# A Compiler for Two-level Phonological Rules

Karttunen, Lauri

1987-06-27

Karttunen, L, Koskenniemi, K & Kaplan, R 1987, A Compiler for Two-level Phonological Rules. in M Dalrymple, R Kaplan, L Karttunen, K Koskenniemi, S Shaio & M Wescoat (eds), Tools for Morphological Analysis. Stanford University, Center for the Study of Language and Information, Stanford, California, pp. 1-51.

http://hdl.handle.net/10138/42174

submittedVersion

# A Compiler for Two-level Phonological Rules

Lauri Karttunen, Kimmo Koskenniemi,
Ronald M. Kaplan

June 25, 1987

Xerox Palo Alto Reseach Center
Center for the Study of Language and Information
Stanford University

## Contents

# A Compiler for Two-level Phonological Rules

Lauri Karttunen, Kimmo Koskenniemi,
Ronald M. Kaplan *

## 1. Introduction

This paper describes a system for compiling two-level phonological or orthographical rules into finite-state transducers. The purpose of this system, called TWOL, is to aid the user in developing a set of such rules for morphological generation and recognition. The current version of TWOL (pronounced "tool") is written in Interlisp-D for Xerox 1100-series workstations ("D-machines"); it will eventually be converted to Xerox Common Lisp when this becomes generally available. The compiler accepts as input a set of rules in the formalism of Kimmo Koskenniemi 1983 and compiles them to finite-state transducers in part using techniques devised by Kaplan and Kay 1985 for the compilation of ordered rewriting rules. The transducers in turn can be used by a morphological analyzer whose task is to associate inflected and derived forms of words and morphemes with their lexical representations either for recognition or generation. One such system is DKIMMO—see Part II of this report—but the output of the compiler is also compatible with other two-level analyzers such as Koskenniemi's Pascal implementation.

The TWOL compiler consists of three parts: (1) the FSM package that performs operations on finite state machines, (2) the basic compiler, and (3) a user interface. Part (1) was written by Ronald Kaplan. The FSM package is owned and copyrighted by the Xerox Corporation. Part (2) was written by Kimmo Koskenniemi at the University of Helsinki and the code was modified and augmented by Lauri Karttunen at SRI International who also wrote most of part (3). This system will be made available to non-profit institutions and individual researchers under a licensing agreement the details of which remain yet to be worked out (June 1987).

We begin this report with instructions on how to use the compiler. Section 2 is written for a novice user who just wants to try out the compiler on a sample grammar. Section 3 gives information about the format of a grammar file for a linguist who intends to write and compile his own set of rules. It contains simple examples of compiler declarations and a few sample rules. Section 4 is devoted to a more detailed presentation of the rule formalism. Finally, section 5 discusses the procedure by which the rules are compiled to finite-state transducers.

## 2. Interface

In this section we give a quick tour of the user interface of the compiler. The purpose is to illustrate, by way of simple examples, what TWOL can do and how it is used. We assume that the reader is already familiar with the basic features of D-machines: windows, menus, the mouse, and the like.

### 2.1 Getting Started

If you have the TWOL system on a floppy disk, you can use it on a D-machine that runs the Koto-release of Interlisp-D. First copy the contents of the floppy to a directory, connect to it—at CSLI, just can just type (CONN {HEINLEIN:}<KIMMO>)—then load the compiler with the command

```
(LOAD 'LOADTWOL)
```

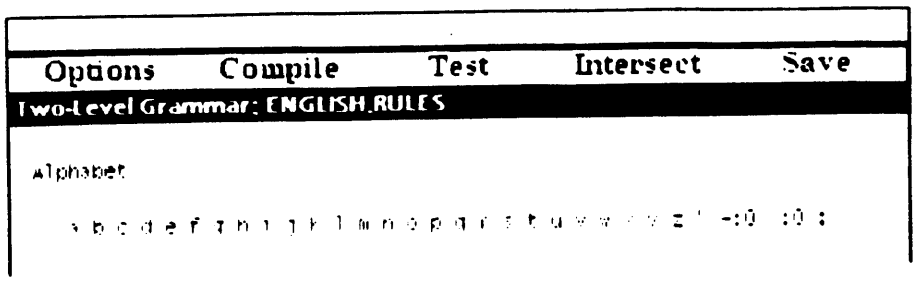
After the files have been loaded, you can start by selecting the command *TwolEdit* from the background menu. As the edit window comes up, it prompts you for a file name. If you just hit the carriage return, you get the section headers for a new rule file. The first time around, we recommend that you type

```
ENGLISH.RULES
```

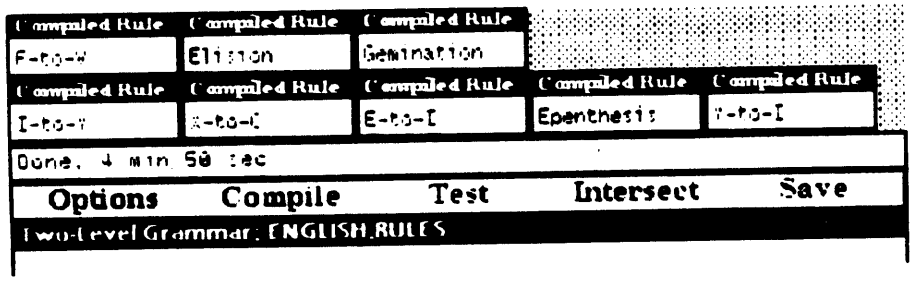
This file contains eight spelling rules for English.

## 2.2 Windows and Menus

At this point, the top of the edit window looks like this:




The top line of the window is a menu with five items: *Options* (section 2.7), *Compile*, *Test* (2.3), *Intersect* (2.5), and *Save* (2.6). The first thing to do is to compile the rules in the file you just loaded. Mousing the command *Compile* brings up a submenu with two options: *Selected rule*, *All rules*; choose the latter one. The compilation of the eight rules in the file takes 6-8 minutes on a Xerox 1108 Dandelion. As the compiler is working, it prints messages to the status line above the menus. For each compiled rule, the compiler creates a small menu window labeled with the name of the rule. By the time the compiler has finished compiling ENGLISH.RULES, eight small menu windows have appeared on the top of the edit window:




You can get information about a rule and the corresponding transducer by clicking the left button in the rule's window. For example, if we click the window labeled "I-to-Y", the following menu pops up:


The commands *Describe*, *Display*, *Test*, *Recompile*, and *Delete* perform the named action on the compiled version of the rule. *Describe* lists the number of the states in the transducers and shows how the rule partitions the alphabet into equivalence classes. (Only the first symbol of each equivalence class appears in the actual transducer.) *Display* prints out the transducer in a tabular form. *Delete* deletes the rule window and the transducer associated with it but it does not affect the source from which it was compiled.

The "I-to-Y" rule itself, the source of the compiled rule, can be found in the edit window:

```
"I-to-Y"
 i:y <=> _ e: -: i ;
```

This is the rule that makes a lexical i to be realized as y in words like *dying* (*die-ing*). More examples of simple two-level rules are given in section 3.5 and section 4.1 discusses the formalism in detail. For now, let us just take note of the fact that, by selecting the *Display* command in the rule window, you can see the corresponding finite state transducer produced by the compiler. Section 5 discusses the methods by which the compiler turns rules to transducers.

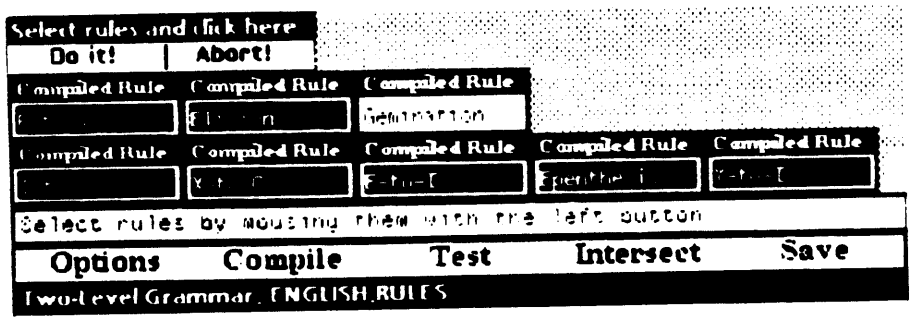
```
I-to-Y
   a e i i:y -:b
0: 0 0 2  1   0
1.   3
2: 0 4 2  1   0
3.              5
4: 0 0 2  1   6
5.     2
6: 0 0    1   0

Equivalence classes:
```

```
(a b c d f g h j k l m n o p q r s t u v w x y z '
 f:v x:c y:i #:0 ':0) (e e:i e:0) (i) (i:y)
(-:b -:d -:e -:f -:g -:l -:m -:n -:p -:r -:s -:t -:0)
```

Each horizontal row represents a numbered state in the transducer; the first column lists the states. State 0 is the initial state, final states are marked with a colon (:), non-final states with a period (.). The other columns are transition functions for a class of input symbols. The symbol on the top of a column is the first symbol in its equivalence class; the other members of the class are listed underneath the transition table. The numbers in the table show for each state (row) and input (column) what state the transducer moves to. For example, the last column, labeled -:b, enumerates the transitions for -:b (morpheme boundary - realized as b) and other surface realizations of -. The column shows that on such input the transducer moves to state 0 from states 0, 2, and 6, from state 3 to 5, and from 4 to 6. The two gaps in the column show that in states 1 and 5, the transducer does not accept any symbol in the -:b class.

## 2.3 Testing Rules

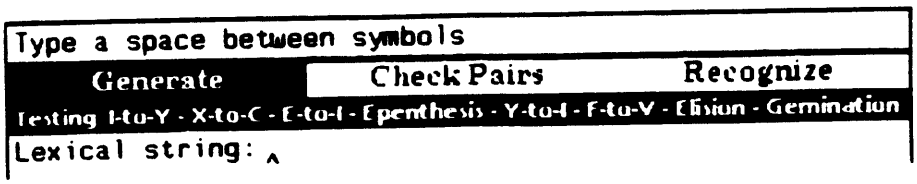
The command *Test* starts a test process in a separate window. You can test rules either individually or in groups; If you click at *Test* in the main command menu, the system prompts you for a selection. To select a rule for testing, mouse its window with the left button. The selected rules show up in inverse video:




To cancel a selection, mouse the rule window a second time. We suggest that, on the first go, you select all the rules. Now click the the *Do it!* command in the Selection window to initiate a test process for the

selected rules. (If you just want to test a single rule by itself, you can also initiate the process by selecting the *Test* command in the pop-up menu of the rule's window.)

The test process opens a new window:




Testing can be done in three modes: *Generate*, *Check Pairs* and *Recognize*. To switch modes, just mouse the mode word on the top of the test window. The current selection (initally *Generate*) is shown in reverse video.

In the *Generate* mode, the system expects you to type lexical strings like

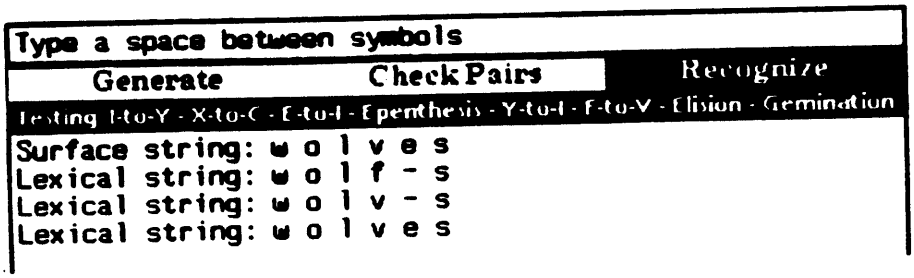
```
w o l f - s
```

and returns with the corresponding surface forms or forms:

```
w o l v e s
```

Please note that the current version of the rule tester expects you to type a space between input symbols because it has not yet been equipped to parse words into a sequences of alphabetic symbols. (A word like *quick* could mean [q u i c k], [qu i c k], [q u i ck], or [qu i ck] depending on the alphabet. It could even have more than one parse because the components of a digraph may also appear in the alphabet separately.)

In the *Recognize* mode, we go from a surface form to a set of possible lexical forms:



In this mode, the rule tester typically "overrecognizes" because it has no access to a lexicon. The rules in this sample grammar do not exclude *wolves* and *wolv* from the class of valid lexical strings. If the lexical symbol is realized as an empty string, the recognizer expects to find a 0 (zero) in its place. For example, to test the recognition of a form like *roof-s*, you should type

```
r o o f 0 s
```

because the rule tester works strictly in term of character pairs. This limitation helps to reduce overrecognition. (The generator is more clever in that it omits 0's when it prints out the results.)

The *Check Pairs* mode expects you to type either single characters or character pairs separated with a colon. For example:

```
w o l f:v -:e s
```

An atomic character here means that the lexical character is the same as its surface realization; for example, `w` means 'lexical `w` realized as surface `w`'. Correspondingly, `f:v` means 'lexical `f` realized as surface `v`' and `-:e` is 'the morpheme boundary - realized as a surface `e`'.

In the *Check Pairs* mode the transducers are operated in succession rather than in parallel and each transducer reports separately whether it accepts the offered pairings. In this case we get back:

```
I-to-Y: accepted
Y-to-I: accepted
X-to-C: accepted
Epenthesis: accepted
F-to-V: accepted
Elision: accepted
Gemination: accepted
```

If you type

```
w o l f:v -:0 s
```

all the other transducers accept the input, but the Epenthesis rule says

```
Epenthesis: FAILED in state 6: s #:0
```

The part after the second colon indicates that all of the string was processed except for the last s—the word boundary marker (#:0) is inserted automatically by the system to the beginning and to the end of your input string. By failing at this point, the Epenthesis rule is telling you that the morpheme boundary (-) cannot be realized as the empty string in this environment.

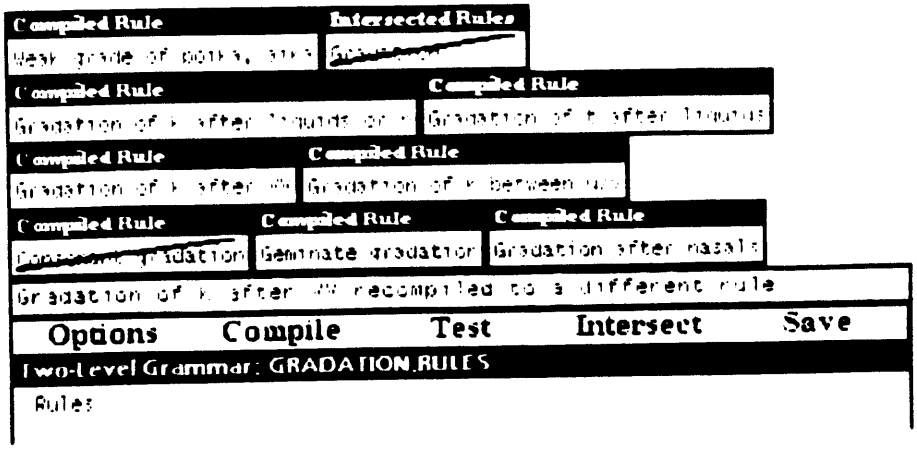
## 2.4 Editing Rules

After you have compiled all the rules once, it is possible to modify them, compile the modified rule or rules separately, and test them again. You can also add new rules and compile them without recompiling any others. This works as long as the alphabet remains the same and you do not introduce any `lex:surf` pairs that the compiler has not already encountered elsewhere. If the alphabet grows or if a new way to realize some lexical character is introduced, all the rules must be recompiled. (The system keeps track of this and tells you when it happens.)

There are two ways to recompile an old (possibly modified) rule. You can pick the option *Recompile* from the appropriate rule window or you can select the name of the rule with the mouse and use the option *Selected rule* in the *Compile* menu. These do the same thing for rules that have already been compiled once but the latter option also can be used to compile new rules that you have just typed in. After recompiling a rule, the compiler checks whether the resulting transducer is equivalent to the result of the previous compilation and tells you whether the new version of the rule is really different or equivalent to the old one.

If the recompilation of a rule produces an automaton that is different from the previous version, it may affect other compilation results. For example, if the rule in question is part of an intersection (section 2.5), the intersection should also be recompiled to bring it up-to-date. Changes in one rule may also affect the compiled versions of other rules,

if the compiler is running in a mode in which it avoids certain types of rule conflicts (section 2.8). The system keeps track of such dependencies. If the recompilation of a rule makes an intersection or some other compiled rule obsolete, the windows of the affected rules and intersection results are crossed over:




The lines across the windows labeled "Consonant gradation" and the "Gradation" indicate that their contents are now obsolete because of a change in the "Gradation of k after VV" rule. The user can update a window by selecting the command *Recompile* from the window's menu. If the old version of a recompiled rule was being tested, the test window is automatically re-initialized with the new version of the rule.

## 2.5 Intersecting Rules

The *Intersect* command prompts you for a selection of rules to be intersected and produces a single transducer that combines the effect of the selected rules. This is useful because having fewer transducers makes recognition and generation more efficient. Intersecting the eight rules in ENGLISH.RULES takes a couple of minutes and produces a single machine with 108 states. The transducer resulting from an intersection gets its own menu window and you can test it in the same way as individual rules. You can also use it to form another intersection. If the number of rules is large, it is advisable not to try to intersect all of them at once. In the worst case, the number of states in the intersection of two transducers could be as large as the product of the sizes of the input machines, but if the rules represent independent generalizations, the

size may be close to the sum. It is best to start by intersecting rules in small groups on the basis of similarity. If the resulting transducers are not too large they can be combined with one another. There is no limit in principle on the size of transducers although the computation may take several hours for intersections with more than a few hundred states.

## 2.6 Saving the Results

The *Save* option writes out the results of the compilation in three alternative modes. The standard form (explained in section 5.7) is meant for the DKIMMO system. By sliding the mouse off to the right, you can elect to save the results in a "tabular" mode intended for the older implementations of KIMMO or in the "display" mode used on the screen. You can save all the compiled rules or select with the mouse the ones that you want to write into the file.

The file ENGLISH.8FSM on the TWOL floppy contains the result of compiling the eight rules in ENGLISH.RULES individually. ENGLISH.1FSM is the intersection of the rules as a single 108 state machine.

## 2.7 Status Messages

While the compiler is working on a file, it prints a continuous stream of messages about its progress to the status line above the command menu. Most of these messages are self-explanatory. When a rule is being compiled, you can see what part the compiler is currently working on. The symbol <= indicates that the compiler is working on the surface coercion part of the rule, => means that it is compiling a context restriction. For rules that have multiple contexts, the => stage involves several intermediate steps. The symbols lcl <l and >l rcl indicate that the compiler is marking the right end of the first left context and the left end of the first right context with a bracket to keep track of partially overlapping contexts; the symbols <i cp >i indicate that it is working on the restriction that the central part of the rule must have matching brackets on both sides. & means that an intersection is in progress. Once the contexts have been intersected, the auxiliary

brackets are no longer necessary and they are removed; the compiler indicates this operation with the symbol -<>.

## 2.8 User Options

The compiler can be set to check for certain types of conflicts between rules and resolve them automatically. We will discuss these options in section 4.3. If you click the *Option* item on the top of the rule window with the left button, you can see the current setting:


If you decide to change the option, mouse with the left button at the appropriate line. The new selection turns into inverted video and the old one goes back to normal. The change becomes effective when you click DONE. This window remains open until you select either DONE or ABORT.

## 2.9 Bugs

The routine that reads in the rules is extremely fragile in the current version. Any syntactic error in the input puts the system into a break. The only way to recover is to do a reset (Control-D or the uparrow command), fix the error and try again. The variable *TwolGetTop* is bound to the last symbol in the input stream that was read before the break. It can give you a hint where to look for the error. (Please send bug reports and comments to LAURI@CSLI.STANFORD.EDU.)

## 3. Grammar Format

Below is a small but complete example of a two-level grammar. It consists of five sections headed by the titles *Alphabet*, *Diacritics*, *Sets*, *Definitions*, and *Rules*. The first and the last are always required by the compiler, the other three sections are optional. Each section consists of items that are separated by a semicolon. Currently there is no facility for adding comments; this will change in the future.

```
Alphabet
  A:ä O:ö U:y
  a b c d e f g h i j k l m n o p q r s t u v x y w z ä ö ;

Diacritics
  '                                                    ;

Sets
  Consonant = b c d f g h j k l m n p q r s t v w x z '   ;
  BackVowel = a o u                                      ;

Definitions
  NonFrontContext =
    :Consonant* [:BackVowel | :i | :e]* :Consonant*      ;

Rules

  "Vowel Harmony"
     Vx:Vy <=> :BackVowel NonFrontContext* _    ;
                   where Vx in (A O U)
                         Vy in BackVowel
                      matched                            ;
```

This grammar contains only one rule, titled "Vowel Harmony." The effect of the rule is to realize lexical A, O, and U as a, o, and u, respectively, after an occurrence of a surface a, o, or u provided that any intervening surface vowel is either a back vowel or an i or an e. Appendices 1 and 2 are examples of larger rule systems. Let us now discuss the format of a two-level grammar in more detail.

## 3.1 Alphabet

As the compiler goes to work on a two-level grammar, it constructs two lists of alphabetic symbols: *lexical alphabet* and *surface alphabet*. Some symbols occur only in one of these lists but the majority of symbols usually belong to both alphabets. A symbol may consists of more than one letter. Besides the composition of the two alphabets, the compiler also needs to know the possible correspondences: what surface symbols each lexical symbol can be realized as. In reading the declarations and rules, the compiler builds a list of all *valid character pairs*. A character pair is a symbol *x*:*y* where *x* is a symbol in the lexical alphabet and *y* is in the surface alphabet. The colon is used as a separator. The compiler does not consider any pair as valid unless the pair is licensed by a declaration or a rule. The purpose of the *Alphabet* section in the beginning of the grammar is to give the compiler a complete list of the two alphabets and a partial declaration about valid character pairs; the compiler will determine from the rules what other valid character pairs there are. Here is another example of an *Alphabet* section:

```
Alphabet
  N:n K:
  a b c d e f g h i j k l o p q r s t u v x y w z
  :m :n :ng                                        ;
```

By convention, a symbol without a colon on either side represents a pair of identical characters. For example, the letter `b` in this list indicates that (1) `b` is a lexical symbol, (2) `b` is a surface symbol and (3) `b:b` is a valid pair. If a symbol belongs only to the lexical alphabet, it is listed with a trailing colon. In this example, `N` and `K` are lexical symbols while `m`, `n`, and `ng` are only in the surface alphabet. We also see that `N` can be realized as `n` but nothing is said here about the surface realizations of `K` or the lexical counterparts of `m` and `ng`. Because `n` appears on the surface side of the pair `N:n`, listing it separately as `:n` is redundant.

The compiler automatically augments the alphabet with one special pair of symbols. The lexical side of this pair is a symbol indicating a word boundary, the surface character is the symbol for the empty string. What these symbols are is determined by the LISP variables

*TwolBoundaryChar* and *TwolZeroChar*. The default values for these system variables are # and 0, respectively; consequently, the pair #:0 is always included in the alphabet regardless of whether its explicitly declared. (These variables can be reset in LISP to other values, if # and 0 are not appropriate characters to serve in these roles.) The rule tester (section 2.3), automatically adds the appropriate boundary marker to the beginning and end of the input it gets from the user.

## 3.2 Diacritics

The optional *Diacritics* section is a list of lexical characters that by default are realized as the empty string. Furthermore, diacritic symbols are "visible" only for rules that explicitly mention them. The purpose of diacritics is to allow rules to make reference to some nonsegmental property of a lexical form such as tone, stress, morphological class and to provide a facility for handling exceptions. For example, because the consonant doubling in pairs like *infér-inférred* vs. *énter-éntered* correlates with stress, one may decide to mark non-initial stress in lexical forms by a diacritic: in'fer. The consonant doubling rule (see Appendix 1) can thus distinguish the two types of stems.

If the stress mark is relevant only for a certain rule or rules and the symbol itself has no direct surface realization, it is convenient to treat it in a special way so that other rules need not take any note of the presence or absence of a stress mark. The declaration

```
Diacritics
  '                                                    ;
```

has just that effect. The compiler works in such a way that diacritics that are not explicitly mentioned in a rule are completely ignored by the resulting transducer; they have no effect on whether the rule applies to a given form.

## 3.3 Sets

The optional *Sets* section consists of declarations that associate a name with a list of alphabetic symbols. Here is another example:

```
Sets
  Consonant = N b c d f g h j k l m n ng p r s t v x z  ;
  Vowel = A O U a e i o u y ä ö                          ;
  VoicelessStop = k p t                                  ;
  VoicedStop = g b d                                     ;
```

Sets play the same role as distinctive features in other phonological formalisms. Rules can be made more general by referring to symbols as a class instead of listing them individually. As the example shows, it is not necessary for all members of a set to be in the same alphabet. A set name can be used as a lexical or a surface side of symbol pair, for example, `Vowel:0`. The interpretation of set symbols in such pairs depends on what side of the colon they occur. If a set that contains both lexical and surface characters is used on the lexical side, only the lexical members of the set are taken into account; similarly for surface sets. We discuss the interpretation of sets in symbol pairs in more detail at the end of section 4.1.

The order in which the members of a set are listed can be significant in rules that involve variables (see section 4.2). For example, by listing the elements of `VoicelessStop` and `VoicedStop` in the manner shown above, you can write a single voicing rule that realizes every voiceless stop as the corresponding voiced stop; that is, `k:g`, `p:b`, and `t:d`.

## 3.4 Definitions

The optional *Definitions* section provides another way to express generalizations. A context that is shared by several rules can be defined just once and referred to by a name. Like the *Sets* section, *Definitions* consists of a list of declarations; the difference is that the content part of a definition consists of a regular expression. This expression may contain other defined terms provided that their definitions are placed earlier in the *Definitions* list. In the current implementation,

definitions must be constructed using concatenation, disjunction, negation, optionality, Kleene-star and plus. Future versions of the system will make other regular predicates available. Here are more examples of what can go into the *Definitions* section:

```
Definitions
  Sibilant = [s|c] h | s | x: | z          ;
  Syllable = C* V (V) C*                   ;
  LastSyllable = Syllable #:0              ;
```

The vertical bar | indicates disjunction; square brackets are used for grouping; concatenation is indicated by juxtaposition; optional elements are enclosed in parentheses. The symbol `x:` in the definition of `Sibilant` designates any valid symbol pair with `x` as the lexical symbol. For example, if `x` and `c` are the only possible surface realizations of `x` in a grammar, then `x:` designates the pairs `x:x` and `x:c`. In this case, `Sibilant` means 'sh, ch, x, x:c, or z.' The meaning of symbols that consist of just one half of a symbol pair, such as `x:`, depends on the set of valid symbol pairs; they are defined *implicitly* (more about this in section 4.1).

An important difference between set symbols and defined terms is that a defined term cannot be a part of a symbol pair. An expression such as `Sibilant:0` does not make sense, given the above definition, because `Sibilant` here designates a two-level environment rather than a set of alphabetic symbols.

### 3.5 Rules

A two-level rule consists of a name (a quoted string) followed by a *correspondence*, *operator*, any number of *environments*, and an optional *variable assignment*. An environment consists of a *left* and a *right context* (both optional) separated by an underscore (_). There are four operators: `<=`, `=>`, `<=>`, and `/<=`. We will illustrate their effect with a few simple examples and discuss the rule formalism in greater detail in the next section. Here is a simple example of a two-level rule:

```
"Voicing rule 1"
  k:g <= Vowel _ Vowel                     ;
```

Every two-level rule is concerned with a particular surface realization of a given lexical string in a certain environment; in our example, the issue is the realization of k between two vowels. The effect of the <= operator is to make the realization of k as g obligatory in this environment. In words, this rule says "k is always realized as g between two vowels." The rule does not in any way constrain the realization of k in other environments; in particular, it allows the realization k as g anywhere. If the user wants only intervocalic k's to be realized as g's, he can state the restriction using the => operator:

```
"Voicing rule 2"
  k:g => Vowel _ Vowel                                    ;
```

Voicing rule 2 is equivalent to "k is realized as g only between two vowels." Note that this rule does not require intervocalic k's to be realized as g's; it only allows it and restricts it to this one environment.

Rules 1 and 2 can be combined to a single rule by using the <=> operator:

```
"Voicing rule 3"
  k:g <=> Vowel _ Vowel                                   ;
```

In words: "k is realized as g always and only between two vowels." Although it is useful to collapse rules when possible, it is not necessary to do so. Rules 1 and 2 have jointly the same effect as rule 3 alone. The main benefit of the => operator is that it can be used to allow a lexical string to be optionally realized in a certain way without requiring that it be always realized that way in the given context.

The fourth operator, /<=, makes it possible to state prohibitions:

```
"Voicing rule 4"
  k:g /<= Consonant _ Vowel                               ;
```

In words: "k is never realized as g between a consonant and a vowel." This operator would be useful in cases in which it is easier to describe

the environments in which something does not happen than to enumerate the positive contexts.

A rule can have more than one environment:

```
"Voicing rule 5"
  k:g <=> Vowel _          ;
          _ Vowel          ;
```

Rule 5 requires a k to be realized as g both before and after a vowel and only in these environments. Note that the two environments must be separated by a semicolon; the second semicolon terminates the rule. This rule is compatible with rule 1—in fact, it makes rule 1 redundant—but it would not work properly in conjunction with rules 2, 3, and 4. Because rule 5 requires a prevocalic k to be realized as g no matter what precedes it, it can conflict with rule 4 that prohibit this realization after a consonant. It also conflicts with rules 2 and 3 that restrict the voicing of k to intervocalic contexts. We will return to the issue of rule conflicts in section 4.3.

As a final example of the kinds of rules that can be written in the two-level formalism, consider a rule that makes k be realized as g between two identical vowels. This type of rule can be written with the help of a variable:

```
"Voicing rule 6"
  k:g <=> Vx _ Vx ; where Vx in Vowel                    ;
```

Here Vx is used as a variable; the assignment of values to the variable is determined in the where-clause—note again the use of ; as a separator. This rule is equivalent to a rule that has multiple environments: a _ a, u _ u, and so on for each vowel. Variables do not add to the descriptive power of the formalism but they make it possible to state many rules in a more concise way. In section 4.2 we illustrate more sophisticated ways of using variables.

## 4. Rule Formalism

The two-level rule formalism was first described in Koskenniemi [1983]. The version used by the current compiler contains some changes and improvements, such as multiple environments and a more general use of variables. The most radical change concerns the interpretation of the formalism in the case of conflicting rules (section 4.3). When two rules are in conflict, some valid lexical forms may be left without any surface realization. Such conflicts are sometimes difficult to detect in practice and their resolution typically forces the linguist to give up simple rules in favor of more complicated variants. The TWOL compiler can detect and resolve certain conflicts automatically in a principled way but the user still has the option of not making use of it and treating all conflicts as errors (section 2.8).

In this section we will discuss three issues related to the formalism: the specification environments, the use of variables, and the principles of conflict resolution.

### 4.1 Two-level Environments

All two-level rules are statements about the realization of lexical forms as surface forms. This concerns not just the correspondence part of a rule but also the specification of the environment. This is a subtle but important point because traditional phonological rules are different in this respect. It is somewhat obscured by the notational "sugar" that lets the rule writer construct environments with simple symbols which actually designate symbol pairs. The fact that two-level rules can refer to both levels in their context specification makes it possible to give simple two-level descriptions of many phenomena which cannot be described elegantly with traditional rules without rule ordering. For example, consider the following two rules:

Rule 1: Assimilate nasals that are unspecified as to the point of articulation to the following stop.
Rule 2: Replace a stop with the preceding nasal.

These rules account for a common process by which underlying Nk, Np, Nt are realized as surface ŋŋ, mm, and nn. In very traditional phonology, it is crucial that Rule 1 is applied before Rule 2 because the application of Rule 2 changes the environment so that Rule 1 fails to apply. Because two-level rules can refer to both lexical and surface levels, problems of this type generally have very simple solutions. The two-level counterparts of rules 1 and 2 can apply simultaneously. We first give a simple version of the rules for N and p and show in section 4.2 how to generalize them to other stops.

```
"Nasal assimilation to p"
      N:m <=> _ p:      ;

"Nasalization of p"
      p:m <=> :m _      ;
```

The subtle detail in these rules that makes them work in the desired way is the location of the colon in p: and :m. The former is a lexical p with some unspecified surface realization, the latter is a surface m with an unspecified lexical source. Because one rule is sensitive to the lexical context and the other to the surface context, they do not interfere with each other. They realize Np as mm without any stipulation about order of application. However, the choice of lexical vs. surface context is a delicate matter, as we shall see shortly. (There are, of course, more sophisticated ways of stating the rules in the traditional framework that also produce the correct outcome without rule ordering. This is not the issue here.)

As we mentioned in section 3.4 symbols such as p: and :m have implicit definitions. If a lexical p can only be realized as p or m, then p: in the nasal assimilation rule is equivalent to the disjunction [p:p | p:m]. It is important to realize that in this formalism *everything* in a context specification constrains both levels although the notation does not always show it. In practice it often happens that a rule does not have the intended effect because it fails to constrain one or the other level in the appropriate way. For example, consider the effect of making a slight change to the nasal assimilation rule. We'll just leave out the colon:

```
"Nasal assimilation to p (wrong version 1)"
     N:m <=> _ p ;
```

Because p in the context specification is just an abbreviation for p:p, the new rule says that N is realized as m always and only *in front of a p that is realized as p*. The two rules are now in conflict: p cannot be realized as p because then N should be realized as m which makes the pair p:p illegal by the nasalization rule. On the other hand, p cannot be realized as m either because that would require N to be realized as m which is now forbidden by the first rule. This is a type of unintentional error that the compiler cannot yet detect; consequently, the rule writer has to keep in mind that a specification, such as p, that seems simpler than p: or :p is actually more restrictive than the latter two although it does not overtly refer to levels.

Overspecifation does not always lead to a conflict, it can also have the opposite effect. Consider another faulty version of the nasal assimilation rule:

```
"Nasal assimilation to p (wrong version 2)"
     N:m <=> _ p:m     ;
```

The replacement of p: by p:m in the right context seems completely innocuous; after all, lexical p should be realized as m when the preceding nasal is realized as m. The problem with this version is that it does not force N to be realized as m in front of a p that is realized as something other than m, and the nasalization rule does not require the realization of p as m except when if follows a surface m. If N is generally realized as n, the rules now allow two realizations for Np, namely, np and mm.

Set symbols are similar to alphabetic symbols in that they, too, are interpreted as symbol pairs when they are used without a colon. For example, if VoicelessStop is defined as the set k, p, t; as part of a context specification, VoicelessStop is an abbreviation for the pair VoicelessStop:VoicelessStop. The interpetation of a symbol pair containing a set name is perhaps not immediately obvious but it is easy to get used to. Such pairs also have implicit definitions: VoicelessStop designates an alternation consisting of valid pairs *x*:*y* such that *x* and *y*

are voiceless stops. Depending on the grammar, that might mean just [k:k | p:p | t:t] but if the grammar happens to allow k to be realized as p in some environment, then the pair k:p is also included in the environment VoicelessStop. Pair symbols consisting of a set name and an unspecified other half are interpreted in the same way. For example, VoicelessStop: designates all valid pairs *x*:*y* in which *x* is a voiceless stop. The identity of *y* does not matter, provided that the pair itself is licensed by some declaration or rule. Consequently, VoicelessStop: is in general less restrictive than VoicelessStop as a context specification.

### 4.2 Variables

Suppose we wish to extend the simple nasal assimilation rule for p in section 4.1 to other stops. One solution would be to write three separate rules that realize Nk, Np, and Nt as ng, mm, and nn, respectively, but that would conceal a generalization. Although the two-level rule formalism does not have a facility for factoring out the common element in these rules in terms of distinctive features, it can achieve a similar result with the help of sets and variables. Let us assume that the alphabet and sets Velar and VelarOrDental have been defined as follows:

```
Alphabet
  N:
   a b c d e f g h i j k l o p q r s t u v x y z
   :m :n                                        ;

Sets
  Velar = g k ;
  VelarOrDental = k g t d s l r n ;
  Vowel = a e i o u ;
  VoicelessStop = k p t ;
  VoicedStop = g b d ;
```

Here are more general versions of the two rules in section 4.1:

```
"Nasal assimilation"
  N:Cx <=> _ Cy:   ; where Cx in (n m)
                         Cy in (VelarOrDental Labial)
                       matched                      ;
```

```
"Nasalization"
  Cx:Cy <=> N: _   ; where Cx in VoicelessStop
                          Cy in (g m n)
                       matched    .*                    ;
```

The effect of the nasal assimilation rule is that a lexical N is realized either as n or m depending on the following consonant; the nasalization rule makes k, p, and t be realized as g, m, and n, respectively, after a lexical N. These rules would be a bit more natural if we did not follow the orthographic convention of using ng to represent a velar nasal.

As these examples indicate, a rule may contain more than just one variable. The range of values for variables is declared in a *where*-clause. The range may be indicated by a set, for example, VoicelessStop, or by a list. The list may consist of alphabetic symbols, as in (g m n), or sets, as in (VelarOrDental Labial) or defined terms. The use of the variable must of course be consistent with the values that are assigned to it. Only alphabetic and set symbols can form symbol pairs (Cy:, Cx:Cy); defined terms do not make sense on either side of a colon because they already designate two-level contexts.

The assignment of values to a group of variables can be coordinated by using one of three keywords: *matched*, *mixed*, or *freely*. When the keyword is *matched*, as in our example, the values are assigned to all variables in the group simultaneously starting from the leftmost value in the range of each variable. For example, in the case of the nasal assimilation rule, this mode of assignment produces two instances of the rule:

```
"Nasal assimilation (Assignment 1)"
  N:n <=> _ VelarOrDental:    ;

"Nasal assimilation (Assignment 2)"
  N:m <=> _ Labial:     ;
```

The keyword *mixed* means that no two variables in a group ever have corresponding values; that is, when one variable has the $n^{th}$ value of its

range, all other variables have something other than the $n^{th}$ value of their range. This makes it possible to write rules that involve dissimilarity; for example, we can a write a rule that deletes the last one of two non-identical vowels:

```
"Vowel truncation"
   Vx:0 <=> Vy:  _  ;  where Vx in Vowel
                             Vy in Vowel
                          mixed ;
```

If the keyword were changed to *matched*, this rule would pertain to a sequence of two identical vowels, and *freely* would mean a sequence of any two vowels. If no assignment mode is specified, the compiler defaults it to *freely*.

A rule may contain any number of variable groups, each with its own assignment mod; the groups must be separated by *and*. For example, the following rule realizes a voiceless stop as the corresponding voiced stop between two identical vowels:

```
"Voicing of stops"
   Cx:Cy <=> :Vz  _  :Vz      ;  where Cx in VoicelessStop
                                       Cy in VoicedStop
                                    matched
                                 and Vz in Vowel
;
```

If a variable group contains just one variable, as the second group in this example, there is no difference between *matched*, *mixed*, and *freely*.

### 4.3 Rule Conflicts

As we pointed out in section 4.1, a grammar may contain rules that are in conflict with one another: one rule prohibits something that another rule requires. The two-level formalism gives rise to two types of conflicts between rules: *environment* conflicts ("=> conflicts") and *realization* conflicts ("<= conflicts"). The => side of two rules are in

conflict when the correspondence part is the same but the contexts differ; for example, the rules

```
"Rule 1"
   k:0 <=> V _ V C C    ;

"Rule 2"
   k:0 <=> C _ V C C    ;
```

conflict in the => direction. Rule 1 says that k is realized as 0 in the environment V _ V C C whereas Rule 2 calls for the environment C _ V C C. If the sets V and C are disjoint, the two rules cancel each other out completely: k cannot be realized as 0 anywhere.

The second type of conflict arises when two rules have the same lexical side, different realizations, and the environment of one rule is subsumed by the environment of the other. Consider the rule

```
"Rule 3"
   k:v <=> u _ u C C    ;
```

Because every instance of the context u _ u C C is also an instance of V _ V C C, Rule 3 and Rule 1 are in conflict with respect to how k should be realized in the environment of Rule 3. Note that there is no conflict here in the => direction because different realizations are involved. In traditional phonology, the problem could be solved by ordering Rule 3 before Rule 1 but in a two-level system, this solution is not available.

Both types of conflicts can of course be avoided by making appropriate changes in the conflicting rules although this generally involves making the rules more complicated. If the conflict arises in one assignment of a value to a variable, a general rule may have to be split into a number of separate rules. In this matter the compiler can help the linguist in two ways. Before a set of rules is compiled, the user can instruct the compiler to check for both types of conflicts and report them as the rules are compiled. This is done by selecting the appropriate command (*Report only*) from the *Option* menu (section 2.8). In this mode, TWOL prints the following messages when it compiles rules 1, 2, and 3:

```
Rules "Rule 1" and "Rule 2"
      overlap with respect to k:0.
=> conflict between "Rule 1" and "Rule 2"
      with respect to k:0
<= conflict between "Rule 1" and "Rule 3"
      with respect to k:0 and k:v
```

These messages are rather specific because a conflict might involve only a particular assignment of value to some variable in a more general rule.

If such conflicts arise by mistake, the grammar writer can correct the errors and recompile the rules. Another option (*Report and Resolve*) is to let the compiler not only check for conflicts but to alter its compilation method to achieve a particular intended effect. In the case of => conflicts, the context part of both rules is modified to include the context of the conflicting rule as an alternative. In resolving the environment conflict between rules 1 and 2, the compiler compiles the => part of these rules with a common context:

```
V _ V C C  ;
C _ V C C  ;
```

Note that the methods of resolving => and <= conflicts may involve changing the compilation of a general rule just under a particular assignment of values to variables leaving other instances of the rule unchanged.

In resolving the realization conflicts, the compiler follows the principle that the more specific rule should take precedence. (This is often called the "Elsewhere Principle.") The specific rule is compiled in the normal way but the general rule is altered so that it does not interfere with the specific one. For example, the compiler resolves the conflict between rules 1 and 3 by weakening the <= side of Rule 1 so that it requires k to be realized as either 0 or v in its environment. The effect is that k is realized as v between two u's because Rule 3 requires and Rule 1 now allows it. Between other vowels k is deleted because Rule 3 does not

permit it to be realized as v whereas Rule 1 requires that it either be deleted or realized as v. Section 5.3 discusses this in more detail.

The effect of the automatic conflict resolution can be seen in the transition tables for Rule 1 under the two modes of compilation. Although it is generally rather difficult to associate the states and transitions of the automaton with any details of the source rule, some general relationships are easy to discern.

*Without Resolution*

```
Rule 1
    a b k y k:0
0:  1 0 0 0
1:  1 0 2 0  3
2:  4 0 0 0
3:  5
4:  1 7 6 0  3
5:    8 8
6:  4       0
7:  1       0
8:    0 0

Equivalence classes:
  (a e i o u)
  (b c d f g h j l p q r
     s t v x z)
  (k k:v) (y #:0) (k:0)
```

*With Resolution*

```
Rule 1
    a b k y k:0
0:  1 2 2 0
1:  1 2 4 0  3
2:  1 2 2 0  3
3:  5
4:  6 2 2 0  3
5:    7 7
6:  1 8 9 0  3
7:    2 2
8:  1       0  3
9:  6       0  3

Equivalence classes:
  (a e i o u)
  (b c d f g h j l p q r
     s t v x z k:v)
  (k) (y #:0) (k:0)
```

Note that the distribution of k:0 is less restricted when the rule is compiled in the *Report and Resolve* mode. This results from the addition of the context of Rule 2 to the environment of Rule 1. Another significant change is that, in the second version of the rule, the k:v pair has moved from the small k-class—the forbidden realizations of k in the rule's environment—to the large b-class that contains all the consonant correspondencies that the rule is not concerned about. This difference is due to the resolution of the <= conflict between Rule 3 and Rule 1.

Appendix 2 contains a set of seven rules that deal with consonant gradation in Finnish. These rules give rise to multiple conflicts of both types. If the compiler did not have the facility to resolve them, these rules could not be as simple as they are.

## 5. Compilation

The TWOL compiler derives from unpublished work by Ronald Kaplan and Martin Kay in the early 1980's. Kaplan and Kay proved that standard phonological rules of the type $\alpha \rightarrow \beta$ / $\lambda$ __ $\rho$ define regular string relations and thus can be modelled by finite-state transducers. (Johnson 1972 also argued that finite-state mappings are powerful enough to model a wide variety of phonological processes.) Kaplan and Kay developed a technique for compiling a system of ordered rewrite rules to a cascade of transducers that could be further composed to a single such device. In the summer of 1985, Kaplan and Koskenniemi worked out the principles that made it possible to apply the method to Koskenniemi's two-level rules. The first version of the compiler was written by Koskenniemi in 1985-86 [Koskenniemi 1986] and re-implemented in Common Lisp by Maarit Kinnunen [1986]. The present version contains a number of improvements due to Karttunen, including a better user interface, a more general treatment of variables, and the facility for resolving rule conflicts. Most of the low-level computation is done by Kaplan's FSM package; the resulting automata are minimal and deterministic. (A rule compiler that produces non-deterministic automata for a simpler two-level formalism has been developed by G. D. Ritchie *et al.* 1985.)

### 5.1 Overview

In the first stage of the compilation, the input file is parsed and the definitions and rules are converted to LISP structures. Except for defined terms, all atomic symbols in the rules are converted to symbol pairs and the basic components of each rule (*correspondence part*, *left contexts*, *right contexts*) and all the definitions are parsed into regular expressions. As it processes the input file, the compiler carefully tracks all pairs that consist of alphabetic symbols (valid pairs).

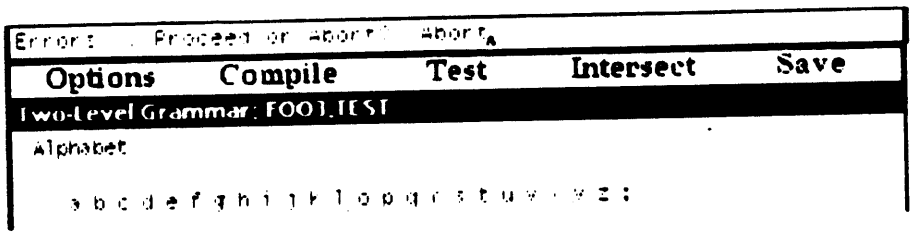
The next step is to remove all variables from the rules by replacing every original rule that contains variables by a set of subrules, one for each value assignment. In general this process produces new valid pairs. If a variable is used in the correspondence part of a rule, one instance of the whole rule is produced for every value assignment;

variables that only occur in the context part produce a new instance of the context for each assignment of value to the variable. For example, the expanded version of

```
Cx:0 <=> Vy _ Vz          ;  where Cx in (k p t)
                                   and Cy in (a o u)
                                       Cz in (a o u)
                                    mixed                ;
```

consists of three subrules for k:0, p:0, and t:0, and each subrule has six contexts: a_o, a_u, o_a, o_u, u_a, and u_o.

After all the rules have been instantiated to sets of subrules, the compiler has the final list of possible lexical-to-surface correspondences. It uses this information to compute a definition for all implicitly defined terms, such as i: and Vowel:Vowel, whose content depends on the list of valid pairs. If the compiler detects any errors, for example, an undeclared symbol in the alphabet or an implictly defined term that doesn't designate anything, it reports the problem and asks the user at this point for a decision:




In the next stage, all definitions, which at this point are regular expressions, are individually compiled to automata. These automata are ordinary finite-state machines (FSM) except that the transition arcs are labeled by symbol pairs. The compiled definitions are stored and a copy of the automaton is used in compiling expressions that contain the defined term.

Before the compiler starts to work on the rules, it makes a preliminary pass during which all the components of every subrule are individually compiled to an FSM. This is useful because the compiler can in the next stage use semantic methods—predicates that are defined on

FSMs—rather than the syntax of regular expressions to find out what rules are in conflict.

In the beginning of the final stage, each source rule is represented by a set of subrules and a subrule may have several contexts. All the subrules are first compiled separately and the resulting FSMs are successively intersected to produce a single FSM. A menu window is created to give the user access to the result.

## 5.2 Compilation of a Rule Set

Unless the compiler is working in the *Don't check* mode, every subrule is first scanned for possible conflicts against all the other subrules in the grammar. This extra work seems to nearly double the total compilation time. If a conflict is detected, a message is printed on the screen and the conflict is recorded on the subrule so that it is available when the relevant part of the rule is compiled. The resolution of <= conflicts only requires a modifcation of the more general subrule, but both parties have to be modified to resolve a => conflict (section 4.3).

Before the actual compilation begins, the compiler first considers the alphabet of the rule. Although the automaton that it is about to produce must function correctly with the full alphabet of the grammar, it is usually possible to ignore many symbols during the compilation. A reduced alphabet means fewer transition arcs and less work for the compiler. In order to define a minimal alphabet for the rule, the compiler canvasses all the individually compiled components in the rule set. Any two symbols that never appear on transitions that lead to distinct states are equivalent, as far as the particular rule is concerned; they can be represented by just one symbol. On that basis, the total alphabet is partitioned to a set of equivalence classes; each equivalence class is ordered alphabetically and the first symbol is picked to represent the whole class. All the other transitions in the component FSMs are pruned.

The first step in compiling a subrule is to expand the left (*lc*) and the right (*rc*) context FMSs to their full length. In the two-level formalims, as well as standard phonology, it is customary to specify only that part

of the total environment that is constrained by the rule. Implicitly, the environment of every rule extends to infinity in both directions. In extending the context, the compiler concatenates the context FSMs with a pruned version of a machine, let us call it *pi**, that represents every possible string over the two-level alphabet. The *pi** FSM is also handy for other purposes as we shall see shortly. We use the symbol *cp* for the FSM that represents the correspondence part of the rule; *←lc* and *rc→* designate the extended context FSMs:

```
←lc = pi* lc
rc→ = rc pi*
```

The next step depends on the operator of the rule. Rules that involve just a one-sided arrow (<= or =>) or the prohibition operator /<= (section 3.5) can be compiled in one step, rules with the <=> operator are compiled twice, once with <= the other time with => and the results are intersected.

### 5.3 Left-Arrow Rules and Prohibitions

Let us assume, for the time being, that we are dealing with a rule that has just one context. The purpose of a <= rule is to require that a lexical string, usually a single symbol, is realized in a certain way in some set of environments. In order to enforce this requirement, the compiler uses the *cp* FSM as a basis for another FSM, call it *anti-cp*. The *anti-cp* is identical to the *cp* machine as far as the lexical side is concerned but has a different surface side. The surface side of *anti-cp* includes all possible surface realizations of the lexical string except the realization specified by *cp*. For example, if the *cp* machine represents the pair k:0, and the other valid k-pairs in the two-level alphabet are k:k, k:v, and k:j, then the *anti-cp* FSM corresponds to the disjunction [k:k | k:v | k:j]. (We use | to represent the operation that combines two FSMs to produce an automaton whose language is the union of the languages of the input machines.)

If the rule in question conflicts with other <= or <=> rules that have a more specifc environment and the compiler is running in the *Report and resolve* mode, the conflicts are resolved at this point. This is

accomplished by removing from the *anti-cp* machine the *cp* strings of the conflicting rules. schematically:

$$\text{anti-cp}' = \text{anti-cp} - [cp_1 \mid cp_2 \mid \ldots \mid cp_n]$$

Here *anti-cp* is the original version and $cp_1 \ldots cp_n$ are the *cp*-parts of the conflicting more specific rules. (We use - for relative complementation: $m_1 - m_2$ is an automaton whose language consists of the strings that are in the language of $m_1$ but not in the language of $m_2$.) This modification amounts to a weakening of the <= part of the rule because realizations that would have been disallowed by the original *anti-cp* FSM now become permissible.

The effect of the <= rule is achieved by building an automaton that prevents the occurrence of *anti-cp'* in the context of *←lc _ rc→*. We first construct, by concatenation, the following intermediate FSM:

```
Fsm1 = ←lc anti-cp' rc→
```

*Fsm1* is an automaton that accepts any string that contains an occurrence of a string from the language of *anti-cp'* sandwiched between an *←lc* string and and *rc→* string. This of course just the opposite of what we want, so the final result, *Fsm2*, is formed with negation. The most efficient way to do that is to take the complement of *Fsm1* with respect to the *pi** machine:

```
Fsm2 = pi* - Fsm1  .
```

*Fsm2* is a machine that accepts every string in the language of *pi**—the total set of strings over the reduced two-level alphabet—except for strings that belong to the language of *Fsm1* and thus contain an unwanted realization of the lexical side of *cp* in the context *←lc _ rc→*.

The compilation of a /<= rule is even simpler in that the *cp* part itself plays the role of the forbidden correspondence. The FSM for a prohibition rule is constructed in accordance with the formulas:

```
Fsm1 = ←lc cp rc→
Fsm2 = pi* - Fsm1  .
```

So far we have assumed that the rule in question just has one context. What if there are several *←lc _ rc→* pairs? We repeat the same procedure for each environment. The final result could be obtained by intersecting the intermediate results but the compiler uses another, equivalent method. When it has finished compiling the rule for one environment and goes on to the next one, the result of the first compilation is used, instead of *pi**, for complementation at the point where *Fsm2* for the next environment is constructed. In that way, the compilation procedes incrementally so that the process is finished when the *Fsm2* for the last context has been compiled.

### 5.4 Simple Right-Arrow Rules

The compilation of => rules is a bit more complicated, especially when there are several contexts. Even if the source rule has only one environment, the resolution of => conflicts may add more contexts. If the compiler is running in the *Report and Resolve* mode, its first action is to augment every subrule with the contexts of conflicting subrules.

Let us first consider the case in which there is only one environment. The purpose of a => rule is to limit the occurrence of the *cp* part to the *←lc _ rc→* context. The first step is again to construct the FSMs that represent the two situations that we wish to exclude, namely: the occurrence of a *cp* string after a string that is not in the language of *←lc* and the occurrence of *cp* string in front of a non-*rc→* string. Let us use $\overline{\leftarrow lc}$ and $\overline{rc\rightarrow}$ to designate the complements of *←lc* and *rc→* with respect to *pi**.

$$\overline{\leftarrow lc} = pi^* - \leftarrow lc$$
$$\overline{rc\rightarrow} = pi^* - rc\rightarrow$$

The machines that represent the two unwanted cases are *Fsm1* and *Fsm2*.

$$Fsm1 = \overline{\leftarrow lc}\ cp\ pi^*$$
$$Fms2 = pi^*\ cp\ \overline{rc\rightarrow}$$

*Fsm1* is an automaton that accepts any string that contains an occurrence of *cp* preceded by a non-*←lc* string regardless of what

follows, *Fsm2* is the symmetric automaton for the right context. The negations of these machines encode the desired restrictions:

```
Fsm3 = pi* - Fsm1
Fsm4 = pi* - Fsm2
```

*Fsm3* is a machine that rejects strings containing an instance of *cp* that is not preceded by the proper left context. In the language of *Fsm4* every occurrence of *cp*, if any, is followed by the proper right context. From here we could get the final result by intersecting the two automata but the compiler actually does it slightly differently; in compiling the *Fsm4* machine, it does the final complementation with respect to *Fsm3* instead of *pi**:

```
Fsm4 = Fsm3 - Fsm2.
```

This produces the desired effect: an automaton that blocks if a *cp* string occurs without an *←lc* string on the left and *rc→* string on the right.

### 5.5 Right-Arrow Rules with Multiple Contexts

Let us now consider a => rule that with several environments. The purpose of such a rule is to constrain the occurrence of the *cp* part so that it satisfies at least one of the alternative context restrictions:

$$\leftarrow lc_1 \_ rc\rightarrow_1,\ \leftarrow lc_2 \_ rc\rightarrow_2,\ \ldots\ ,\ \leftarrow lc_n \_ rc\rightarrow_n$$

This constraint is more complicated to implement than the corresponding case for <= rules because we need, in effect, to take the union of $\leftarrow lc_1 \ldots \leftarrow lc_n$ and the union of $rc\rightarrow_1 \ldots rc\rightarrow_n$ without losing the connection between the matching $\leftarrow lc_i \_ rc\rightarrow_i$ pairs. Another complication is that a string of symbol pairs that constitute the *cp* part on one application of the rule may overlap with the *←lc* or *rc→* part of another rule application. We return to this problem section 5.6.

In order to accomplish its task, the compiler adopts the technique devised by Kaplan and Kay [1985] to handle similar overlaps in ordered rewriting rules. It uses pairs of indexed brackets for bookkeeping: $<_1 >_1, <_2 >_2, \ldots, <_n >_n$, one pair for each context. These brackets are

symbols disjoint from the alphabet so they cannot be confused with other parts of the rule. The $pi^*$ automaton and all the component FSMs of the rule are modified so that they freely allow auxiliary brackets to occur anywhere; that is, every state of the modified automata has a transition back to the same state over every bracket. Let us call the augmented context automata $pi^{*\prime}$, $lc'$ and $rc'$, with the convention that, for any automaton $x$,

$$x' = x \text{ with } <_1 >_1, <_2 >_2, \ldots, <_n >_n \text{ occurring freely}$$

The first step is to associate every context pair with a matching pair of brackets. We want each left bracket to be preceded by the corresponding left context, and each right bracket to occur in front of the corresponding right context. The compiler constructs an auxiliary automaton, *Fsm1*, that embodies these constraints. *Fsm1* is derived by compiling the following two simple rules:

$$<_i \Rightarrow lc_i' \_$$
$$>_i \Rightarrow \_\ rc_i'$$

for all $i$ from 1 to $n$ in an incremental way so that the final result is equivalent to the intersection of all the instances:

$$Fsm1 = <_1 \Rightarrow lc_1' \_ \ \&\ >_1 \Rightarrow \_\ rc_1' \ \&\ \ldots \ \&$$
$$<_n \Rightarrow lc_n' \_ \ \&\ >_n \Rightarrow \_\ rc_n'$$

These rules are compiled just as we described in section 5.4; the result of one compilation is used for complementation in the next round so that all the constraints at the end come together in the same automaton. The only difference is that we are using the modified version of $pi^*$, $pi^{*\prime}$, in extending the left and right contexts. The resulting automaton, *Fsm1*, accepts strings in which any occurrence of $<_l$ is immediately preceded by the first left context and any occurrence of $>_l$ is immediately followed the corresponding right context; similarly for the other contexts and brackets.

The second step is to compile the main part of the rule using pairs of brackets to represent the multiple environments. A second auxiliary machine, *Fsm2*, is constructed by compiling the rule

$$cp \Rightarrow <_i \_ >_i$$

separately for each value of $i$ and by applying the Kleene-star operation to the disjunction of the results. Here, too, contexts are extended to their full length with $pi^{*\prime}$. Schematically,

$$Fsm2 = [cp \Rightarrow <_1 \_ >_1 \mid \ldots \mid cp \Rightarrow <_n \_ >_n]^*$$

*Fsm2* accepts strings in which any occurrence of cp is surrounded by a pair of matching brackets. We now combine the constraints encoded in *Fsm1* and *Fsm2* by intesecting the two automata. (The intersection operation, &, produces an automaton whose language is the intersection of the languages of the input machines.)

$$Fsm3 = Fsm1 \ \& \ Fsm2$$

*Fsm3* is an automaton that accepts strings in which any occurrence of *cp* is in the context

$$pi^{*\prime} \ lc_i' <_i \_ >_i \ rc_i' \ pi^{*\prime}$$

for some $i$ between 1 and $n$. This is almost what we want except for the spurious brackets. To remove them, we first convert all bracket transitions in *Fsm3* to epsilon transitions and then determinize and minimize the result. This final modification of *Fsm3* leaves in effect just the constraint we wished to express: every occurrence of *cp* is licensed by at least one of the alternative contexts.

### 5.6 Overlapping Contexts

The compilation technique in section 5.5 may seem overly complicated. Couldn't we get the same result without any brackets from the formula we just used to derive *Fsm2*? At the first glance there appears to be no reason why the following method would not give the right result:

$$Fsm4 = [\ cp \Rightarrow lc_1 \_ rc_1 \mid \ldots \mid cp \Rightarrow lc_n \_ rc_n\ ]^*$$

Each of disjuncts from which *Fsm4* is composed requires that any occurrence of *cp* be flanked by a particular left and right context.

In many cases, this method does indeed produce the same result as the more complicated procedure in section 5.5 but there are situations in which the results are not equivalent. Consider the following simple rule:

```
"A-to-B"
  a:b => a: _     ;
           _ :b   ;
```

The "A-to-B" rule says that a lexical a is realized as a b only when it either follows another lexical a or precedes another surface b. If we compile the rule following the steps we used in the previous section, we get the following transducer

```
A-to-B                Fsm3 (correct)
    a  b  a:b
0:  1  0   2
1:  1  0   1
2.     0   1
```

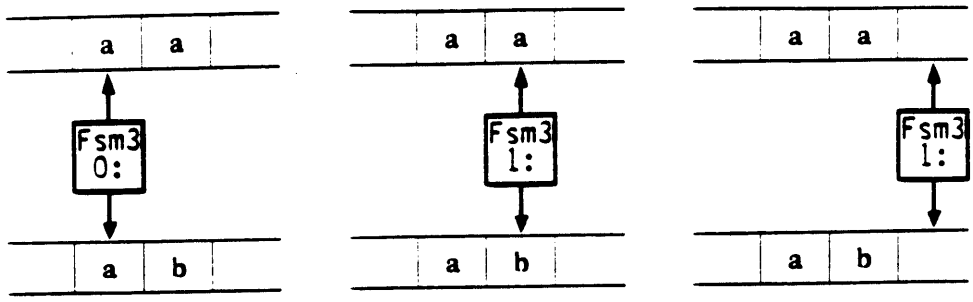
The formula for *Fsm4* leads to the following resul t:

```
A-to-B                Fsm4 (incorrect)
    a  b  a:b
0:  1  0   2
1:  1  0   1
2.     0   2
```

The two transducers differ with respect to just one transition on the bottom of the rightmost column. In state 2 (non-final), *Fsm3* has an a:b arc to state 1 whereas the a:b arc in *Fsm4* loops back to state 2. (Note that, the symbols a and b here are abbreviations for a:a and b:b. respectively.)

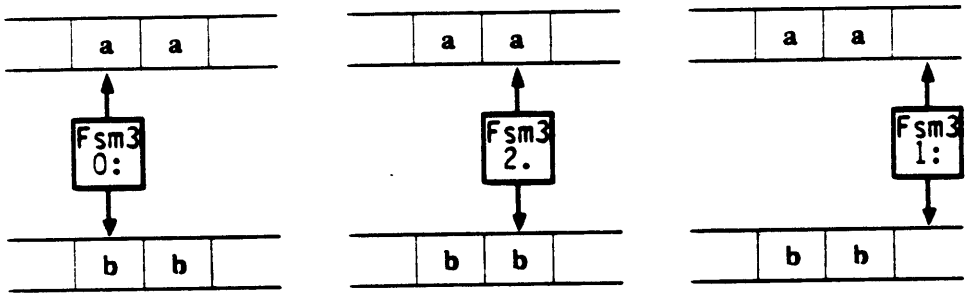
Let us examine the behavior of these transducers with the help of diagrams. We represent a transducer as box with two scanning heads. The upper tape is the lexical side of the correspondence, the lower tape represents the surface side. The number in the box indicates the state of the machine; a colon shows that the state is final, a period marks the state as non-final.

Consider the realization of the lexical string aa as the surface string ab. As the following sequence of snap shots shows, *Fsm3* accept this correspondence.

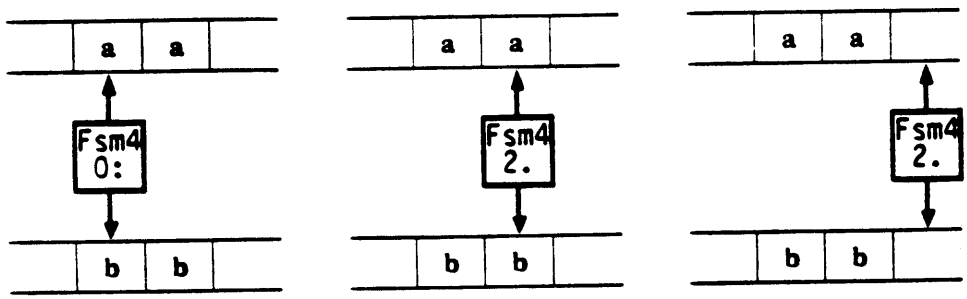



This is as it should be, because the rule in question allows (but does not require) a lexical a to be realized as b after another lexical a. The behavior of *Fsm4* is the same; both machine make the same moves over this pair of tapes.

The difference between the two transducers becomes manifest when we consider the realization of lexical aa as surface bb. *Fsm3* accepts it:




but *Fsm4* does not because it ends up in a non-final state:



Which machine is correct? It is easy to see that *Fsm4* is in error in rejecting the correspondence. The realization of the first a as b is licensed by the fact that it precedes a surface b; the realization of the second a as b is licensed by the preceding lexical a, hence *Fsm3* is faithful to the rule but *Fsm4* is not.

The characteristic feature of this example that brings out the difference between the two machines is that the *cp* part of the rule, a:b, overlaps with a context specification. The flaw in the simple technique that gave rise to *Fsm4* is that it does not allow for the possibility that some correspondence—here the first a:b pair—which is sanctioned by one disjunct in the rule, may at the same time be a crucial part of the context in another disjunct that validates some preceding or following correspondence—in this case, the second a:b pair. In *Fsm4* the two environments are compiled in a way that makes this type of interaction impossible: one alternative environment cannot start to "take-over" until the other one has "finished." As Kaplan and Kay showed for rewriting rules, the role of the auxiliary brackets is to break up the environments so that this can happen while the link between the left and the right side of the same environment is maintained with the aid of the indices.

### 5.7 Compiler Output

When a rule is compiled, the compiler first partitions the total two-level alphabet to a set of equivalence classes and then picks just one symbol pair to represent all the symbol pairs in the same class. In order to use the results of the compilation, we of course need to know how each symbol in the full alphabet is represented in the local alphabet of each machine. If we think of the automata in a tabular form, this mapping is a function that assigns to every symbol in the alphabet one column from each transition table, for that reason, we call refer to it as an "alignment" function.

Consider the following mini-grammar:

```
Alphabet
  A:e a g e :k                    ;
```

```
Rules
  "Harmony"
     A:a <=> :a [g: | :a]* _    ;
  "Devoicing"
     g:k <=> _ #:               ;
```

The first rule realizes a lexical A as a after a surface a provided that all the intervening symbols either have g on the lexical side or a on the surface side. The second rule realizes a lexical g as k in front of a word boundary.

On the screen, the output from the compiler is shown in a tabular format. The columns are labeled with the symbols of the minimal alphabet chosen by the compiler and the other symbols they represent are shown by listing the equivalence classes for the rule:

```
Harmony
    a e g A:a A:e
0: 1 0 0      0
1: 1 0 1  1

Equivalence classes:
  (a) (e #:0) (g g:k)
  (A:a) (A:e)
```

```
Devoicing
    a g g:k #:0
0: 0 2  1   0
1.          0
2: 0 2  1

Equivalence classes:
  (a e A:a A:e) (g)
  (g:k) (#:0)
```

When the compiler writes out the results of the compilation into a file (section 2.6) in LISP format, the automata and the alphabet are listed separately:

```
(AUTOMATA
  (("Harmony" 2 5)
    (0 T 1 0 0 - 0)
    (1 T 1 0 1 1 -))
  (("Devoicing" 3 4)
    (0 T 0 2 1 0)
    (1 NIL - - - 0)
    (2 T 0 2 1 -)))
(ALIGNMENTS
  (a a 1 1) (e e 2 1) (g g 3 2) (A a 4 1)
  (A e 5 1) (g k 3 3) (# 0 2 4))
```

In the LISP format, each state is represented by a list that resembles the corresponding row in the tabular representation. The difference is that

blanks are converted to dashes ( = no transition) and the finality of the state is encoded separately (T = final, NIL = nonfinal).
The list of alignments consists of entries of the form

```
(<lexical symbol> <surface symbol> . <columns>)
```

in which <columns> is a list that picks out the position in the states of each automaton that contains the transition (if any) for the symbol pair in question. For example, the entry for the pair A:a is (A a 4 1) because the transitions for A:a are in the fourth colum in the first automaton ("Harmony") and in the first column of the second automaton ("Devoicing").

To enable the rule tester to use the automata for generation the compiler also constructs a hash table that maps every lexical symbol to its potential surface counterparts:

```
g (g k)
e (e)
A (a e)
a (a)
# (0)
```

A similar map is built for recognition. These tables are not saved because they can easily be reconstructed from the alignments.

## Acknowledgments

The research of the first author has been supported in part by a grant from the Nippon Telegraph and Telephone Company to SRI International.

## References

Johnson, C. Douglas 1972 *Formal Aspects of Phonological Description.* (Monographs on linguistic analysis, no. 3.) The Hague: Mouton.

Kaplan, Ronald M. and Kay, Martin. 1985 Phonological rules and finite-state transducers. Manuscript.

Karttunen, Lauri 1983 KIMMO: A general morphological processor. Texas Linguistic Forum 22 165-186.

Kinnunen Maarit. 1986 Morfologisten sääntöjen kääntäminen äärellisiksi automaateiksi. (Translating morphological rules to finite-state automata.) Masters thesis. Department of Computer Science. University of Helsinki.

Koskenniemi, Kimmo. 1983 *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production.* Department of General Linguistics. University of Helsinki. Publication No. 11.

Koskenniemi, Kimmo. 1986 Compilation of automata from morphological two-level rules. *Papers from the Fifth Scandinavian Conference of Computational Linguistics.* Helsinki, December 11-12, 1985. Department of General Linguistics. University of Helsinki. Publication No. 15. 143-149.

Ritchie, G.D., Black, A.W., Pulman, S.G., and Russell, G.J. 1985. *Dictionary and Morphological Analyser.* [PROTOTYPE] User Guide: Version 1.12. Department of Artificial Intelligence. University of Edinburgh.

## Appendix 1

This small grammar is included as an example of the two-level rule formalism. To aid the reader to interpret the formalism, we have added some commentary in square brackets.

### ENGLISH.RULES

[The beginning of the file consists of declarations that define the alphabet and some auxiliary symbols that are used to state the rules more concisely.]

```
Alphabet

   a b c d e f g h i j k l m n o p q r s t u v
   w x y z ' -:0 ':0                          ;
```

[All the symbols in this list are interpreted as pairs consisting of a lexical character and its surface realization but only one needs to be listed when the two are identical. The symbol a, for example, actually means a:a 'lexical a realized as a surface a.' The colon is used as a separator. The symbols - and ' belong only to the lexical alphabet, 0 only to the surface alphabet. 0 represents the empty string. The full list of possible *lex:surf* pairs also contains all the pairs that appear in the rules. For example, besides the x:x pair that comes from the definition of the alphabet, the pair x:c is also valid because it is licensed by a rule. We use the symbol - to mark morpheme boundaries and ' as a marker for non-initial stress, for example *in'fer* vs *enter*. The symbol # is added automatically by the system to the lexical alphabet to mark word boundaries; ]

```
Sets

   V = a e i o u ;
   C = b c d f g h k l m n p q r s t v w x z ;
   G = b d f g l m n p r s t ;
   SoftC = g c ;
   NonSoftC = b d f h j k l m n p q r s t v w x z ;
   NonVelarC = b d f l m n p r s t v w z ;
   NonFC = b c d g h k l m n p q r s t v w x z ;
```

[The members of a set must all be alphabetic symbols. The interpretation of a set name in a rule is analogous to the interpretation of an alphabetic symbol. For example, the symbol V in a rule stands for all the valid pairs that contain a lexical vowel and a surface vowel.]

```
Definitions
   Sib = [s|c| h | s | x: | z ;
   NonVelar = V | NonVelarC ;
   StressedSyllNucl = [': | #: ] [C* | q u| V  ;
```

[Definitions associate a name with a regular expression. The vertical bar | indicates disjunction; square brackets are used for grouping. The definition for Sib, for example, is the set of expressions containing sh, ch, s, x: and z. What is x:? It is any valid pair with x on the lexical side; in this grammar that stands for x:x and x:c. The former pair is introduced in the lexicon, the latter in the "X-to-C" rule. All symbols of this type, for example, e:, -:, :i, :v, and y: below have similar *implicit definitions*.]

```
Rules

"I-to-Y"
 i:y <=> _ e: -: i ;
```

[Lexical i is realized as surface y always and only when it is followed by a lexical e, morpheme boundary and a lexical i realized as surface i. (We could also say i: or :i without changing the effect of the rule because no rule allows i to be realized as anything but i following the boundary.) This rule accounts for the alternation in *die-ing → dying*.]

```
"X-to-C"
  x:c <=> NonFCons :i _ -: s ;
```

[Lexical x is realized as surface c always and only when it is preceded by a NonFCons :i sequence and followed by a morpheme boundary and s. The rule accounts for the contrast *matrix-s → matrices*, *vertex-s →*

*vertices* vs. *suffix-s → suffixes*. Note that we refer to the surface i so that the rule applies both to *vertex* and *matrix*.]

```
"E-to-I"
 e:i =>  _ x: -:  ;
```

[Lexial e is realized as surface i only if it is followed by a lexical x and a - boundary. Note that this rule does not require that e be realized as i in this environment allowing words like *vertex* to have two plurals: *vertexes* and *vertices*.]

```
"Epenthesis"
  -:e <=>  Sib | :v | C y:  _ s ;
```

[Morpheme boundary is realized as surface e always and only when it is preceded either by a Sib sequence (see the section *Definitions*) or a surface v or a C y: sequence. C stands for any pair whose lexical and surface characters are in C. This rule accounts for pairs such as: *church-s → churches*, *fox-s → foxes*, *wolf-s → wolves*, *spy-s → spies*. Note that this rule does not specify what the lexical counterpart of the v in *wolves* is or what the y in *spy-s* is realized as. Other rules take care of this.]

```
"Y-to-I"
 y:i <=>    C _ [-:e  | -: e ] ;
          V C+ _ -: C ;
```

[This rule has two environments. Lexical y is realized as i always and only when it either is preceded a C and followed by a surface e, perhaps separated by a boundary, or when the lexical y is preceded by a sequence consisting of a vowel and at least one consonant and followed by a morpheme boundary and C. The first alternative accounts for the alternations *spy-s → spies*, *spy-ed → spied*, *lobby-ed → lobbied*, *dry-er → drier* and the lack of it in *shy-ness*. The second part is for *happy-ly → happily*, *happy-ness → happiness*. In its present form the rule is not satisfactory because it produces realizations such as *shy-er → shier*

instead of the correct comparative form *shyer*. We leave it as an exercise for the reader to correct the error.]

```
"F-to-V"
  f:v <=>  [h i e | V a | NonVelar V^1 ] _ -: s ;
                                        V _ e -: s  ;
```

[Lexical f is realized as surface v always and only when it is in either of the two contexts. The first context accounts for realizations like *thief-s → thieves*, *grief-s → griefs*, *loaf-s → loaves*, *wolf-s → wolves*, *scarf-s → scarfs*, which show the alternation, and pairs such as *roof-s → roofs* and *gulf-s → gulfs* where f remains f. The second context accounts for *life-s → lives*.]

```
"Elision"
  e:0 <=>  NonSoftC _ -: V ;
              SoftC _ -: [i | e ] ;
                 i: _ -: i ;
                  V _ -: e ;
```

[Lexical e is realized as the empty string always and only when it occurs in any of the four contexts; examples are given below:

| | |
|---|---|
| `NonSoftC _ -: V` | *move-ed → moved*, *move-ing → moving*, *move-able → movable*, |
| `SofC _ -: [i \| e]` | *change-ed → changed*, *space-ing → spacing*, *change-able → changeable*, |
| `i: _ -: i` | *die-ing → dying*, |
| `V_ -: e` | *tiptoe-ed → tiptoed*.] |

```
"Gemination"
  -:St <=> [': | #: ] C* V :St _ V ; where St in G ;
```

[For every surface St in the set G, the morpheme boundary is realized as St always and only when - is preceded by a sequence consisting of a stress mark (') or a word boundary (#) followed by any number of consonants, a vowel and a surface St and followed by a vowel. This rule accounts for the pairs like *big-er → bigger*, *in'fer-ed → inferred*, and *enter-ed → entered*.]

## Appendix 2

### Consonant Gradation in Finnish

Consonant gradation refers to a set of alternations that involves voiceless stops k, p, and t in Finnish. Historically, the alternate forms occur in closed syllables but in modern Finnish the conditions are in part morphological. The following set of eight gradation rules was constructed to test the capabilities of the TWOL compiler. These rules abstract away from reality in that the alternation is here controlled solely by the phonological environment. We also ignore the fact that many words are exempted from gradation. Except for these simplifications, the rules give a complete account of the phenomenon. In accordance with standard Finnish orthography, a velar nasal is denoted by n in front of a k and by ng as a geminate.

The set of possible lexical:surface correspondences involving k, p, and t are listed below:

```
k:k k:0 k:g k:j k:'
p:p p:v p:m
t:t t:d t:n t:l t:r
```

The surface g in the pair k:g represents a part of a geminate velar nasal. The surface apostrophe is an orthographic marker for syllable boundary. The following table illustrates the various realizations of k, p, and t in the weak grade. The examples are all singular genitive forms of nouns.

| Lexical form | Surface form | Gloss |
|---|---|---|
| tikkan | tikan | woodpecker |
| loppun | lopun | end |
| katton | katon | roof |
| | | |
| sikan | sian | pig |
| papun | pavun | bean |
| sotan | sodan | war |

| | | | |
|---|---|---|---|
| vankin | vangin | prisoner | |
| kumpun | kummun | hill | |
| rantan | rannan | shore | |
| tiukun | tiu'un | chime | |
| leukan | leuan | jaw | |
| pukun | puvun | dress | |
| jalkan | jalan | foot | |
| kurken | kurjen | crane | |
| sylken | syljen | spit | |
| iltan | illan | evening | |
| partan | parran | beard | |
| aikan | ajan | time | (exception) |
| poikan | pojan | boy | (exception) |

A complete TWOL grammar for this set of data is given below. Comments are enclosed in square brackets. These rules give rise to several conflicts that the new version of the compiler can automatically resolve. For example, "Consonant gradation" and "Geminate gradation" conflict in the => direction with respect to the context for the pair k:0. The *Elsewhere Principle* needs to be invoked in five cases to resolve conflicts in the <= direction that occur between the general rule "Consonant gradation" and three four more specific gradation rules: "Gradation of k to '" and "Gradation of k to v," "Gradation of k to j," and "Gradation of t to a liquid." The last of the eight rules (in conjunction with the first rule) takes care of the exceptional gradation of *aika* 'time' and *poika* 'boy.'

## GRADATION.RULES

[The apostrophe plays the role of a silent consonant. A lexical ' is realized as the empty string, the surface ' is one of the realizations of k in the weak grade.]

```
Alphabet

  a b c d e f g h i j k l m n o p q r s t u v x y
    { } w z ':0 :' ;

Sets
  Cons = b c d f g h j k l m n ng p r s t v x z ' ;
  Vowel = a e i o u y { } ;
  VclessStop = k p t ;
  Liquid = l r ;
  HighLabial = u y ;
```

[In all the gradation rules the affected stop is in the beginning of a closed lexical syllable. Note that the definition of *ClosedOffset* does not specify how these consonants are realized.]

```
Definitions

   ClosedOffset =  Cons: [Cons: | #: ]                    ;
   ClosedCoda = Vowel ClosedOffset                        ;

Rules
```

[A single voiceless stop is realized by its corresponding weak grade in the beginning of a closed syllable when preceded by an h or a liquid or a lexical vowel. We stipulate a lexical vowel here in order to allow the rule to apply even if the vowel in question is realized as consonant—see the last rule. Because "closeness" is defined with respect to lexical environment, "Consonant gradation" applies to the lexical p in *aputtom* → *avuton* 'helpless' although its surface counterpart is not in the beginning of a closed surface syllable. Other examples: *sikan* → *sian* 'pig,' *papun* → *pavun* 'bean,' *sotan* → *sodan* 'war,' *leukan* → *leuan* 'jaw,' *jalkan* → *jalan* 'foot,']

```
"Consonant gradation"
   Cx:Cy <=>  h | Liquid | Vowel: _ ClosedCoda ;
                          where Cx in VclessStop
                                Cy in (0 v d)
                             matched ;
```

[A voiceless geminate is shortened in the beginning of a closed syllable: *tikkan* → *tikan*, 'woodpecker', *loppun* → *lopun* 'end', *katton* → *katon* 'roof.']

```
"Geminate gradation"
   Cx:0 <=>  Cx _ ClosedCoda ; where Cx in VclessStop ;
```

[After a nasal, the weak grades of k, p, t assimilate to the preceding nasal: *vankin* → *vangin* 'prisoner,' *kumpun* → *kummun* 'hill,' *rantan* → *rannan* 'shore.']

```
"Gradation after nasals"
   Cx:Cy <=> Cz _ ClosedCoda ; where Cx in (k p t)
                                     Cy in (g m n)
                                     Cz in (n m n)
                                  matched ;
```

[The weak grade of k is ' between two identical vowels when another vowel precedes: *tiukun* → *tiu'un* 'chime,' *raakan* → *raa'an* 'raw.']

```
"Gradation of k to '"
   k:' <=> Vowel Vx _ Vx ClosedOffset ; where Vx in Vowel ;
```

[The weak grade of of k is v between two high labial vowels (u or y): *pukun* → *puvun* 'dress.']

```
"Gradation of k to v"
   k:v <=> Cons :HighLabial _ :HighLabial ClosedOffset ;
```

[The weak grade of k is j after a liquid or h when followed by a lexical e or a surface i: *kurken* → *kurjen* 'crane,' *sylken* → *syljen* 'spit.' Note that this excludes cases such as **aljetaan* 'let's begin' in which the surface e

is realization of a lexical a; instead we get *aletaan* by the main gradation rule.]

```
"Gradation of k to j"
   k:j <=> Liquid | h _ [:i | e:] ClosedOffset ;
```

[The weak grade of t assimilates to the preceding liquid: *kultan* → *kullan* 'gold', *partan* → *parran* 'beard.']

```
"Gradation of t to a liquid"
   t:Cx  <=>  Cx _ ClosedCoda  ; where Cx in Liquid ;
```

[This rule takes care of two exceptional words: *poikan* → *pojan* 'boy' and *aikan* → *ajan* 'time.' Without this rule, these lexical forms would be realized as **poian*, **aian*. Note that the left environment of the "Consonant gradation" rule must only require a lexical vowel so that k can be realized as an empty string here.]

```
"Weak grade of poika, aika"
   i:j <=> #: [p o | a ] _ k: a ClosedOffset ;
```