JSLIM – Computational Morphology in the Framework of the SLIM Theory of Language

Johannes Handl, Besim Kabashi, Thomas Proisl, and Carsten Weber

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Germanistik und Komparatistik
Professur für Computerlinguistik
Bismarckstr. 6, 91054 Erlangen
{jshandl,kabashi,tsproisl,cnweber}@linguistik.uni-erlangen.de

Abstract. JSLIM is a software system for writing grammars in accordance with the SLIM theory of language. Written in Java, it is designed to facilitate the coding of grammars for morphology as well as for syntax and semantics. This paper describes the system with a focus on morphology. We show how the system works, the evolution from previous versions, and how the rules for word form recognition can be used also for word form generation.[1] The first section starts with a basic description of the functionality of a Left Associative Grammar (LAG) and provides an algebraic definition of a JSLIM grammar. The second section deals with the new concepts of JSLIM in comparison with earlier implementations. The third section describes the format of the grammar files, i. e. of the lexicon, of the rules and of the variables. The fourth section broaches the subject of the reversibility of grammar rules with the aim of an automatic word form production without any additional rule system. We conclude with an outlook on current and future developments.

Introduction

The NLP system JSLIM is the latest in a sequences of implementations within the framework of the SLIM[2] theory of language, which was introduced in [2]. The theory models the cycle of natural language communication, consisting of the hearer mode, the think mode, and the speaker mode. Providing the basis for human-machine communication, it uses the data structure of flat (non-recursive) feature structures called proplets and the time-linear algorithm of Left Associative Grammar (LAG)[3].

[1] The aim of the paper is to give a gentle introduction to the way grammars are written in JSLIM. The more technical aspects of the system, i. e. its scalability, the time required for analysis and generation, an evaluation of the system based on corpora, and a comparison of JSLIM to other existing systems, are the topic of a forthcoming paper.

[2] SLIM is an acronym for **S**urface compositional **L**inear **I**nternal **M**atching, i. e. for the basic principles on which the theory is based [1, p. 30].

C. Mahlow and M. Piotrowski (Eds.): SFCM 2009, CCIS 41, pp. 10–27, 2009.

JSLIM builds on the experiences made in the earlier systems Malaga and JLAG.[3] The present implementation is designed to free the grammar writer from all unnecessary work, thus simplifying development and upscaling. This is achieved by a declarative syntax for writing the rules. Implicit category value checks render the explicit insertion/deletion of values by means of imperative statements obsolete.

Currently, rules can be applied at different levels of the grammar. This allows an easier encapsulation of paradigmatic morphologic phenomena, e. g. inflection. The lexicon entries are flat, i. e. the nesting of feature structures as in Malaga is not permitted, though nesting can be simulated by means of symbolic references. These changes allow the extensive use of templates which considerably reduce the size of the lexicon and which improve its performance, readability and maintainability. The declarative syntax for inflectional and derivational combinations is bidirectional in that it may be used not only for the analysis of inflectional and derivational forms, but also for their generation.

1 Foundations

This section briefly describes the fundamental principles on which the system is built. First we explain the way an LAG works and on which principles it is based. Then we give the algebraic definition of a JSLIM grammar, because it works slightly different from the definition of an LAG given in [2, pp. 186–187].

1.1 LAG and SLIM

> Grammar does not tell us how language must be constructed in order to fulfil its purpose, in order to have such-and-such an effect on human beings. It only describes and in no way explains the use of signs. [6, p. 138]

The purpose of a grammar is therefore not to understand language but to find means by which to describe it. One of the most evident properties of natural language is its linearity in time. Therefore, LA-grammar uses a strictly time-linear derivation order.[4] [3] describes LAG as a bottom-up left-associative parsing scheme as illustrated in figure 1. In each derivation step, the *sentence start*, i. e. the part of the input already parsed, is combined with the *next word*, which in the case of morphological analysis corresponds to the next allomorph of the input. The number of combination steps required to successfully parse an input therefore corresponds to the number of allomorphs of the input minus one.

[3] The system Malaga [4] is quite elaborate and allows the construction of a grammar. Written in C, it provides a language for grammar rules and a single lexicon with nested feature structures. However, the style of the code requires programming experience. Also, the complexity of the lexicon entries makes the grammar less readable and upscaling more difficult. The next system, called JLAG, was implemented by [5] as an attempt to apply the paradigm of object-oriented programming to NLP. Coding became even more difficult and it never left the prototype state. As in Malaga the main problem was that the grammar writer was burdened with too much work and too little work was done by the system.

[4] It is the topic of the SLIM theory of language to model a real understanding of language.

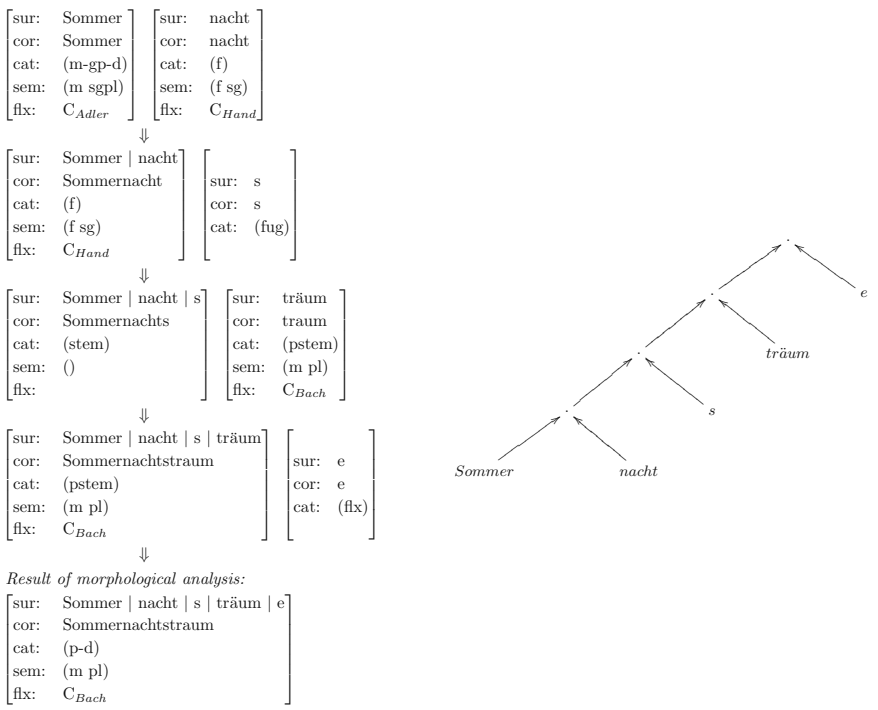

Fig. 1. The bottom-up left associative analysis and derivation order of an LA-grammar

1.2 The Algebraic Description of a JSLIM Grammar

According to [7, p. 4] a JSLIM grammar G' is defined as an 8-tuple

$$(W', A', C', LX', CO', RP', ST'_S, ST'_F)$$

Thereby

1. W' is a finite set of surfaces
2. A' is a finite set of attribute names
3. C' is a finite set of category segments
4. $LX' \subset W' \times (A' \times C'^*)^+$ is a finite set called the lexicon
5. $CO' = (co'_0, \ldots co'_{n-1})$ is a finite set of total recursive functions called categorical operations of the form

$$\mathscr{P}(FS) \times FS \rightarrow \mathscr{P}(FS) \tag{1}$$

Thereby, $FS \subset (A' \times C'^*)^+$ and $\mathscr{P}(FS)$ is the function to generate the power set for a given set. Let $co \in CO'$ and $co(\{fs_0, \ldots, fs_{n-1}\}, fs_n) \in \mathscr{P}(FS)$ with $fs_i \in FS$ for $i \leq 0 < n$. Let $FS_c = \{M \subseteq \mathscr{P}(FS) : |M| = c\}$. There exists an $m < n, k < n$ with

$$co\left(\bigcup_{i=0}^{k-1}\{fs_i\}\cup\bigcup_{i=k}^{n-1}\{fs_i\},fs_n\right)=\bigcup_{i=0}^{m-1}\{fs'_i\}\cup\bigcup_{i=k}^{n-1}\{fs_i\} \tag{2}$$

$$0\leq k<c_k=const \tag{3}$$

$$1\leq m\leq k+1 \tag{4}$$

$$\forall(M\in FS_{n-1-k})\ :\ co(\bigcup_{i=0}^{k-1}\{fs_i\}\cup M,fs_n)=\bigcup_{i=0}^{m-1}\{fs'_i\}\cup M \tag{5}$$

where $fs'_i \in FS$ for $i \leq 0 < m$.

6. $RP' = (rp'_0 \ldots rp'_{n-1})$ is a sequence of the same length with $rp'_i \subseteq \{i | 0 \leq i < n\}$, called rule packages
7. $ST'_S \subseteq (A' \times C'^*)^* \times RP'^*$ is a finite set called start states
8. $ST'_F \subseteq (A' \times C'^*)^* \times RP'^*$ is a finite set called final states

The definition is in accordance with the definition given in [2, p. 187]. However, feature structures are used instead of a category list to represent *sentence start* and *next word*. An LAG which works on feature structures was already defined in [8, pp. 37–38], but the concept of categorical operations has changed. One of the recent innovations regarding LAGs was to postulate a sentence start which is no longer coded in the form of a hierarchical data structure, but in the form of a set of flat feature structures. As a consequence, it is possible to access single values not only by using the underlying hierarchy of the data structure but also by using various – mainly syntactic-semantic – relations between the feature structures.[5]

Although those changes are fundamental as far as syntax is concerned they have little impact on morphological analysis where the sentence start is still coded into one single feature structure. I. e. in morphology we have the special case that both sets, *sentence start* and *resulting set*, always have a cardinality of one.

2 Applied Techniques

In this section, the newer concepts of JSLIM are presented. One of those concepts is the idea of *undirected programming*.[6] The idea is to specify rules as bijective functions, so that they can be executed in either direction. Here we only briefly cover the declarative syntax used in JSLIM, but we will go into more detail in section 4. We then investigate the techniques of *indirection*[7] and *common subtree sharing*, as they have an impact on

[5] This facilitates the way rules can be written and provides the grammar developer with new means of how to express constraints in the rules. [9] showed that rules for sentences with gapping and coordination can be modelled more accurately by exploiting word order.

[6] The most famous example of a programming language which allows undirected programming is Prolog. It is widely known among linguists for allowing easy coding of natural language grammars by using the definite clause grammar (DCG) notation. Although it is not the intention of the developers to create a new logic programming language, Prolog had a certain influence on the design of the rule syntax.

[7] Indirection has been widely used before, e. g. for constraint parsers, but never in combination with an LAG.

the storage and design of the lexicon.[8] The technique of *common subtree sharing* is also applied to the parsing process, e. g. for the internal representation of the parsing state.

2.1 Undirected Programming and Declarative Syntax

In JSLIM, a declarative syntax can be used to code inflection. Listing 1 illustrates the declension *table* of the German noun 'Bach' (creek) as it is coded in JSLIM. A table definition starts with the keyword `table` and the table name. The first letter of the table name must be an upper case letter. The table name is followed by a colon and the signature of the table. The signature defines, which attributes[9] of the combined feature structures are changed by one of the combination rules defined within the table body. The body comprises several rows with implication arrows. The left side of the arrow specifies the category values of *sentence start* and *next word*, whereas the right side specifies the resulting category values. The signature helps to investigate the attributes and the feature structure to which the values belong. To avoid the repetitive specification of the same category values for the *sentence start*, a semicolon can be put at the end of the last combination definition to indicate that the next definition will reuse the missing values from the current. A full stop, in contrast, indicates that the category values of the current definition are not reused.

```
table C_Bach: [cat,sem]      [sur] => [cat,sem]
              (m-g)  (m sg)   es   => (mg) (m sg)   ; # Baches
                              s    => (mg) (m sg)   ; # Bachs
                              e    => (md) (m sg)   . # Bache
              (pstem)(m pl)   e    => (p-d)(m pl)   . # Bäche
              (p-d)  (m pl)   n    => (pd) (m pl)   . # Bächen
```

Listing 1. A morphological rule in JSLIM

Hence, the above table definition defines the following combination steps illustrated in figure 2. This definition is declarative as it does not enforce by any means in what way the modifications have to be performed, but merely describes them.

2.2 Indirection

When developing a morphology for a natural language one of the first tasks of the developer is to somehow code the conjugation and declension tables. Although it is quite a simple task for an experienced traditional linguist to do so on a sheet of paper, it is not so clear at first sight how to perform this task within the framework of Left Associative

[8] The techniques of common subtree sharing, DAGs and suffix trees have been frequently used to reduce the size of lexica, inter alia in the field of chart parsers. The elegance of the approach of using templates as presented here is that the grammar developer benefits directly from the compact storage, as the continuous support through all the stages of parsing eases the building and maintenance of the lexicon.

[9] A description of the used attributes can be found in [1, p. 335]. The values of the attributes are coded using a distinctive (instead of an exhaustive) categorization [2, p. 244] [1, pp. 335–337].

$$\begin{bmatrix}\text{cat: (m-g)}\\ \text{sem: (m sg)}\end{bmatrix} \circ \begin{bmatrix}\text{sur: es}\end{bmatrix} \Rightarrow \begin{bmatrix}\text{cat: (mg)}\\ \text{sem: (m sg)}\end{bmatrix} \quad \text{Baches}$$

$$\begin{bmatrix}\text{cat: (m-g)}\\ \text{sem: (m sg)}\end{bmatrix} \circ \begin{bmatrix}\text{sur: s}\end{bmatrix} \Rightarrow \begin{bmatrix}\text{cat: (mg)}\\ \text{sem: (m sg)}\end{bmatrix} \quad \text{Bachs}$$

$$\begin{bmatrix}\text{cat: (m-g)}\\ \text{sem: (m sg)}\end{bmatrix} \circ \begin{bmatrix}\text{sur: e}\end{bmatrix} \Rightarrow \begin{bmatrix}\text{cat: (md)}\\ \text{sem: (m sg)}\end{bmatrix} \quad \text{Bache}$$

$$\begin{bmatrix}\text{cat: (pstem)}\\ \text{sem: (m pl)}\end{bmatrix} \circ \begin{bmatrix}\text{sur: e}\end{bmatrix} \Rightarrow \begin{bmatrix}\text{cat: (p-d)}\\ \text{sem: (m pl)}\end{bmatrix} \quad \text{Bäche}$$

$$\begin{bmatrix}\text{cat: (p-d)}\\ \text{sem: (m pl)}\end{bmatrix} \circ \begin{bmatrix}\text{sur: n}\end{bmatrix} \Rightarrow \begin{bmatrix}\text{cat: (pd)}\\ \text{sem: (m pl)}\end{bmatrix} \quad \text{Bächen}$$

Fig. 2. A morphological rule in JSLIM

Grammar. An inflectional form is a combination of a stem with an inflectional affix. To restrict possible combinations of stems and flexives, the agreement conditions have to be coded into the category values of the combined parts. There are four possibilities to achieve this.

1. A *naive approach* would be to store a paradigm feature flag in the stem as well as in the inflectional suffix. This would reduce the inflectional check to a simple agreement check based on those two flags. The disadvantage of this approach is that, though agreement checks could be realized very efficiently, e. g. via a bitmap, the suffixes would become rather complicated, as suffixes are normally used in more than one paradigm, and difficult to code and to maintain. E. g. changing the paradigm feature of one paradigm also requires the paradigm feature flag of the inflectional endings of this paradigm to be changed which may inflict side-effects on other paradigms. Creating suffix entries for each paradigm is also no satisfactory solution, as this has a negative impact on the runtime.
2. The *stem approach* tries to restrict the combination of stem and inflectional affix mainly by focusing on the category of the stem. The disadvantage of this approach is obvious. As inflectional properties of the whole paradigm are coded into every single lexicon entry, storage is extremely redundant.
3. The *affix approach* tackles the problem from the other end. Instead of storing the inflectional properties of the paradigm in the stems, those properties are coded into the affixes. This avoids redundant storage of information but unnecessarily complicates the affix entries. A possible solution would be to multiply the lexicon entries of the affixes to simplify their categories. However, this has a noticeable negative impact on the run-time, as all these lexicon entries must be matched for a single combination with this affix.
4. Like the *stem approach*, the *indirection approach* accesses inflectional properties of the paradigm via the stem. However, the *indirection approach* is an improvement over the *stem approach* in so far as the information is only referenced, but not stored directly. The information itself is coded into a table, which is defined externally.

JSLIM supports all three approaches. However, the grammar developer is encouraged to use the *indirection approach*. In the next paragraph that approach is described in more detail.

$$\begin{bmatrix} \text{sur:} & \text{Bach} \\ \text{cat:} & \text{(m-g)} \\ \text{sem:} & \text{(m sg)} \\ \text{flx:} & C_{Bach} \end{bmatrix} \begin{bmatrix} \text{sur:} & \text{Bäch} \\ \text{cat:} & \text{(pstem)} \\ \text{sem:} & \text{(m pl)} \\ \text{flx:} & C_{Bach} \end{bmatrix}$$

Fig. 3. Allomorphs of the German noun 'Bach' (creek)

We code the allomorphs of a word as illustrated in figure 3. The lexicon entries contain an attribute for the surface (*sur*), attributes for the syntactic and semantic categories (*cat* and *sem*), and an attribute for inflection (*flx*). The value of the latter is a reference to the table defined earlier in listing 1. When the parser tries to combine the stem with an inflectional affix,[10] this trial is redirected to the table C_{Bach} which can be accessed via the attribute *flx*. Stem and inflectional affix become the so-called arguments of the table, i. e. a lookup in the table is performed, if an applicable combination rule can be found. In the case of a successful lookup the combination rule is applied, otherwise the table lookup fails and, as a consequence, the rule application also fails. Therefore, it is much easier to handle inflection by means of indirection, as matching is reduced to a simple test of whether the inflectional affix is allowed in the paradigm. No complex category checks are required.

2.3 Common Subtree Sharing

Reducing the memory needed by a system can be critical when trying to improve its performance. If a simple value requires much space and has to be stored multiple times, the easiest method to reduce the amount of memory needed is to store the value only once and then reference it. Though this may drastically reduce the amount of memory needed, still much memory is wasted for the pointers. Better results can be achieved by applying this principle of single storage to compound values. As illustrated in figure 4 and figure 5, most of the data structures used in JSLIM can be regarded as directed acyclic graphs. Those data structures bear a close resemblance to trees, with few nodes at the upper layers and almost all the nodes at the bottom layers. Besides, the co-domain of the values at the lower levels is quite restricted.[11] Therefore, the technique of common subtree sharing can be applied.

To allow the **sharing of common values in the lexicon**, values can be marked explicitly in the base lexicon and in the allo[12] lexicon. We will go into more detail in the next section. The notation reflects the way lexicon entries are stored internally. A lexicon entry is stored in the form of a 3-tuple which consists of the surface, the base form

[10] Inflectional affixes can easily be marked as such by an adequate category value.

[11] An exception are the values of the surface and base form attributes, which of course are specific.

[12] Following a tradition of former systems, JSLIM uses an allo lexicon and allo rules [10].

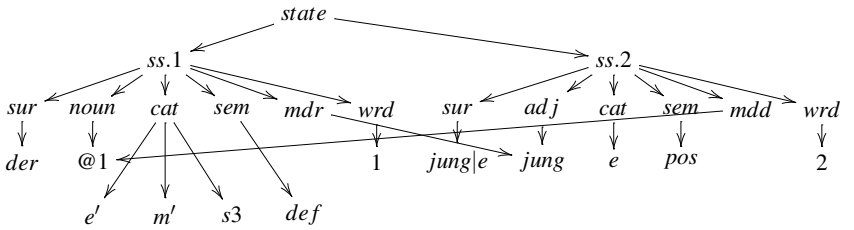

Fig. 4. Inclusion hierarchy of a parser, cf. [7, p. 55]

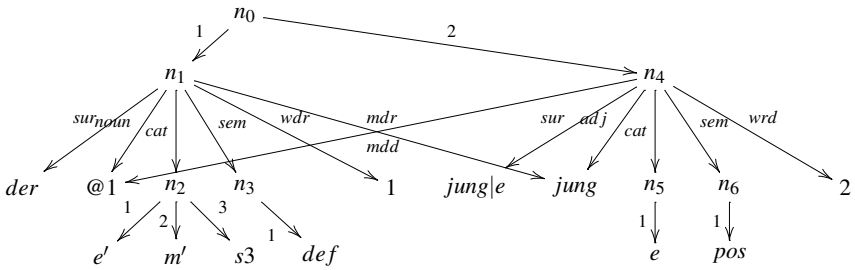

Fig. 5. Graph of a parsing state, cf. [7, p. 55]

$$\begin{bmatrix} \text{sur}: & \text{der} \\ \text{noun}: & @1 \\ \text{cat}: & (\text{e' m' s3}) \\ \text{sem}: & (\text{def}) \\ \text{mdr}: & (\text{jung}) \\ \text{wrd}: & 1 \end{bmatrix} \begin{bmatrix} \text{sur}: & \text{jung | e} \\ \text{adj}: & \text{jung} \\ \text{cat}: & (\text{e}) \\ \text{sem}: & (\text{pos}) \\ \text{mdd}: & @1 \\ \text{wrd}: & 2 \end{bmatrix} + \begin{bmatrix} \text{sur}: & \text{Mann} \\ \text{noun}: & \text{Mann} \\ \text{cat}: & (\text{m-g}) \\ \text{sem}: & (\text{m sg}) \\ \text{mdr}: & () \\ \text{wrd}: & 3 \end{bmatrix} \rightarrow \begin{bmatrix} \text{sur}: & \text{der} \\ \text{noun}: & \text{Mann} \\ \text{cat}: & (\text{s3}) \\ \text{sem}: & (\text{m sg def}) \\ \text{mdd}: & (\text{jung}) \\ \text{wrd}: & 1 \end{bmatrix} \begin{bmatrix} \text{sur}: & \text{jung | e} \\ \text{adj}: & \text{jung} \\ \text{cat}: & (\text{e}) \\ \text{sem}: & (\text{pos}) \\ \text{mdd}: & \text{Mann} \\ \text{wrd}: & 2 \end{bmatrix}$$

Fig. 6. Parser state when executing DET+N, cf. [7, p. 54]

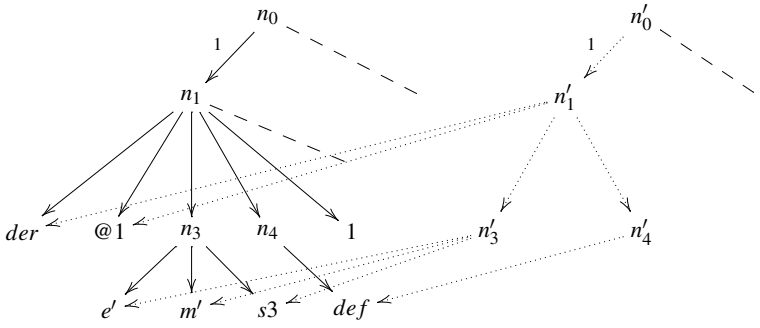

Fig. 7. Flat copy of a parser state, cf. [7, p. 56]

and a reference to a template which contains the additional values, normally shared with other entries.[13]

Sharing of common values of parser states is advisable, in so far as during a derivation ambiguity arises in (almost) every parsing algorithm from time to time. Ambiguities are normally handled by creating a new branch for each reading. The drawback of this approach is that it might not be possible to copy the temporary result of the derivation in constant time if it is coded in the current state. This, however, can be guaranteed if just flat copies are created and the depth of the graph is limited by a constant. However, the parser has to take care that no side effects are created. An example of copying a state in constant time is shown in figure 7. All values which will be modified by the action described in figure 6 are copied.

3 The JSLIM Grammar Files

In this section, the grammar files of a JSLIM grammar are briefly described. The focus lies on showing the differences to earlier and different implementations.[14]

3.1 The Lexicon

There are several ways of coding lexicon entries due to the following reasons:

- Depending on the state of a project the priorities may vary. While developing a grammar, the main focus might be the fast creation of a lexicon. Later on, it will probably shift to the readability, maintainability and space efficiency of the lexicon.
- Normally, a lexicon is not constructed by hand but by scripts which migrate existing lexica, or fill a lexicon with data extracted from corpora. Depending on the structure of the original data, conversion into one format might be easier to accomplish than into another possible format. And as long as the different formats are easily interchangeable, there is no reason why to restrict the lexicon to a single format.
- Normally, constructing a morphology component is a bottom-up process. First, a lexicon with a representative of each paradigm is needed to test inflection. Then, the lexicon is filled. Therefore it may be easiest, to start with a rather simple lexicon as long as it can also be automatically converted into a template based lexicon.

The different types of lexicon entries are described below.

Plain old lexicon entries are the simplest kind of lexicon entries. They are mainly used for lexicon prototyping. Their structure reminds us of the style used in [13]. All entries are coded separately. Although at first sight this might seem to be the best and easiest choice, plain old lexicon entries render the lexicon definition highly redundant as common values are not shared. An example for plain old lexicon entries is given in listing 2.

```
[sur: lern, cor: lernen, cat: (n' v), sem: (pres), ...]
```

Listing 2. A plain old lexicon entry for 'lernen' (to learn)

[13] By applying that technique, the amount of required memory can be considerably reduced, as was shown by the Italian morphology implemented by [11].

[14] A more detailed explanation of the grammar files can be found in [12].

The purpose of the **instance notation** is to avoid the above mentioned redundancy of categorical value. This aim is achieved with the help of templates. A template looks like an ordinary feature structure but is prefixed by the string `!template`. A feature structure can be marked as an instance of the last defined template by prefixing it with `!+`. A template followed by three instances is illustrated in listing 3. It is possible to override attribute values specified in the template by specifying them again in an instance. This style may be helpful for small to medium sized lexica, but only to a lesser extent for very large lexica.

```
!template[cat: (n' v), sem: (pres), ...]
!+[sur: lern,   cor: lernen]
!+[sur: erb,    cor: erben]
!+[sur: schenk, cor: schenken] ...
```

Listing 3. The instance notation for 'lernen', 'erben' (to inherit), 'schenken' (to make a gift)

The **sequence notation** can be used if feature structures differ in only one attribute value. Instead of specifying instances, the attribute to be added and the list of corresponding values are specified. For each value in the list an entry is added to the lexicon consisting of all the values specified by the last template in addition to the indicated attribute value pair. This style, however, is only possible if the entries belonging to a certain template differ in only one attribute value. This may be the case for a base form lexicon, but certainly not for the allo lexicon. In the latter, surface and base form attributes are word form specific and will differ in the majority of cases.

```
!template[flx: C_lernen, all: A_lernen]
!+sur cor: lernen erben ...
```

Listing 4. The sequence notation

A variant of the sequence notation is the **regexp notation**. It can be used in the case of instances differing in more than one attribute while all the attribute values are derivable from one value by means of regular expressions. This notation allows attribute names to be followed by regular expressions (see listing 5).

```
!template[cat: (n' a' v), sem: (pres), ...]
!+sur /(.+)en/\$1/ cor: lernen erben ...
```

Listing 5. The regexp notation

The **column notation** is the most compact one. It avoids the repeated declaration of attribute names and thus not only reduces the lexicon size but also increases processing speed: This kind of entry can be read in very efficiently as number and type of the attributes to be added is known in advance. Therefore, this is the preferred style for very large lexica.

```
!template[cat: (n' a' v), sem: (pres), ...]
![sur cor]
lern lernen
erb erben ...
```

Listing 6. The column notation

3.2 The Allomorph Method

Like the previous implementations of the LAG system (see [10, pp. 103–104]) JSLIM uses the allomorph method presented in [14, pp. 255–256]:

> The allomorph method uses two lexica, called the elementary lexicon and the allomorph lexicon, whereby the latter is automatically derived from the former by means of allo-rules before run-time. [...] During run-time, the allomorphs of the allomorph lexicon are available as precomputed, fully analyzed forms [...], providing the basis for a maximally simple segmentation: the unknown surface is matched from left to right with suitable allomorphs - without any reduction to morphemes.

Hence, the lexicon of the base forms coded in the above described notations merely serves as input for the allo rules to create the allo lexicon. Though, we will show in the next section, that the structure of the base form lexicon is preserved within the allo lexicon.

3.3 Allo Rules

Allo rules are coded in a declarative manner using the afore mentioned table notation (see section 2.1) and indirection. I. e. a reference to the allo rule is coded within the lexicon entry of the base form. Listing 7 shows the allo rule for the German noun ‘Bach’. Allo rules are executed before run-time to create the allo lexicon [10]. The approach presented here is an improvement over earlier systems in which the allomorphs of a base form had to be generated by applying all allo rules.[15]

```
!template[cat: (n' a' v), sem: (pres), ...]
![sur cor]
lern lernen
erb erben ...
table A_Bach: [cor]              => [sur,cor,cat,sem]
/(.*?)([AOUaou])(u?[^aouäöü]*)/ => /$0/ /$0/ (m-g) (m sg) ;
                                 => /$1$2"$3/ /$0/ (pstem) (m pl) .
```

Listing 7. Allo rule for the German noun ‘Bach’

3.4 Allo Lexicon

In previous implementations the template structure of the lexicon files could not be maintained by the allo generator. The allo generator expanded the templates and the corresponding template instances. The output of the generator was a sequence of complete feature structures. Hence, templates were merely used to ease the coding of lexicon entries. This way of proceeding, however, is inefficient a) as far as the execution of the allo rules is concerned and b) with respect to further parser passes, e. g. the loading of the allo lexicon. An advantage of storing the allo table as an attribute value is that the allo rules which are needed can be called directly from the feature structure of the

[15] The generated allomorphs still contain a reference to the allo table. The purpose of this will be explained in section 4.

base form (indirection). It is therefore clear before run-time which allo rules have to be applied to which lexeme. What is more, the expansion of the templates can be avoided. The allo generator processes templates as follows: If a template contains an attribute *all*, the semantics of the allo rule is split into two parts: One affects the template and the other merely affects the template instance. The result of the generation therefore consists of one or more modified templates, one for every allo rule, each followed by a sequence of (probably) modified instances (see listing 8).

```
!+[all: A_Bach, flx: C_Bach]
!+cor: Aalkorb Abbrand Abbruch Abdampf ... Bach ...

                        ⇓ ⇓ ⇓

!+[all: A_Bach, flx: C_Bach, cat: (m-g), sem: (m sg)]
!+sur cor: Aalkorb Abbrand Abbruch Abdampf ... Bach ...

!+[all: A_Bach, flx: C_Bach, cat: (pstem), sem: (m pl)]
![sur cor]
Aalkörb Aalkorb
Abbränd Abbrand
Abbrüch Abbruch
Abdämpf Abdampf
...
Bäch Bach
...
```

Listing 8. Generation of the allo lexicon

3.5 Combi Rules

Combi rules define when two allomorphs can be combined. Normally, combi rules delegate most of the work to the tables which are referenced by the combined parts.[16] Figure 9 illustrates the rule used for inflection. A rule starts with its rule name, here STEM+FLX, followed by its rule package, i. e. by the set of follow-up rules. The *sentence start pattern* asserts that the *sentence start* is a stem. This is enforced by the declaration of the attribute *flx*. That the *next word* is an inflectional affix is ensured by the category value `(flx)` in the *next word pattern*. The value `*F` of the attribute `flx` triggers a table look-up with the feature structure which matches the sentence start and the feature structures which matches the next word as its argument.

```
STEM+FLX {STEM+FLX}

[cat: _, flx: *F] [cat: (flx)] => [...] [-]
```

Listing 9. Morphology rule for inflection

The rule is only applied, if the table look-up succeeds. The result of the look-up defines the way the two features structures are changed. The result patterns after the

[16] This is an example of indirection as described in section 2.2.

implication arrow can also be used to specify the way the two feature structures are changed. The pattern `[...]` indicates, that the features structure which matches the sentence start pattern remains unchanged within the sentence start. The pattern `[-]` means, that the feature structure which matches the next word pattern is excluded from the resulting sentence start.

3.6 Variables

For the patterns to become more abstract, variables can be used. For example, the rule in listing 9 contains the predefined anonymous variable _, which can be the placeholder for an arbitrary value. A variable can have a predefined data type and a co-domain as is illustrated in figure 10. The variable `CAT` has the data type `string` and may be bound to the values `{s1', s13', ...}`.[17]

```
string CAT <- {s1' s13' s2' s3' s3p2' p13' p2' m-g mg md ...}
```

Listing 10. Definition of the variable `CAT`

4 Word Form Generation

Word form generation deals with the creation of word forms from various inputs, be it a direct human request or the parameter values of the internal and external sensors of a robot. The application spectrum ranges from providing suggestions within a spell checker to the creation of a speaking robot. It is our goal to reuse the rules used for word form analysis for word form generation. This requires that the allowed combinations of lexicon entries are defined as a bijective function. The tables used in JSLIM can be seen as bijective functions in so far as they map a sequence of attribute values to another sequence of attribute values and are therefore reversible. If inflection is coded by means of tables, the code for the analysis can also be used for generation. As there is ongoing research in this field [15]. we cannot present a complete word form generation system with conceptualization and linearization. But we show that it is possible to generate a surface form for a given lemma and a given category – a process that is sometimes referred to as surface generation.

4.1 Surface Generation

We distinguish two cases of surface generation: a) the generation of a complete paradigm and b) the generation of a single word form. The first case may be of interest to check the correctness of a grammar, whereas the second case is needed for natural language production in the speaker mode.

[17] The grammar developer can also define constraints between pairs of variables. This feature is particularly expedient for specifying the agreement conditions in a syntax but is rarely used in morphology.

4.2 Generation of a Paradigm

To generate a complete paradigm for a base form, only an allomorph of the stem is needed.[18] To generate the complete paradigm the following steps are performed.

1. Look up the lexicon entry of the provided allomorph.
2. The allomorphy of the base form is specified in the attribute *all* of the lexicon entry, the value of which is a reference to a table. Apply that table to the allomorph. The circumstance that the rules in the table are applied to the allomorph and not to the base form, is negligible due to the fact that the application of the rules only depends on the base form of an entry. That base form is the value of the attribute *cor* and is the same for all allomorphs of a given stem. Besides, the category values are all changed by the applied rule, hence their particular values are insignificant.
3. Each allomorph has an attribute *flx* with an inflection table as its value. For each rule in the table, we look up the inflectional affix and execute the combination rule with the stem and the affix as its arguments. As it is possible to combine a stem with more than one inflectional affix, we repeat this step until no rule matches anymore.

We provide an example to illustrate the process of generating all word forms of a paradigm in more detail.

Example. Let ‘geb’ be the allomorph of a word form for which the paradigmatic forms shall be generated. The four allomorphs of the morpheme are ‘geb’ (Präsens), ‘gib’ (Präsens), ‘gab’ (Imperfekt) and ‘gäb’ (Konjunktiv 2). The first trivial step is to perform a lexical look-up for the allomorph ‘geb’, which returns the corresponding feature structure as illustrated in figure 8a.

Listing 11 shows the allo table which is used to create the allomorphs of the morpheme ‘geb’.

```
table A_geben: [cor] => [sur,cat,sem]
       /(.+)e(.+)en/ => /$1e$2/ (n' v)   (pres) ;   # geb
                     => /$1i$2/ (i' v)   (pres) ;   # gib
                     => /$1a$2/ (s13' v) (ipf)  ;   # gab
                     => /$1ä$2/ (n' v)   (k2)   .   # gäb
```

Listing 11. The table for the allomorphy of the German verb ‘geben’ (to give)

The application of the table is shown in figure 8b. The four allomorphs of the morpheme ‘geb’, namely ‘geb’, ‘gib’, ‘gab’ and ‘gäb’, are created on the basis of the allomorph ‘geb’. It is evident from the syntax of the rule that applying the table to different allomorphs of the same morpheme will not change the result, as the attribute values of the resulting feature structures are set by the respective applied rule. Listing 12 shows the inflection table of the generated allomorphs.[19]

[18] The stem morpheme is accessed via the surface of one of its allomorphs as an allo lexicon is used.

[19] Normally, only one inflection table is used for one paradigm. This, however, is solely a design decision and by no means obligatory.

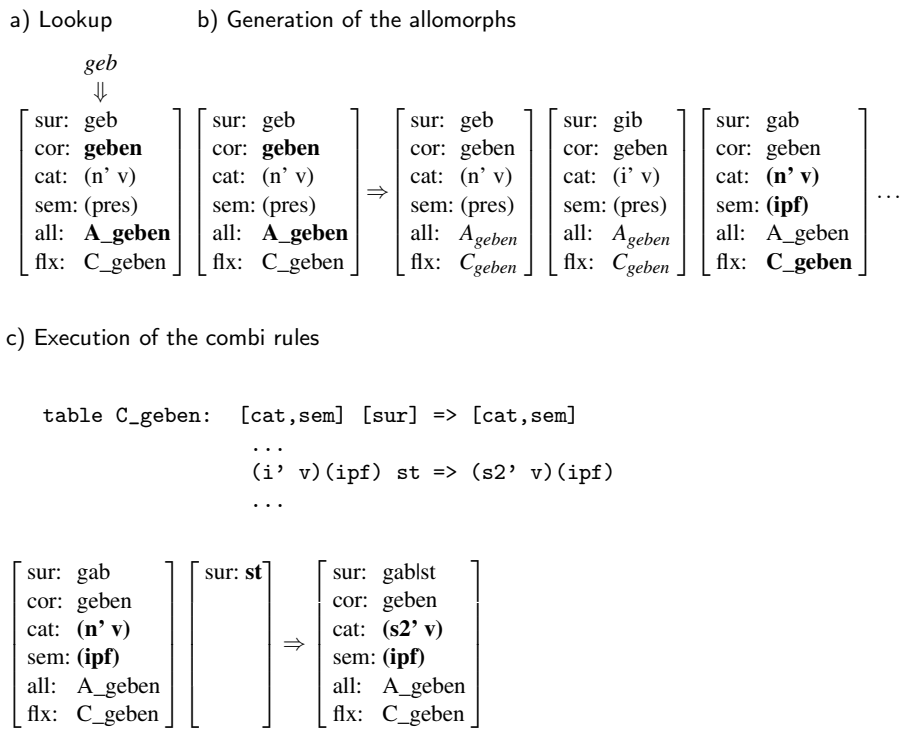

Fig. 8. Generation of a paradigm

```
table C_geben:  [cat,sem]        [sur] => [cat,sem]
                (n' v)  (pres)    e    => (s1' v) (pres) ;    # geb-e
                                  en   => (p13' v)       ;    # geb-en
                                  t    => (p2' v)        .    # geb-t
                (i' v)  (pres)    st   => (s2' v) (pres) ;    # gib-st
                                  t    => (s3' v)        .    # gib-t
                (s13' v)(ipf)     st   => (s2' v)        ;    # gab-st
                                  en   => (p13' v)(ipf)  ;    # gab-en
                                  t    => (p2' v)        .    # gab-t
                (n' v)  (k2)      e    => (s13' v)(k2)   ;    # gäb-e
                                  est  => (s2' v)        ;    # gäb-est
                                  en   => (p13' v)       ;    # gäb-en
                                  t    => (p2' v)        .    # gäb-t
```

Listing 12. The table for the conjugation of the verb 'geben'

For each generated allomorph the attribute values are matched with the left sides of the rules. In case of a match, we append the surface of the inflectional affix to the surface of the allomorph and perform the category changes defined by the rule.

4.3 Conditions for the Approach to Work

For surface generation to work, the following conditions must hold:

1. All categorical values are created by allo rules based on the value of the attribute of the base form.

2. The allo tables are stored as attribute values of the allomorphs. The alternative would be to read in the base form lexicon which would be impractical.
3. The inflection tables are stored as attribute values of the allomorphs. The alternative would be to try all inflection rules on a stem which would be impractical.
4. Allomorphy and inflection are coded only by means of tables. This is necessary as the tables allow different access orders as needed for generation.

These conditions must also hold for generating a single word form. However, they can easily be fulfilled without losing generative capacity and they rather facilitate lexicon design than hamper it. Hence, generation based on the rules for analysis is always possible.

Algorithm 1. generate($\mathscr{A}$,$\mathscr{F}$,v)

```
1: for all a in 𝒜 do
2:    if matches(rightside(a),v) then
3:       return ⟨a⟩
4:    end if
5: end for
6: for all f in ℱ do
7:    if matches(rightside(f,v)) then
8:       if ⊥ ≠ (r = generate(𝒜,ℱ,leftside(f))) then
9:          return r + ⟨a⟩
10:      end if
11:   end if
12: end for
13: return ⊥
```

4.4 Generation of a Single Word Form

If all forms of a paradigm can be generated, it is possible to generate a single form by first generating the paradigm and then using a filter. The obvious disadvantage, however, is a lot of overhead. By using the following algorithm it is possible to use the tables to generate only a single word form of a given category:

1. Indicate the word form to be generated by providing an allomorph of the stem and one or more attribute values of the word form to create, e. g. `geb` and `[cat: (s2' v), sem: (ipf)]`.[20]
2. Look up the lexicon entry for the provided allomorph. This step is exactly the same as when generating a complete paradigm in figure 8a.
3. Determine the set of inflection rules for the allomorphs of the stem. If the allo rule does not change the inflection table (normal case), it can be taken directly from the lexicon entry. Otherwise, it can be investigated by looking at the right side of the rules of an allo table, as this is where they may be set. As the signature of the table `A_geben` in figure 11 does not contain the attribute `flx`, the referenced inflection table will not change when generating the allomorphs of the stem *geb* and can therefore be taken from an arbitrary allomorph.

[20] A filter can be used to allow a more flexible input which map more general user inputs to attribute values used in the paradigm.

4. For all allo rules, check if the result of one of these allo rules matches the provided category values. If it matches, the word form can be generated by applying the allo rule. E. g. if the provided attribute values were `[cat: (s13' v), sem: (ipf)]`, a matching form of *geben* could be created by executing the third row of the table specified in figure 11.
5. If the provided category values do not match any right hand side of an allo rule, check the inflection rules. If the right hand side of one of the inflection rules matches, the word form can be generated by applying that rule. Other inflection rules and at least one allo rule must be applied and the result of those applications must match the left side of the matching inflection rule. Therefore, we continue recursively with step 3, but use the values of the left side of the inflection table instead of the provided category values. E. g. let the specified attribute values of the verb *geben* be `[cat: (s2' v), sem: (ipf)]`. As these values do not occur on the right side of any rule in the table `A_geben`, the desired word form cannot be created by merely applying an allo rule. However, the values would match the right side of the fourth row in the table `C_geben` (cf. figure 12). Hence, we memorize that this rule has to be applied and try to find a way to generate the word form with the category values `[cat: (s13' v), sem: (ipf)]`, i. e. with the values of the left side of the applicable row.

Let $\mathscr{A}$ be the set of allo rules of the word form, $\mathscr{F}$ the set of inflection rules, and v the specified values. Let `leftside` be a function which returns the values of the left side of a combination rule except the next word, and `rightside` be the function which returns the right side of the combination rule. The algorithm is illustrated more formally in Alg. 1.

5 Conclusion

In this article, we presented the JSLIM system of automatic word form recognition and production. It has been shown how the current implementation uses techniques borrowed from computer science, e. g. common subtree sharing, undirected programming, and indirection. The combination of indirection and undirected programming in the form of tables seems to be an elegant approach for handling word form recognition as well as word form generation.

Currently, the morphological system is evaluated on the basis of a multitude of medium-scale grammars for different languages[21] and optimized for an increased performance. Further development will include extensive tests on corpora and a tighter integration of the morphology with the syntactic-semantic components.

References

1. Hausser, R.: A Computational Model of Natural Language Communication: Interpretation, Inference, and Production in Database Semantics. Springer, Heidelberg (2006)
2. Hausser, R.: Foundations of Computational Linguistics. Springer, Heidelberg (1999)

[21] This comprises the building of grammars with an average lexicon size of about 80000–100000 entries for languages like German, Italian, French and Polish.

3. Hausser, R.: Complexity in left-associative grammar. Theoretical Computer Science 106, 283–308 (1992)
4. Beutel, B.: Malaga 7.12. User's and Programmer's Manual. Technical report, Friedrich-Alexander-Universität Erlangen-Nürnberg (1995), http://home.arcor.de/bjoern-beutel/malaga/malaga.pdf (June 4, 2009)
5. Kycia, A.: Implementierung der Datenbanksemantik für die natürlichsprachliche Kommunikation. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2004)
6. Wittgenstein, L.: Philosophical Investigations. Basil Blackwell Ltd., Oxford (1953)
7. Handl, J.: Entwicklung einer abstrakten Maschine zum Parsen von natürlicher Sprache. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2008)
8. Schulze, M.: Ein sprachunabhängiger Ansatz zur Entwicklung deklarativer, robuster LA-Grammatiken mit einer exemplarischen Anwendung auf das Deutsche und das Englische. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2004)
9. Kapfer, J.: Inkrementelles und oberflächenkompositionales Parsen von Koordinationselipsen. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2009)
10. Schüller, G., Lorenz, O.: LA-Morph - Ein linksassoziatives Morphologiesystem. In: Linguistische Verifikation, pp. 103–119 (1994)
11. Weber, C.: Implementierung eines automatischen Wortformerkennungssystems für das Italienische mit dem Programm JSLIM. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2007)
12. Weber, C., Handl, J., Kabashi, B., Proisl, T.: Eine erste Morphologie in JSLIM (in progress). Technical report, Friedrich-Alexander-Universität Erlangen-Nürnberg (2009), http://www.linguistik.uni-erlangen.de/clue/fileadmin/docs/jslim/morphology_docu.pdf (June 4, 2009)
13. Lorenz, O.: Automatische Wortformenerkennung für das Deutsche im Rahmen von Malaga. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (1996)
14. Hausser, R.: Modeling natural language communication in database semantics. In: Proceedings of the APCCM, vol. 96, Australian Computer Science Inc. CIPRIT (2009)
15. Kabashi, B.: Sprachproduktion im Rahmen der SLIM-Sprachtheorie. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (in progress, presumably 2009)