FLAN User Manual 0.55x

CONTENTS

FLAN User Manual 0.55x

Objective: *Providing guidelines for entering language data for FLAN*

Intended Users: *Linguists*

1. Introduction

FLAN is a generic shell that can be used for morphological analysis. It is based on augmented FSTs (Finite State Transducers). This document gives guidelines for entering language data to be used by the shell for morphological analysis. Certain morphological processes like assimilation, epenthesis, apocope, vowel harmony which involve operations like addition, deletion, reversal are also discussed. Sample entries have been provided for them.

See LLSTI report on FLAN, May 2004, for more information on the formalism of the shell and specifications for providing data.

2. Formalizing Morphological Changes

2.1. Methods to represent morphological changes

There are three ways to represent data for inflectional morphology – (a) using paradigm tables, (b) using transducers/transition tables or (c) using both. For languages with simpler morphology [eg. Hindi] paradigm tables may be given and for languages with complex phenomenon that show consistency in the morphological changes[eg. Ibibio], rules may be given using variables in the transducers. For agglutinative languages like Telugu, both the methods have been found useful for giving the data. **[See the illustration of these methods in Appendix 1]**

2.2. Main points to be noted

1. The shell is based on language data. So, one has to give the FSTs according to the needs of one's language. The input to a FST may be a string, dictionary, paradigm or any other FST. The FSTs are to be given as transition tables. If one is calling a dictionary, then the relevant dictionaries should be provided as well. And if one is calling a paradigm, then the data to the paradigm handler shell should be provided along with a FST calling paradigm handler shell i.e. with input as ‘pdgm’

2.One has to know what all is required for one’s language i.e. if just FSTs are sufficient or if there is a need to use paradigm tables to represent data. Accordingly, data has to be provided.

Let us look at the shell and the steps involved in giving data for morphological decomposition.

3. Entering Data in FLAN Version 0.55

3.1. Directory Structure

First, let us get familiar with the directory structure of FLAN. After untarring the archive FLANver_0.55.tgz, a directory FLANver_O.55 is created. This contains the following directories – DOC, PH, FST and testing and two files - phases.lst and install.sh

- *DOC* directory contains the readme file and other relevant manual.
- *Phases.lst* is a file in which the list of the pathnames of the first FST for each phase has to be given. The pathname is followed by the separator to be used, in double quotes after a TAB. A separator can be 'nil', 'space', '+' or anything else.
- *PH* is the directory pertaining to the Paradigm Handler Shell.
- *FST* is the directory having the data (containing the transition tables and related dictionaries) and the programs.
- *Install.sh* – this program has to be run after giving all the required data and before testing the input word i.e. the shell should be installed before testing the input word.
- *Testing* is the directory where the input to be tested is given as a file.

3.2. Giving data in steps

Once you know which method (discussed in section 2 above and Appendix 1) to follow to represent data for your language, then the steps given below may be followed in giving the data.

(i). Languages NOT using paradigms

For languages requiring FSTs calling FSTs, strings and dictionaries (ex. Ibibio)

a. phases.lst
   -give the path to the first FST
b. FST
   - FSTs
   - dict [root, affix and alphabet dictionaries]

(ii) Languages using just paradigms

For languages requiring FSTs calling just paradigms (ex: Hindi)

a. phases.lst
   -give the path of the FST calling paradigm
b. FST
   - FST with input being pdgm

c. PH
- Ca, Ce, Fe, map_file, pc_data, dict.final

(iii) For languages with FSTs requiring FSTs, strings, dictionaries and paradigms (ex: Telugu)

a. phases.lst
-give two paths
-first FST of phase1 i.e. FST calling the pdgm
-first FST of phase 2

b. FST
- FSTs including the one calling paradigm handler shell and dictionaries
- dict [root and affix dictionaries]

c. PH
- Ca, Ce, Fe, map_file, pc_data, dict.final

3.3. Overall Procedure

Step 1: *Doc* – read the "readme" file to use the shell and the manual for the format of giving data.

Step 2: *Phases list* - give the path

FLANver0.55/FST/data/FST1 [#!for eg. FST1 may call PH shell]
FLANver0.55/FST/data/FST2 [#!for eg. FST2 may call root and affix dictionaries]

Step 3: *PH* - if calling paradigm handler shell, give data for Ca, Ce, Fe, map_file, pc_data and dict.final

In the directory FLANver0.55/PH/anusAraka/hindi/morph/test, give data for the files Ca, Ce, Fe and map_file.

After compilation using the command "./make_pf.pl", go to pc_data directory –

FLANver0.55/PH/anusAraka/hindi/morph/pc_data

Here .pf files are created automatically. Fill in the corresponding data in the place of dash '-' . Then run the command "./make_p.pl" by which the .p files are created.

Now go to the directory - FLANver0.55/PH/anusAraka/hindi/dict and give data in the dict.final file.

[See Appendix 4 for giving data in paradigm tables]

Step 4: *FST* – time to give the transition tables

In the directory FLANver0.55/FST/data, give all the required transition tables and the dictionaries if using them, pertaining to all the phases.

[See Appendices 2 and 3 and the 'samples' in section 4.3. for giving transition tables]

Step 5: *Testing* - give the input to be tested.

Check if the path of the FSTs for each phase in the **phases.lst** is correct. Then install using the command "sh install.sh" at the FLANver0.55 directory level.

Follow the message and add the given lines in the **.bash_profile** file in your home directory.

Now, in the directory FLANver0.55/testing give the input file with the input word to be tested by using the command"./run.pl input_filename"

Hey, presto!! – you get the analysis of the given input word!!!

All the steps given above are exemplified using the shell for Telugu language as all the steps mentioned above are utilized for it. Other languages like Hindi, Russian and Ibibio have been used to exemplify some other aspects of language that can be handled by the shell.

4. Some Samples

4.1. Analyzing Telugu agglutinative verbs

Look at the following verbs in Telugu:

tinTunnADu = tinu + Tunna + A +Du
vinnADu = vinu + A + Du

We see that during the process of affixation, there are morphophonemic changes taking place. We will see how the shell will give the analysis.

For Telugu, we are using two phases – first one to identify the root and suffixes in a given word and in the second phase to give their features. We will see how the data has been given for Telugu.

Step 1: *Doc* – read the "readme" file to use the shell and the manual for the format of giving data. .

Step 2: *Phases list* - has two lines stating the phases; the separator is "space".

FLANver0.55/FST/data/FST1 " "
[#!FST1 calls PH shell for identification of suffixes]

FLANver0.55/FST/data/FST2 " "
[#!FST2 calls the root and affix dictionaries for extracting the features of the suffixes identified]

Step 3: *PH* – the data for Ca, Ce, Fe, map_file, pc_data and dict.final are given in the PH shell as follows:

In the directory FLANver0.55/PH/anusAraka/hindi/morph/test, the data for the files Ca, Ce, Fe and map_file are given.

Ca
verb1 verb
verb2 verb

Ce
verb1 suffa suffb
verb2 suff1 suff2 suff3 suff4 suff5 suff6

Fe
suffa tunnA tunTA tA A
suffb nu mu vu yi Du ru
suff1 a
suff2 goTTu veyyi
suff3 a
suff4 manu nivvu
suff5 a
suff6 ledu rAdu vaccu Ali

Map_file
suffa suffa
suffb suffb
suff1 suff1
suff2 suff2
suff3 suff3
suff4 suff4
suff5 suff5
suff6 suff6

To compile, the command "./make_pf.pl" is run. In the **pc_data** directory –

FLANver0.55/PH/anusAraka/hindi/morph/pc_data

.pf files are created automatically.

For example, verb1.pf

```
verb1
-       #ROOT
-       #{suffa=tunTA,suffb=yi,}
-       #{suffa=tunTA,suffb=vu,}
-       #{suffa=tunTA,suffb=Du,}
-       #{suffa=tunTA,suffb=mu,}
-       #{suffa=tunTA,suffb=ru,}
-       #{suffa=tunTA,suffb=nu,}
-       #{suffa=tunnA,suffb=yi,}
-       #{suffa=tunnA,suffb=vu,}
-       #{suffa=tunnA,suffb=Du,}
-       #{suffa=tunnA,suffb=mu,}
-       #{suffa=tunnA,suffb=ru,}
-       #{suffa=tunnA,suffb=nu,}
-       #{suffa=tA,suffb=yi,}
-       #{suffa=tA,suffb=vu,}
-       #{suffa=tA,suffb=Du,}
-       #{suffa=tA,suffb=mu,}
-       #{suffa=tA,suffb=ru,}
-       #{suffa=tA,suffb=nu,}
-       #{suffa=A,suffb=yi,}
-       #{suffa=A,suffb=vu,}
-       #{suffa=A,suffb=Du,}
-       #{suffa=A,suffb=mu,}
-       #{suffa=A,suffb=ru,}
-       #{suffa=A,suffb=nu,}
```

The corresponding data in the place of dash ‘-‘ is given as shown below:

```
verb1
tinu    #ROOT
tinTunTAyi      #{suffa=tunTA,suffb=yi,}
tinTunTAvu      #{suffa=tunTA,suffb=vu,}
tinTunTADu      #{suffa=tunTA,suffb=Du,}
tinTunTAmu      #{suffa=tunTA,suffb=mu,}
tinTunTAru      #{suffa=tunTA,suffb=ru,}
tinTunTAnu      #{suffa=tunTA,suffb=nu,}
tinTunnAyi      #{suffa=tunnA,suffb=yi,}
tinTunnAvu      #{suffa=tunnA,suffb=vu,}
tinTunnADu      #{suffa=tunnA,suffb=Du,}
tinTunnAmu      #{suffa=tunnA,suffb=mu,}
tinTunnAru      #{suffa=tunnA,suffb=ru,}
tinTunnAnu      #{suffa=tunnA,suffb=nu,}
tinTAyi         #{suffa=tA,suffb=yi,}
```

```
tinTAvu      #{suffa=tA,suffb=vu,}
tinTADu      #{suffa=tA,suffb=Du,}
tinTAmu      #{suffa=tA,suffb=mu,}
tinTAru      #{suffa=tA,suffb=ru,}
tinTAnu      #{suffa=tA,suffb=nu,}
tinnAyi      #{suffa=A,suffb=yi,}
tinnAvu      #{suffa=A,suffb=vu,}
tinnADu      #{suffa=A,suffb=Du,}
tinnAmu      #{suffa=A,suffb=mu,}
tinnAru      #{suffa=A,suffb=ru,}
tinnAnu      #{suffa=A,suffb=nu,}
```

Then the command "./make_p.pl" is run by which the .p files are created.
For example: verb1.p

```
verb1
tinu
tinTunTAyi
tinTunTAvu
tinTunTADu
tinTunTAmu
tinTunTAru
tinTunTAnu
tinTunnAyi
tinTunnAvu
tinTunnADu
tinTunnAmu
tinTunnAru
tinTunnAnu
tinTAyi
tinTAvu
tinTADu
tinTAmu
tinTAru
tinTAnu
tinnAyi
tinnAvu
tinnADu
tinnAmu
tinnAru
tinnAnu
```

Now, in the directory - FLANver0.55/PH/anusAraka/hindi/dict, the data in the dict.final file is given as shown:

dict.final

"tinu","1","tinu","verb1"
"vinu","1","tinu","verb1"

Step 4: *FST* - transition tables

In the directory FLANver0.55/FST/data, all the required transition tables pertaining to all the phases and the dictionaries are given as shown. In the phases.lst this is the path that has to be given.

FST1
[#!pertaining to phase 1]

```
S1
S1 S2 S3 S4
S4
S1    S2    pdgm:dict_final !root<*=!ALL>  <*=!ALL>    <>
S2    S3    @     FS.suffa      <>    <>
S3    S4    @     FS.suffb      <>    <>
```

The notation in the action field of S1-S2 arc '<*=!ALL>' gets the root and suffixes from the paradigm handler shell. The notation in its output field '!root<*=!ALL>' prints the root and its features. Similarly, in the output fields of the subsequent arcs the suffixes and its features, if any, are printed.

The root and the suffixes identified are used as input to the next phase to do the final analysis.

FST2
[#!pertaining to phase 2]

```
S1
S1 S2
S2
S1      S2      dict:root_dict          !root<*=!ALL>       <>              <>
S2      S2      dict:affix_dict         $_INP_<*=!ALL>      <*=!ALL>    <>
```

Dictionaries - root and affix dictionaries - are called to verify and give the features of the root and suffixes. The following dictionaries were used for Telugu verbs.

root dictionary

```
tinu    <root=tinu/cat=v/class=tinu>
vinu    <root=vinu/cat=v/class=tinu>
```

affix dictionary

a <opr1 = infinitive>
goTTu <opr1 = transitiviser1>
nivvu <opr1 = permit>
manu <opr1 = tell_someone>
a <opr2 = negation>
rAdu <modal = should_not>
ledu <modal = did_not>
tunnA <aspect = durative/ tense = non_past>
tunTA <aspect = durative-habitual/ tense = non_past>
A <aspect = perfective / tense = past>
tA <aspect = habitual / tense = non_past>
nu <number = sg / person = 1>
mu <number = pl / person = 1>
vu <number = sg / person = 2>
ru <number = pl / person = 2|3>
Du <gender = m / number = sg / person = 3>
di < gender = f / number = sg / person = 3>
yi <gender = f / number = pl / person = 3>

Step 5: *Testing*

The path of the FSTs for each phase in the **phases.lst** is checked. Then using the command “sh install.sh” at the FLANver0.55 directory level, the shell is installed.

As indicated, the given lines are added in the **.bash_profile** file in the home directory.

Now, in the directory FLANver0.55/testing the input file with the input word to be tested is tested by using the command”./run.pl input_filename”
Examples:
tinTunnADu = tinu + Tunna + A +Du
vinnADu = vinu + A + Du

Input: tinTunnADu
Output: tinTunnADu<root=tinu/cat=verb/type=tinu/ aspect=durative/tense=non_past/ gender=m/number=sg/person=3>

Input: vinTunTAnu
Output: vinTunTAnu<root=vinu/cat=verb/type=tinu/aspect=durative-habitual/ tense= non_past/number=sg/person=1>

4.2. Representing Stress in Paradigm Tables

Using paradigm tables, the shifting of the stress can also be handled as shown below for examples from English and Russian:

Set 1: English
Sample input: "acade'micians"

Phase 1:
INPUT: acade'micians
OUTPUT: a'cademy<root=a'cademy/cat=noun> ic<> ian<> s<n=pl>
Phase 2:
INPUT: a'cademy<root=a'cademy/cat=noun> ic<> ian<> s<n=pl>
OUTPUT: acade'mician<root=a'cademy/cat=noun/n=pl>

The relevant data given for stress handling for the above input word is as shown below:

root_dict:
a'cademy <root=a'cademy/cat=noun>

affix_dict:
ic <cat=adj>
ian <cat=noun>
s <n=pl>

paradigm class [dict.final]:
acade'mician acade'mician

paradigm table[pc-data]:
academy
academician
academicians

transition tables
FST: eng1
[using strings]

S1
S1 S2 S3 S4 S5
S4 S5

S1	S2	str:a'cademy	root=a'cademy/cat=noun	<>	<>
S2	S3	str:ic	<>	<>	<>
S3	S4	str:ian	<>	<>	<>
S4	S5	str:s	number=plural	<>	<>

The features of the strings are given in the action field.

FST: eng2
[using dictionaries]

S1
S1 S2 S3
S3
S1 S2 dict:root_dict !root<*=!ALL> <> <>
S2 S3 dict:affix_dict $_INP_<*=!ALL> <*=!ALL> <>

Set 2: Russian

(i) Sample input 1: " stolej' "
Phase 1:
INPUT: stolej'
OUTPUT: sto'l<s=1,n=sg> ej<c=nom,g=f>

Phase 2:
INPUT: sto'l<s=1,n=sg> ej<c=nom,g=f>
OUTPUT: stolej'<root=sto'l/cat=noun/s=1/g=f/c=nom/n=sg>

(ii) Sample input 2: " sadej' "
Phase 1:
INPUT: sadej'
OUTPUT: sa'd<> ej<>

Phase 2:
INPUT: sa'd<> ej<>
OUTPUT: sadej'<root=sa'd/cat=noun/s=2/g=f/c=nom/n=sg>

The relevant data given for stress handling for the above input word is as shown below:

root_dict:

sto'l <s=1,n=sg>
sto'l <s=1,n=pl>
sa'd <s=2,n=sg>
sa'd <s=2,n=pl>

affix_dict:
ej <c=nom,g=f>
ej <c~=nom,g=f>
ej <c=instr,g=f>

paradigm class:
sto'l sto'l, sa'd

paradigm table:
stolej' sto'l<s=1,n=sg> ej<c=nom,g=f>
sto'lej sto'l<s=1,n~=sg> ej<c~=nom,g=f>
sa'dej sa'd<s=2,n=sg> ej<c=instr,g=f>
sadej' sa'd<s=2,n=pl> ej<c=instr,g=f>

transition table:
FST:russ1
[using dictionaries]

S1
S1 S2 S3
S3

S1	S2	dict:russ_root_dict	!root<*=!ALL>	<>	<>
S2	S3	dict:russ_aff_dict	$_INP_<*=!ALL>	<*=!ALL>	<>

FST:russ2
[using strings]

S1
S1 S2 S3
S3

S1	S2	str:sto'l	root=sto'l/s=1	<>	<>
S1	S2	str:sa'd	root=sa'd/s=1	<>	<>
S2	S3	str:ej	n=sg/c=instr/g=f	<>	<>

4.3. Representing Linguistic Rules in FSTs

The following examples illustrate how the rules may be represented using FSTs for some of the morphophonological phenomenon. We have taken examples from Ibibio and English to illustrate them. Some of the processes like reduplication and vowel harmony are modeled better using general rules rather than using paradigm tables. [The notations that can be used in the action field are given in Appendix 2].

Examples using variables in FSTs:

Set 1: covers negation in Ibibio
Set 2: covers deletion, addition, substitution and reverse processes
Set 3: covers vowel harmony and reduplication

Variables should be defined at the end of the transition tables using TABs as shown. Operations on variables are performed in a similar way as on attributes. However, variables cannot be stated on left hand side of the actions. They cannot be modified in the action; they can only be accessed/read.

4.3.1. Set 1

Examples from Ibibio for CVC and CV roots[1].

Example 1: Negation [for CVC roots]

INPUT: deppe

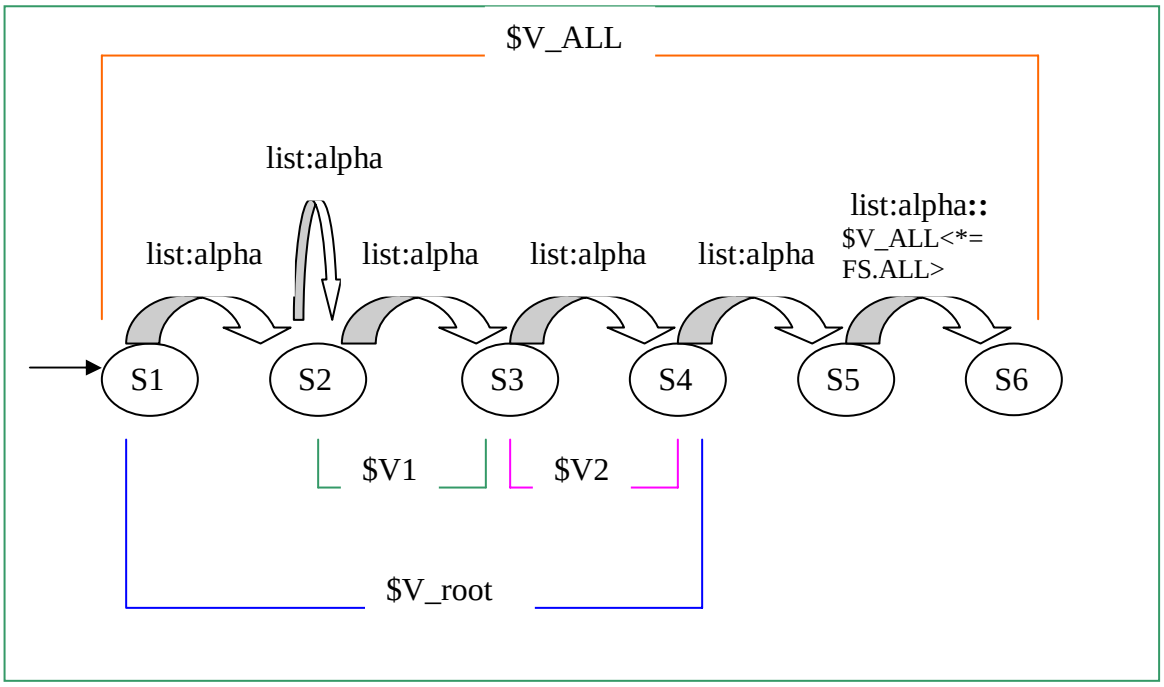
OUTPUT: deppe <root=dep / cat= v / fe=neg>

(where cat=category, v=verb, fe=feature, neg=negation)

General rule for suffix - the negative suffix is identical to the preceding consonant and vowel (from the stem) in case of CVC roots.

dep	dep+pe*(rising tone)* (not buying)
nam	nam+ma*(rising tone)* (not performing)
yet	yet+te*(rising tone)* (not washing)

The FSTs to describe this kind of word-formation using variables is shown below:



[1] The differentiation of different root types (CV, CVC, CVVC, CVCV and CVVCV) is required for most verb forms generation. There is a basic root structure, which is considered the ground for generating all other structures (see “Root Dictionary” in the “Example of a paradigm”).

The input label ‘list’ is calling a list of alphabets ‘alpha’. The input consumed while traversing the arc S2-S3 is stored in the variable $V1 and the input consumed while traversing the arc S3-S4 in the variable $V2. Similarly, the input consumed while traversing the arc S1-S4 is stored in the variable $V_root and the entire word form consumed while traversing the arc S1-S6 is stored in $V_ALL The condition on S4-S5 arc requires the input label to be the $V2 variable and the condition on S5-S6 arc requires the input label to be $V1. All the features associated with the arcs are given as output finally. The transition table is given below. The variables are defined at the end of the transition table using TAB.

S1
S1 S2 S3 S4 S5 S6
S6

S1	S2	list:alpha		<>	<>	
S2	S2	list:alpha		<>	<>	
S2	S3	list:alpha		<>	<>	
S3	S4	list:alpha		<root=$V_root/cat=v>	<>	
S4	S5	list:alpha		<>	<$_INP_=$V2>	
S5	S6	list:alpha	$V_ALL<*=FS.ALL>	<fe=neg>	<$_INP_=$V1>	
$V_root		S1	S4			
$V1	S2	S3				
$V2	S3	S4				
$V_ALL		S1	S6			

==========

Example 2: Negation [for CV roots]

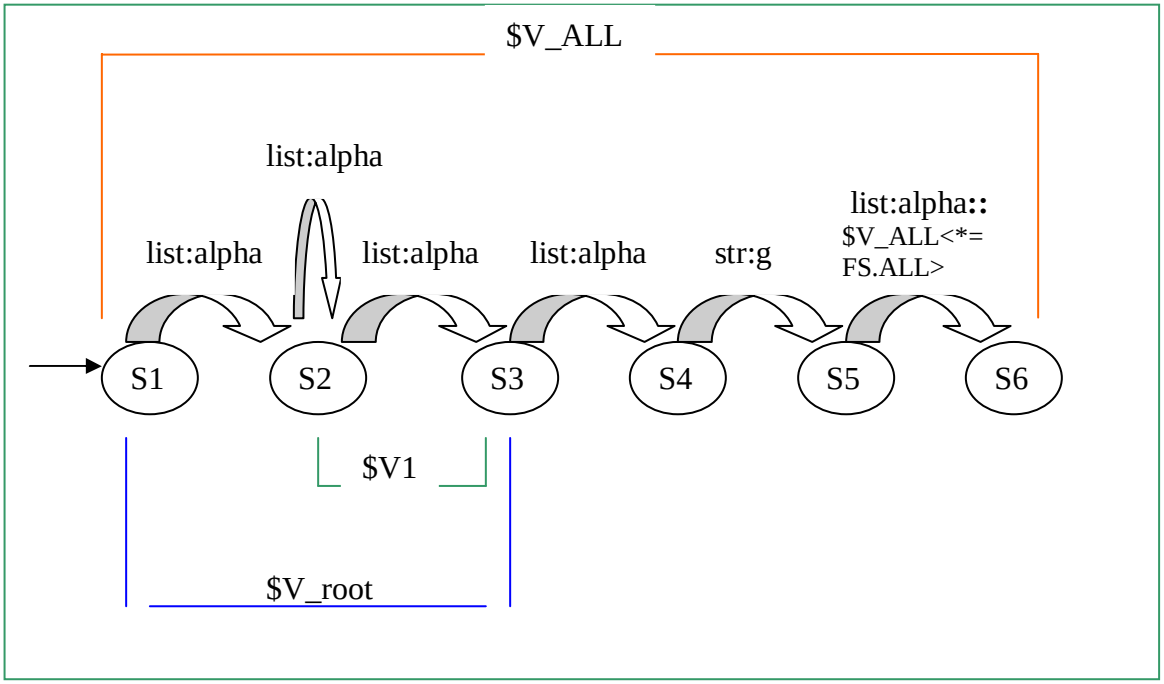
Stem alteration – CV root becomes CVV

se	see-ge*(rising tone)*
no	noo-go*(rising tone)*
da	daa-ga*(rising tone)*

General rule for suffix - the negative suffix takes the form of a consonant /g/ plus a preceding vowel.

INPUT: seege
OUTPUT: seege <root=se / cat= v / fe=neg>
(where cat=category, v=verb, fe=feature, neg=negation)

Using variables, the FST for this data would be:



The corresponding transition table would be:

S1
S1 S2 S3 S4 S5 S6
S6

S1	S2	list:alpha		<>	<>
S2	S2	list:alpha		<>	<>
S2	S3	list:alpha		<root=$V_root/cat=v>	<>
S3	S4	list:alpha		<>	<$_INP_=$V1>
S4	S5	str:g		<>	<>
S5	S6	list:alpha	$V_ALL<*=FS.ALL> <fe=neg>		<$_INP_=$V1>
$V_root		S1	S3		
$V1	S2	S3			
$V_ALL		S1	S6		

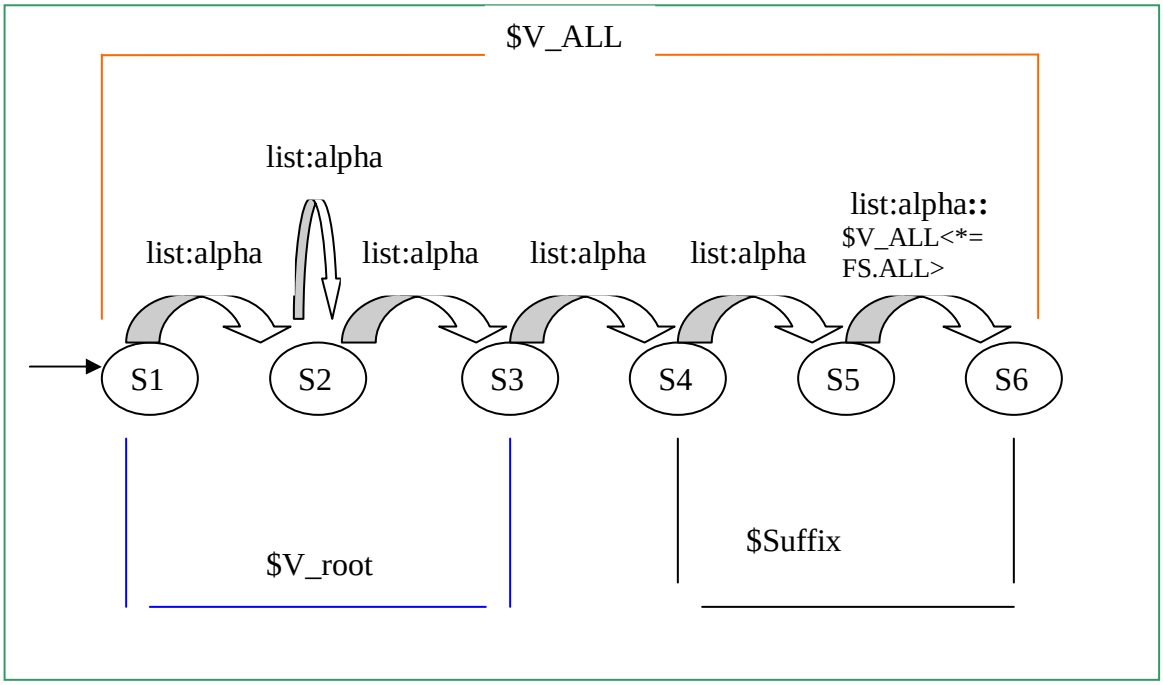
==========

4.3.2. Set 2

Example 3: Deletion

panicked = panic + ed [k is deleted]
referred = refer + ed [r is deleted]

INPUT: panicked
OUTPUT: panicked<root=panic/cat=v/tense=past>



The corresponding transition table would be:

S1
S1 S2 S3 S4 S5 S6
S6

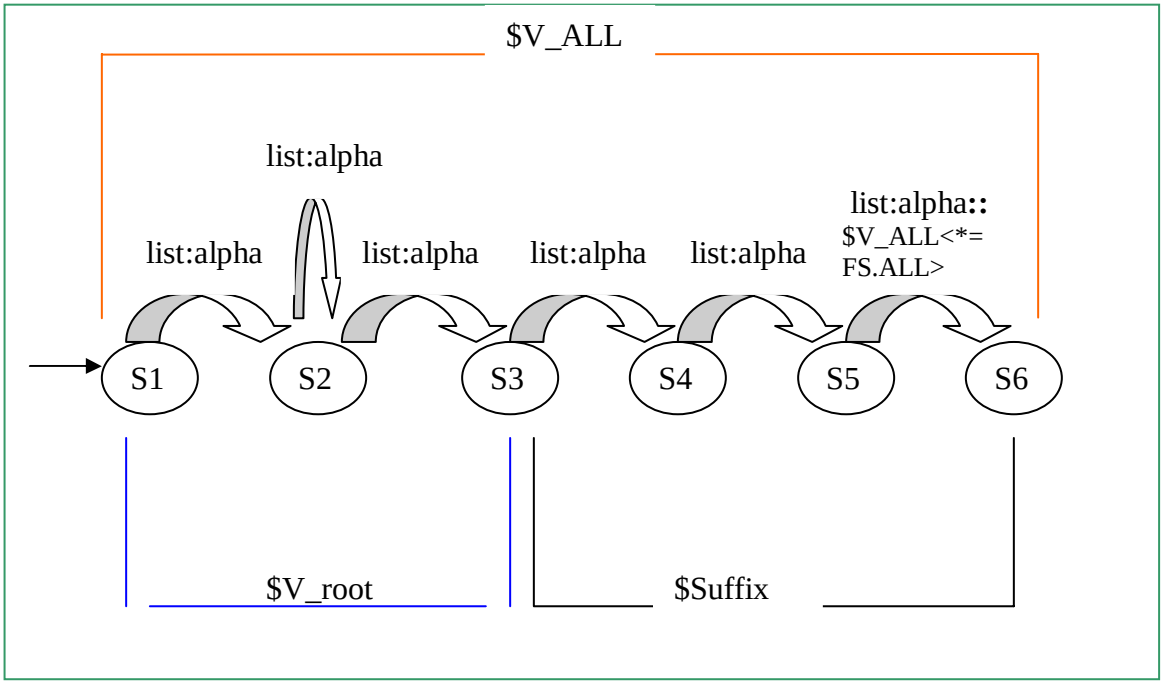
S1	S2	list:alpha		<>	<>
S2	S2	list:alpha		<>	<>
S2	S3	list:alpha		<root=$V_root/cat=v>	<>
S3	S4	list:alpha		<>	<>
S4	S5	list:alpha		<>	<>
S5	S6	list:alpha	$V_ALL<*=FS.ALL>	<tense=past>	<>
$V_root	S1	S3			
$Suffix	S4	S6			
$V_ALL	S1	S6			

==========

Example 4: Addition

taking = take + ing [e is added to root]
liking = like + ing [e is added to root]

INPUT: taking
OUTPUT: taking<root=take/cat=v/aspect=continuous>



The corresponding transition table would be:

S1
S1 S2 S3 S4 S5 S6
S6

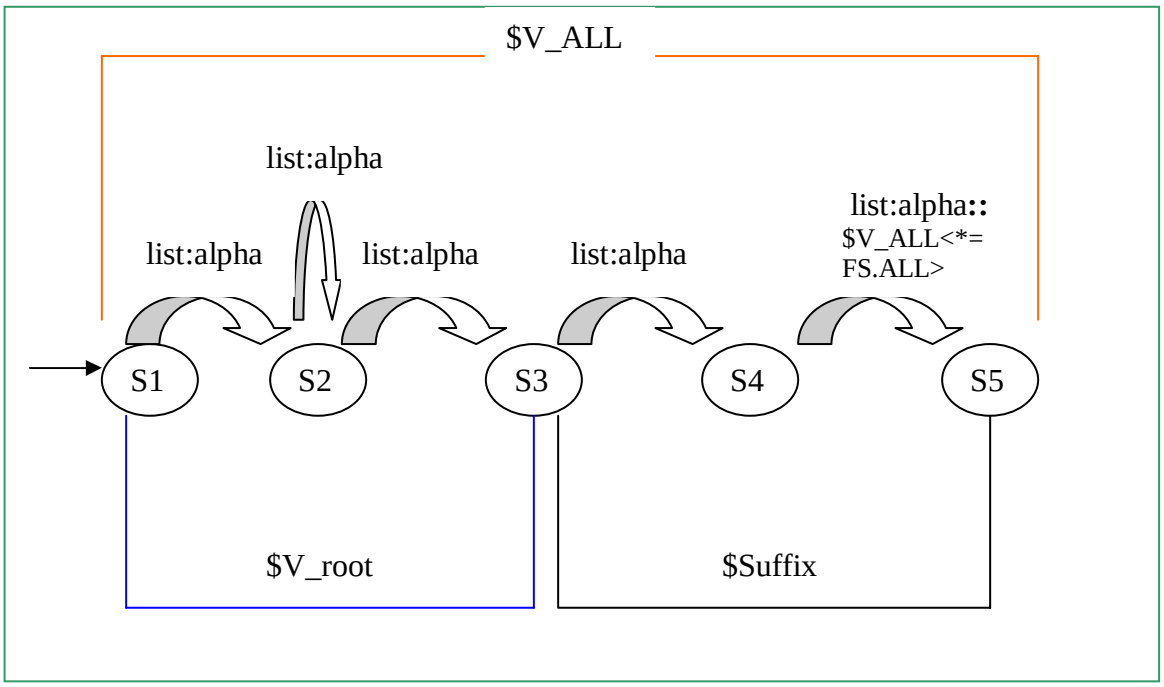
S1	S2	list:alpha	<>	<>
S2	S2	list:alpha	<>	<>
S2	S3	list:alpha	<root=add($V_root,e)/cat=v >	<>
S3	S4	list:alpha	<>	<>
S4	S5	list:alpha	<>	<>
S5	S6	list:alpha	$V_ALL<*=FS.ALL> <asp=cont>	<>
$V_root	S1	S3		
$Suffix	S4	S6		
$V_ALL	S1	S6		

==========

Example 5: Substitution

cried = cry + ed [y replaces i]
carried = carry + ed [y replaces i]

INPUT: cried
OUTPUT: cried<root=cry/cat=v/tense=past>



The corresponding transition table would be:

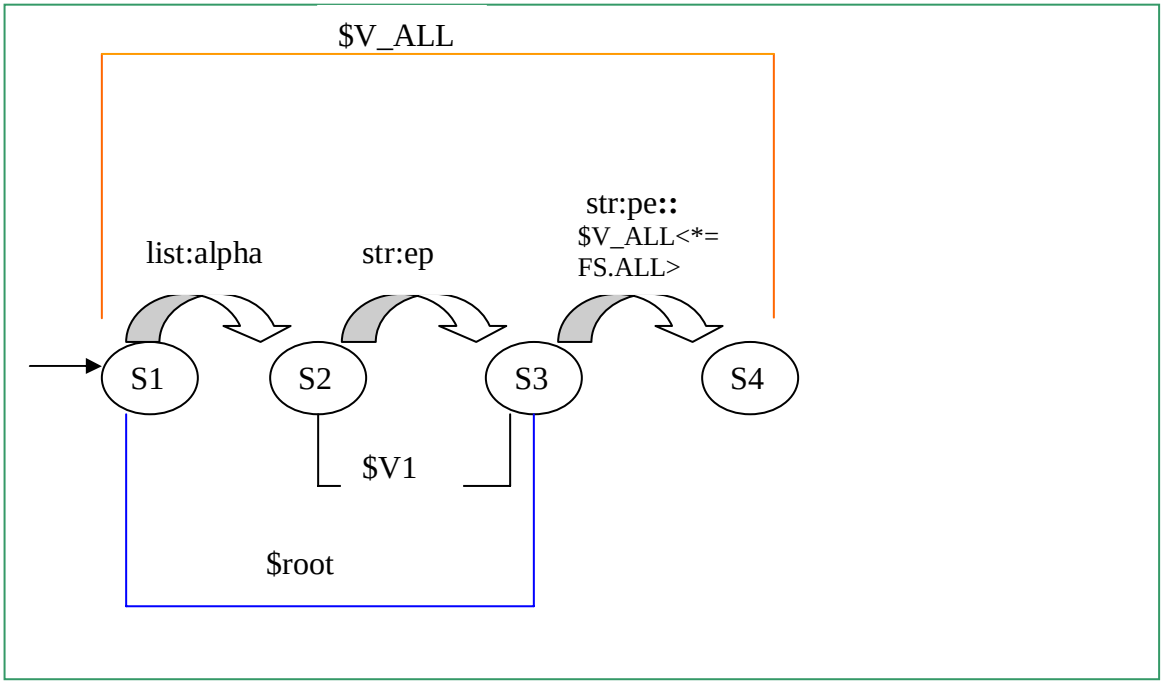
S1
S1 S2 S3 S4 S5
S5
S1 S2 list:alpha <> <>
S2 S2 list:alpha <> <>
S2 S3 list:alpha
<root'=delete($root,1,1)/root=add(FS.root',y)/cat=v> <>
S3 S4 list:alpha <> <>
S4 S5 list:alpha $V_ALL<*=FS.ALL> <tense=past> <>
$root S1 S3
$Suffix S3 S5
$V_ALL S1 S5

==========

Example 6: Reverse

deppe = d + ep + reverse of ep

INPUT: deppe
OUTPUT: deppe <root=dep/ cat=v/ fe=neg>



S1
S1 S2 S3 S4
S4

S1	S2	list:alpha	<>		<>
S2	S3	str: ep	<root=$root/cat=v /A=reverse($V1) >		<>
S3	S4	str: pe	$V_ALL<*=FS.ALL>	<fe=neg>	<$_INP_=FS.A>
$root	S1	S3			
$V1	S2	S3			
$V_ALL	S1	S4			

4.3.3. Set 3

Example 7: Vowel Harmony

(from Telugu)
manushulu = manishi + lu
pandulu = pandi + lu

INPUT: manushulu
OUTPUT: manushulu<root=manishi / cat=noun/ num=pl>

When the plural marker ‘lu’ gets affixed to the noun ‘manishi’, then except the first vowel, all other vowels change to ‘u’.

S1
S1 S2 S3 S4 S5

```
S5
S1      S2      list:alpha              <root=$_INP_>       <>
S2      S2      list:const              <root=add(FS.root,$_INP_)> <>
S2      S3      list:vowel              <root=add(FS.root,$_INP_>  <>
S3      S3      list:const              <root=add(FS.root,$_INP_)> <>
S3      S3      list:vowel              <root=add(FS.root,i)>       <$_INP_=u>
S3      S4      str:l                           <>      <>
S4      S5      str:u   $V_All<*=FS.ALL>  <>      <>
$V_All          S1      S5
```

List ‘alpha’ consists of all the alphabets, list ‘vowels’ all the vowels and list ‘const’ all the consonants. The first character is the root and at every state another character gets appended to it. The condition in S3-S3 arc states that if the input is ‘u’ then change it to ‘i’. So, when the input is ‘u’ in the S3-S3 arc, it changes to ‘i’. The first vowel remains the same, but the following ‘u’ vowels change to ‘i’.

Example 8: Reduplication

(from Ibibio)

d**o**l**a**ndolen = dolen + dolen
b**o**l**a**bali = bali + bali

The second form is the root and the first one is the reduplicated part. From this limited data, it is noticed that only the consonants get reduplicated and the vowels are fixed as ‘o’ and ‘a’.

INPUT: dolandolen
OUTPUT: dolandolen<root=dolen/ fe=reduplication>

```
S1      S2      list:alpha              <>      <>
S2      S3      str:o                   <>      <>
S3      S4      list:alpha              <>      <>
S4      S5      str:a                   <>      <>
S5      S6      list:alpha              <>      <>
S5      S6      @                       <>      <>
S6      S7      list:alpha              <>      <$_INP_=$v1>
S7      S8      list:alpha              <>      <>
S8      S9      list:alpha              <>      <$_INP_=$v2>
S9      S10     list:alpha              <>      <>
S10     S11     list:alpha      $V_ALL<*=FS.ALL>        < root=$root>
        <$_INP_=$v3>
S10     S11     @               $V_ALL<*=FS.ALL>        <root=$root>  <>
$root   S6      S11
$v1     S1      S2
$v2     S3      S4
```

APPENDIX 1

[Methods to give data]

We will look at some examples to see how the data can be represented using the three methods – (i) paradigm tables (ii) FSTs (iii) both

Data: English verb forms

Root	**Word forms**
eat	eat, eats, ate, eaten, eating
play	play, plays, played, played, playing
cry	cry, cries, cried, cried, crying

table 1

The data in table 1 above can be represented in the following ways.

First Method: In a paradigm table

The paradigm table is given with the root and its features for each of its word form as shown below:

Line 1	Root
Line 2	Present tense- 1st person sg|pl / 2nd person sg|pl / 3rd person pl
Line 3	Present tense- 3rd person sg
Line 4	Past tense
Line 5	Perfective Aspect
Line 6	Progressive Aspect

The actual data after compilation in the shell would be like this:

eat
eat
eats
ate
eaten
eating
play
play
plays
played
played
playing
cry
cry
cries
cried

cried
crying

The first line of each table is the root form. Each subsequent line is associated with a feature. This feature along with the root is extracted when a given form of any word in the paradigm class is given for analysis.

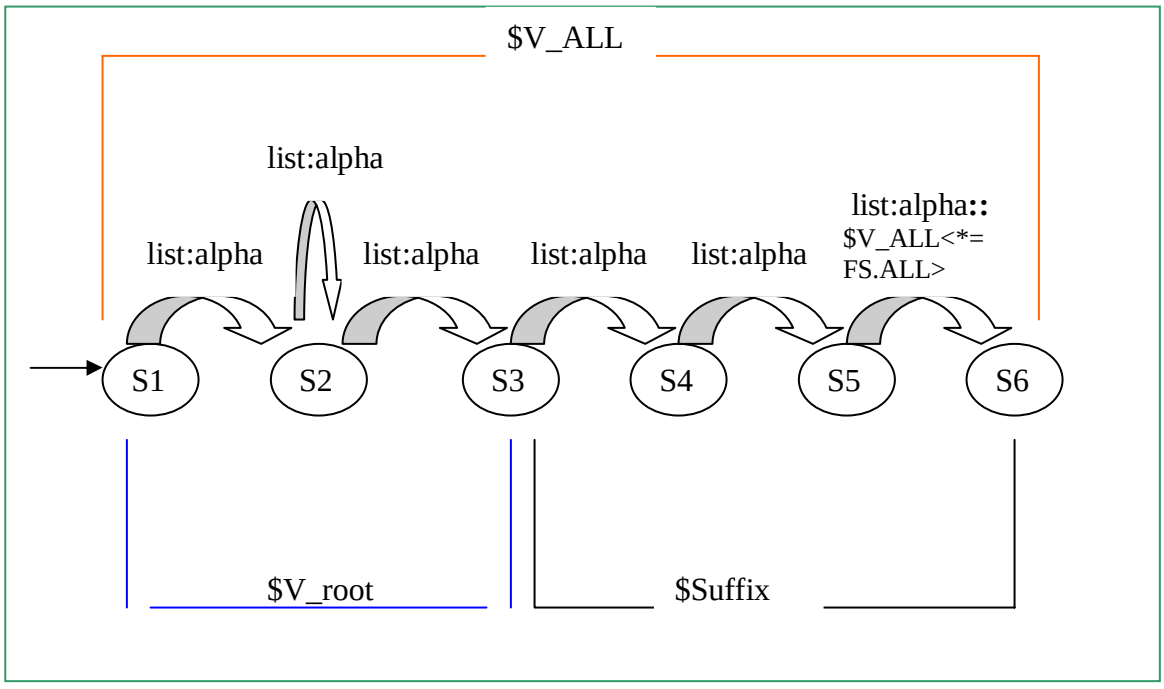
Each table is for a verb of a different paradigm class. For instance, the members in the paradigm class of *eat* are *eat* itself, whereas in the class of *play* we find *pray*, *look*, *talk*, *walk*, *kill* etc. All the members of a class inflect in a similar fashion. Internally, the machine creates paradigm tables for all the members using add/delete rules. To give a table for just one member in a paradigm class will suffice.

Second Method: Using FSTs

Instead of using paradigm tables, we can use just the transducer and define variables specifying the root and suffix. We can do more using operations like addition, deletion, and substitution to handle the morphophonemic changes.

INPUT: eating
OUTPUT: eating<root=eat/cat=v/aspect=progressive>

The transducer used for this will be of the following type.



The input label ‘list’ is calling a list of alphabets ‘alpha’. The input consumed while traversing the arc S1-S3 is stored in the variable $V_root and the input consumed while traversing the arc S4-S6 in the variable $Suffix. The entire word form consumed while

traversing the arc S1-S6 is stored in $V_ALL. All the features associated with the arcs are given as output finally. The transition table is given below. The variables are defined at the end of the transition table, using TAB.

The corresponding transition table would be:

S1
S1 S2 S3 S4 S5 S6
S6

S1	S2	list:alpha		<>	<>	
S2	S2	list:alpha		<>	<>	
S2	S3	list:alpha		<root=$V_root/cat=v>		<>
S3	S4	list:alpha		<>	<>	
S4	S5	list:alpha		<>	<>	
S5	S6	list:alpha	$V_ALL<*=FS.ALL>	<asp=prog>	<>	
$V_root	S1	S3				
$Suffix	S4	S6				
$V_ALL	S1	S6				

The action field of S2-S3 states the root and the category i.e. verb, and the action field of S5-S6 states the features of the suffix i.e aspect is progressive. The final output is given in the output field of the last arc which is S5-S6, where the notation $V_ALL <*=FS.ALL> means 'print all the features' i.e. of the root and suffix.

This transition table works well for words belonging to different paradigm class ending in *ing* like eating, playing, praying, crying etc. where the suffix is *ing* and the rest is the root. [More examples of using variables to represent the changes that occur in a word due to affixation are given in section 4.]

Third Method:

The same data can be represented using both paradigm tables and transducers as follows:
The paradigm table was used for identifying root and suffixes and then dictionaries were used to extract features of the root and suffixes identified.

For the paradigm table, if we give suffixes instead of features, then the parts of a word i.e. the root and the suffix would be identified which then can be given to the transducer with dictionaries. So instead of the following features,

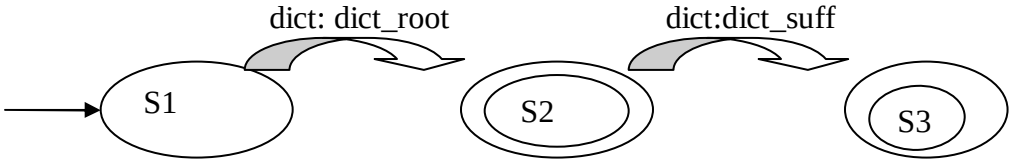
Root
Present tense- 1st person sg|pl / 2nd person sg|pl / 3rd person pl
Present tense- 3rd person sg
Past tense
Perfective Aspect
Progressive Aspect

suffixes may be given. The paradigm table would compile based on the following suffixes.

Root
suffix1
suffix2
suffix3
suffix4
suffix5

At this stage, the analysis of 'eating' would be <root=eat / suffix=ing>. When this output is given to the transducer, then it will extract the features from the respective dictionaries. The transducer for the various forms of the three verbs – eat, play and cry may be as shown:

The other possible ways to represent the same would be:



Where dict_root would have the following entries:

eat <root=eat/cat=verb>
play <root=play/cat=verb>
cry <root=cry/cat=verb>

and dict_suff the following entries.

s <tense = present/ number = sg/ person = 3 >
ed <tense = past >
en <aspect = perfective>
ing <aspect = progressive >

The transducer given above says that a part of a word has to be checked in the root_ dictionary and the rest in affix dictionary. If the given identified parts are correct then the output is given.

For example, the analysis of the word 'crying' would be:
crying<root = cry / aspect = progressive>

APPENDIX 2
Giving FSTs

Transition Tables

The data for the AFSTs for affixation has to be provided in the form of transition tables. FSTs may be given pictorially or by transition tables, which are its textual representation. The following is the format for the transition table.

The first line lists the initial state.
The second line lists all the states.
The third line lists the accepting states.
From the fourth line, the actual data is given using the following six fields separated by TABS:

SOURCE DESTINATION INPUT OUTPUT ACTION CONDITION

The last two fields should be given in angular brackets. If there is no output, then that field should be left blank. For a given arc, the fields mean:

SOURCE = the state denoting the beginning of the arc
DESTINATION = the state denoting the end of the arc
INPUT = it can be a string, dictionary, list, paradigm or another FST
OUTPUT = it is a string with an optional feature structure
ACTION = setting any feature in the current machine
CONDITION = testing any feature in the current machine

The input labels can be any of the following:

str: xxx
FST: xxx
dict: xxx
pdgm: xxx

where ‘xxx’ is any name corresponding to the input label..

The output labels can be any of the strings(a,b,c) along with any of the feature structure (f1,f2,f3):

a. Any string (ex. a string named BEGC)
b. !attr = take the value of the attribute ‘attr’ from feature structure of the current arc
c. FS.attr = take the value of the attribute ‘attr’ from feature structure of current machine.

f1. <attr=value> ; the feature structure of the string
f2. <attr=value/*=!ALL> ; take the value to be printed from child machine, eg. dictionary

f3. <attr=value/*=FS.ALL> ; take the value to be printed from current machine

The field action can be represented by any of the following notation represented using an example:

a. g = f ; the feature structure is set in the current machine
b. g =!g ; the down_arrow symbol '!' indicates that the value of the attribute is called either from a previous phase (in case of a string), or from a child machine (in case of a FST) or from the dictionary specified (in case of a dictionary)
c. * = !ALL ; the notation '*' indicates that all attribute-value pairs have to be taken according to the down_arrow symbol '!'

An example for a transition table:

S1
S1 S2
S2

S1	S2	dict:root_dict	!root<*=!ALL>	<>	<>
S2	S2	dict:affix_dict	$_INP_<*=!ALL>	<*=!ALL>	<>

APPENDIX 3
Notations used in Action field of FST

There are four operations that can be defined on the variables (or attributes of the Feature Structure of the current FST). They are:

1. ADD

syntax:

new_attr=add(arg1,arg2) where arg1 and arg2 can take a variable ($var) or an attribute of the FS of the current FST (FS.attr) or a string (str)

This operation adds the two strings in the arguments of the function and assigns the value to the attribute "new_attr" of the FS of the current FST.

Ex:- final=add($var,abc)

The above operation adds the string contained the value of the variable $var and the string "abc". The final value is stored in the attribute "final" of the FS of the current FST.

Ex: final2=add(abc,FS.attr) //if FS.attr has "def", then "final2" will get the value "abcdef" after the operation.
final3=add($var,FS.attr) //if $var has "abc" and FS.attr has "def", then "final3" will get "abcdef".
final4=add(abc,def) //after the operation "final4" will have "abcdef"

2.DELETE

syntax:

new_attr=delete(arg1,arg2,arg3) where
arg1 can either be a variable ($var) or an attribute of the FS of the current FST (FS.attr).
arg2 is the number of characters to be deleted.
arg3 can take either 0 or 1. 0 indicates delete from the beginning and 1 indicated delete from the end of the string.

This operation deletes the number of string specified in "arg2" from the value of "arg1" and assigns the resulting string to the attribute "new_attr" of the FS of the current FST.

Ex: final=delete($var,2,0) //deletes 2 characters from the start of the value of the variable $var and assigns it to the attribute "final2" of the FS of the current FST.
final2=delete(FS.attr,3,1) //deletes 3 characters from the end of the value of the attribute "attr" of the FS of the current FST and assigns it to the attribute "final2" of the FS of the current FST.

3. REVERSE

syntax:

new_attr=reverse(arg1) where arg1 can either be a variable ($var) or an attribute of the FS of the current FST (FS.attr).

This operation reverses the string in the argument and assigns it to the attribute "new_attr" of the FS of the current FST.

Ex: final=reverse($var) //reverses the string in the variable $var and assigns it to the attribute "final" of the FS of the current FST.

final2=reverse(FS.attr) //reverses the string in the attribute "attr" fo the FS of the curren FST and assigns it to the attribute "final2" of the FS of the current FST.

4. SUBSTR

syntax:

new_attr=substr(arg1,arg2,arg3)
arg1 can either be a variable ($var) or an attribute of the FS of the current FST (FS.attr).
arg2 is the left index
arg3 is the right index

This operation gets the substring from the string given in "arg1".

Ex: final=substr(FS.attr,2,3) //if FS.attr has "abcdef", "final" will get "bc" after the operation.

final=substr(FS.attr,2,2) //if FS.attr has "abcdef", "final" will get "b" after the operation.

final=substr($var,1,4) //if $var has "abcdef", "final" will get "abcd" after the operation.

Appendix 4
Data Specifications for Paradigm Handler Shell

If you are giving data using paradigm tables, then the following files have to be provided in the specified format:

- Ca
- Ce
- Fe
- map_file
- pc_data
- dict.final

Also, one FST calling paradigm handler shell 'pdgm' has to be given.

First, go to the directory PH/anusAraka/hindi/morph/test. Create four files Fe, Ce, Ca.and map_file explained below.

1. Features: Feature enumerator file **[Fe]**:

This file is required in all PH sessions. It has to be created only once in the very beginning and must stay the same way, unaltered, in all subsequent sessions of PH. A change in it may call for a recompilation of the two data files: pc_data and dict_final. Fe has as many lines as there are feature definitions, i.e., each feature definition occupies a line. No feature definition should extend across more than one line. A feature definition consists of a feature followed by a list of feature-values it takes. The number of feature-values following a feature in the same line is referred to as the "length" or "feature-length" of that feature. No two lines in the file can have the same first entry. In case of multiple definitions of a feature the outcome is undefined; PH does NOT produce any messages when it encounters such a situation. The list following the feature must contain at least one element. For example, a valid feature definition is:

number sg pl <enter>

The above definition implies that feature "number" takes two values "sg" and "pl". Feature-Enumerator-File is nothing but a collection of such definitions. A Feature-Enumerator could look like:

number sg pl <enter>
gender m f <enter>
person 1 2 3 <enter>

If feature definition without any feature-values is encountered, PH exits after indicating an error. Invalid feature definitions may look like:

num <enter>
p <enter>
s <enter>

Example: English

gender m f nt
number sg pl
person 1 2 3

Example: Hindi

case dir obl
number_obl sg pl
number_dir sg pl
gender m f
person 1 2 3

2. Categories: Category enumerator file **[Ce]**:

This file contains all category names that the user defines for his language. Each category name is followed by the set of features relevant for that category. The features, as we have seen, are enlisted in the Fe file. In case, a word of certain category has only one form, that category-name will only be specified in the file with no feature associated with it. For example, for words of the category preposition in English, there will be one category name "prep" specified in the Ce file.

A category-enumerator-file based on the sample Feature-Enumerator-File given above may look like the following:

Example: English

noun number
pronoun gender number person

Example: Hindi

noun_m g_m case number
noun_f g_f case number
adj_m_s g_m n_s ANY_c
adj_f_s g_f n_s ANY_c

PH ignores invalid (undefined or misspelt) features and moves on after displaying the corresponding warning. Like the feature-enumerator-file (Fe), the category-enumerator-file (Ce) is referred to in all PH sessions and any modification of this file often leads to recompilation of all data.

3. Map Files: Category name map file **[Ca]** and Feature name map file **[map_file]**

This file has two entries in each line. The first entry is the category-name/feature-name of a valid category/feature specified in the category-enumerator-file (Ce)/ feature-enumerator-file(Fe). Each category/feature name is mapped to a string (consisting of letter followed by any number of letter/digit/"_"), which consists the second entry of each line. This string is to be displayed by the analyzer as an output. Any category not mentioned in this file automatically gets a "" assigned to it as a category-name-map/feature-name-map. In case the category-name/feature-name is itself its own category/feature-name-map then it should be entered twice on the same line. A sample category-name-map-file based on the examples given in the previous definitions is given below.

English:

noun n
pronoun pr

Hindi:

noun_m n
noun_f n
Adj_m_s adj
Adj_f_s adj

As we have seen above, two different category names may be mapped to one category. A sample feature-name-map-file based on the examples given in the previous definitions is given below.

English:

gender g
number num
person p

Hindi:

case case
case_d case
person person
person_1 person

4. Paradigm Tables: [pc_data]

[After giving the information in Ce, Fe , Ca and map_file, go to the directory hindi/morph/pc_data.]
Run the program make_pf.pl. ".pf" file will be generated. Open the file, follow the instruction given and enter your data. If for some field, your language does not have any word form, leave that line as it is. All spelling variants should be entered followed by

slash "/". Once you completed entering the data, close the file. Then run make_p.pl within the same directory.

For example, for English noun, we have assigned above the feature 'number' enumerated as below:

number sg pl

Now using this data, we get the .pf file as:

- root
- sg
- pl

After filling the blanks, it will look like:

Boy root
Boy sg
Boys pl

For other class, same format may be repeated and the data may be given as shown:

Boy root
Boy sg
Boys pl
Man root
Man sg
Men pl
Knife root
Knife sg
Knives pl

Once the data is filled, after compiling we get the following paradigm table.

Boy
Boy
Boys
Man
Man
Men
Knife
Knife
Knives

In this table, the first line is the root and the subsequent lines have the features we had specified.

5. Lexicon: [dict.final]

[Now go to the PH/anusAraka/hindi/dict directory. Open the dict.final file].
This file provides "lex-input" to PH. The syntax of the file is explained below.

Example: English

"dog", "1", "boy", "noun"
"woman","1", "man", "noun"
"wife", "1", "knife", "noun"

Each line will have four entries written within quotes and separated by comma ",". The first entry is the root. The second entry defines the priority of the occurrence of the word in the specified category. At present all roots have been assigned to the priority "1". The third field contains the representative of the paradigm class, i.e., the root "dog" is inflected like the root "boy". The fourth entry specifies the category information.

Transition Table:

One FST pertaining to this has to be given in the FST/data/ directory. The transition table should have as input 'pdgm' as exemplified below.

FST1
[#!pertaining to phase 1]

S1
S1 S2 S3 S4
S4

S1	S2	pdgm:dict_final	!root<*=!ALL>	<*=!ALL>	<>
S2	S3	@	FS.suffa	<>	<>
S3	S4	@	FS.suffb	<>	<>

The notation in the action field of S1-S2 arc '<*=!ALL>' gets the root and suffixes from the paradigm handler shell. The notation in its output field '!root<*=!ALL>' prints the root and its features. Similarly, in the output fields of the subsequent arcs the suffixes and its features, if any, are printed.

The root and the suffixes identified are used as input to the next phase to do the final analysis using root and suffix dictionaries.

Reference:

Radhika Mamidi, Dipti Sharma and Rajeev Sangal. LLSTI Report: **Morphological Analyser FLAN**.LLSTI 3rd Partners Workshop. Lisbon, 22-23 May 2004.