

Tamil Morphological Analyser API (Version 1.0)

The documentation of the Tamil morphological analyser contains the following sections.

- 1) [Files which are needed for analyser](#)
- 2) [The Problem Description](#)
- 3) [Design of analyser](#)
- 4) [Description of the files and their formats](#)
- 5) [Extending this API](#)
- 6) [Reporting bugs](#)
- 7) [Operating environment](#)
- 8) [Interfacing with other systems](#)
- 9) [Dependencies](#)

Files Needed:

Morphological analyser API contains the following java files.

- TamilMorph.java
- Rootdict.java
- Suffixdict.java
- word1.java
- Suffix.java
- Root.java
- SplitString.java
- SPlitstring1.java
- Sandhineew.java

and the following data files.

- rootwords.dbt
- suffixverb.dat
- statetable.dbt
- sandhi.dbt

Compile the test program with the following command at the prompt

```
javac TamilMorph.java
```

and test the API with the following command:

```
java TamilMorph.java
```

The Problem Description:

Words in Tamil and many other Indian languages have a strong postpositional inflectional component. As an example, for verbs, these inflections carry information on the gender, person and number of the subject. Further, modal and tense information for verbs is also collocated in the inflections. For nouns, inflections serve to mark the case (accusative, dative & etc) of the noun. The aims of the morphological analyser described in this API are twofold: (1) strip the word of its inflections and (2) decipher the syntactic information carried by these inflections. By itself aim (1) is useful in that it is necessary for spellchecker in an inflected language. Aim (2) is useful in further grammatical processing at the sentence level e.g. in parsing. A complication encountered (in Tamil at least) is that the last syllable of a word or its inflection is further altered when additional inflections are added. These changes, described as "sandhi", are thought to be present for increasing the euphony of the word. Regardless of the reason, these sandhi mutations pose a formidable problem for the implementation of an efficient morphological analyser.

Design of the analyser:

As described above, the aim of the analyser is to strip a word of its inflections. The implementation that we have adopted is based on an approach based on a finite automata state-table. The following describes the steps carried out in sequence:

BEGIN: The state of the machine is set to be 0.0

ROOT: Is the word in the dictionary (dictionary words are called root words)? If so, we are done, go to **FINISH:**

This step is implemented in the file root.java

MORPHEME: Find the longest (inflectional) morpheme at the end of the word. These morphemes are listed in the file statetable.dat. If no such morpheme is found, go to SANDHI:

If a morpheme is found, remove it from the end of the word. and then go to ROOT:

SANDHI: The sandhi replacement tables are in the file sandhidict.dat

The morphological analyser fully depends on the automata table in the data file statetable.dat. This table explains the various paths needed to cut the Tamil morphemes from the given word starting from right to left (we will see the structure of this file later). The analyser first checks whether the given word is a root or not. For this purpose it uses root.java. If the given word is a root, then the analyser will print the grammatical features of that word. Otherwise it will check whether there are morphemes in the given word. For this purpose analyser uses the state-table file. If there is a match in the state-table then the morphemes are cut down from the given word. Every time the word is checked to find whether any root is present in it or not after cutting the morpheme. If there is no match found in the state-table the analyser would check whether there is any sandhi present in the given word. For this purpose it uses the sandhidict.dat file.

Description of the files:

Format of the data files

statetable.dat

- The statetable for Tamil is based on the chart prepared by Dr.S.Rajendran, Tamil University, Thanjavur
- This table explains the paths needed to decompose the morphemes of a given word from right to left.

The format of each line of this file is as follows

stateno: morpheme,nextstate[-nextstate....]; [morpheme,nextstate];

An example is illustrated below.

0.0:avanY,1.1;avalY,1.1;avar,1.1;awu,1.1;avE,1.1;anYa,1.1;kE,2.1;wal,2.1;#wwal,2.1-End;rYal,2.1-End

The first entry in the line (stateno) is an arbitrary number (or word) designating the state that the automata must be in when it scans the line. This 'stateno' is separated

from the other entries by a colon. Any number of entries may follow the 'stateno'. A semicolon must terminate each entry (except for the last entry, where it is optional). An entry consists of two fields: (1) the morpheme and (2) the number of the next state that the automata will transit into if the morpheme in the entry is present at the end of the word. A comma separates these two fields.

Sample Morpheme entry-1
avanY,1.1;

Here *avanY* is the morpheme and "1.1" is the next automata state. However, if there is more than one state that the automata can transit into upon encountering a morpheme, these multiple states are separated by a hyphen.

Sample Morpheme entry-2
rYal,2.1-End;

Here "2.1" and "End" are both possible states of the automata after it strips the word of the morpheme *rYal*.

Note: COMMENTS MUST BE TAKEN CARE OF HERE

sdict.dat

This file is used for sandhi changes. When two words are conjoined- there may or may not be changes taking place in the shape of these words. These changes are generally reflected in Tamil through changes in the graphemes (letters) used. These changes are called sandhi.

Example for sandhi
maram + E → marawwE

Here *ww* is the sandhi letter added when the two words *maram* and *E* are added. So when we analyse the word *marawwE*, the case-marker *E* is separated first from the word using the automata table and then there is no match found for the remaining word *maraww* in the state-table due to sandhi character *ww*. This file used to remove these type of sandhi changes from the given word and the format of the file is as follows

+/- <replacementCharacters>,{Suffix{+postfix:[postfix..]};{Suffix{+postfix:[postfix..]};

An example entry will illustrate it better as in the following case.

-m, {ww{+E:+Al:+utanY:+inY:+wwinY:+il:};f{+kaY:}}

For example there is a word *maraww* with the sandhi characters *ww* which could not find a match in the state-table for a morpheme. Then in that case there is an entry as above. We have a sandhi character *ww*, which is to be replaced by 'm'.

+ → Removal of characters

If the replacement character is preceded by + sign then there is no need to replace that character. The character itself is present in the word. For example there is an entry +u, {v{E}} in this file which is used for the word *pacuvE*. This word is split into *pacuv+E*. Here E is the morpheme and v is the sandhi attached with the word *pacu*. So here it is not needed to add the character *u* with the word.

- → Addition of characters

If the character is preceded by - then that character should be added with the word. For example there is an entry

-m, {ww{+E:}} in this file which is used for the word *marawwE*. This word is split into *maraww+E*. Here E is the morpheme and *ww* is sandhi character which is to be removed from the word. So here we have to add the character *m* to *mara* after chopping down the sandhi character *ww* to get the root stem *maram*.

<replacementCharacters> → Characters either to be added or not based on the - or + sign

Replacement characters, suffix and prefixes are separated by a comma. Curly braces enclose suffixes and prefixes.

Suffix:

These characters exist with the word in the form of sandhi. To get the stem these sandhi characters are to be removed from the word. These characters must be unique. There are multiple suffixes for a single replacement character(s). In such case suffixes are separated by semicolon.

Postfix:

These characters are morphemes, which are cut down from the given word. These postfixes are preceded by + sign. There is no meaning for this + sign. There is a possibility for multiple postfixes for a single suffix. In such case postfixes are separated by colon. These postfix entries are separated from the word using the state-table.

suffixverb.dat

The format of this file is as below.

morpheme, category, name of the suffix

Sample entry,

AnY, Verb, 3MSg

This file is used to retrieve the suffix information. Here morpheme indicates the entry, which is cut down from the word using state-table. Category for which the morpheme is separated is indicated in the category field. For example with the help of state-table we can cut down the suffix *AnY* from the word *patiwvAnY*. For this *AnY* we can get the grammatical information as third person singular masculine with the above entry from this file.

kdict.dat

This file is used to retrieve the root and its category for the given word (for both noun and verb). The format of this file is as follows.

Root word,paradigm,verbtype(strong,middle,weak),category of the root word

The paradigm and the verb type are kept empty in this file. They are kept for future purposes. They are useful for generating the Tamil words.

Note:

This dictionary is obtained from two different sources. The notation used to indicate a particular grammatical category differs for these two sources. For example, the grammatical category of *noun* was denoted as 'pev.' in one source and as 'n' in another source. The analyser shows both these information in its output as a distinct entry.

The java files are explained as below.

TamilMorph.java

This is the main class which calls all other classes and their methods. This class contains a constructor `TamilMorph()` and it is used to load the root-dictionary, suffix-dictionary and the state-table

The methods and their usage are as follows.

```
public Word1 getAnalysis(String WORD)
```

WORD – which is to be analysed, and
Word1 – object contains two data fields

static vector root – To store all the information about root

static vector suffix – To store all the information about suffixes and a method

public string toString() – To display the root and suffix information by overriding the *println* method.

The above method analyses the given word by using the *getRoot*, *getSuffix* and *handleSandhi* methods.

public Vector getEvents(String STATE)

STATE – is the state from which the possible match is found for morpheme in a word. It returns the morpheme and possible next states for that morpheme through a vector.

The above method is used to derive the events from the state-table to split the morphemes from the given word.

public boolean getRoot(String MORPHEME)

MORPHEME – is the string passed to this method to find whether it exists in the root dictionary.

The above method is used check whether the given word is found in the root dictionary through the *isRoot* method found in the file Rootdict.java.

public boolean getSuffix(String SUFFIX)

SUFFIX – is the string passed to this method to find whether it exists in the suffix dictionary or not.

The above method is used to check whether the suffix is found in the suffix dictionary through the *isSuffix* method found in Suffixdict.java file

public boolean loadSTable(String FILENAME)

FILENAME – is the automata filename that is passed to this method to load it.

The above method is used to load the state-table through the constructor TamilMorph.

RootDict.java

This file contains the following methods.

public Vector getCategory(String ROOT, String CAT)

ROOT – is the root of the given word, and

CAT – is the category of the root word

The above method is used to get the grammatical information of the root stem by passing it along with the major part of speech unit (refers to noun or verb).

public boolean isRoot(String ROOT)

ROOT – is the string passed to this method to find out whether it is root or not.

The above method is used to check whether the morpheme is a root stem or not by passing it.

```
public boolean loadRootdict(String DICTNAME)
```

DICTNAME – is the name of the root dictionary passed to this method to load it.

The above method is used to load the root dictionary through the Tamilmorph constructor in the file Tamilmorph.java.

SuffixDict.java

This file contains the following methods

```
public boolean loadSuffixdict(String DICTNAME)
```

DICTNAME – is the name of the suffix dictionary passed to this method to load it.

This method is used to load the suffix dictionary through the Tamilmorph constructor in the file Tamilmorph.java.

```
public Vector getCategory(String SUFFIX, String CAT)
```

SUFFIX – is the morpheme, which is chopped from the given word using automata table.
CAT – is the category of the suffix.

The above method is used to get the grammatical information of the suffix by passing it along with the major part of speech unit (refers to noun or verb).

Word1.java

This class is used for printing the morphemes and root along with their features by passing the object of this class to *println* method.

Suffix.java

This class deals with morphemes. It contains a constructor, which is used to store the morpheme category and its information apart from a method that is used for printing purposes by overriding the *println* method.

Root.java

This class is used to store the root stem of the given word along with its category.

SplitString.java

This class is used to split the word based on a particular delimiter – the string and the delimiter are passed as arguments to a method *split*.

SandhiNew.java

This class is used to cut the sandhi characters from the given word.

Extending this API:

This API mainly depends on the two data files `sdict.dat` and `statetable.dat`. So it is possible to extend the coverage of this api by extending the rules in this datafiles. The software is independent of these data files. So once we add rules in the proper format we can extend the coverage of this API.

Interfacing with other systems:

This API has been designed based on *object oriented methodology*, it is easier to interface this API with other systems. Interfacing is achieved by creating objects to *TamilMorph* class and *Word* class. The *getAnalysis* method of *TamilMorph* class will do all the things and return a 'word' object, which contain the root and morphemes of a given word with its grammatical features.

Operating Environment:

This API has been tested under Windows NT with jdk1.3 for the coverage listed in the coverage section.

Reporting Bugs:

Currently this analyser will show some duplicated information for a single word. For example for a word 'pati', it will show both 'peV.' as well as 'n' which are same for that word. This is because the root dictionary currently contains both the information. If the analyser fails to analyse a word it will show information as shown below.

word<UNK>

It means that some data are needed for this analyser either in the form of sandhi or in the state-table or in the root dictionary.

Mail to visu@au-kbc.org to report bugs.

Dependencies:

This API depends on the delimiters between the fields in the data files and the structure of the data in it.

Document prepared by: [S Ramesh Kumar](#), [S Viswanathan](#)