Building Morphological Analysers and Generators for Indian Languages using FST

Amba Kulkarni
Department of Sanskrit Studies,
and
G UmaMaheshwar Rao
CALTS,
University of Hyderabad,
Hyderabad, India

PART II

Morphological Analyser:

A tool that takes a word in language L as an input and produces an analysis in terms of root and suffix along with various associated features.

e.g.

Language	Input:word	Output:analysis
Hindi	ladake	rt=ladakā, cat=n, gen=m, num=sg, case=obl rt=ladakā, cat=n, gen=m, num=pl, case=dir
Telugu	pillalaki	rt=pilla, cat=n, gen=m, num=pl, suffix=ki
Marathi	Garāsamora	rt=Gara, cat=n, gen=n, num=sg, suffix=samora

Morphological Generator:
A tool that takes a root along with a list of feature-value pairs as an input, and produces the corresponding word form.

The input thus will be the contents of column 3, and the output will be the contents of column 2.

Language	input:analysis	o/p:word
Hindi	rt=ladakā, cat=n, gen=m, num=sg, case=obl	ladake
Hindi	rt=ladakā, cat=n, gen=m, num=pl, case=dir	ladake
Telugu	rt=pilla, cat=n, gen=m, num=pl, suffix=ki	pillalaki
Marathi	rt=Gara, cat=n, gen=n, num=sg, suffix=samora	Garāsamora

Morphological analysis and generation: Inverse processes.

Analysis may involve non-determinism, since more than one analysis is possible.
Generation is a deterministic process.

In case a language allows variation, to that extent, generation also involves non-determinism.
But this non-determinism can be controlled by allowing variations only during the analysis, and not during the generation.

History of Computational Morphology
Generative Grammars: Ordered sequence of rules

$x -> y / z \_ w$

Looks like Context Sensitive Rule

Johnson (1972) showed that these rules are much less powerful than Context Sensitive rules.

Kaplan and Kay (1981) showed that phonological rewrite rules describe REGULAR RELATIONS, can be handled by Finite State Transducer.

Akshar Bharati(1994) used the Word and Paradigm model.

Finite State Automata/Transducers

Computational model to handle the morphological analysis/generation:

- used in morphology, phonology, TTS, Data mining.
- Popular, since mathematically extremely well understood.
- Easy to implement.
- Many off-the-shelf tools available for direct use.
- optimisation: finding a m/c with minimum number of states that performs the same function: Well worked out.
- simple extensions to Markov Models useful for POS tagging.

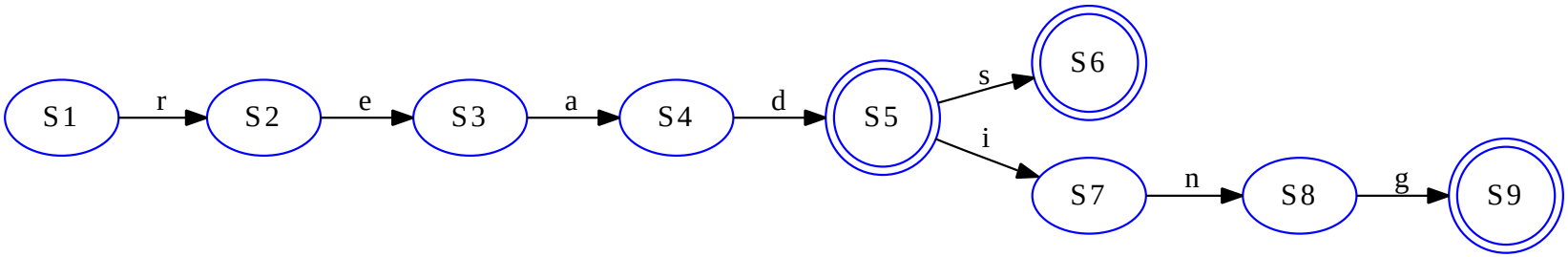
Finite State Automaton

A Finite State Automaton is a machine composed of

- An input tape
- A finite number of states, with one initial and one or more accepting states
- Actions in terms of transitions from one state to the other, depending on the current state and the input

The input tape has a sequence of symbols written on it. The tape, initially is in the intial state. The reader reads one symbol at a time, and depending on the current state and input symbol, transition to another state takes place.

A simple FSA that recognises various verb forms of 'read' viz. *read*, *reads*, and *reading* is shown below.

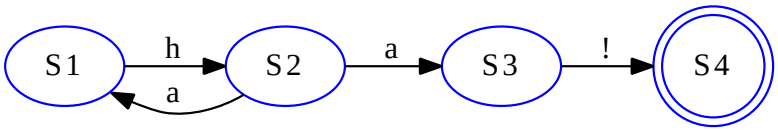


Recogniser and Generator

- The language accepted / recognised by an FSA =
  set of all strings it recognises when used in recognition mode.
- The language generated by an FSA =
  set of all strings it generates when used in generation mode.

The language accepted and the language generated by a FSA are exactly the same.

Deterministic Versus Non-Deterministic FSA



- Mathematically non-determinism does not add anything to FSA can do.
- We can always find a deterministic automaton that recognizes/generates exactly the same language as a non-deterministic automaton.
  (The determinization process may lead to explosion of number of states.)
- However, if we use this machine for recognition, a wrong choice of transition at non-detreministic state may lead to non-recognition.
- So a kind of search/backtracking is necessary to ensure covering all possible paths.
- Good news: In morphology, we rarely come across non-determinism.

Linguistic models for describing morphology

- Item and Arrangement (morpheme based)
- Item and Process (lexeme based)
- Word and Paradigm (word based)

Computational Models

Two level Morphology	Word and Paradigm Model
Requires good linguistic background	Not much Linguistic background needed Anybody with adequate language background can implement
The systems were not very efficient Relies on set compositions (expensive)	Several fast tools are available

Two level rule formalism which was dominant in early 90s are falling out of use, and are superseded by the sequential replace rules of XFST.

Several off-the-shelf tools availble for FST

which support Word and Paradigm model

1. XFST (Xerox Finite State Tool)
2. SFST (Stuttgart Finite State Transducer Tools)
3. HFST (Helsinki Finite State Toolkit)// (opensource rewrite of XFST)
4. Lttoolbox

Word and Paradigm model

GodA, e.g., behaves like **ladakA** as is obvious from the following tables.

case/num	singular	plural	vocative
direct	ladak**A**	ladak**e**	ladak**e**
oblique	ladak**e**	ladak**oM**	lada**ko**

case/num	singular	plural	vocative
direct	God**A**	God**e**	God**e**
oblique	God**e**	God**oM**	God**o**

Lttoolbox is Part of a bigger system – Apertium.

Apertium is a

Free/open-source platform for developing rule-based MT platform that provides

- language-independent machine translation engine
- tools to manage the linguistic data necessary to build a machine translation system for a given language pair and
- linguistic data for a growing number of language pairs.

Developers: Transducens research group at the Departament de Llenguatges i Sistemes Informtics of the Universitat d'Alacant in collaboration with Prompsit Language Engineering.

URL: http://wiki.apertium.org

Lttoolbox contd..

- Apertium Community supported various special requirements of our group such as support for WX notation, etc.
- Reasonably fast (thousands of words per second),
- handles huge dictionaries (size in Million).
- Follows XML standard, so it is easy to convert the data to any other format.

Lttoolbox:
A toolbox for lexical processing, morphological analysis and generation of words.
The package is split into three programs,

- lt-comp, the compiler,
  This builds a FST from the XML description
- lt-proc, the processor,
  This generates / analyses the string, and
- lt-expand, generates all possible mappings between surface forms and stems in the dictionary.

The XML specification for the mappings of surface forms with their stem consists of 3 sections.

- alphabet: This lists all the letters in an alphabet of a language. (No whitespace, since it may cause problems for tokenisation)
- definitions: This contains a list of all grammatical symbols we will be using to describe the morphology.
- main section: This is the main stuff containing various word forms with their analysis and the lexicon.

An example:

Let us look at various forms of the word ladakā in Hindi. They are:

case/num	singular	plural	vocative
direct	ladak**A**	ladak**e**	ladak**e**
oblique	ladak**e**	ladak**oM**	lada**ko**

Stem : ladak

Representation as per Lttoolbox specifications:

```
<dictionary>

<alphabet>
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
</alphabet>

<sdefs>
<sdef n="n" c="cat=noun" />
<sdef n="sg" c="number=singular" />
<sdef n="pl" c="number=plural" />
<sdef n="dir" c="case=direct" />
<sdef n="obl" c="case=oblique" />
<sdef n="voc" c="case=vocative" />
</sdefs>
```

```
<pardefs>
<pardef n="ladak/A__n">
    <e> <p>
      <l>A</l>
      <r>A<s n="n" /><s n="sg" /><s n="dir" /></r>
    </p> </e>
    <e> <p>
      <l>e</l>
      <r>A<s n="n" /><s n="sg" /><s n="obl" /></r>
    </p> </e>
    <e> <p>
      <l>e</l>
      <r>A<s n="n" /><s n="sg" /><s n="voc" /></r>
    </p> </e>
```

```
<e> <p>
  <l>e</l>
  <r>A<s n="n" /><s n="pl" /><s n="dir" /></r>
</p> </e>
<e> <p>
  <l>oM</l>
  <r>A<s n="n" /><s n="pl" /><s n="obl" /></r>
</p> </e>
<e> <p>
  <l>o</l>
  <r>A<s n="n" /><s n="pl" /><s n="voc" /></r>
</p> </e>
</pardef>
</pardefs>
</section>
```

<section id="main" type="standard">
<e lm="ladakA">ladak<par n="ladak/A__n" /></e>
</section>
</dictionary>

Compiling the data:

This data can be compiled for recognition as well as generation.

The data in XML file has the surface form within <l> tag (i.e. left side) and its analysis within <r>tag (i.e. right side).

ladakoM = ladakA, noun, plural, oblique

So to compile the analyser, lt-comp is provided an argument 'lr', and to compile the generator, we provide an argument 'rl'.

Thus the command
lt-comp **lr** <morf_file> <dict_file>
(e.g. lt-comp **lr** hindi_morf.bin ladakA.dix)
will produce an **analyser** for Hindi.

lt-comp **lr** <gen_file> <dict_file>
(e.g. lt-comp **rl** hindi_gen.bin ladakA.dix)
will produce a **generator** for Hindi.

Analysis:
Command: lt-proc -c hindi_ana.bin

input:surface form – >
ladake

—> Output:analysis
^ladakA<noun><singluar><oblique>/
ladakA<noun><plural><direct>/
ladakA<noun><singular><vocative>$

generation:
lt-proc -c hindi_gen.bin

input : analysis – >
ladakA<noun><singular><oblique>

– > surface form:output
ladakA

lt-expand ladakA.dix

This command produces all the surface form resulting from the given dix file.

ladakA:ladakA<noun><singular><direct>
ladake:ladakA<noun><singular><oblique>
ladake:ladakA<noun><singular><vocative>
ladake:ladakA<noun><plural><direct>
ladakoM:ladakA<noun><plural><oblique>
ladako:ladakA<noun><plural><vocative>

Exercise 1

- Choose a typical masculine noun, ending in 'A', from your language.
- Call it a head word.
- Write down its various forms along with various features and their values associated with them.
- Note down all the features and their values.
- This forms your sdef section.
- Now identify the stem for the given word.
- Mark the number of characters to be deleted from the word form.
- and the number of characters to be added so as to get the head word.
- Develop the pardef section using this information.
- Add the entry of this word in the dictionary section.

- Follow the instructions given earlier to
- compile, analyse, generate and expand

Paradigm

We follow the Word and Paradigm model to capture the generalisations in a language.

GodA, e.g., behaves like **ladakA** as is obvious from the following tables.

case/num	singular	plural	vocative
direct	ladak**A**	ladak**e**	ladak**e**
oblique	ladak**e**	ladak**oM**	lada**ko**

case/num	singular	plural	vocative
direct	God**A**	God**e**	God**e**
oblique	God**e**	God**oM**	God**o**

Therefore, we do not enter all the surface forms for **GodA**.
But instead, we **declare** that **GodA** and **ladakA** behave alike.
The declaration in lt-toolbox go in the main section.

```
<section id="main" type="standard">
<e lm="ladakA"><i>ladak</i><par n="ladak/A__n" /></e>
<e lm="GodA"><i>God</i><par n="ladak/A__n" /></e>
</section>
```

rAjA, on the other hand, has a different paradigm, as is clear from the following table:

case/num	singular	plural	vocative
direct	rAjA	rAjA	rAjA
oblique	rAjA	rAjA**oM**	rAjA**o**

So we introduce a new paradigm **rAjA**, and the corresponding entry in the dictionary as follows:

```
<pardef n="rAjA/__n">
    <e> <p>
      <l></l>
      <r><s n="noun" /><s n="singular" /><s n="direct" /></r>
    </p> </e>
    <e> <p>
      <l></l>
      <r><s n="noun" /><s n="singular" /><s n="oblique" /></r>
    </p> </e>
    <e> <p>
      <l></l>
      <r><s n="noun" /><s n="singular" /><s n="vocative" /></r>
    </p> </e>
```

```
    <e> <p>
      <l></l>
      <r><s n="noun" /><s n="plural" /><s n="direct" /></r>
    </p> </e>
    <e> <p>
      <l>oM</l>
      <r><s n="noun" /><s n="plural" /><s n="oblique" /></r>
    </p> </e>
    <e> <p>
      <l>o</l>
      <r><s n="noun" /><s n="plural" /><s n="vocative" /></r>
    </p> </e>
    </pardef>
<section id="main" type="standard">
<e lm="rAjA"><i>rAjA</i><par n="rAjA/__n" /></e>
</section>
```

Exercise 2

For 'A' ending masculine words in your language,

- Enhance the dictionary by adding more token head words corresponding to the types.
- Compile and test.

Exercise 3

For 'A' ending masculine words in your language

- Add the exceptional paradigms as well.
- Enhance the dictionary by adding more token head words corresponding to the types.
- Compile and test.
- Extend this for other type nouns.

Verb Morphology

Verb Root: jA
gender = masc
tense = future

person	singular	plural
first	jA**UMgA**	jA**yeMge**
second(tU/tuma)	jA**egA**	jA**oge**
second(tuma/Apa)	jA**oge**	jA**eMge**
third	jA**yegA**	jA**yeMge**

Verb Root: jA
gender = fem
tense = future

person	singular	plural
first	jA**UMgI**	jA**yeMge**
second(tU/tuma)	jA**egI**	jA**ogI**
second(tuma/Apa)	jA**ogI**	jA**eMgI**
third	jA**yegI**	jA**yeMgI**

Relevant part of the sdef section

```
<sdefs>
<sdef n="verb" />
<sdef n="singular" />
<sdef n="plural" />
<sdef n="first" />
<sdef n="second" />
<sdef n="second_h" />
<sdef n="third" />
<sdef n="masc" />
<sdef n="future" />
</sdefs>
```

Sample entries in the pardef section

```
<pardef n="jA/__v">
<e><p>
<l>UMgA</l>
<r><s n="verb"/><s n="singular"/><s n="first"/>
<s n="masc"/><s n="future"/></r>
</p></e>
<e><p>
<l>eMge</l>
<r><s n="verb"/><s n="plural"/><s n="first"/>
<s n="masc"/><s n="future"/></r>
</p></e>
```

Sample entries in the dict section

```
<section id="main" type="standard">
<e lm="jA"><i>jA</i><par n="jA/__v" /></e>
<e lm="KA"><i>KA</i><par n="jA/__v" /></e>
<e lm="gA"><i>gA</i><par n="jA/__v" /></e>
</section>
```

Exercise 4

For a selected verb, and a chosen tense

- write down all the verb forms.
- enumerate all the associated features and their values.
- Prepare the sdef, pardef and main section as explained above.
- Compile and test.

Exercise 5

Repeat the above for one more tense/mood, preferably with different feature set than the earlier one.

Exercise 6

Combine the noun and verb dix files into one.
Compile and test.

Derivational Morphology

Derivation:

- A process to form new words, which further undergoes inflection.
- May change the syntactic category or may not.
- Changes the meaning

It involves two steps

- Creating new stems from the given stems
- Linking these stems with the inflectional paradigms

For example:

Telugu: (1) chinna (young) -> chinnadi (younger one)

adi (pron, 3p, sg, non-masc) - > dAniki (to that)
Similarly
chinnadi (younger one) - > chinnadAniki (to the younger one)

(2) Linking chinnadi with adi

Hindi: (1) dUdha (milk) -> dUdhavAlA (milkman)

ladakA (boy) - > ladakoM (boy, plural, obl)
Similarly
dUdhavAlA (milkman) - > dUdhavAloM (milkman, plural, obl)

(2) Linking dUdhavAlA with ladakA

Creating new stems: 1^{st} example

<pardef n="dUdha/__dn">
<e>
<l>vAlA</l>
<r><s n="noun" /><s n="vAlA" /></r>
</e>

Linking with the existing paradigm

<e> <l></l><r></r>
vAl<par n="ladak/A__n" /></e>
</pardef>

Creating new stems: 2^{nd} example

<pardef n="pag/ulu__dv">
<e>
<l>alagoVttu</l>
<r><s n="verb"/>< *s* n="koVttu"/></r>
</e>

Linking with the existing paradigm

<e> <l></l><r></r>
alagoVtt<par n="koVtt/u__v"/></e>
</pardef>

Exercise 7

Identify the Suffixes related to Derivational Morphology in your language.
For one such suffix, declare the pardef and test the morphological analyser.

Handling Variants

The variants may occur at two levels: at thw word form level or at the stem level.

Stem level variant

This is indicated in the lexicon/dictionary.

For example, in Hindi, consider the two spellings of the word ladakA viz.

ladakA and laRakA, say

Then, the following lines declare their variant paradigm.

```
<pardef n="d_R_var">
<e><p><l>d</l><r>d</r></p></e>
<e><p><l>R</l><r>d</r></p></e>
</pardef>
```

And the following line in the main section takes care of the variation.

```
<e lm="ladakA"><i>la</i><par n="d_R_var"/><i>ak</i><par
n="ladak/A__n"/></e>
```

Word form level variant

If the variation is at the level of word form, and not the stem,
in Lttoolbox, there is a provision to specify it as an entry which is to be considered only when analysing (LR) and not when generating(RL).

For example, consider the two spellings of the suffix taruvAta in telugu viz. taruvAta and tarvAta.

But out of these only taruvAta is a standard form and tarvAta is dialectical variant.
That is, this dialectical variant should not be generated.

This is specified as follows:

```
<pardef n="pilla/__n">
<e> <p>
<l>lataruvAta</l>
<r><s n="noun"/><s n="singular"/><s n="taruvAta"/></r>
</p> </e>
<e r="LR"> <p>
<l>latarvAta</l>
<r><s n="noun"/><s n="singular"/><s n="taruvAta"/></r>
</p> </e>
</pardef>
```

Note the use of LR in the declaration of variant.

Exercise 8

Modify any of the earlier dix files so as to handle variants in your language.

Handling Vowel Harmony

Vowel Harmony: Assimilation of Vowels that are not adjacent to each other.

Examples:
maniSi -> manuSulu
samiti -> samutulu

So, from the point of view of add-del pair, these two words do not belong to the same paradigm.

In case of maniSi,
the string iSi is deleted, and
the string uSulu is added,
while

In case of samiti,
the string iti is deleted, and
the string utulu is added,

However, if we treat ‘t’ in ‘samiti’, and ‘S’ in ‘maniSi’ as variables, then we notice that, both these words behave in a similar manner, or may be considered to be belonging to the same paradigm.

iti – > utulu
iSi – > uSulu

Replacing them by variables, we have,

i[X]i – > u[X]ulu

The Metaparadigms in Lttoolbox may be deployed to handle the Vowel Harmony.

Metaparadigm:

The normal pardef paradigms handle only inflectional regularity.
The concept of Metaparadigms facilitate handling the root variations as well as the inflectional regularity.

Let ‘samiti’ be our regular type paradigm, and ‘maniSi’ be the token that belongs to the ‘samiti’ type.

We define the Metapardef as follows:

```
<pardef n="sam/i[t]i__n">
<e><p>
<l> u <prm/>ulu</l>
<r>i<prm/>i<s n="noun"/><s n="plural"/></r>
</p></e>
</pardef>
```

The corresponding dictionary section is:

```
<section id="main" type="standard">
<e lm="samiti"><i>sam</i><par n="sam/i[t]i__n" prm="t"/></e>
<e lm="maniSi"><i>man</i><par n="sam/i[t]i__n" prm="S"/></e>
</section>
```

Note the use of 'prm'. This takes care of the variable component.

The Metaparadigm file is first compiled with XSLT style sheet, which expands the paradigms, and produces a regular dix file.

This dix file is then compiled further for analysis and generation.

Exercise 9

Does your language have Vowel Harmony phenomenon?

If yes, write down a small dix file to handle a pair of such words.

Handling Prefixes

Some languages like Sanskrit, in addition to having the suffixes, also have prefixes.

For example, the third person singular past tense form of the root 'gam' is 'agacchat'.

Lttoolbox has a special provision to handle the prefixes.
Here is the relevant declaration for gam -> agacchat.

```
<pardef n="ga/m__v">
<e> <p>
<l>cchat</l>
<r>m<s n="verb"/><s n="past"/><s n="third"/>
<s n="singular"/></r>
</p> </e>
</pardef>
<pardef n="a__prefix">
<e lm="a" r="LR"><p> <l>a</l> <r></r> </p></e>
<e lm=""> <p> <l></l> <r></r> </p> </e>
CO </pardef>
<section id="main" type="standard">
<e lm="gam"><par n="a__prefix"/><i>ga</i>
<par n="ga/m__v"/></e>
</section>
```

Exercise 10

If your language has prefixes, handle them.

Handling words outside the Lexicon

Many a times we need an analyser which can recognise words that are not part of the lexicon. This is very much needed when we deal with agglutinative languages.
Here is the Lttoolbox Treatment for them.
We make use of the regular expressions to specify the words.

<pardef n="unkn__n">
<e><re> [a − zA − Z]+ </re>

<l>ki</l>
<r><s n="noun" /><s n="singular" /><s n="ki" /></r>
</e>
</pardef>

<section id="main" type="standard">
<e lm=""><par n="unkn__n" /></e>
</section>

Recap:

FST: An efficient way to handle morphology

Lttoolbox: An implementation of FST that handles

- Word and Paradigm model
- inflectional Morphology
- Derivational Morphology
  (linking derived words to existing paradigms)
- Variants at word level
- Variants at stem level
- Vowel Harmony
- Prefixes
- words outside the lexicon