The Linguist's Guide to PLAIN – Part 1

Drawing up a Morphological Component

Peter Hellwig, University of Heidelberg

(Version July 2015)

Abstract

PLAIN (Programs for Language Analysis and Inference) is an integrated development environment (IDE) which provides comprehensive facilities to (computational) linguists for creating and processing lingware. PLAIN adheres to Dependency Unification Grammar (DUG), a particular linguistic approach to natural languages. DUG aims at a simple and, at the same time, broad coverage of linguistic phenomena.

Dependency Representation Language (DRL) is the formalism of DUG. DRL is so to say a programming language - to be used by a linguist in order to have the computer analyze natural language. In this paper we discuss the DUG approach to morphology and the resources that have to be drawn up for a morpho-syntactic component.

The morpho-syntactic component can be integrated in the parser. It can also drive a tagger which recognizes and classifies words in corpora. The component can be used for generating forms which belong to given categories, or for generating all forms of a word together with their classification. The morpho-syntactic resources are also employed if surface strings are created corresponding to a syntactic description in DRL.

Considerably large resources for German and English are available. In this article, we focus on the system as a development tool, though. En passant, theoretical assumptions are mentioned that are hiding in the system's architecture. The software may be especially helpful for languages that do not dispose of large computational resources yet.

The system is open-source and can be downloaded from the internet. The program is apt to cope with many morphological phenomena. But there may be problems we are not aware of yet. That is why we are looking for people who want to use the software and give us feed-back.

Contents

Introduction

The objective of the component described below is the recognition and classification of words in running texts. The term "word" is defined operationally. A word is the smallest unit of the syntax component. Hence, words are the units at the interface between the scanner (which reads the text and classifies the segments) and the parser (which finds out the structure of these units). There is room for arbitrary decisions here. What is treated as a word in a concrete implementation is a matter of practicality rather than truth.

In fact, the morphological component we advocate is a morpho-syntactic one. It classifies the words in such a way that the categories are suitable to discriminate syntagmatic relationships.

The implementation is committed to taxonomic linguistics. The system needs resources to solve its task. The resources must cover any form and any feature of the language in question. Gathering this information is much work and must be done by linguists. The software facilitates this work.

We assume that the reader is familiar with taxonomic heuristics. Let us just recall some aspects. One principle is opposition. One compares word forms and observes what they have in common and in what they differ. In this way the relevant attributes are detected and defined.

Compared...	...with	In common:	Different:
man	*men*	*lexeme=man*	*number=singular/plural*
ox	*oxen*	*lexeme=ox*	*number=singular/plural*
men	*oxen*	*number=plural*	*lexeme=man/ox*
men	*mice*	*number=plural*	*lexeme=man/mouse*
like	*liked*	*lexeme=like*	*tense=present/past*
go	*went*	*lexeme=go*	*tense=present/past*
went	*took*	*tense=past*	*lexeme=go/take*

Figure 1 Deriving attributes from oppositions

Syntagmatic relationships are taken into account, too. Here, the basic technique is substitution. The phenomenon of agreement is emerging. For example:

I go
* I goes
he goes
* he go
I went
he went

Compared...	...with	In common:	Different:
I	*he*	*part of speech=pronoun*	*person=I / he*
go	*goes*	*part of speech=verb*	*person=I,you,we,they / he,she,it*
go	*went*	*part of speech=verb* *person=I,you,we,they*	*person=he,she,it*
goes	*went*	*part of speech=verb* *person=he,she,it*	*person=I,you, we,they*

Figure 2 Deriving contextual attributes from oppositions

Similar heuristics can be applied to forms. Word forms can often be broken down into segments, which are similar or different to other forms. Most relevant are particular forms that correspond with particular attributes.

Compared...	...with	In common:	Different:
car	*car-s*	*"car" lexeme=car*	*"- / -s" number=singular / plural*
ox	*ox-en*	*"ox" lexeme=ox*	*"- / -en" number=singular / plural*
call	*call-s*	*"call" lexeme=call*	*"- / -s" person=I,you,we,they / he,she,it*
car-s	*ox-en*	*number=plural*	*"car / ox" lexeme=car / ox* *"-s / -en"*
call-s	*pass-es*	*person=he,she,it*	*"call / pass" lexeme=call / pass* *"-s / -es"*

Figure 3 Segmentation of forms and associating attributes with segments

At this point, it is worth recalling the dichotomy between the syntagmatic and paradigmatic relationship and, correspondingly, between syntagm and paradigm. A syntagmatic relationship exists between items which occur in the same construction, e.g. *I + go.* The particular construction, e.g. *subject + predicate*, is the syntagm. A paradigmatic relationship exists between an item and another item if both occur in the same syntagm, e.g. *I, you* as *subject* and *go, goes, went* as *predicate.* A set of forms in a particular paradigmatic relationship is a paradigm.

The realm of syntagms is the syntax component. However, the principle of syntagmatic relation is also relevant in morphology. On the one hand, words must be classified in a way that they can be accepted or rejected in a syntactic construction. On the other hand, words may be composed, consisting of stems, prefixes, endings. These elements then form paradigms of substitutable items.

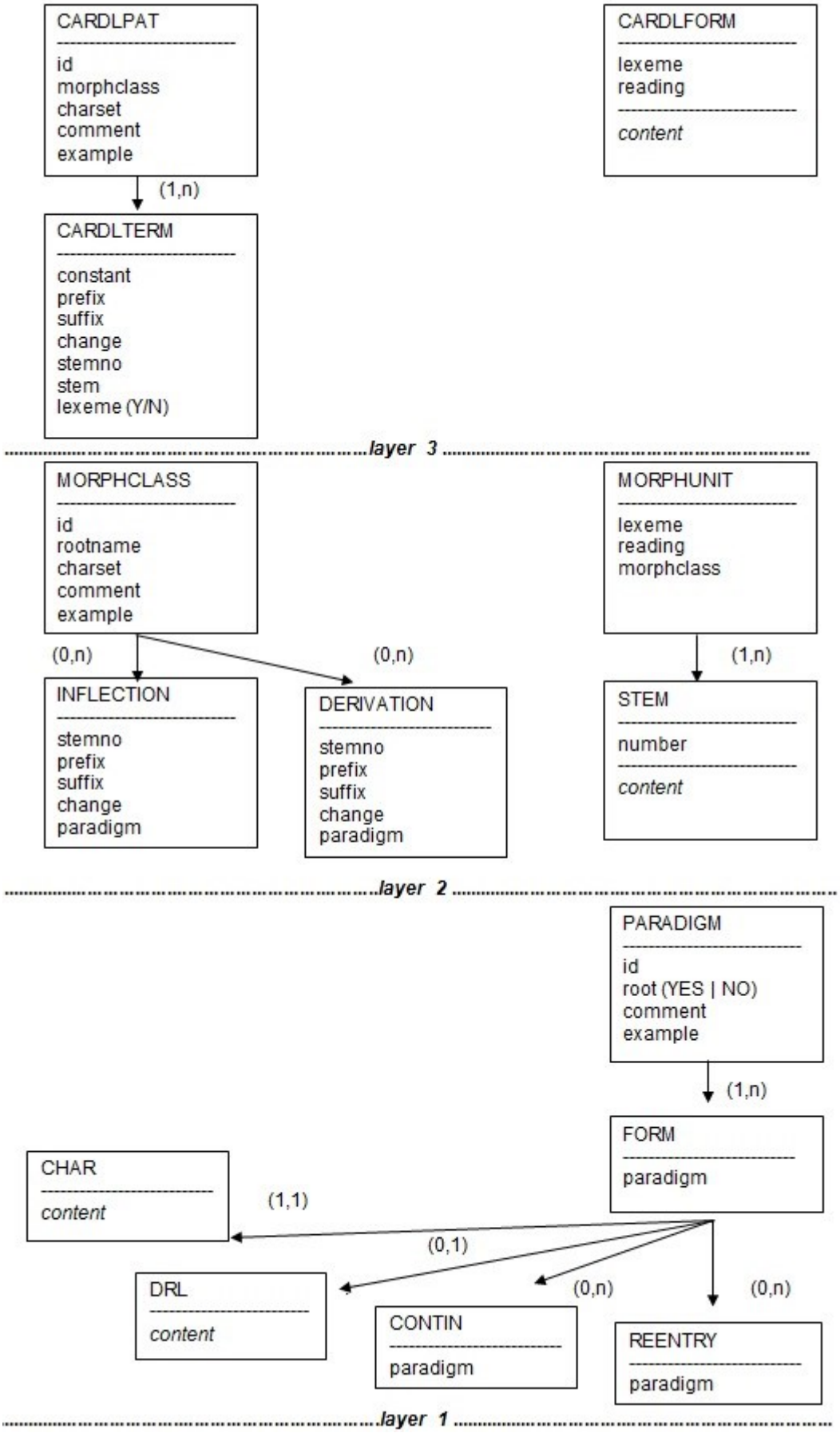

Figure 4 EAR model of morphological resources

According to taxonomic linguistics, the sum of all oppositions in which an element takes part is its "distribution". Finding a design for representing the distribution of all words has been the goal of our project.

The resulting architecture includes a set of category definitions and three layers of resources. Figure 4 displays the conceptual scheme of these resources with entities, attributes and relationships (EAR). In what follows we try to explain these constructs.

Category representation and category definition

The system's resource files are written in XML, according to a particular DTD which corresponds roughly to Figure 4 (except for the capitalization of entities).

The elementary unit of the morpho-syntactic component is a FORM. A form consist of a string (CHAR) and a categorization (DRL). For the XML interpreter, the content of a DRL element is character data (CDATA) which is not to be parsed.

The PLAIN IDE, however, interprets the content of the DRL element as an expression of the DUG formalism. The general format of DRL expressions is a tree structure of complex categories. Each category is surrounded by brackets. A category contains an arbitrary list of attributes. Each attribute consists of an attribute name followed by a list of values. The list of values is surrounded by square brackets. Values are represented by their name. Several values (usually denoting a disjunction) are separated by commas.

This notation is practical for translating informal linguistic descriptions into formal ones. Bear in mind that traditional linguistic is the primary source of the type of data we need. The ordinary school teacher's statement

> "The character string 'goes' is an inflectional form of the word 'go' indicating a verb in the third person singular present tense."

is encoded as in Figure 5.

```
<form>
    <char>goes</char>
    <drl> (lexeme[go] category[verb] person[third] number[singular]
     tense[present])</drl>
</form>
```

Figure 5 Elementary morphological encoding

A notation with complex categories is superior to simple tags. As demonstrated above, several attributes may characterize the same form. (One morph may carry several morphemes.) What is more, morphological categories are multi-dimensional. Each attribute may have emerged by substitution in another syntagmatic context. Try to imagine the heuristic operations which lead to the attributes *lexeme, category, person, number,* and *tense* in Figure 5. Complex categories are cross classifications, which allow to identify the same string under various aspects. They are space saving. Disjunctive values of attributes increase this space saving effect.

Complex categories lend themselves to calculating agreement of words and phrases. The simplest case is the intersection of values of those attributes that must agree. If the intersection is empty then there is no agreement. With this criterion, improper instances of ambiguous classifications can be ruled out. Compare the classifications and results in Figure 6.

```
1     'I'         (person[first] number[singular])
2     'he'        (person[third] number[singular])
3     'we'        (person[first] number[plural])
```

```
a     'go'        (person[first, second] number[singular]
b     'go'        (person[first, second, third] number[plural]
c     'goes'      (person[third] number[singular]
```

```
1+a   'I go'      (person[first] number[singular])
1+b   'I go'            EMPTY number[]
1+c   'I goes'          EMPTY person[]
2+a   'he go'           EMPTY person[]
2+b   'he go'           EMPTY number[]
2+c   'he goes'   (person[third] number[singular])
3+a   'we go'           EMPTY number[]
3+b   'we go'     (person[first] number[plural])
3+c   'we goes'         EMPTY person[]EMPTY number[]
```

Figure 6 Calculating agreement by intersection of attribute values

Treating agreement in this way is similar to solving equations in mathematics and logic. This is, in fact, the essence of unification grammars like DUG. They determine the grammatical constructions of a language by means of complex equations rather than by applying rules one after the other.

Most grammars of this family possess one uniform mechanism of unification. Linguistic reality is too variable, though, to tar all phenomena with the same brush. That is why several equations are available in DUG for calculating agreement. Each attribute must make a choice on a particular method. There is not just one algorithm of unifying the categories of several words or phrases. Instead, different built-in routines are associated with attributes according to a type declaration. Each attribute in a complex category invokes a little program which interacts with the same or with different attributes in the complex category of another item.

As a consequence, the first thing to do when building a morpho-syntactic component is to define the attributes. Figure 7 gives an example. A file with such definitions has to be drawn up.

```
<catdef>
    <lx>
        <name>lexeme</name>
        <unrestricted/>
    </lx>
</catdef>

<catdef>
    <mc>
        <name>category</name>
        <val>sentence</val>
        <val>verb</val>
        <val>noun</val>
        <val>adjective</val>
        <val>determiner</val>
        <val>preposition</val>
        <val>conjunction</val>
        <val>adverb</val>
        <val>empty</val>
     </mc>
</catdef>

<catdef>
    <df>
        <name>number</name>
        <val>singular</val>
        <val>plural</val>
    </df>
</catdef>
```

Figure 7 Example of attribute definitions

The XML marker of an attribute definition is <catdef>. First the type of the attribute has to be declared, e.g. <lx>, <mc> or <df>. This type determines the built-in routine that is invoked if the attribute occurs. Then the name and possibly the values of the attribute must be specified. You are completely free as to what name the attributes should have. You might adjust it to the language you are dealing with. Values may be declared as <unrestricted> or they may be listed.

PLAIN offers many types of attributes (see the file plain-xml.dtd). Most of them do not play a role in morphology. Nevertheless, in order to give you an impression of the total framework, here is a list of the actual assortment:

```
Semantic features:

      lx  lexeme
      rd  reading
      hy  hyperonym

Grammatical features:

      ut  utterance property, illocution
      rl  role, syntactic function
      mc  main syntactic category, part of speech
      df  disjunctive feature
      cf  conjunctive feature
      ef  exclusive feature
      of  overwriting feature

Surface form features:

      ch  character string
      qu  quotation
      lp  left punctuation mark
      rp  right punktuation mark
      cs  upper and lower case
      ud  utterance delimiter

Attribute excluding attributes:

      ne  unacceptable feature

Word order features:

      lt  left side dependent (within tree projection)
      rt  right side dependent (within tree projection)
      sc  numbered succession
      aj  adjacency
      mg  margin position


DUG constructs in syntax descriptions:

      tp   template name in a template
      sl   slot indicator in a template
      cp   complement in a synframe
      ad   adjunct in a synframe
      ea   expected adjunct in a synframe
      co   conjunct in a synframe
      nc   nucleus complement in a synframe
      rs   raising complements in a synframe
      tc  a trace of an elliptic conjunct

Logical constants and transducer rules:

      no   logical not
      tr   logical true
      fl   logical false
      rr   replacement rule
      er   expansion rule
```

Figure 8 List of actual types of attributes

In addition to the attribute declaration, there is an on-line device to sort attributes into visible ones and hidden ones. The user can shift a particular attribute from one group into the other and, thus, create different classifications. For example, using the component as a lemmatizer one may just leave visible the lexeme attribute. The system works, so to speak, with varying tag sets.

Lexicon base

Now we have to draw up the proper morphological data. From linguistic viewpoint, the conceptual model is taxonomic distribution. The central construct for handling distribution is the paradigm. A paradigm is a list of substitutable forms, each form consisting of a surface string and a list of attributes.

The most general paradigm of an implementation is the set of all words. The word forms *go, goes* and *went* could be encoded in the following way. (In this version of English data the usual attributes 'person' and 'number' are merged into a single attribute 'person' with the values *I, you, he, she, it, we, they, they_personal.* This simplyfies the distribution of nouns, pronouns and verbs.)

```
<paradigm id="start" root="yes">
    <form>
        <char>go</char>
        <drl> (lexeme[go] category[verb] person[I,you,we,they,
        they_personal] tense[present]) </drl>
    </form>
    <form>
        <char>goes</char>
        <drl> (lexeme[go] category[verb] person[he,she,it]
        tense[present]) </drl>
    </form>
    <form>
        <char>went</char>
        <drl> (lexeme[go] category[verb] person[I,you,he,she,
        it,we,they, they_personal] tense[past]) </drl>
    </form>
</paradigm>
```

Figure 9 Encoding word forms

It is helpful to have a conceptual model of the implementation, too. From the computational viewpoint paradigms can be conceived as finite transition networks (FTN). The initial state of the FTN is the start of the paradigm, before any form is read. The arcs of the FTN are the forms. They are labelled by the character string and the category of the form. The final states of the FTN are the points after a form is read. A transition from the initial to a final state is permitted if the character string in the label matches the input of the automaton.

Of course, it should be possible to represent the composition of words too. For example, there may be stems and endings and several stems may have the same endings. In this case, we need a method

for connecting forms with paradigms that contain their syntagmatic continuations. The element <contin> is introduced into the XML representation for this purpose. Compare Figure 4. For example, we could now spread the description of the word forms *car, cars* and *ox, oxen* over several paradigms as follows.

```
<paradigm id="start" root="yes">
    <form>
        <char>car</char>
        <drl> (lexeme[car]) </drl>
        <contin paradigm="noun-s"/>
    </form>
    <form>
        <char>ox</char>
        <drl> (lexeme[ox]) </drl>
        <contin paradigm="noun-en"/>
    </form>
</paradigm>

<paradigm id="noun-s">
    <form>
        <char></char>
        <drl>(category[noun] number[singular])</drl>
    </form>
    <form>
        <char>s</char>
        <drl>(category[noun] number[plural])</drl>
    </form>
</paradigm>

<paradigm id="noun-en">
    <form>
        <char></char>
        <drl>(category[noun] number[singular])</drl>
    </form>
    <form>
        <char>en</char>
        <drl>(category[noun] number[plural])</drl>
    </form>
</paradigm>
```

Figure 10 Encoding stems and endings in different paradigms

The paradigms are now subnets within an overall FTN. The contin-element causes a transition from a particular final state of one subnet into the initial state of another subnet.

Under the aspect of processing there is nothing else. At run time, the system is a finite state automaton. Anything necessary to handle the peculiarities of a full-fletched morphology must accommodate somehow with this confinement. Our strategy is to introduce higher layers of resources and a conversion from the higher to the lower ones.

The well-known distinctions of Hocket (1954) and Spencer (1991) may serve for orientation. There are allegedly three principal approaches to morphology: the Item-and-Arrangement approach, the Item-and-Process approach, the Word-and-Paradigm approach. The difference is not so much a matter of approach, though. It is rather a matter of phenomena that occur within a language, similar to the Chomsky hierarchy of formal grammars. Therefore, PLAIN allows for all of the three models.

The Word-and-Paradigm approach always works. One can draw up a lexicon which consists of a single list of word forms. The entries would look like those in Figure 9. However, the number of forms explodes in some languages. In the case of an inflecting language, listing all forms of each word is not a favorable way of lexicon acquisition. A simple list of words also lacks linguistic transparency.

That is why the Word-and-Paradigm approach should be followed for non-inflecting words only. Often the closed classes of words, e.g. determiners, prepositions, numbers etc., are candidates. For the rest, this method of encoding is a lifeline in case of idiosyncrasies, as for example the paradigm of *be* with the inseparable forms *am, are, is.*

In the case of agglutinative languages, a large subset of words can be treated according to the Item-and-Arrangement approach. Regular inflection and derivation can be encoded in terms of paradigms of stems, affixes and endings. Allomorphs among endings simply result in alternative ending paradigms, compare *car-s* and *ox-en* in Figure 10.

So far, we made use only of the layer 1 of resources in Figure 4.

Morphological units and morphological classes

If there is a variation within the stem of a word, e.g. *fall* versus *fell, each* stem must be inserted separately in the stem paradigm and linked with the appropriate subset of ending paradigms. In this way, the principle of Item-and-Arrangement can be preserved (and it must be preserved because it is paired with the implementation of the system as FTN). However, the unity of the word is lost.

There is a desire, though, to represent the unity of the word in some way. As a consequence, a new layer of resources is established. See layer 2 in Figure 4. This layer is characterized by a new conceptual element, the morphological unit (MORPHUNIT). A morphunit is a complete paradigm of a lexical item with the same lexeme, no matter how much the forms within the paradigm vary.

The lexicon, or parts thereof, can now be encoded in terms of morphunits. The XML element <morphunit> is used to mark up such a description. Here is an example:

```
<morphunit lexeme="try" morphclass="vc14">
    <stem number="1">try</stem>
    <stem number="2">trie</stem>
</morphunit>

<morphunit lexeme="fall" morphclass="ve17">
    <stem number="1">fall</stem>
    <stem number="2">fell</stem>
</morphunit>
```

Figure 11 The morphological units *try* and *fall*

Two kinds of information characterize a morphunit: its lexeme and a key for reconstructing its forms. The latter appears as a morphological class associated with the item. Stems can also be specified. (This is an option. A particular change of stems can also be a property of the indicated morphclass.)

A morphological class (MORPHCLASS) displays a particular morphological behavior. The morphclasses must be spelled out, so that all forms of a morphunit belonging to the class can be derived. The morphclasses assigned to morphunits in Figure 11 are encoded as follows:

```
<morphclass id="vc14" rootname="start"
    <inflection stemno="1"   paradigm="vinf"/>
    <inflection stemno="1"   paradigm="vprs-0"/>
    <inflection stemno="2"   paradigm="vprs-s"/>
    <inflection stemno="2"   paradigm="vpas-d"/>
    <inflection stemno="1"   paradigm="vprp-ing"/>
    <inflection stemno="2"   paradigm="vpap-d"/>
</morphclass>

<morphclass id="ve17" rootname="start"
    <inflection stemno="1"   paradigm="vinf"/>
    <inflection stemno="1"   paradigm="vprs-0"/>
    <inflection stemno="1"   paradigm="vprs-s"/>
    <inflection stemno="2"   paradigm="vpas-0"/>
    <inflection stemno="1"   paradigm="vprp-ing"/>
    <inflection stemno="1"   paradigm="vpap-en"/>
</morphclass>
```

Figure 12 The morphological classes assigned to the morphunits of *try* and *fall*

The morphclasses in Figure 12 denote particular stem and ending combinations. The ending paradigms must be encoded in the lexicon base. They must display the appropriate characters of the endings and the resulting attributes. Examples of ending paradigms are "noun-s" and "noun-en" in Figure 10. We refrain from displaying the paradigms of verb endings in Figure 12 here. Their content is the following:

vinf	base form, infinitive, imperative, no inflection as in 'to call, call'
vprs-0	present tense, no ending as in 'I call'
vprs-s	present tense, third person as in 'he call-s'
vpas-0	past tense, no inflection as in 'I put, he put'
vpas-d	past tense, base –e, ending –d as in 'I like-d'
vprp-ing	present participle, adjective, gerund as in 'call-ing'
vpap-d	past participle, base –e, ending –d as in 'like-d'
vpap-en	past participle, strong inflection as in 'beat-en'

Given the morphclass and stems, a converter program turns morphunits (layer 2) into forms (layer 1). In the latter format, the data is stored in the lexicon base. According to the instructions in the class descriptions, the converter links up the numbered stems of the morphunit with the indicated ending paradigms. For example, the converter output for morphunit of *try* in Figure 8 is the following.

```
<form paradigm="start">
    <char>try</char>
    <drl>(lexeme[try])</drl>
    <contin paradigm="vinf"/>
</form>
<form paradigm="start">
    <char>try</char>
    <drl>(lexeme[try])</drl>
    <contin paradigm="vprs-0"/>
</form>
<form paradigm="start">
    <char>trie</char>
    <drl>(lexeme[try])</drl>
    <contin paradigm="vprs-s"/>
</form><form paradigm="start">
    <char>trie</char>
    <drl>(lexeme[try])</drl>
    <contin paradigm="vpas-d"/>
</form>
<form paradigm="start">
    <char>try</char>
    <drl>(lexeme[try])</drl>
    <contin paradigm="vprp-ing"/>
</form>
<form paradigm="start">
    <char>trie</char>
    <drl>(lexeme[try])</drl>
    <contin paradigm="vpap-d"/>
</form>
```

Figure 13 Forms automatically derived from the morphunit *try*

Note: The above examples of morphclasses do not extend the framework of Item-and-Arrangement. Oft course, this is not the final solution. In fact, we would like to grasp the exact relationship between

try and *trie*, as well as *fall* and *fell.* There is a device to specify these changes within the morphclass description. Using this facility would mean a shift to the item-and-process approach. In this article, we leave this option to layer 3.

Cardinalforms and cardinal patterns

Specifying morphunits manually is cumbersome, although not as bad as encoding inflectional word forms. Can't we make the computer recognize the morphclass of a word and create the morphunit automatically? Remember how school children learn the irregular inflection of words. They are to memorize cardinal or principal forms, e.g. *to go, went, gone, to fall, fell, fallen.* This is an instance of learning by example. Why not take advantage of this method in computational morphology? As a consequence, we introduce cardinal forms (CARDLFORM) as a third layer of resources. A set of cardlforms could look as follows:

```
<cardlform>agree agrees agreed agreeing agreed</cardlform>
<cardlform>call calls called calling called</cardlform>
<cardlform>eat eats ate eating eaten</cardlform>
<cardlform>fall falls fell falling fallen</cardlform>
<cardlform>put puts put putting put</cardlform>
<cardlform>show shows showed showing shown</cardlform>
<cardlform>try tries tried trying tried</cardlform>
```

Figure 14 Cardinalforms of English verbs

A cardinal form must display enough of the peculiarities of a word so that the whole inflection can be deduced. For English verbs, the following forms must be shown: infinitive, 3rd person singular present tense, past tense, ing-participle, past participle.

In order to draw up morphunits from cardinal forms automatically, the computer must separate stems and endings, extract the lexeme and the various stems and recognize the morphclass on the basis of the demonstrated inflection. This task is not difficult if the program is provided with patterns of the cardinal forms. So, what we have to do is augment layer 2 by the element "cardinal patterns" (CARLDPAT). A cardinal pattern must include a description of each word in the corresponding cardinal form. Such elements are coined "cardinal term" (CARDLTERM).

```
<cardlpat  morphclass="vc14" >
    <cardlterm stemno="1" lexeme="yes"/>
    <cardlterm change="(.+)([y])/$1ie" suffix="s" stemno="2"/>
    <cardlterm change="(.+)([y])/$1ie" suffix="d"/>
    <cardlterm suffix="ing"/>
    <cardlterm change="(.+)([y])/$1ie" suffix="d"/>
</cardlpat>

<cardlpat morphclass="ve17">
    <cardlterm stemno="1" lexeme="yes"/>
    <cardlterm suffix="s"/>
    <cardlterm change="([f])([a])(.+)/$1e$3)" stemno="2"/>
    <cardlterm suffix="ing"/>
    <cardlterm suffix="en"/>
</cardlpat>
```

Figure 15 Cardinal patterns

The first pattern in Figure 15 matches *try tries tried trying tried*. The second one matches *fall, falls, fell, falling, fallen.* Applying the pattern in Figure 15 to these cardinal forms results in the same morphunits as in Figure 11. The cardlterms describe the differences between the items in the cardinal forms. Prefixes and suffixes are stripped off first, if any. The differences between the remaining strings are conveyed by the attribute "change". If the string resulting from the changes should be turned into a stem in the emerging morphunit then the attribute "stemno" with the number of this stem must be included in the cardlterm. The word in the cardinal forms that is to be turned into the lexeme attribute of the morphunit is marked by the attribute lexeme="yes" in the corresponding cardlterm.

The introduction of the change attribute is the final step towards the Item-and-Process model. The value of the attribute "change" consists of two parts, separated by a slash. The first part is a regular expression that always refers to the first item in the cardlforms after prefix and suffix is stripped off . Let us call this the base form. In the case of *fallen* it is *fall.* The second part of the expression describes the shape of the cardinal form in question, usually in form of replacements of the base form. Substrings that should be copied from the base form to the changed form must be put in brackets in the regular expression. These substrings are referred to in the changed form as "$n", where n is a count of the bracketed expressions. All the usual facilities of regular expressions are at hand. "." is a generic variable to substitute for any character. Repetitions are symbolised in the usual way, e.g. "(.*)", "(.+)". Sets of characters can be defined by means of the attribute "charset" and be used in the expression. Names of sets in the regular expressions must be preceded by a backslash.

English does not stand out as a language with a rich morphology. Let us choose an example from Latin, to give an impression of the power of the Item-and-Process device.

```
<cardlpat  morphclass="vred1"
    example="mordere mordeo momordi morsum">
    <cardlterm suffix="ere" stemno="1" lexeme="yes"/>
    <cardlterm suffix="eo"/>
    <cardlterm suffix="i" change="([pmt][eo])(.+)/$1$1$2"/>
    <cardlterm suffix="sum" change="(.+)d/$1"/>
</cardlpat>

<cardlpat morphclass="vred2" charset="C=[spndr] V=[oe]"
    example="spondere spondeo spopondi sponsum">
    <cardlterm suffix="ere" stemno="1" lexeme="yes"/>
    <cardlterm suffix="eo"/>
    <cardlterm suffix="i" change="(\C)(\C)(\V)(.+)/$1$2$3$2$3$4"
    stemno="2"/>
    <cardlterm suffix="sum" change="(.+)d/$1"/>
</cardlpat>
```

Figure 16 Patterns for Latin cardlforms illustrating reduplication

The verbs *mordere* and *spondere* show reduplication in the perfect tense. Just for illustration, we use different techniques for the two verbs. *mordere* belongs to a group of verbs beginning with one consonant that must be "p", "m" or "t" and followed by the vowel "e" or "o". This syllable is duplicated in the perfect tense. The process is recorded directly in the corresponding cardlterm: change="`([pmt][eo])(.+)/$1$1$2`" .

In the case of *spondere* we have defined consonants and vowels by means of the charset attribute of cardlpat, namely `charset="C=[spndr] V=[oe]"`. The change attribute now looks as follows: `change="(\C)(\C)(\V)(.+)/$1$2$3$2$3$4`. This means that *spondere* and similar reduplicating verbs begin with two consonants followed by a vowel. They reduplicate the second consonant together with the vowel. The morphunits created on the basis of these patterns are the following:

```
    <morphunit lexeme="mordere" morphclass="vred1">
        <stem number="1">mord</stem>
    </morphunit>

    <morphunit lexeme="spondere" morphclass="vred2">
        <stem number="1">spond</stem>
        <stem number="2">spopond</stem>
    </morphunit>
```

Figure 17 Derived morphunits in Latin

Remember that the same machinery for Item-and-Process models is already available on level 2. In the case of *mordere,* it would be necessary to create the stem with the reduplication when converting the morphunit into forms. This can be achieved by means of a change attribute in the morphclass. In the case of *spondere* the stem *spopond* can be directly linked to the perfect ending.

Besides, there is also a converter from cardlforms directly to forms, i.e. from level 3 to level 1. It needs cardlpatterns and morphclasses as well, but skips the level of morphunits.

Problems and solutions

Space is not a distinguished character. Multi word lexemes, as '*in front of*', are just treated as one character string within the basic encoding:

<form><char>in front of</char><form>

The segmentation of continuous text is achieved by matching incoming characters with the internal network until a final state is reached. The next incoming character is automatically matched with the root of the whole network again. Provisions are taken for alternative final states and corresponding ambiguous segmentations. Originally this device has been used for compounds as the German *zweitausendvierhundertdreiundvierzig* (two thousand four hundred forty three) or *Reiseschreib-maschine* (travelling typewriter).

There is an annoying side effect of this method, though. Some endings coincide in German with independent words, for example, the adjective endings *-er, -es* (e.g. *schöner, schönes*) are identical to the personal pronouns *er* (he) and *es* (it). This leads to nonsensical compounds *schön + er, schön + es.* This is why we decided to make the composition of compounds explicit by means of the a element REENTRY as an alternative to CONTIN. Compare Figure 4.

The XML markers <contin> and <reentry> differ in the output. While all segments via a contin-transition through the network are combined into one word and associated with a single <drl>, all segments found via a reentry-transitions are kept separate as independent words, each one with its own <drl>. The transition from one part of a compound to another can now be tuned to the special circumstances. For example, the reentry for a number can be restricted to the paradigm of numbers. The recombination of the parts of compounds and the disambiguation of different segmentations is a matter of the parser.

In some languages there are discontinuous morphs. For example, the German past participle is represented by the prefix *ge-* and the ending *–t* at the same time. We have *ge-mach-t,* which must have the attribute "verb past participle". We also have the form *mach-t* which has the attribute (among others) "verb present tense 3rd person singular". When the automaton arrives at the *–t* it needs information about the presence or absence of *ge-.* This context-sensitive information can indeed be provided in form of the attributes "ge-prefix[+]" and "ge-prefix[-]". The first one is associated with the participle affixes *ge-* and *–t*, while the finite verb affix *–t* is associated with the latter.

At this point, it is convenient that our morpho-syntactic component is, in fact, a module of a unification grammar. Context-sensitive unification is deployed in word formation in the following way. The categories of all forms encountered in a path through the morpho-syntactic network are collected. If the same attribute occurs several times then the agreement of values is calculated. If there is an agreement violation then the reading is rejected. A form with *ge-* at the beginning and the ending *–t* at the end classified as finite verb form is ruled out.

Let us conclude the survey with pointing out some advantages of the described system. The Item-and-Arrangement model of morphology is ideal for computers. However, it is suited only for agglutinative languages, which combine morphological elements without changes of form or loss of meaning. For many languages an Item-and-Process model is more appropriate, because the morphological elements of these languages vary due to phonological, etymological or other reasons. In some cases even context-sensitivity is required.

Formalisms have been invented that model such processes, for example the influential Two-Level Morphology introduced by Koskenniemi (1983). Koskenniemi's morphology works with an underlying lexical level and a surface morpho-syntactic level. So-called transducers derive the surface word forms from the lexical representation, e.g. the past tense *fell* from the canonical representation *fall.* This is done on the basis of a set of intricate replacement rules. The process is executed at run-time, i.e. it is repeated each time a word is analysed.

Cardinal forms in our system cope with the same phenomenon. As opposed to intricate encodings, cardinal forms just demonstrate the behaviour of words. The different cases of behaviour are reflected in the cardinal patterns. Rather than executing morphological processes at runtime, our implementation applies these processes just once, namely while the morpho-syntactic lexicon is drawn up and stored in a database. At run-time the original complexity of the morphological structures does not harm the efficiency of the program any more.

However, the biggest advantage of the method of cardinal forms is the fact that the morpho-syntactic resources can easily be updated by personal that is not especially trained. Cardinal forms can be drawn up by everybody with a normal school education. This is beneficial In broad-coverage applications.

The method of data acquisition by cardinal forms is also favourable for NLP systems. If a word is unknown to the computer, the user can be easily guided to enter cardinal forms interactively. He can then profit immediately from the improved linguistic knowledge of the system.

Finally, the level of cardinal forms is interesting for the exchange of data between systems that may differ in their theory and classifications. Cardinal forms are free of theoretical commitments. They just demonstrate the phenomena.

Tools and test

The PLAIN IDE is made to facilitate the linguist's work. Let us have a look which tools are at hand for the morpho-syntactic component. Examples come from the German demonstration project which is included in the downloded files at *www.plain-nlp.de*. The following routines help to confirm correctness or to understand the malfunction of a morphological object:

- *Morphology > Lookup > String*
- *Morphology > Lookup > File (all, unknows only, duplicates only)*
- *Morphology > Generate Word Forms from Root*
- *Generator > Generate Word Forms from Lexeme (Select part of speech (optional), Show traversed paradigms)*
- *Converters > Cardlforms to Morphunits (show debug output)*
- *Converters > Cardlforms to Paradigms*

Implementing a morphological component for a new language with PLAIN implies three tasks:

1. drawing up a lexicon base of paradigms for closed classes, inflection and derivation,
2. creating the interface of cardinal forms,
3. entering carldinal forms and reaching broad coverage of open class vocabulary.

Closed classes of vocabulary like prepositions, conjunctions, all kind of particles are directly inserted in the lexicon base. A paradigm for stems and various paradigms for inflection suffixes and derivation infixes must also be established. Single forms can be checked by manual lookup with *Morphology > Lookup > String.* It is advisable, however, to maintain a system of test files which contain examples for each phenomenon. Such a file can be processed with *Morphology > Lookup > File (all).* The output should be kept and the test file processed again after any change in the lexicon. The actual output and prior outputs can then be compared in order to detect possible side effects of the change.

The following little test file picks out German examples with different features:

```
dem
Männern
gedacht
auf
zum
"hallo"
hallo!
(hallo hallo)
Staubecken
StauBecken
neunundneunzig
```

Figure 18 Small test file

Morphology > *Lookup* > *File (all)* yields the following output:

```
1   'dem '
      (lexem[definit'] kategorie[artikelwort] flexion[stark-schwach]
      genus[maskulin,neutrum] kasus[dativ] numerus[singular]
      schreibung[klein]);

6   'Männern '
      (lexem[mann] kategorie[nomen] kasus[dativ] numerus[plural]
      person[dritte] pronomen[nein] schreibung[gross]);

16  'gedacht '
      (lexem[denken] kategorie[verb] form[partizip] schreibung[klein]);

25  'auf '
      (lexem[auf] kategorie[praefix] kompositum[-] schreibung[klein]);

25  'auf '
      (lexem[auf] kategorie[praeposition] kasus[dativ,akkusativ]
      schreibung[klein]);

25  'auf '
      (lexem[auf] kategorie[partikel] kompositum[-] steigerung[keine]
      verwendung[praedikativ] schreibung[klein]);

30  'zum '
      (lexem[zu] kategorie[praeposition] genus[maskulin,neutrum]
      kasus[dativ] numerus[singular] schreibung[klein])
      (lexem[definit'] kategorie[artikelwort] flexion[stark-schwach]
      genus[maskulin,neutrum] kasus[dativ] numerus[singular]);

35  '"hallo"'
      (lexem[hallo] schreibung[klein] zeichen_links[zitat]
      zeichen_rechts[zitat]);

43  'hallo'
      (lexem[hallo] schreibung[klein]);

48  '!'
      (lexem[ausruf'] kategorie[satz] aeusserung[+] schreibung[klein]);

50  '(hallo '
      (lexem[hallo] schreibung[klein] zeichen_links[klammer]);

57  'hallo)'
      (lexem[hallo] schreibung[klein] zeichen_rechts[klammer]);

64  'Stau'
      (lexem[stau] kategorie[partikel] kompositum[+] numerus[singular]
      verwendung[attributiv] schreibung[gross]);

64  'Staub'
      (lexem[staub] kategorie[partikel] kompositum[+] numerus[singular]
      verwendung[attributiv] schreibung[gross]);

68  'becken '
      (lexem[becken] kategorie[nomen] genus[neutrum]
      kasus[nominativ,dativ,akkusativ] numerus[singular] person[dritte]
      pronomen[nein] schreibung[klein]);
```

```
69  'ecken '
      (lexem[ecke] kategorie[nomen] kasus[nominativ,genitiv,dativ,
      akkusativ] numerus[plural] person[dritte] pronomen[nein]
      schreibung[klein]);

68  'becken '
      (lexem[becken] kategorie[nomen]
      kasus[nominativ,genitiv,dativ,akkusativ] numerus[plural]
      person[dritte] pronomen[nein] schreibung[klein]);

76  'Stau'
      (lexem[stau] kategorie[partikel] kompositum[+] numerus[singular]
      verwendung[attributiv] schreibung[gross]);

80  'Becken '
      (lexem[becken] kategorie[nomen]
      kasus[nominativ,genitiv,dativ,akkusativ] numerus[plural]
      person[dritte] pronomen[nein] schreibung[gross]);

80  'Becken '
      (lexem[becken] kategorie[nomen] genus[neutrum]
      kasus[nominativ,dativ,akkusativ] numerus[singular] person[dritte]
      pronomen[nein] schreibung[gross]);

88  'neun'
      (lexem[neun] kategorie[adjektiv] kleiner[10,100,1000]
      schreibung[klein]);

92  'und'
      (und[+] schreibung[klein]);

95  'neun'
      (lexem[neun] kategorie[adjektiv] kleiner[10,100,1000]
      schreibung[klein]);

99  'zig '
      (lexem[zig] kleiner[100,1000] schreibung[klein]);

No (more) entries found
```

Figure 19 Output of the lookup function

Some words are displayed with several classifications, e.g. 'auf' or 'Becken' in Figure 19. One may wonder whether the words are homonyms or whether they are entered several times by mistake. One should run *Morphology* > *Lookup* > *File* with the option *duplicates only.* With this setup any strings with exactly the same classification is displayed and can be ruled out.

The accuracy of the morphological description can also be tested by generation. What is generated can be looked up. The function *Morphology* > *Generate Word Forms from Root* is suitable to check the forms that are built with a particular stem, let us say 'Mann'. There is a more powerful generator, though. The function *Generator* > *Generate Word Forms from Lexeme* is able to generate all forms even if the stem changes. If we try the lexeme 'mann' we get:

```
Generated word forms of lexeme 'mann':

'Mann'
        (lexem[mann] kategorie[nomen] genus[maskulin]
        kasus[nominativ,dativ,akkusativ] numerus[singular] person[dritte]
        pronomen[nein]);

'Manne'
        (lexem[mann] kategorie[nomen] genus[maskulin] kasus[dativ]
        numerus[singular] person[dritte] pronomen[nein]);

'Mannes'
        (lexem[mann] kategorie[nomen] genus[maskulin] kasus[genitiv]
        numerus[singular] person[dritte] pronomen[nein]);

'Manns'
        (lexem[mann] kategorie[nomen] genus[maskulin] kasus[genitiv]
        numerus[singular] person[dritte] pronomen[nein]);

'Männer'
        (lexem[mann] kategorie[nomen] kasus[nominativ,genitiv,akkusativ]
        numerus[plural] person[dritte] pronomen[nein]);

'Männern'
        (lexem[mann] kategorie[nomen] kasus[dativ] numerus[plural]
        person[dritte] pronomen[nein]);

'Männer'
        (lexem[mann] kategorie[partikel] kompositum[+] numerus[plural]
        verwendung[attributiv]);
```

Figure 20 Generated word forms belonging to the lexeme 'mann'

Everything is OK here. But this might not be the case while the work is still in progress. It may even be dubious why the correct forms don't appear. If one wants to reenact the transitions through the lexicon network, one should enable the option *Show traversed paradigms*. The output for debugging now contains the traversed paradigms in arrow brackets:

```
Generated word forms of lexeme 'mann':

'Mann<msg-s-es><wortende>'
        (lexem[mann] kategorie[nomen] genus[maskulin]
        kasus[nominativ,dativ,akkusativ] numerus[singular] person[dritte]
        pronomen[nein]);

'Mann<msg-s-es>e<wortende>'
        (lexem[mann] kategorie[nomen] genus[maskulin] kasus[dativ]
        numerus[singular] person[dritte] pronomen[nein]);

'Mann<msg-s-es>es<wortende>'
        (lexem[mann] kategorie[nomen] genus[maskulin] kasus[genitiv]
        numerus[singular] person[dritte] pronomen[nein]);
```

```
'Mann<msg-s-es>s<wortende>'
        (lexem[mann] kategorie[nomen] genus[maskulin] kasus[genitiv]
        numerus[singular] person[dritte] pronomen[nein]);

'Männ<pl-er>er<wortende>'
        (lexem[mann] kategorie[nomen] kasus[nominativ,genitiv,akkusativ]
        numerus[plural] person[dritte] pronomen[nein]);

'Männ<pl-er>ern<wortende>'
        (lexem[mann] kategorie[nomen] kasus[dativ] numerus[plural]
        person[dritte] pronomen[nein]);

'Männ<pl-er>er<kompos-pl>'
        (lexem[mann] kategorie[partikel] kompositum[+] numerus[plural]
        verwendung[attributiv]);
```

Figure 21 Generated forms with an indication of traversed paradigms

Generator > Generate Word Forms from Lexeme has another option: *Select part of speech.* This is interesting if the system of paradigms includes derivations and one wants to look up just some of them. For example, selecting the lexeme 'denken' (think) and the part of speech 'nomen' (noun), results in the nouns that are derived from the verb "denken":

```
Generated word forms of lexeme 'denken' (nomen):

'Denken'
        (lexem[denken] kategorie[nomen] derivation[vorgang]
        genus[neutrum] kasus[nominativ,dativ,akkusativ] numerus[singular]
        person[dritte] pronomen[nein] schreibung[gross]);

'Denkens'
        (lexem[denken] kategorie[nomen] derivation[vorgang]
        genus[neutrum] kasus[genitiv] numerus[singular] person[dritte]
        pronomen[nein] schreibung[gross]);

'Denker'
        (lexem[denken] kategorie[nomen] derivation[agens] genus[maskulin]
        kasus[nominativ,dativ,akkusativ] numerus[singular] person[dritte]
        pronomen[nein] schreibung[gross]);

'Denkers'
        (lexem[denken] kategorie[nomen] derivation[agens] genus[maskulin]
        kasus[genitiv] numerus[singular] person[dritte] pronomen[nein]
        schreibung[gross]);

'Denker'
        (lexem[denken] kategorie[nomen] derivation[agens]
        kasus[nominativ,genitiv,akkusativ] numerus[plural] person[dritte]
        pronomen[nein] schreibung[gross]);

'Denkern'
        (lexem[denken] kategorie[nomen] derivation[agens] kasus[dativ]
        numerus[plural] person[dritte] pronomen[nein] schreibung[gross]);
```

```
'Denkerin'
        (lexem[denken] kategorie[nomen] derivation[agens] genus[feminin]
        kasus[nominativ,genitiv,dativ,akkusativ] merkmal[weiblich]
        numerus[singular] person[dritte] pronomen[nein]
        schreibung[gross]);

'Denkerinnen'
        (lexem[denken] kategorie[nomen] derivation[agens]
        kasus[nominativ,genitiv,dativ,akkusativ] merkmal[weiblich]
        numerus[plural] person[dritte] pronomen[nein] schreibung[gross]);
```

Figure 22 Generating forms of a particular part of speech only

The morpho-syntactic system is complete, if a test file is processed satisfactorily that contains at least one example of any morpho-syntactic phenomenon of the language in question. The next problem is updating the vocabulary of open classes like verbs, nouns and adjectives. This means adding tens of thousands of words. The easiest way to encode this huge amount is by cardinal forms.

The minimum of forms must be determined that allow to derive all other forms. Studying the variation of forms is the next step which leads to cardinal patterns. Each pattern must be assigned to a morphological class. Hence, morphclasses must be drawn up as well. It is good practice to attach cardinal forms as an example to each pattern and each morphclass. These examples of cardinal forms should be collected in a separate test file as well.

If this test file is fed to *Converters > Cardlforms to Morphunits* then each entry is turned into a morphological unit. The output is easy to scan in order to debug the cardinal patterns. If a result seems inconsistent, the option *Show debug output* may be activated. In this case, each step of matching a form with a term in the pattern is documented in the log file. The first lines of the input and the output of the converter regarding German verbs look like this:

```
Input:
    <cardlform>machen machst macht machtest machtest gemacht</cardlform>
    <cardlform>hauen haust haut hautest hautest gehauen</cardlform>
    <cardlform>verreisen verreist verreist verreistest verreistest verreist</cardlform>

Output:
    <morphunit lexeme="machen" morphclass="v1-sw-0-ge" pattern="machen">
    <stem number="1">mach</stem></morphunit>

    <morphunit lexeme="hauen" morphclass="v1-swn-0-ge" pattern="hauen">
    <stem number="1">hau</stem></morphunit>

    <morphunit lexeme="verreisen" morphclass="v1-sw-s" pattern="verreisen">
    <stem number="1">verreis</stem></morphunit>
```

Figure 23 Conversion of cardinal forms into morphological units

Now everything is prepared for reaching broad coverage quite fast. Drawing up cardinal forms by an experienced linguist will certainly not require more than half a minute per item. The converter *Converters* > *Cardlforms to Paradigms* creates <drl> code from these resources which can be loaded in the database.

Corpora must be scanned in order to find words that have not yet been encoded. For this purpose, one should scan the corpus with the function *Morphology* > *Lookup* > *File.* If one turnes on the option *unknows only,* those word in the corpus that can **not** be tagged are listed. These words must be added to the cardinal forms. Then the corpus check is repeated.

If there is no unknown-word output any more then the vocabulary is complete (with respect to the corpus). Now the PLAIN IDE can be applied as a tagger.

References

Hockett, Charles F. (1954): “Two models of grammatical description”. *Word* 10: 210-234.Koskenniemi, Kimmo (1983): Two-level morphology : a general computational model for word-form recognition and production. Publications (Helsingin yliopisto. Yleisen kieliteteen laitos 11)

Spencer, Andrew (1991) Morphological Theory. Oxford: Blackwell.

Hellwig, Peter (2003) "Dependency Unification Grammar". In: V. Agel , L.M. Eichinger, H.-W. Eroms, P. Hellwig, H.-J. Heringer, H. Lobin: Dependency and Valency. An International Handbook of Contemporary Research. Berlin: Mouton.