Here we explain how two-level rules work, how
they can be implemented as finite-state machines, and
all the types of rule constraints can be translated
into finite-state tables. We then summarize the rule
semantics. This is followed by a detailed discussion
of rule conflicts; specficity and conflicts amongst
SUBSETS; and finally, and explanation of the rule
file format and the rules
in the pc-kimmo file english.rul.

It's a lot to read through... but I hope, complete,
and will guide you through Spanish.

1. How two-level rules work.

Consider Rule 2 (R2) below.

R2   t:c ==> ____i

The operator ==> means that lexical t is realized as a surface c only
(but not always) in the environment preceding i:i.

The correspondence t:c declared in R2 is a special correspondence.
All two-level descriptions must also contain a set of *default*
correspondences, such as t:t, i:i, etc. (This is the so-called
"BOGUS RULE" - it isn't really bogus, it is a default.)
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 R2 (and all default
correspondences) is applied to the lexical (underlying) form "tati"
(without the quote marks) PCKIMMO proceeds as follow to produce the
corresponding surface form(s). (NOTE this is why you can use
GENERATE without a dictionary and JUST the .rul file)

Beginning with the first character of the input form, it looks to see
if there is a correspondence declared for it. Due to R2, it will find
that lexical t can correspond to surface c, so it will begin by
positing that correspondence.

```
Lexical:   t   a     t   i
           |   |     |   |
Rule:     R2
           |
Surface:   c
```

At this point the generated has entered R2. For the posited t:c
correspondence to succeed, the generator MUST find an i:i
correspondence next - that is what R2 says. When the generator moves
on to the second character of the input word, it finds that it is a
lexical a, and thus R2 FAILS, so the generator must back up, undo
what it has done so far, and try to find a different path. Backing up
to the first character t, it now tries the DEFAULT correspondence t:t
(which is guaranteed to succeed, since it has NO conditions):

```
Lexical:   t   a     t   i
           |   |     |   |
Rule:     R2
           |
```

```
Surface:   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:

```
Lexical:   t   a     t   i
           |   |     |   |
Rule:     R2   |
           |   |
Surface:   t   a
```

Moving on to the third character, the generator again finds a lexical t, so it posits a surface c and enters R2 again:

```
Lexical:   t   a     t   i
           |   |     |   |
Rule:     R2   |    R2
           |   |     |
Surface:   t   a     c
```

Now the generator looks at the fourth character, a lexical i. This SATISFIES the environment of R2, so it keeps the i (NOTE that the constraint refers only to a surface i, and says nothing about the lexical, underlying character):

```
R2   t:c ==> ____i
```

Since the context of R2 requires an i, the generator must also posit a surface i, so it does, and exits R2. NOTE that by the time R2 is finished, TWO characters will have been posited.

```
Lexical:   t   a     t   i
           |   |     |   |
Rule:     R2   |    R2   |
           |   |     |   |
Surface:   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 yet done. 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 consider the third character, lexical t. Having already tried the correspondence t:c, it will try the default correspondence t:t:

```
Lexical:   t   a     t   i
           |   |     |
Rule:     R2   |     |
           |   |     |
Surface:   t   a     t   i
```

Now the generator will try the final correspondence and succeed, since R2 does NOT prohibit t:t before an i (rather, it prohibits t:c in any environment EXCEPT BEFORE i). It will then output "tati". The reader may confirm that no other outputs will be generated.

2. The ==> rule as a finite-state machine.

A key insight of PCKIMMO is that if phonological rules are written as two-level rules, they can be implemented as FST's running in parallel. In the next 4 sections we briefly show how each of the four rule types (==>, <==, <==>, and \<==) translates to an FST. We then go on to describe conflicts in SUBSETS, and RULES.

2.1 A ==> rule.
Consider rule R2 again.

A possible paraphrase 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 must be incorporated into our two-level FST, call this T2 (for table 2)

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

The @:@ arc means ANY OTHER symbol than t, i, or c, i.
State 2 is a kind of 'default'state that ignores everyting except the substring crucial to the rule. It is also the only final, accepting state.

Importantly, the state table is constructed such that the entire set of feasible pairs in the rule description is partition among the column headers WITH NO OVERLAP (this is the source of MANY bugs in Kimmo rule systems). T2 specifies the special correspondence t:c and the environment in which it is allowed. (the machine goes to state 2 to anticipate that an i:i comes next - if it does, success, and goes to state 1; if not, it goes to state 0, the rejecting state.)

The column header @:@ in T2 matches ALL the feasible pairs that are defined by ALL THE OTHER FSTs of the system - thus saying that R2 'takes a pass' and doesn't care about any other feasible pairs. So, with respect to T2, @:@ does not stand for all feasible pairs, rather, all feasible pairs except i:c and i:i.

The default correspondences of the system must be declared in a trivial FST like T3: (also see below where we cover the .rul file format). If we assume p, t, k, a, i, u in our alphabet, then we need:

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

(Table T3)

Even this table of correspondences must include @:@ as a column. Otherwise, it would fail when it encountered a special correspondence such as t:c, because all the rule 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, T2 and T3 together will generate the surface forms tatik and tacik.

IMPORTANT. To understand how to represent two-level rules as state tables, we must understand what the rules really mean. It is a common tendency to think of them positively, that is, as statement of

where the correspondence succeeds. IN FACT STATE TABLES ARE FAILURE DRIVEN, THEY SPECIFY WHERE THE CORRESPONDENCES MUST FAIL.

This point is perhaps THE biggest source of difficulty in building the FSTs.

In our case above, it is natural to think of R2 as saying that t:c succeeds when it occurs preceding i:i. But T2 actually works because it FAILS when ANYTHING BUT i:i follows t:c.

2.2 A <== rule.

Now consider R4.

R4 t:c <== ____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, 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 following i:i.
Remember that state tables are failure-driven, so the strategy of writing the state table for R3 is to force it to fail if it recognizes the sequence t:t i:i. So the state table for R4, viz., T4, looks like this:

T4

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

In state 1, any occurrences of the pairs t:c, i:i, or any other feasible pairs are allowed without leaving state 1. It is only the correspondence t:t that forces a transition to state 2, where all feasible pairs succeed except i:i. Note that state 2 must be a final state - this allows all the correspondences to succeed and return to state 1. Also note that in state 2 the cell under the t:t column contains a 2. This is necessary to allow for the possibility of a tt sequence in the input. For example, tatti will surface as the form tatci. This phenomenon is called "backlooping" - more on this below.

Actually T4 is potentially over-specified. It is not really the pair t:t that is disallowed before i, but rather the pair t:not-c (lexical t and surface anything but c) Given that the more specific correspondence t:c is already in the table, the more general correspondence t:@ will take care of all the rest of the characters, including t:t. (I'll leave the details of this to you..)

In summary, the rule type L:S <==E positively says that L is ALWAYS realized as S in the environment E. Thus, it is a kind of OBLIGATORY rule. Negatively, it says that L is realized as any character but S is not allowed in E. The state table must be written so that it forces all correspondences of L with anything BUT S to fail.

2.3 A <==> rule.

R5 t:c <==> ____i

The state table for a <==> rule is simply the combination of the tables for ==> and <==. You build it by anding the two fst's together. So here, t:c MUST occur before i, and NOWHERE ELSE.

We next turn to the problem of what can happen when you have more than one rule - rule conflicts, the use of SUBSETS, and overlapping character descriptions.

3.0 Writing rules: conflicts, SUBSETS, and character descriptions.

3.1 SUBSETS