PBML

The Prague Bulletin of Mathematical Linguistics
NUMBER 103 APRIL 2015 139–159

A Python-based Interface for Wide Coverage Lexicalized Tree-adjoining Grammars

Ziqi Wang, Haotian Zhang, Anoop Sarkar

School of Computing Science, Simon Fraser University

Abstract

This paper describes the design and implementation of a Python-based interface for wide coverage Lexicalized Tree-adjoining Grammars. The grammars are part of the XTAG Grammar project at the University of Pennsylvania, which were hand-written and semi-automatically curated to parse real-world corpora. We provide an interface to the wide coverage English and Korean XTAG grammars. Each XTAG grammar is lexicalized, which means at least one word selects a tree fragment (called an *elementary tree* or *etree*). Derivations for sentences are built by combining etrees using substitution (replacement of a tree node with an etree at the frontier of another etree) and adjunction (replacement of an internal tree node in an etree by another etree). Each etree is associated with a feature structure representing constraints on substitution and adjunction. Feature structures are combined using unification during the combination of etrees. We plan to integrate our toolkit for XTAG grammars into the Python-based Natural Language Toolkit (NLTK: nltk.org). We have provided an API capable of searching the lexicalized etrees for a given word or multiple words, searching for a etree by name or function, display the lexicalized etrees to the user using a graphical view, display the feature structure associated with each tree node in an etree, hide or highlight features based on a regular expression, and browsing the entire tree database for each XTAG grammar.

1. Introduction

Tree-Adjoining Grammars (TAG) represents a tree manipulating system. TAG was first proposed in (Joshi et al., 1975) as grammatical formalism that extends context-free grammars (CFG) with an extended domain of locality (namely a tree structure

Corresponding author: anoop@sfu.ca
Cite as: Ziqi Wang, Haotian Zhang, Anoop Sarkar. A Python-based Interface for Wide Coverage Lexicalized Tree-adjoining Grammars. The Prague Bulletin of Mathematical Linguistics No. 103, 2015, pp. 139–159.
doi: 10.1515/pralin-2015-0008.

rather than a CFG rule). The main reasons to consider TAG as a suitable formalism for the analysis of the syntax of natural language are:

- TAG can be viewed as a system that can be used for lexicalization of a CFG. Unlike the Griebach normal form for CFGs, a lexicalized TAG can preserve the tree set of the original CFG (Schabes et al., 1988; Schabes and Waters, 1995). Most linguistic applications of TAG use a lexicalized TAG (LTAG).
- Like CFGs, TAGs are parsable in polynomial time.
- TAG can generate crossing dependencies which are widespread in natural language, while CFGs can only generate nested dependencies.

A comprehensive introduction to TAG, both the formalism and the linguistic application of TAG appears in (Joshi and Schabes, 1997). Large scale LTAG grammars have been constructed by hand for English as part of the XTAG project at the University of Pennsylvania (UPenn) (XTAG Group, 2001) and French (at the TALANA group, University of Paris 7, France) and somewhat smaller ones for German (at DFKI, Saarbrücken, Germany), Korean (at UPenn), Chinese (at UPenn), and Hindi (at CDAC, Pune, India). The earliest stochastic variant of TAG was proposed by (Resnik, 1992; Schabes, 1992). LTAG grammars have been extracted from annotated corpora (Xia et al., 2001; Xia, 2001; Chiang, 2000; Chen and Vijay-Shanker, 2000), which in turn have been used for lexicalized statistical parsing (Chiang, 2000; Sarkar, 2001). In this paper, our focus will be on the hand-written large scale LTAG grammars developed as part of the XTAG project at the University of Pennsylvania (XTAG Group, 2001).

The goal of this work is to provide a modern programming interface to the large scale LTAG grammars developed as part of the XTAG project at the University of Pennsylvania. We provide a conversion of the grammar files for English and Korean wide coverage LTAG grammars. To this end, we describe in this paper our Python-based API for viewing and manipulating these LTAG grammars. We plan to incorporate our modules into the Python-based Natural Language Toolkit (NLTK) project so that anybody who downloads the NLTK project can have access to the linguistic annotations that are part of the LTAG grammars developed during the XTAG project. All the data is also available through online repositories and packaged for distribution and convenient use by our codebase. This is particularly useful as the software that was originally developed for manipulating these grammars by the XTAG project is no longer being maintained.

2. Tree-adjoining grammars

In this section we summarize the representation used in the XTAG grammars. The material here is a summary of the extended description provided in the XTAG technical report describing the English grammar (XTAG Group, 2001).

Tree-adjoining grammar (TAG) is a formal tree rewriting system. TAG and Lexicalized Tree-Adjoining Grammar (LTAG) have been extensively studied both with respect to their formal properties and to their linguistic relevance. TAG and LTAG are formally equivalent, however, from the linguistic perspective LTAG is the system we will be concerned with in this paper. We will often use these terms TAG and LTAG interchangeably.

The motivations for the study of LTAG are both linguistic and formal. The elementary objects manipulated by LTAG are structured objects (trees or directed acyclic graphs) and not strings. Using structured objects as the elementary objects of the formal system, it is possible to construct formalisms whose properties relate directly to the study of strong generative capacity (i.e., structural descriptions), which is more relevant to the linguistic descriptions than the weak generative capacity (sets of strings).

Rather than giving formal definitions for LTAG and derivations in LTAG we will give a simple example to illustrate some key aspects of LTAG. We show some elementary trees of a toy LTAG grammar of English. Figure 1 shows two elementary trees for a verb such as *likes*. The tree α_1 is anchored on *likes* and encapsulates the two arguments of the verb. The tree α_2 corresponds to the object extraction construction. Since we need to encapsulate all the arguments of the verb in each elementary tree for *likes*, for the object extraction construction, for example, we need to make the elementary tree associated with *likes* large enough so that the extracted argument is in the same elementary domain. Thus, in principle, for each 'minimal' construction in which *likes* can appear (for example, subject extraction, topicalization, subject relative, object relative, passive, etc.) there will be an elementary tree associated with that construction. By 'minimal' we mean when all recursion has been factored away. This factoring of recursion away from the domain over which the dependencies have to be specified is a crucial aspect of LTAGs as they are used in linguistic descriptions. This factoring allows all dependencies to be localized in the elementary domains. In this sense, there will, therefore, be no long distance dependencies as such. They will all be local and will become long distance on account of the composition operations, especially adjoining.

Figure 2 shows some additional trees. Trees α_3, α_4, and α_5 are initial trees and trees β_1 and β_2 are auxiliary trees with foot nodes marked with *. A derivation using the trees in Figure 1 and Figure 2 is shown in Figure 3. The trees for *who* and *Harry* are substituted in the tree for *likes* at the respective *NP* nodes, the tree for *Bill* is substituted in the tree for *think* at the *NP* node, the tree for *does* is adjoined to the root node of the tree for *think* tree (adjoining at the root node is a special case of adjoining), and finally the derived auxiliary tree (after adjoining β_2 to β_1) is adjoined to the indicated interior S node of the tree α_2. This derivation results in the **derived tree** for *who does Bill think Harry likes* as shown in Figure 4. Note that the dependency between *who* and the complement *NP* in α_2 (local to that tree) has been stretched in the derived tree in Figure 4. This tree is the conventional tree associated with the sentence.

However, in LTAG there is also a derivation tree, the tree that records the history of composition of the elementary trees associated with the lexical items in the sentence. This derivation tree is shown in Figure 5. The nodes of the tree are labeled by the

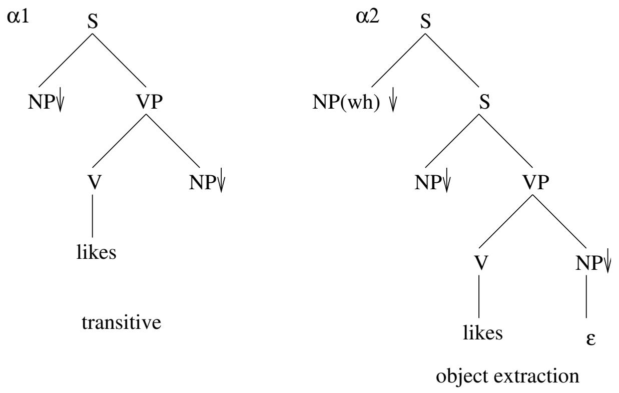


Figure 1. LTAG: Elementary trees for likes. The same predicate-argument structure can be syntactically realized in different ways. All the different syntactic transformations produce a set of elementary trees for each predicate. This set is grouped together into a tree family *for each type of predicate.*

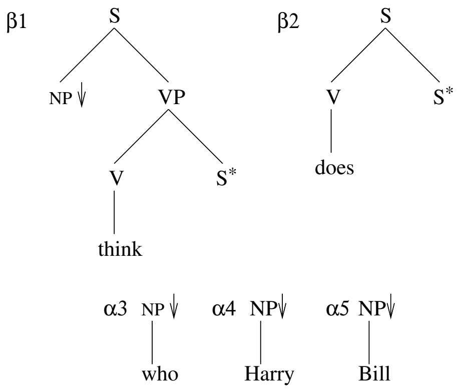


Figure 2. LTAG: Sample elementary trees

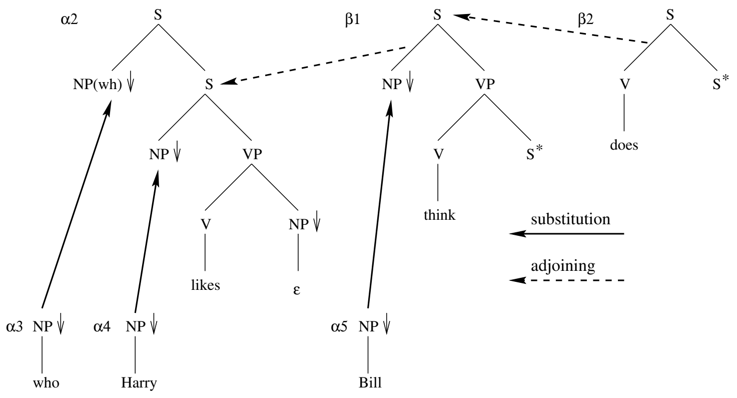


Figure 3. LTAG derivation for who does Bill think Harry likes

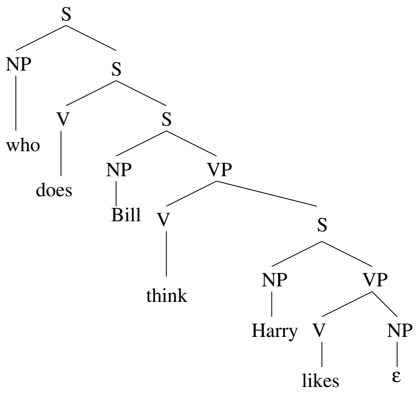


Figure 4. LTAG derived tree for who does Bill think Harry likes

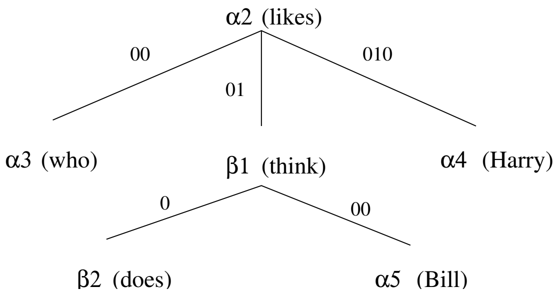


Figure 5. LTAG derivation tree

tree labels such as α_2 together with the lexical anchor.[1] The derivation tree is the primary derivation structure for LTAG: the derived tree can be directly created using the information in the derivation tree.

Large scale wide coverage grammars have been built using LTAG, the XTAG English grammar (an LTAG grammar and lexicon for English) being the largest so far (for further details see (XTAG Group, 2001).

2.1. Feature Structures and TAG

A feature structure is a set of key-value pairs (i.e. features), in which the value could be of basic type or other feature structure. Two or more keys could share the same value, and value could be shared among keys. This property facilitates interactions between tree nodes. The main operation defined for feature structure is unification, through which two feature structures are merged into one. Unification could fail if there is conflicting feature. A feature structure based TAG is a TAG with feature structures attached to tree nodes.

In the XTAG grammars, each node in each LTAG tree is decorated with two feature structures (top and bottom feature structures), in contrast to the CFG based feature structure grammars. This is necessary because adjoining can augment a tree internally, while in a CFG based grammar or even in a tree substitution grammar a tree can be augmented only at the frontier. It is possible to define adjoining and substitution (as it is done in the XTAG system) in terms of appropriate unifications of the top and bottom feature structures.

When doing substitution, the top feature of the new node is the result of unification of the root node and substitution node, while the bottom feature of the new node is

[1]The derivation trees of LTAG have a close relationship to the notion of dependency trees, although there are some crucial differences; however, the semantic dependencies are the same. See (Rambow and Joshi, 1995) for more details.

the bottom feature of the root node in substituting tree. See Figure 6 for a schematic view of how unification is done during the substitution operation in LTAG.

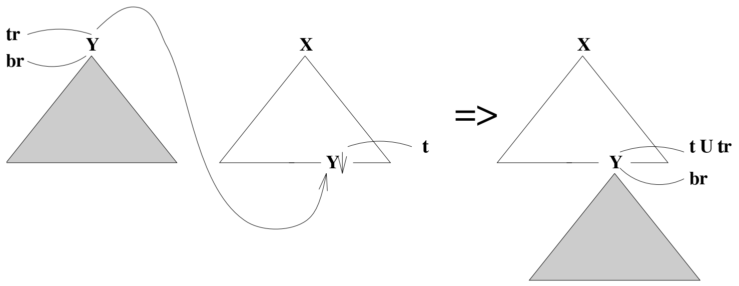


Figure 6. Substitution in feature-based LTAG

Adjunction is more complicated: after the adjunction node has been split into two, the top feature is now attached with the top half, and the bottom feature is attached with another half. In addition, in the resulting tree produced by adjunction, the top feature attached with the root of the auxiliary tree is then unified with the top feature of the top half, and similarly, the bottom feature attached with the adjunction node of the auxiliary tree is unified with the bottom feature of another half. See Figure 7 for a schematic view of how unification is done during the adjunction operation in LTAG.

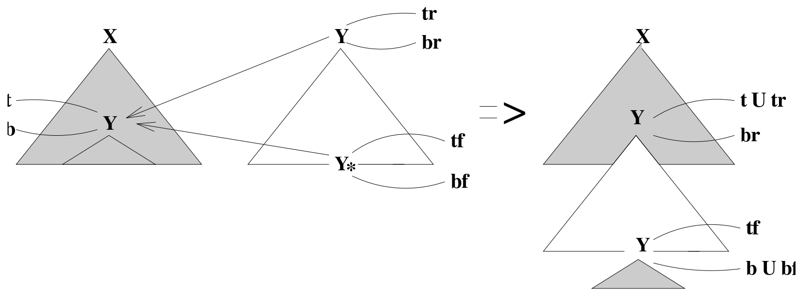


Figure 7. Adjunction in feature-based LTAG

Constraints on substitution and adjoining are modelled via these feature structures (Vijay-Shanker, 1987) (except for the null adjunction constraint or NA constraint which rules out adjunction at a node).

3. Structure of the XTAG Grammar

Each XTAG grammar (both the English and Korean grammar) is a large lexicalized Tree Adjoining Grammar. The grammar itself is represented in machine readable format in a series of files, each of which represents a different module in the grammar. In this section, we describe each module.

3.1. Trees and Tree Families

The first important concept in the XTAG grammar is the notion of a tree family. As seen in Figure 1 each predicate lexicalizes many different elementary trees where the predicate-argument structure is the same but the syntactic construction is different. All of the different syntactic realizations of a predicate are grouped into a *tree family*. In XTAG, the name of the tree family encapsulates the predicate-argument structure. For instance, the tree family name `Tnx0Vplnx1` refers to all the syntactic variations of a single predicate argument structure where `nx0` and `nx1` represent the subject and object NP arguments of the predicate respectively. The predicate is a verb represented by the `V` in the name and `pl` stands for particle. Thus, `Tnx0Vplnx1` is a tree family associated with a multi-word predicate such as *walk off*. All the trees in the tree family `Tnx0Vplnx1` are lexicalized by the anchor words *walk off* including trees for passives, wh-movement, relative clauses, etc.

Function words or non-predicates are assigned trees as well, and so there are *tree files* as well in the XTAG grammar where words lexicalize the individual trees in these tree files rather than all the trees at once.

In tree files, trees are defined as objects, encapsulating the skeleton of elementary tree, tree name, node information, feature structure for nodes, constraints, comments, and other GUI-related options, such as switches for drawing.

Tree skeletons are the frameworks of un-lexicalized elementary objects, and they are essentially DAGs (directed acyclic graphs), with tree nodes being graph nodes, and internal connections being edges. Tree nodes are again objects with at least three members: node label, node type, and node constraint. Node label is the symbol it represents, which must be a non-terminal, and appropriate measures must be taken to avoid naming collision (see below). Node type annotates whether a node on tree frontier is a substitution node, adjunction node (for internal nodes this field is empty) or head node (where the lexical item, i.e. anchor, is attached). We follow the convention that for substitution node there is a down arrow, for adjunction node there is an asterisk, and for head node, there is a diamond when displaying trees graphically. Node constraints rules out combinations that are not qualified, and should be blank if no constraint is applicable.

In order to prevent node label collision (two nodes on the same tree having identical label), which can happen in elementary trees, the notion of node label is extended to have two parts. The first part is exactly the non-terminal symbol it represents, and the second part is appended to differentiate this node from others. Any naming scheme that guarantees uniqueness is acceptable, however, according to our convention, root nodes are given suffix `.r`, and foot nodes are given suffix `.f`. Besides, all other nodes that have a common non-terminal symbol are given an increasing numeric suffix. For example, if we have four NP nodes in an auxiliary tree, one of them being the root node, another being the foot node, and the rest two are generic nodes, then their label should be `NP.r`, `NP.f`, `NP`, `NP.1` respectively.

There are two types of elementary tree, initial tree and auxiliary tree, and in order to distinguish them, trees with a name that starts with an "alpha" α denote initial trees, and those whose name starts with a "beta" β denote auxiliary trees. This is just a naming convention.

3.2. Feature structures

XTAG is organized such that feature structures are specified in three different components of the grammar: a *Tree* database defines feature structures attached to tree *families*; a *Syn* database defines feature structures attached to lexically anchored trees; and a *Morph* database defines feature structures attached to (possibly inflected) lexical entries.

As an example, consider the verb *seems* This verb can anchor several trees, among which are trees of auxiliary verbs, such as the tree βVvx, depicted in figure 8. This tree, which is common to all auxiliary verbs, is associated with the feature structure descriptions listed in Figure 8 (independently of the word that happens to anchor it).[2]

When the tree βVvx is anchored by *seems*, the lexicon specifies additional constraints on the feature structures in this tree:

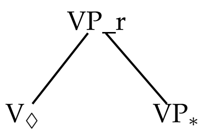
```
seem betaVvx VP.b:<mode>=inf/nom,
             V.b:<mainv> = +
```

Finally, since *seems* is an inflected form, the morphological database specifies more constraints on the node that this word instantiates, as shown in figure 9.

The full set of feature structures that are associated with the lexicalized tree anchored by *seems* is the unification of these three sets of path equations.

The atomic values in feature structures fall into three categories. The first two are generic strings and booleans (+ or -). A third type of disjunction is also used, usually represented as `a/b/c`, which means "a or b or c" (it represents a disjunctive relation).

[2]We use "handles" such as `V.b` or `NP.t` to refer to the feature structures being specified. Each node in a tree is associated with two feature structures, 'top' (.t) and 'bottom' (.b). Angular brackets delimit feature paths, and slashes denote disjunctive (atomic) values.



```
V.t:<agr> = VP_r.b:<agr>
V.t:<assign-case> = VP_r.b:<assign-case>
V.t:<assign-comp> = VP_r.b:<assign-comp>
V.t:<displ-const set1> = VP_r.b:<displ-const set1>
V.t:<mainv> = VP_r.b:<mainv>
V.t:<mode> = VP_r.b:<mode>
V.t:<neg> = VP_r.b:<neg>
V.t:<tense> = VP_r.b:<tense>
VP.t:<assign-comp> = ecm
VP.t:<compar> = -
VP.t:<displ-const set1> = -
VP_r.b:<compar> = -
VP_r.b:<conditional> = VP.t:<conditional>
VP_r.b:<perfect> = VP.t:<perfect>
VP_r.b:<progressive> = VP.t:<progressive>
```

Figure 8. An example tree and its associated feature structure descriptions

```
seems   seem  V   <agr pers> = 3,
                  <agr num> = sing,
                  <agr 3rdsing> = +,
                  <mode> = ind,
                  <tense> = pres,
                  <assign-comp> = ind_nil/that/rel/if/whether,
                  <assign-case> = nom
```

Figure 9. The morphological database entry for seems

3.3. Morphology

The morphology information in XTAG establishes mappings from inflected words to a list of tuples. The morphology file contains a mapping from inflected words to the root word, part of speech (POS) and inflectional information. If there are multiple records returned, then they are organized as a list. Figure 10 contains an example of a morphology entry in XTAG. The leftmost word of the first line, *facile,* is the index of record. The other two columns are the stem form and part of speech (POS) respectively.

Inflected Word	Stem	POS	Inflectional Information
facile	facile	A	
cajoles	cajole	V	3sg PRES
chills	chill	N	3pl
chills	chill	V	3sg PRES

Figure 10. Morphology Example

Index	Sense 1	Sense 2	Sense 3	Sense 4
crabs	crab N 3pl	crabs N 3sg	crab V PPART STR	crab V 3sg PRES

Figure 11. Multiple Records Example

3.4. Syntax and Feature Templates

Syntax information in XTAG consist of two components: the syntactic mapping from stems to trees and the list of feature templates.

The syntactic database stores mappings from a stem to a list of structures. It is queried using an uninflected word as index. The query returns a list of syntactic items. Each syntactic item is recognized as a structure with header and body, which contains information to lexicalize TAG trees.

Each entry has a header that contains ENTRY and POS (part of speech) keys. ENTRY and POS must appear in pairs, the values of which are inflected word and POS tag respectively. Duplicated pairs of ENTRY and POS are allowed, and this happens when the identifier is a phrase rather than a single word. An example of the phrase "walk off" is presented below.
«INDEX»walk«ENTRY»walk«POS»V«ENTRY»off«POS»PL«FAMILY»Tnx0Vplnx1

After the header the rest of the entry contains either TREES or FAMILY and FEATURES. These three are all optional, but in order to define a non-trivial syntactic item, at least one of TREES and FAMILY must exist. TREES selects a group of trees by the file name of one or more generic tree files, while FAMILY selects a group of trees by the file name of one or more family files. Once the groups are selected, all trees in these groups will be lexicalized. In the example above, the value of FAMILY is "Tnx0Vplnx1"; therefore, all trees defined in that file will be lexicalized using "walk" and "off".

The FEATURES key in the body specifies feature structures for lexicalization. Please note that features are also defined in morphology files. The reason of scattering them in two separate files is that features in morphology file are associated with inflectional information, while those in syntactic database capture lexical idiosyncrasies. Features from both files should be applied to the trees.

@1sg	@1st, @sg, <agr 3rdsing> = -!
#_AWH+	A.b:<wh> = +!

Figure 12. Feature Template Example

In a feature template, feature identifiers from the morphology and from syntax should not be mixed up. Although they coexist in the same file, appropriate measures must be taken to help distinguish one from another. We extracted two lines from our implementation of feature template as an example.

Lines starting with a "@" is recognized as inflectional (morphological) features, while lines starting with a "#" is recognized as syntactic database features. Besides, "!" is used to terminate one line. In addition, shorthand is allowed when defining feature entries. Reference to already defined feature structures would expand them in-place as part of the definition.

3.5. Configuration files

Configuration files consist of metadata files and list of trees, families and other information about the grammar. Also included here are properties (represented as feature structures) expected for every root node in a derivation tree (for instance: the main clause should carry tense, e.g. *the toy exploded* which rules out untensed main clauses such as ∗*the toy to explode* — this requirement is implemented as a feature structure that will be unified with the root node of the elementary tree which is the root of the derivation tree for an input sentence).

4. A Client-Server Architecture

Our XTAG grammar viewer is divided into two components: the backend and frontend. The backend is responsible for data processing, such as finding TAG trees for a given word; while the frontend implements presentation and user interaction, such as drawing a TAG tree on the frame buffer with feature structures. We have designed an API that bridges the frontend and backend to make things more modular, e.g. one could use the backend API to iterate through the grammar and one could use the frontend API to draw elementary trees that are not in the XTAG grammar.

The server and client could be running on different machines, communicating using network protocols, such as remote procedure call (RPC), or even web protocol like HTTP. In the following, we use the term "backend" and "server", "frontend" and "client" interchangeably.

Interface	Description
unpack_data()	Reads XTAG dataset, and returns memory objects
dump_binary()	Dumps memory objects onto file system
load_binary()	Loads memory objects from file system

Figure 13. Data Preprocessing Interface Description

5. Backend

In this section, we present the backend of grammar viewer. As previous section has indicated, the backend serves as a data processor, and handles requests from the client.

5.1. Data Preprocessing

Data preprocessing or initialization, happens only once on startup. Data preprocessing takes the raw source grammar files and creates machine readable data structures on which the grammar viewer operates. In the reader step unpack_data() should take the path or descriptor of the configuration file, extracting global user path, relative file paths, as well as file names, and concatenate the global user path with each relative file path and file name respectively (possibly we also need to append a suffix using file type information). Then it reads contents of each file, sending them for unpacking into memory, and returns these data structures. Since there is critical information contained in the configuration file, it should be unpacked and included as well.

Developers can dump a binary image of unpacked memory objects onto the file system after loading for the first time, and only use that binary dump for later. This reduces processing time and file size. Interfaces for this purpose are defined as dump_binary() and load_binary() respectively.

5.2. Forward Searching

Forward searching, also known as tree selection (please note that we use the term “tree selection” to refer to using tree file name to include all TAG trees in that file in Section 3, “Tree Files” Section), is the process of selecting TAG trees given a word or phrase, and then to lexicalize the selected trees with information about the word or phrase. This process involves many individual steps, and could be broken down into smaller independent steps:

After receiving user input from the client, which is usually a string consisting of one or more words, the first step is to split the string into tokens, and each token is an inflected word. These tokens are then passed to a series of procedures, including morphological analyzer, syntactic analyzer, feature manager and tree manager. Eventually the backend returns a list of TAG trees which are lexicalized by the tokens with feature structures attached to tree nodes.

The morphological analyzer accepts an inflected word as argument, and returns a set of its possible stem, POS tag, and morphological feature structures. This is accomplished by querying the morphological database. This database, as described in Section 3, is designed to be indexed using an inflected word, and the result is a list of records. Each record is a three tuple of the stem (root word, we use these two interchangeably), POS tag and feature structures. These records are then returned as a list. This procedure is defined as `word_to_morph()`.

The syntactic analyzer takes output from the morphological analyzer, and if there are multiple words (i.e. the lexical entry is a phrase), then we need to collect them using separate calls to the morphological analyzer into an aggregation, and then return the trees for the aggregated set of words. The syntactic analyzer extracts syntactic information from the syntactic database. As we have discussed in Section 4, the syntactic database is indexed using the root word from previous step, and the query returns a set of syntactic items. Each item has a header acting as an identifier, which is a stream of ENTRY and POS combinations, and a body that contains at least one of TREES, FAMILY and FEATURES. The syntactic analyzer combines morphological information with syntactic information by iterating through all morphological tuples, querying the syntactic database using the root word (if we are handling phrases, then only use the first one), and examine the result. Only those syntactic items whose identifier matches the one constructed using the root word and POS in the parameter is reserved; others are discarded, because their morphological and syntactical information do not conform. Once the syntactic items have been determined, the function returns with tree files, family files, and features extracted from those items. This procedure is defined as `morph_to_syntax()`.

The feature manager manages feature structures. This component would return a feature structure when called with a feature identifier. To avoid instantiating every feature structure objects during initialization, feature manager could be implemented with a cache that stores recently used feature structures, only duplicating and returning them when requested, and all other features remains un-instantiated until they are requested for the first time. Such a parse-on-demand manner would moderately save initialization time, while introducing slight performance harm later. The interface to access feature structures is defined as `get_feature_struct()`.

The tree manager, similar to feature manager, manages trees. However, it maintains tree database at different levels. If trees are requested by tree family name then a list of trees in that tree family is returned. The request could also be for all trees in a file that contains related trees but without the same predicate argument structure (files that contain all the adverb trees, etc.). If the word does not exist in the tree manager a default set of trees is returned. The trees can also be accessed by tree names,

Interface	Description
`word_to_morph()`	Given one token, returns its morphological records
`morph_to_syntax()`	Given morphological records of user inputs, returns their syntactical items
`get_feature_struct()`	Given feature identifier, returns feature structure object
`get_tag_tree()`	Given tree file name or tree name, returns tree object
`tree_lexicalization()`	Given morphological records, syntactic items, tree set and feature set, returns lexicalized TAG trees with feature structure

Figure 14. Forward Searching Interface Description

and in this case exactly one tree is returned if it is found. The interface for acquiring trees is defined as `get_tag_tree()`.

To lookup a particular lexicalized tree we search using a word or group of words. We then access the morphological records, syntactic items, tree set, and two feature sets (one from the morphological records, and another from the syntactic database) using the dataset. Still we need a procedure to combine them, producing a lexicalized TAG tree. This process is exactly what was called tree lexicalization in Section 2. Lexicalization is done in two steps.

The first step is to attach word(s) to TAG trees at the anchor nodes. Trees selected by the syntactic database possess one or more anchor nodes (represented by a diamond in the graphical view), and the number of anchor nodes is exactly the same as the words provided by the user. Tokens are attached to anchor nodes from left to right, i.e. the order they appear in trees is the same the order they are in user input.

The second step is to unify any feature structures that are to be merged into the lexicalized tree. Since we are dealing with both morphological features and syntactic features, they should be handled separately. Morphological features only carry inflectional information; therefore, they are unified into the bottom features of the anchor nodes respectively. In contrast, syntactic features capture lexical idiosyncrasies, and so they could be attached to any tree nodes. After this step, the result is a list of lexicalized trees with feature structures decorating each node in each elementary tree.

The procedure for final lexicalization using morphological and syntactical information is defined as `tree_lexicalization()`. And after this function returns, the data is returned to the client.

Interface	Description
`construct_tree_to_word_db()`	Constructs tree to words relations using syntactic database
`tree_to_word()`	Given TAG tree, returns words or phrases that could be used to lexicalized the tree

Figure 15. Backward Searching Interface Description

5.3. Backward Searching

Backward searching is invoked with an un-lexicalized TAG tree, and it returns words or phrases that could be used to lexicalize this tree. The data structure for backward searching is another mapping relation, which is constructed using the dataset in Section 3.

In essence, backward searching requires a database indexed by tree names, and the query returns a list of words and phrases. The mapping from tree names to words or phrases is constructed using only the syntactic database, since the mapping from words to tree files is established in syntactic database. In addition, because only tree files are involved in the mapping relation, all trees under the same file are mapped to the same set of words and phrases.

In order to construct a backward searching database, we need to iterate through each item in the syntactic database, using the tree file or family file in that item as the index, and append words or phrases in the syntactic item to the record in the backward searching database. The interface is defined as `construct_tree_to_word_db()`.

When the backend receives a request from the user to find all the words or phrases that could lexicalize an elementary tree, it queries the database with the tree file name to which the TAG tree in the request belongs, and returns the result. This interface is defined as: `tree_to_word()`.

5.4. Exploring The Tree Repository

The backend also allows requests be made for a single tree or for listing out all trees in a tree file. These two are usually needed when users are exploring the tree repository. Essentially, listing out trees using a file name is simply enumerates the name of the trees in that file, while querying for a single tree using a tree name retrieves that tree using the given name and parses it into a tree instance. This feature requires a database keeping the relationship between a tree name and a tree file name.

Although these two interfaces are straightforward, the tree returned is un-lexicalized, which means there are no feature structures and anchor words attached to it. However, for an un-lexicalized tree, backward searching should also be available, because

Interface	Description
`list_trees()`	Given a tree file name, returns all trees contained in the file
`get_tree()`	Given a tree name, return a TAG tree object

Figure 16. Exploring The Tree Repository Interface Description

all backward searching needs is the tree name, and we always know it.

5.5. Exception Handling

An exception is raised when the dataset appears to be inconsistent, or the backend could not find the desired entry. It interrupts the normal processing steps and then it tries to fix the problem, and then resume. By using *exception*, we do not always means the exception handling provided by programming language, but we refer to a more general situation, which indicates that something special must be done in order to complete the task.

There are several types of exceptions, and generally speaking, they could happen during any stage of processing. The most typical ones are, morphological record does not exist, syntactic item does not exist, and syntactic item exists but does not match. We will discuss them respectively.

By saying morphological record does not exist, we mean that a token from user input does not have a query result into morphological database. This can be quite common, especially when dealing with proper nouns. If it is this case, then a special default morphological record is used, which represents all tokens not in the morphological database for each part of speech tag.

In the second situation, "syntactic item does not exist", when a morphological record is passed to the interface for syntactic searching, if query returns no result, then the next step could not be taken, because of lack of information. In this case, the grammar viewer should return a special default syntactic item instead, which is designed for those unseen words in syntactic database, and has a default set of trees and features. It does not always yield good result, but is the best we could do.

In the last case, "syntactic item exists but does not match", we are in a situation very similar to the previous one. The major difference is that query to syntactic database does return something, but after comparing morphological information against their syntactic item identifiers (i.e. ENTRY and POS), there is not even one item that matches, and therefore the result is empty. To deal with this, the most straightforward method is to disable POS comparisons, matching only on the words.

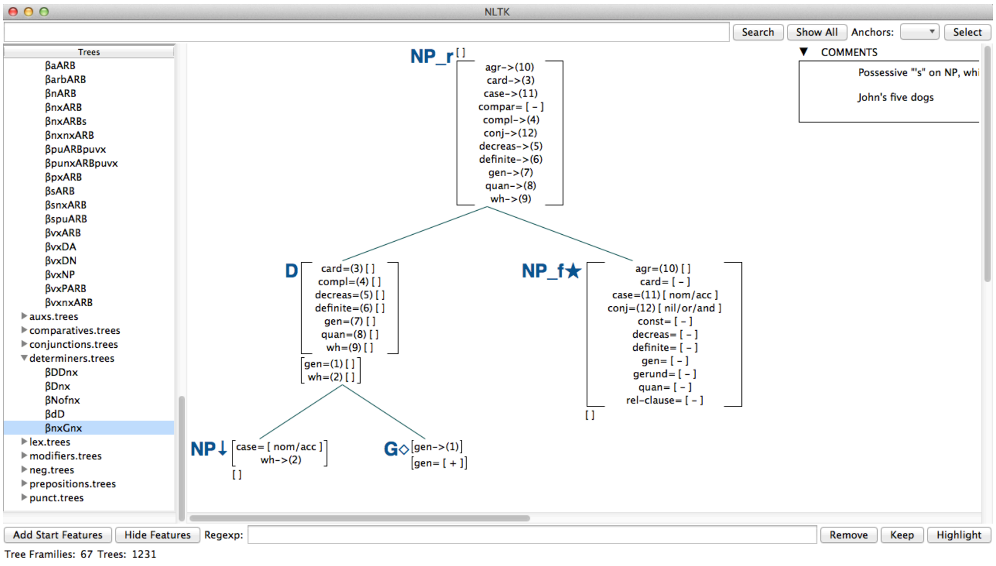


Figure 17. GUI to view the elementary trees in the XTAG grammar.

6. Graphical User Interface Design

Graphical User Interface (GUI), or called the front end, or the client, is the part that interacts with users, collecting user input and feedback, and encapsulates them into requests that will be sent to the server. The backend and GUI have complementary functionalities. In Section 5, we have seen that large portion of the backend contributes to its data processing ability. In this section, we will present knowledge about the GUI part of the grammar viewer.

The GUI is implemented using the Python GUI package Tkinter and uses additional helper functions from NLTK. The most useful feature of a GUI windows interface is that users can see the elementary TAG trees drawn on the canvas, and interact with them.

6.1. Window Layout

The main window layout is illustrated by Figure 17. It is divided into five sections, each being specialized for one or several similar purposes. The five sections are responsible for searching (the top), tree listing (middle-left), tree and feature drawing (middle-right), feature matching, displaying control (bottom), and status report (status bar) respectively.

On the top of the window, there are three buttons, one text box, and one dropdown list. They help users to do both forward searching and backward searching. More specifically, the left half is used for forward searching, while the right half is used for backward searching. In order to do forward searching, users need to enter the word or phrase into the textbox, and then click "Search" button. After the backend has finished processing data, the result will be shown on the list box. After searching, if the user wants to restore the list box to the starting state, just click "Show All" button, and the list box will be redrawn to show all available trees and tree families. For backward searching, users could select from the dropdown list on the right part when there is a tree drawn on the canvas, and the list will be filled with all possible words and phrases that could be used to lexicalize the current displaying tree. Choose one word in the dropdown list and click "Select", then the word will be selected, and the effect is same as entering the selected word into the textbox and then click "Search".

On the middle left of the window, there is a list box. After application startup, the list box simply lists out all trees and tree families in a hierarchical manner, with tree file name being the first level label, and individual trees being the second level. If users would like to explore tree repository without lexicalizing any of them, they could began their journey from here. If the content of the list box has been changed due to a previous searching, users could restore it to the initial state by click the "Show All" button on the top of the window.

The middle right part of the window is the canvas dedicated to drawing trees and feature structures. If there are comments recorded in tree database, then they will also be drawn. The content on the canvas will change in response if user selects tree from the list box.

The bottom of the window is responsible for controlling feature structure display. There are five buttons and one textbox. The "Add Start Features" button on the left is used to attach the start feature in configuration file to the root node as its top feature. And the "Hide Features/Show Features" button is used to control whether to draw feature structure on the canvas. The three other buttons along with the textbox is used to match features on the current tree. The matching algorithm is presented in Section 5. The matching pattern is a regular expression that conforms to Python's re library, and we have three options: keep those matched, remove those matched, or simply highlight those matched. These three options correspond to the three buttons respectively.

A status report is shown in the status bar. In the current version we only display number of trees and tree families. However, future developers could in addition choose report the status of the morphological database, syntactic database and the feature template database.

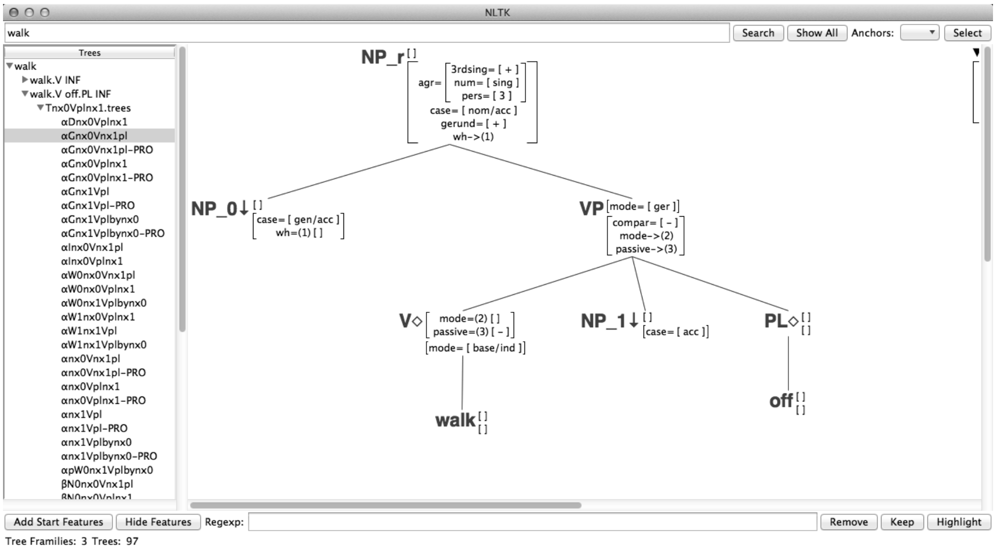


Figure 18. GUI to view lexicalized elementary trees in the XTAG grammar.

7. Conclusion

In this paper, we have described a Python-based interface for wide coverage Lexicalized Tree-adjoining Grammars. The grammars are part of the XTAG Grammar project and we hope that by providing an easy to use API to use these grammars that these grammars can be used for various NLP projects. We plan to incorporate our grammar viewer and the associated XTAG grammars for English and Korean into the Python Natural Language Toolkit (NLTK, `http://www.nltk.org/`).

Bibliography

Chen, John and K. Vijay-Shanker. Automated Extraction of TAGs from the Penn Treebank. In *Proc. of the 6th International Workshop on Parsing Technologies (IWPT-2000), Italy*, 2000.

Chiang, David. Statistical Parsing with an Automatically-Extracted Tree Adjoining Grammar. In *Proc. of ACL-2000*, 2000.

Joshi, Aravind and Yves Schabes. Tree adjoining grammars. In Rozenberg, G. and A. Salomaa, editors, *Handbook of Formal Languages and Automata*. Springer-Verlag, 1997.

Joshi, Aravind K, Leon S Levy, and Masako Takahashi. Tree Adjunct Grammars. *Journal of Computer and System Sciences*, 10(1):136–163, 1975. URL `http://dx.doi.org/10.1016/S0022-0000(75)80019-5`.

Rambow, O. and A. Joshi. A formal look at dependency grammars and phrase-structure grammars, with special consideration of word-order phenomena. In Wanner, Leo, editor, *Current Issues in Meaning-Text Theory*. Pinter, London, 1995.

Resnik, Philip. Probabilistic tree-adjoining grammars as a framework for statistical natural language processing. In *Proc. of COLING '92*, volume 2, pages 418–424, Nantes, France, 1992. URL `http://anthology.aclweb.org/C92-2065.pdf`.

Sarkar, Anoop. Applying co-training methods to statistical parsing. In *Proceedings of NAACL 2001*, Pittsburgh, PA, June 2001.

Schabes, Y. Stochastic lexicalized tree-adjoining grammars. In *Proc. of COLING '92*, volume 2, pages 426–432, Nantes, France, 1992. URL `http://dx.doi.org/10.3115/992133.992136`.

Schabes, Yves and Richard Waters. Tree insertion grammar: A cubic-time, parsable formalism that lexicalizes context-free grammar without changing the trees produced. *Computational Linguistics*, 21(4):479–513, 1995.

Schabes, Yves, Anne Abeillé, and Aravind K. Joshi. Parsing strategies with 'lexicalized' grammars: Application to Tree Adjoining Grammars. In *Proceedings of the* 12^{th} *International Conference on Computational Linguistics (COLING'88)*, Budapest, Hungary, August 1988.

Vijay-Shanker, K. *A Study of Tree Adjoining Grammars*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1987.

Xia, Fei. *Investigating the Relationship between Grammars and Treebanks for Natural languages*. PhD thesis, University of Pennsylvania, Philadelphia, PA, 2001.

Xia, Fei, Chunghye Han, Martha Palmer, and Aravind Joshi. Automatically Extracting and Comparing Lexicalized Grammars for Different Languages. In *Proc. of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001)*, Seattle, Washington, 2001.

XTAG Group. A lexicalized tree adjoining grammar for English. Technical Report IRCS-01-03, IRCS, University of Pennsylvania, 2001.

Address for correspondence:
Anoop Sarkar
`anoop@sfu.ca`
School of Computing Science
Simon Fraser University
8888 University Drive, Burnaby, BC, Canada