1. While I find the gemination rules utterly incomprehensible in spite of their internal regularity (suggesting to me that perhaps this is *not* the best representation), it appears that the key issue is accent sensitivity -- refer+ing ==> refering, but re`fer+ing ==> referring

2. As implemented, the recognizer does not make a single left-to-right pass but rather can back-track due to nondeterministic choices (both correct and incorrect), potentially many times. While it would be possible to simultaneously follow all branches of the nondeterminism, buffering up the different alternative outputs, this still wouldn't eliminate the possibility of an exponential number of parses. In this particular example, the problem seems to be due to the lexicon mis-recognizing 'fly' as a noun, for which +er is not a valid suffix. It then has to back-track and re-recognize 'fly' as a verb. This particular problem cannot occur during generation, because the lexicon is not consulted, but other problems involving FSA rule nondeterminism can still occur.

(Consider how the lexical form "xxxxx" is processed by the FSA

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

which produces an exponential number of failing parses before producing "yyyyy".)

4. 1) The N->m FSA backs up to the N after announcing success because the generator made a nondeterministic choice between following the N:n and N:m paths. Although in this case they are mutually exclusive, that is not always true, and back-tracing may yield another valid parse.

2) boundary: '#'
defaults: "a b c d e f g h i j k l m n o p q r s t u v w x y z + # 0"
subsets:
"@": "a b c d e f g h i j k l m N n o p q r s t u v w x y z + # 0"
rules:
N realized as m:
  start:
  'N:n': two
'N:m': three
'@:@': start
two:
'N:n': two
'N:m': three
'p:@': reject
'@:@': start
rejecting state three:
'p:@': start
'@:@': reject

p realized as m:
start:
'@:m': two
'p:m': reject
'@:@': start
two:
'@:m': two
'p:m': two
'p:p': reject
'@:@': start

>>> print list(load("rule12_mod.yaml").generate('kaNpat', TextTrace(3)))

['kammat']

3) The independent rules method would appear to be much more easily extended when
the independent rules are in fact "independent" -- that is, they have applicability
apart from their mutual interaction -- because the straightforward generalizations
happen automatically, and unnecessary state machine products don’t need to be
expanded. On the other hand, when the rules involve inherently complicated cross
coupling, it may be difficult or impossible to factor them cleanly.
FSA-coded rules are never particularly clear and transparent. In the simple case, unified rules are perhaps somewhat more transparent, because the few interactions the rules have are manifestly laid out. But, in the complex case, when there are many interactions, this breaks down as the product FSA becomes unnecessarily complex. There, independent rules are more transparent, because each interaction does not need to be explicitly expanded.

4) As the number of interacting rules grows, how well the system holds up would depend on the kinds of interactions the rules had. Rules that naturally followed a long pipeline might fare very poorly under two-level morphology, because the independent rules method would no longer be an option; chains of logically separate rules would have to be collapsed into just two layers. On the other hand, rules that had a shallow interaction depth but interacted with many neighbors would likely be efficient and straightforward to represent.