Chapter 5
Implementation

A computer implementation of the content and processes of Aṣṭādhyāyī can be aptly compared with the automatic machines for buying railway tickets. When interacting with such a machine, a traveller needs to supply specific information about the destination, route, date and time, number of travellers, class etc. Depending upon the interactive design of the program this information is required by the machine as and when it needs to take some decision for which the wish and consent of the traveller is necessary. On its part, the machine can put constraints upon the choices depending upon the current situation, e.g. whether an option to take a particular route or train on a particular day is possible or not. Finally, the desired ticket is printed.

In a similar manner, a computer program that implements the derivation of a linguistic expression requires input by the user regarding her or his intention (*vivakṣā*). This information is necessary to make appropriate decisions so that finally the desired expression is formed. The program, on its part, puts constraints upon the possible choices that are admissible.

In the following, I provide an overview of the computer implementation of the Pāṇinian system. As discussed in the previous chapters, my rendering of the grammar is through its formal representation. Therefore, the basis of the present computerisation is the formalisation discussed in chapters 3 and 4.

The programming codes specified here aim to show that my model of the Aṣṭādhyāyī facilitates its computer implementation. Because of the constraints of the space, there is no attempt to furnish the details which run into several thousand lines of program codes. Moreover, thus far there is no user interface for the system. This is because it does not directly contribute to the research questions and would simultaneously require considerable time which is beyond the scope of the present work. The programs are written in the Python Programming Language which runs on Windows, Linux/Unix, Mac

OS X and is available under open source license.[1] Needless to mention, there can be different ways of implementation depending upon the overall design of the program, the nature of the programming environment as well as the choices of the implementational platform.

There are two main aspects of the computer implementation which I will deal with in this chapter. Firstly, how to implement the grammatical content on a computer, and secondly, how to simulate the processes of derivation. The former has to deal with the data-structures of the program and the latter with its application and dynamics.

The basic categories of the formal framework introduced in chapters 2 and 3 aim to represent the grammar in such a way that its application can be effected in an algorithmic manner. The data-structures specified below are developed to render the categories of this formal framework.

5.1 Data-structures

5.1.1 Elements

A collection of elements—components, attributes and meaning-expressions—constitutes the basic database. A typical entry of this database looks as follows:

```
1 paTha_a;
2   ait_9 udAttet_9;
3   p_0;
4   a_0 hrasva_0;
5   Th_1;
6   a_0 hrasva_0 it_0 udAtta_0
```

The above string is separated by semi-colons (;). Here, the first entry (line 1) is the ID of the element. It is the ID of the verbal root `paTha_a`. After this (line 2) the IDs of its attributes are noted with a blank space in between. There are two IDs mentioned here: `ait_9` which stands for the attribute that has the phoneme `a_0` as `it_0` marker. Secondly, the attribute `udAttet_9` says that the marker sound here is high-pitched. This is followed by entries for each phoneme and their attributes. There are four phonemes here (lines 3-6). The final phoneme `a_0` carries the attribute `it_0` and `udAtta_0`, that makes the whole component `ait_9` and `udAttet_9`.

The collection of elements is implemented by the `Elements` class.

[1] The Python Programming Language—Official website `https://www.python.org` (accessed on 22.03.2016) supplies free download of the Python interpreter as well as comprehensive documentation.

```
class Elements:
  def __init__(self):
    self.elements = dictionary_of_elements(ELEMENTS_FILE_NAME)
```

The class variable is a dictionary of elements initialized by the file `ELEMENTS_FILE_NAME`. The class of elements can be instantiated as follows.

```
>>> from elements import Elements
>>> element = Elements()
```

The first command imports the class of Elements and the second one instantiates it. The variable `element` is now an object variable belonging to the Elements class. The class functions can now be executed. For example, the following function returns the boolean value `True` or `False`, depending upon whether an element is present in the database or not.

```
>>> element.is_an_element('paTha_a')
True
```

Another function of this class returns a component as a list of sets, which can be represented as language-components.

```
>>> element.get_component_in_langComp_form('paTha_a')
[set(['paTha_a','udAttet_9','p_0','ait_9']),
 set(['hrasva_0','paTha_a','udAttet_9','a_0','ait_9']),
 set(['paTha_a','udAttet_9','Th_1','ait_9'])]
```

A list consisting of three sets is returned in the above example. These three sets correspond to the three sounds of the said component. It should be noted that certain attributes, like `ait_9` i.e. having a as *it*-marker or `udAttet_9` having an *udātta*-marker are included in the database itself.

The value returned is suitable for representation in the new framework. Thus, the final marker sound is not included as part of the form of the component, but as an attribute. Thus, it has only three sets corresponding to the three sounds /p a ṭh/, and not four sets. This is because the final phoneme in the original corpus is only a marker sound.

5.1.2 Sound-sets

Each of the three sets representing the three phonemes of the component `paTha_a` are stored as sound-sets which are implemented by the class `SoundSets`.

```
class SoundSets:
  def __init__(self,a_set_of_item_ids):
    self.soundSet = a_set_of_item_ids
```

It is instantiated by a set of IDs. For example, the above list of three sets can be represented through three `SoundSets`.

```
>>> from soundSets import SoundSets
>>> soundSet_1 = SoundSets(set(['paTha_a','udAttet_9',
  'p_0','ait_9']))
>>> soundSet_2 = SoundSets(set(['paTha_a','udAttet_9',
  'hrasva_0','a_0','ait_9']))
>>> soundSet_3 = SoundSets(set(['paTha_a','udAttet_9',
  'Th_1','ait_9']))
```

A sound-set contains a collection of IDs. There is, however, an important constraint: it must have *exactly one* element from the following set of fundamental sounds.

```
FUNDAMENTAL_SOUNDS=['a_0','i_0','u_0',
  'R_1','lR_0',
  'e_0','o_0',
  'ai_0','au_0',
  'h_0','y_0','v_0','r_0',
  'l_0',
  'J_1','m_0','G_1','N_1','n_0',
  'jh_0','bh_0',
  'gh_0','Dh_1','dh_0',
  'j_0','b_0','g_0','D_1','d_0',
  'kh_0','ph_0','ch_0','Th_1','th_0','c_0','T_1','t_0',
  'k_0','p_0',
  'z_0','S_1','s_0',
  'h_0',
  'H_1','M_1',
  'x_0']
```

In the above list, IDs in the first fourteen lines correspond to the fourteen Śiva-sūtras. In the 15th line, `H_1` and `M_1` correspond to the aspirated sound *visarjanīya* and the nasal sound *anusvāra* respectively. Finally, `x_0` in the 16th line is for a pause or *virāma*.

Because of the constraint that exactly one ID from the above set must be present within a sound-set, the following set of IDs are invalid candidates for a sound-set.

```
>>> from soundSets import SoundSets
>>> soundSet_4 = SoundSets(set(['paTha_a']))
Invalid sound-set!
```

The `soundSet_4` does not contain any ID from the above set of `FUNDAMENTAL_SOUNDS` and hence is an invalid set for a sound-set.

```
>>> soundSet_5 = SoundSets(set(['a_0','i_0']))
Invalid sound-set!
```

Here, `soundSet_5` has more than one ID from the set of `FUNDAMENTAL_SOUNDS`.

The following function of the class returns the phonetic form of a sound-set. The phonetic form may contain additional attributes like the length or intonation of vowels.

```
>>> soundSet_1.get_phoneme_as_a_set()
set(['p_0'])
>>> soundSet_2.get_phoneme_as_a_set()
set(['hrasva_0','a_0'])
```

A new attribute can be added to a sound-set. The following function implements it.

```
>>> soundSet_2.addAttributes(set(['at_1']))
>>> soundSet_2.get_soundSet()
set(['paTha_a','udAttet_9','a_0','hrasva_0','at_1','ait_9'])
```

Here, the attribute `at_1` is added to a sound-set. While adding the attributes, the consistency condition of the sound-set is taken care of. Thus, an attribute like `dIrgha_0` can not coexist with `hrasva_0` within the same sound-set.

```
>>> soundSet_2.addAttributes(set(['dIrgha_0']))
dIrgha_0 cannot coexist with hrasva_0 !!
```

Similarly, existing attributes can be removed from a sound-set.

```
>>> soundSet_2.remAttributes(set(['at_1']))
>>> soundSet_2.get_soundSet()
set(['paTha_a','udAttet_9','a_0','hrasva_0','ait_9'])
```

5.1.3 Language-components

Language-components constitute an intermediate unit between the whole sentences and the individual sounds. They may roughly (but not necessarily) correspond to an inflected word within a sentence. From the point of view of data-structures, they are a list of sound-sets. The `LangComps` class implements the language-components.

```
class LangComps:
  def __init__(self,list_of_sets=[]):
    self.langComp=[SoundSets(ss) for ss in list_of_sets]
```

In order to initialize a language-component, the component should be rendered in a special form, namely as a sequence of sets of IDs. This is achieved by a function from the `Elements` class.

```
>>> from elements import Elements
>>> element = Elements()
>>> paTha=element.get_component_in_langComp_form('paTha_a')
>>> print paTha
[set(['paTha_a','udAttet_9','p_0','ait_9']),
 set(['hrasva_0','paTha_a','udAttet_9','a_0','ait_9']),
 set(['paTha_a','udAttet_9','Th_1','ait_9'])]
```

An object of the class LangComps can now be instantiated by using the above rendering of the component paTha_a.

```
>>> from langComps import LangComps
>>> langComp = LangComps(paTha)
>>> print langComp
p  : p_0,ait_9,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,paTha_a,udAttet_9 *
Th : Th_1,ait_9,paTha_a,udAttet_9
```

The output shows that the language-component has three sound-sets representing the sounds /p, a, Th/ respectively. The sound-sets contain several other IDs that characterise them as well as the language-component.

A number of functions are required to execute operations on language-components. For example, in order to check the conditions, it is important to identify the range of indices in which some attribute occurs. The question as to which sound-set in a particular language-component has any of the given attributes is implemented by the following function.

```
>>> langComp.range_withAny(['hrasva_0'])
[1]
```

It says that the attribute hrasva_0 is in the second sound-set of the current language-component.[2] In case more than one IDs are searched, then this function returns all indices where *any* of the IDs occur.

```
>>> langComp.range_withAny(['a_0','p_0'])
[0, 1]
```

In the example above, indices of those sound-sets that contain any of the IDs in the list ['a_0','p_0'] are returned.

If one wants to attach the attribute dhAtu_0 to those parts of language-component which contains the ID paTha_a then the range of indices with the ID paTha_a needs to be identified first, followed by the addition of the attribute dhAtu_0 to the respective indices.

```
>>> langComp.range_withAny(['paTha_a'])
[0, 1, 2]
>>> langComp.addAttributes(set(['dhAtu_0']),[0,1,2])
>>> print langComp
p  : p_0,ait_9,dhAtu_0,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,dhAtu_0,paTha_a,udAttet_9 *
Th : Th_1,ait_9,dhAtu_0,paTha_a,udAttet_9
```

[2] The lists in Python programming language are indexed beginning with 0. So the first element of a list list is list[0] and the second one is list[1] etc.

The above example shows that the attribute dhAtu_0 is added to the indices corresponding to the sound-sets that contain paTha_a.

5.1.4 Sentences

Sentences consist of one or more language-components. They represent the whole unit of a typical linguistic expression with the possibility of a number of inflected words. This class is necessary since the rules of grammar consider the whole sentence and not just one word to be the unit of derivation.

From the point of view of a formal representation of grammatical processes, sentences can simply be defined as a sequence of language-components. Accordingly, the class of Sentences is implemented as a list of LangComps.

```
class Sentences:
  def __init__(self,list_of_LangComps=[LangComps()]):
    self.sentence = list_of_LangComps
```

The present formal framework uses three levels to represent any linguistic expression. Sentences correspond to the whole unit of a particular linguistic expression, while sound-sets correspond to the individual sounds. Language-components are an intermediate level between the two and are tentatively related to an inflected word. Depending upon the level from which conditional information can be gathered in a sufficient manner, the grammatical operations can be distinguished as those that apply to a sound-set or to a language-component or at the level of the whole sentence.

5.1.5 Derivational states

The process of derivation is carried out when an operational statement is applied to a sentence or to its constituents, i.e. the language-components or the sound-sets. Together with an operational statement, a sentence forms the next data-structure of the system, namely the class DStates.

```
class DStates: # Derivational state
  def __init__(self,(sentence,statement_string)=(None,None)):
    self.dState = sentence
    self.applied_statement_str = statement_string
```

The application of a particular statement brings about some change in the current state of a sentence. This change may be at the level of a sound-set if, for example, it gets a new attribute, or at the level of a language-component, for example, addition of a new sound-set, or even at the level of a sentence itself, in the case of addition of new language-components. The `dState` variable of the class saves the changed state of the sentence after the application of some statement. The operational statement which is applied is stored in the variable `applied_statement_str` (see section 5.1.8 for the nature of an operational string).

5.1.6 Slices

The current state of a sentence saved in a derivational state is the result of application of an operational statement on the previous state of that sentence. The sequence of such derivational states is stored in a slice and is implemented by the class `Slices`. From the point of view of data-structures, a slice is simply a sequence or list of derivational states or `DStates`.

```
class Slices:
  def __init__(self,list_of_DStates=[]):
    self.slice = list_of_DStates
```

There are two kinds of changes that the operational rules of grammar bring about: either the derivational state is *saturated* or it progresses towards *completion*. The process of saturation is associated with attachment of attributes, while addition of new components is related with incremental steps of completion of the derivational process. Slices contain only those changes where the derivational process is saturated, i.e. only when attributes are added to the components. The other case, when a new component is introduced, leads to the formation of a new slice.

5.1.7 Process-strips

The incremental steps of completion of the process of derivation results in a sequence of slices. Process-strips record this sequences of slices. This is implemented through the class `PStrips`.

```
class PStrips: # process-strips
  def __init__(self,list_of_Slices=[]):
    self.pStrip=list_of_Slices
```

The data-structures introduced thus far adequately represent the constituents and processes of the grammar in an integral manner. The processes of grammar are enacted through the operational rules. The next data-structure comprehends them.

5.1.8 Statements

In the previous chapter, the concept of `statements` in comparison with the *sūtra*s was introduced and specified. Statements are operational rules that are formulated in a formal framework and can be implemented through algorithmic functions.

The information related to a statement is stored in a particular format within the database. In order to point out the structure of the database of statements, consider the entry corresponding to the attachment of the attribute `vRddhi_0`. In the original corpus it is the very first *sūtra*.

```
Xm_ATT_a *
ATT_a & vRddhi_0 *
Xm:[At_2,aic_0] AND Xm_NOT:[vRddhi_0] *
ST_TYPE:[STABILIZING] *
A_RULES:[a_11001]
```

The above entry consists of five parts that are separated by a * sign. Each part is listed here in a separate line.

1. The first part denotes the type of the statement. In the present case, it is given by `Xm_ATT_a`. It implies that this statement is about `ATT`achment of an attribute `a` to some sound-set `Xm`.
2. The second part specifies the operation. In the present case it is given by `ATT_a & vRddhi_0`. It implies that `vRddhi_0` is the `ATT`achment here.
3. The third part notes the conditions that should be fulfilled in order to execute the operation. There are two parts and both of them need to be fulfilled. Hence they are conjoined by the logical `AND`.
   a. The first part of the condition is given by `Xm:[At_2,aic_0]` that can be interpreted as the presence of either the attribute `At_2` or the attribute `aic_0` in the `sound-set` `Xm`.
   b. The second part `Xm_NOT:[vRddhi_0]` ensures that the attribute is attached only if it is not already included in the said sound-set. This is important to avoid recursive attachment of an attribute.
4. The inter-relations between other statements within the database are noted in the fourth part. It also records the nature of the statements. Here, for example, it says that this statement is a `STABILIZING` statement which contributes to the *saturation* of the slice.
5. Finally, the fifth part records the links to the external associations, especially the correspondence with the original corpus of the Aṣṭādhyāyī. Here, the *sūtra* number `a_11001` is noted.

An individual operational statement is implemented by the `Statements` class and is instantiated by the corresponding entry string within the database.

```
class Statements:
  def __init__(self,st_str=''):
    self.statement_string=st_str
    self.statement = parse_a_statement_str(st_str)
```

The function `parse_a_statement_str(st_str)` parses and returns the output as a five-tuple for the instantiation variable.

```
>>> statement_string = 'Xm_ATT_a *
  ATT_a & vRddhi_0 *
  Xm:[At_2,aic_0] AND Xm_NOT:[vRddhi_0] *
  ST_TYPE:[STABILIZING] *
  A_RULES:[a_11001]'
>>> from statements import Statements
>>> statement = Statements(statement_string)
```

This class implements several functions that are important for the application of the statements. One such function is to get the *signature* of a given statement.

```
>>> statement.get_signature()
  'Xm_ATT_a__ATT_a__Xm_Xm_NOT'
```

The signature of a statement specifies its *structure*. Associated with a statement, there is a function which executes its application. The nature of this applicational function is defined by the signature of the statement. All statements with the same signature can be applied by using the same function.

Another function supplies the information about the conditions that need to be fulfilled in order to apply that statement.

```
>>> statement.get_condition_type_vals_dict()
  {'Xm': [['At_2','aic_0']],
   'Xm_NOT': [['vRddhi_0']]}
```

Here, the return value is a dictionary. Its keys are the types or nature of the conditions together with the corresponding values for the particular case. Thus, the first condition type is `Xm` implying that the sound-set must contain any of the IDs `At_2` or `aic_0`. The type of the second condition is `Xm_NOT` and it says that the ID `vRddhi_0` should not be in that sound-set.

5.1.9 Statement groups

The collection of `Statements` is implemented through the `StatementGroups` class.

```
class StatementGroups:
  def __init__(self):
    self.statementGroup = list_of_Statements
```

The functions of this class are primarily meant for organizing and referencing the statements.

```
>>> from statements import StatementGroups
>>> statementGroup = StatementGroups()
>>> statementGroup.get_statements_for_some_operation(
  'ATT_a & guru_0')
  [<statements.Statements instance at 0x10a24c3f8>,
   <statements.Statements instance at 0x10a24c488>]
```

The list returned by the above function consists of two `Statements` instances corresponding to the two statements that provide for application of the attribute guru_0.

```
>>> st1,st2=statementGroup.get_statements_for_some_operation(
  'ATT_a & guru_0')
>>> st1.statement_string
  'Xm_ATT_a * ATT_a & guru_0 * Xm:[hrasva_0] AND Xn:[saMyoga_0]*
   ST_TYPE:[STABILIZING] * A_RULES:[a_14010][a_14011]'
>>> st2.statement_string
  'Xm_ATT_a * ATT_a & guru_0 * Xm:[dIrgha_0] *
   ST_TYPE:[STABILIZING] * A_RULES:[a_14012]'
```

Similarly, the following function returns a dictionary with signature of statements as keys and the corresponding statements as their values.

```
>>> d=statementGroup.get_signature_statements_dict()
>>> d
{'Xm_ATT_a__ATT_a__Xm_Xm_NOT': [
  <statements.Statements instance at 0x10a382638>,
  <statements.Statements instance at 0x10a382950>,
  <statements.Statements instance at 0x10a382998>,
  ...], ... }
```

5.2 Processes of grammar

During the process of their derivation, linguistic expressions are represented through a sentence which consists of one or more language-components corresponding to the individual inflected words. Each language-component contains a sequence of sound-sets. Each sound-set corresponds to a single phoneme.

The derivational process is effected through a number of operational statements which are applied to a sentence. A sentence, together with an operational statement, constitutes a derivational state. There are two fundamental types of operations: (i) to saturate a sentence, in that all attributes that can be attached are added to it, and (ii) to add a new component and graduate towards completion of the derivational process. Accordingly, a slice contains a sequence of derivational states that arise during the process of saturation. A new slice is added, once a new component is introduced. A process-strip records a sequence of slices and thus registers the process of completion.

Given the above framework and corresponding data-structures, the general algorithm of the derivational process can be specified as follows.

```
initialize a process-strip
repeat the following steps:
    saturate the process-strip
    look for completing statements
    if there is no statement to be applied
      return the process-strip
    else:
      select a completing statement
      apply it to the process-strip
```

After initialisation, the process-strip is populated with new components and saturated repeatedly, till there is no admissible component available. This brings the process of derivation to an end.

5.2.1 Initialisation

The process of initialisation is implemented by the function `initialize()`. The initial process-strip contains an empty slice.

```
>>> from pStrips import PStrips
>>> pStrip = PStrips()
>>> pStrip.get_list_of_Slices()
[]
```

The empty `pStrip` needs to be populated with some components. At this moment the meaning-expressions and a user become relevant. The user must be able to express her / his intention (*vivakṣā*) by interacting with the system. Suppose, for example, the user wants to express the sentence *bālakaḥ paṭhati* (a boy recites). Then the choice of the components `bAlaka_k` and `paTha_a` become imminent.[3] The following statement is chosen for application.

```
>>> st01 = Statements(
  'E_ADD_y *
```

[3] For the sake of simplicity and space, I will continue with the derivation of the verbal conjugation only.

```
    ADD_y & paTha_a *
    yM:[vyaktAyAM_x vAci_x] *
    ST_TYPE:[COMPLETING] * ')
>>> st01.get_signature()
  'E_ADD_y__ADD_y__yM'
```

The signature of the above statement is significant for choosing the appropriate `signature-functions`. These are required to execute some statement. Consider the following function for the application of statements with signature: `E_ADD_y__ADD_y__yM`.

```
1  def E_ADD_y__ADD_y__yM(pStrip,statement):
2    (op_type,op_val) = statement.get_operation_part()
3    list_of_sets=Elements().get_component_in_langComp_form(op_val)
4    langComp = LangComps(list_of_sets)
5    sentence = Sentences(
6      ([langComp],statement.get_vals_for_condition_type('yM')))
7    dState = DStates((sentence,statement.get_statement_string()))
8    slice = Slices([dState])
9    pStrip = PStrips([slice])
10   return pStrip
```

`[1]` The function takes `pStrip` and `statement` objects and returns the updated `pStrip` after application of the `statement`. `[2]` The values of operation-type and operation-value are stored in the variables `op_type` and `op_val` respectively. These are `'ADD_y'` and `'paTha_a'`. `[3]` The variable `list_of_sets` contains the list of sets of IDs corresponding to the `op_val` which in this case is `'paTha_a'`. `[4]` The `langComp` is instantiated, followed by `[5]` `sentence`, `[7]` `dState`, `[8]` `slice` and `[9]` `pStrip`. `[10]` The updated `pStrip` is returned.

The application of the above signature function results in the introduction of a new slice within the process-strip. Within this slice a new derivational state is added which records the changes in the sentence and saves the statement that is applied as well. The results are as follows.

```
>>> from signatureFunctions import E_ADD_y__ADD_y__yM
>>> pStrip = E_ADD_y__ADD_y__yM(pStrip,st01)
>>> print pStrip
p  : p_0,ait_9,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,paTha_a,udAttet_9 *
Th : Th_1,ait_9,paTha_a,udAttet_9
 :-: E_ADD_y * ADD_y & paTha_a *
     yM:[vyaktAyAM_x vAci_x] *
     ST_TYPE:[COMPLETING] *
```

5.2.2 Saturation

At this stage the process of saturation is carried out. Consider the following statement.

```
>>> st02 = Statements(
  'Xm_ATT_a *
   ATT_a & at_1 *
   Xm:[a_0][hrasva_0] AND Xm_NOT:[at_1] *
   ST_TYPE:[STABILIZING] *
   A_RULES:[a_11070]')
```

This statement attaches the attribute `at_1` to a sound-set `Xm` that fulfils the following two conditions:

1. `Xm:[a_0][hrasva_0]` i.e. the sound-set `Xm` must contain the phoneme `a_0` and the attribute `hrasva_0`.
2. `Xm_NOT:[at_1]` implies that the attribute should not already be present in the said sound-set. This is necessary to avoid recursive attachment of an attribute.

The signature of this statement is as follows.

```
>>> st02.get_signature()
  'Xm_ATT_a__ATT_a__Xm_Xm_NOT'
```

The implementation of this statement is effected through the following `signature-function`.

```
1  def Xm_ATT_a__ATT_a__Xm_Xm_NOT(pStrip,statement):
2    sentence = _get_a_deepcopy_of_sentence(pStrip)
3    for langComp in sentence.sentence:
4      for soundSet in langComp.langComp:
5        chk_results_dict = _Xm_ATT_a__ATT_a__Xm_Xm_NOT__CHECK(
6          soundSet,statement)
7        if chk_results_dict.get('APPLICABILITY'):
8          soundSet.addAttributes(
9            set([chk_results_dict.get('ATT_a')]))
10         pStrip.get_last_Slice().extend_Slice(
11           DStates((sentence,statement.get_statement_string())))
12   return pStrip
```

`[1]` Again the function takes up the `pStrip` and a `statement` and `[12]` returns the updated strip after the application of the `statement`. `[2]` A deep copy of the sentence is needed to avoid over writing. `[3-4]` Since the operation is executed at the level of sound-sets, the two `for` loops are carried out. `[5-6]` The function `_...__CHECK` checks the conditions whether the `statement` is applicable to the `soundSet` or not. The results of this check function are stored in the dictionary `chk_results_dict`. `[7]` If applicable, then `[8-9]` the appropriate attribute is added to that soundSet, and `[10-11]` `pStrip` gets updated.

The above function uses another function to check the conditions.

```
1  def _Xm_ATT_a__ATT_a__Xm_Xm_NOT__CHECK(soundSet,statement):
2    chk_results_dict = {}
3    oper_part_key, oper_part_val = statement.get_operation_part()
4    if CHK_Xm_Xm_NOT(soundSet,statement):
5      chk_results_dict['APPLICABILITY'] = True
6      chk_results_dict[oper_part_key] = oper_part_val
7    return chk_results_dict
```

[1] The `_...__CHECK` function takes a `soundSet` and a `statement` and returns `chk_results_dict` a dictionary of results. [4] It uses another function `CHK_Xm_Xm_NOT` that checks whether the `Xm` and `Xm_NOT` conditions are fulfilled.

The statement `st02` can now be applied.

```
1  >>> from signatureFunctions import Xm_ATT_a__ATT_a__Xm_Xm_NOT
2  >>> pStrip = Xm_ATT_a__ATT_a__Xm_Xm_NOT(pStrip,st02)
3  >>> print pStrip
4  p   : p_0,ait_9,paTha_a,udAttet_9 *
5  a   : hrasva_0,a_0,ait_9,paTha_a,udAttet_9 *
6  Th  : Th_1,ait_9,paTha_a,udAttet_9
7   :-: E_ADD_y * ADD_y & paTha_a *
8       yM:[vyaktAyAM_x vAci_x] *
9       ST_TYPE:[COMPLETING] *
10  :::
11 p   : p_0,ait_9,paTha_a,udAttet_9 *
12 a   : hrasva_0,a_0,ait_9,at_1,paTha_a,udAttet_9 *
13 Th  : Th_1,ait_9,paTha_a,udAttet_9
14  :-: Xm_ATT_a * ATT_a & at_1 *
15      Xm:[a_0][hrasva_0] AND Xm_NOT:[at_1] *
16      ST_TYPE:[STABILIZING] * A_RULES:[a_11070]
```

The application of the above statement has resulted in an extension of the slice. [10] A new derivational state is added. [12] The effect of this function is visible in this line, where the sound-set gets the attribute `at_1` added to it. This is the only sound-set where the conditions of the statement are fulfilled.

It should be noted that multiple application of this statement to the same `pStrip` does not result in any further changes.

```
1  >>> pStrip = Xm_ATT_a__ATT_a__Xm_Xm_NOT(pStrip,st02)
2  >>> pStrip = Xm_ATT_a__ATT_a__Xm_Xm_NOT(pStrip,st02)
3  >>> print pStrip
4  p   : p_0,ait_9,paTha_a,udAttet_9 *
5  a   : hrasva_0,a_0,ait_9,paTha_a,udAttet_9 *
6  Th  : Th_1,ait_9,paTha_a,udAttet_9
7   :-: E_ADD_y * ADD_y & paTha_a *
8       yM:[vyaktAyAM_x vAci_x] *
9       ST_TYPE:[COMPLETING] *
10  :::
11 p   : p_0,ait_9,paTha_a,udAttet_9 *
```

```
a  : hrasva_0,a_0,ait_9,at_1,paTha_a,udAttet_9 *
Th : Th_1,ait_9,paTha_a,udAttet_9
 :-: Xm_ATT_a * ATT_a & at_1 *
     Xm:[a_0][hrasva_0] AND Xm_NOT:[at_1] *
     ST_TYPE:[STABILIZING] * A_RULES:[a_11070]
```

Consider now the following two statements.

```
>>> st03 = Statements(
  'Xm_ATT_a *
   ATT_a & vRddhi_0 *
   Xm:[At_2,aic_0] AND Xm_NOT:[vRddhi_0] *
   ST_TYPE:[STABILIZING] * A_RULES:[a_11001]')
>>> st04 = Statements(
  'Xm_ATT_a *
   ATT_a & guNa_0 *
   Xm:[at_1,eG_0] AND Xm_NOT:[guNa_0] *
   ST_TYPE:[STABILIZING] * A_RULES:[a_11002]')
```

Both of them have the same signature as the previous statement.

```
>>> st03.get_signature()
  'Xm_ATT_a__ATT_a__Xm_Xm_NOT'
>>> st04.get_signature()
  'Xm_ATT_a__ATT_a__Xm_Xm_NOT'
```

This implies that both of them can be applied by using the same signature function. The execution of st03 shows that it is not applied at all. This is obvious as none of the sound-sets fulfill the required conditions. The pStrip remains as it is. This is evident from the output below.

```
>>> pStrip = Xm_ATT_a__ATT_a__Xm_Xm_NOT(pStrip,st03)
>>> print pStrip
p  : p_0,ait_9,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,paTha_a,udAttet_9 *
Th : Th_1,ait_9,paTha_a,udAttet_9
 :-: E_ADD_y * ADD_y & paTha_a *
     yM:[vyaktAyAM_x vAci_x] *
     ST_TYPE:[COMPLETING] *
 :::
p  : p_0,ait_9,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,at_1,paTha_a,udAttet_9 *
Th : Th_1,ait_9,paTha_a,udAttet_9
 :-: Xm_ATT_a * ATT_a & at_1 *
     Xm:[a_0][hrasva_0] AND Xm_NOT:[at_1] *
     ST_TYPE:[STABILIZING] * A_RULES:[a_11070]
```

The execution of st04 however brings out some changes.

```
>>> pStrip = Xm_ATT_a__ATT_a__Xm_Xm_NOT(pStrip,st04)
>>> print pStrip
p  : p_0,ait_9,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,paTha_a,udAttet_9 *
```

```
Th : Th_1,ait_9,paTha_a,udAttet_9
 :-: E_ADD_y * ADD_y & paTha_a *
     yM:[vyaktAyAM_x vAci_x] *
     ST_TYPE:[COMPLETING] *
 :::
p  : p_0,ait_9,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,at_1,paTha_a,udAttet_9 *
Th : Th_1,ait_9,paTha_a,udAttet_9
 :-: Xm_ATT_a * ATT_a & at_1 *
     Xm:[a_0][hrasva_0] AND Xm_NOT:[at_1] *
     ST_TYPE:[STABILIZING] * A_RULES:[a_11070]
 :::
p  : p_0,ait_9,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,at_1,guNa_0,paTha_a,udAttet_9 *
Th : Th_1,ait_9,paTha_a,udAttet_9
 :-: Xm_ATT_a * ATT_a & guNa_0 *
     Xm:[at_1,eG_0] AND Xm_NOT:[guNa_0] *
     ST_TYPE:[STABILIZING] * A_RULES:[a_11002]
```

[18] The attribute guNa_0 is now attached to the sound-set having the attribute at_1. There is a hierarchy among the saturating statements that is decided by the dependency of their conditions together with the operational attachment. Thus, while st03 and st04 can be applied in any order, st04 can only be applied after the application of st02.

The next statement attaches an attribute not only to just one sound-set, but to several of them.

```
>>> st05 = Statements(
  'X_ATT_a *
   ATT_a & bhvAdi_0 *
   X:[bhU_a,paTha_a,ji_a,pUG_a,dheT_a] AND X_NOT:[bhvAdi_0] *
   ST_TYPE:[STABILIZING] * ')
>>> st05.get_signature()
  'X_ATT_a__ATT_a__X_X_NOT'
```

This statement attaches the group name bhvAdi_0 for the *bhvādi-gaṇa* of the verbal roots. Here, for the sake of readability, I have not listed all the roots that are mentioned in this group.

```
>>> from signatureFunctions import X_ATT_a__ATT_a__X_X_NOT
>>> pStrip = X_ATT_a__ATT_a__X_X_NOT(pStrip,st05)
>>> print pStrip
 ...
 :::
p  : p_0,ait_9,bhvAdi_0,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,at_1,bhvAdi_0,guNa_0,paTha_a,
     udAttet_9 *
Th : Th_1,ait_9,bhvAdi_0,paTha_a,udAttet_9
 :-: X_ATT_a * ATT_a & bhvAdi_0 *
     X:[bhU_a,paTha_a,ji_a,pUG_a,dheT_a] AND X_NOT:[bhvAdi_0] *
     ST_TYPE:[STABILIZING] *
```

The attribute bhvAdi_0 is added to all three of the sound-sets that constitute the component paTha_a. In the above output, I have deleted the earlier derivational states to save space and to aid readability.

The next statement that attaches the attribute dhAtu_0 is similar.

```
>>> st06 = Statements(
  'X_ATT_a *
   ATT_a & dhAtu_0 *
   X:[bhvAdi_0,adAdi_0,juhotyAdi_0,divAdi_0,svAdi_0,
      tudAdi_0,rudhAdi_0,tanAdi_0,kryAdi_0,curAdi_0] AND
   X_NOT:[dhAtu_0] *
   ST_TYPE:[STABILIZING] * A_RULES:[a_13001]')
>>> st06.get_signature()
  'X_ATT_a__ATT_a__X_X_NOT'
>>> from signatureFunctions import X_ATT_a__ATT_a__X_X_NOT
>>> pStrip = X_ATT_a__ATT_a__X_X_NOT(pStrip,st06)
>>> print pStrip
 ...
 :::
p  : p_0,ait_9,bhvAdi_0,dhAtu_0,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,at_1,bhvAdi_0,dhAtu_0,guNa_0,
     paTha_a,udAttet_9 *
Th : Th_1,ait_9,bhvAdi_0,dhAtu_0,paTha_a,udAttet_9
 :-: X_ATT_a * ATT_a & dhAtu_0 *
     X:[bhvAdi_0,adAdi_0,juhotyAdi_0,divAdi_0,svAdi_0,
        tudAdi_0,rudhAdi_0,tanAdi_0,kryAdi_0,curAdi_0] AND
     X_NOT:[dhAtu_0] *
     ST_TYPE:[STABILIZING] * A_RULES:[a_13001]
```

5.2.3 Completion

The next statement introduces the component laT_0. This component is added *after* the language-component Xi. The semantic condition for its addition is stated in the condition yM:[vartamAna_x]. This condition is satisfied once the user confirms her or his intention to express vartamAna_x or present tense.

```
>>> st07 = Statements(
  'Xi_ADD_y *
   ADD_y & laT_0 *
   Xi:[dhAtu_0] AND Xj_NOT:[lakAra_9] AND yM:[vartamAna_x] *
   ST_TYPE:[COMPLETING] * A_RULES:[a_32123]')
>>> st07.get_signature()
  'Xi_ADD_y__ADD_y__Xi_Xj_NOT_yM'
```

The above statement is applied through the corresponding signature function. It is a completing statement and therefore a new slice is added. Line [16] notes the slice boundary.

```
>>> from signatureFunctions import Xi_ADD_y__ADD_y__Xi_Xj_NOT_yM
>>> pStrip = Xi_ADD_y__ADD_y__Xi_Xj_NOT_yM(pStrip,st07)
>>> print pStrip
 ...
 :::
p  : p_0,ait_9,bhvAdi_0,dhAtu_0,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,at_1,bhvAdi_0,dhAtu_0,guNa_0,
     paTha_a,udAttet_9 *
Th : Th_1,ait_9,bhvAdi_0,dhAtu_0,paTha_a,udAttet_9
 :-: X_ATT_a *
     ATT_a & dhAtu_0 *
     X:[bhvAdi_0,adAdi_0,juhotyAdi_0,divAdi_0,svAdi_0,
      tudAdi_0,rudhAdi_0,tanAdi_0,kryAdi_0,curAdi_0] AND
     X_NOT:[dhAtu_0] *
     ST_TYPE:[STABILIZING] * A_RULES:[a_13001]
-::-::-
p  : p_0,ait_9,bhvAdi_0,dhAtu_0,paTha_a,udAttet_9 *
a  : hrasva_0,a_0,ait_9,at_1,bhvAdi_0,dhAtu_0,guNa_0,
     paTha_a,udAttet_9 *
Th : Th_1,ait_9,bhvAdi_0,dhAtu_0,paTha_a,udAttet_9 *
l  : l_0,Tit_9,ait_9,laT_0,lakAra_9
 :-: Xi_ADD_y *
     ADD_y & laT_0 *
     Xi:[dhAtu_0] AND Xj_NOT:[lakAra_9] AND yM:[vartamAna_x] *
     MEXGRP_0001 * A_RULES:[a_32123]
```

[21] The above statement also extends the language-component and a new sound-set for laT_0 is added. laT_0 is a Tit_9 and ait_9 lakAra_9 and these attributes are already specified in the database.

The next statement specifies whether the components within a particular language-component form a sentence in active, passive or middle voice. Again, the decision to employ active voice is reached on the basis of the semantic condition, which the user must address directly.

```
>>> st08 = Statements(
  'X_ATT_a *
   ATT_a & xkartR_9 *
   X:[dhAtu_0] AND X_NOT:[xkarman_9][xbhAva_9][xkartR_9] AND
   X_M:[kartRprayoga_x] *
   ST_TYPE:[STABILIZING] *')
>>> st08.get_signature()
  'X_ATT_a__ATT_a__X_X_M_X_NOT'
>>> from signatureFunctions import X_ATT_a__ATT_a__X_X_M_X_NOT
>>> pStrip = X_ATT_a__ATT_a__X_X_M_X_NOT(pStrip,st08)
>>> print pStrip
 :::
p  : p_0,ait_9,bhvAdi_0,dhAtu_0,paTha_a,udAttet_9,xkartR_9 *
a  : hrasva_0,a_0,ait_9,at_1,bhvAdi_0,dhAtu_0,guNa_0,
     paTha_a,udAttet_9,xkartR_9 *
Th : Th_1,ait_9,bhvAdi_0,dhAtu_0,paTha_a,udAttet_9,
     xkartR_9 *
l  : l_0,Tit_9,ait_9,laT_0,lakAra_9
```

```
 :-: X_ATT_a *
     ATT_a & xkartR_9 *
     X:[dhAtu_0] AND X_NOT:[xkarman_9][xbhAva_9][xkartR_9] AND
     X_M:[kartRprayoga_x] *
     ST_TYPE:[STABILIZING] *
```

The following statement attaches the attribute parasmaipada_0 to the language-component.

```
>>> st09 = Statements(
  'X_ATT_a *
   ATT_a & parasmaipada_0 *
   X:[dhAtu_0][xkartR_9] AND X_NOT:[Atmanepada_0] *
   ST_TYPE:[STABILIZING] *')
>>> st09.get_signature()
  'X_ATT_a__ATT_a__X_X_NOT'
>>> from signatureFunctions import X_ATT_a__ATT_a__X_X_NOT
>>> pStrip = X_ATT_a__ATT_a__X_X_NOT(pStrip,st09)
>>> print pStrip
 ...
 :::
p  : p_0,ait_9,bhvAdi_0,dhAtu_0,paTha_a,parasmaipada_0,
     udAttet_9,xkartR_9 *
a  : hrasva_0,a_0,ait_9,at_1,bhvAdi_0,dhAtu_0,guNa_0,
     paTha_a,parasmaipada_0,udAttet_9,xkartR_9 *
Th : Th_1,ait_9,bhvAdi_0,dhAtu_0,paTha_a,parasmaipada_0,
     udAttet_9,xkartR_9 *
l  : l_0,Tit_9,ait_9,laT_0,lakAra_9
 :-: X_ATT_a *
     ATT_a & parasmaipada_0 *
     X:[dhAtu_0][xkartR_9] AND X_NOT:[Atmanepada_0] *
     ST_TYPE:[STABILIZING] *
```

The following statement provides for the substitution of laT_0 by the third person singular suffix tip_0.

```
>>> st10 = Statements(
  'Xi_REP_y *
   REP_y & tip_0 *
   Xh:[dhAtu_0][parasmaipada_0] AND Xi:[lakAra_9] AND
   yM:[prathama_0][ekavacana_0] *
   ST_TYPE:[COMPLETING] * A_RULES:[a_34077][a_34078]')
>>> ST10_obj.get_signature()
  'Xi_REP_y__REP_y__Xh_Xi_yM'
```

Again, the decision to opt for the third person and singular is taken by the user on the basis of the semantic conditions prathama_0 and ekavacana_0 respectively.

```
>>> from signatureFunctions import Xi_REP_y__REP_y__Xh_Xi_yM
>>> pStrip = Xi_REP_y__REP_y__Xh_Xi_yM(pStrip,st10)
>>> print pStrip
-::-::-
```

```
5  p  : p_0,ait_9,bhvAdi_0,dhAtu_0,paTha_a,parasmaipada_0,
6       udAttet_9,xkartR_9 *
7  a  : hrasva_0,a_0,ait_9,at_1,bhvAdi_0,dhAtu_0,guNa_0,
8       paTha_a,parasmaipada_0,udAttet_9,xkartR_9 *
9  Th : Th_1,ait_9,bhvAdi_0,dhAtu_0,paTha_a,parasmaipada_0,
10      udAttet_9,xkartR_9 *
11 l  : l_0,REPLACED_9,Tit_9,ait_9,laT_0,lakAra_9 *
12 t  : t_0,pit_9,tip_0 *
13 i  : hrasva_0,i_0,pit_9,tip_0
14  :-: Xi_REP_y *
15      REP_y & tip_0 *
16      Xh:[dhAtu_0][parasmaipada_0] AND Xi:[lakAra_9] AND
17      yM:[prathama_0][ekavacana_0] *
18      ST_TYPE:[COMPLETING] * A_RULES:[a_34077][a_34078]
```

Replacement is implemented by attaching the attribute REPLACED_9 to those parts that are replaced (line [11]) and adding the replacement components at the appropriate index (line [12-13]).

The following statement attaches the attribute tiG_0 to the third person singular suffix tip_0.

```
>>> st11 = Statements(
  'X_ATT_a *
   ATT_a & tiG_0 *
   X:[tip_0,tas_0,jhi_0,sip_0,thas_0,tha_0,mip_0,vas_0,mas_0,
    ta_0,AtAm_0,jha_0,thAs_1,AthAm_0,dhvam_0,iT_1,vahi_0,mahiG_0]
        AND
   X_NOT:[tiG_0] *
   ST_TYPE:[STABILIZING] * A_RULES:[a_34078][a_11071]')
>>> st11.get_signature()
  'X_ATT_a__ATT_a__X_X_NOT'
>>> from signatureFunctions import X_ATT_a__ATT_a__X_X_NOT
>>> pStrip = X_ATT_a__ATT_a__X_X_NOT(pStrip,st11)
>>> print pStrip
 :::
p  : p_0,ait_9,bhvAdi_0,dhAtu_0,paTha_a,parasmaipada_0,
     udAttet_9,xkartR_9 *
a  : hrasva_0,a_0,ait_9,at_1,bhvAdi_0,dhAtu_0,guNa_0,
     paTha_a,parasmaipada_0,udAttet_9,xkartR_9 *
Th : Th_1,ait_9,bhvAdi_0,dhAtu_0,paTha_a,parasmaipada_0,
     udAttet_9,xkartR_9 *
l  : l_0,REPLACED_9,Tit_9,ait_9,laT_0,lakAra_9 *
t  : t_0,pit_9,tiG_0,tip_0 *
i  : hrasva_0,i_0,pit_9,tiG_0,tip_0
 :-: X_ATT_a *
     ATT_a & tiG_0 *
     X:[tip_0,tas_0,jhi_0,sip_0,thas_0,tha_0,mip_0,vas_0,mas_0,
        ta_0,AtAm_0,jha_0,thAs_1,AthAm_0,dhvam_0,iT_1,vahi_0,
            mahiG_0] AND
     X_NOT:[tiG_0] *
   ST_TYPE:[STABILIZING] * A_RULES:[a_34078][a_11071]
```

To a tiG_0 or a zit_9 component, the attribute sArvadhAtuka_0 is attached by the following statement.

```
>>> st12 = Statements(
  'X_ATT_a *
   ATT_a & sArvadhAtuka_0 *
   X:[tiG_0,zit_9] AND X_NOT:[sArvadhAtuka_0] *
   ST_TYPE:[STABILIZING] * A_RULES:[a_34113]')
>>> st12.get_signature()
  'X_ATT_a__ATT_a__X_X_NOT'
```

The application of this statement is similar to the other attachments of the attributes.

```
>>> from signatureFunctions import X_ATT_a__ATT_a__X_X_NOT
>>> pStrip = X_ATT_a__ATT_a__X_X_NOT(pStrip,st12)
>>> print pStrip
 ...
 :::
p  : p_0,ait_9,bhvAdi_0,dhAtu_0,paTha_a,parasmaipada_0,
     udAttet_9,xkartR_9 *
a  : hrasva_0,a_0,ait_9,at_1,bhvAdi_0,dhAtu_0,guNa_0,
     paTha_a,parasmaipada_0,udAttet_9,xkartR_9 *
Th : Th_1,ait_9,bhvAdi_0,dhAtu_0,paTha_a,parasmaipada_0,
     udAttet_9,xkartR_9 *
l  : l_0,REPLACED_9,Tit_9,ait_9,laT_0,lakAra_9 *
t  : t_0,pit_9,sArvadhAtuka_0,tiG_0,tip_0 *
i  : hrasva_0,i_0,pit_9,sArvadhAtuka_0,tiG_0,tip_0
 :-: X_ATT_a *
     ATT_a & sArvadhAtuka_0 *
     X:[tiG_0,zit_9] AND X_NOT:[sArvadhAtuka_0] *
     ST_TYPE:[STABILIZING] * A_RULES:[a_34113]
```

Finally, the infix zap_0 is introduced by the following statement.

```
>>> st13 = Statements(
  'Xi_ADD_y *
   ADD_y & zap_0 *
   Xi:[dhAtu_0] AND Xj:[sArvadhAtuka_0] AND Xj_M:[kartR_0] * *
   A_RULES:[a_31068]')
>>> st13.get_signature()
  'Xi_ADD_y__ADD_y__Xi_Xj_Xj_M'
```

The conditions also provide the index where the new components should properly be added.

```
>>> from signatureFunctions import Xi_ADD_y__ADD_y__Xi_Xj_Xj_M
>>> pStrip = Xi_ADD_y__ADD_y__Xi_Xj_Xj_M(pStrip,st13)
>>> print pStrip
 ...
-::-::-
p  : p_0,ait_9,bhvAdi_0,dhAtu_0,paTha_a,parasmaipada_0,
     udAttet_9,xkartR_9 *
a  : hrasva_0,a_0,ait_9,at_1,bhvAdi_0,dhAtu_0,guNa_0,
     paTha_a,parasmaipada_0,udAttet_9,xkartR_9 *
Th : Th_1,ait_9,bhvAdi_0,dhAtu_0,paTha_a,parasmaipada_0,
     udAttet_9,xkartR_9 *
a  : hrasva_0,a_0,pit_9,zap_0,zit_9 *
l  : l_0,REPLACED_9,Tit_9,ait_9,laT_0,lakAra_9 *
t  : t_0,pit_9,sArvadhAtuka_0,tiG_0,tip_0 *
i  : hrasva_0,i_0,pit_9,sArvadhAtuka_0,tiG_0,tip_0
 :-: Xi_ADD_y *
     ADD_y & zap_0 *
     Xi:[dhAtu_0] AND Xj:[sArvadhAtuka_0] AND Xj_M:[kartR_0] * *
     A_RULES:[a_31068]
```

The above steps demonstrate the dynamics of the process of derivation. At each step, a larger number of characterising statements are employed. The main algorithm of the derivational process is summarized in the following main function.

```
1  def execute():
2    pStrip = initialize()
3    while 1:
4      pStrip = saturate(pStrip)
5      possible_statements = interpret(pStrip)
6      if len(possible_statements) == 0:
7        return pStrip
8      else:
9        statement = select(pStrip,possible_statements)
10       pStrip = apply(pStrip,statement)
```

[1] The main function which returns a process-strip. [2] The function `initialize()` initializes the process-strip. [3] The third line specifies a loop. [4] At this stage, a given process-strip is saturated, i.e. attributes are attached to the language-components or sound-sets. [5] Once the process-strip is saturated, it is tested for any possible completing statement that could be applied. The list of all such candidates is stored in `possible_statements`. [6] In case the list is empty, i.e. there is no statement that may be applied, [7] then the process-strip is returned. Otherwise, [9] one statement is selected, depending upon the semantic considerations and the intention (*vivakṣā*) of the user. [10] Finally, that statement is applied and the process-strip is updated.