Shakti Natural Language Analyzer: SSF Representation

Akshar Bharati
Rajeev Sangal
Dipti M Sharma

Language Technologies Research Centre
International Institute of Information Technology
Hyderabad, India
{sangal,dipti}@iiit.ac.in

Abstract

Shakti analysis system has been designed to be used for a variety of purposes. The system design is such that the overall analysis task has been broken up into as many as 40 modules (with at least 10 major modules). Many of the modules are along recent research lines in NLP, and some of them handle special problems. All this allows different research advances in NLP to be easily incorporated in the system.

Technically, the system uses a combination of rule-based and statistical approaches. For representation of analysis, it uses phrase-structure at the lowest level (chunks or simple phrases), and dependency relations among the chunks.

At the heart of the system, is a highly readable common representation called Shakti Standard Format (SSF), which is a kind of blackboard, on which all modules operate. This permits partial analysis to be represented and operated upon by different modules. It also allows the modules to be located on different machines, if necessary.

Although, currently Shakti Analyzer is available for only English, it is expected to be available for several Indian languages in due course of time. It has been used for analyzing English in Shakti MT System.

1 Introduction

Shakti Analyzer has been designed for analyzing natural languages. Currently, it is available for analysing English. At a future date, it will also be available for analyzing a number of Indian languages.

The Shakti Analyzer can incorporate new modules as black boxes or as open-source software. The simplicity of the overall architecture makes it easy to do so. Different available English parsers have been extensively adapted, and the current version runs using Collins parser.

The system has a number of innovative design principles which are described below.

2 System Organization Principles

A number of system organization principles have been used which have led to the rapid development of the system. While the principles by themselves might not appear to be new, their application in this manner is unique.

2.1 Modularity

The system consists of a large number of modules, each one of which typically performs a small logical task. This allows the overall machine translation task to be broken up into a large number of small sub-tasks each of which can be accomplished separately. Currently the system has 69 different modules. About 9 modules are used for analyzing the source language (English), 24 modules are used for performing bilingual tasks such as substituting target language roots and reordering etc., and the remaining modules are used for generating target language.

2.2 Simplicity of Organization

The overall system architecture is kept extremely simple. All modules operate on data in a standard format called Shakti standard format (SSF). SSF allows information to be represented in the form of one or more trees together with a set of attribute-value pairs on nodes of the trees.

The attribute value pairs allow features or properties to be specified with every node. Relations of different types across nodes can also be specified using an attribute-value like representation. The representation is specially designed to allow different levels and kinds of linguistic analyses to be stored. The developer uses APIs to store or access information regarding structure of trees and attribute value pairs.

If a module is successful in its task, it adds a new attribute or analysis to the output stream (in the same SSF). Thus, even though the format is fixed, it is extensible in terms of attributes or analyses. This approach allows other ready made packages (such as, POS tagger, chunker, and parser) to be used easily. In order to interface such packages to the Shakti system, all that is required is to convert the output to SSF, and the rest of the modules continue to operate seamlessly.

The format has both in-memory representation as well as stream representation. They are inter-convertible using a reader (stream to memory) and printer (memory to stream). The in-memory representation is good in speed of processing, while the stream is good for portability, heterogenous machines, and flexibility, in general.

This approach follows the dictum: "Simplify globally, complicate, if at all, locally." However, since the number of modules is large and each local module does a small job, the local complexity of individual modules remains under tight control for most of the modules. Even if complexity is introduced, it remains localized.

2.3 Designed to Deal with Failure

NLP analysis modules are known to have limited coverage. They are not always able to produce an output. They fail to produce output either because of limits of the best known algorithms or incompleteness of data or rules. For example, a sentential parser might fail to parse either because it does not know how to deal with a construction or because a dictionary entry is missing. Similarly, a chunker or part of speech tagger might fail, at times, to produce an analysis. The system is designed to deal with failure at every level. This is facilitated by a

common representation for the outputs of POS tagger, chunker and parser (all in SSF). The downstream modules continue to operate on the data stream, albeit less effectively, when a more detailed analysis is not available. (If all modules were to fail, a default rule of no-reordering and dictionary lookup would still be applied.)

As another example, if the word sense disambiguation (WSD) module fails to identify the sense of a word in the input sentence, it does not put in the sense feature for the word. This only means that the module which substitutes the target language root from the available equivalents from dictionary, will use a default rule for selecting the sense because the detailed WSD was not successful (say, due to lack of training data).

The SSF is designed to represent partial information, routinely. Appropriate modules know what to do when their desired information is available and use defaults when it is not available. In fact, for many modules, there are not just two but several levels at which they operate, depending on availability of information corresponding to that level. Each level represents a graceful degradation of output quality.

The above flexibility is achieved by using two kinds of representation: constituent level representation and feature-structure level representation. The former is used to store phrase level analysis (and partial parse etc.) and the latter for outputs of many kinds of other tasks such as WSD, TAM computation, case computation, dependency relations, etc.

2.4 Transparency for Developers

An extremely important characteristic for the successful development of complex software such as a machine translation system is to expose the input and output produced by every module. This transparency becomes even more important in a research environment where new ideas are constantly being tried with a high turnover of student developers.

In the Shakti system, unprecedented transparency is achieved by inventing a highly readable textual notation for the SSF, and requiring every module to produce output in this format. In fact, the textual SSF output of amodule is not only for the human consumption, but is used by the subsequent module in the data stream as its input. This ensures that no part of the resulting analysis is left hidden in some global variables; all analysis is represented in readable SSF (otherwise it is not processed at all by the subsequent modules).

Experience has shown that this methodology has made debugging as well as the development of the system convenient for programmers and linguists alike. In case, an output is not as expected, one can quickly find out which module went wrong (that is, which module did not function as expected). In fact, linguists are using the system quite effectively to debug their linguistic data with ease.

3 Technical Design Decisions

In the Shakti analyzer we have taken a number of decisions regarding the nature of analyses or generation to be performed. Our approach is summarized here.

3.1 Hybrid Approach

Shakti analyzer combines rule-based approach with statistical approach. The SSF representation is designed to keep both kinds of information. The rules are mostly linguistic in nature, and the statistical approach tries to infer or use linguistic information. For example, statistical POS tagger tries to infer linguistic (part-of-speech) tags, whereas WSD module uses grammatical relations together with statistics to disambiguate the word sense.

However, the rule-based component is not always linguistic. Some modules also use semantic information. For example, such a markup is used in domain specific filters to recognize and name the entity relevant to the domain. Such a markup is also used in a module that accepts translations of multi-word expressions given by bilinguals who are not linguists.

Although the system accommodates multiple approaches, the backbone of the system is linguistic analysis. Statistical and other approaches are interpreted in linguistic terms wherever possible. This allows generalizations to be performed which are not possible otherwise. Although the system has a strong linguistic bias, it is also capable of dealing with non-linguistic information.

3.2 Constituent Structure with Dependency Relations

The system takes advantage of phrase structure or constituency structure at the bottom most level. It represents the analysis in terms of chunks, i.e., non-recursive noun phrases, prepositional phrases, verb groups, etc. Relations among these chunks are shown using dependency relations. This leads to two advantages:

1. Compactness and transparency of representation, and
2. Flexibility in approach or method to be used in identifying relations.

The latter follows because a module with "expertise" in identifying a particular kind of relation can be called without waiting to build the entire constituent structure. For example, a statistics based PP-attachment module can do its job given the chunks and some appropriate relations between them without waiting or trying to build the entire parse tree for the sentence first. It is much more difficult to modularize the building of constituent structure.

3.3 Named Entity Recognition and Information Extraction

Named entities occur quite frequently in the text. Domain specific modules for recognizing and marking up expressions that name entities in a text, may be run first. Such marked up expressions are then treated as a unit, and sentence analysis continued.

3.4 Sub-dividing Problems along Standard Lines

The modularity of the system allows problems to be broken up along well understood lines when convenient, and in novel ways when necessary. Thus, PP-attachment can be processed independent of the rule-based parser as it is a well understood problem for which statistical techniques work well. On the other hand, identification of phrasal verbs (i.e., verb-particle pairs) is not handled well by most existing parsers. In this case statistical techniques can be tried and the resulting phrasal verb groups can be passed over to the parser.

4 Representation of Analysis: Shakti Standard Format (SSF)

As mentioned earlier, the Shakti Standard Format (SSF) is used for representing the analysis of a sentence. It is especially designed to represent the different kinds of linguistic analysis, as well as different levels of analysis. It can routinely represent partial analysis.

All the modules of Shakti analyzer operate on the data in SSF. Thus, the input and output are in the same format. Appropriate modules know what to do when their desired information is

available and use defaults when it is not available. If a module is successful in its task, it adds a new attribute or analysis to the representation. SSF has many similarities with the blackboard.

The above flexibility is obtained by using two kinds of analyses:

- Constituent level analysis and
- Relational-structure level analysis

The former is used to store phrase level analysis and the latter for storing relations between separate phrases (which are not a part of the same phrase structure tree). Feature structures are used to store attribute-value pairs for a phrasal node. Attribute value pairs are also used to store relations, as will be seen later. Outputs of many other kinds of analysis, such as grammatical relations, TAM computation, case computation, dependency relations, word sense disambiguation etc. are stored using feature-structures.

Though the SSF format is fixed, it is extensible to handle new features. It also has a text representation, which makes it easy to read the output. The following example illustrates the SSF. For example, the following English sentence,

```
Children are watching some programmes on television in the house. --(1)
```

contains the following chunks (enclosed by double brackets),

```
((Children)) [[are  watching]] ((some  programmes))
((on television)) ((in the house))
```

All the chunks are noun phrases, except for one ('are watching') which is a verb group and is shown enclosed in square brackets. If we mark the part-of-speech tag for each word, we have the following:

```
((Children_NNS)) [[are_VBP watching_VBG]]
((some_DT programmes_NNS)) ((on_IN television_NN))
((in_IN the_DT house_NN))
```

The representation above is shown in SSF in Fig. 1 (below).

```
-------------------------------------------------
1       ((              NP
1.1     children        NNS
        ))

2       ((              VG
2.1     are             VBP
2.2     watching        VBG
        ))

3       ((              NP
3.1     some            DT
3.2     programmes      NNS
        ))

4       ((              PP
4.1     on              IN
```

```
4.1.1   ((              NP
4.1.2   television      NN
        ))
        ))

5       ((              PP
5.1     in              IN
5.2     ((              NP
5.2.1   the             DT
5.2.2   house           NN
        ))
        ))
-------------------------------------------------
```

Fig. 1: Towards Shakti Standard Format

As shown in Fig. 1, each line represents a word or a group (except for lines with '))' which only indicate the end of a group). For each group, the symbol used is '((' . Each word or group has 3 parts. The word or group is in the second part, with part of speech tag in the third part. The first part stores the tree address of each word or group, and is for human readability only.

The example below shows the SSF for the first noun phrase where feature information is also shown, as the fourth part on each line.

```
1       ((              NP      <fs root=child cat=np gend=m num=p pers=3>
1.1     children        NNS     <fs root=child cat=n gend=m num=p pers=3 case=0>
        ))
```

Some frequently occurring attributes (such as root, cat, gend, etc.) may be abbreviated using a special attribute called 'af' or abbreviated attributes, as follows:

```
1       ((              NP
1.1     children        NNS     <fs af='child,n,m,p,3,0,,' >
                                        |    | | | | |
                                        |    | | | |  \
                                      root   | | |pers |
                                             | | |    case
                                    category | number
                                             |
                                           gender
```

The field for each attribute is at a fixed position, and a comma is used as a separater. Thus, in case, no value is given for a particular attribute the field is left blank, e.g. last two fields in the above example.

The representation in SSF of sentence 1 with feature structures is given in Fig. 2 (abbreviated attribute 'af' is used).

```
-----------------------------------------------
1       ((              NP
1.1     children        NNS     <fs af=child,n,m,p,3,0,,>
        ))
2       ((              VG
2.1     are             VBP     <fs af=be,v,m,p,3,0,,>
```

```
2.2     watching        VBG     <fs af='watch,v,m,s,3,0,,' aspect=PROG>
        ))
3       ((              NP
3.1     some            DT      <fs af=some,det,m,s,3,0,,>
3.2     programmes      NNS     <fs af=programme,n,m,p,3,0,,>
        ))
4       ((              PP
4.1     on              IN      <fs af=on,p,m,s,3,0,,>
4.1.1   ((              NP
4.1.2   television      NN      <fs af=television,n,m,s,3,0,,>
        ))
        ))
5       ((              PP
5.1     in              IN      <fs af=in,p,m,s,3,0,,>
5.2     ((              NP
5.2.1   the             DT      <fs af=the,det,m,s,3,0,,>
5.2.2   house           NN      <fs af=house,n,m,s,3,0,,>
        ))
        ))
-----------------------------------------------
        Fig. 2: Shakti Standard Format
```

5 Relations between Chunks

Having introduced the basic SSF representation, we will now see how different relations are specified between chunks or groups. There are several types of relations that are used in Shakti, and new relations can be introduced as needed. Some of the major relations are:

1. Grammatical roles (sentential)
2. Dependency relations (karaka relations)
3. Theta relations
4. Semantic relations
5. Discourse level relations
6. Other relations

5.1 Grammatical Role

Grammatical role gives the syntactic relations (such as subject, object etc.) of the main verb (in the clause or the sentence). The grammatical arguments are marked to have the role. For example,

```
  He     gave     me     a book     in the garden     this morning.

((He)) ((gave)) ((me)) ((a book)) ((in the garden)) ((this morning)).
 subj            obj      obj2      prep__in          prep__0
```

Here, 'He' is the subject, and 'me' is the syntactic object of the verb 'gave', and 'book' is the second object (obj2) of the verb. Additionally, 'in the garden' has the preposition 'in', and 'this morning' occurs after the preposition phrase. Prepositions are also taken as grammatical role markers (vibhakti). All this is shown as follows:

```
he                  role=subj:give
gave    name=give
me                  role=obj:give
book                role=obj2:give
garden              role=prep__in:give
this_morning        role=prep__0:give
```

The above shows the grammatical roles by means of features. As stated earlier, a feature consists of attribute value pairs. The attribute 'role' stands for grammatical role, and its value consists of two parts : (i)the type of grammatial role and (ii) the verb with which the role holds. In the example above, the first word 'he' is related to 'give' by means of the grammatical role 'subj' which holds this role with the verb 'give'. This is represented as 'role=subj:give'.

These grammatical features (attribute and value) have the following format:

```
Format:  role = ROLE-R : NCHNK     (Ex. role='subj:give')
```

where, ROLE-R is the grammatical role, and NCHNK is the named chunk. Prepositions are incorporated as showing the actual preposition prefixed by 'prep__'.

5.2 Dependency Relations

Dependency relations give the relation between the heads and their modifiers at the next deeper level of analysis. They are more semantic than grammatical relations, and less than theta roles. They are in fact considered at the interface of syntax and semantics (where the grammatial relations are clearly syntactic and the theta roles are semantic). The dependency relations used here are based on the Panini's karaka theory (Bharati et al., 1995).

The dependency relations are specified between noun-noun chunks, verb-noun chunks, etc. We introduce them by means of an example, principally between noun-verb chunks. Here is an example sentence:

```
He gave me a book in the garden this morning.
```

We have already seen the grammatical relations between the chunks. Now, we show the dependency relations as well. 'He' is the karta or k1 and 'a book' is the karma or k2 of the verb 'gave'. 'Me' is the beneficiary or k4. (Refer to Bharati et al. (1995) for details of the relations k1, k2, and k4.) Additionally, 'garden' specifies the location (written as k7p), and 'this morning' specifies the time (written as k7t). All the karaka relations other than the k1, k2, and k4 are given as 'ky'. 'kx' stands for karaka relation, but the specific karaka relation is yet to be determined. The dependency relations are shown as follows:

```
he              role=subj:give       drel='k1:give'
gave  name=give
me              role=obj:give        drel='k4:give'
book            role=obj2:give       drel='k2:give'
garden          role=prep__in:give   drel='k7s:give'
this_morning    role=prep__0__r_adv:give     drel='k7t:give'
```

As in the case of grammatical relations, the dependency tree is represented by means of features. The attribute 'drel' stands for dependency relation, and its value consists of two parts: the type of dependency relation and the verb with which the relation holds. In the first case, 'he' is related to 'give' by means of the dependency relation 'k1' and is written as 'k1:give'.

Format of attribute and filler for dependency relations:

```
drel = DEP-R : NCHNK    (Ex. drel ='k1:give')
```

where, DEP-R is the dependency role name, and NCHNK is the named chunk or word.

There are four types of dependency relations: vmod, nmod, jjmod, and rbmod. Some examples illustrating them are given below:

5.2.1 Verb Modifier (vmod)

Vmod relation specifies that a given chunk modifies a verb group:

```
Ex. ((He)) ((lifted)) ((the red block)) ((easily)).
      n            v               n         rb (adverb)
  drel='vmod:lift'    drel='vmod:lift'  drel='vmod:lift'
```

Thus, the chunk 'the red block' is modifiying the verb 'lift' by the relation 'vmod'. 'vmod' relationship can be shown in a finer detail by splitting it into two relations: varg (argument) and vad (roughly, adjunct but not identical to linguistic adjunct). Thus the above sentence can be shown with the following relations:

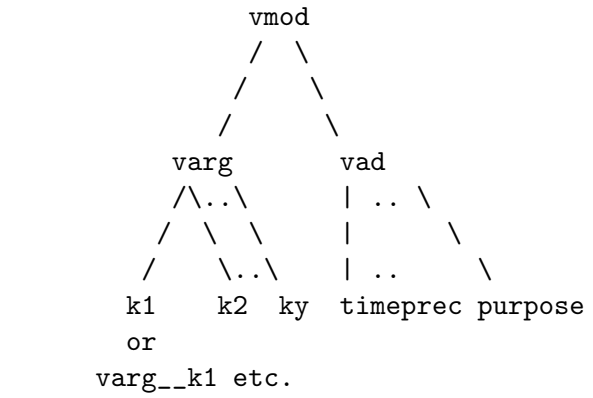
```
Ex. ((He)) ((lifted)) ((the red block)) ((easily)).
      n            v               n         rb (adverb)
drel='varg:lift'          drel='varg:lift'  drel='vad:lift'
```

These can be made even more detailed (using k1, k2, etc.):

```
Ex. ((He)) ((lifted)) ((the red block)) ((easily)).
      n            v               n         rb (adverb)
drel='varg__k1:lift'    drel='varg__k2:lift'  drel='vad:lift'
```

The representation is designed to be progressively more refined or detailed, because the NLP system can represent as much detail as it is able to obtain successfully.

The following tree shows progressively more fine representation for the vmod dependency relation:



Note that:

- *vmod* stands for either *varg* or *vad*,
- *varg* stands for any of the karakas (*k1, k2* etc.) with a more explicit way of writing: *varg__k1*,
- *kx* stands for any of the karakas
- *ky* stands for a karaka other than *k1, k2*.
- *vad* stands for relations of an element with a verb, wherein the element is not an argument of the verb. It would typically relate a verb with other versbs where the relation is that of purpose, temporal precedence, etc.

5.2.2 Noun Modifier (nmod)

'nmod' specifies that a given chunk is modifying a noun group. For example:

```
((He)) ((ate)) ((the tomatos))  ((in the salad))
  n       v           n                  n
                                 drel='nmod:tomato'
```

Clearly, 'in the salad' modifies the noun chunk 'the tomatoes' by the relation 'nmod'.

The relation 'nmod' can also be used, provided we are prepared:

```
          nmod
          / | \
         /  |  \
        /   |   \
    relc  relr  n_to_n
```

The relations 'relc' and 'relr' are used for relative clause, and reduced relative, respectively, explained in the next section (Sec. 5.3). 'nmod_n' is used when an NP or PP modifies another NP, as given in the above example ('tomatoes in the salad').

5.2.3 Adjectival Modifier (jjmod)

jjmod specifies that the chunk is modifying an adjective group. For example:

```
((This fruit))  is ((as sweet)) ((as sugar))
       n        v       adj            n
                                 drel='jjmod:sweet'
```

Here, 'as...as sugar' modifies 'sweet' by the relation jjmod. (Here, the traditional chunks are shown, rather than 'as X as sugar'.)

5.2.4 Adverbial Modifier (rbmod)

rbmod specifies that the chunk is modifying an adverb group. For example:

```
((He))  swam ((as easily)) ((as fish))
  n       v       adv            n
                          drel='rbmod:easily'
```

Here, 'as...as fish' modifies 'easily' by the relation rbmod. (Only the traditional chunks are shown, rather than 'as X as fish'.)

5.2.5 Coordination

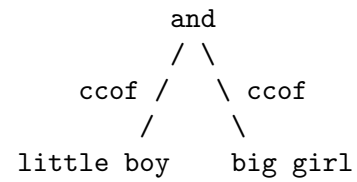
This section describes coordinating units formed by conjunct words such as 'and', 'or', 'but', 'if' etc.

Coordination Phrase

Coordination specifies a grouping of similar chunks, all of which behave as a single unit (for example, they have a similar relation with the related chunks). In the following sentence, the 'boy' chunk and the 'girl' chunk are grouped together, and are related to 'and' by means of 'ccof' relation forming the coordination phrase:

```
((The little boy)) and ((the big girl)) ((rode)) ((the bicycle)).
       n            cc        n            v          n
   ccof:and               ccof:and
```

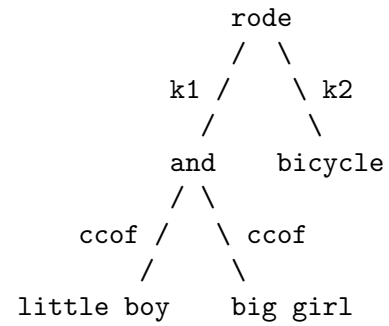
The relationship between 'and' and its parts are:



Thus, 'and' behaves like an operator, representing the coordination of the NPs (*the little boy* and *the big girl*), shown by the following:

```
the_little_boy              drel=ccof:and
and               name=and
the_big_girl                drel=ccof:and
```

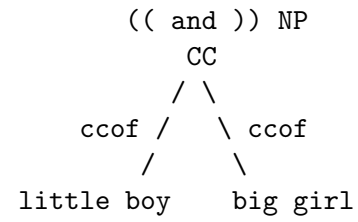
Now, the operator is the subject or k1 of the verb 'rode', shown by the following tree:



The tree indicates that the operator 'and' is the k1 (or karta) of the verb 'rode'. The tree is shown by the following representation:

```
the_little_boy         drel=ccof:and
and               name=and            role=subj:rode drel=k1:rode
the_big_girl           drel=ccof:and
rode              name=rode
the_bicycle                           role=obj:rode  drel=k2:rode
```

'And' as a word has its part-of-speech as 'cc', however, to keep the grammar uniform, its chunk is of type 'NP'.



shown in SSF as follows:

```
((     NP
and    CC          name=and
))
```

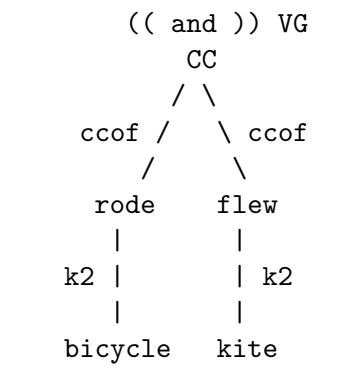
In other words, 'subject' or 'k1' of 'rode' continues to be of syntactic type 'NP'.

Coordination Clause

We now see an example of a coordination clause where two verbs are grouped:

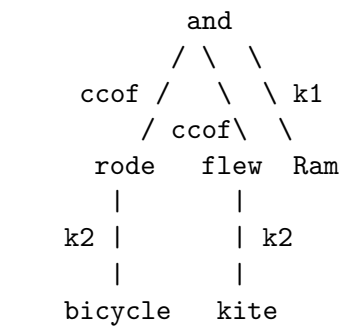
```
Ram rode a bicycle and flew a kite.
```

Its tree would be:



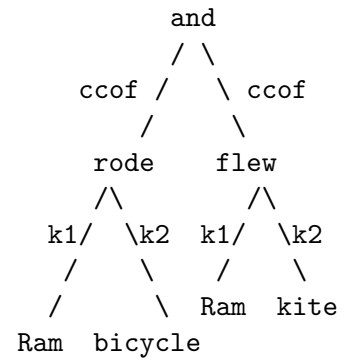
Note that the chunk tag of 'and' is now VG. Thus 'and' is truly an operator and gets the tag of its 'ccof' parts.

The tree for the sentence now looks as follows:



Note that an important property needs to be mentioned at this stage. 'and' has relationships with three chunks: rode, flew and Ram. The first two have a "closer" relationship, than 'Ram', because they are as good as a part of 'and' chunk. This is a property of 'ccof' relation type.

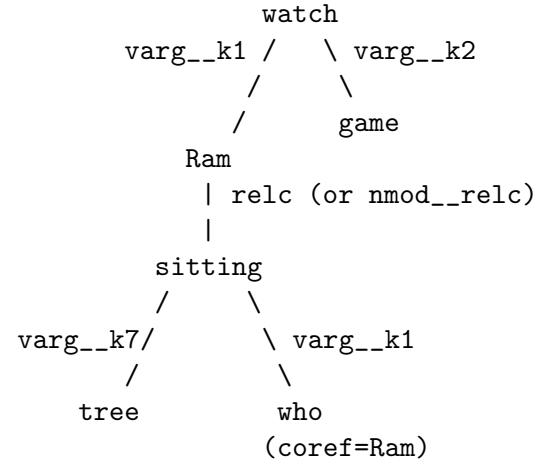
Inference can produce the following structure out of the above, but a discussion on it is beyond the scope of this paper:



5.3 Relative Clause

In case of the relative clause, its main verb modifies the noun chunk to which the clause refers to.

```
e.g. Ram ((who is sitting on the tree)) is watching the game.
```



Representation of the above tree is shown below, with first four lines showing the tree centred on 'Ram'.

```
Ram             name=ram                        drel=varg__k1:watch
who                         drel=varg__k1:sit, coref=Ram
is_sitting      name=sit    drel=relc:Ram
on_the_tree                 drel=varg__k7:sit

is_watching     name=watch
the_game                                        drel=k2:watch
```

'coref' specifies that the chunk is coreferent with the other chunk, in example above, 'who' is coreferent with 'Ram'.

5.3.1 Embedded Relative Clause

Now we look at a relative clause with nested clauses. Consider an example sentence and the drel tree for its relative clause:

```
e.g. Ram ((who Mohan said is sitting on the tree)) is watching the game.

          Ram
           | relc (or nmod__relc)
           |
          said
varg__k1 /   \ varg__k2
        /     \
     Mohan   sitting
               / \
      varg__k7/   \ varg__k1
             /     \
          tree    who
                  (coref=Ram)
```

Note that the above representations assume that for the wh-phrase ('who' above), its relationship with the verb, and the type of relationship can be identified. In case the verb is identified but the type of relationship is not identified, there would be 'lnk' relationship between the wh-phrase and the verb ('who' with 'sitting' in the above example):

```
who    drel=lnk:sitting
```

If the verb is also not identified, a link with the main verb in the relative clause is created 'comp'. For the above example, it would be 'who' having the following:

```
who    drel=comp:said
```

5.3.2 Reduced Relative

For reduced relative clause, here is an example sentence:

```
Ram ((sitting on the tree)) is watching the game.
```

Its drel tree looks like:

```
Ram
 | nmod__relr
 |
sitting
 | varg__k7
 |
tree
```

If the relationship between 'Ram' and 'who' is identified in more detail, one would get 'nmod__relr__k1-INV' indicating that there is inverse (INV) karta karaka (k1) relationship.

In case the machine fails to identify the reduced relative, and simply marks it as a participial verb modifying a noun, it will show the relation as: 'vkr' between 'Ram' and 'sitting' (instead of 'nmod__relr').

```
Ram
 | nmod__vkr
 |
```

```
sitting
 | varg__k7
 |
tree
```

5.4 Discourse Relations

Currently there is only one discourse relation in the system, namely, coreference. It is produced by the syntactic module, currently. In due course, the pronoun co-reference module would also be included.

Coref specifies that the current chunk is coreferent with the other chunk.

5.5 Some Other Relations

5.5.1 Local Chunk Relations

Besides the above major relations, there are also local relations. They typically show relationships between chunks in terms of their relative positions. It may also be used to make some intra-chunk relations explicit (if necessary), as follows:

```
Format: cpos = REL : NCHNK
```

Attribute used is 'cpos' (for chunk position), where REL stands for 'pre' or 'post', and NCHNK stands for a chunk name or word name.

1. Pre

'pre' says that the word/chunk occurs before a noun or a verb. Example for noun: "iron pump"

```
   ((iron))         ((pump))
 cpos=pre:pump    name=pump
```

Example for verb: "easily read"

```
 ((easily))       ((read))
  cpos=pre:read   name=read
```

(Note that because these are local relations, they typically specify a relation between two chunks. When we come to grammatical relations, they become sentence or verb centred.)

2. Post

'post' says that the word/chunk occurs after a noun or a verb. Ex. for noun: "girls, smart and young"

```
 ((girls))      ((smart and young))
  name=girl        cpos=post:girl
```

5.5.2 Sentence Positions(spos)

Apart from the roles mentioned above, certain sentence positions are also marked. Spos attribute indicates sentential position. For example, prepositional phrases which occur before the subject have the value (of 'spos' attribute as) 'init', and those which occur after obj or obj2 position have the value 'finit'.

1. init – Sentence initial position (before the subject). For example,

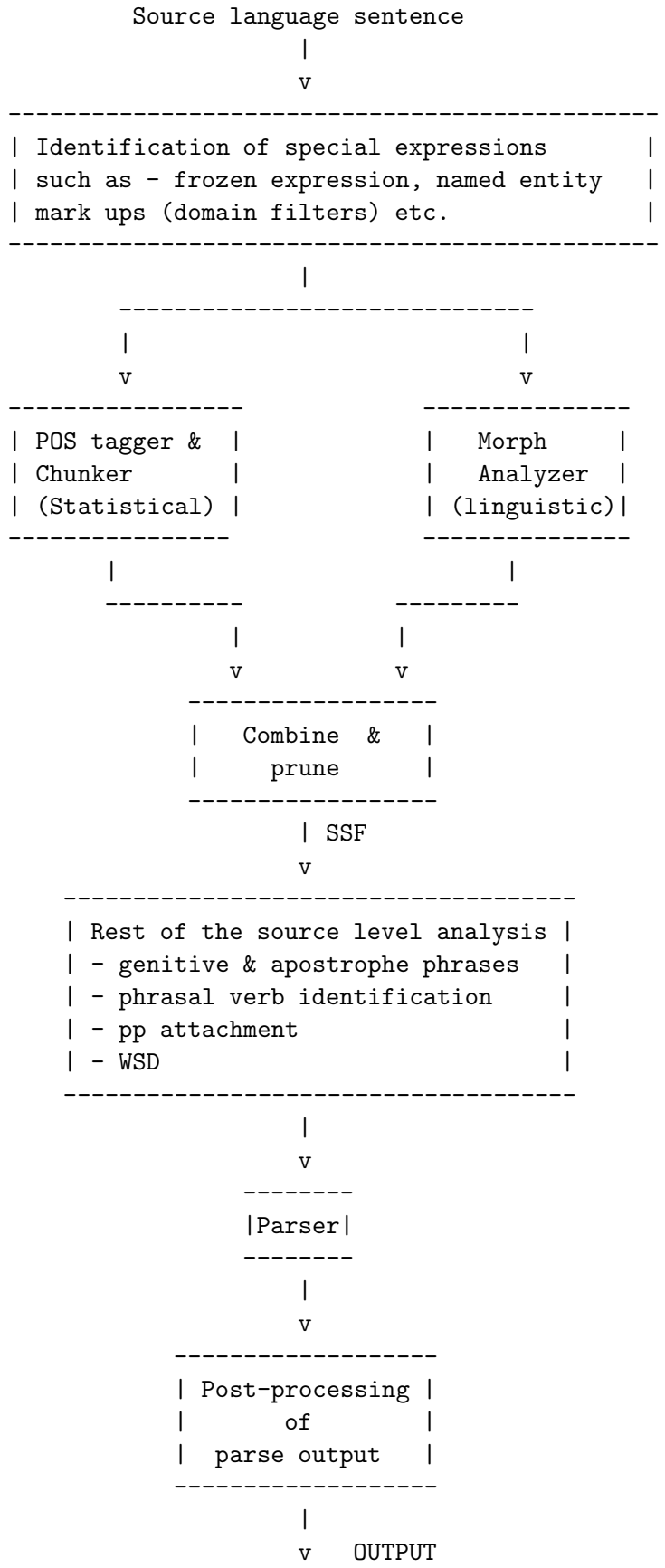
```
((In the morning of Sept 8th, 2003)), we left for Farakkabad.
       spos=init:left
```

2. finit – Sentence later position (after the object). For example,

```
Only those people were left behind ((who had not arrived on time))
                                    spos=finit:left
```

6 System Block Diagram

A indicative system level block diagram is given below. The most important thing is that the traditional NLP systems were monolithic, whereas Shakti architecture is modular, wherein the tasks are broken up in small modules, as indicated below.


Source language sentence
Identification of special expressions
such as - frozen expression, named entity
mark ups (domain filters) etc.
POS tagger &
Chunker
(Statistical)
Morph
Analyzer
(linguistic)
Combine &
prune
SSF
Rest of the source level analysis
- genitive & apostrophe phrases
- phrasal verb identification
- pp attachment
- WSD
Parser
Post-processing
of
parse output
OUTPUT

7 Conclusions

We have presented a design of an NLP system which allows us to put together different modules, using a common representation for storing natural language analysis. With a common representation, all the modules have to operate on a single data structure. Modules typically add their analysis to the common structure (or to the shared memory structure), and when a module fails to produce the correct analysis it still might leave partial information.

Concrete details have been shown how different phenomena in language can be analyzed and represented using SSF.

References

Bharati, Akshar, Vineet Chaitanya and Rajeev Sangal, "Natural Language Processing: A Paninian Perspective", Prentice-Hall of India, New Delhi, 1995. (URL://ltrc.iiit.ac.in)