

# A1 Understanding two-level rules



[ [Guide contents](#) | [Appendix A contents](#) | [Next section: A2 Implementing two-level rules as finite state machines](#) ]

---

## [A1.1 Generative rules and two-level rules](#)

## [A1.2 Correspondences and feasible pairs](#)

## [A1.3 Two-level rule notation](#)

## [A1.4 Using character subsets in rules](#)

## [A1.5 The four rule types](#)

## [A1.6 Expressing complex environments](#)

## [A1.7 Understanding two-level environments](#)

## [Figure A1 Diagnostic properties of the four rule types](#)

---

Section [A1](#) describes the form and meaning of two-level rules: how they differ from the rules of generative phonology, their notation, the four types of two-level rules, and the concept of two-level environments.

## A1.1 Generative rules and two-level rules

Two-level rules are similar to the rules of standard generative phonology, but differ in several crucial ways. Rule [R1](#) is an example of a generative rule.

R1             $t \rightarrow c / \_\_ i$

Rule [R2](#) is the analogous two-level rule.

R2             $t:c \Rightarrow \_\_ i$

The difference between the two rule formalisms is not just notational; rather their meanings are different.

Generative rules have three main characteristics. First, they are transformational rules---they transform or rewrite one symbol into another symbol. Rule [R1](#) states that  $t$  becomes (is changed into)  $c$  when it precedes  $i$ . After rule [R1](#) rewrites  $t$  as  $c$ ,  $t$  no longer "exists." Second, sequentially applied generative rules convert underlying forms to surface forms via any number of intermediate levels of representation; that is, the application of each rule results in the creation of a new intermediate level of representation. Third, generative rules are unidirectional---they can only convert underlying form to surface form, not vice versa.

In contrast, two-level rules are declarative; they state that certain correspondences hold between a lexical (that is, underlying) form and its surface form. Rule [R2](#) states that lexical  $t$  corresponds to surface  $c$  before  $i$ ; it is not changed into  $c$ , and it still exists after the rule is applied. Because two-level rules

express a correspondence rather than rewrite symbols, they apply in parallel rather than sequentially. Thus no intermediate levels of representation are created as artifacts of a rewriting process. Only the lexical and surface levels are allowed. It is this aspect of their nature that is emphasized by the name "two-level" rules. Furthermore, because the two-level model is defined as a set of correspondences between lexical and surface representation, two-level rules are bidirectional. Given a lexical form, PC-KIMMO will return the surface form. Given a surface form, PC-KIMMO will return the lexical form.

## A1.2 Correspondences and feasible pairs

Two-level rules treat each word as a correspondence between its lexical representation (LR) and its surface representation (SR). For example, consider the lexical form *tati* and its surface form *taci*:

```
LR:  t a t i
SR:  t a c i
```

Each pair of lexical and surface characters is a **correspondence pair** or simply correspondence. We write a correspondence with the notation **lexical-character:surface-character**, for instance *t:t*, *a:a*, and *t:c*. There must be an exact one-to-one correspondence between the characters of the lexical form and the characters of the surface form.

There are two types of correspondences exemplified in these forms: **default** correspondences such as *t:t* and *a:a*, and **special** correspondences such as *t:c*. The sum of the default and special correspondences makes up the set of **feasible pairs** sanctioned by the description. In other words, all feasible pairs must be explicitly declared in the description, either as default or as special correspondences.

## A1.3 Two-level rule notation

Looking again at rule [R2](#) (from section [A1.1](#)), we see that a two-level rule is made up of three parts: the **correspondence**, the **rule operator**, and the **environment** or **context**. The first part of rule [R2](#) is the correspondence *t:c*. It specifies a lexical *t* that corresponds to a surface *c*.

The second part of rule [R2](#) is the rule operator *=>*. Although this operator is shaped like an arrow, its meaning is quite different from the rewriting arrow of generative rules (for instance, rule [R1](#)). The rule operator specifies the relationship between the correspondence and the environment in which it occurs. There are four operators: *=>*, *<=*, *<=>*, and */<=*. The semantics of the rule operators are discussed in section [A1.5](#) in detail, but briefly they mean the following:

- =>*  
the correspondence only occurs in the environment
- <=*  
the correspondence always occurs in the environment
- <=>*  
the correspondence always and only occurs in the environment
- /<=*  
the correspondence never occurs in the environment

The third part of rule [R2](#) is the environment or context, written as     *i*. It specifies the phonological environment in which the correspondence is found. As in standard phonological notation, an underline, called an environment line, indicates the position of the correspondence in the environment.

The environment of rule [R2](#) contains a notational shorthand. In its full form the rule is written this way:

```
R3      t:c => ____ i:i
```

Rule [R3](#) states that a lexical  $\mathfrak{t}$  corresponds to (or is realized as) a surface  $\mathfrak{c}$  only when it precedes a lexical  $\mathfrak{i}$  that corresponds to (or is realized as) a surface  $\mathfrak{i}$ . If the lexical and surface characters of a correspondence pair are identical, the correspondence can be written as a single character. Thus rule [R2](#) is equivalent to rule [R3](#).

Rule [R3](#) illustrates that environments are also stated in terms of two-level correspondences. We cannot just say that  $\mathfrak{t}$  corresponds to  $\mathfrak{c}$  before  $\mathfrak{i}$ ; we must specify whether it is a lexical or surface  $\mathfrak{i}$ . This means that a two-level rule has access to both the lexical and surface environments (see section [A1.7](#) below on two-level environments). In contrast, rules in generative phonology can refer only to the environment of the local level of representation, which is often an intermediate level. To emphasize the two-level nature of rule [R3](#), we can write it on multiple lines:

$$\begin{array}{rcl} \mathfrak{t} & & \text{---} \mathfrak{i} \\ \Rightarrow & & \\ \mathfrak{c} & & \text{---} \mathfrak{i} \end{array}$$

The environment of a rule can also make use of a special "wildcard" or ANY symbol (written here as @) that stands for any alphabetic character (as qualified below). For example,

R4             $\mathfrak{t}:\mathfrak{c} \Rightarrow \text{---} \mathfrak{i}:@$

Rule [R4](#) states that  $\mathfrak{t}$  corresponds to  $\mathfrak{c}$  before any feasible pair whose lexical character is  $\mathfrak{i}$  (that is, a lexical  $\mathfrak{i}$  regardless of how it is realized). In the example above, this could include both the default correspondence  $\mathfrak{i}:\mathfrak{i}$  and any special correspondences such as  $\mathfrak{i}:\mathfrak{y}$ . Note carefully that, when used in a rule, the ANY symbol does not really mean any alphabetic character, rather it means any alphabetic character that constitutes a feasible pair with the other character in the correspondence (see section [A3.1](#)). As a notational shorthand, a correspondence such as  $\mathfrak{i}:@$  is simplified to just the lexical character followed by a colon, that is,  $\mathfrak{i}:$ . The ANY symbol can also be used on the lexical side of a correspondence:

R5             $\mathfrak{t}:\mathfrak{c} \Rightarrow \text{---} @:\mathfrak{i}$

Rule [R5](#) states that  $\mathfrak{t}$  corresponds to  $\mathfrak{c}$  before any feasible pair whose surface character is  $\mathfrak{i}$  (that is, a surface  $\mathfrak{i}$  regardless of what lexical character it realizes). This notation is simplified to just a colon followed by the surface character, that is,  $:\mathfrak{i}$ . It should be noted that the ANY symbol can only be used in the environment part of a rule, not in the correspondence part.

Another important characteristic of two-level rules is that they require a one-to-one correspondence between the characters of the lexical form and the characters of the surface form. That is, there must be an equal number of characters in both lexical and surface forms, and each lexical character must map to exactly one surface character, and vice versa. Phonological processes that delete or insert characters are expressed in the two-level model as correspondences with the NULL symbol, written here as 0 (zero). In the following forms, a lexical + (morpheme boundary) corresponds to a surface 0 (that is, it is deleted) and a surface ' (stress mark) corresponds to a lexical 0 (that is, it is inserted).

LR:    0 t a t + i  
SR:    ' t a c 0 i

The NULL symbol is used only internally by the rules; it is not printed in output forms and does not need to be written in input forms. PC-KIMMO will accept the lexical input form  $\mathfrak{t}\mathfrak{a}\mathfrak{t}+\mathfrak{i}$  and return the surface output form  $\mathfrak{'t}\mathfrak{a}\mathfrak{t}\mathfrak{i}$ . The NULL symbol can be used both in the environment and the correspondence parts of a rule.

Another special symbol is the BOUNDARY symbol, written here as #. It indicates a word boundary, either initial or final. It can be used only in the environment of a rule and can only correspond to another BOUNDARY symbol, that is,  $\#:\#$ .

## A1.4 Using character subsets in rules

In generative phonology, classes of characters are referred to using distinctive features; for example, vowels are referred to using the cluster of features [+syllabic, +sonorant, -consonantal]. PC-KIMMO does not support distinctive features. Instead, classes of characters are enumerated in lists that are given single-word names (one or more characters, no spaces).

These character classes are defined in SUBSET statements in the rules file (see section [A4.3](#)). For example, the following declarations define *c* as the set of consonants, *v* as the set of vowels, *s* as the set of stops, and *NAS* as the set of nasals:

```
SUBSET C      p t k b d g m n ng s l r w y
SUBSET V      i e a o u
SUBSET S      p t k b d g
SUBSET NAS    m n ng
```

Suppose that after writing rule [R2](#) above (section [A1.1](#)), further data show that all the alveolar obstruents are palatalized before high, front vowels. For example,

```
LR:  ati  ade  asi  aze
SR:  aci  aje  aSi  aZe
```

Rather than write a separate rule for each correspondence, we will define subset *D* for alveolar obstruents, subset *P* for their palatalized counterparts, and subset *Vhf* for the high, front vowels:

```
SUBSET D      t d s z
SUBSET P      c j S Z
SUBSET Vhf    i e
```

Rule [R2](#) can now be generalized by writing it with subsets:

```
R6      D:P => ____ Vhf
```

## A1.5 The four rule types

The rule operator specifies the logical relation between the correspondence and the environment of a two-level rule. The rule operators are roughly equivalent to the conditional or implicative operators of formal logic. Rule [R7](#) (rule [R2](#) above) is written with the rule operator  $\Rightarrow$ .

```
R7      t:c => ____ i
```

The  $\Rightarrow$  operator means "only but not always." Rule [R7](#) states that lexical *t* corresponds to surface *c* only preceding *i*, but not necessarily always in that environment. Thus other realizations of lexical *t* may be found in that context, including *t:t*. In logical terms, the  $\Rightarrow$  operator means that the correspondence implies the context, but the context does not necessarily imply the correspondence. To state it negatively, rule [R7](#) prohibits the occurrence of the correspondence *t:c* everywhere except preceding *i*.

The  $\Rightarrow$  rule is roughly equivalent to an optional rule in generative phonology, and is typically used in cases of so-called free variation. Rule [R7](#) would be used if the occurrence of *t* and *c* freely varies before *i*. Given the lexical input form *tati* and rule [R7](#), the PC-KIMMO generator will produce both surface forms *taci* and *tati*.

Rule [R8](#) is the same as rule [R7](#) except that it is written with the rule operator  $\Leftarrow$ .

```
R8      t:c <= ____ i
```

The  $\Leftarrow$  operator means "always but not only." Rule [R8](#) states that lexical *t* always (obligatorily) corresponds to surface *c* preceding *i*, but not necessarily only in that environment. Thus *t:c* is permitted to occur in other contexts. In logical terms, the  $\Leftarrow$  operator means that the context implies the correspondence, but the correspondence does not necessarily imply the context. To state it negatively, if *t:¬c* (where  $\neg c$  means the logical negation of *c*) means the correspondence of lexical *t* to surface not-*c* (that is, anything except *c*), then rule [R8](#) prohibits the occurrence of *t:¬c* in the specified context.

The  $\leq$  rule is roughly equivalent to an obligatory rule in generative phonology. It is used in cases where a correspondence is obligatory in one environment but also occurs in some other environment as specified by another rule. Given the lexical input form  $tati$  and rule [R8](#), the PC-KIMMO generator will produce both surface forms  $taci$  and  $caci$ , unless constrained by some other rule.

Rule [R9](#) is again the same, except that it is written with the rule operator  $\leq\Rightarrow$ .

R9             $t:c \leq\Rightarrow \_\_\_ i$

The  $\leq\Rightarrow$  operator is the combination of the operators  $\leq$  and  $\Rightarrow$  and means "always and only." Rule [R9](#) states that lexical  $t$  corresponds to surface  $c$  always and only preceding  $i$ . The  $\leq\Rightarrow$  rule is used when a correspondence obligatorily occurs in a given environment (compare the  $\leq$  operator) and in no other environment (compare the  $\Rightarrow$  operator). It is equivalent to the biconditional logical operator and means that the correspondence is allowed if and only if it is found in the specified context. Given the lexical form  $tati$  and rule [R9](#), the PC-KIMMO generator will return only the surface form  $taci$ . Thus rule [R9](#) is equivalent to the combination of rules [R7](#) and [R8](#). It is up to the analyst to choose between writing separate  $\leq$  and  $\Rightarrow$  rules or collapsing them into one  $\leq\Rightarrow$  rule.

Rule [R10](#) is written with the rule operator  $/\leq$ .

R10             $t:c /\leq \_\_\_ i:\hat{e}$

The  $/\leq$  operator means "never." It means that the correspondence specified by the rule is prohibited from occurring in the specified context. A  $/\leq$  rule is usually used to cover "exceptions" to a more general rule. Rule [R10](#) states that lexical  $t$  cannot correspond to surface  $c$  preceding  $i:\hat{e}$ . Given the lexical form  $tati$ , rule [R10](#), and a rule sanctioning a  $i:\hat{e}$  correspondence, the PC-KIMMO generator will allow the surface forms  $tat\hat{e}$  and  $cat\hat{e}$  but disallow  $tac\hat{e}$  or  $cac\hat{e}$ . As the operator symbol suggests, the  $/\leq$  operator is similar to the  $\leq$  operator in that it does not prohibit the correspondence from occurring in other environments.

Figure [A1](#) summarizes the diagnostic properties of rules [R7](#) through [R10](#). For more on the semantics of the four rule types, see section [A3.3](#).

**Figure A1** Diagnostic properties of the four rule types

Rules 7-10	Is $t:c$ allowed preceding $i$ ?	Is preceding $i$ the only environment in which $t:c$ is allowed?	Must $t$ always correspond to $c$ before $i$ ?
$t:c \Rightarrow \_\_\_ i$	yes	yes	no
$t:c \leq \_\_\_ i$	yes	no	yes
$t:c \leq\Rightarrow \_\_\_ i$	yes	yes	yes
$t:c /\leq \_\_\_ i$	no	—	—

## A1.6 Expressing complex environments

Several notational conventions exist that can be used to build complex environment expressions. These involve optional elements, repeated elements, and alternative elements. As an example we will use a vowel reduction rule, which states that a vowel followed by some number of consonants followed by stress (indicated by  $'$ ) is reduced to schwa ( $\hat{e}$ ). For example,

LR:     $bab'a$      $bamb'a$   
 SR:     $b\hat{e}b'a$      $b\hat{e}mb'a$

Parentheses indicate an optional element. Rule [R11](#) requires either one or two consonants.

R11             $V:\hat{e} \leq\Rightarrow \_\_\_ C(C)'$

Rule [R12](#) requires either zero, one, or two consonants.

R12             $V:\hat{e} \leq => \_\_\_ (C)(C)'$

An asterisk indicates zero or more instances of an element. (The asterisk functions the same as a Kleene-star in regular expressions.) Rule [R13](#) requires either zero, one, or more consonants.

R13             $V:\hat{e} \leq => \_\_\_ C^*'$

Rule [R14](#) requires one or more consonants.

R14             $V:\hat{e} \leq => \_\_\_ CC^*'$

A correspondence may occur in more than one environment. Consider a rule of vowel lengthening whereby the correspondence  $a:\ddot{a}$  (short and long a) occurs in two distinct environments: when it occurs in the syllable preceding stress (pretonic lengthening) and when it occurs in the stressed syllable (tonic lengthening). For example,

LR:     $ladab'ar$

SR:     $lad\ddot{a}b'\ddot{a}r$

Rule [R15](#) expresses pretonic lengthening and rule [R16](#) expresses tonic lengthening.

R15             $a:\ddot{a} \Rightarrow \_\_\_ C'$

R16             $a:\ddot{a} \Rightarrow ' \_\_\_$

Note carefully that a description containing these two rules is self-contradictory. Both rules use the  $\Rightarrow$  operator, which permits the correspondence to occur only in the specified environment. Rule [R15](#) says that  $a:\ddot{a}$  occurs only in a pretonic syllable; rule [R16](#) says that  $a:\ddot{a}$  occurs only in a tonic syllable. Thus the two rules conflict with each other. This type of rule conflict is called an environment conflict (or a  $\Rightarrow$  conflict for short, since it involves  $\Rightarrow$  rules) and is discussed more fully in section [A3.13](#). The conflict between rules [R15](#) and [R16](#) can be resolved by collapsing the two rules into one. In rule [R17](#) the vertical bar indicates disjunction between expressions and the square brackets delimit the disjunctive expressions from the rest of the environment (which in this case is empty).

R17             $a:\ddot{a} \Rightarrow [ \_\_\_ C' \mid ' \_\_\_ ]$

Rule [R17](#) now says, correctly, that the  $a:\ddot{a}$  correspondence is allowed only in either pretonic or tonic position.

Now consider rules [R18](#) and [R19](#), which are the same as rules [R15](#) and [R16](#) except that they are written with the  $\leq =$  operator.

R18             $a:\ddot{a} \leq = \_\_\_ C'$

R19             $a:\ddot{a} \leq = ' \_\_\_$

The  $\leq =$  operator means that the correspondence occurs always (obligatorily) in the environment but not only there. Rules [R18](#) and [R19](#) do not conflict with each other, and so do not have to be collapsed into a single rule. However, if the analyst so chooses, they can be collapsed into rule [R20](#).

R20             $a:\ddot{a} \leq = [ \_\_\_ C' \mid ' \_\_\_ ]$

Given the meanings of the rule types as explained in section [A1.5](#), we have the following choices for writing rules for lengthening. If vowel lengthening occurs only but not always in the specified environments, we must use rule [R17](#). If vowel lengthening occurs always but not only in the specified environments, we must use rule [R20](#) (or rules [R18](#) and [R19](#)). And if vowel lengthening occurs always and only in the specified environment, we must use both rules [R17](#) and [R20](#). If the last case is true, the analyst also has the option of collapsing rules [R17](#) and [R20](#) into one  $\leq = \Rightarrow$  rule, rule [R21](#).

R21             $a:\ddot{a} \leq = \Rightarrow [ \_\_\_ C' \mid ' \_\_\_ ]$

As an example of an exclusive disjunction, consider the situation where the vowel of the ultimate (final) syllable of a word is lengthened unless it is schwa, in which case the vowel of the penultimate (next to final) syllable is lengthened. For example,

Assume these subsets, where  $v$  is the set of short vowels and  $v_{\text{long}}$  is the set of their lengthened counterparts (but  $\hat{e}$  is not lengthened):

Rules [R22](#) and [R23](#) (where # represents word boundary) demonstrate a => conflict (see above and section [A3.13](#) on rule conflicts) and must be collapsed.

R23             $V:V\text{lng} \Rightarrow \underline{\hspace{1cm}} C^*\#$

R24  $V:V\text{lng} \Rightarrow \text{C}^*(\hat{e})\#$

## A1.7 Understanding two-level environments

### Nasal Assimilation:

## Stop Voicing:

These rules relate the lexical sequences  $N_p$ ,  $N_t$ , and  $N_k$  to the surface sequences  $mb$ ,  $nd$ , and  $ngg$ , respectively. To account for these correspondences, generative phonology would require two rules (the following rules account only for labials):



R25            Nasal Assimilation  
               N ---> m / \_\_\_\_ p

R26            Stop Voicing  
               p ---> b / m \_\_\_\_

The rules must apply in this order, since if rule [R26](#) were applied first it would destroy the environment needed for rule [R25](#). The two-level versions of these rules do not need to be ordered; in effect they apply simultaneously:

R27            Nasal Assimilation  
               N:m <=> \_\_\_\_ p:

R28            Stop Voicing  
               p:b <=> :m \_\_\_\_

These rules work because of the careful specification of lexical and surface environments. Rule [R27](#) says that a lexical *N* is realized as a surface *m* preceding a lexical *p*. In this context the notation *p:* (equivalent to *p:@*) stands for the correspondences *p:p* (by default) and *p:b* (from rule [R28](#)). Rule [R28](#) says that a lexical *p* is realized as a surface *b* following a surface *m*. The notation *:m* (equivalent to *@:m*) stands for the correspondences *m:m* (by default) and *N:m* (from rule [R27](#)). Because the two-level model allows rule environments to have access to both the lexical and surface levels, rule ordering and intermediate levels are not needed.

A common error in writing two-level rule environments is to overspecify the environment. Consider this overspecified version of rule [R27](#):

R27a           Overspecified version of Nasal Assimilation  
               N:m <=> \_\_\_\_ p

Even though the symbol *p* seems simpler than *p:*, it actually is more specific, as it stands for the correspondence *p:p* only. Rule [R27a](#) is now in conflict with the voicing rule ([R28](#)), which says that after a surface *m* only the correspondence *p:b* can occur. Conversely, the correspondence *p:b* of rule [R28](#) is in conflict with rule [R27a](#), which allows only *p:p* to occur after a surface *m*.

Now consider another incorrectly specified version of the Nasal Assimilation rule ([R27](#)):

R27b           Overspecified version of Nasal Assimilation  
               N:m <=> \_\_\_\_ p:b

At first this version seems correct, since it is precisely in the environment of *p:b* that we want the rule to apply. The problem is that rule [R27b](#) does not require *N* to be realized as *m* preceding a lexical *p* that is realized as anything other than surface *b*, and the Voicing rule ([R28](#)) does not require *p* to be realized as *b* except when it follows a surface *m*. Assuming that otherwise lexical *N* corresponds to surface *n*, the lexical form *Np* will be realized as both *np* and *mb*. Thus overspecification does not always result in a rule conflict; it may also result in overgeneration. (See also section [A3.12](#) on two-level environments.)

[Up](#) [Next](#) [Previous](#)

[ [Guide contents](#) | [Appendix A contents](#) | [Next section: A2 Implementing two-level rules as finite state machines](#) ]



# A2 Implementing two-level rules as finite state machines

[Up](#) [Next](#) [Previous](#)

[ [Guide contents](#) | [Appendix A contents](#) | [Next section: A3 Compiling two-level rules into state tables](#) | [Previous section: A1 Understanding two-level rules](#) ]

---

## [A2.1 How two-level rules work](#)

## [A2.2 How finite state machines work](#)

## [A2.3 A => rule as a finite state machine](#)

## [A2.4 A <= rule as a finite state machine](#)

## [A2.5 A <=> rule as a finite state machine](#)

## [A2.6 A /<= rule as a finite state machine](#)

## [Figure A2 FSA for language L1](#)

## [Figure A3 FST diagram for the correspondence between languages L1 and L2](#)

## [Figure A4 FST diagram of a => rule](#)

---

Section [A2](#) explains how two-level rules operate, how they can be implemented as finite state machines, and how the four types of two-level rules can be translated into finite state tables.

## A2.1 How two-level rules work

To understand how two-level rules work, consider again rule [R2](#), repeated here as rule [R29](#).

R29             $t:c \Rightarrow \_\_\_ i$

The operator  $\Rightarrow$  in this rule means that lexical  $t$  is realized as surface  $c$  only (but not always) in the environment preceding  $i:i$ .

The correspondence  $t:c$  declared in rule [R29](#) is a special correspondence. A two-level description containing rule [R29](#) must also contain a set of default correspondences, such as  $t:t$ ,  $i:i$ , and so on. The sum of the special and default correspondences are the total set of valid correspondences or feasible pairs that can be used in the description.

If a two-level description containing rule [R29](#) (and all the default correspondences also) is applied to the lexical form  $tati$ , the PC-KIMMO generator would proceed as follows to produce the corresponding surface form.

Beginning with the first character of the input form, it looks to see if there is a correspondence declared for it. Due to rule [R29](#) it will find that lexical  $t$  can correspond to surface  $c$ , so it will begin by positing

that correspondence.

LR:	t	a	t	i
Rule:	29			
SR:	c			

At this point the generator has entered rule [R29](#). For the  $t:c$  correspondence to succeed, the generator must find an  $i:i$  correspondence next. When the generator moves on to the second character of the input word, it finds that it is a lexical  $a$ ; thus rule [R29](#) fails, and the generator must back up, undo what it has done so far, and try to find a different path to success. Backing up to the first character, lexical  $t$ , it tries the default correspondence  $t:t$ .

LR:	t	a	t	i
Rule:				
SR:	t			

The generator now moves on to the second character. No correspondence for lexical  $a$  has been declared other than the default, so the generator posits a surface  $a$ .

LR:	t	a	t	i
Rule:				
SR:	t	a		

Moving on to the third character, the generator again finds a lexical  $t$ , so it enters rule [R29](#) and posits a surface  $c$ .

LR:	t	a	t	i
Rule:			29	
SR:	t	a	c	

Now the generator looks at the fourth character, a lexical  $i$ . This satisfies the environment of rule [R29](#), so it posits a surface  $i$ , and exits rule [R29](#).

LR:	t	a	t	i
Rule:			29	
SR:	t	a	c	i

Since there are no more characters in the lexical form, the generator outputs the surface form  $taci$ . However, the generator is not done yet. It will continue backtracking, trying to find alternative realizations of the lexical form. First it will undo the  $i:i$  correspondence of the last character of the input word, then it will reconsider the third character, lexical  $t$ . Having already tried the correspondence  $t:c$ , it will try the default correspondence  $t:t$ .

LR:	t	a	t	i
Rule:				
SR:	t	a	t	

Now the generator will try the final correspondence  $i:i$  and succeed, since rule [R29](#) does not prohibit  $t:t$  before  $i$  (rather it prohibits  $t:c$  in any environment except before  $i$ ).

LR:	t	a	t	i
Rule:				

SR:    t   a   t   i

All other backtracking paths having failed, the generator quits and outputs the second surface form `tati`.

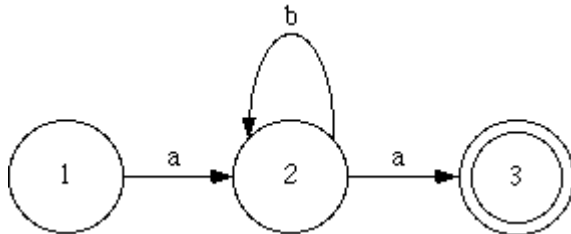
The procedure is essentially the same when two-level rules are used in recognition mode (where a surface form is input and the corresponding lexical forms are output).

## A2.2 How finite state machines work

The basically mechanical procedure for applying two-level rules makes it possible to implement the two-level model on a computer by using a formal language device called a **finite state machine**. The simplest finite state machine is a **finite state automaton** (FSA), which recognizes (or generates) the well-formed strings of a **regular language** (a certain type of formal language---see [Chomsky 1965](#)). While finite state machines are commonplace in computer science and formal language theory (see, for instance, [Hopcroft and Ullman 1979](#)), they may not be familiar to all linguists. They have, however, had their place in the linguistic literature. Most widely known would be chapter 3 of Chomsky's *[Syntactic Structures](#) (1957)* in which finite state grammars are dismissed as an inadequate model for describing natural language syntax. Other notable treatments of finite state automata in the linguistic literature are [Chomsky 1965](#) and [Langendoen 1975](#). A good introduction to finite state automata written for linguists is chapter 16 of [Partee and others 1987](#).

An FSA is composed of states and directed transition arcs. There must minimally be an initial state, a final state, and an arc between them. A successful transition from one state to the next is possible when the next symbol of the input string matches the symbol on the arc connecting the states. For example, consider the regular language L1 consisting of the symbols a and b and the "grammar" *abNa* where  $N \geq 0$ . Well-formed strings or "sentences" in this language include aa, aba, abba, abbbba, and so on. The language L1 is defined by the FSA in figure [A2](#).

**Figure A2** FSA for language L1



State 1 is the initial state and state 3 is the final state (signified by the double circle). States 1 and 2 are both nonfinal states (signified by the single circle). To recognize the string *abba*, proceed as follows:

1. Start at state 1.
2. Input the first symbol of the string and traverse the *a* arc to state 2.
3. Input the second symbol and traverse the *b* arc back to state 2.
4. Input the third symbol and again traverse the *b* arc back to state 2.
5. Input the last symbol and traverse the *a* arc to state 3, which is a final state.

Because the input string is exhausted and the machine is in a final state, we conclude that *abba* is a string in the language L1.

An FSA can also be represented as a **state transition table**. The FSA above is represented as this state table:

a    b

			----
1.	2	0	
2.	3	2	
3:	0	0	

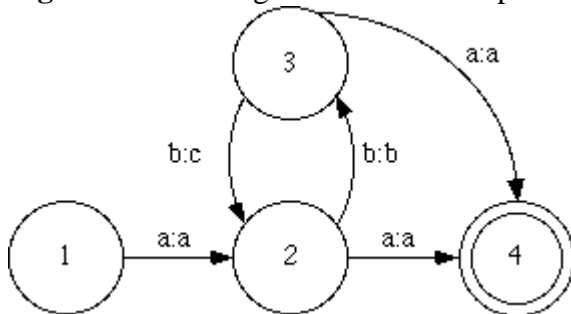
The rows of the table represent the three states of the FSA diagram, with the number of the final state marked with a colon and the numbers of the nonfinal states marked with periods. The columns represent the arcs from one state to another; the symbol labeling each arc in the FSA diagram is placed as a header to the column. The order of the columns in the table has no effect on the operation of the table, but is written to reflect the order of the FSA. Notice that the two *a* arcs in the FSA diagram are represented as a single column labeled *a*. The cells at the intersection of a row and a column indicate which state to make a transition to if the input symbol matches the column header. Zero in a cell indicates that there is no valid transition from that state for that input symbol; in the FSA diagram this is equivalent to having no arc labeled with that input symbol. The machine is said to fail when this happens, thus rejecting the input form. Note that the colon marks state 3 as the only final state. This means that if the machine is in any state but 3 when the input string is exhausted, the string is not accepted. The 0 transitions in state 3 say that once the machine gets to that state, the string is not accepted if there are remaining input symbols.

An FSA operates only on a single input string. A **finite state transducer** (FST) is like an FSA except that it simultaneously operates on two input strings. It recognizes whether the two strings are valid correspondences (or translations) of each other. For example, assume the first input string to an FST is from language L1 above, and the second input string is from language L2, which corresponds to L1 except that every second *b* is *c*. Here is an example correspondence of two strings:

L1: abbbba  
L2: abcbca

Figure A3 shows the FST in diagram form. Note that the only difference from an FSA is that the arcs are labeled with a correspondence pair consisting of a symbol from each of the input languages.

**Figure A3** FST diagram for the correspondence between languages L1 and L2



FSTs can also be represented as tables, the only difference being that the column headers are pairs of symbols, such as *a:a* and *b:c*. For example, the following table is equivalent to the FST diagram in figure A3.

		a	b	b
		a	b	c
		----		
1.	2	0	0	
2.	4	3	0	
3.	4	0	2	
4:	0	0	0	

## A2.3 A => rule as a finite state machine

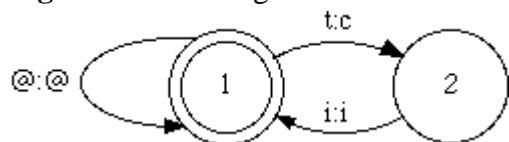
The key insight of PC-KIMMO is that if phonological rules are written as two-level rules, they can be implemented as FSTs running in parallel. In the next four subsections we briefly show how each of the four rule types (see section A1.5) translates into an FST. Detailed procedures for compiling rules into state tables are found in section A3.

Consider rule [R30](#).

R30             $t:c \Rightarrow \_\_\_ i$

In terms of FSTs, this rule defines two "languages" that are translations of each other. The "upper" or lexical language specifies the string  $ti$ ; the "lower" or surface language says that  $ci$  may correspond to it. Note, however, that a two-level rule does not specify the grammar of a full language. Rather it deals with allowed substrings. A possible paraphrase of rule [R30](#) is, "If ever the correspondence  $t:c$  occurs, it must be followed by  $i:i$ ." In other words, if anything other than  $t:c$  occurs, this rule ignores it. This fact must be incorporated into our translation of a two-level rule into a transition diagram, shown in figure [A4](#). In this FST, state 1 is both the initial and only final state. The  $@: @$  arc (where  $@$  is the ANY symbol, see section [A1.3](#)) allows any pairs to pass successfully through the FST except  $t:c$  and  $i:i$ .

**Figure A4** FST diagram of a  $\Rightarrow$  rule



Rule [R30](#), represented as the FST in figure [A4](#), translates into the following state table, labeled [T30](#). (In the remainder of this chapter, each rule and its corresponding state table have the same label number, for instance R30 and T30.) Notice that state 1, the initial state, is a kind of "default" state that ignores everything except the substring crucial to the rule.

T30      Example of a  $\Rightarrow$  table

	t	i	@
	c	i	@
1:	2	1	1
2:	0	1	0

A set of two-level rules is implemented as a system of parallel FSTs that specify all the feasible pairs and where they can occur. A state table is constructed such that the entire set of feasible pairs in the description is partitioned among the column headers with no overlap. That is, each and every feasible pair of the description must belong to one and only one column header. Table [T30](#) specifies the special correspondence  $t:c$  and the environment in which it is allowed. The  $@: @$  column header in table [T30](#) matches all the feasible pairs that are defined by all other FSTs of the system. Thus, with respect to table [T30](#),  $@: @$  does not stand for all feasible pairs; rather it stands for all feasible pairs except  $t:c$  and  $i:i$ .

The default correspondences of the system must be declared in a trivial FST like table [T31](#) (see section [A3.2](#) (step 2) and section [A4.4](#)):

T31      Table of default correspondences}

	p	t	k	a	i	u	@
	p	t	k	a	i	u	@
1:	1	1	1	1	1	1	1

Even this table of default correspondences must include  $@: @$  as a column header. Otherwise it would fail when it encountered a special correspondence such as  $t:c$ . This is due to the fact that all the rules in a two-level description apply in parallel, and for each character in an input string all the rules must succeed, even if vacuously. Now given the lexical form  $tatik$ , tables [T30](#) and [T31](#) will work together to generate the surface forms  $tatik$  and  $tacik$ .

To understand how to represent two-level rules as state tables, we must understand what the two-level rules really mean. (See section [A1.5](#) above.) We tend to think of two-level rules positively, that is, as statements of where the correspondence succeeds. In fact, state tables are failure-driven; they specify where correspondences must fail. It is natural to think of rule [R30](#) as saying that the correspondence  $t:c$  succeeds when it occurs preceding  $i:i$ . But state table [T30](#) actually works because it fails when anything

but  $i:i$  follows  $t:c$  (the zeros in state 2 indicate that the input has failed). This reorientation to thinking of phonological rules as failure-driven is one of the most difficult barriers to overcome in learning to write two-level rules and state tables, but it is the primary key to success with PC-KIMMO.

In summary, the rule type  $L:S \Rightarrow E$  (where  $L:S$  is a lexical:surface correspondence and  $E$  is an environment) positively says that  $L$  is realized as  $S$  only in  $E$ . Negatively it says that  $L$  realized as  $S$  is not allowed in an environment other than  $E$ . The state table for a  $\Rightarrow$  rule must be written so that it forces  $L:S$  to fail in any environment except  $E$ . In logical terms, the  $\Rightarrow$  operator indicates conditionality, such that, if  $L:S$  exists, then it must be in  $E$ .

## A2.4 A $\Leftarrow$ rule as a finite state machine

Rule [R32](#) below is the same as rule [R30](#) except that it is written with the  $\Leftarrow$  operator.

R32             $t:c \Leftarrow \_ i$

This rule says that lexical  $t$  is always realized as surface  $c$  when it occurs before  $i:i$ , but not only before  $i:i$ . Thus the lexical form  $tati$  will successfully match the surface form  $taci$  but not  $tati$ . Note, however, that it would also match  $caci$  since it does not disallow  $t:c$  in any environment. Rather, its function is to disallow  $t:t$  in the environment of a following  $i:i$ . Remembering that state tables are failure-driven, the strategy of writing the state table for rule [R32](#) is to force it to fail if it recognizes the sequence  $t:t i:i$ . This allows the correspondence  $t:c$  to succeed elsewhere. The state table for rule [R32](#) looks like this:

T32        Example of a  $\Leftarrow$  table

	$t$	$t$	$i$	@
	$c$	$t$	$i$	@
	-----			
1:	1	2	1	1
2:	1	2	0	1

In state 1 any occurrences of the pairs  $t:c$ ,  $i:i$ , or any other feasible pairs are allowed without moving from state 1. It is only the correspondence  $t:t$  that forces a transition to state 2, where all feasible pairs succeed except  $i:i$  (the zero in the  $i:i$  column indicates that the input has failed). Notice that state 2 must be a final state; this allows all the correspondences except  $i:i$  to succeed and return to state 1. Also notice that in state 2 the cell under the  $t:t$  column contains a two. This is necessary to allow for the possibility of a  $tt$  sequence in the input form; for instance, table [T32](#) will apply to the lexical form  $tatti$  to produce the surface form  $tatci$ . This phenomenon is called backlooping and is treated in detail in section [T32](#) is potentially overspecified. It is not really the pair  $t:t$  that is disallowed before  $i$ , but rather the pair  $t:\neg c$ ; that is, lexical  $t$  and surface not- $c$  (anything but  $c$ ). Suppose our two-level description also contained a rule that defined a  $t:n$  correspondence. Like  $t:t$ ,  $t:n$  should fail before  $i:i$ . Rather than add another column for  $t:n$  to table [T32](#), we can replace the column header  $t:t$  with  $t:@$ . Given that the more specific correspondence  $t:c$  is also declared in the table,  $t:@$  will match all other valid surface correspondences to lexical  $t$ , such as  $t:t$ ,  $t:n$ , and so on. Here is table [T32](#) revised to use a  $t:@$  column header.

T32a        A  $\Leftarrow$  table

	$t$	$t$	$i$	@
	$c$	@	$i$	@
	-----			
1:	1	2	1	1
2:	1	2	0	1

In summary, the rule type  $L:S \Leftarrow E$  positively says that  $L$  is always realized as  $S$  in  $E$ . Negatively it says that  $L$  realized as any character but  $S$  is not allowed in  $E$ . The state table for a  $\Leftarrow$  rule must be written so that it forces all correspondences of  $L$  with anything but  $S$  to fail in  $E$ . Logically, the  $\Leftarrow$  operator indicates that if  $L$  is in  $E$ , then it must correspond to  $S$ .

## A2.5 A $\Leftarrow\Rightarrow$ rule as a finite state machine

Rule [R33](#) uses the  $\Leftarrow\Rightarrow$  operator and is equivalent to combining rules [R32](#) and [R30](#).

R33             $t:c \Leftarrow\Rightarrow \_\_ i$

The state table for a  $\Leftarrow\Rightarrow$  rule is simply the combination of the  $\Leftarrow$  and the  $\Rightarrow$  tables. The state table for rule [R33](#) is built by combining tables [T32a](#) and [T30](#) to produce table [T33](#) below. The column headers of table [T33](#) are the same as table [T32a](#). States 1 and 2 represent the  $\Leftarrow$  part of the table (corresponding to table [T32a](#)) while states 1 and 3 represent the  $\Rightarrow$  part (corresponding to table [R30](#)).

T33        Example of a  $\Leftarrow\Rightarrow$  table

	t	t	i	@
	c	@	i	@
	-----			
1:	3	2	1	1
2:	3	2	0	1
3:	0	0	1	0

In summary, the rule type  $L:S \Leftarrow\Rightarrow E$  says that  $L$  is always and only realized as  $S$  in  $E$ . It implies that  $L:S$  is obligatory in  $E$  and occurs nowhere else. The state table for a  $\Leftarrow\Rightarrow$  rule must be written so that it forces all correspondences of  $L$  with anything but  $S$  to fail in  $E$ , and forces  $L:S$  to fail in any environment except  $E$ .

## A2.6 A $/\Leftarrow$ rule as a finite state machine

Rule [R34](#) exemplifies the fourth type of rule, the  $/\Leftarrow$  rule.

R34             $t:c /\Leftarrow \_\_ i:\hat{e}$

This rule states that the correspondence  $t:c$  is disallowed when it precedes a lexical  $i$  that is realized as a surface  $\hat{e}$ . As the  $/\Leftarrow$  operator suggests, this rule type shares properties of the  $\Leftarrow$  type rule. It states that the correspondence always fails in the specified environment, but allows it (does not prohibit it) in any other environment. The strategy for building a state table for the  $/\Leftarrow$  rule is to recognize the correspondence in the specified environment and then fail. In table [R34](#) the correspondence  $t:c$  forces a transition to state 2 where the table fails if it encounters the environment  $i:\hat{e}$ , but succeeds otherwise. Notice that like table [T32](#) above for a  $\Leftarrow$  rule, state 2 is final.

T34        Example of a  $/\Leftarrow$  table

	t	i	@
	c	$\hat{e}$	@
	-----		
1:	2	1	1
2:	2	0	1

In summary, the rule type  $L:S /\Leftarrow E$  positively says that  $L$  is never realized as  $S$  in  $E$ . Negatively it says that  $L$  realized as  $S$  is not allowed in  $E$ . Logically, the  $/\Leftarrow$  operator indicates that if  $L$  is in  $E$ , then it must correspond to  $\neg S$ .

---

[Up](#) [Next](#) [Previous](#)

[ [Guide contents](#) | [Appendix A contents](#) | [Next section: A3 Compiling two-level rules into state tables](#) | [Previous section: A1 Understanding two-level rules](#) ]



# A3 Compiling two-level rules into state tables

[Up](#) [Next](#) [Previous](#)

[ [Guide contents](#) | [Appendix A contents](#) | [Next section: A4: Writing the rules file](#) | [Previous section: A2 Implementing two-level rules as finite state machines](#) ]

---

[A3.1 Overview of the rules component](#)

[A3.2 General procedure for compiling rules into tables](#)

[A3.3 Summary of two-level rule semantics](#)

[A3.4 Compiling rules with a right context](#)

[A3.5 Compiling rules with a left context](#)

[A3.6 Compiling rules with both left and right contexts](#)

[A3.7 Compiling insertion rules](#)

[A3.8 Using subsets in state tables](#)

[A3.9 Overlapping column headers and specificity](#)

[A3.10 Expressing word boundary environments](#)

[A3.11 Expressing complex environments in state tables](#)

[A3.12 Expressing two-level environments](#)

[A3.13 Rule conflicts](#)

[A3.14 Comments on the use of => rules](#)

[A3.15 Comments on the use of morpheme boundaries](#)

[A3.16 Expressing phonotactic constraints](#)

[Figure A5 Semantics of two-level rules](#)

[Figure A6 Truth tables for two-level rules](#)

[Figure A7 FST with backlooping](#)

---

Section [A3.1](#) gives an overview of the structure of the rules component. Section [A3.2](#) is a reference summary of the general procedure for compiling (translating) two-level

rules into state transition tables. Detailed examples of how to apply the general procedure are found in sections [A3.4](#) through [A3.7](#). Sections [A3.8](#) through [A3.16](#) treat in detail various topics related to rule compilation, such as subsets, word boundary environments, complex environments, rule conflicts, and phonotactic constraints.

## A3.1 Overview of the rules component

This section discusses alphabetic characters, feasible pairs, using the ANY symbol and subset names, declaring default and special correspondences, mapping feasible pairs to column headers, and applying rules in parallel. Some of these topics, discussed here concisely, are covered in more detail in sections [A3.4](#) through [A3.16](#) and in section [A4](#).

### Alphabetic characters

All characters or symbols used in either lexical or surface forms in the description constitute the alphabet used by the rules. The NULL symbol and the BOUNDARY symbol are also considered alphabetic characters, though in the rules file they are declared separately from the rest of the alphabet (see section [A4.1](#)). The ANY symbol and subset names are not part of the alphabet.

### Feasible pairs

A feasible pair is a specific correspondence between a lexical alphabetic character and a surface alphabetic character. The set of feasible pairs is the set of all such correspondences used in a description. Some of these correspondences are default correspondences, where the lexical and surface characters are identical (for instance t:t and i:i); others are special correspondences, where the surface character differs from the lexical character (for instance t:c and i:ĩ). Each feasible pair, whether a default or special correspondence, must be explicitly declared in a description. This is done by including each feasible pair as a column header in at least one state table. Only column headers consisting of alphabet characters, including the NULL symbol and the BOUNDARY symbol, are considered feasible pairs. Column headers containing the ANY symbol or subset names are ignored for the purpose of declaring feasible pairs.

When the user runs PC-KIMMO and loads a set of rules from a disk file, the column headers of every state table are scanned as they are read in and a list of feasible pairs is compiled. After the rules are loaded, the user can see the entire set of feasible pairs currently in use by the rules component by issuing the **list pairs** command (see section [4.5.5](#)). It should also be remembered that the set of feasible pairs is revised each time one or more rules is turned on or off by means of the **set rules [ on | off ]** command (see section [4.5.6.1](#)).

### Using the ANY symbol and subset names in column headers

Although correspondences that contain the ANY symbol or subset names are not feasible pairs and cannot serve as the correspondence part of a rule, they do occur in the environment part of a rule and therefore appear as column headers in state tables. In order to write correct state tables, the analyst must understand exactly what set of pairs is specified by column headers that contain the ANY symbol or subset names. Although the ANY symbol is said to be a "wildcard" character that can stand for any alphabetic character, its effective meaning relative to a given set of rules is determined by the set of feasible pairs sanctioned by the rules. For example, in a set of rules the correspondence t:@ (where @ has been declared to be the ANY symbol)

does *not* represent all possible correspondences that have *t* as their lexical character and any other member of the alphabet as their surface character; rather, it represents only the feasible pairs that match its pattern, for instance *t:t* and *t:c* if those correspondences are feasible pairs by virtue of appearing as column headers in one or more tables.

Similarly, a subset name is said to stand for a set of alphabetic characters; but its effective meaning relative to a given set of rules is determined by the set of feasible pairs sanctioned by the rules. For example, in a set of rules where the subset *Spal* has been declared to have the members *s*, *x*, and *z*, the correspondence *s:Spal* does *not* represent all possible correspondences that have *s* as their lexical character and one of the members of the subset *Spal* as their surface character; rather, it represents only the feasible pairs that match its pattern, for instance *s:s* and *s:z* if these are the only correspondences involving lexical *s* that have been used as column headers in one or more tables. This means that using the correspondence *s:Spal* as a column header in a rule does *not* implicitly declare as feasible pairs all correspondences that match it. That is, unless the correspondence *s:x* is explicitly declared as a feasible pair somewhere in the set of rules, it is not included in the set of feasible pairs represented by the correspondence *s:Spal*. (For more on using subsets, see section [A3.8](#).)

## Declaring default correspondences

The fact that each valid correspondence used in a description must be explicitly declared as a feasible pair in the rules has consequences for how default correspondences are declared. Since rules are written to express the conditions under which special correspondences occur, default correspondences are not normally included in each rule. Thus, in order to get every feasible pair into a column header of a state table, the rules component must contain a table strictly for the purpose of declaring the default correspondences (see sections [A2.3](#) and [A4.4](#)). A table of default correspondences has only one state (which is a final state), and each transition is back to state 1. The column headers provide the list of default correspondences, with *@:@* (where *@* is the ANY symbol) appended to the end of the list so as not to block the occurrence of special correspondences. It is impossible to include too many correspondences in this list. That is, it would be possible to make the list include every feasible pair and dispense with the final *@:@*. It is possible to underspecify, however. If a feasible pair is left out of the table of default correspondences and does not occur explicitly in any other table, then that correspondence will never be recognized as valid. For the sake of consistency, the table of default correspondences should even include pairs that also appear in the environments of other tables; the redundancy has no effect on the operation of the rules. (For more on writing tables of default correspondences, see section [A4.4](#).)

## Declaring special correspondences

Special correspondences do not need to be gathered into one table as is done with default correspondences since most special correspondences are used as column headers in the rules that apply to them. However, if a set of special correspondences is represented with subsets, it may be necessary to write a separate table declaring the special correspondences as feasible pairs. For example, consider a rule whose correspondence part is *D:P*, where *D* is a subset that contains the alveolar consonants *t*, *d*, and *s* and *P* is a subset that contains the palatalized consonants *c*, *j*, and *S*. The intention of the analyst is that the subset correspondence *D:P* should stand for the feasible pairs *t:c*, *d:j*, and *s:S*. However, in the state table for this rule the column header *D:P* will not match any feasible pairs except those that have been explicitly declared elsewhere. In this situation it is best to write a separate table

where the feasible pairs intended to match the subset correspondence are explicitly used as column headers. For the sake of consistency this should be done even if the pairs do appear in tables elsewhere in the description; the possible redundancy has no effect on the operation of the rules. (For more on using subsets, see section [A3.8.](#))

## Mapping feasible pairs to column headers

As was described above, when a set of rules is loaded and the list of feasible pairs is compiled, the set of pairs that match each column header is determined. In the example above, the pairs `t:c`, `d:j`, and `s:S` should match the column header `D:P`. In this instance the meaning of the column header `D:P` is understood relative to the entire set of rules. However, relative to a single rule, a column header may actually specify only a subset of the pairs that it specifies relative to the entire rule set. This situation arises as follows.

Each state table must be constructed such that every feasible pair is represented by one of its column headers. That is, for each table in a rule set, the entire list of feasible pairs is partitioned among the column headers with no overlap. In a table, each feasible pair belongs to one and only one column header. After loading a set of rules and compiling the list of feasible pairs, PC-KIMMO goes through the set of rules again to interpret the column headers of each table. For each table it scans the list of all the feasible pairs and assigns each one to a column header. If a feasible pair matches more than one column header, it is assigned to the most specific one, where the specificity of a column header is defined as the number of feasible pairs that match it. For example, consider a table that contains both an `s:S` column header and a `D:P` column header, where the feasible pairs that match it include `t:c`, `d:j`, and `s:S`. When PC-KIMMO tries to assign the feasible pair `s:S` to a column header in this table, it finds that it matches both the `s:S` column and the `D:P` column. PC-KIMMO will assign it to the `s:S` column, since it is more specific than the `D:P` column (one versus three pairs that match). This means that, relative to this particular rule, the `D:P` column header represents only two feasible pairs, namely `t:c` and `d:j`. When running PC-KIMMO, the user can see exactly how feasible pairs are assigned to the column headers of a table by using the **show rule** command (see section [4.5.9](#)).

It is possible to construct a state table in which a feasible pair matches multiple column headers that have the same specificity value, thus making it impossible to uniquely assign the pair to a column header. This constitutes an incorrectly written state table. When a rules file containing such a state table is loaded, a warning message is issued alerting the user that two columns have the same specificity. If the user proceeds to analyze forms with the incorrectly written table, the pair will be assigned (arbitrarily) to the leftmost column that it matches. Correct results cannot be assured. (For more on the problem of overlapping column headers, see section [A3.9.](#))

In order to get every feasible pair in the column headers of a table without having to literally specify each pair, a column header of the form `@: @` (where `@` is the ANY symbol) is included in the table. This covers all pairs that are not part of the correspondence and environment of the rule.

## Applying rules in parallel

To understand how the PC-KIMMO rules component works to generate and recognize forms, it must be kept in mind that two-level rules, represented as finite state tables, apply in parallel (or simultaneously). This means that for an input form to be

successfully processed by PC-KIMMO, all of the rules must succeed. In other words, as each character of the input form is processed, it must pass successfully through every rule before the next character can be processed. It is precisely because of PC-KIMMO's parallel rule application that each state table must represent in its column headers the entire set of feasible pairs.

## A3.2 General procedure for compiling rules into tables

This section presents a step-by-step procedure for compiling rules into state tables. The following abbreviations are used in the discussion.

**L**  
a lexical character

**S**  
a surface character

**¬S**  
any surface character but S

**L:S**  
the lexical-to-surface correspondence on left side of rule

**E**  
an environment

**¬E**  
any environment but E

**lc**  
the left context in the environment

**rc**  
the right context in the environment

**0**  
the NULL symbol

**@**  
the ANY symbol

**#**  
the BOUNDARY symbol

1. Make a complete inventory of all the possible lexical-to-surface correspondences found in the data. From this, compile a list of all the symbols used as lexical and surface characters, including the NULL symbol and the BOUNDARY symbol. This full list is the alphabet used by the rules. The rules will also use the ANY symbol and subset names.
2. Declare all the default correspondences required by the description. This is done by writing one or more tables that contain only and all the default correspondences (see section [A4.4](#)).
3. For each special correspondence (L:S), write down your hypothesis about the environment (E) in which it occurs. (The environment may be disjunctive; that

is, E1 or E2 or E3.) For each correspondence, answer the following two questions:

- a. Is E the only environment in which L:S is allowed?
- b. Must L always be realized as S in E?

There are four possible outcomes. Depending on the outcome, do one of the following:

- a. If *a* is *yes* and *b* is *no*, posit the rule  $L:S \Rightarrow E$  and proceed to [step 4](#).
- b. If *a* is *no* and *b* is *yes*, posit the rule  $L:S \Leftarrow E$  and proceed to [step 5](#).
- c. If both *a* and *b* are *yes*, posit the rule  $L:S \Leftrightarrow E$  and proceed to [step 6](#).
- d. If neither is *yes*, find the other environments in which L:S is allowed, combine these into a single disjunctive environment, and go through [step 3](#) again.

It is also possible that it is easier to express the constraint on L:S in terms of the environment in which it is prohibited. In this case, posit the rule  $L:S / \Leftarrow E$  and proceed to [step 7](#). If L:S contains subset names, it may be necessary to write a separate table to declare as feasible pairs the correspondences that L:S is intended to represent (see sections [A3.8](#) and [A4.4](#)).

4. Compile each  $\Rightarrow$  rule. The rule  $L:S \Rightarrow lc\_rc$  can be paraphrased as "the expression  $lc\ L:S\ rc$  is allowed, but L:S in any other context is not allowed." The strategy for compiling a  $\Rightarrow$  rule to a state table is to construct a table that recognizes the sequence  $lc\ L:S\ rc$ , forbids any other occurrence of L:S, and permits any other correspondences to occur anywhere. The steps in building the table are as follows:
  - a. Make a list of column headers for the table by writing down all the correspondences used in the expression  $lc\ L:S\ rc$  (including correspondences with @ and subset names). Add @:@ to the end of the list.
  - b. Beginning with state 1, add states (rows) and fill in the state transitions in the appropriate cells in the table to recognize the expression  $lc\ L:S\ rc$ . The final symbol in the expression normally should result in a transition back to state 1, except when backlooping is involved (see [step 8](#) below).
  - c. Use a colon to mark state 1 as a final state (that is, 1:). Mark every state that is traversed before L:S is reached as a final state. Mark the state in which L:S is recognized as a final state. Use a period to mark all states traversed after that point as nonfinal (for instance, 2.). That is, once L:S is encountered it is not in the correct environment unless the full right context is found; thus these states cannot be final.
  - d. Since L:S in any other environment is not allowed, fill in the rest of the column for L:S with zeros. Furthermore, in any state traversed during the recognition of the right context, any correspondence encountered other than those provided for in *rc* means that L:S is in the wrong context. Thus, the rest of the cells for the states traversed in *rc* should be filled with zeros.
  - e. All remaining cells in the transition table denote successful transitions as far as this rule is concerned. In most cases, these cells are filled with transitions back to the initial state (that is, 1), except where backlooping

occurs (see [step 8](#)).

5. Compile each  $\leq$  rule. The rule  $L:S \leq lc\_rc$  can be paraphrased as "the expression  $lc L:\neg S rc$  is not allowed." The strategy for compiling a  $\leq$  rule to a state table is to construct a table that recognizes the sequence  $lc L:\neg S rc$  and forbids it, while permitting any other correspondences to occur anywhere. (Note that the strategy for building the  $\leq$  rule for an insertion, where  $L$  is 0 (the NULL symbol), is slightly different; see section [A3.7](#).) The steps in building the table are as follows:
  - a. Make a list of column headers for the table. First, put down  $L:S$ . Next, put down  $L:@$ , which now represents  $L:\neg S$ . Next, write down all the correspondences used in  $lc$  and  $rc$  (including correspondences with  $@$  and subset names). Add  $@:@$  to the end of the list.
  - b. Beginning with state 1, add states (rows) and fill in the state transitions in the appropriate cells in the table to recognize the expression  $lc L:@ rc$ . The final symbol in the expression should result in failure (that is, the cell representing recognition of the final symbol should contain 0 (zero)).
  - c. Use a colon to mark every state as a final state.
  - d. All remaining cells in the transition table denote successful transitions as far as this rule is concerned. In most cases, these cells are filled with transitions back to the initial state (that is, 1), except where backlooping occurs (see [step 8](#)).
6. Compile each  $\leq \Rightarrow$  rule. The rule may be compiled as two separate state tables, one for the  $\leq$  rule and the other for the  $\Rightarrow$  rule. Or, the rule may be compiled as a single table that combines the  $\leq$  and  $\Rightarrow$  rules. In this case, construct the column headers as in 5a. Then perform steps 5b through 5d to encode the  $\leq$  side of the rule. Finally, perform steps 4b through 4e to add the  $\Rightarrow$  side of the rule. In 4b, add new states only as needed for the recognition of  $rc$ ; the recognition of  $lc$  is the same. In 4c, mark as nonfinal the states added to recognize  $rc$ . (Alternatively, steps 4b to 4e can be done before steps 5b to 5d.)
7. Compile each  $\nless$  rule. The rule  $L:S \nless lc\_rc$  can be paraphrased as "the expression  $lc L:S rc$  is not allowed." The strategy for compiling a  $\nless$  rule to a state table is to construct a table that recognizes the sequence  $lc L:S rc$  and forbids it, while permitting any other correspondences to occur anywhere. The steps in building the table are as follows:
  - a. Make a list of column headers for the table by writing down all the correspondences used in the expression  $lc L:S rc$  (including correspondences with  $@$  and subset names). Add  $@:@$  to the end of the list.
  - b. Beginning with state 1, add states (rows) and fill in the state transitions in the appropriate cells in the table to recognize the expression  $lc L:S rc$ . The final symbol in the expression should result in failure (that is, the cell representing recognition of the final symbol should contain 0 (zero)).
  - c. Use a colon to mark every state as a final state.
  - d. All remaining cells in the transition table denote successful transitions as far as this rule is concerned. In most cases, these cells are filled with transitions back to the initial state (that is, 1), except where backlooping occurs (see [step 8](#)).



8. Check for backlooping. A **backloop** is a transition to a state that represents a previous point in the expression being recognized. The final step in compiling a rule (see steps 4e, 5d, and 7d) is to specify the transitions for all the remaining cells in the state table that are not part of the environment expression. Normally these are transitions back to state 1. However, backloops to states other than 1 must be specified if an input pair (or sequence of pairs) is recognized that matches the first symbol (or sequence of symbols) of the expression  $lc\ L:S\ rc$ . Transitions must be specified back to the states that represent the successful recognition of that symbol (or sequence of symbols). A detailed example of backlooping is given in [part 2](#) of section [A3.4](#).

### A3.3 Summary of two-level rule semantics

The semantics of the four kinds of two-level rules are now summarized in two ways. First, in figure [A5](#) a number of paraphrases are given for each rule type. Second, in figure [A6](#) truth tables (as in formal logic) are given. Note that the  $\leq$  and  $\Rightarrow$  rules have the familiar conditional pattern of formal logic. The  $\Leftrightarrow$  rule is the conventional biconditional.

**Figure A5** Semantics of two-level rules

$L:S \Rightarrow E$	<p>"Only but not always."</p> <p>L is realized as S only in E.</p> <p>L realized as S is not allowed in <math>\neg E</math>.</p> <p>If L:S, then it must be in E.</p> <p>Implies <math>L:\neg S</math> in E is permitted.</p>
$L:S \leq E$	<p>"Always but not only."</p> <p>L is always realized as S in E.</p> <p>L realized as <math>\neg S</math> is not allowed in E.</p> <p>If L is in E, then it must be L:S.</p> <p>Implies L:S may occur elsewhere.</p>
$L:S \Leftrightarrow E$	<p>"Always and only."</p> <p>L is realized as S only and always in E.</p> <p>Both <math>L:S \Rightarrow E</math> and <math>L:S \leq E</math>.</p> <p>Implies L:S is obligatory in E and occurs nowhere else.</p>
$L:S / \leq E$	<p>"Never."</p> <p>L is never realized as S in E.</p> <p>L realized as S is not allowed in E.</p> <p>If L is in E, then it must be <math>L:\neg S</math>.</p>

**Figure A6** Truth tables for two-level rules

There is an L.		Is the rule satisfied?			
Is it realized as S?	Is it in E?	$L:S \Rightarrow E$	$L:S \leq E$	$L:S \Leftrightarrow E$	$L:S / \leq E$
T	T	T	T	T	F
T	F	F	T	F	T
F	T	T	F	F	T

	F		F		T		T		T		T	
+-----+												

## A3.4 Compiling rules with a right context

This section gives step-by-step examples of how to apply the general procedure given in section [A3.2](#) for compiling rules into state tables to rules with only a right context. In the exposition of the following examples, phrases such as "[part4](#)" or "step 4a" refer to the numbered subparts of section [A3.2](#).

### (1) Compiling a => rule with a right context

As an example, we will posit a p:b correspondence preceding +m (strictly, +:0 m:m), where the symbol + stands for a morpheme boundary. Assume that + is always deleted in surface forms and can thus be declared as the default correspondence +:0. Examples of the p:b correspondence are:

LR: ap+ma ap+ma ap+ba  
 SR: ab0ma ap0ma ap0ba

According to the diagnostic questions in [part 3](#), these correspondences indicate that p is not always realized as b before +m (p:p also occurs before +m), but that +m is the only environment in which p:b is allowed. Therefore, posit a => rule:

R35          p:b => \_\_\_\_ +:0 m

To compile rule [R35](#) to a state table, follow the steps in [part 4](#). First (step 4a), make a list of the column headers, consisting of all the correspondences used in rule [R35](#) plus @:@:

p + m @  
 b 0 m @

The order of the columns of a state table do not affect its operation, but it is helpful to the reader to keep the columns in the same order as lc L:S rc so far as possible.

Next (step 4b), add rows (representing states) and fill in the cells with transitions to recognize the sequence p:b +:0 m:m. When the final symbol of the sequence (m:m) is reached, a transition is made back to state 1.

	p + m @	
	b 0 m @	
	-----	
1	2	
2	3	
3	1	

Next (step 4c), mark state 1 as a final state (that is, 1:). This allows the table to succeed on any correspondence that does not occur in L:S rc. Step 4c says that every state traversed before and including the state where L:S is recognized is marked as a final state. Since p:b is recognized in state 1, this is irrelevant. However, all states traversed after that point must be marked as nonfinal; that is, once p:b is recognized, it is not in the correct environment until the entire right context is found. Thus, states 2 and 3 cannot be final.

	p + m @
	b 0 m @
	-----
1:	2
2.	3
3.	1

Next (step 4d), fill in the rest of the column for p:b with zeros, since p:b in any other environment is not allowed. Also, for all the states traversed during the recognition of the right context, any correspondences other than those that are part of the right context mean that p:b is in the wrong context. Thus, the rest of the cells in rows 2 and 3 must be filled with zeros:

	p + m @
	b 0 m @
	-----
1:	2
2.	0 3 0 0
3.	0 0 1 0

Finally (step 4e), all remaining cells are successful transitions for this rule and can be filled in with transitions back to the initial state (that is, state 1). Note that since the remaining empty cells are in state 1 and do not involve the first correspondence of L:S rc, backlooping ([part 8](#)) is not involved. Table [T35](#) now gives the complete transition table for rule [R35](#).

T35 => table with right context

	p + m @
	b 0 m @
	-----
1:	2 1 1 1
2.	0 3 0 0
3.	0 0 1 0

## (2) Compiling a <= rule with a right context

Now suppose that the p:b correspondence is found in these forms (rather than the ones above):

LR: ap+ma ap+ba ap+ba  
 SR: ab0ma ap0ba ab0ba

According to the questions in [part 3](#), these correspondences indicate that p is always realized as b before +m, but +m is not the only environment in which p:b is allowed (p:b also occurs before +b). Therefore, posit a <= rule:

R36 p:b <= \_\_\_\_ +:0 m

To compile rule [R36](#) to a state table, follow the steps in [part 5](#). First (step 5a), make a list of the column headers, consisting of p:b, p:@, the correspondences used in the right context, and @:@:

	p p + m @
	b @ 0 m @

Due to the presence of p:b, p:@ means all other feasible pairs with a lexical p. In other words, it represents p:¬b.

Next (step 5b), add rows (states) and fill in the cells with transitions to recognize the sequence p:@ +:0 m:m. When the final symbol of the sequence is reached (m:m), the cell is filled with zero, indicating failure.

	p	p	+	m	@
b	@	0	m	@	
1		2			
2			3		
3				0	

Next (step 5c), mark every state as final:

	p	p	+	m	@
b	@	0	m	@	
1:		2			
2:			3		
3:				0	

Finally (step 5d), all remaining cells denote successful transitions back to the initial state and are filled in with ones, with the exception of cells where backlooping applies. To demonstrate backlooping, we will first ignore it and fill in the table with ones:

T36

	p	p	+	m	@
b	@	0	m	@	
1:	1	2	1	1	1
2:	1	1	3	1	1
3:	1	1	1	0	1

There is a problem with this state table. As written, table [T36](#) does not work correctly with more than one p in succession. For instance, given the lexical form app+ma, it will return appma, without voicing the second p. Step the example app+ma through table [T36](#) to verify this. When the second p:p is encountered while in state 2, the FST will make a transition back to state 1, where +:0 and m:m are recognized, leaving the FST in state 1. To remedy this, the FST must loop back to state 2 when it encounters p:p:

T36a

	p	p	+	m	@
b	@	0	m	@	
1:	1	2	1	1	1
2:	1	2	3	1	1
3:	1	1	1	0	1

But table [T36a](#) will still fail to recognize an input form such as ap+p+ma. This is because when the second p:p is encountered in state 3, the FST will make a transition back to state 1, again losing the fact that we are in the environment for the change. The table must be revised so that the FST will loop back to state 2 when a p:p is encountered in state 3 as well:

T36b    <= table with right context

	p	p	+	m	@
b	@	0	m	@	
1:	1	2	1	1	1

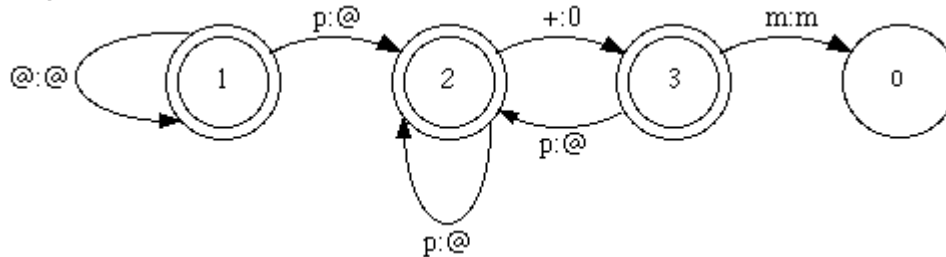
```

2: 1 2 3 1 1
3: 1 2 1 0 1

```

This is an example of the explanation given in [part 8](#) of how to handle backlooping. Backlooping is a subtle but important point, and is the source of many errors in compiling tables. The name is intended to convey the idea that the FST must have transitions (loops) back to the states where symbols (or sequences of symbols) of the expression `lc L:S rc` have been recognized. The notion of backlooping transitions is clearer when the FST is represented as a diagram; see figure [A7](#) (which is equivalent to table [T36b](#), except that transitions back to state 1 are not drawn). There are two backloops in this FST: from state 2 there is an arc for `p:@` back to state 2, and from state 3 there is another arc for `p:@` back to state 2.

**Figure A7** FST with backlooping



Why didn't we encounter backlooping while writing the state table for rule [R35](#) above? With respect to backlooping, it may seem that table [T35](#) should be written as follows (where states 2 and 3 have an arc back to state 2 if a `p:b` is recognized):

```

T35a      p + m @
          b 0 m @
          -----
          1: 2 1 1 1
          2: 2 3 0 0
          3: 2 0 1 0

```

The answer is that in this case step 4d overrides backlooping. Rule [R35](#) is a  $\Rightarrow$  rule, which means that the correspondence `p:b` is disallowed in any environment other than preceding `+m`. Table [T35a](#), however, would allow `p:b` to occur preceding another `p:b` or preceding the sequence `+:0 p:b`. To prevent this, step 4d requires the `p:b` column to contain zeros in states 2 and 3.

As a matter of practice in writing a state table, the analyst should carefully check the column that represents the first symbol of the expression `lc L:S rc` to see which state to loop back to in each state of the table.

Often it does not seem necessary in practice to account for backlooping because of language-specific phonotactic constraints. Taking the example word `app+ma` discussed above, suppose that the language being described does not have morphemes like `app`; that is, a phonotactic constraint prohibits the sequence `VCC`. In such a case, state table [T36](#) (written without backloops) would always work correctly as long as it was given input conforming to the phonotactic constraints of the language. There are two reasons why we recommend that the user of PC-KIMMO write tables that account for backlooping even when it seems unnecessary.

First, if you are inductively developing a phonological analysis, you will not necessarily know all the phonotactic constraints until the entire analysis is completed. If rules are written with incorrect backloops, puzzling failures may occur when further data are collected that contain new phonotactic patterns.

Second, it is conceptually cleaner to keep phonotactic constraints separate from the general phonological rules. Rather than incorporating phonotactic constraints in tables that encode phonological rules, it is better to write tables so that they are minimally restrictive with respect to phonotactics. The analyst can encode phonotactic constraints in a set of rules (tables) dedicated specifically to that purpose. For more discussion on expressing phonotactic constraints, see section [A3.16](#).

### (3) Compiling a $\langle \Rightarrow \rangle$ rule with a right context

Now suppose that the p:b correspondence is found only in forms like these:

```
LR:  ap+ma  ap+ba
SR:  ab0ma  ap0ba
```

According to the questions in [part 3](#), these correspondences indicate that p is always realized as b before +m, and that +m is the only environment in which p:b is allowed. Therefore, posit a  $\langle \Rightarrow \rangle$  rule:

R37      p:b  $\langle \Rightarrow \rangle$  \_\_\_\_ +:0 m

As was explained in [part 6](#) of section [A32.](#), a  $\langle \Rightarrow \rangle$  rule can be compiled as two separate state tables, one for the  $\langle =$  rule and one for the  $\Rightarrow$  rule. This is what has been done to produce state tables [T35](#) and [T36b](#) above. Alternatively, a  $\langle \Rightarrow \rangle$  rule can be compiled as a single table. To do this (see [part 6](#)), first construct the column headers by following the instructions in step 5a:

```
p p + m @
b @ 0 m @
```

Next perform steps 5b through 5d to construct the  $\langle =$  part of the rule:

```
p p + m @
b @ 0 m @
-----
1:  2 1 1 1
2:  2 3 1 1
3:  2 1 0 1
```

Now perform steps 4b through 4e to add the  $\Rightarrow$  part of the rule:

T37       $\langle \Rightarrow \rangle$  table with right context

```
p p + m @
b @ 0 m @
-----
1:  4 2 1 1 1
2:  4 2 3 1 1
3:  4 2 1 0 1
4:  0 0 5 0 0
5:  0 0 0 1 0
```

Notice that to recognize p:b and the right context +:0 m:m, two states (4 and 5, corresponding to states 2 and 3 in table [T35](#)) must be added to the table. These states are nonfinal states.

Special attention must be paid to the transitions in states 2 and 3 of table [T37](#). For the p:@ column, states 2 and 3 must loop back to state 2, which is the state in the

$\leq$  part of the rule where  $p:@$ , the first symbol of the expression  $lc\ L:S\ rc$ , has been recognized. This is identical to table [T36b](#). For the  $p:b$  column, states 2 and 3 must make a transition to state 4, which is the second state of the  $\Rightarrow$  part of the rule. This is the same transition as in state 1.

#### (4) Compiling a $\nless$ rule with a right context

The fourth rule type, the  $\nless$  rule, disallows the correspondence in the specified environment. For example, rule [R38](#) prohibits  $p:b$  before  $+m$ .

R38       $p:b\ \nless\ \_\_ \ +:0\ m$

The state table that encodes rule [R38](#) must recognize the sequence  $p:b\ +:0\ m:m$  and then forbid it. As the left arrow of the  $\nless$  operator suggests, the semantics of this rule type is most similar to the  $\leq$  rule. Whereas rule [R36](#) above (a  $\leq$  rule) states that  $p$  is always (obligatorily) realized as  $b$  before  $+:0\ m$  but may also be realized as  $b$  in some other environment, rule [R38](#) (a  $\nless$  rule) states that  $p$  is always (obligatorily) prohibited before  $+:0\ m:m$ , but may be realized as  $b$  in some other environment.

To compile rule [R38](#) to a state table, follow the steps in [part 7](#). First (step 7a), make a list of the column headers needed to recognize  $p:b$  and the environment plus  $@: @$ :

```
p + m @
b 0 m @
```

Notice that unlike table [T36b](#), which expresses a  $\leq$  rule, we do not need a  $p:@$  column header. This is because table [T36b](#) is built to prohibit the sequence  $p:\neg b\ +:0\ m:m$ , but the table we are building for a  $\nless$  rule must prohibit  $p:b\ +:0\ m:m$ .

Next (step 7b), add rows (states) and fill in the cells with transitions to recognize the sequence  $p:b\ +:0\ m:m$ . When the final symbol of the sequence is reached ( $m:m$ ), the cell is filled with zero, indicating failure.

```
  p + m @
  b 0 m @
-----
1  2
2   3
3   0
```

Next (step 7c), mark every state as final:

```
  p + m @
  b 0 m @
-----
1: 2
2:  3
3:   0
```

Finally (step 7d), all remaining cells denote successful transitions and are filled in with ones, with the exception of cells that meet the conditions of backlooping ([part 8](#)). Specifically, the cells in column  $p:b$  for states 2 and 3 must make a transition back to state 2, since state 2 represents the state where the first symbol ( $p:b$ ) of the expression  $lc\ L:S\ rc$  has been recognized.



```

T38    /<= table with right context
        p + m @
        b 0 m @
        -----
        1: 2 1 1 1
        2: 2 3 1 1
        3: 2 1 0 1

```

It is instructive to compare table [T38](#) with both table [T35](#) (for a => rule) and table [T36b](#) (for a <= rule).

## A3.5 Compiling rules with a left context

This section gives step-by-step examples of how to apply the general procedure given in section [A3.2](#) for compiling rules into state tables to rules with only a left context. In the exposition of the following examples, phrases such as "[part 4](#)" or "step 4a" refer to the numbered subparts of section [A3.2](#).

### (1) Compiling a => rule with a left context

In this section our example rule states that the correspondence p:b occurs following m+. For example,

```

LR: am+pa  am+pa  ab+pa
SR: am0ba  am0pa  ab0pa

```

According to the diagnostic questions in [part 3](#), these correspondences indicate that p is not always realized as b after m+ (p:p also occurs after m+), but that m+ is the only environment in which p:b is allowed. Therefore posit a => rule:

```

R39    p:b => m +:0 ____

```

To compile rule [R39](#), first (step 4a) make a list of the column headers:

```

m + p @
m 0 b @

```

Next (step 4b), add rows and fill in the cells to recognize the sequence m:m+:0 p:b:

```

m + p @
m 0 b @
-----
1  2
2   3
3    1

```

Next (step 4c), mark state 1 as a final state. Also, every state traversed up to and including the state where p:b is recognized is marked as a final state. Since there is no right context, there are no more states after that point.

```

m + p @
m 0 b @
-----
1: 2
2:  3
3:   1

```

Next (step 4d) fill in the rest of the p:b column with zeros, because the p:b correspondence cannot succeed until the entire left context has been recognized:

	m + p @	
	m 0 b @	
	-----	
1:	2 0	
2:	3 0	
3:	1	

Finally (step 4e), all remaining cells are successful transitions for this rule and can be filled in with transitions back to the initial state (state 1), with the exception of cells that meet the conditions of backlooping ([part 8](#)). Specifically, the cells in column m:m for states 2 and 3 must make a transition back to state 2, since state 2 represents the state where the first symbol (m:m) of the expression lc L:S rc has been recognized. Now table [T39](#) will work correctly with input forms such as amm+pa and am+m+pa.

T39 => table with left context

	m + p @	
	m 0 b @	
	-----	
1:	2 1 0 1	
2:	2 3 0 1	
3:	2 1 1 1	

## (2) Compiling a <= rule with a left context

Now suppose that the p:b correspondence is found in these forms:

LR: am+pa ab+pa ab+pa  
 SR: am0ba ab0pa ab0ba

According to the questions in [part 3](#), these correspondences indicate that p is always realized as b after m+, but m+ is not the only environment in which p:b is allowed (it also occurs after b+). Therefore, posit a <= rule:

R40 p:b <= m +:0 \_\_\_\_

To compile rule [R40](#), first (step 4a) make a list of the column headers, including p:@:

	m + p p @	
	m 0 b @ @	

Due to the presence of p:b, p:@ means all other feasible pairs with a lexical p. In other words, it represents p:¬b.

Next (step 5b), add rows and fill in the cells to recognize the sequence m:m +:0 p:@. When the final symbol of the sequence is reached (p:@), the cell is filled with zero, indicating failure.

	m + p p @	
	m 0 b @ @	
	-----	
1	2	
2	3	

3	0
---	---

Next (step 5c), mark every state as final:

	m + p p @
	m 0 b @ @
	-----
1:	2
2:	3
3:	0

Finally (step 5d), all remaining cells are successful transitions for this rule, and can be filled in with transitions back to the initial state (state 1), with the exception of cells that meet the conditions of backlooping ([part 8](#)). Specifically, the cells in column m:m for states 2 and 3 must make a transition back to state 2, since state 2 represents the state where the first symbol (m:m) of the expression lc L:S rc has been recognized. Now table 4 will work correctly with input forms such as amm+pa and am+m+pa.

T40 <= table with left context

	m + p p @			
	m 0 b @ @			
	-----			
1:	2	1	1	1
2:	2	3	1	1
3:	2	1	1	0

### (3) Compiling a <=> rule with a left context

Now suppose that the p:b correspondence is found only in forms like these:

LR: am+pa ab+pa  
 SR: am0ba ab0pa

According to the questions in [part 3](#), these correspondences indicate that p is always realized as b after m+, and m+ is the only environment in which p:b is allowed. Therefore, posit a <=> rule:

R41 p:b <=> m +:0 \_\_\_\_

As is explained in [part 6](#) of section [A3.2](#), a <=> rule can be compiled as two separate state tables, one for the <= rule and one for the => rule. This is what has been done to produce state tables [T39](#) and [T40](#) above. Alternatively, a <=> rule can be compiled as a single table. To do this (see [part 6](#)), first construct the column headers following the instructions in step 5a:

	m + p p @
	m 0 b @ @
	-----

Next perform steps 5b through 5d to construct the <= part of the rule:

	m + p p @
	m 0 b @ @
	-----
1:	2 1 1 1
2:	2 3 1 1
3:	2 1 0 1

Now perform steps 4b through 4e to add the  $\Rightarrow$  part of the rule. Since rule [R41](#) has no right context, no new states need to be added. Simply fill in the column for p:b. Notice that in states 1 and 2 the p:b column must be filled with zeros just as it is in rule [R39](#). If we encounter p:@ in state 3, then we fail; if we encounter p:b, then we succeed.

T41  $\langle \Rightarrow \rangle$  table with left context

```

      m + p p @
      m 0 b @ @
      -----
1: 2 1 0 1 1
2: 2 3 0 1 1
3: 2 1 1 0 1

```

#### (4) Compiling a $\wedge \leq$ rule with a left context

The fourth rule type, the  $\wedge \leq$  rule, disallows the correspondence in the specified environment. For example, rule [R42](#) prohibits p:b m:m +:0.

R42        p:b  $\wedge \leq$  m +:0 \_\_\_\_

To compile rule [R42](#) to a state table, follow the steps in [part 7](#). First (step 7a), make a list of the column headers needed to recognize p:b and the environment plus @:@:

```

      m + p @
      m 0 b @

```

Next (step 7b), add rows (states) and fill in the cells with transitions to recognize the sequence m:m +:0 p:b. When the final symbol of the sequence is reached (p:b), the cell is filled with zero, indicating failure.

```

      m + p @
      m 0 b @
      -----
1  2
2   3
3    0

```

Next (step 7c), mark every state as final:

```

      m + p @
      m 0 b @
      -----
1: 2
2:  3
3:   0

```

Finally (step 7d), all remaining cells denote successful transitions and are filled in with ones with the exception of cells that meet the conditions of backlooping ([part 8](#)). Specifically, the cells in column m:m for states 2 and 3 must make a transition back to state 2, since state 2 represents the state where the first symbol (m:m) of the expression lc L:S rc has been recognized.

T42  $\wedge \leq$  table with left context

```

      m + p @
      m 0 b @

```

```

-----
1: 2 1 1 1
2: 2 3 1 1
3: 2 1 0 1

```

## A3.6 Compiling rules with both left and right contexts

This section gives step-by-step examples of how to apply the general procedure given in section [A3.2](#) for compiling rules into state tables to rules with both a left and right context. In the exposition of the following examples, phrases such as "[part 4](#)" or "step 4a" refer to the numbered subparts of section [A3.2](#).

### (1) Compiling a => rule with left and right contexts

The example rule used in this section states that the correspondence s:z occurs intervocalically. For example,

```

LR:  sasa  sasa
SR:  saza  sasa

```

According to the diagnostic questions in [part 3](#), these correspondences indicate that s is not always realized as z between vowels (s:s also occurs between vowels), but that between vowels is the only environment in which s:z is allowed. Therefore, posit a => rule:

```
R43      s:z => V ____ V
```

To compile rule [R43](#) to a state table, follow the steps in [part 4](#):

```

T43  => table with left and right contexts
      V s @
      V z @
      -----
1: 2 0 1
2: 2 3 1
3. 2 0 0

```

While rule [R43](#) contains the correspondence V:V twice, table [T43](#) has only one V:V column header. The single V:V header serves for both instances of the correspondence in the environment. Having two identical column headers in a table will result in an error. (See also section [A3.8](#) on using subsets in state tables.)

Notice that states 1 and 2 are final, while state 3 is nonfinal. Also note carefully that accounting for backlooping requires the transition in state 2 in the V:V column to remain in state 2. This is necessary to allow the correct recognition of words with consecutive vowels, for instance saasa. Less obvious is that when V:V is recognized in state 3 the FST must return to state 2 rather than the expected state 1. This is necessary to allow the rule to apply more than once in the same word where the environments overlap. For example, consider these forms:

```

LR:  asasa
SR:  azaza

```

In this example, the second a serves both as the right context of the first s:z correspondence and as the left context of the second s:z correspondence. Therefore,

when it is first recognized in state 3 of the table, a transition must be made back to state 2 so that the rule can apply again.

## (2) Compiling a $\leq$ rule with left and right contexts

Now suppose that the s:z correspondence is found in these forms:

LR: sasa sasa  
SR: saza zaza

According to the questions in [part 3](#), these correspondences indicate that s is always realized as z between vowels, but between vowels is not the only environment in which s:z is allowed (it also occurs word-initially). Therefore, posit a  $\leq$  rule:

R44 s:z  $\leq$  V \_\_\_\_ V

To compile rule [R44](#), follow the steps in [part 5](#):

T44  $\leq$  table with left and right contexts

	V	s	s	@
	V	z	@	@
	-----			
1:	2	1	1	1
2:	2	1	3	1
3:	0	1	1	1

To account for backlooping, state 2 must have a 2 in the V:V column, parallel to table [T43](#). But unlike table [T43](#), state 3 must have a 0 in the V:V column, not a 2. This is because rule [R44](#) is a  $\leq$  rule and must disallow the sequence V:V s:@ V:V. However, table [T44](#) still correctly handles lexical forms such as asasa because only states 1 and 2 are used.

## (3) Compiling a $\Leftrightarrow$ rule with left and right contexts

Now suppose that the s:z correspondence is found only in an intervocalic position and s:s never is:

LR: sasa  
SR: saza

According to the questions in [part 3](#), these correspondences indicate that s is always realized as z between vowels, and between vowels is the only environment in which s:z is allowed. Therefore, posit a  $\Leftrightarrow$  rule:

R45 s:z  $\Leftrightarrow$  V \_\_\_\_ V

To compile rule [R45](#), follow the steps in [part 6](#):

T45  $\Leftrightarrow$  table with left and right contexts

	V	s	s	@
	V	z	@	@
	-----			
1:	2	0	1	1
2:	2	4	3	1
3:	0	0	1	1
4:	2	0	0	0

Rows 1 through 3 constitute the  $\leq$  part of the rule (compare rule [R44](#)), and rows 1, 2, and 4 constitute the  $\Rightarrow$  part of the rule (compare rule [R43](#)).

#### (4) Compiling a $\leq$ rule with left and right contexts

The fourth rule type, the  $\leq$  rule, disallows the correspondence in the specified environment. For example, rule [R46](#) prohibits V:V s:z V:V.

R46            s:z  $\leq$  V \_\_\_\_ V

To compile rule [R46](#) to a state table, follow the steps in [part 7](#):

T46             $\leq$  table with left and right contexts  
                 V s @  
                 V z @  
                 -----  
                 1: 2 1 1  
                 2: 2 3 1  
                 3: 0 1 1

### A3.7 Compiling insertion rules

The procedure for compiling two-level rules into state tables is slightly different for rules that insert characters. Compiling a  $\Rightarrow$  insertion rule is the same as described in the previous sections, but compiling a  $\leq$  rule requires a different strategy. We will demonstrate the procedure for handling insertion rules with an example from the Hanunoo language of the Philippines. In Hanunoo, the consonant h is inserted to break up a vowel cluster. This occurs, for instance, when the suffix i is added to a root that ends with a consonant; compare the following forms ([Schane 1973:54](#)). (Note that the character ? is used here to represent glottal stop.)

ROOT		ROOT+i	
----		-----	
?unum	`six'	?unumi	`make it six'
?usa	`one'	?usahi	`make it one'

In the following two-level representations, the inserted h is represented as corresponding to a lexical NULL symbol (zero):

LR: ?unum+i    ?usa+0i  
 SR: ?unum0i    ?usa0hi

The  $\Rightarrow$  rule for h-insertion is written as expected:

R47            h-insertion  
                 0:h  $\Rightarrow$  V +:0 \_\_\_\_ V

and is compiled into a state table in the usual way:

T47            h-insertion  
                 V + 0 @  
                 V 0 h @  
                 -----  
                 1: 2 1 0 1  
                 2: 2 3 0 1  
                 3: 2 1 4 1  
                 4: 2 0 0 0



Constructing the  $\leq$  table, however, is not as straightforward. Following the general procedure for compiling  $\leq$  rules to tables, we might expect to construct a the  $\leq$  table using the column headers 0:h and 0:@, where 0:@ is intended to specify 0: $\neg$ h (that is, a lexical 0 corresponding to anything except a surface h):

R48      h-insertion  
0:h  $\leq$  V +:0 \_\_\_\_ V

T48      h-insertion  
V + 0 0 @  
V 0 h @ @  
-----  
1: 2 1 1 1 1  
2: 2 3 1 1 1  
3: 2 1 1 4 1  
4: 0 1 1 1 1

Unfortunately, if we submit the lexical input form ?usa+i to rules [R47](#) and [R48](#), both the correct result ?usahi and the incorrect result ?usai will be returned. Why didn't rule [R48](#), the  $\leq$  rule, force the insertion of h as expected? The answer is in the meaning of the column header 0:@. What we really want the  $\leq$  rule to do is to recognize the absence of an inserted h in the specified environment and then to fail, that is, to prohibit the sequence V +:0 V. In effect this means that the table would have to recognize the correspondence 0:0 as an instance of the column header 0:@. However, 0:0 is not a feasible pair (and indeed never could be); thus the column header 0:@ cannot specify 0:0. As a matter of fact, if there are no other insertion correspondences in the description, PC-KIMMO will report an error when it tries to interpret table [T48](#), since there would be no feasible pairs that would match the 0:@ column header.

The answer to writing a table that makes h-insertion obligatory (that is, the effect of a  $\leq$  rule) is that it is necessary only to disallow the sequence V +:0 V. This can be easily done with a  $\wedge\leq$  rule of this form:

R49      h-insertion  
0:0  $\wedge\leq$  V +:0 \_\_\_\_ V

This rule must be understood in a special way. Although it follows the general syntax of two-level rules (correspondence, operator, environment), it departs from the normal meaning of two-level rules in that its correspondence part, namely 0:0, is not a feasible pair. However, its intended meaning is clear when it is compared to the corresponding  $\Rightarrow$  rule (see the rule in the header line in table [T47](#)). It simply means that something must be inserted where the environment line is located. The  $\Rightarrow$  rule provides the h, which is inserted at this point. The table that expresses rule [R49](#) looks like this:

T49      h-insertion  
V + @  
V 0 @  
-----  
1: 2 1 1  
2: 2 3 1  
3: 0 1 1

Now it is obvious that the two rules can be combined as a single  $\leq\Rightarrow$  rule.

R50      h-insertion

0:h <=> V +:0 \_\_\_\_ V

T50      h-insertion  
           V + 0 @  
           V 0 h @  
           -----  
           1: 2 1 0 1  
           2: 2 3 0 1  
           3: 0 1 4 1  
           4. 2 0 0 0

## A3.8 Using subsets in state tables

Section [A1.4](#) introduced the use of subsets in two-level rules. This section discusses their use in state tables. Assume that a two-level description contains these subsets (see section [A4.3](#) on subset declarations):

SUBSET D          t d s  
 SUBSET P          c j S  
 SUBSET Vhf        i e

In section [A1.4](#) a rule using these subsets was introduced, repeated here as rule [R51](#).

R51      Palatalization  
           D:P => \_\_\_\_ Vhf

Rule [R51](#) states that the alveolar consonants in subset D may be realized as the palatalized consonants in subset P when they occur preceding the high, front vowels in subset Vhf. Specifically, we want the subset correspondence D:P to stand for the feasible pairs t:c, d:j, and s:S. Translating rule [R51](#) into a state table is straightforward:

T51      Palatalization  
           D Vhf @  
           P Vhf @  
           -----  
           1: 2    1    1  
           2. 0    1    0

However, a two-level description containing table [T51](#) will produce no correct results unless the feasible pairs t:c, d:j, and s:S are declared explicitly. The pairs must appear as column headers in a table somewhere in the description. This is typically done by constructing a table specifically for the purpose of declaring special correspondences. For example, the following table declares the feasible pairs that we want for the column header D:P:

T52      Palatalization correspondences  
           t d s @  
           c j S @  
           -----  
           1: 1 1 1 1

Now the D:P column header in table [T51](#) will recognize all and only the pairs declared in table [T52](#). Similarly, the feasible pairs that Vhf:Vhf stands for (that is, i:i and e:e) must be declared somewhere in the description. Since in this case the pairs are default correspondences, they will typically be included in the table with all the other default correspondences.

## A3.9 Overlapping column headers and specificity

Using subsets in rules often leads to a situation where a state table has column headers that potentially overlap. In such a case, unexpected results may occur. For example, consider this rule, which states that t:c occurs between any vowel and i:

R53            t:c => V \_\_\_\_ i

A first attempt at writing a state table for rule [R53](#) might look like this:

```
T53           V t i @
              V c i @
              -----
              1: 2 0 1 1
              2: 2 3 1 1
              3. 0 0 1 0
```

Given the lexical form *mati*, table [T53](#) will correctly produce the surface form *maci*. But given the form *miti*, it will fail to produce the expected result *mici*. This is because of the interaction of the column headers V:V and i:i. Because the feasible pair i:i is an instance of V:V, we might expect that the first i in the input form *miti* would match the V:V column header and cause a successful transition to state 2. This is not the case. For each table in a PC-KIMMO description, the entire set of feasible pairs must be partitioned among the column headers with no overlap. Each feasible pair belongs to one and only one column header. When PC-KIMMO interprets the column headers of a table, it scans the list of all the feasible pairs and assigns each one to a column header. If a feasible pair matches more than one column header, it assigns it to the most specific one, where the specificity of a column header is defined as the number of feasible pairs that matches it. In order to see exactly how the feasible pairs are assigned to the column headers of a rule, use the **show rule** command (see section [4.5.9](#)).

Thus in table [T53](#) the feasible pair i:i potentially matches both the column headers V:V and i:i; but because i:i is more specific than V:V, the pair i:i is assigned to the column header i:i. This means that the column header V:V stands for all the feasible pairs of vowels except i:i. Thus the input pair i:i matches only the column header i:i. To work correctly, table [T53](#) must allow i:i to be an instance of V:V in the left context by placing a 2 in states 1 and 2 under the i:i header. Note also that the order of the columns has no effect on which column header an input pair is matched to. Table [T53a](#) reflects these changes.

```
T53a          V t i @
              V c i @
              -----
              1: 2 0 2 1
              2: 2 3 2 1
              3. 0 0 2 0
```

Now consider a description that contains a subset Vrd for rounded vowels and a subset Vhi for high vowels:

```
SUBSET Vrd  o u
SUBSET Vhi  i e o u
```

Notice that the Vhi subset properly includes the Vrd subset. Assume that the description contains the following rule:

R54      t:c => Vrd \_\_\_\_ Vhi

We first write a state table for rule [R54](#) like this:

T54		Vrd t	Vhi	@
		Vrd c	Vhi	@
		<hr/>		
	1:	2	0	1 1
	2:	2	3	1 1
	3:	0	0	1 0

But the feasible pairs o:o and u:u, which match both the Vrd:Vrd and Vhi:Vhi column headers, must belong to the Vrd:Vrd column, since it is more specific. Thus the Vhi column represents only the pairs i:i and e:e. This means that a lexical input form such as utu will not produce the expected surface form ucu, because the second u will always match Vrd, not Vhi. This problem is fixed by including u:u and o:o as column headers in table [T54a](#):

T54a		Vrd t	Vhi u	o	@
		Vrd c	Vhi u	o	@
		<hr/>			
	1:	2	0	1	2 2 1
	2:	2	3	1	2 2 1
	3:	0	0	1	2 2 0

The solution, then, in cases of overlapping column headers is to explicitly include as headers in the table the feasible pairs that belong to both headers.

It is possible to construct a state table in which a feasible pair matches multiple column headers that have the same specificity value, making it impossible to uniquely assign the pair to a column. This constitutes an incorrectly written state table. When the rules file containing such a state table is loaded, a warning message is issued alerting the user that two columns have the same specificity. If the user proceeds to analyze forms with the incorrectly written table, a pair will be assigned (arbitrarily) to the leftmost column that it matches. Correct results cannot be assured.

## A3.10 Expressing word boundary environments

Consider a phonological rule that states that stops are devoiced when they occur in word-final position. For example,

LR: mabab  
SR: mabap

Assume these subsets for voiced stops (B) and voiceless stops (P):

SUBSET B    b d g  
SUBSET P    p t k

Two-level rules use the BOUNDARY symbol (#) to indicate word boundary:

R55      Devoicing  
         B:P <=> \_\_\_\_ #

The corresponding state table is written with #:# as the column header representing word boundary. Note that a boundary symbol used in a column header can only

correspond to another boundary symbol; that is, correspondences such as #:0 are illegal.

```
T55      Devoicing
          B B # @
          P @ # @
          -----
          1: 3 2 1 1
          2: 3 2 0 1
          3: 0 0 1 0
```

Rules and tables that refer to an initial word boundary are written in a similar way. Here is a rule for word-initial spirantization.

```
R56      Spirantization
          p:f <=> # ____ V
```

```
T56      Spirantization
          # p p V @
          # f @ V @
          -----
          1: 2 0 1 1 1
          2: 1 4 3 1 1
          3: 1 0 1 0 1
          4: 0 0 0 1 0
```

(Notice that since the first symbol of lc L:S rc is initial word boundary, backlooping is irrelevant.)

## A3.11 Expressing complex environments in state tables

Section [A1.6](#) discussed the notational conventions used to express complex environments in two-level rules. Those rules are repeated here with instructions on how to express them in state tables.

As an example we will use a vowel reduction rule. It states that a vowel followed by some number of consonants followed by stress (indicated by ') is reduced to schwa (ê). For example,

```
LR:   bab'a   bamb'a
SR:   bêb'a   bêm̃b'a
```

In rule [R57](#) we treat the case where there is exactly one or two intervening consonants. Parentheses indicate that the second consonant is optional.

```
R57      V:ê => ____ C(C)'
```

In table [T57](#), the second, optional consonant is implemented in state 3. The table succeeds when it recognizes the stress, either in state 3 after finding one consonant, or in state 4 after finding another consonant.

```
T57      Vowel Reduction
          V C ' @
          ê C ' @
          -----
          1: 2 1 1 1
          2: 0 3 0 0
```

```

3. 0 4 1 0
4. 0 0 1 0

```

Rule [R58](#) and table [T58](#) specify either zero, one, or two consonants.

R58      Vowel Reduction  
V:ê => \_\_\_\_ (C) (C)'

T58      Vowel Reduction

	V	C	'	@
	ê	C	'	@
	-----			
1:	2	1	1	1
2:	0	3	1	0
3:	0	4	1	0
4:	0	0	1	0

The only difference from table [T57](#) is found in state 2 of table [T58](#), where it is allowed to encounter the stress immediately after the V:ê correspondence.

In rule [R59](#) the asterisk indicates zero or more instances of C.

R59      Vowel Reduction  
V:ê => \_\_\_\_ C\*'

Table [T59](#) succeeds in state 2 either by immediately finding stress or by repeating state 2 to find consonants until stress is reached.

T59      Vowel Reduction

	V	C	'	@
	ê	C	'	@
	-----			
1:	2	1	1	1
2:	0	2	1	0

Rule [R60](#) specifies one or more consonants.

R60      Vowel Reduction  
V:ê => \_\_\_\_ CC\*'

R60      Vowel Reduction

	V	C	'	@
	ê	C	'	@
	-----			
1:	2	1	1	1
2:	0	3	0	0
3:	0	3	1	0

Here state 2 requires that at least one consonant be found. Then state 3 functions like state 2 of the previous example to repeat consonants until stress is found.

Section [A1.6](#) discussed multiple environments in two-level rules. In this section the state tables for those rules are provided. The example used here is a vowel lengthening rule. It states that the correspondence a:ä (short and long a) occurs in two distinct environments: when it is stressed (tonic lengthening) or when it occurs in the syllable preceding stress (pretonic lengthening). For example,

LR: ladab'ar

SR: ladäb'är

First, the tonic and pretonic lengthening rules and tables are written as separate rules:

R61 Pretonic Lengthening  
a:ä => \_\_\_\_ C'

T61 Pretonic Lengthening  
a C' @  
ä C' @  
-----  
1: 2 1 1 1  
2: 0 3 0 0  
3: 0 0 1 0

R62 Tonic Lengthening  
a:ä => ' \_\_\_\_

T62 Tonic Lengthening  
' a @  
' ä @  
-----  
1: 2 0 1  
2: 2 1 1

Note that in state 2 the 2 under the stress header is due to backlooping, even though we do not expect to have two stress marks in succession (see section [A3.4](#)).

As discussed in section [A1.6](#), rules [R61](#) and [R62](#) are contradictory; they both claim to specify the only environment in which a:ä is allowed. They must be combined into a single rule, rule [R63](#), which is expressed as state table [T63](#).

R63 Pretonic and Tonic Lengthening  
a:ä => [ \_\_\_\_ C' | ' \_\_\_\_ ]

T63 Pretonic and Tonic Lengthening  
a C' @  
ä C' @  
-----  
1: 2 1 4 1  
2: 0 3 0 0  
3: 0 0 4 0  
4: 1 1 4 1

There is one key difference between table [T63](#), and tables [T61](#) and [T62](#). This is the change in state 3 where stress now makes a transition to state 4 rather than back to state 1. This is necessary because stress (which is in the right context of rule [R61](#)) is the first symbol of the left context of rule [R62](#). (Note that in state 4 in the stress column the transition back to state 4 is due to backlooping.)

Rules [R64](#) and [R65](#) and tables [T64](#) and [T65](#) express the same lengthening rules, only using the <= operator.

R64 Pretonic Lengthening  
a:ä <= \_\_\_\_ C'

T64      Pretonic Lengthening

a	a	C	'	@
ä	@	C	'	@

---

1: 1 2 1 1 1  
 2: 1 2 3 1 1  
 3: 1 2 1 0 1

R65      Tonic Lengthening

a:ä <= ' \_\_\_\_

T65      Tonic Lengthening

'	a	a	@
'	ä	@	@

---

1: 2 1 1 1  
 2: 2 1 0 1

In table [T64](#) under the a:@ header, there are transitions back to state 2 in both state 2 and state 3. This is due to backlooping.

Rules [R64](#) and [R65](#), being <= rules, do not conflict, since each allows the a:ä correspondence in environments other than its own. Nevertheless, if the analyst so chooses, they can be combined into one table:

R66      Pretonic and Tonic Lengthening

a:ä <= [ \_\_\_\_ C' | ' \_\_\_\_ ]

T66      Pretonic and Tonic Lengthening

a	a	C	'	@
ä	@	C	'	@

---

1: 1 3 1 2 1  
 2: 1 0 1 2 1  
 3: 1 3 4 2 1  
 4: 1 3 1 0 1

## A3.12 Expressing two-level environments

Section [A1.7](#) discussed the use of two-level environments in phonological rules. The two rules developed in that section to account for Nasal Assimilation and Stop Voicing are repeated here with their state tables (assume that a default N:n correspondence is declared elsewhere in the description):

R67      Nasal Assimilation

N:m <=> \_\_\_\_ p:

T67      Nasal Assimilation

N	N	p	@
m	@	@	@

---

1: 3 2 1 1  
 2: 3 2 0 1  
 3: 0 0 1 0

R68      Stop Voicing



p:b <=> :m \_\_\_\_

T68      Stop Voicing  
           @ p p @  
           m b @ @  
           -----  
           1: 2 0 1 1  
           2: 1 1 0 1

These rules relate the lexical sequence Np to the surface sequence mb. Note carefully that the symbol p: in rule [R67](#) is expressed as the column header p:@ in table [T67](#), and the symbol :m in rule [R68](#) is expressed as the column header @:m in table [T68](#). (See section [A1.7](#) on overspecification in rules of this type.)

Now assume that the lexical sequence Nb is realized as the surface sequence mb (that is, both lexical Np and Nb are realized as surface mb). This shows that the N:m correspondence is found before a surface b that realizes either a lexical p or b. The distribution of the p:b correspondence is the same. Rule [R67](#) then must be revised as follows:

R67a      Nasal Assimilation  
           N:m <=> \_\_\_\_ :b

T67a      Nasal Assimilation  
           N N @ @  
           m @ b @  
           -----  
           1: 3 2 1 1  
           2: 3 1 0 1  
           3. 0 0 1 0

Unfortunately, if a description containing tables [T67a](#) and [T68](#) is given the lexical input form aNpa, it produces not only the expected surface form amba but also the incorrect form anpa. The reason for this failure is similar to the problem of overspecification discussed in section [A1.7](#). Notice the symmetrical, interlocking relationship between rules [R67a](#) and [R68](#). The environment of each rule is the surface character of the correspondence part of the other rule; that is, the environment of rule [R67a](#) is :b, which is the surface character of the p:b correspondence of rule [R68](#), and the environment of [R68](#) is :m, which is the surface character of the N:m correspondence of rule [R67a](#). This means, with respect to the lexical form aNpa, that rule [R67a](#) does not require N to be realized as m before a p that is realized as anything other than b, and rule [R68](#) does not require p to be realized as b after an N that is realized as anything other than m. Thus the form aNpa can pass through the two rules vacuously. Assuming that the analyst is correct in positing surface environments for these two rules, the only way to fix the problem is to prohibit the sequence N:n p:p. This can be done either by adding the rule N:n /<= \_\_\_\_ p, or by incorporating this prohibition into one of the existing tables. For example, we can revise table [T67a](#) as follows:

T67b      Nasal Assimilation  
           N N @ p @  
           m @ b p @  
           -----  
           1: 3 2 1 1 1  
           2: 3 1 0 0 1  
           3. 0 0 1 0 0

By including the column header p:p (or perhaps @:p) in table [T67b](#), we can recognize N:@ p:p and force failure. Now the lexical form aNpa will match only the surface form amba. (Alternatively, table [T68](#) could be revised to include the column header N:n and fail when the sequence N:n p:@ is recognized.)

### A3.13 Rule conflicts

The two main types of rule conflicts are the  $\Rightarrow$  (or *environment*) conflict and the  $\Leftarrow$  (or *realization*) conflict ([Dalrymple and others 1987:25](#)). The  $\Rightarrow$  conflict arises when two conditions are met: (1) two  $\Rightarrow$  rules have the same correspondence on the left side of the rule, but (2) they have different environments on the right side. (This type of conflict has already been discussed in section [A1.6](#).) For example,

R69      Intervocalic Voicing  
p:b  $\Rightarrow$  V \_\_\_\_ V

R70      Voicing after nasal  
p:b  $\Rightarrow$  m \_\_\_\_

Since the rule operator  $\Rightarrow$  means that the correspondence can occur only in the specified environment, rules [R69](#) and [R70](#) contradict each other. The simplest resolution of the conflict is to combine the two rules into one rule with a disjunctive environment:

R71      Voicing  
p:b  $\Rightarrow$  [ V \_\_\_\_ V | m \_\_\_\_ ]

The state table for rule [R71](#) looks like this:

T71      Voicing  
          V m p @  
          V m b @  
          -----  
1: 2 4 0 1  
2: 2 4 3 1  
3: 2 0 0 0  
4: 2 4 1 1

where states 1, 2, and 3 correspond to the V \_\_\_\_ V part of rule [R71](#) and states 1 and 4 correspond to the m \_\_\_\_ part.

Now assume that rules [R69](#) and [R70](#) have been initially written as  $\Leftarrow$  rules:

R72      Intervocalic Voicing  
p:b  $\Leftarrow$  V \_\_\_\_ V

R73      Voicing after nasal  
p:b  $\Leftarrow$  m \_\_\_\_

Their state tables look like this:

T72      Intervocalic Voicing  
          V p p @  
          V b @ @  
          -----  
1: 2 0 1 1  
2: 2 4 3 1

```

3: 0 0 1 1
4. 2 0 0 0

```

```

T73      Voicing after nasal
         m p p @
         m b @ @
         -----
1: 2 0 1 1
2: 2 1 0 1

```

A description containing tables [T72](#) and [T73](#) will not work, because the => sides of the rules conflict, just like rules [R69](#) and [R70](#). There are two ways to resolve the conflict between rules [R72](#) and [R73](#). First, the rules can be separated into their <= parts and => parts, and the => parts combined as above:

```

R74      Intervocalic Voicing
         p:b <= V ____ V

R75      Voicing after nasal
         p:b <= m ____

R76      Voicing
         p:b => [ V ____ V | m ____ ]

```

State tables are easily written for rules [R74](#) and [R75](#) (not included here), and table [T71](#) encodes rule [R76](#) (same as rule [R71](#)).

The second way to resolve the conflict between rules [R72](#) and [R73](#) is to modify the environment of each table to allow the environment of the other. Tables [T72](#) and [T73](#) are revised as [T72a](#) and [T73a](#).

```

T72a     Intervocalic Voicing
         V p p m @
         V b @ m @
         -----
1: 2 0 1 5 1
2: 2 4 3 5 1
3: 0 0 1 5 1
4. 2 0 0 0 0
5: 2 1 1 5 1

T73a     Voicing after nasal
         m p p V @
         m b @ V @
         -----
1: 2 0 1 3 1
2: 2 1 0 3 1
3: 2 1 1 3 1

```

Table [T72a](#) contains the column header m:m from table [T73](#), and table [T73a](#) contains the column header V:V from table [T72](#). This enables the sequence m:m p:b to pass vacuously through table [T72a](#) and the sequence V:V p:b V:V to pass vacuously through table [T73a](#).

It should also be noted that tables [T72a](#) and [T73a](#) can be combined into a single table that expresses the disjunctive rule  $p:b \Leftrightarrow [V \text{ ____ } V \mid m \text{ ____ }]$ . This can be done by dispensing with table [T73a](#) and placing a zero in the cell at the intersection of row 5 and the p:@ column of table [T72a](#). However, when dealing with very complex rules

with perhaps more than one conflict, it may be clearer to keep the rules separate as shown above.

The second type of rule conflict is the  $\leq$  (or realization) conflict. It arises when two conditions are met: (1) the correspondence parts of two  $\leq$  rules have the same lexical character but different surface realizations of it, and (2) the environment of one rule is subsumed by the environment of the other rule. For example, to account for the following correspondences, we posit rules [R7](#) and [r78](#) (where Z stands for a voiced alveopalatal grooved fricative):

LR: asa isi  
SR: aza iZi

R77 Intervocalic Voicing  
s:z  $\leq$  V \_\_\_\_ V

R78 Palatalization  
s:Z  $\leq$  i \_\_\_\_ i

These rules meet both conditions of a  $\leq$  conflict. First, the lexical characters of their correspondence parts are the same (namely s), while the surface characters are different (z and Z). Second, because i is a member of the subset V, the environment of rule [R77](#) subsumes the environment of rule [R78](#); that is, i \_\_\_\_ i is a specific instance of the more general environment V \_\_\_\_ V. The state tables for rules [R77](#) and [R78](#) are as follows:

T77 Intervocalic Voicing  
V s s @  
V z @ @  
-----  
1: 2 1 1 1  
2: 2 1 3 1  
3: 0 1 1 1

T78 Palatalization  
i s s @  
i Z @ @  
-----  
1: 2 1 1 1  
2: 2 1 3 1  
3: 0 1 1 1

Given the lexical input form asa, only rule [R77](#) will apply and return the correct surface form aza. Given the lexical form isi, we want rule [R78](#) to apply and produce the surface form iZi, but in fact the rules fail to return any result. This is because rule [R77](#) disallows s:Z between vowels (including i's), while rule [R78](#) disallows s:z between i's. Also, the rules cannot produce the surface form izi, because this contradicts rule [R78](#), which states that s must be realized as Z.

In generative phonology this type of conflict is resolved by ordering the specific rule before the general rule, in this case Palatalization before Voicing. Rule ordering is of course not available in the two-level model. To resolve a  $\leq$  conflict in a two-level description, the general rule must be altered to allow (but not require) the correspondence of the specific rule to occur in its environment. Table [T77](#) must therefore be revised as [T77a](#).

T77a Intervocalic Voicing

	V	s	s	s	@
	V	z	@	Z	@
	-----				
1:	2	1	1	1	1
2:	1	1	3	1	1
3:	0	1	1	1	1

In table [T77](#) the column header s:@ stands for the set of correspondences s:s and s:Z, but in table [T77a](#) the inclusion of the header s:Z restricts the meaning of s:@ to only s:s. Thus the occurrence of s:Z is not restricted by table [T77a](#). Now s will be realized as Z in the environment i \_\_\_ i because table [T77a](#) allows it and table [T78](#) requires it.

## A3.14 Comments on the use of => rules

The two major rule types, the <= rule and the => rule, have been described informally as the "obligatory" rule and the "optional" rule. The meaning and use of the obligatory <= rule are fairly straightforward, but the use of the optional => rule in actual two-level descriptions deserves more comment. There are three ways in which => rules are employed.

First, as the term optional suggests, a => rule is used in cases where two surface characters are truly in free variation, regardless of morphological or lexical context. For example, in many dialects of American English t is in free variation with an alveolar flap D when it occurs after a vowel and before an unstressed vowel; for example, the word writer can be pronounced either [r'aytêr] or [r'ayDêr]. This is expressed by a => rule such as rule [R79](#) (where the absence of the stress symbol (') indicates no stress). Such rules of free variation are typically low-level phonetic rules.

R79      Flapping  
t:D => V \_\_\_ V

Second, a => rule may be used in cases where a correspondence is restricted to certain lexical items or classes of lexical items (for instance, nouns or verbs), or to certain morphological contexts (for instance, nominative case). For example, English needs a rule for the f:v correspondence in pairs of words such as wife and wives, leaf and leaves. But this rule is restricted to a very small and arbitrary number of lexical items (it does not apply to fife, reef, and so on). The simplest solution is to write the f:v rule as a => rule and let it overgenerate and overrecognize. That is, it will generate and recognize nonwords such as wives (the plural of wife) and fives (the plural of fife). For purposes of testing a two-level description, the files of test data should contain only well-formed words.

In generative phonology the solution to this problem is to mark the lexical entries of the words wife, leaf, and so on for a "positive rule exception," which says that only words so marked can undergo the f:v rule. The lexical component of PC-KIMMO does not allow lexical entries to be so marked for lexical features. However, the same effect can be produced by introducing a special character (often called a diacritic) in the lexical forms of exceptional words. This character serves as the "trigger" for certain rules to apply. Thus wife and leaf could be given the lexical forms wayf\* and liyf\* while fife and reef would have the lexical forms fayf and riyf. The f:v rule would then be written like this (where +z stands for the plural morpheme):

R80      f:v <=> \_\_\_ \*:0 +:0 z

While this solution works, it has the undesirable effect of positing lexical

representations that contain nonphonological elements. Many linguists would reject such representations on theoretical grounds. (A similar solution is to posit lexical forms such as wayF and liyF and a rule for F:v. The same linguistic objections apply.)

Third, a  $\Rightarrow$  rule is used to "clean up"  $\Leftarrow$  rules. This is a nonobvious but very important use of  $\Rightarrow$  rules. For example, assume that a two-level description contains two obligatory rules for lengthening, namely rules [R64](#) and [R65](#) in section [A1.6](#) for pretonic and tonic lengthening. While these rules may express intuitively that lengthening applies obligatorily in the specified environments, running PC-KIMMO with just these two rules will result in overgeneration. Because  $\Leftarrow$  rules do not restrict the occurrence of the correspondence in other environments, rules [R64](#) and [R65](#) will produce forms with the a:ä correspondence in environments where they do not occur. For example, given the lexical input labad'ar, rules [R64](#) and [R65](#) will return both labäd'är (correct) and läbäd'är (incorrect). To prevent this type of overgeneration,  $\Leftarrow$  rules must be accompanied by analogous  $\Rightarrow$  rules. Thus when rule [R63](#) is added to the description containing rules [R64](#) and [R65](#), only correct surface forms will be generated.

As a practical procedure in developing a two-level description, the user will typically write all the obligatory  $\Leftarrow$  rules for a given correspondence first. Then to correct the resulting overgeneration, the user must write a single  $\Rightarrow$  rule for the correspondence; it must contain as a multiple environment (to avoid  $\Rightarrow$  conflicts) all the contexts of the  $\Leftarrow$  rules for the correspondence.

As another example of the use of  $\Rightarrow$  rules as "clean-up" rules, consider again an example used in section [A1.6](#) where the vowel of the ultimate syllable of a word is lengthened unless it is schwa, in which case the vowel of the penultimate syllable is lengthened (for example, mamän and mamänê). Assume these subsets:

```
SUBSET V      i a u ê
SUBSET Vlng   î ä ü @
```

and these special correspondences:

```
Lengthening correspondences
  i a u @
  î ä ü @
  -----
1: 1 1 1 1
```

Following the procedure described above, assume that this is an obligatory process. Here is the  $\Leftarrow$  rule and its state table:

```
R81      Lengthening
         V:Vlng  $\Leftarrow$  ____ C(ê)#
```

```
T81      Lengthening
         V      V C ê # @
         Vlng @ C ê # @
         -----
1: 1      2 1 1 1 1
2: 1      2 3 1 1 1
3: 1      2 1 4 0 1
4: 1      2 1 1 0 1
```

Now to prevent the overgeneration of ill-formed surface forms such as mämän and mämänê, this "clean-up"  $\Rightarrow$  rule must be included:

R82      Lengthening  
 V:Vlng => \_\_\_\_ C(ê)#

T82      Lengthening  
           V      C ê # @  
           Vlng C ê #&@  
           -----  
 1: 2      1 1 1 1  
 2. 0      3 0 0 0  
 3. 0      0 4 1 0  
 4. 0      0 0 1 0

### A3.15 Comments on the use of morpheme boundaries

In standard generative phonology, a phonological rule that applies to the segments XY also applies to X+Y, where + indicates a morpheme boundary ([Chomsky and Halle 1968:364](#)). In other words, a phonological rule that applies within a morpheme is assumed also to apply across morpheme boundaries. Thus it is not necessary to include optional morpheme boundaries in rules. Clearly two-level rules can also be written without optional morpheme boundaries; but state tables must explicitly include a morpheme boundary column even if they are optional at each point in the input string. To make a morpheme boundary completely optional in a table, simply loop back to the current state in each state of the table. For example, here is a rule and table for intervocalic voicing:

R83      Intervocalic voicing  
 s:z lt;=gt; V \_\_\_\_ V

T83      Intervocalic voicing  
           V    s    s    +    @  
           V    z    @    0    @  
           -----  
 1: 2    0    1    1    1  
 2: 2    4    3    2    1  
 3: 0    0    1    3    1  
 4. 2    0    0    4    0

(Notice that rows 1--3 encode the <= part of the rule and rows 1--2 and 4 encode the => part.) This table will allow a morpheme boundary at any point in the lexical form, for instance sa+za and saz+a.

It should be noted that generative descriptions do use explicit morpheme boundaries in rules; in such cases the rule only applies in the presence of the boundary. Often this is done to limit the rule's application to a specific morpheme by actually "spelling out" the morpheme in the rule's environment. This trick is necessary also in PC-KIMMO, since PC-KIMMO does not allow the application of rules to be limited to certain lexical items by means of lexical features. For example, the English prefix in+ has the allomorphs il+ and ir+ in words such as illegal and irregular (compare intolerable). But we do not want to write a rule that changes n to l or r everywhere (compare unlawful, inlet, enlarge, unreal). Therefore we write the => rule and table for n:l to limit the application of the rule to the lexical form in+. (The rule could be made even more specific by requiring the prefix to be word-initial.)

R84      n:l => i \_\_\_\_ +l

T84      i    n    +    l    @

	i	l	o	l	@
1:	2	1	1	1	1
2:	2	3	1	1	1
3:	0	0	4	0	0
5:	0	0	0	1	0

## A3.16 Expressing phonotactic constraints

In section [A3.4](#) we recommended that tables should be written without incorporating phonotactic constraints in them. As a matter of practice, this approach may result in less time spent debugging a set of rules. But more importantly, a linguistic description should distinguish between phonological rules (correspondences between lexical and surface characters) and phonotactic constraints (restrictions on permitted sequences of characters). For instance, just as the phonological description of English includes allophonic rules stating the distribution of aspirated and unaspirated voiceless stops, it also includes phonotactic constraints such as restrictions on possible word-initial consonant clusters.

As an example of how to encode phonotactic constraints as state tables, consider a language that allows words of the phonological shape CV(C)CV(C). That is, a word minimally consists of two open (CV) syllables, each of which can optionally be closed by a consonant. Possible words are baba, bamba, bambam, and so on. The following state table restricts all words to this pattern:

T85 CV(C)CV(C) pattern

	#	C	V	@
	#	@	@	@
1:	2	1	1	1
2:	0	3	0	2
3:	0	0	4	3
4:	0	5	0	4
5:	0	6	7	5
6:	0	0	7	6
7:	1	8	0	7
8:	1	0	0	8

By using the column headers C:@ and V:@ rather than C:C and V:V, table [T85](#) is a statement of phonotactic constraints on lexical forms, not surface forms. Phonological rules such as deletions could result in surface forms that do not conform to the lexical-level phonotactic pattern. To allow for diacritics such as stress ('), the @:@ column in table [T85](#) ignores all symbols that are not either consonants or vowels. Thus a word such as bab'a is allowed by the table.

As another example, we will attempt to describe the constraints on initial consonant clusters in English. First we will define the following subsets for voiceless stops (P), liquids (L), and nasals (N):

```
SUBSET P  p t k c
SUBSET L  l r
SUBSET N  m n
```

We want to allow word-initial clusters of the following types: sP, sL, sN, sPL, and PL. These constraints on clusters at the lexical level are encoded in table [T86](#).

T86 Word-initial consonant cluster constraints



	#	s	P	L	N	V	C	@
	#	@	@	@	@	@	@	@
-----								
1:	2	1	1	1	1	1	1	1
2:	1	3	4	5	5	1	5	2
3:	0	0	4	5	5	1	0	3
4:	0	0	0	5	0	1	0	4
5:	0	0	0	0	0	1	0	5

Table [T86](#) will allow the lexical forms of words such as spit, slit, snip, prick, click, split, string, and so on, but disallow sbit, slpit, spmit, mlik, and so on. Unfortunately, it will also allow nonoccurring words such as srit, tlick, and sklit (though scl does occur in words of Greek origin, for instance sclera). To disallow these, another table can encode refinements to the above table:

T87 More initial consonant cluster constraints

	#	s	t	k	l	r	N	V	C	@
	#	@	@	@	@	@	@	@	@	@
-----										
1:	2	1	1	1	1	1	1	1	1	1
2:	1	3	4	1	1	1	1	1	1	2
3:	0	0	4	4	1	0	1	1	1	3
4:	0	0	0	0	0	1	0	1	0	4

(Note that tables [T86](#) and [T87](#) disallow the clusters sph and sv, which occur in words of foreign origin such as sphere and svelte.)

---

[Up](#) [Next](#) [Previous](#)

[ [Guide contents](#) | [Appendix A contents](#) | [Next section: A4: Writing the rules file](#) | [Previous section: A2 Implementing two-level rules as finite state machines](#) ]

# A4 Writing the rules file



[ [Guide contents](#) | [Appendix A contents](#) | [Previous section: A3 Compiling two-level rules into state tables](#) ]

---

## [A4.1 The ALPHABET](#)

## [A4.2 NULL, ANY, and BOUNDARY symbols](#)

## [A4.3 Subsets](#)

## [A4.4 Rules](#)

## [A4.5 Example of a rules file](#)

### [Figure A8 The skeleton of a PC-KIMMO rules file](#)

### [Figure A9 Sample rules file](#)

---

This section contains instructions on how to write the rules file for the PC-KIMMO program (a more detailed specification of the rules file is found in section [4.7.1](#)). We will develop a sample rules file for a set of hypothetical data.

The general structure of the rules file is a list of declarations composed of a keyword followed by data. The set of valid keywords in a rules file includes COMMENT, ALPHABET, NULL, ANY, BOUNDARY, SUBSET, RULE, and END. The COMMENT, SUBSET and RULE declarations are optional and also can be used more than once in a rules file. The END declaration is also optional, but can only be used once.

The COMMENT declaration (new in PC-KIMMO version 2) sets the comment character used in the rules file, lexicon files, and grammar file. The COMMENT declaration can only be used in the rules file, not in the lexicon or grammar file. The COMMENT declaration is optional. If it is not used, the comment character is set to ; (semicolon) as a default.

The ALPHABET declaration must either occur first in the file or follow one or more COMMENT declarations only. The other declarations can appear in any order. The COMMENT, NULL, ANY, BOUNDARY, and SUBSET declarations can even be interspersed among the rules. However, these declarations must appear before any rule that uses them or an error will result.

To begin creating a rules file, use your text editor or word processing program to create a file with the extension .RUL (for example, SAMPLE.RUL). When you save the file to disk, be sure to save it as plain text (ASCII). We also recommend that you use an editor that handles column blocks; this makes manipulating state tables much easier. Type the basic skeleton of a PC-KIMMO rules file as shown in figure [A8](#) (a template of a rules file is also available in the file RULES.RUL on the PC-KIMMO release diskette):

### **Figure A8** The skeleton of a PC-KIMMO rules file

```
COMMENT
ALPHABET
NULL
```

```

ANY
BOUNDARY
SUBSET
RULE
END

```

Comments can be added to the rules file that are ignored by PC-KIMMO. The default comment delimiter character is semicolon (;), but can be changed by using the COMMENT declaration. Anything on a line following a semicolon is considered a comment and is ignored. Extra spaces and blank lines are also ignored.

## A4.1 The ALPHABET

The rules file must first declare the alphabet. This is the entire set of symbols (characters), both lexical and surface, used by the rules and lexicon. The ALPHABET declaration must either occur first in the file or follow one or more COMMENT declarations only. It is followed by any number of lines of symbols, each separated by at least one space. For example,

```

ALPHABET
  p t k b d g m n ng ɸ j s S z Z h l r w y
  i e a o u ï ë ä ö ü
  + '

```

The alphabet can consist of any alphanumeric characters, including those available in the extended character set on IBM PC-compatible computers. Uppercase and lowercase are considered distinct characters. Nonalphabetic characters such as \$, &, !, ', #, and + may also be used. In the above alphabet, + indicates a morpheme boundary and ' indicates stress. In this section, the examples printed in typewriter style use only those characters available on IBM PC compatible computers.

An alphabetic symbol can also be a multigraph, that is, a sequence of two or more characters. The individual characters composing a multigraph do not necessarily have to also be declared as alphabetic characters. For example, an alphabet could include the characters s and z and the multigraph sz%, but not include % as an alphabetic character. Note that a multigraph cannot also be interpreted as a sequence of the individual characters that comprise it. For example, if you declare t, h, and th as alphabetic symbols, then the th in a word such as rathole will match only the digraph th, not the sequence t plus h.

## A4.2 NULL, ANY, and BOUNDARY symbols

Next, the NULL (empty or zero) symbol is declared. Any character not already in the alphabet can be chosen, but for obvious reasons 0 (zero) is typically used. The NULL symbol is used for deletions, for instance h:0, and insertions, for instance 0:h. The NULL symbol is declared by including this line:

```

NULL 0

```

Next, the ANY ("wildcard") symbol is declared. Again, any character not already in the alphabet can be chosen; in this book we use @ ("at" sign). The ANY symbol is declared by including this line:

```

ANY @

```

Next, the BOUNDARY (word boundary) symbol is declared. Again, any character not already in the alphabet can be chosen; in this book we use # (crosshatch or pound sign). The BOUNDARY symbol is declared by including this line:

```

BOUNDARY #

```

## A4.3 Subsets

Next in the rules file the subsets, if any, are declared. A subset declaration is composed of the keyword

SUBSET followed by a subset name followed by a list of subset characters. A subset name can be any alphanumeric string (one or more characters, no spaces) so long as it is unique; that is, it cannot be a single character already declared in the alphabet. Uppercase characters are useful for subset names because they are usually distinct from their lowercase equivalents. All characters defined as belonging to a subset must also be in the complete alphabet. Subsets are declared by including lines such as these:

```
SUBSET C      p t k b d g m n ng ç j s S z Z h l r w y
SUBSET V      i e a o u
SUBSET Vlng   İ ě ä ö ü
SUBSET Cvd    b d g m n ng z l r w y
SUBSET Oalv   t d s z
SUBSET Opal   ç j S Z
SUBSET Ovd    b d g z
SUBSET Ovl    p t k s
```

## A4.4 Rules

The rest of the rules file consists of the rules. A rule declaration is composed of the keyword RULE followed by the rule name, number of states, number of columns, and the state table itself. The rule name is enclosed in a pair of identical delimiter characters such as double quotes. The rule name has no effect on the operation of the table. It actually can contain any information, but by convention we use it for the name and the two-level notation of the rule. It is also useful to include a sequence number for each rule, as rules are referred to by number in some of the diagnostic displays for rule debugging. Notice that the horizontal and vertical lines printed in the tables shown in this chapter are not present in an actual rules file.

By common convention, the first rules listed are the tables of default correspondences, though these correspondences can be listed anywhere in the file. For the sake of consistency, it is best to place all the default correspondences in these tables even if they also occur in other tables. The possible redundancy has no effect on the operation of the tables. Tables of default correspondences for the alphabet given in section [A4.1](#) look like this:

```
RULE "1 Consonant defaults" 1 17
      p t k b d g m n ng s z h l r w y @
      p t k b d g m n ng s z h l r w y @
1: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

RULE "2 Vowels and other defaults" 1 8
      i e a o u ' + @
      i e a o u ' 0 @
1: 1 1 1 1 1 1 1 1
```

The two tables could be combined into one; consonants and vowels have been separated here for increased readability. Notice that the morpheme boundary symbol (+) is deleted by default; that is, it has no surface realization other than 0.

After the tables of default correspondences come the rules for special correspondences. The rules developed below account for examples such as these:

```
LR:   s'ati   s'adi   bab'at   bab'ad
SR:   s'açi   s'äji   bab'at   bab'ät
```

These rules account for Palatalization:

```
RULE "3 Palatalization correspondences" 1 5
      t d s z @
      ç j S Z @
1: 1 1 1 1 1

RULE "4 Palatalization, Oalv:Opal lt;= gt; ____i" 3 4
      Oalv Oalv i @
```

```

      Opal  @    i  @
1:   3     2    1  1
2:   3     2    0  1
3.   0     0    1  0

```

Rule 4 is a palatalization rule that states that the alveolar consonants are realized as palatalized consonants before *i*. Because rule 4 uses subsets, the feasible pairs represented by the correspondence *Oalv:Opal* must be explicitly declared. Rule 3 contains these correspondences which are relevant only to rule 4. The special correspondences from all the rules in the description could be combined into one table (or they could even be combined with the tables of default correspondences). However, for readability and to make it easier to modify and debug the rules file, we recommend that a separate table of special correspondences be kept with each rule that uses subsets.

Rules 5 and 6 state that vowels are lengthened when they are stressed (that is, follow *'*) and precede a lexical voiced consonant (that is, a member of the subset *Cvd*).

```

RULE "5 Lengthening correspondences" 1 6
      a e i o u @
      ä ë ï ö ü @
1: 1 1 1 1 1 1

RULE "6 Vowel Lengthening, V:Vlng lt;= gt; '___Cvd:" 4 5
      ' V      V Cvd @
      ' Vlng @  @  @
1: 2 0      1  1  1
2: 2 4      3  1  1
3: 2 1      1  0  1
4. 0 0      0  1  0

```

The environment of rule 6 contains the correspondence *Cvd:@* rather than *Cvd:Cvd* because of rules 7 and 8, which device obstruents word finally.

```

RULE "7 Devoicing correspondences" 1 5
      b d g z @
      p t k s @
1: 1 1 1 1 1

RULE "8 Final Devoicing, Ovd:Ovl lt;= gt; ___#" 3 4
      Ovd Ovd # @
      Ovl  @  # @
1:  3   2  1  1
2:  3   2  0  1
3.  0   0  1  0

```

The rules file optionally ends with a line containing only the word **END**. Any material in the file after this line is ignored by PC-KIMMO.

END

## A4.5 Example of a rules file

As a ready model for the format of a rules file, the example developed above is repeated in its entirety in figure [A9](#). This file is found on the PC-KIMMO release diskette in the **SAMPLE** subdirectory.

**Figure A9** Sample rules file

```

ALPHABET
  p t k b d g m n ng ç j s S z Z h l r w y
  i e a o u i ä ë ä ö ü
  + '
NULL 0
ANY @
BOUNDARY #
SUBSET C      p t k b d g m n ng ç j s S z Z h l r w y

```

```

SUBSET V      i e a o u
SUBSET Vlng   i ë ä ö ü
SUBSET Cvd    b d g m n ng z l r w y
SUBSET Oalv   t d s z
SUBSET Opal   ç j S Z
SUBSET Ovd    b d g z
SUBSET Ovl    p t k s
END

RULE "1 Consonant defaults" 1 17
      p t k b d g m n ng s z h l r w y @
      p t k b d g m n ng s z h l r w y @
1: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

RULE "2 Vowels and other defaults" 1 8
      i e a o u ' + @
      i e a o u ' 0 @
1: 1 1 1 1 1 1 1 1

RULE "3 Palatalization correspondences" 1 5
      t d s z @
      ç j S Z @
1: 1 1 1 1 1

RULE "4 Palatalization, Oalv:Opal lt;= gt; ___i" 3 4
      Oalv Oalv i @
      Opal  @   i @
1: 3      2   1 1
2: 3      2   0 1
3. 0      0   1 0

RULE "5 Lengthening correspondences" 1 6
      a e i o u @
      ä ë ï ö ü @
1: 1 1 1 1 1 1

RULE "6 Vowel Lengthening, V:Vlng lt;= gt; '___Cvd:" 4 5
      ' V      V Cvd @
      ' Vlng @  @  @
1: 2 0      1 1 1
2: 2 4      3 1 1
3: 2 1      1 0 1
4. 0 0      0 1 0

RULE "7 Devoicing correspondences" 1 5
      b d g z @
      p t k s @
1: 1 1 1 1 1

RULE "8 Final Devoicing, Ovd:Ovl lt;= gt; ___#" 3 4
      Ovd Ovd # @
      Ovl  @  # @
1: 3      2 1 1
2: 3      2 0 1
3. 0      0 1 0

END

```

---

[Up](#)
[Next](#)
[Previous](#)

[\[ Guide contents | Appendix A contents | Previous section: A3 Compiling two-level rules into state tables \]](#)