be realized as \(m\) but does not restrict surface \(m\) in the same way. Under the intended interpretation, the rule allows surface \(m\) in this environment also to be a realization of some other lexical character besides \(n\). Because of this inherent asymmetry in the "correspondence" part of the the rule, it would be better not to use the pair notation there at all. A notation such as

\[n \succ m \leftrightarrow \_ \_ \_ p\]

would be more felicitous. It would also provide a formalism for rules with the opposite directionality \(\ll\) for doing "upside-down
phonology." However, we follow Koskenniemi's convention here.

To make the rules and the corresponding automata more compact Koskenniemi has designed an ingenious system of abbreviatory conventions based on definitions of character classes. For example, \( V \) could be defined as the symbol for the class \( a, e, i, o, u \). The abbreviation \( (V.V) \), however, does not necessarily stand for all possible vowel-vowel pairs. Its interpretation varies from grammar to grammar and from rule to rule. The \( (V.V) \) schema in a rule includes a pair like \( (a.e) \) only if there is some rule in the grammar which explicitly mentions the possibility that a lexical \( a \) might correspond to a surface \( e \). But if \( (V.V) \) and \( (a.e) \) occur in the very same rule, say, as alternative context specifications, then the latter pair is not subsumed by the former because of a principle that gives precedence to specificity over abbreviatory conventions. Thus the interpretation of \( (V.V) \) in a rule varies depending on what other rules say and what the other pairs in the same rule are.

The relative nature of Koskenniemi's abbreviations is important in the case of \( = \) which represents the class of all characters. The ubiquitous \( (=.) \) in the automata definitely does not mean "any character realized as any character." Instead, it corresponds roughly to the phonologist's "elsewhere." It is limited to pairs that are not explicitly mentioned in the same rule/automaton or subsumed under some more specific schema. Furthermore, only pairs that are explicitly sanctioned somewhere are taken into account. If there is no rule involving the pair \( (._u) \), for example, that pair is not represented by \( (=.) \).

Symbols for character classes can be freely mixed with ordinary characters in Two-level rules. For example, a more realistic version of the assimilation rule given earlier would be

\[
(n.m) \quad \leftrightarrow \quad \_ \quad (=.p)
\]

In words: Lexical \( n \) corresponds to surface \( m \) when followed by surface \( p \), and only then. The symbol \( (=.p) \) does not mean that any lexical counterpart for \( p \) is legal. Its interpretation is determined by other rules in the grammar. If no rule ever mentions any other possible lexical counterpart for a surface \( p \), then \( (=.p) \) has exactly the same effect here as \( (p.p) \). An even more general version of the rule is

\[
(n.m) \quad \leftrightarrow \quad \_ \quad (=.b)
\]

where \( b \) stands for the class \( b, p, m \) of bilabial consonants.
In the current version, it is not possible to write rules in terms of distinctive features. That facility could be added fairly easily by expanding the existing conventions for character classes.

3. Rules as Transducers

For the purpose of efficient processing, each rule is represented in the processor as a finite state transducer. The advantages of converting rules to such automata were first pointed out by Martin Kay and Ronald Kaplan. A transducer is in other respects identical to a simple finite state automaton except that the input symbols consist of pairs of characters. Alternatively, a transducer can be thought of as a finite state machine with two scanning heads that move along two parallel tapes:

\[
\begin{array}{c}
\ldots \text{t r y } + \text{s} \ldots \\
\uparrow \\
\text{FSA} \\
\downarrow \\
\ldots \text{t r i e s} \ldots \\
\end{array}
\]

(\(\uparrow\) = current position)

The input to the machine consists of the pair of characters it is currently scanning, \((y, 1)\) in the above example. Besides the two input symbols, the behavior of the machine depends on the state it has reached on the basis of the input it has already read. If the automaton accepts the pair, the two reading heads both move one step to the right and the process continues until the machine either blocks or reaches the end of the two input tapes. In the latter case, if the state it ends up in is one of the states designated as final, the pair of tapes is accepted.

The internal structure of a finite state machine can be represented by a diagram where circles correspond to states and labelled arcs indicate how the machine responds to a particular input. The transducer corresponding to the rule we just discussed (repeated here for reference) looks as follows.
Although the example is very simple, the comparison with the automaton and the rule shows that the conversion from one to the other is not as trivial as one might think. The two halves of the transducer correspond, roughly, to the dual function of the rule. The arcs between 1 and 2 represent the $\rightarrow$ ("only if") side of the rule. Their effect is to allow the $(n.m)$ correspondence only in front of a surface bilabial. The transitions between 1 and 3 encode the $\leftarrow$ ("if") aspect of the rule. They prevent lexical $n$ from being realized as anything but $m$ in this environment.

1 is the initial state; "::" here indicates that it is a final state. 3 is also a final state. The ":" after 2 marks it as a non-final state. If the machine ends up in this state with no more input to process, the pair of input strings has been rejected. A pair of strings is also rejected when the machine is in a state from which it has no transition for the current input. For example, in state 2 the machine blocks if the next surface character is something other than $b$, $p$, or $m$. Note that the machine is in state 2 only immediately after receiving $(n.m)$; any other pair of input characters puts it to state 1 or state 3. The role of state 2 is to ensure that every occurrence of $(n.m)$ is followed by a surface bilabial. The transitions to state 2 represent the correspondence (CP) part of the rule, the transitions away from 2 encode the right context (RC).

State 3 is complementary to 2. The machine goes to 3 whenever a lexical $n$ is not realized as $m$. In 3 it blocks if the next surface character is a bilabial stop. The notation $-B$ refers to the complement of set $B$. Note that the transition away from 3 represents the negative of the right context. Every input that allows the machine to leave 3 would block in 2, and vice versa. The asymmetry in the correspondence part of the rule that was pointed out earlier shows up here very clearly. The role of state 3 is to restrict the surface realization of lexical $n$, not the
lexical counterparts of surface $m$.

The information contained in the above state diagram can also be encoded in tabular form. The table below represents the same transducer.

\[
\begin{array}{c|c|c|c}
\text{n} & \text{n} & \text{=} & \text{=} \\
\text{m} & \text{B} & -B & -B \\
1: & 2 & 3 & 1 \\
2: & 2 & 1 & \\
3: & 3 & 1 & \\
\end{array}
\]

Rows correspond to states here. The columns show the transitions for particular input pairs. In order to make the correspondence between the table and the diagram more transparent, no symbols appear in places that do not represent one of the arcs in the diagram. In later discussions, and in the appendices that show transducers for particular languages, similar tables are presented as completely filled. $0$ is used to indicate that the transducer blocks. Note that the interpretation of all the pairs except $(_n,m)$ is relative, they represent only pairs that are not covered by some more specific pair pertaining to the same state. For example, the pair $(n.n)$ in state 1 moves the automaton to state 3. It matches both $(n.=)$ and $(.=.)$ but the more specific one of the two takes precedence.

Because of this convention, the task of filling out the gaps in such tables is complicated. It is not correct, in general, to replace all blanks with zeros. For example, all the gaps in the top row (state 1) of the above table have to be filled with 1's in order to prevent the machine from blocking on inputs like $(p.p)$. In the partially filled table, the closest match for $(p.p)$ in 1 is $(=.=)$. When all the blanks in the top row are filled, $(p.p)$ becomes an instance of $(.=B)$ because of the specificity principle. Consequently, the transition for $(=.B)$ in state 1 must be the same as that for $(=.=)$. The corresponding completely filled matrix is shown below.

\[
\begin{array}{c|c|c|c|c|c}
\text{n} & \text{n} & \text{=} & \text{=} & \text{=} & \text{=} \\
\text{m} & \text{B} & -B & -B & -B & -B \\
1: & 2 & 1 & 3 & 1 & 1 \\
2: & 2 & 1 & 0 & 0 & 0 \\
3: & 0 & 0 & 0 & 3 & 1 & 0 \\
\end{array}
\]

Following Koskenniemi, in later sections all transducers are represented as completely filled tables. Since the redundant entries in the tables blur the correspondence between transducers and rules, it would be better to use partially filled matrices.
It does not make any difference as far as our Lisp implementation is concerned. In later versions of the system, the conversion from rules to transducers will be done automatically. In that case, there is not even any need to display them directly.

All the transducers discussed here are deterministic, that is, there is never more than one transition available for any combination of states and input pairs.

The recognizer and the generator both use transducers (rules) in exactly the same way. The only significant difference in their operation is that one is driven by a surface string, the other by lexical input. In either case, the controlling program constructs a pair consisting of its current input character and some possible lexical/surface counterpart for it. The transducers receive these pairs as input, it makes no difference whether they are being used for generation or recognition.

4. Ordering of Transducers

4.1. Serial configuration

In their work that assumes a transformational theory of phonology, Kay and Kaplan have shown that one can capture the effects that arise from rule ordering by arranging the corresponding transducers in a cascading sequence. In effect, each automaton can be thought of as a filter that lets through just those pairs of strings that represent adjacent intermediate stages in a derivation. The diagram on the left represents a series of such automata. The lower tape of each machine is the upper tape of the next machine in the sequence.

Since it is possible to construct single transducers that are equivalent to a cascade of separate machines, one can in principle
build just one large finite state automaton that represents an entire morphology consisting of any number of ordered transformational rules. This possibility is suggested by the single large automaton on the right.

Because finite state transducers are inherently non-directional, such machines can be used either for generation or recognition. One big transducer would be more efficient for recognition than a series of machines since only one pair of strings is processed. This takes less time than running several pairs through small transducers because the speed of a transducer is the same regardless of its size. Secondly, if there are no intermediate stages, less time is wasted in pursuing derivations that ultimately fail. For example, in analyzing words like changing the serial process would at some point have to branch into two paths since there are two equally plausible stem candidates: chang and change. The fact that change is an English word while chang is not would not get discovered until the very end of the path, possibly after many rule applications. A recognizer using a single transducer can immediately rule out one of the two alternatives by accessing the lexicon.

Although a single machine that is equivalent to a series of transducers could in principle always be built, it remains yet to be seen whether the idea is feasible for "abstract" descriptions of languages with complicated morphology. The number of automata that need to be combined to a single machine depends on the number of rules that are required to describe the language. It may turn out in some cases that the single transducer that would be required to represent the cascade is just too large to be of practical use. The size of the lettering in the picture is intended to give some indication of the relative sizes of the single machine vs. the equivalent cascade of rule-by-rule transducers.

4.2. Parallel configuration

Koskenniemi's Two-level model is essentially a simplified version of the Kay-Kaplan idea. It is based on the assumption that rule ordering and the interactions between rules that result from it are not essential for morphological description. If that restriction on the descriptive framework is adopted, there is no need to arrange the transducers in a cascading series. That is required only if it is important for phonological derivations to have intermediate stages. In a Two-level system transducers
operate in parallel. In linguistic terms, this corresponds to all rules being applied simultaneously. The arrangement looks as follows

```
... t r y + s ... Lexical string
```

```
FSA 1 FSA 2 FSA 3 ... FSA n
```

```
... t r i e s ... Surface string
```

All of the transducers operate independently but they share the same two scanning heads. Thus they receive the same pair of input characters at the same time. If any one of them blocks, then the pair of strings is rejected. It is accepted only if the scanning heads reach the end of the strings and all of the automata are in a final state. Since the two heads always move together one character at a time, both strings must in principle consist of the same number of characters to be accepted. However, since there is a special character, 0, representing the null string, the surface string may actually be shorter.

Just as in the case of serially linked transducers, it is possible to merge any number of parallel transducers into a single equivalent machine. As far as efficiency is concerned, the advantage resulting from such a merge is not as important in the Two-level model as it is in the Kay-Kaplan system. Under Koskenniemi's approach, only a single pair of strings needs to be scanned and lexical lookup can proceed in parallel with the rule application regardless of the number of transducers that are involved. The only improvement in performance comes from having just one rather than many automata to operate.

A single transducer resulting from such a merge typically has many more states than the automata it replaces. If \( k_1, k_2, \ldots, k_n \) represent the number of states in the original transducers, the corresponding single machine in principle might need \( k_1 \times k_2 \times \cdots \times k_n \) states. In the tests we ran using our transducer-merging program, we found that the actual numbers are much smaller. For example, a single transducer corresponding to six rules in our description for English only has 109 states out of 266,112 possible ones. However, the single machine corresponding to
Koskenniemi's 21 rules for Finnish would have thousands of states. Because the number of transducers has a very small effect on the speed of recognition and generation in Koskenniemi's model, it may be best in such cases to not to carry out the merge.

5. Lexicon

In the current version of KIMMO, every lexical item has an entry that consists of a string of features expressing the syntactic or semantic properties of the morpheme and a symbol for its continuation class. The latter is used by the recognizer to determine what sorts of things can follow. There can be any number of lexicons depending on the number of distinctions that are needed to encode selectional restrictions.

In order to make the accessing of lexical information more efficient, every lexicon is internally represented in the form of a tree whose branches spell out the morphemes. For example, a lexicon for English containing the words car, card, cat and cup has the following structure:

```
     r—d
    /   \
   c—a
  /   \  
 t    u—p
```

Lexical entries are associated with the last character of each word in the tree. The c—a—r—d branch, for example, has the entry for car at the r and the entry for card at the d. There are no entries associated with c and a because c and ca are not words. In the current version of KIMMO, lexical entries are pairs consisting of a continuation class and a string of features. The entry for car in our present lexicon is simply

```
/N ""
```

The symbol /N is a name of a continuation class that includes the genitive and plural markers. It is used by the recognizer to determine what lexicon to go to next when it has discovered that the initial part of some input string matches car in the stem lexicon. The empty string "" is used here as a placeholder for whatever syntactic and semantic features one wishes to associate with the word. Without such distinguishing features, the entries for car and card look alike.

A more interesting case is the plural marker +s which has the
following entry in the N lexicon

C2 "N PL"

C2 is the lexicon containing the plural genitive marker, the string "N PL" is assigned to all forms containing this morpheme.

In the course of trying to find a match for a surface string in a particular branch of some lexicon, the recognizer has to consider all the possible surface realizations of the first lexical character. If one of them matches the first character of the string, the same process continues with the second character of the branch. At every point, each potential lexical-surface correspondence also has to be accepted by the automata representing phonological and morphological rules. When an entry is found, the process continues in the lexicons indicated by the continuation class of the entry. For example, in producing the response "car+s' N PL GEN" for the input string cars', the recognizer picks up pieces of the word in three different lexicons:

| Substring: | car +s ' |
| Lexicon:   | Root /N C2 |
| Cont. class: | /N C2 # |
| Features: | "" "N PL" "GEN" |

It is a common technique to represent lexicons as letter trees because it minimizes the time spent on searching for the right entry. The recognizer only makes a single left-to-right pass through the string as it homes in on its target in the lexicon. The novel aspect of this procedure in the Two-level model is that the application of all phonological and morphological rules takes place as part of the lexical lookup itself. Because there are no intermediate stages, the lexicon in effect acts as a filter that eliminates unproductive derivational paths. Since there are no entries for chang or ringe in a lexicon for English, the recognizer never attempts to analyze changing as chang+ing or ringing as ringe+ing.

6. Morphosyntax

The system assumes that stems and suffixes are grouped into lexicons on the basis of their syntactic properties. For example, if forms of nouns in some language conform to the following rule there would be a separate lexicon for stem, case, plural, and
singular morphemes. The recognizer would start in the stem lexicon and continue until it had found an analysis that matches the above rules.

Noun --> Stem Case Number
Number --> Singular
Number --> Plural

In the current version of KIMMO, there unfortunately is no facility for expressing such rules directly. Instead, every item in every lexicon carries a marker that indicates what group of lexicons the next morpheme in the word could come from. Continuation classes are defined in the beginning of the lexicon. In the simplest possible case, there is just one lexicon in each continuation class. For example, if all noun stems occur with all case suffixes, as the above phrase structure rule dictates, then each noun in the lexicon could have Case as its continuation class.

In natural languages, the situation is often complicated by the fact that morphemes have idiosyncratic selectional restrictions that are not predictable from their form or syntactic class. The continuation class facility in KIMMO can deal with some but not with all complications of that sort. The limits can easily be seen by elaborating our hypothetical example. We start with an easy case and then make it harder.

Assume that there are two kinds of noun stems, N1 and N2, which select partially overlapping classes of case suffixes. Instead of a single case lexicon there would then be three, say C1, C2, and C3. These in turn could be grouped into continuation classes using the definitions like

N1 = C1 C2
N2 = C1 C3

Each noun in the stem lexicon would then be marked either for N1 or for N2 indicating what type of case suffixes it goes with. Each case suffix in C1, C2, and C3 would have Number as its continuation class. This would stand for the Plural and Singular lexicons, as shown by the definition below.

Number = Plural Singular

In the Plural and Singular lexicon, all the entries would have the end-of-word symbol, $\hat{\theta}$, in the place of a continuation class.

This is a convenient system for many purposes. For example, it
would be easy to describe the declension of nouns in languages like Latin. However, it is somewhat ad hoc and not general enough for other applications. Implicit in such a description is the idea that the morphosyntax of a language can be represented as a transition network. Each continuation class, in effect, stands for a state in such a network and the lexicons represent classes of transitions to new states. In that format, the example given above would look as follows:

![Diagram showing a transition network with nodes for 'Begin', 'Stem', 'N1', 'C1', 'N2', 'C2', 'Number', 'Sing', 'Plural', 'End'.]

The same information could also be represented in the form of context free phrase structure rules.

In this representation it is easy to see the limitations of the system. The most obvious defect is that it is not possible to express discontinuous dependencies without duplicating all the intervening transitions. We can see it by making the example just a little bit more complicated.

Suppose that the choice of the plural suffix, in the situation we have sketched, turned out to depend on some morphological property of the stem. For example, assume that there are two classes of stems, A stems and B stems that occur with different plural markers. Instead of a single plural suffix lexicon there would be have to be two, say, PluralA and PluralB. Since the information about the stem class has to be carried in the states of the network, we would need to split N1 and N2 to N1A, N1B, N2A, and N2B. Similarly, the state Number would have to be replaced by NumberA and NumberB with transitions through PluralA and PluralB respectively.

The most unpleasant consequence, however, is that the intervening case lexicons would have to be duplicated. Instead of C1, there would be C1A and C1B whose entries would otherwise be identical except that the entries in C1A would point to NumberA and their twins in C1B to NumberB. Since discontinuous dependencies of this sort are quite common, we regard this state of affairs as a serious defect in the current model.
Koskenniemi's own description of Finnish in fact contains several cases of just this sort of duplication.

The only other solution in the current version of KIMMO would be to flag the two kinds of stems and plurals with matching diacritic symbols, say $ for A and # for B, both realized as 0. An automaton would be set up to block pairs of strings with mismatches like

\[ \text{[A-stem]} \ldots \, \$ \quad \ldots \, \# \quad \ldots \, \text{[B-suffix]} \]
\[ \ldots \, 0 \quad \ldots \, \ldots \, 0 \quad \ldots \]

While this is formally similar to the treatment that one needs to give for phenomena such as vowel harmony, we do not regard it as a good solution to the present problem. Note that we have assumed here that the relevant distinctions, 1 vs. 2 and A vs. B, are not definable in phonological terms. Hence the use of special characters as a means to encode selectional restrictions seems unsatisfactory.

We expect that future versions of KIMMO will be improved in this respect.

7. Generation and Recognition Procedure

A sketch of the generation and recognition process is given below. One important preliminary step that has to be carried out in advance is to make a list of all correspondences that actually appear in the rules. The assumption is that a lexical character LexCha can be realized as SurfCha only if there is some rule that explicitly mentions this pair. The list of "Feasible pairs" contains all such character correspondencies. There is currently no convention for accepting any correspondencies without a rule, not even pairs like (a, a). Consequently, all the descriptions in later section contain a special transducer that corresponds to the conjunction of all rules of the form

\[ (\text{OrdCha}.\text{OrdCha}) \leftrightarrow \text{___} \]

where OrdCha is any "ordinary" lexical character. Since no context is given, the effect of these rules is simply to allow these correspondencies where they are not prohibited by some other rule.

The data structures and procedures for the generator and the recognizer are described below.
Data Structures

Feasible pairs: List of possible correspondencies
Automata: Set of finite state transducers
Configuration: List of automata states
Initial configuration: List of 1's
   Root: Initial continuation class
Continuation class: List of lexicons
Lexicon: Letter tree
Entry: Continuation class and List of features
Generator task: Result, Configuration, and Remainder
Recognizer task: Result, Configuration, Remainder, Lexicon, and Features.

Generator

Initial step
Construct a task with Result = null, Configuration = Initial configuration, Remainder = Input string and execute it.

Basic procedure
In executing a generator task, consider all feasible pairs (LexCha, SurfCha) where LexCha is the first character of Remainder. If MoveAutomata succeeds with that pair and Configuration, construct a new task where Result = Result with SurfCha appended, Configuration = new configuration returned by MoveAutomata, Remainder = tail of Remainder.

Final step
If Remainder is null and FinalConfiguration succeeds, return Result.

Recognizer

Initial step
For each Root lexicon, construct a task with Result = null, Remainder = Input string, Configuration = Initial configuration, Lexicon = that lexicon, Features = null and execute it.

Basic process
In executing a Recognizer task proceed as follows.

If Lexicon starts with an entry, construct new tasks for each lexicon in its continuation class with current Remainder, Configuration, and Result where Lexicon = the new lexicon and Features = Features with the features of the entry appended to it.
Consider all feasible pairs (LexCha.0) where LexCha is the head of a branch in Lexicon. If MoveAutomata succeeds with that pair and Configuration, construct a new task with current Remainder and Features where Result = Result with LexCha appended, Lexicon = that branch, and Configuration = the new configuration returned by MoveAutomata.

Consider all feasible pairs (LexCha.SurfCha) where LexCha is the head of a branch in Lexicon and SurfCha is the first character of Remainder. If MoveAutomata succeeds with that pair and Configuration, construct a new task with current Features where Result = Result with LexCha appended, Lexicon = that branch, Remainder = the tail of Remainder, and Configuration = the new configuration returned by MoveAutomata.

Final step
If there are no lexicons in the continuation class of an entry, then if Remainder is null and FinalConfiguration succeeds, return Result and Features with the features of that entry appended.

Auxiliary functions

MoveAutomata
Input: a pair of characters and automata configuration.
Operation: constructs a new configuration by using the input pair to move each automaton from the state it has in the input configuration to a new state, fails if any automaton blocks.
Output: a new configuration or failure.

FinalConfiguration:
Input: a configuration
Operation: for each automaton, checks whether the state indicated by the input configuration is a final state, fails if some automaton is in a non-final state.
Output: success or failure.

The above description focuses on the procedures themselves without any of the details of Koskenniemi's Pascal and our Lisp implementation. The recognition procedure is more complicated here than generation for two reasons. First of all, it uses the lexicon as a guide in deciding what pairs to consider. Secondly, it takes into account the possibility that there are null
characters (deletions) that do not show up in the input string.

It would be a simple change to make the recognizer work without a stem lexicon. In that case it would construct all morphologically possible lexical stems that correspond to the input string, assign them to all possible syntactic categories and proceed to look for appropriate suffixes. That option would be useful for many applications. Human speakers obviously proceed in much of the same manner when presented with an unknown word in their language.

Note that the above description says nothing about the order in which the generator and recognizer should carry out the tasks they create for themselves. Koskeniemi's Pascal implementation follows a "breadth first" schedule, it completes every task before it takes up the next assignment from a queue. Our Lisp program operates "depth first", it goes to work on subordinate tasks as soon as they arise. The choice of a scheduling principle is independent of other features of the program. It has no effect on the final result or the amount of work that has to be performed.

8. Merging of transducers

This section describes a procedure for merging several transducers into a single equivalent machine. Linguistically this is equivalent to collapsing rules except that there is no requirement that the transducers be in some way conceptually related. The one-to-one correspondence between rules and automata is lost when transducers are merged.

A configuration of transducers is a list showing what state each automaton is in. When two or more automata are merged into a single equivalent machine, each state of the resulting new machine corresponds to a possible configuration of the source automata. The following table expresses that relationship.

<table>
<thead>
<tr>
<th>States of the new automaton</th>
<th>FSA1</th>
<th>FSA2</th>
<th>FSA3</th>
<th>...</th>
<th>FSAn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In principle the number of states in the new automaton could be as large as the product of the numbers of states in the source automata. However, typically it is much smaller because some configurations never arise due to the properties of the source automata. For example, if FSA2 is in state 7 only after receiving the pair (=.V) and FSA3 always moves to state 5 on that same input, then every configuration which has 7 for FSA2 must have 5 for FSA3 and not some other state.

The merging procedure is described below.

Data Structures

| Transducer: Name, a list of states |
| State: Finality, a list of transitions |
| Transition: InputPair, State |
| ConfigList: List of configurations |

Merging procedure

Initial step
Put the initial configuration on ConfigList.

Basic process
For every configuration on ConfigList, construct a state with ConstructState and put it on StateList.

Final step
Select a name and return the automaton consisting of that name and StateList.

Auxiliary procedures

ConstructState
Input: a configuration.
Operation: make a transition for every feasible pair (LexCha.SurfCha) with MakeTransition and determine the finality of the state using FinalConfiguration.
Output: finality and the list of transitions

MakeTransition
Input: a character pair and configuration.
Operation: if MoveAutomata succeeds with that pair and configuration, add the resulting new configuration to ConfigList unless it is already there.
Output: input pair and either zero or the ordinal number indicating the position of the new configuration on ConfigList.
Initially, ConfigList contains just the initial configuration of the source automata, namely, a list of 1's. When new configurations emerge in the course of MakeTransition, they are added to ConfigList to wait their turn to be processed. Since every feasible pair of input characters is tried out with every configuration that arises, all possible configurations are eventually computed and the corresponding states are added to the resulting single transducer.

If it turns out to be impractical to merge the source automata into a single transducer, the same procedure can be used to merge them selectively to yield whatever turns out to be the optimal space/time tradeoff.

9. Conclusion

We think that the KIMMO system already has enough good features to make it useful for many applications. We would like to continue our collaboration with Kimmo Koskenniemi to make further improvements. More suggestions on what could be done are presented in later sections. The most valuable addition would be to incorporate in it a compiler of the kind that Kay and Kaplan have built for their system. In that way, the conversion of rules to finite state transducers could be done automatically instead of by hand. Many aspects of KIMMO are also quite interesting from a linguistic point of view. It would be important to know what the limitations of two-level phonology are and whether equally efficient implementations can be built for theories that have a more abundant selection of descriptive devices.