

Finite State Morphology: A tutorial

OUTLINE

PART I:

1. Finite state technology
2. Regular expressions
3. Finite state linguistics

PART II:

4. Morphological analysis
 5. The *lexc* language
 6. Issues in morphophonology
-

PART I

1. Finite State Technology

1.1 Introduction

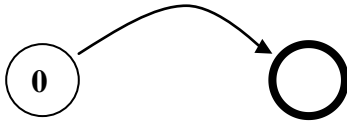
- Finite State Machines and their properties are well known mathematical objects. However, their uses for natural language processing have only really been explored since the mid 80s.
- The current effort is pioneered by: Kenneth Beesley, Ronald Kaplan, Lauri Karttunen, Martin Kay, Kimmo Koskenniemi.
- Finite State Lexical Transducers are mathematically well understood. Applications to natural language are efficient, robust and elegant.
- Applications: spell-checking, part-of-speech disambiguation, tokenization, shallow parsing, and specifically morphological analysis.

1.2 The Basics

In order to understand how to build the linguistic applications, we first need to get acquainted with the basics of how finite-state machines work.

The basic concepts are very simple, but the possibilities allowed by composing different transducers become extremely complex (and versatile) very quickly.

Finite state machine / Finite state network: A network consisting of *states*, including one *start state* and one or more *final states*. *Transitions* between states are possible only if the required input is recognized. *Path* is a sequence of transitions over arcs to a particular state.



Initial State (leftmost state) Final State

► *Example:* The light switch

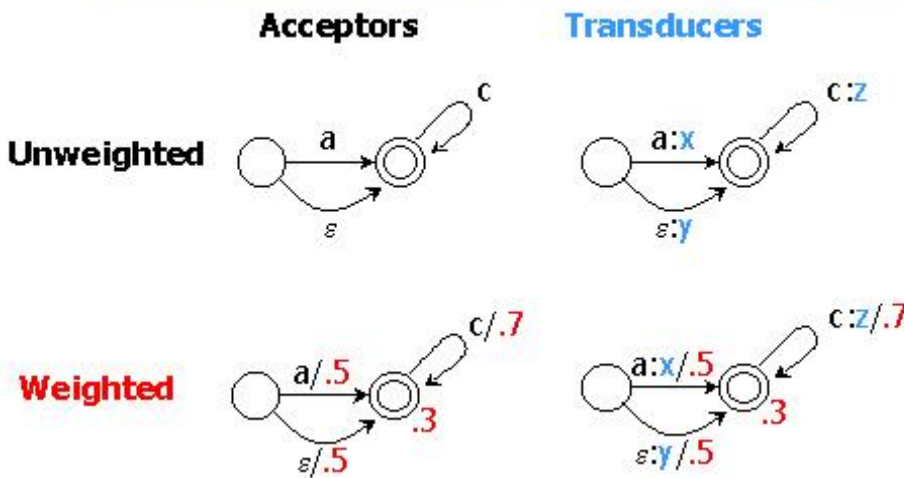
FSMs are FINITE in the sense that they can model any finite number of states, but not an infinite gradation of states.

Example: While an FST can model a light switch, it could not model a light dimmer.

Types of FSMs¹

- **Finite state automaton:** FSM that only accepts a set of given strings (a language) → describes languages.

- **Finite state transducer:** FSM that provides a set of outputs from an accepted input → expresses *relations* between languages.

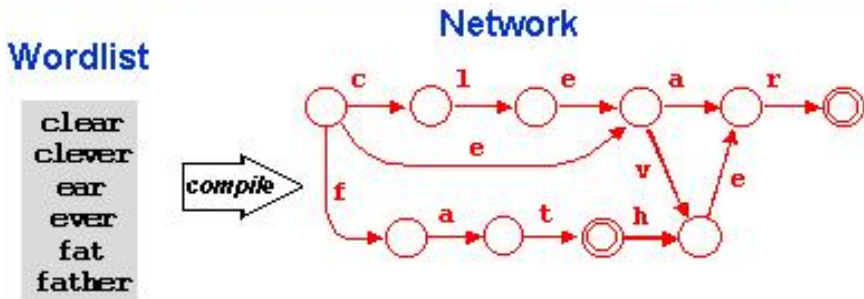


Example of a FSA:

- Can be used for lookup (analysis) → Symbol matching: Either accepts or rejects the input
- Result is accepted only if the word is in the language of the network

Note: analysis of *coffee* or *table* or *fathom* will fail. Analysis of illegal word *fath* will also fail.

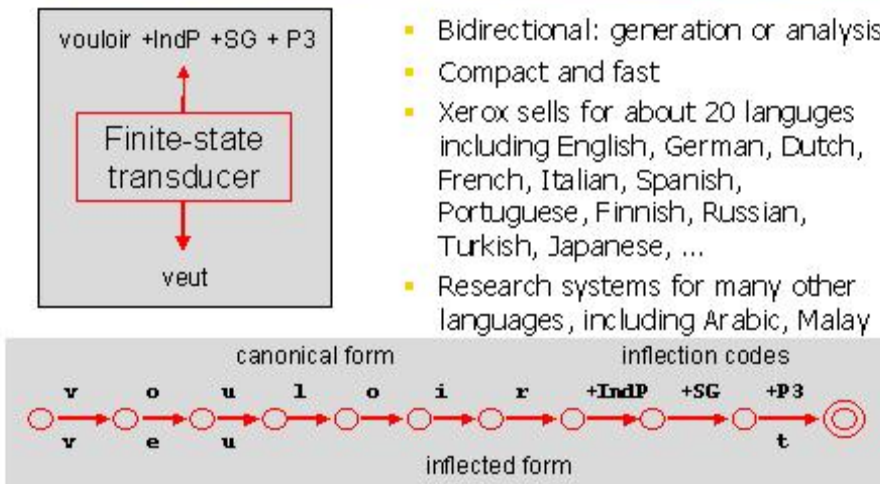
¹ The gif images used in this document were obtained through a Google search. If you are the author of these images, please contact me so that I can provide the appropriate authorship information.



Example of a FST:

- Can do analysis (lookup) and generation (lookdown)
- Input is the lower side symbols on the path and output is the upper side symbols → establishes the relation between strings

Note: Tags or Symbols like +Noun or +Verb are *arbitrary*: the naming convention is determined by the (computational) linguist and depends on the larger picture (type of theory/type of application).



- Bidirectional: generation or analysis
- Compact and fast
- Xerox sells for about 20 languages including English, German, Dutch, French, Italian, Spanish, Portuguese, Finnish, Russian, Turkish, Japanese, ...
- Research systems for many other languages, including Arabic, Malay

Analysis (lookup) Process:

- Start at the Start State
- Match the input symbols of string against the *lower-side* symbol on the arcs, consuming the input symbols and finding a path to a final state.
- If successful, return the string of *upper-side* symbols on the path as the result.
- If unsuccessful, return nothing.

Generation (lookdown) Process:

- Start at the Start State
- Match the input symbols of string against the *upper-side* symbol on the arcs, consuming the input symbols and finding a path to a final state.
- If successful, return the string of *lower-side* symbols on the path as the result.

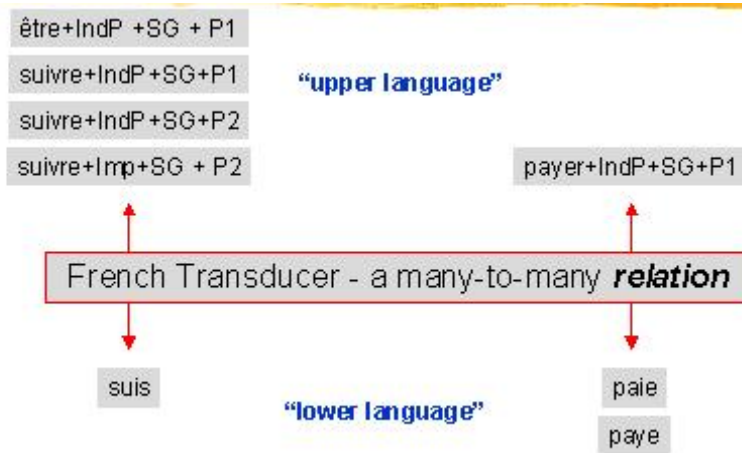
- If unsuccessful, return nothing.

Some more terminology:

The set of valid symbols the machine accepts is its ALPHABET or SIGMA .
 The sequences of symbols are WORDS.
 The entire set of words is the LANGUAGE.

Ambiguous analyses

► *Example: two possible paths - tables*



A weighted version of the FST to resolve ambiguities in analysis:



1.3 Operations on FSTs

Union

The union of two networks S1 and S2 is another set that contains all the elements in S1 and all the elements in S2. There is no ordering of the arcs in the network. $\blacktriangleright [A \mid B]$

► *Example:* See figure above for union of the network for the language {"clear", "clever", "ear", "ever"} and the network for the language {"father", "fat"}.

Concatenation

One can also concatenate two existing languages (finite state networks) with one another to build up new words productively/dynamically. $\blacktriangleright [A B]$

► *Example:* concatenation of *work* and tense morphology

Note: Problem with things like **trys*, **tryed*, though *trying* is okay. One has to write extra rules to avoid these errors.

Composition

Composition is an operation on two relations $\blacktriangleright [A .o. B]$

Composition of the two relations $\langle x,y \rangle$ and $\langle y,z \rangle$ yields $\langle x, z \rangle$

► *Example:* $\langle \text{"cat"}, \text{"chat"} \rangle$ with $\langle \text{"chat"}, \text{"Katze"} \rangle$ gives $\langle \text{"cat"}, \text{"Katze"} \rangle$

Composition forms a sequence of transducers. Builds a cascade of FSTs into a single one by eliminating the common intermediate outputs \rightarrow It allows for a modular structure.

Intersection

The intersection of two networks contains the set containing all the members that are common to both. $\blacktriangleright [A \& B]$

Intersection of $\langle x, y, z \rangle$ with $\langle a, b, z \rangle$ yields $\langle z \rangle$

Note: applies only to simple networks

Subtraction

The subtraction of two networks contains the set containing all the members that are common to both. $\blacktriangleright [A - B]$

$\langle x, y, z \rangle$ minus $\langle z \rangle$ yields $\langle x, y \rangle$

Note: applies only to simple networks

Complementation (or Negation)

The complement language of the network A is the set of all strings that are not in the language A. $\blacktriangleright \sim A$

Note: applies only to simple networks

2. Regular Expressions

To represent a (finite) language, we can use *regular expressions* (or RegExp), which compile into a finite state network.

- Most popular use is to search for patterns in a corpus.
- Once compiled, the set of strings can be encoded by a finite-state automaton, and the relations between strings can be encoded by a finite-state transducer.

Epsilon	0	Empty string
Any	?	Any single symbol string
Boundary	#	Beginning or end of a string
Optionality	()	
Character	a	Any 'a'
Kleene-plus	a+	Iteration of A one or more times
Kleene-star	a*	Iteration of A zero or more times
	[A-Z]	Any upper case letter
	[0-9]	Any single digit

► *Example:* the sheep language: [baa+!]

Special Symbols

- Symbols that have a special meaning in regular expressions must be preceded by % or it should be enclosed in “”. These symbols are: 0, ?, +, *, |, &, -, %, #, ; etc.
- “\n” = new line; “\t” = tab
- Encodings are preceded by slash in double quotes: “\u0633”

Multicharacter Symbols

Morphological and syntactic tags are usually represented by multicharacter symbols that convey information about part-of-speech, tense, number, gender, etc.

To treat +**Noun** as an atomic entity, it has to be defined as a multicharacter symbol. In a regular expression, it has to be specified as %+Noun or “+Noun”

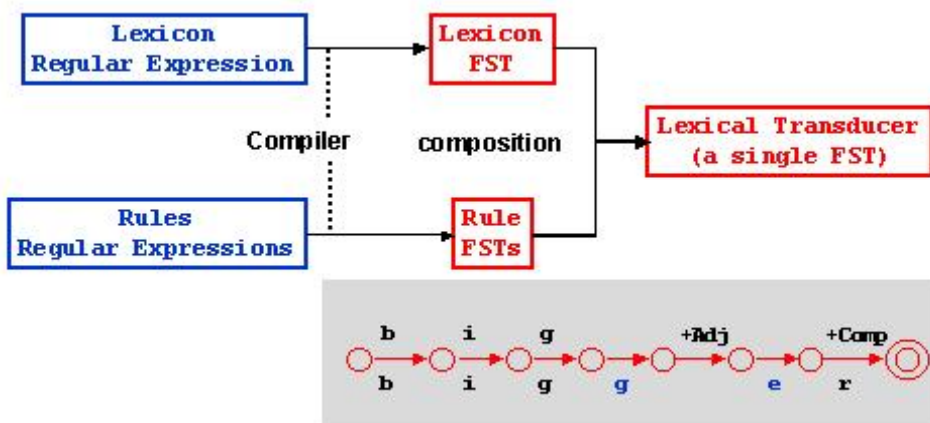
3. Finite-State Linguistics

- A set of strings, a LANGUAGE, can be represented as a simple finite-state network that consists of states and arcs that are labeled by atomic symbols. Each string (= *word*) of the language corresponds to a path in this network.
- A set of ordered pairs of strings, a RELATION, can be represented as a finite-state transducer. The arcs of the transducer are labeled by symbol pairs. Each path of the transducer represents a pair of strings in the relation. The first string of each ordered pair belongs to the UPPER language, the second string belongs to the LOWER language of the relation.
- New languages and relations can be constructed by set operations such as UNION, CONCATENATION, and COMPOSITION. These operations can also be defined for finite-state networks to create a network that encodes the resulting language or relation. Some network operations such as INTERSECTION and SUBTRACTION can be defined only for languages, not for relations.

Things that FST can be used for

(work developed at Xerox PARC)

1. The legal combination of morphemes (morphotactics) can be encoded as a finite-state network.
2. The rules that determine the form of each morpheme (alternation) can be implemented as finite-state transducers.
3. A lexicon network and the rule transducers can be composed together into a single network (lexical transducer). This contains all the morphological information about a language (morphemes, derivation, inflection, compounding, etc.)



Steps to consider in building a morphological analyzer

The ultimate goals of any morphological analyzer are to accept and correctly analyze all valid words, and not to accept any invalid words.

1. Formal linguistics: a generalized description of the linguistic phenomena to be represented
 - inflectional patterns, productive derivation, morphotactics, and phonological alternations
 2. Application of system: morphological analysis is the basis for many applications
 - design decisions based on final application (e.g., noun phrase parsing, entity extraction, coverage of dialects)
 - keep design flexible (e.g., can use for generation also)
 3. Selection of tags and order of tags: determining which morphological features need to be coded and defining a standard combination and order of tags
 - a. Tags should be convenient: choose names that are easy to remember but not too long to type – linguistically accepted tags are recommended (e.g., Sg, Pl, Nom, Acc, Gen).
 - b. Don't use tags if the linguistic phenomenon doesn't exist in the language. Do not try to force your language into another language's descriptive framework. (e.g., Nom and Gen are probably not useful for Persian).
 - c. If a set of words act the same syntactically, then they should probably be analyzed with the same tags. If two words act differently in the syntax, they should probably be analyzed with distinct tags (e.g., superlative and comparative adjectives in Persian should have different tags).
 - d. It's better to have more information than less, since it's easy to
 - remove unwanted tags for a particular application
 - change tag names later, using replace rules and composition
 4. Modular design: organizing rules and lexicons in separate FSTs allows for easier testing and debugging.
-

PART II

4. Morphological Analysis

Finite-State Morphology/Phonology:

1. Morphotactics,
2. Phonological/Orthographical Variation

Morphotactics. In word formation, *morphemes* can only appear in a certain order and in certain combinations:

<i>piti-less-ness</i>	<i>*piti-ness-less</i>
<i>un-guard-ed-ly</i>	<i>*un-elephant-ed-ly</i>

→ Morphotactics captured in *lexc*

Alternations. Phonological and orthographic alternations across morpheme boundaries

pity → *piti* when followed by *less*

fly → *flie* when followed by *s*

swim → *swimm* when followed by *ing*

→ Alternations captured in *replace rules*

Simple Replace Rules

(Extended Regular Expressions)

► *Example:* the kaNpat exercise

Consider a fictional language, where *kaNpat* is an abstract lexical string consisting of the morpheme *kaN* (containing an underspecified nasal morphophoneme *N*) concatenated with the suffix *pat*. Here, as in many natural languages, strange things happen at morpheme boundaries. It so happens that in this language, an underspecified nasal **N** that occurs just before *p* gets REALIZED as (or “replaced by”) an **m**. A second rule in this language states that a **p** that occurs just after an **m** gets realized as **m**. Formalize this in replace rules.

► *Example:* *trys, *tryed, though *trying* is okay.

5. The LEXC language

Finite state programming languages: **lexc** and **xfst** are notations that compile into finite-state networks

- **xfst** (Xerox Finite State Technology)

- core Xerox tool providing interface to the finite state calculus for building and manipulating FSMs

- provides a compiler for regular expressions and replacement rules → good for expressing phonological and orthographic alternations

- not optimized and cannot handle huge unions

- **lexc** (Lexicon Compiler)

- convenient for defining natural language lexicons and morphotactics

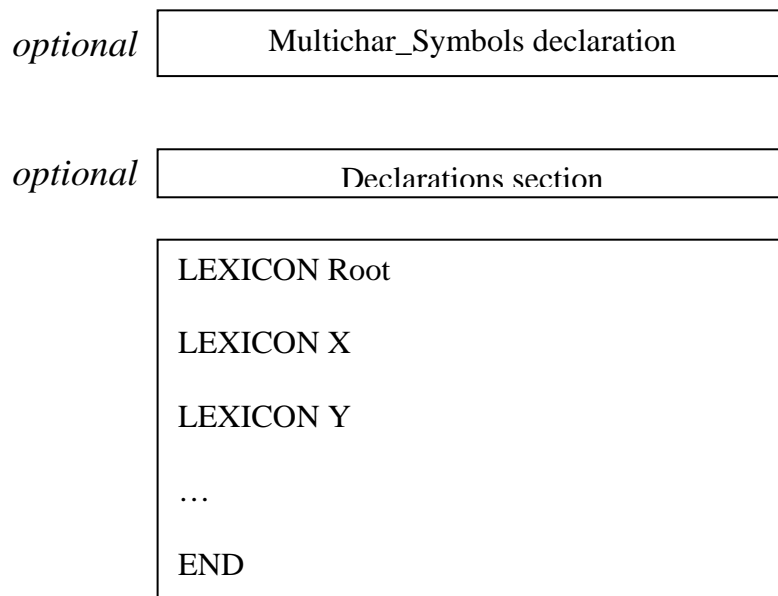
- optimized to handle huge lexical unions efficiently

- can encode compounding, irregularities, recursion

lexc files

lexc source files are declarative statements about the morphotactics of a language, and they are compiled into finite-state networks, which are data structures rather than procedural programs. The order in which the LEXICONS are written is not significant.

Note: To put regular expressions in a lexc file, use < >. All notations within angle brackets are compiled like the regular expressions in the xfst program.



! example-lex1 (this line is a comment)

Multichar_Symbols +Verb +Noun +Pl +Sg
+Pres +Past +Gerund

Definitions

Vowel = a | e | o | u | i ;

LEXICON Root ! empty string branches to various LEXICONS
Nouns ; ! There is no Form here (empty string), only a Continuation Class
Verbs ;
IrregVerbs ;

LEXICON Verbs

walk V ; ! this is equivalent to walk:walk
talk V ;
pack V ;

LEXICON Nouns

table N ;
dog N ;

LEXICON IrregVerbs

swim:swam V2 ;
go:went V2 ;

LEXICON V

%+Verb:0 Infl ;

LEXICON N

%+Noun:0 Number ;

LEXICON V2

%+Verb:0 Infl2 ;

LEXICON Number

%+Pl:s # ;
%+Sg:0 # ;

LEXICON Infl

%+Pres:s # ;
%+Past:ed # ;
%+Gerund:ing# ;
;

LEXICON Infl2

%+Past:0 # ;

Compiling the source file using xfst

```
xfst[0]: read lexc < example-lex1.txt
Opening 'example-lex.txt' ...
Root...3, Verbs...3, Nouns...2, IrregVerbs...1, V...1,
N...1, V2...2, Number...2, Infl...3, Infl2...1
Building lexicon...Minimizing...Done!
SOURCE: 23 states, 34 arcs, 23 paths.
xfst[1]:
```

Lexical: walk+Verb+Past
Surface: walked

Lexical: pack+Verb+Gerund
Surface: packing

Lexical: dog+Noun+Sg
Surface: dog

Lexical: swim+Verb+Past
Surface: swam

6. Issues in morphophonology

6.1 Issues for a linear morphotactic approach

lexc can capture the dependencies between a morpheme and the morphemes that can appear next, but if there are dependencies or restrictions on non-adjacent morphemes, the *lexc* formalism cannot really handle it with the continuation classes.

Non-concatenative processes:

- Separated dependencies = Long-distance dependencies = Discontiguous dependencies
example: see last page for Arabic example
→ use *flag diacritics* or *finite-state filters*
- Interdigitation
example: Arabic template morphology (dars, dorus, madrase, madAres, modarres, etc.)
- Reduplication
example: *dars-mars; ashxal-pashxal*
example: pili → pipili, kuha → kukuha (Tagalog)
- Infixation
example: Persian causative morpheme (*tarsAndan*)

6.2 Exercises in Persian

► *Example: Persian inflectional morphology on nouns*

Description of surface forms of Ezafe:

- appears as y after a noun ending in a vowel (Sday)
- appears as detached y after a noun ending in silent ‘h’ (xanh_y)
- not written after a consonant → its presence cannot be determined (ktab)

Morphotactics: Follows the plural morpheme

Phonological alternations on the plural morpheme ‘an’

Ezafe and noun phrase boundaries

!*****
! Nominal Inflectional Morphology in Persian
!*****

Multichar_Symbols

+Noun ! POS noun
+Pl ! Plural
+Sg ! Singular
+Ez ! Ezafe: linking morpheme for NP-internal constituents
+IndEncl ! same morpheme used for both the Indefinite article
! and the Enclitic which marks a relativized noun
+NPB ! marks noun phrase boundary
+NoNPB ! no noun phrase boundary
! Noun phrase boundary markers needed for NP extraction

!! Delimiters

^NB ! marks morpheme boundary for phonological rule

!*****

LEXICON Root

NounCons ; ! All these continuation classes are the lexicon for nouns
NounVowel ; ! and they lead to the "NounTag" Lexicons below
NounY ;
NounSilenth ;

LEXICON NounTagCons

%+Noun:0 NumberCons ;

LEXICON NounTagVowel

%+Noun:0 NumberVow ;

LEXICON NounTagY

%+Noun:0 NumberY ;

LEXICON NounTagH

%+Noun:0 NumberH ;

!*****

! Nominal Suffixes

!*****

! Pl Morpheme

! -----

! "ha" can appear on any noun

! "an" and its phonological variants can appear on any (animate) noun

! "yn" appears on nouns ending in consonants (Arabic participles)

! "at" appears on nouns ending in consonants or "y" (of Arabic origin)

! "vn" appears on nouns ending in "y"

! "at" and its phonological variants can appear on nouns ending in silent "h"

!

! NOTE: need to keep track whether word ends in consonant or vowel for next Suffix rule

!!-----

! Note: all ezafe after a consonant is undefined (EzafeCons). Eventually, we can skip the EzafeCons step.

LEXICON Number ! this ending can appear on any word
 %+Pl:%^NBan EzafeCons ; ! this should be only on animates
 %+Pl:_ha Ezafe ;
 %+Pl:ha Ezafe ;

LEXICON NumberCons
 %+Sg:0 EzafeCons ;
 %+Pl:at EzafeCons ;
 %+Pl:yn EzafeCons ;
 Number ;

LEXICON NumberY
 %+Sg:0 EzafeCons ;
 %+Pl:at EzafeCons ;
 %+Pl:vn EzafeCons ;
 Number ;

LEXICON NumberH
 %+Sg:%_y EzafeH ;
 %+Sg:0 EzafeH ;
 %+Pl:%^Nbat EzafeCons ; ! needs phonological rules
 Number; ! this is wrong since silent h can't take ha only _ha
 ! but don't want to create a special path for this

LEXICON NumberVow
 %+Sg:0 Ezafe ;
 Number ;

!-----

! EZAFE SUFFIX

!-----

! appears as "y" after a noun ending in a vowel
 ! appears as ";" after a noun ending in silent h
 ! is a short vowel following a noun ending in a consonant and therefore it is not written.
 ! Its presence cannot be determined after a consonant.
 ! NOTE: ezafe can appear on the root or after the plural.
 !
 ! Ezafe can be used to figure out the boundaries of the NP:
 ! - if ezafe is absent, there is a NP boundary (it's the end of the NP)
 ! - if ezafe is present, there is no NP boundary (the noun is linked to the following element)
 ! NOTE: if ezafe is not determined, we can't define the NP boundary either.
 !-----

LEXICON Ezafe ! For words ending in a vowel

```

%+Ez%+NoNPB:y # ; ! ezafe marked; no NP boundary; no more suffixes
IndefNPB ; ! no ezafe; NP boundary; check for other suffixes
CliticNPB ;

```

```

LEXICON EzafeCons ! For words ending in a consonant
IndefCons ; ! ezafe is undefined
CliticCons ;

```

```

LEXICON EzafeH ! For words ending in a silent 'h'
%+Ez%+NoNPB:%; # ; ! or hamze
%+Ez%+NoNPB:_y # ; ! or detached "y"
IndefCons ; ! ezafe is undefined if not written after "h"
CliticCons ;

```

```

#=====
# Phonological rules for nominal suffixes
#=====

```

```

#-----
# Allomorph Rules
#-----

```

```

define Vowel [ a | i | e | u | A ];

```

```

# Consonants or glides
define Cons [ b | p | t | B | J | C | H | x | d | D
             | r | z | j | s | G | S | Z | T | P
             | e | Q | f | q | k | g | l | m | n | v | h | y ];

```

```

# note: i = vowel pronunciation of 'y'
# example: qrbany^NBnd -> qrbany~and
# example: ayrany^NBy -> ayrany~ay

```

```

define detachY [ %^NB -> %_ a || i _ ];

```

```

define ialter [ i -> y ];

```

```

# example: qrbany^NBan -> qrbanyan
# example: anHSargray^NBan -> anHSargrayan

```

```

define plural [ %^NB -> 0 || i _ a n ];

```

```

# note: e = silent 'h'
# example: bynnde^NBan --> bynndgan

```

```

define replace1 [ e %^NB -> g || _ a n ];

```

```

# example: karxane^NBat --> karxanJat
# example: klme^NBat --> klmat

```

```
define replace2 [e %^NB -> [ J | 0 ] || _ a t ];
```

```
# the vowrule captures:
```

```
# example: danGJv + IndefEncl ==> danGJvyy
```

```
# example: danGJv + Plural (an) ==> danGJvyan
```

```
# example: danGJv + Copula ==> danGJvyyd
```

```
# example: ktab + Plural + IndefEncl ==> ktabhayy
```

```
define vowrule [%^NB -> y || Vowel _ ];
```

```
define consrule [%^NB -> 0 || Cons _ ];
```

```
etc.
```