Implementing FUF in NLTK

Petro Verkhogliad
vpetro@gmail.com

Abstract

Natural language processing is a wide and varied field. This variety can been seen by the large number of available frameworks and formalisms for working with text. The two toolkits of interest to us are the *Natural Language Toolkit* written in the Python programming language and the *FUF/SURGE* system written in Common LISP. This paper describes the development and implementation of FUF module, `nltk.fuf`, within the NLTK library. The complete source code of `nltk.fuf` is located in the NLTK public code repository, `https://nltk.svn.sourceforge.net/svnroot/nltk`.

Contents

List of Figures

Acknowlegments

This work would not be possible without the help of Leo Ferres, Robert Dale, Edward Loper, Michael Elhadad and Jean-Pierre Corriveau. The author would also like to thank all the people at NLTK, Python Foundation and Google for the creation and support of the Summer of Code program.

1 Introduction

The Python programming language has long been viewed as a valuable tool for scientific research and teaching. At the same time, the Python *Natural Language Toolkit* (NLTK) is widely recognized as one of the best packages for exploratory natural language parsing and understanding However, prior to the work discussed here NLTK lacked support for natural language generation

There are a number of natural language generation packages implemented in languages such as Java, C, Prolog and LISP. One of the well received natural language generation projects is FUF/SURGE [8, 7] (or FUF). Initially implemented in Common LISP, FUF (Functional Unification Formalism) is an implementation of a natural language surface realizer which relies on *functional unification*. SURGE (Systemic Unification Realization Grammar of English) is the generation grammar for the English language. In the past, FUF/SURGE has been used successfully in a number of different projects. The addition of a FUF-like realizer to the NLTK toolkit brings NLTK closer to completeness in terms or natural language processing tasks. This addition further encourages the use of the Python language for research and education.

2 Background

2.1 Natural Language Generation

Natural language generation (NLG) is a subfield of natural language processing (NLP), which is in turn a subfield of artificial intelligence. The task of generating text may be considered opposite of natural language understanding[1] (NLU) as it starts with non-linguistic machine-understandable representation of text and aims to produce one or more grammatically and syntactically correct sentences.

2.1.1 NLG as an Application

NLG systems can be viewed from several distinct points of view. From an application perspective, they generally serve as a front-end to some larger system. In this case the NLG system receives the result of some processing and is tasked to render this information palatable to the human user. The most popular systems of this type are *data-to-text* systems similar to SumTime[2], STOP[3] or iGraph[4] However, it should be mentioned that

[1] Natural language processing = natural language understanding + natural language generation

[2] `http://www.csd.abdn.ac.uk/research/sumtime/`

[3] `http://www.csd.abdn.ac.uk/research/stop`

[4] `http://alba.carleton.ca/`

NLG technology has been used for other practical purposes such as teaching, marketing, behaviour change and providing accessibility [2].

2.1.2 NLG Research

NLG systems are an active area of research concerned primarily with being able to answer questions in the area of human-computer interaction. For instance, what can be considered "good" and "readable" text? What role do certain constructs play in our understanding of the description of some phenomenon or piece of data? How can different types of representation (graphical, auditory) be converted into a textual representation? These questions are related to a variety of disciplines, from psychology to computer science. The availability of different NLG frameworks or toolkits is essential for researchers and educators and may help in answering these questions.

2.1.3 The Many Parts of NLG

A typical NLG system, whether it is an application front-end or research prototype, consists of the modules responsible for [3]:

- *Discourse Planner*
  The module that starts with the communicative goal (ie what the system intends to communicate) and makes the following choices:
  - Content selection
    responsible for selecting the appropriate content from a possibly overspecified input.
  - Lexical selection
    responsible for selecting the appropriate lexical expressions.
  - Sentence structure
    responsible for selecting the size of the sentences as well for the choice of referring expressions[5].
- *Surface Realizer*
  The module that receives a complete specification of the discourse plan and produces complete and correctly structured sentences as determined by its grammatical and lexical resources.

Although the description above is "good enough" to describe a number of NLG systems it is by no means complete. Other architectures have been proposed and used with considerable success in research prototypes.

[5] http://en.wikipedia.org/wiki/Referring_expression

This paper is centered around surface realization, as such, the reader will benefit from an exploration of this topic.

2.1.4 Surface Realization

The surface realization module produces ordered sequences of words as determined by the provided lexicon and grammar. The vital point is that the surface realizer takes chunks of input (sentence specifications expressed with a given formalism) and combines them with a given grammar (expressed with the same formalism). One of the more popular approaches for expression of the input and the grammar and combining them to produce a textual output is the *Functional Unification Grammar* [1] which is explored in more detail in the next section.

2.2 Functional Unification Grammar

This formalism was first introduced by Michael Kay in the paper titled "*Functional Unification Grammar: A Formalism for Machine Translation*". The motivation for the development of the grammar was two-fold [6]:

1. maintaining a computational aspect to the formalism[6].
2. to allow structure and functional notions to work side by side

The grammar formalism uses unification to manipulate and reason about *feature structures*. The basic idea in FUG is to build the generation grammar as a feature structure with lists of possible alternations and then to unify the grammar with the input structure built using the same type of specification. It is precisely this approach[7] that is used in the FUF/SURGE system which is the focal point of the work discussed in this paper.

2.2.1 Feature Structures

A feature structure[8] is essentially a set of key-value pairs that provide a partial description of a sentence[9]. Feature structures resemble first-order logic terms but have several restriction lifted:

- Sub-structures are labeled symbolically, and are not inferred by argument position.

[6] After all, the original design intended for this to be used for machine translation tasks

[7] with some extensions

[8] Over the years these structures have surfaced under a variety of names such as feature bundles, feature matrices, functional structures, functional descriptions, or terms

[9] A feature structure can also be viewed as a partial function from features to their values.

- Fixed arity is not required.
- The distinction between function and argument is removed
- Addition of variables and reentrance[10].

Although varied in naming the generally accepted notation for feature structures is shown in figure 1.

$$\mathcal{F}_0 \rightarrow \begin{bmatrix} article: & "the" \\ noun: & "cat" \end{bmatrix}$$

Figure 1: A simple feature structure

The key set[11] of $\mathcal{F}_0$ is $\{article, noun\}$ and the value set is $\{$"the", "cat"$\}$. Note that in a given structure a key can only be defined once.[12] The mapping defined by the structure shown in figure 2.

$$\mathcal{F}_0(article) \rightarrow "the"$$
$$\mathcal{F}_0(noun) \rightarrow "cat"$$

Figure 2: Feature structures as functions

One of the important features of the unification-based formalisms is that the feature structures may be nested (ie a value of a key in the structure may be another feature structure) or contain variables. In figure 3, $\mathcal{F}_1$ is an example of a structure that contains sub-features, whereas $\mathcal{F}_2$ contains the variable $x1?$.

Another important component of feature structures is *reentrance*. A reentrant feature structure is one in which two keys share common values. It must be noted that all the values shared by two different keys must be the same, otherwise the feature structure is not reentrant.

[10] Also known as, co-reference

[11] or attribute set

[12] Keys with the same name and different values may appear in the sub-features but not on the same "level" of a given structure.

$$\mathcal{F}_1 \rightarrow \begin{bmatrix} cat: & s & \\ prot: & \begin{bmatrix} n: & "john" \end{bmatrix} \\ verb: & \begin{bmatrix} v: & "like" \end{bmatrix} \\ goal: & \begin{bmatrix} n: & "mary" \end{bmatrix} \end{bmatrix}$$

$$\mathcal{F}_2 \rightarrow \begin{bmatrix} cat: & s & \\ prot: & \begin{bmatrix} n: & "john" \end{bmatrix} \\ verb: & \begin{bmatrix} v: & "like" \end{bmatrix} \\ goal: & \begin{bmatrix} n: & x1? \end{bmatrix} \end{bmatrix}$$

Figure 3: Feature structures with nested values and variables

As an example consider the structure $\mathcal{F}_3$ in figure 4. In $\mathcal{F}_3$ there are two keys *prot*, *goal* map onto different sub-features that contain the same values of the same type. The notation for this can be simplified as shown in figure 5. Note that a given feature structure may contain a loop by the use of reentrance.

$$\mathcal{F}_3 \rightarrow \begin{bmatrix} cat: & s & \\ prot: & \begin{bmatrix} n: & "john" \end{bmatrix} \\ verb: & \begin{bmatrix} v: & "like" \end{bmatrix} \\ goal: & \begin{bmatrix} n: & "john" \end{bmatrix} \end{bmatrix}$$

Figure 4: $\mathcal{F}_3$ - A Reentrant Feature Structure

$$\mathcal{F}_4 \rightarrow \begin{bmatrix} cat: & s & \\ prot: \boxed{1} & \begin{bmatrix} n: & "john" \end{bmatrix} \\ verb: & \begin{bmatrix} v: & "like" \end{bmatrix} \\ goal: & \boxed{1} & \end{bmatrix}$$

Figure 5: $\mathcal{F}_4$ - Reentrant Feature Structure Notation

A *path* in a feature structure is a sequence of keys which can be used to select a specific value (be it primitive or complex) by repeated application. For example, the path $< prot\ n >$ can be applied to the feature structure $\mathcal{F}_3$ (Figure 6).

$$\mathcal{F}_3(< prot\ n >) \rightarrow "john"$$

Figure 6: Paths in Feature Structures

Feature structures can also be viewed as graphs. This representation becomes useful when we discuss some of the special features in FUF/SURGE .

2.2.2 Feature Structures as Graphs

Feature structures can be viewed as rooted, directed, graphs[13]. As an example we can represent the structure in figure 3 as the graph in figure 7. Note, the graph in figure shows the *lex* keys which are implicit in the abstract representation of feature structures we have been working with up to now.

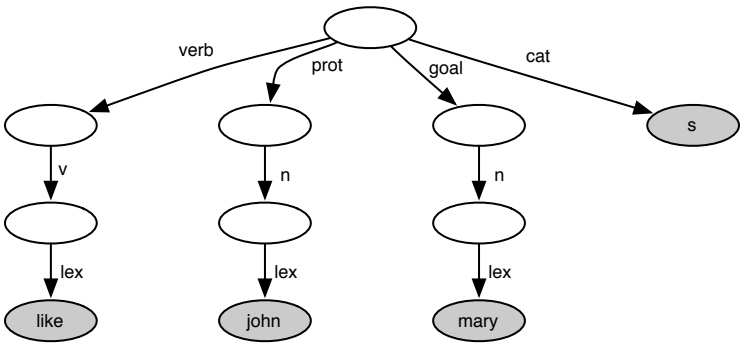

Figure 7: $\mathcal{F}_1$ - Feature Structure as a Graph

The edges of the graph represent the keys of the features (or attributes). The grey colored nodes are terminal nodes[14], while the empty nodes are nested sub-features. There are two compelling reasons for using graph notation with feature structures. One, graph theory gives us a simple and well-defined vocabulary with which to talk about and model

[13] In some literature the structures are restricted to being acyclic only. However, there is little that prevents one from building a cyclic feature structure as we have seen above.

[14] atomic values

features structure systems. Two, it provides a sound framework for investigating potential structure-combining operations.

2.2.3 Unification with Feature Structures

The most important operation that can be performed on feature structures is *unification*. Two given structures $\mathcal{F}_{s0}$ and $\mathcal{F}_{s1}$ unify if their key-value pairs are in agreement [4]. The extra key-value pairs are simply copied into the result. At the same time, the structures $\mathcal{F}_{s0}$ and $\mathcal{F}_{s2}$ do not unify due to a conflict in the value of $< cat >$.

$$\mathcal{F}_{s0} \rightarrow \begin{bmatrix} cat: & np \\ number: & singular \end{bmatrix}$$

$$\mathcal{F}_{s1} \rightarrow \begin{bmatrix} cat: & np \\ sub: & \begin{bmatrix} person: & third \end{bmatrix} \end{bmatrix}$$

$$\mathcal{F}_{s2} \rightarrow \begin{bmatrix} cat: & v \end{bmatrix}$$

$$\mathcal{F}_{r0} \rightarrow \begin{bmatrix} cat: & np \\ number: & singular \\ sub: & \begin{bmatrix} person: & third \end{bmatrix} \end{bmatrix}$$

Figure 8: Unification of $\mathcal{F}_{s0}$ and $\mathcal{F}_{s1}$ and their result $\mathcal{F}_{r0}$

Given the example above we have an intuitive understanding of simple unification with basic feature structures. The rest of the discussion builds on these topics.

3 Feature Structures in NLTK and FUF/SURGE

Both NLTK and FUF use slightly different syntax for representing feature structures. Furthermore, FUF departs from the basic unification and syntax described above. In the following sections we will explore these differences.

3.1 NLTK

Given that a feature structure is a mapping from keys to values NLTK uses *feature dictionaries* and *feature lists* for representing features and their values Creating a structure in NLTK is quite straightforward (figure 9).

```
1       >>> from nltk.featstruct import FeatStruct
2       >>> fs1 = FeatStruct(number="singular", person=3)
3       >>> fs2 = FeatStruct(tense="past", arg=[number="sing", person=3])
```

Figure 9: Defining a feature structure with NLTK FeatStruct

In line 1 the code[15] imports the libraries for representing and reasoning with feature structures. In line 2, a structure `fs1` is created which contains the keys $\{number,\ person\}$ and their values $\{'singular',\ 3\}$. Finally in line 3, another more interesting feature structure, `fs2`, is defined.

Earlier it was mentioned that that a single key may not have several values assigned to it. This is not the case in in the NLTK implementation. Keys may have lists assigned to them as shown[16] in figure 10. While this is a significant departure from the original definition it has little impact on this work as unification of keys with list values is undefined[17]. At the same time, the use of lists allows us to handle `cset` and `pattern` attributes during processing discussed in section 3.3.

```
>>> fs3 = FeatStruct('[x = (1, 2, 3 4)]')
>>> print fs3
[ x = (1, 2, 3, 4) ]
```

Figure 10: NLTK feature structure with lists as values

Figure 11 compares NLTK's "pretty" representation of feature structures to the notation we have been using up to now.

NLTK allows for reentrance. Figure 12 shows the interaction in the Python interpreter that creates a reentrant feature structure. In the structure in figure 12 the value at $< a >$ is an empty structure and $< c\ d >$ shares that value[18].

Feature structures in NLTK can also contain variables as shown in the figure 13.

[15] For more information about the Python programming language the reader is invited to consult `http://python.org`

[16] In this example we use slightly different syntax for defining the feature structure. The NLTK FeatStruct definition parser does not allow us to specify the feature structure as we have done in 9

[17] Unless we define custom unification for the value of said key

[18] The = is used for assignment of a value to a feature. The -> explicitly states that the value of b is the value of a by reentrance

$$\mathcal{F}_{s1} \rightarrow \begin{bmatrix} cat: & np \\ sub: & \begin{bmatrix} person: & third \end{bmatrix} \end{bmatrix}$$

```
[ cat = 'np'                ]
[                           ]
[ sub = [ person = 'third' ] ]
```

Figure 11: Notational difference between NLTK and the abstract feature structure syntax

```
1       >>> from nltk.featstruct import FeatStruct
2       >>> fs2 = FeatStruct('[a=(1)[], b->(1), c=[d->(1)]]')
3       >>> print fs2
```

```
[ a = (1) []      ]
[                 ]
[ b -> (1)        ]
[                 ]
[ c = [ d -> (1) ] ]
```

Figure 12: Reentrant feature structure with NLTK

```
1       >>> from nltk.featstruct import FeatStruct
2       >>> from nltk.sem.logic import Variable
3       >>> fs4 = FeatStruct("[number='?x']")
4       >>> print fs4
```

```
[ number = '?x' ]
```

Figure 13: NLTK feature structure with a variable

In conclusion, the differences between the feature structure representation in NLTK and found in the computational linguistics literature is mostly syntactic. There are no semantics defined on any of the keys or values in the feature structures defined in NLTK.

3.2 FUF Feature Structures

The basics in FUF structures are very similar to those in the literature and NLTK. However, some additional constructs were added during the development of FUF that simplify grammar writing and the unification process. Before discussing the "special" keys and values we begin with some representational differences. Figure 14 compares the abstract feature structure representation to its concrete implementations in FUF and NLTK.

```
((article "the") (noun "cat"))

    [ article = 'the' ]
    [ noun    = 'cat' ]
```

$$\begin{bmatrix} article: & 'the' \\ noun: & 'cat' \end{bmatrix}$$

Figure 14: Syntactic representations of feature structures

3.3 Special Features and Values in FUF

There are a number of special features and values defined in FUF to ease grammar writing and debugging. The first of these is *relative and absolute paths*. Paths in FUF are represented as lists of symbols enclosed in curly braces. This notation escapes ambiguity since at each level of the structure there is at most one feature with a given attribute.

As previously mentioned, paths can be *absolute* or *relative*. Absolute paths always begin at the very top of the feature structure and flow down along specified keys. A relative path uses the caret (^) notation to signify that we must go up one level in the current structure before following the list of symbols down. A given relative path may include any number of ^ in order to go up any number of levels. As the FUF representation of feature structures is not very easy to follow (figure 15) we will use the NLTK representation from now on. The feature structure in figure 16 contains a value {^^prot number}. The dark arrows in the accompanying graph take the path specified by this link.

```
((cat s)
  (prot ((cat np)
         (number sing)))
  (verb ((cat vp)
         (number {^^ prot number}))))
```

Figure 15: Simple FUF Feature

The astute reader will notice that in the above feature structure we could have used reentrance instead of a relative link to specify that $< verb\ number >$ shares its value with $< prot\ number >$. However, FUF does not define extra syntax for reentrance. At the same time, FUF links need not reference an existing value. The referenced value may be added to the structure during unification. This, in turn, extends the expressibility of the formalism. [7]

The next set of "special" features are those that have special unification behaviour. The most frequently used "special" features are:

```
[ cat  = 's'                          ]
[                                     ]
[ prot = [ cat    = 'np'   ]          ]
[        [ number = 'sing' ]          ]
[                                     ]
[ verb = [ cat    = 'vp'           ] ] 
[        [ number = {^^prot number} ] ]
```

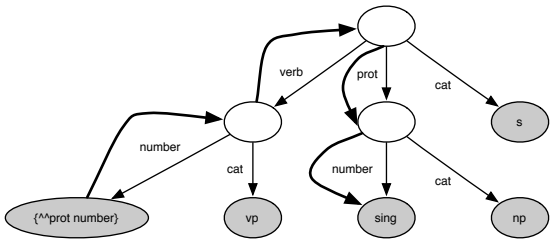

Figure 16: Feature structure with a relative path

- *alt* - ordered alternation
- *ralt* - random alternation
- *opt* - optional unification
- *cset* - specification of unification elements
- *pattern* - sentence ordering

3.3.1 `alt` and `ralt`

In FUF the syntax for specifying alternation is shown in figure 17a. Figure 17b shows an `alt` with a name, while figure 17c shows an indexed `alt`.

```
(ALT annotations* (fs1 fs2 ... fsn))
```
(a) `alt`

```
(ALT TOP (fs1 fs2 ... fsn))
```
(b) Named `alt`

```
(ALT (index topalt) (fs1 fs2 ... fsn))
```
(c) Indexed `alt`

Figure 17: `alt` syntax

When the unifier encounters the `alt` key it does not try to unify against the entire `alt` sub-feature. It selects the sub-features of the `alt` one by one and attempts to unify them with the input. Should the unification fail with the first alternation, the next alternation is tried. If all the possible branches have been tried without a successful unification, the process is said to have failed. The ordering of the branches of the `alt` specifies the order

in which the unification of the input structure is attempted with each of the alternations. However, if the order is not important the `ralt` feature can be used. Aside from the difference in the order of unification attempts there are no other differences between `alt` and `ralt`.

As is usually the case with feature structures one member of the `alt` family can be embedded within another, figure 18 illustrates this.

```
(alt ( (cat n)
       (ralt (1 2 3 4))))
```

Figure 18: Nested `alt`

3.3.2 `opt`

`opt` specifies those features which, if they would cause unification to fail, can be disregarded in order for the unification to succeed. The syntax for this feature is:

```
(opt (fs1 fs2))   (opt top (fs1 fs2))   (opt top ({ prot number } fs2))
```

(a) Simple `opt` (b) Named `opt` (c) Named `opt` containing a path

Figure 19: `opt` syntax

The above code causes that the unifier to attempt the unification with any of the feature structures specified in the `opt` list. However, if this fails then the unifier can try again without the optional features. If the unification fails again (ie without the optional features), the entire process is considered a failure.

3.3.3 `cset`

In general the unifier in FUF begins at the root of the graph and works downward, recursively unifying the constituents of the grammar. By default the feature structures which contain the key `cat` or are in the `pattern` list are considered constituents.

In order to simplify and reduce the amount of work the unifier has to do, the *constituent set* (`cset`) can be specified explicitly as shown in figure 20.

Since the `cset` value is treated as a set, its members are unordered and can be unified with another differently ordered constituent set definition. At the same time, the `cset` can contain paths (figure 20b).

```
(cset (prot verb goal))
```
(a) Simple `cset`

```
(cset ({prot} verb goal))
```
(b) `cset` containing a path

Figure 20: `cset` syntax

3.3.4 `pattern`

The `pattern` feature describes the ordering of constituents that must be processed during the generation of the textual output given the result of unification. The syntax for the pattern feature is shown in figure 21.

```
(pattern (prot verb goal))
```
(a) Simple `pattern`

```
(pattern ({prot} verb goal))
```
(b) `pattern` with a path

```
(pattern ({prot} verb dots goal pound))
```
(c) `pattern` with special values

Figure 21: `pattern` syntax

Each of the members mentioned in the `pattern` may contain another pattern specification. Given the example syntax above the *linearizer* module must sequentially (depth-first) process each of the members. This process will result in the final output sentence. The `pattern` directives are generally added to the final result by the grammar, since the input to the unifier should be a semantic, rather than syntactic, representation. Similar to the `cset` directives, `pattern` may use paths as elements. Furthermore, the symbols `dots` and `pound` were added to relax the `pattern` constraints during unification (figure 21c). The `dots` elements acts as the regular expression ".*", allowing for another value (or any number of values) to be substituted in its place. The `pound` symbol acts as the regular expression "?", allowing for zero or one appearances of the substituting symbol.

For example, the pattern `(f1 dots f2)` specifies that constituent `f1` must precede constituent `f2`. However, they need not be adjacent. The pattern `( pound f1 f2)` specifies that another term may appear before `f1`. It is important to note that similar to the `cset` directive the `pattern` directive specifies a list of terms as its value. This, in turn, requires special unification to be defined for all `pattern` values. Due to the addition of `dots` and `pound`, the unification of `pattern` values is non-deterministic. Figure 22 defines two `pattern` values. Figure 23 shows possible results of unifying[19] `p1` and `p2`.

```
(pattern (dots a dots b dots))
```

(a) p1

```
(pattern (dots c dots d dots))
```

(b) p2

Figure 22: `pattern` values to be unified

```
(pattern (dots a dots b dots c dots d dots))
(pattern (dots a dots c dots b dots d dots))
(pattern (dots a dots c dots d dots b dots))
(pattern (dots c dots a dots b dots d dots))
(pattern (dots c dots a dots d dots b dots))
(pattern (dots c dots d dots a dots b dots))
```

Figure 23: Possible results of pattern unification

3.4 Typed Feature Structures in FUF

Aside from using the "special" keys and values described above, FUF allows more general changes to the semantics of features within a grammar.

```
(define-feature-type <name> <children>)
```

(a) Type definition syntax

```
(define-feature-type noun (pronoun proper common))
(define-feature-type pronoun (personal-pronoun question-pronoun
          demonstrative-pronoun quantified-pronoun))
(define-feature-type common (count-noun mass-noun))
```

(b) Type definition example

```
    ((cat noun)
     (alt (((cat pronoun)
        (cat ((alt (question-pronoun personal-pronoun
             demonstrative-pronoun quantified-pronoun)))))
        ((cat proper))
        ((cat common)
        (cat ((alt (count-noun mass-noun) )))))))
```

(c) Type definition usage

Figure 24: Type definitions

So far we have seen that feature structures in FUF can contain three types of values: sub-features, primitives and variables. However, FUF adds another value to the preceding list: typed value. Figure 24b shows how we can define types in FUF. Figure 24c shows how we can use the defined types in a grammar. By using type definitions we can use `((cat`

[19]This example is from the FUF Manual v5.2

`personal-noun))` instead of the much more verbose `((cat noun) (noun pronoun) (pronoun personal))`.

Furthermore due to the added semantics of types, the FUF unification algorithm must be modified: two primitive typed values $\mathcal{T}_0$ and $\mathcal{T}_1$ are unified if one subsumes the other (ie one of them is more general then the other). The result of their unification is the most specific type.

3.5 Special Operations on Feature Structures in FUF

Prior to generating an output sentence a unified feature structure must pass through the FUF's *linearizer* and *morphology* modules. The linearizer interprets the `pattern` directives and assembles the words of a sentence into a single string while taking care of punctuation and capitalization. The general algorithm for the linearizer is as follows [7]:

1. If there is a feature gap, return an empty string
2. Otherwise,
   (a) find the value of the `pattern` key and if found recursively linearize its elements.
   (b) If the `pattern` is not found try the following:
      i. Find the `lex` feature pass it through the morphology and return the result
      ii. Identify and apply the punctuation

If the linearization process is successful the resulting string (with appropriate punctuation and correct morphological features) is returned.

4 Implementing FUF Features and Operations in Python

The following describes the implementation, `nltk.fuf`, of the library features in the FUF realizer. This section is structured from the point of view of an application developer attempting to use a FUF grammar and input in order to generate a natural language sentence. The code is packaged in a library within NLTK under `nltk.fuf`. The most important modules are:

- `fufconvert.py` - converts a given FUF grammar written as an s-expression to an `nltk.FeatSturct` object type.
- `fuf.py` - unifies the input and grammar structures according to the FUF guidelines.
- `linearizer.py & morphology.py` - uses the result of the unification to produce the final output.

4.1 `fufconvert.py` or Parsing the Input

The `fufconvert.py` module is responsible for converting a given s-expression[20] into an NLTK feature structure. There are two main entry points to `fufconvert.py`.

- `fufconvert.fuf_to_featstruct` function. This function takes as input a single s-expression and returns an `nltk.FeatStruct` object.
- `fufconvert.fuf_file_to_featstruct` function. This function takes as input a FUF grammar file that may contain type definitions. It returns a Python tuple the first element of which is the type table while the second element is the converted grammar (`nltk.FeatStruct`).

```
1   >>> # import the module
2   >>> import fufconvert
3   >>> # read a line from the text file
4   >>> line = open('tests/gr0.fuf').readlines()[0]
5   >>> print line
6   >>> # convert the line to feature structure
7   >>> fstruct = fufconvert.fuf_to_featstruct(line)
8   >>> print fstruct
```

Figure 25: Converting s-expression to `FeatStruct` object

The code in figure 25 actually takes several internal steps before returning the result. The trivial step is reading and showing the input (lines 4-6). The next step is the call to the conversion function. This function must correctly interpret the s-expression before outputting the result. The best way of doing this is by using a push-down automata (PDA) [5]. The PDA removes all the comments and directives irrelevant to the semantic content of the structure, and proceeds to iterate through its states[21]. Finally, the PDA returns its result: a Python list of lists with the same structure as the original s-expression (figure 26).

The code in figure 27 is very similar to that in figure 25. The difference is that the file `'tests/typed_gr4.fuf'` contains type definitions as well as a generation grammar. Once

[20] `http://en.wikipedia.org/wiki/S-expression`

[21] For the exact specification of the states used in the PDA consult the code in the NLTK Subversion repository

```
((cat s)
(prot ((n ((lex john)))))
(verb ((v ((lex like)))))
(goal ((n ((lex mary))))))
```

(a) Input s-expression

```
>> import sexp
>> line = open('tests/sexp.txt').readlines()[-4]
>> result = sexp.SexpListParser().parse(line)
```

(b) Parsing code

```
<SexpList:  ((cat s)
(prot ((n ((lex john)))))
(verb ((v ((lex like)))))
(goal ((n ((lex mary))))))>
```

(c) S-expression parsing result

Figure 26: Parsing s-expressions

```
>>> from fufconvert import fuf_file_to_featstruct
>>> type_table, grammar = fuf_file_to_featstruct('tests/typed_gr4.fuf')
```

Figure 27: Converting grammar with type definitions

again, the PDA performs no semantic processing. It simply converts the type definitions into lists. Afterwards, it performs grammar conversion from s-expression to Python lists.

Once the lists have been generated, the next step is to convert them into `FeatStruct` objects. For basic feature structures this is quite straightforward. However, the "special" features of FUF structures must be processed. Those features with special unification semantics are discussed in later sections. First we will focus on the structures with special syntax. These structures are:

- `alt` and `ralt` - as mentioned previously these directives specify alternations
- `opt` - as discussed above `opt` specifies optional unification elements
- relative and absolute links

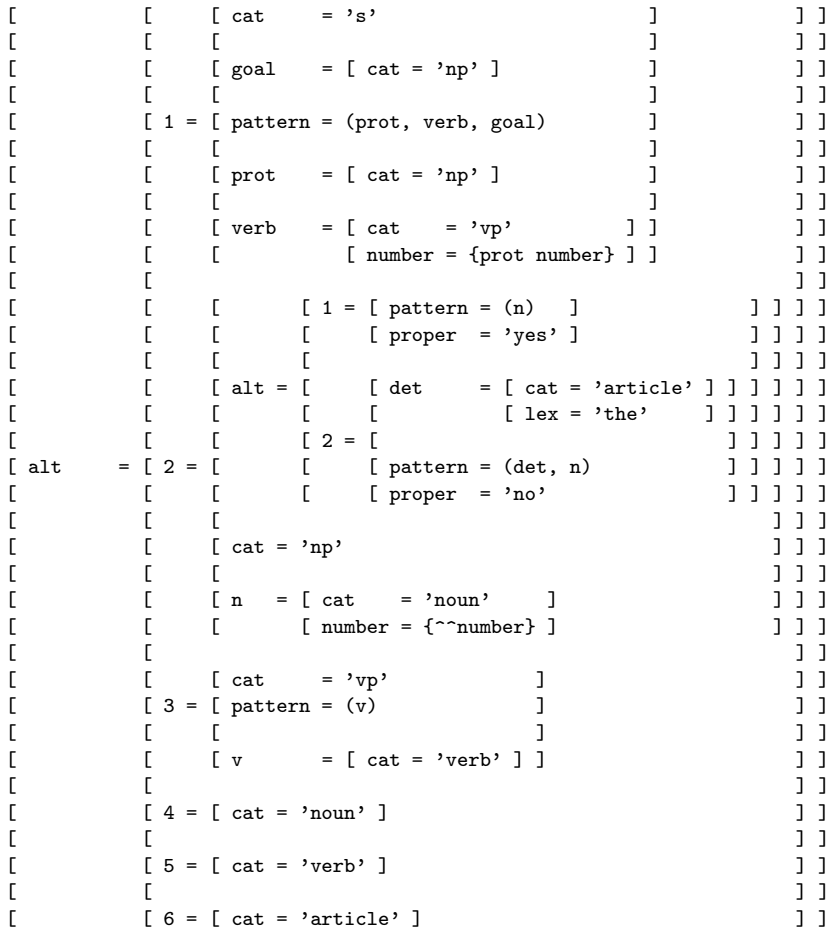

Figure 28: `alt` feature structure

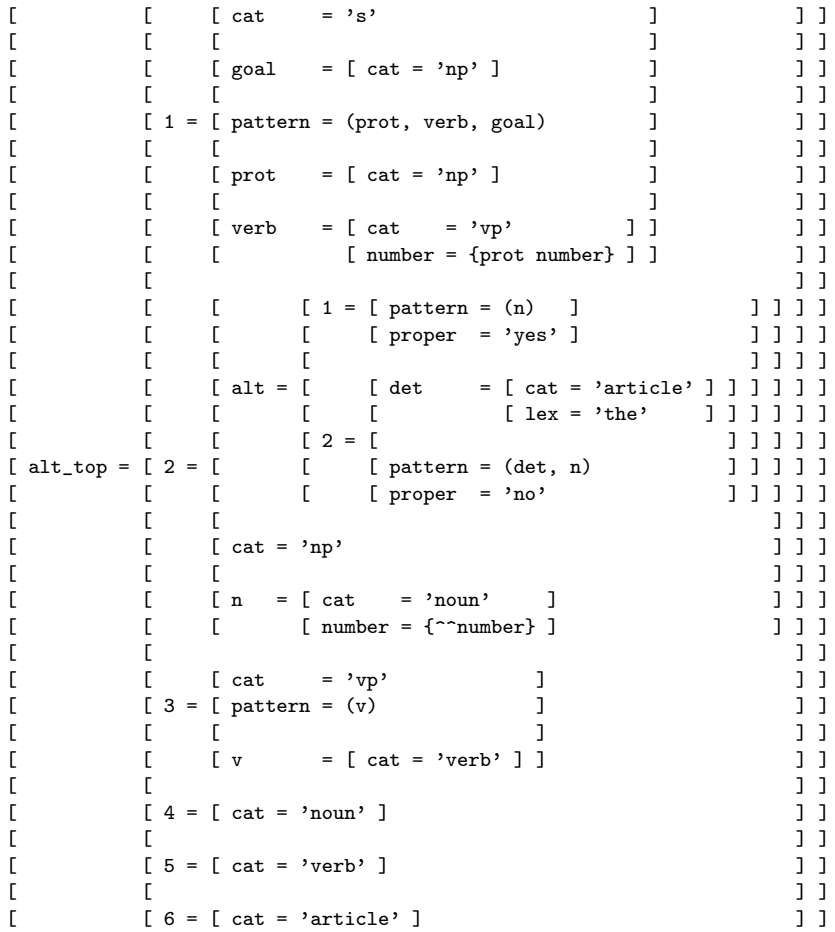

Figure 29: Feature structure containing a named `alt`

A typical alt-containing feature structure is shown in figure 28. It contains a top level alt feature with six possible alternations. One of the alternations contains another alt feature. These can be named, as shown in figure 29. (The name is the string following the "_" or "top".)

The next attribute of interest is opt. As previously shown (section 3.3.2) opt is essentially an alt with an empty feature structure added to the list of alternatives. Upon encountering an opt feature the implemented FUF grammar parser renames it to an alt and adds the empty feature to the list of alternatives. Figure 30a shows a simple opt feature and figure 30b shows the same structure after conversion. Figure 31 shows a named opt converted to a named alt.

```
(opt ((punctuation ((after "."))) ) )
```

(a) opt input s-expression

```
[       [ 1 = [ punctuation = [ after = '.'  ]  ]  ]]
[ alt = [                                          ]]
[       [ 2 = []                                   ]]
```

(b) opt parsing result

Figure 30: opt converted to alt

```
[               [ 1 = [ punctuation = [ after = '.' ] ] ] ]
[ alt_somename = [                                      ] ]
[               [ 2 = []                                ] ]
```

Figure 31: Named opt converted into a named alt

Since there is a little bit more work that needs to be done to convert these feature the code for doing so is found in the nltk.fuf.specialfs module.

The larger alt structures also contains relative and absolute links. These values remain unprocessed until unification starts. Looking at the converted structure in figure 29, one of the links is found at alt_top['2']['n']['number'] key. The process of link resolution is discussed in section 4.2.1. Finally, after the input feature structure and the grammar have been converted, we can proceed to their unification.

4.2 fuf.py or Towards a Result

Unification of FUF based feature structures is performed through the nltk.fuf.fuf.Unifier class. Before the unification can be started some housekeeping must be done. Part of

this task revolves around processing the `alt` features. Before the start of unification the code goes through all the possible alternations in the grammar and creates a list of those paths. This is done through the `nltk.fuf.fuf.GrammarPathResolver`. The result of the resolution is a list of all possible feature structures that can be generated through the alternations. The original LISP FUF does not do this, rather it picks one `alt` sub-feature after another and tries to unify them with the input. The expansion of all the nested paths in `nltk.fuf` leads to a exponential growth of unification attempts [7]. In the future, this could be dealt with by implementing the `index` annotation present in FUF. The unifier can use the value of the index to choose one branch from the alternatives without ever considering the other branches[22].

Once the paths of the grammar have been generated, the next step is to check for feature value types. FUF types have been discussed earlier, therefore, we only focus on their implementation. Feature value type handling is done through the `nltk.fuf.fstypes` module. The two classes contained within the module are:

- `FeatureValueTypeTable` - the type table does all the maintenance of the value types. The relationships are stored within a Python `dict` object. Figures 32 and 33 demonstrate the conversion of the type table from FUF to `nltk.fuf`. The type table can also be used to check for subsumption. This is shown in figure 34.

- `TypedFeatureValue` - this class is used to represent the typed value within a feature structure. The `TypedFeatureValue` is a subclass of the `CustomFeatureValue` class defined in `nltk.featstruct` module. The `CustomFeatureValue` allows us to define special unification on the values that are instances of this class.

```
(define-feature-type mood (finite non-finite))
(define-feature-type finite (declarative interrogative bound relative))
(define-feature-type non-finite (imperative present-participle infinitive))
(define-feature-type interrogative (yes-no wh))
```

Figure 32: FUF Type Definitions

As in the case with the `alt` unpacking, before the grammar can be unified with the input feature structure, feature values which are defined in the type table have to be instantiated within the grammar. This work is performed by the `fstypes.asssign_types` function. It

[22]Consult the FUF manual for an extended example.

```
relative <--- ['simple-relative', 'embedded-relative', 'be-deleted-relative',
                    'wh-nominal-relative', 'wh-ever-nominal-relative']
mood <--- ['finite', 'non-finite']
non-finite <--- ['imperative', 'present-participle', 'infinitive']
deontic-modality <--- ['duty', 'authorization']
pronp <--- ['personal-pronoun', 'question-pronoun',
               'quantified-pronoun', 'demonstrative-pronoun']
det <--- ['possessive-det', 'demonstrative-det', 'regular-det']
interrogative <--- ['yes-no', 'wh']
process-type <--- ['action', 'mental', 'attributive', 'equative']
np <--- ['pronp', 'common', 'proper']
finite <--- ['declarative', 'interrogative', 'bound', 'relative']
possessive-det <--- ['np']
modality <--- ['epistemic-modality', 'deontic-modality']
epistemic-modality <--- ['fact', 'inference', 'possible']
```

Figure 33: `nltk.fuf.fstypes` Type Table

```
>>> type_table.subsume('np', 'common')
>>> # or
>>> types_table.subsume('mood', 'imperative')
```

Figure 34: Checking for subsumption in `nltk.fuf.fstypes`

traverses the given feature structure and replaces the primitive values with the instances of `TypedFeatureValue`. For an example of this consult code in `nltk.fuf.fstypes` module.

4.2.1 Unification

Once all of the housekeeping is done we can proceed to the actual unification of the input and grammar feature structures. The process of unification is defined in the `nltk.fuf.fuf.Unifier` class and more specifically the `unify` method, shown in figure 36.

The unifier attempts to unify one of the alternations from the list of alternatives generated by the `GrammarPathResolver` with the input. If all of the attempts fail to unify with the input then the unification has failed. If one of the attempts succeeds, the resulting feature structure must be checked for relative or absolute links.

Dynamic link resolution is performed by the `nltk.fuf.link.LinkResolver` class. and the `resolve` method in particular. This class is capable of handling both types of links (figure 35). It is worth nothing that the `resolve` method will try to find the value of the link whether it is another feature structure or a primitive value. However, if the value cannot be found the link will be replaced with `nltk.sem.Variable` variable object. If the value of the link is found to be a variable, the variable in question will be copied. If the value of the link is another link, the resolver will first attempt to resolve the newly found link and then copy the value to the link that was encountered at the start of the resolution.

```
>>> fs1 = fufconvert.fuf_to_featstruct("""
                   ((cat s)
                      (prot ((cat np)
                              (number sing)))
                      (verb ((cat vp)
                              (number { ^ ^ prot number})))) """)
>>> print fs1
>>> lr = LinkResolver()
[ cat  = 's'                            ]
[                                       ]
[ prot = [ cat    = 'np'   ]            ]
[        [ number = 'sing' ]            ]
[                                       ]
[ verb = [ cat    = 'vp'            ] ]
[        [ number = {^^prot number} ] ]
>>> lr.resolve(fs1)
>>> print fs1
[ cat  = 's'                ]
[                           ]
[ prot = [ cat    = 'np'   ] ]
[        [ number = 'sing' ] ]
[                           ]
[ verb = [ cat    = 'vp'   ] ]
[        [ number = 'sing' ] ]
```

Figure 35: Link Resolution Example

Once the link resolution is finished on the result of the unification, the unifier goes through the sub-features of the result and attempts to unify them with the alternations in the list of alternatives. Only those features which are considered to be constituents are unified (what it means for a feature to be a constituents has been described earlier). Figures 37, 38 and 39 show the above described process.

```
@staticmethod
def _unify(fs, grs, resolver=None, trace=False):
    unifs = None
    for gr in grs:
        unifs = fs.unify(gr)
        if unifs:
            resolver.resolve(unifs)
            for fname, fval in unifs.items():
                if Unifier._isconstituent(unifs, fname, fval):
                    newval = Unifier._unify(fval, grs, resolver)
                    if newval:
                        unifs[fname] = newval
            return unifs
    return unifs


def unify(self):
    """
    Unify the input feature structure with the grammar feature structure

    @return: If unification is succesful the result is the unified
    structure. Otherwise return value is None.

    """

    self.lr.resolve(self.fsinput)
    # make a copy of the original input
    return Unifier._unify(self.fsinput, self.grammar_paths, self.lr)
```

Figure 36: `unify` method

```
>>> itext, gtext = open('tests/uni.fuf').readlines()
# set up the input structure
>>> fsinput = fuf_to_featstruct(itext)
>>> print fsinput
[ cat  = 's'                      ]
[                                 ]
[ goal = [ n = [ lex = 'mary' ] ] ]
[                                 ]
[ prot = [ n = [ lex = 'john' ] ] ]
[                                 ]
[ verb = [ v = [ lex = 'link' ] ] ]
```

Figure 37: Unification part I: Convert the input

```
# set up the grammar structure
>>> fsgrammar = fuf_to_featstruct(gtext)
>>> print fsgrammar
[           [     [ cat     = 's'                        ]            ] ]
[           [     [                                      ]            ] ]
[           [     [ goal    = [ cat = 'np' ]             ]            ] ]
[           [     [                                      ]            ] ]
[           [ 1 = [ pattern = (prot, verb, goal)         ]            ] ]
[           [     [                                      ]            ] ]
[           [     [ prot    = [ cat = 'np' ]             ]            ] ]
[           [     [                                      ]            ] ]
[           [     [ verb    = [ cat    = 'vp'          ] ]            ] ]
[           [     [           [ number = {prot number} ] ]            ] ]
[           [                                                         ] ]
[           [     [       [ 1 = [ pattern = (n)   ]              ] ] ] ]
[           [     [       [     [ proper  = 'yes' ]              ] ] ] ]
[           [     [       [                                      ] ] ] ]
[           [     [ alt = [     [ det     = [ cat = 'article' ] ] ] ] ] ]
[           [     [       [     [           [ lex = 'the'     ] ] ] ] ] ]
[           [     [       [ 2 = [                               ] ] ] ] ]
[ alt_top = [ 2 = [       [     [ pattern = (det, n)            ] ] ] ] ]
[           [     [       [     [ proper  = 'no'                ] ] ] ] ]
[           [     [                                                 ] ] ]
[           [     [ cat = 'np'                                      ] ] ]
[           [     [                                                 ] ] ]
[           [     [ n   = [ cat    = 'noun'     ]                  ] ] ]
[           [     [       [ number = {^^number} ]                  ] ] ]
[           [                                                         ] ]
[           [     [ cat     = 'vp'            ]                     ] ]
[           [ 3 = [ pattern = (v)             ]                     ] ]
[           [     [                           ]                     ] ]
[           [     [ v       = [ cat = 'verb' ] ]                    ] ]
[           [                                                         ] ]
[           [ 4 = [ cat = 'noun' ]                                    ] ]
[           [                                                         ] ]
[           [ 5 = [ cat = 'verb' ]                                    ] ]
[           [                                                         ] ]
[           [ 6 = [ cat = 'article' ]                                 ] ]
```

Figure 38: Unification part II: Convert the grammar

```
# unify the input and the grammar
>>> fuf = Unifier(fsinput, fsgrammar)
>>> result = fuf.unify()
# show the result
>>> print result
[ cat     = 's'                                  ]
[                                                ]
[           [ cat     = 'np'                 ] ]
[           [                                  ] ]
[           [           [ cat    = 'noun' ] ] ]
[           [ n       = [ lex    = 'mary' ] ] ]
[ goal    = [           [ number = ?x2    ] ] ]
[           [                                  ] ]
[           [ number  = ?x2                  ] ]
[           [ pattern = (n)                  ] ]
[           [ proper  = 'yes'                ] ]
[                                                ]
[ pattern = (prot, verb, goal)                   ]
[                                                ]
[           [ cat     = 'np'                 ] ]
[           [                                  ] ]
[           [           [ cat    = 'noun' ] ] ]
[           [ n       = [ lex    = 'john' ] ] ]
[ prot    = [           [ number = ?x1    ] ] ]
[           [                                  ] ]
[           [ number  = ?x1                  ] ]
[           [ pattern = (n)                  ] ]
[           [ proper  = 'yes'                ] ]
[                                                ]
[           [ cat     = 'vp'             ]    ]
[           [ number  = ?x1              ]    ]
[ verb    = [ pattern = (v)              ]    ]
[           [                            ]    ]
[           [ v       = [ cat = 'verb' ] ]    ]
[           [           [ lex = 'link' ] ]    ]
```

Figure 39: Unification part III: Result

4.3 linearizer.py & morphology.py or Reaching the Final Output

As previously mentioned the linearizer produces the final output given the result of the unification. The Python code for this is in `nltk.fuf.linearizer`, specifically the `linearize` function. The process for linearization has been previously described in section 3.3.

Packaged alongside the `morphology` and `linearizer` modules is a lexicon module. It contains predefined sets of irregular plurals, verbs as well as pronoun cases and comparatives. This dictionary is used by the `morphology` module to return the correct string to the `linearizer`.

The `nltk.fuf.morphology` module must be given special attention as prior to the work on `nltk.fuf` NLTK did not contain a morphology system. The current implementation is a direct port of the LISP code in the FUF implementation. This system defines a useful application programming interface (API) that can be used for the development of alterna-

tive morphology systems in NLTK. The categories handled by the `nltk.fuf.morphology` module are the same ones handled by FUF. For a complete specification the reader if invited to consult the FUF manual. The API outlined in figure 40 is the current morphology implementation used in `nltk.fuf.mophology`. Note that the functions shown here are equivalent to abstract functions in languages such as Java. The `nltk.fuf.morphology` contains both the abstract definitions and their concrete implementations.

```
def pluralize(word):
    raise NotImplementedError()

def form_ing(word):
    raise NotImplementedError()

def form_past(word):
    raise NotImplementedError()

def form_present_verb(word, number, person):
    raise NotImplementedError()

def morph_fraction(lex, num, den, digit):
    raise NotImplementedError()

def morph_be(number, person, tense):
    raise NotImplementedError()

def morph_verb(word, ending, number, person, tense):
    raise NotImplementedError()

def form_adj(word, ending):
    raise NotImplementedError()

def morph_adj(word, superlative, comparative, inflection):
    raise NotImplementedError()

def morph_pronoun(lex, pronoun_type, case, gender,
                  number, distance, animate,
                  person, restrictive):
    raise NotImplementedError()

def morph_number(word, number):
    raise NotImplementedError()

def morph_number(word, number):
    raise NotImplementedError()
```

Figure 40: Morphology API

Finally, the `linearizer` returns the generated sentence which concludes the work done by `nltk.fuf`.

5 Conclusion and Future Work

In conclusion, the result of this work is two-fold. First and foremost, with with inclusion of `fuf` module NLTK becomes a more fully-featured natural language processing library. It can be used for both research and education. Second, prior to the development of `nltk.fuf` the NLTK library did not include any kind of morphology module. The author hopes that the new morphology API will lead to the development of new and exciting additions to NLTK.

At the same time, there are a number of improvements and features that can be made and added to the `nltk.fuf` library.

- The index annotation in the alt feature is largely ignored at the moment. In LISP FUF this is used to control the number of the possible paths through grammar. While this information is available in `nltk.fuf` it is not used at the moment.
- It should be possible to do the unification without unpacking all the alt paths through the grammar. (ie, expand the `alt` a little bit and go from there).
- Changing the alt handling requires changes to link resolution.
- During parsing all the comments and tracing calls are removed. It would be nice to be able to enable tracing all stages of processing.
- There are utility functions defined to control backtracking during unification and to reduce the computational complexity of the unification algorithm.
- Currently there is no implementation of dependency directed backtracking.

References

[1] Martin Kay. Functional unification grammar: a formalism for machine translation. In *ACL-22: Proceedings of the 10th International Conference on Computational Linguistics and 22nd annual meeting on Association for Computational Linguistics*, pages 75–78, Morristown, NJ, USA, 1984. Association for Computational Linguistics.

[2] Robert Dale Ehud Reiter. *Building Natural Language Generation Systems*. Studies in Natural Language Processing. Cambridge University Press, 2000.

[3] James H. Martin Keith Vander Linden, Daniel Jurafsky. *SPEECH and LANGUAGE PROCESSING: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, first edition, 2000.

[4] Mark Johnson. *Attribute-Value Logic and the Theory of Grammar*. Number 16 in Lecture Notes. CSLI, 1988.

[5] Thomas A. Sudkamp. *Languages and Machines An Introduction to the Theory of Computer Science*. Addison-Wesley, third edition edition, 2006.

[6] Stuart M. Sheiber. *An Introduction to Unification-Based Approaches to Grammar*. Number 4 in Lecture Notes. CSLI, 1986.

[7] Michael Elhadad. Fuf manual. Technical report, Dept of Mathematics and Computer Science, Ben Gurion University, 03 1996.

[8] Jacques Robin Michael Elhadad. Surge: a comprehensive plug-in syntactic realization component for text generation. Technical report, Dept of Mathematics and Computer Science, Ben Gurion University, 03 1996.