TR-I-0362

Morphological Facilities for German Generation in ASURA

Mark Seligman

March, 1993

Abstract

This document describes all of the morphological facilities used for German generation in the ASURA speech-to-speech translation system (CSTAR demo version of January 28, 1993): a morphological network and related programs; a morphological dictionary and associated lexical programs; a set of morphological rules which add endings to word stems (and sometimes alter the stems as well); and functions for postprocessing punctuation and capitalization. It emphasizes information needed for maintenance and extension.

ATR自動翻訳電話研究所
ATR Interpreting Telephony Research Laboratories

Contents

Chapter 1

Introduction

This document describes all of the morphological facilities used for German generation in the ASURA speech-to-speech translation system (CSTAR demo version of January 28, 1993):

- a morphological network and related programs;
- a morphological dictionary and associated lexical programs;
- a set of morphological rules which add endings to word stems (and sometimes alter the stems as well);
- and functions for postprocessing punctuation and capitalization.

After a brief overview of morphological processing and a listing of relevant files, each facility will be discussed in a separate chapter. Emphasis will be on the information needed for maintenance and further development.

The primary reference source for ASURA's morphological system is [Kikui 93]. This resource describes basic system functions and discusses English morphological facilities.

Concerning maintenance: Some remaining bugs or problem areas must be expected. While the facilities function without errors in translating twelve dialogs concerning conference registration ("mset" dialogs da through d12), there has been no chance to systematically test every path through the morphology network, every dictionary entry, every rule, etc. Another likely source of error: in an attempt to cover German morphology as generally as possible, we have treated in passing some phenomena which appear rarely or not at all in the test dialogs. (For example, we support inflection of derived nouns like *die Vortragenden*, although there is only one example in our corpus; and we enable the use of verb present participles as adjectives — as in *ein laufender Hund* — even though no examples occur.)

Concerning future development: We will point out below several areas in which our treatment has been incomplete or unsatisfactory.

1.1 Overview of Morphological Processing

Let us now begin our overview. During morphological processing, terminal leaves representing German words are harvested in order from a complete syntactic tree. Each word leaf is a *feature structure*, a collection of features and their values. The features may indicate, for instance, that the current word has `[cat aux]` or `[cat v]` (verb), where `cat` means category, or part of speech. Each feature structure must also contain a `lexid` feature, giving the unique lexical identification of the current word, for example `[lexid muessen-1]` or `[lexid gehen-1]`. Additional features may indicate the word's person, number, tense, etc.

The feature structure of each leaf (word) in turn is given to the *morphology network*. At each node of the network, the value of a different feature is examined. For instance, at one node, near the top of the network, the value of the word's `cat` feature is checked: is it `n(oun)`, `v(erb)`, `aux`, etc.? Each answer will point toward a lower node, where a different feature's values are checked.[1] If the current word has `[cat v]`, for example, we go to a node where the `tense` feature is checked: is the value `pres`, `past`, or `future`? and so on. Once this series of questions is complete — that is, once a terminal network node has been reached — the morphology network delivers as output either (a) an inflected word string ready for pronunciation or (b) a position within a dictionary entry where a string can be found. The string may represent the entire word (e.g. "gehen"); or it may represent just its stem ("geh"), and in this case a second position in the dictionary entry must also be indicated where a *morphological ending rule* can be found. The ending rule then operates on the stem string, and an inflected word is returned.[2]

The morphology network avoids visiting the dictionary (case a) when the current word is inflected irregularly, for example if it is an auxiliary verb. (For `[cat aux]`, `[lexid muessen-1]`, `[person 1]`, `[number sing]`, and `[tense pres]` the morphology network directly produces "mu:s", as in "ich "mu:s".)[3] All closed-class lexical items (e.g. pronouns, determiners) are handled within the morphology network, without dictionary entries.

However, the dictionary *is* visited when inflection is regular (case b), and of course this occurs more often — for nouns, most verbs, adjectives, and so on.

A look at a dictionary entry[4] may be helpful. Consider the dictionary entry for lexid `gehen-1`:

```
("gehen" V (:ALL "geh" "geh" "ging" "ging" "gegangen" H NIL
                E ST T EN T Z ST Z EN T))
```

If the current word has `[cat verb]`, `[lexid gehen]`, `[person 1]`, `[number sing]`, and `[tense pres]`, the morphology network will indicate positionally that word stem string "geh" and ending rule E should be retrieved from this dictionary entry. After this rule operates on this stem, the result will be "gehe", as in "ich gehe".

Several functions have been written to partly automate the production of dictionary entries and dictionaries. All are described below. Some such functions contain *morphophonemic* knowledge of German: one program, for instance, knows how to form singular genitive endings for nouns, considering the gender, the final letters of the stem, membership within a list of exceptional cases, and so on. The function thus automatically indicates which ending rules should be included in a noun's dictionary entry.

Morphological rules add a certain ending to a word stem, sometimes altering the stem. Each rule is defined by a call to the function DEF-MG-RULE:

```
def-mg-rule(rulename  no-of-chars-to-remove  string-to-attach  obsolete)
```

The rule named E, for instance, adds the ending "e" without making any stem changes:

```
(def-mg-rule e 0 "e" nil)
```

[1]The morphology network usually examines the features and values of the *current* word. However, it is also possible to query about the features and values of the *immediately preceding* word, or even about those of the word which will be processed next. Morphological decisions about the current word can thus be made *context sensitive*; and this capacity can be used to handle juncture phenomena like English contraction or German cliticization.

[2]ASURA's earlier English morphology system always added inflectional endings to a *default* word stem: the value of the lex feature of the current word. Thus, to enable English inflection, it was only necessary to indicate *one* dictionary entry position, that of an ending rule. In German, by contrast, the stem may vary. Thus it is sometimes necessary to indicate *two* positions, the position of the right stem and the position of the ending. The functions which make this extension possible are described below.

[3]To enable unambiguous conversion to standard print characters ü, ä, ö, and ßin output strings, we adopt a special spelling convention for umlauts (:u, :a, :o) and scharfes S (:s). The more readable alternative spellings ue, ae, oe, and ss are unfortunately ambiguous: they sometimes result from junctures.

[4]There is some danger of confusing the morphological dictionary with the syntactic lexicon. In ASURA, these are separate data resources with separate functions. A morphological dictionary entry contains the stems and endings which are necessary to assemble a word's surface string: it helps construct the *output* of morphological processing. A lexical syntax rule, by contrast, provides a partial feature structure description of a word, and thus helps to construct the complete syntactic tree which becomes the *input* for morphological processing.

Thus, as we have seen, "geh" altered by rule E gives final output "gehe", as in "ich gehe". (For comparison, the rule LE changes a stem like "klingel" into "kling" before adding "le".)

Notice that a rule's operations are specified via its internal structure, not by its name, which is simply a memory aid. Notice also that a rule name is a Lisp symbol, while a word stem is a string.

When all of the words (leaves) of the syntactic tree have been processed, the result is a Lisp list of strings, e.g. (" Harald" " weint" "." "," " aber" " Helmut" " lacht" "."). The string includes punctuation marks, since these are included as terminal elements in the syntactic tree. *Postprocessing* then creates a single string which will become the final result of German generation: "Harald weint, aber Helmut lacht."

The strings in the list cannot simply be concatenated, however, since punctuation and capitalization can create special problems. For example, the punctuation of an embedding construction can override the default punctuation of an embedded construction. (The medial punctuation for a compound sentence, for instance, normally overrides the default final punctuation of the embedded clauses.) Further, since the German grammar can deliver multiple sentences, capitalization after a final punctuation mark sometimes becomes necessary.[5]

In the chapters which follow, we examine the four facilities in more detail, each in its own section: the morphology network, the morphological dictionary, morphological rules, and postprocessing.

1.2 Pathnames of Files

First, however, we give full pathnames for the files mentioned (by filename only) throughout the text. We include some of the code below where we think it will be most helpful, but recommend keeping the complete, commented versions handy while reading.

All are stored in

`as25:/home/tropf/gen-off/`

- The most recent version of the morphology network:

  `as25:/home/tropf/gen-off/mgen/mg-net/<date>`

- Morphology network functions:

  `as25:/home/tropf/gen-off/mgen/mg-net/functions/net-macro-patch.lisp`

- Numerous lexical survey and input files:

  `as25:/home/tropf/gen-off/lexical/`

- Lexical functions:

  `as25:/home/tropf/gen-off/lexical/readfns.lisp`
  `as25:/home/tropf/gen-off/lexical/lexfns.lisp`

- Morphological dictionary:

  `as25:/home/tropf/gen-off/dict/<date>`

- Morphological rules:

  `as25:/home/tropf/gen-off/mgen/mg-rule/<date>`

- Postprocessing functions:

  `as25:/home/tropf/gen-off/mgen/mg-net/functions/mgen-main-patch.lisp`

[5]The earlier postprocessing for English did not support such punctuation override or multiple sentence output. The programs which give the German postprocessor added flexibility are described below.

Chapter 2

The Morphology Network

The morphology network receives a feature structure representing a German word. After considering the values of the features in a specified order, the function which traverses the net delivers a word ready for output. The output word may be included in the network itself; or it may be retrieved from the morphological dictionary. It may be retrieved as a single element from the dictionary entry (e.g. "gehen"); or a stem and an ending rule may be retrieved (e.g. "geh" and E), and in this case the rule adds to the stem (perhaps changing the stem as well) to produce the final inflected output (e.g. "gehe").

2.1 Defining Nodes and Arcs

The network is built of nodes and descending arcs. Nodes and arcs are defined using the function DEF_MG_NET. A single call defines one node and the arcs leading down from it.

```
(def_mg_net
    :name <node name>
    :path (ftr1 ftr2 ... ftrN)
    :default :warning
    :body ((value1 action action action ...)        ;arc1
           (value2 action action action ...)        ;arc2
           ...
           (value3 action action action ...)))      ;arcN
```

The node name is usually arbitrary, but may be mentioned in other function calls. (There is one exception: the root node must be named `mg_net-main`.)

2.1.1 Path Specification in a Network Node

The *path specification* in a node definition is an ordered series of features, referring to the feature structure of a terminal node in the syntax tree. (The relevant feature structure is usually that of the *current* node, but see below.) The path `(syn agr person)`, for instance, says, "Fetch the value along the path `(syn agr person)` in the current leaf's feature structure". (`Agr` stands for agreement.)

The allowed values for this path in the German generation grammar are `1` (first person), `2fam` (2nd person, familiar), `2pol` (2nd person, polite) and `3` (3rd person).

Notice that the notation `(:previous [path])` tests the value of [path] in the preceding word rather than the current one.

2.1.2 Body Specification in a Network Node

The *body* of a node definition defines downward arcs as a series of value/action tuples. (There may be several actions, but not every combination of actions makes sense. The possible actions are described below.) In the node defined below, if the value of the specified path is `1`, a certain node will be visited ("called") next; if the value is either `2fam` or `2pol`, a different node will be visited; and if the value is `3`, a third node will be visited.

```
(def_mg_net
  :name    mg_net-v_regular_finite_indicative_pres
  :path    (syn agr person)
  :default :warning
  :body    ((1 (:call mg_net-v_regular_finite_indicative_pres_1))
            ((:set 2fam 2pol) (:call mg_net-v_regular_finite_indicative_pres_2))
            (3 (:call mg_net-v_regular_finite_indicative_pres_3))))
```

Test Operators

Several test operators are provided for the test part of a test/action combination:

- The above example shows that simple values, such as `1` and `3`, can be used as tests.
- Notice also the notation `(:set value1 value2)`, which allows *disjunctive* value tests: the action will be taken if either `value1` or `value2` is found.
- Another special test operator is `:previous`. The combination `(:previous "ich")`, for instance, tests whether the immediately preceding word was "ich".
- The operator `:next` can be used comparably to point toward the immediately following word.
- Finally, the operator `otherwise` provides a default test condition.

Actions

Several action operators are provided for the action part of a test/action combination.

We have seen that the action `:call` means to visit (i.e. traverse to) the node which is named. The other actions used in the German morphology network can be indicated using a *string*, or using the keywords `:dict`, `:inflectroot`, `:warning`, `:left_space`, or `:nop`. We now give examples of each usage.

String At the node below, we test the current word's number, along path `(syn agr number)`. If the value is `sing`, the string "mu:s" (as in "ich mu:s") will be the output of the morphology network; or, if the value is `plur`, the output will be "m:ussen". In these cases, the morphological dictionary is never visited. Instead, the morphology network's own output becomes the final morphological output for this word.

```
(def_mg_net
  :name    mg_net-muessen_finite_indicative_pres_1
  :path    (syn agr number)
  :default :warning
  :body    ((sing  "mu:s" )
            (plur  "m:ussen" )))
```

:dict If the current word is the past participle of a verb, we reach the following node. We then test the word's `(syn type)` path. If the value is `nonseppref` ("does not have a separable prefix"), we fetch a single element from the verb's dictionary entry: the one found at position 6, which of course should be the verb's past participle, e.g. "gegangen".

```
(def_mg_net
  :name    mg_net-v_pastpart
  :path    (syn type)
  :default :warning
  :body    ((seppref (:dict :nadine91 :lex-cat-id 6) (:left_space nil)  )
            (nonseppref (:dict :nadine91 :lex-cat-id 6) )
            (otherwise :warning)))
```

Concerning the use of numerical positions within a dictionary entry, see discussion of the `:inflectroot` functions, below.

:inflectroot If the current word is a regular verb, used finitely in the indicative present tense, first person, we then test its number along path `(syn agr number)`. If the value is `sing`, we fetch the appropriate verb stem ("geh") from position 2 in the dictionary entry and the appropriate ending rule (E) from position 9. The inflected verb "gehe" (as in "ich gehe") will be returned. By contrast, if the value were `plur`, the same stem would be fetched with a different ending rule (EN), found in position 12, to given "gehen".

```
(def_mg_net
  :name    mg_net-v_regular_finite_indicative_pres_1
  :path    (syn agr number)
  :default :warning
  :body    ((sing  (:inflectroot :nadine91 :lex-cat-id (2 9))) ;ich
            (plur  (:inflectroot :nadine91 :lex-cat-id (2 12)))))
```

Note: The `:inflectroot` action is enabled for German generation only. The enabling functions are described below. Concerning the use of numerical positions within a dictionary entry, see discussion of the `:inflectroot` functions, below.

:warning If this action is found, an error message is printed before the final text string appears in the output. By default, it reports the name of the node where a problem occurred. An error message can also be included, e.g. `(:warning "bad part of speech entering morph network")`.

:left_space We have already seen the following function call. It defines a node which is reached if the current word is the past participle of a verb.

```
(def_mg_net
  :name    mg_net-v_pastpart
  :path    (syn type)
  :default :warning
  :body    ((seppref (:dict :nadine91 :lex-cat-id 6) (:left_space nil))
            (nonseppref (:dict :nadine91 :lex-cat-id 6) )
            (otherwise :warning)))
```

Notice now that the same dictionary fetch is made whether the word's `(syn type)` value is `seppref` ("separable prefix") or `nonseppref`: in both cases, the element in the 6th position of the dictionary entry is taken. However, in the `seppref` case, the word should be printed without the space to its left which would normally appear. It

will thus appear to be fused with the separable prefix which should immediately precede it. For example, the words "mit" and "gegangen" will be written to the output stream as "mitgegangen" (rather than the default "mit gegangen").

:nop This is the null operator, indicating "no operation": in other words, the value of the `lex` feature of the current word should become the output of the morphology network.

It is used, for instance, when the current word belongs to a non-inflected cat, e.g. adv.

2.2 Comments on Selected Network Nodes

We now comment on selected nodes or subnetworks of the morphology network. Since we cannot reproduce large sections of the network here, we assume the reader can refer to the code.

2.2.1 mg_net-main (network root)

The highest (root) node must have this name. Our root node unconditionally calls `mg_net-trace-check`.

2.2.2 mg_net-trace-check (checking for traces)

This node pre-checks for *traces*: special feature structures whose `(trace)` path has the value `t`. Such trace feature structures result from the grammar's HPSG-style treatment of long-distance dependencies. They are printed as null strings (""), and with no preceding space (using the action `(:left_space nil)`). Thus they become invisible in the output string.

2.2.3 mg_net-cats (listing of word categories)

This is the node in the network where parts of speech (values of the `cat(egory)` feature) are recognized.

The following cats require no inflection (their `lex` value is returned unchanged via the action `:nop`):

```
adv coord idiom particle propp sign vpref
```

Most other cats branch to their own subnets. Exception: q(uantifier) calls the same subnet as d(eterminer), since these cats are identically inflected.

```
(def_mg_net
  :name    mg_net-cats
  :path    (syn cat)
  :default :warning
  :body    ((a  (:call mg_net-a))              ;adjective
            (adv :nop)                         ;adverb
            (aux (:call mg_net-aux))           ;aux verb
            (comp :nop)                        ;complementizer
            (coord :nop)                       ;coordinator
            (d  (:call mg_net-d))              ;determiner
            (idiom :nop)                       ;idiom
```

```
(n (:call mg_net-n))                ;noun
(num (:call mg_net-num))            ;numeral
(p (:call mg_net-p))                ;prep
(particle :nop)                     ;particle
(pronp (:call mg_net-pronp))        ;pro-noun-phrase
(propp :nop)                        ;pro-prep-phrase
(q (:call mg_net-d))                ;quantifier
(sign :nop)                         ;punctuation
(v (:call mg_net-v))                ;verb
(vpref :nop)                        ;verb prefix
(otherwise (:warning "bad part of speech entering morph network"))))
```

2.2.4 mg_net-a (adjectives)

Adjective dictionary entry format is as follows:

```
(dict-form    A   (:ALL     compar-stem   super-stem
      0       NA      1          2             3
("gut"        A   (:ALL       "besser"       "best"

-------------------------------------------------

er-ending  e-ending  es-ending  en-ending  em-ending))
     4         5         6          7         8
     ER        E         ES         EN        EM   ))
```

Numbers refer to ordering in the sublist whose first element is a lexid symbol (usually :ALL). The number 0 can be treated as a reference to the dict-form.

Adjectives are used *attributively* when they modify the following noun (*der gute Mann*). They can also be used as predicate adjectives (*der Mann ist gut*) or adverbially (*der Mann spricht gut*). Adjectives whose path `(syn type)` has the value `attr` are inflected; while those with value `advpred` ("adverb or predicate adjective") take an *en* ending if they are superlative (*(am) schnellsten*), but otherwise are not inflected.

There are three possible values for the path `(syn degree)`: `pos(itive)` (*gut*), `comp(arative)` (*besser*), and `sup(erlative)` (*best*). The dict-form serves as the stem for inflecting positive adjectives. We list as dictionary entries, rather than compute, the comparative and superlative stems.

For adjectives, it is convenient to associate ending positions 4 - 8 with specific phonological endings: position 7, for instance, always gives an EN ending. Compare this "phonological" ending pattern with the "morphological" ending pattern for verbs. For verbs, an ending position is associated with a specified combination of case and number, (such as genitive singular); and a combination may have several quite different phonological expressions, (the sing-gen morphophoneme, for instance, may be zero, S, ES, etc.).

Note carefully our terminology regarding "strength". We recognize two patterns for NP's containing both determiner and adjective: weak (*der gute Mann*), and strong (*ein guter Mann*). In a weak NP, both determiner and adjective are called weak; in a strong NP, both are called strong. This treatment facilitates agreement, but may be confusing. See the section on determiners for fuller discussion.

2.2.5 mg_net-aux (auxiliary verbs)

Sein, *haben*, *d:urfen*, *k:onnen*, *m:ussen*, *sollen*, *werden*, *wollen*, and *m:ogen* are handled under the category aux.

By request from the syntactic component, subjunctiveii uses of *m:ogen*, as in *Ich m:ochte*, meaning "I'd like",

can be treated as indicative, present tense uses of a pseudo-auxiliary "m:ochten". However, more orthodox treatment — as *m:ogen*, subjunctiveii — is also enabled.

Sein, *haben*, and "mo:chten" can act as main verbs instead of aux, as in *Er ist ein Mann*, *Er hat einen Buch*, or *Er m:ochte einen Buch*. Syntax indicates the proper cat (part of speech) in each case.

2.2.6 mg_net-d (determiners)

Dictionary entry format for d(eterminers) is as follows:

```
(dict-form   DET   (:ALL     stem   zero-ending  er-ending
      0       NA      1        2       3              4
("dieser"     D    (:ALL    "dies"    Z              ER

-------------------------------------------------------

e-ending  es-ending  en-ending  em-ending))
    5          6         7         8
    E          ES        EN        EM    ))
```

Numbers refer to ordering in the sublist whose first element is a lexid symbol (usually :ALL). The number 0 can be treated as a reference to the dict-form.

For determiners, the dict-forms are unusual: for easy recognition, inflected forms ending in "er" are used, e.g. "jener". However, these are for human reading only. They are not used by inflection rules. A determiner stem, e.g. "jen" is used instead.

For determiners, it is convenient to associate ending positions 3 - 8 with specific phonological endings: position 7, for instance, always gives an EN ending. Compare this "phonological" ending pattern with the "morphological" ending pattern for verbs. For verbs, an ending position is associated with a specified combination of case and number, (such as genitive singular); and a combination may have several quite different phonological expressions, (the sing-gen morphophoneme, for instance, may be zero, s, es, etc.).

Note carefully our terminology regarding "strength". We recognize two patterns for NP's containing both determiner and adjective: weak (*der gute Mann*), and strong (*ein guter Mann*). In a weak NP, both determiner and adjective are called weak; in a strong NP, both are called strong. This treatment facilitates agreement, but may be confusing.[1]

Notice that the strength or weakness of the "*ein* words" (*ein*, *mein*, and *kein*) depends on their number, gender, and case. These dets are usually "weak", but become "strong" for the number/gender/case combinations sing/mas/nom, sing/neu/acc, and sing/neu/nom.[2]

Strength values are assigned to determiners by syntactic rules, and adjectives must agree.[3]

The current subnet for determiners enables prep/det *cliticization* (e.g. *von* plus *dem* becomes *vom*) for preps *von*, *in*, and *an*. These facilities, however, were not used in the demo version of the German grammar. Instead,

[1]Memory aid: in a "weak" NP, the adjective ending gives little information about case, gender, and number; in a strong one, the adjective ending gives a lot of information. In our terms, if an adjective is weak in this sense, the determiner is, too. Beware, however: in some discussions, an adjective which is "weak" (i.e. has an ending giving little information about case, gender, and number) is said to be compensated by a "strong" determiner (one which has an informative ending) and vice versa.

[2]For these combinations, the determiner ending gives little information about case, gender, and number, so an informative adjective is required. As noted, an NP with an informative adjective is called "strong", and has a "strong" det.

[3]This is presently the only area of the German grammar in which syntactic rules make feature value assignments. Eight rules are needed for each determiner to assign a strength value for every relevant combination of number, gender, and case. Note, however, that the strength feature is *redundant* precisely because it does entirely depend on the values of number, gender, and case. The grammar could omit it entirely, using only these three features for agreement. For now, the strength feature has been retained at the request of syntax.

cliticization was treated within the syntactic component. Thus all determiners received the value no for path `(syn dclitic)`. The cliticization facilities remain in place, however. See the discussion of prepositions for further explanation.

2.2.7 mg_net-n (nouns)

The format of a morphological dictionary entry for a noun is as follows:

```
(sing-stem syn-cat  (:ALL  plur-stem  sing-gen  plur-dat   other))
      0       NA      1       2          3         4         5
("Konferenz"  N     (:ALL "Konferenzen"  Z         Z         Z  ))
```

Numbers refer to ordering in the entry sublist whose first element is a lexid symbol (usually :ALL). The number 0 can be treated as a reference to the dict-form.

There are two stems, sing-stem and plur-stem, and three endings, sing-gen(itive), plur-dat(ive), and "other". ("Other" covers singular nouns in the accusative and dative (sing-acc and sing-dat) and plural nouns in the nominative, accusative, and genitive (pl-nom, pl-acc and pl-gen).)

Note that noun gender is not marked in the morphological dictionary entry, but rather in the relevant lexical syntactic rule.

Singular Nouns

The `:dict` action rather than the `:inflectroot` action is used to access sing-nom nouns, since it is unnecessary to fetch any ending from the dictionary entry.

For singular genitive nouns, `:inflectroot` is used to add the sing-gen ending to sing-stem. The morphological dictionary entry for a given noun indicates the particular morphophonemic form for this genitive morpheme: zero, S, ES, etc. See further the discussion of the GET-SING-GEN function below.

For sing-acc and sing-dat nouns, we add the "other" ending to sing-stem: N or EN for "weak" masculine nouns, zero for all others, as indicated by the dictionary entry. See further the discussion of the GET-OTHER function below.

Plural Nouns

The plur-stem (plural stem) is not computed, but rather supplied in a dictionary entry, e.g. as "b:ucher", "bl:atter", etc.

For plural dative nouns, the plur-dat ending is added to plur-stem: N for most nouns; zero for nouns ending in *n* or *s*; and N or EN for weak masculine nouns.

For plur-nom, plur-acc, and plur-gen nouns, the "other" ending is added to the plur-stem: N or EN for weak masculine nouns, zero for all other nouns. (Note that for weak masculine nouns, we always treat (E)N as a case ending, and never as a plural ending. Thus, for these nouns, plur-stem = sing-stem.)

Derived Nouns

We label derived nouns like *die Vortragenden* as `partaderived`, or "derived from a participle". (In fact, the name is too restrictive: nouns can be derived from other cats as well, e.g. as *der Alte* is derived from an adjective.) In our present treatment, such nouns are not actually derived by rules. Instead, they are listed in the syntactic and morphological lexicons. Their entries are unusual for noun entries because they contain *adjective* (rather than noun) ending rules. When the morphology network recognizes a derived noun — because its path `(syn partaderived)` has value `yes` — it branches to the a(djective) subnet, and inflects the noun just as if it were an adjective.

2.2.8 mg_net-num (numerals)

Ordinal numbers (e.g. *zweite*) are inflected like positive adjectives, and cardinal numbers (e.g. *zwei*) are not inflected.

2.2.9 mg_net-p (prepositions)

The current subnet for prepositions enables prep/det *cliticization* (e.g. *von* plus *dem* becomes *vom*) for preps *von*, *in*, and *an*.

For reasons of syntactic run-time efficiency, these facilities were not used in the demo version of the German grammar. Instead, cliticization was treated within the syntactic component. Thus all prepositions received the value no for path `(syn dclitic)`, and prepositions were never inflected (that is, action `:nop` was always used).

The cliticization facilities remain in place, however. If the value of the path `(syn dclitic)` is `yes`, the prep is produced in its combining rather than full form (e.g. "vo" rather than "von"). In the same way, the following determiner can be produced in its combining form (e.g. "m" rather than "dem"), using the special action `(:left_space nil)` to avoid separation between the words: "vom" rather than "vo m".

2.2.10 mg_net-pronp (pronouns)

We treat *das* and *was* specially at the top of the pronp network. (This treatment is temporary; it indicates only that we are unsure how these pronouns should be subcategorized.) The action `:nop` indicates that they are uninflected.

Other pro-np's fall into four groupings according to their `(syn type)` values:

- pers(onal) and refl(exive) pronps share most of a network, since in many cases they share a single surface form: compare "Sie liebt mich (pers); ich liebe mich (refl)". The two types are distinguished only for 3rd person and 2nd person polite forms: "Sie liebt ihn (pers); er liebt sich (refl)." "Sie liebt Sie (pers); Sie lieben sich (refl)."
- rel(ative) and demonstr(ative) pronps share a network;
- wh pronp's (e.g. *wer*) have a separate network;
- indef(inite) pronp's (e.g. *etwas*) have a separate network, indicating via the action `:nop` that they are uninflected.
- and poss(essive) pronp's are redirected to the determiner subnet

2.2.11 mg_net-q (quantifiers)

Quantifiers (q) are distinguished from determiners (d) for syntactic reasons. Morphological treatment is identical, however, so there is no separate subnet for q. Instead, both d and q are directed to the determiner net in node `mg_net-cats`.

2.2.12 mg_net-v (verbs)

The dictionary entry format for verbs is as follows:

```
(dict-form syn-cat  (:ALL base-stem 2/3ps-stem past-stem
      0       NA      1       2          3         4
("fahren"     V     (:ALL   "fahr"    "f:ahr"    "fuhr"

-------------------------------------------------------------------------

subjii-stem ppart haben-sein separable ich-pres du-pres
    5          6       7         8        9       10
  "f:uhr" "gefahren"   H        NIL       E       ST

-------------------------------------------------------------------------

es-pres wir-pres ihr-pres  ich-past  du-past  es-past wir-past ihr-past))
  11       12        13        14       15       16       17       18
   T       EN         T        Z        ST       Z        EN        T   ))
```

Notes concerning the dictionary format:

- Numbers refer to ordering in the entry sublist whose first element is a lexid symbol (usually :ALL). The number 0 can be treated as a reference to the dict-form.
- Positions 7 and 8 are obsolete. The original intent was to give information which is now in lexical syntax rules.
- Endings es-pres and es-past can be understood as short ways of writing, "present ending for third person singular" and "past ending for third person singular". In other words, the verbs which are handled are not only those following pronoun *es* but all third person singular verbs.
- Ending wir-pres can be understood as a short ways of writing, "present ending for *wir*, *sie* ("they"), and *Sie* ("you", singular and plural)." The same person-number combinations are covered by wir-past.
- Position 16 gives the past ending for the third person singular form. As this is always the same as 13 (the past ending for the first person singular), this information is redundant and could be eliminated. In this case, one position would give the ending for both first- and third-person singular past. Such treatment would be more economical, but more confusing.

Notes concerning the verb network:

- Our treatment of *imperatives* is unfinished. We enable *Sie* imperatives only: we give "seien" for *sein*, and point to the infinitive string for other verbs. Treatment of imperatives with *du* and *ihr* would require additional stem positions in the dictionary entry for verbs.
- Our present corpus contains no *verb present particles*, as in *der laufende Hund*. However, the morphology network tentatively enables treatment as follows: when it recognizes a present participle — a verb whose path `(syn vform)` has value `prespart` — it branches to the a(djective) subnet, and inflects the verb just

as if it were an adjective. Participles could be listed in the morphological dictionary (though this is not yet done). Their entries would be unusual: they should contain *adjective* (rather than verb) ending rules. Further, it would be possible to automatically build such entries for all verbs, or to automatically complete partial entries prepared by hand; but we have not enabled either sort of automation yet. (Compare our handling of derived (`partaderived`) nouns: automatic completion of handmade partial entries *has* been enabled.) In general, derivational morphology is not handled by the present system.

- Notice the treatment of *separable verbs* in several nodes of the verb subnet. The German generation syntax treats a separable verb and its prefix as two elements. Morphology must rejoin them under the proper circumstances (especially when the feature `vpos`, "verb position", has the value `v-last`). When such joining is necessary, the special action `(:left_space nil)` is used, so that the verb itself will be printed to the output stream without the usual space to its left. An alternative solution, in which syntax itself made the join, led to syntactic rule duplication and thus to slower generation times. In the current solution, large sections of the verb network must be duplicated for separable (`[type seppref]`) and non-separable (`[type nonseppref]`) verbs, but this duplication affects generation time very little.
- By request of the syntactic component, possible values for the path `(syn agr person)` are `1`, `2fam` (*du* or *ihr*, depending on the value of `(syn agr number)`), `2pol` (singular or plural *Sie*), and `3`. An alternative analysis, in which familiarity is treated as a separate feature with values familiar or polite, would of course be possible.
- Subjunctiveii (the "past subjunctive") has been implemented for three verbs only, *werden*, *m:ogen*, and *k:onnen*. Subjuntiveii stems are properly supplied for all dictionary entries, however.

2.3 Morphology Network Functions

ASURA's earlier English morphology system always added inflectional endings to a *default* word stem, which was the value of the lex feature of the current word's feature structure. Thus, to enable English inflection, it was necessary to indicate at most *one* dictionary entry position, that of an ending rule. In German, by contrast, the stem may vary according to the tense, mood, etc. Thus it is sometimes necessary to indicate *two* positions in a dictionary entry, the position of the proper stem and the position of the ending. We now describe the functions which make this extension possible.

We define a new network action `:inflectroot`, which visits the dictionary twice; first to fetch a proper root, e.g. past, and then again to find a rule which can add endings to (and perhaps modify) the fetched root. For example, input "gehen" might fetch "ging" plus rule EN to give "gingen".

Note the format of an `:inflectroot` action: *two* numbers are supplied, one for each visit to the dictionary, as in `(sing (:inflectroot :nadine91 :lexcat (2 7)))`.

Program changes have been made in two places:

- First, in the system file `net-macro.lisp`,
  - a cond clause has been added to the definition of the Lisp function MGN_EXECUTE-ACTIONS:

    ```
    ((and (consp action) (eq (car action) c-inflectroot_key))
     (setq lex (mgn_inflect_root f_node lex (cdr action))) )
    ```

  - and a constant variable C-INFLECTROOT_KEY has been defined with the value `:inflectroot`:

    ```
    (defconstant c-inflectroot_key :inflectroot)
    ```

- Second, a patch file `net-macro-patch.lisp` has been created. It is loaded after the normal generation system load is complete in order to define two *new* functions: MGN_INFLECT_ROOT and MGN_GET_ROOT. We now describe their operation.

```
*****************************************************************
mgn_inflect_root (f_node lex specs &aux dict_data)
*****************************************************************
```

This new function is called if the keyword `:inflectroot` is recognized via the COND clause shown above. It normally takes two steps:

- it fetches the root of the current word from the dictionary entry using the new function MGN_GET_ROOT
- it calls the system function MGN_APPLY-DICT_RULE (patched for German, in `net-macro-patch.lisp`) in order to inflect the root, using a morphological rule which it fetches from the dictionary entry

However, if no dictionary entry exists for the current word, a default output is used: the value of the current word's `lex` feature.

```
(defun mgn_inflect_root (f_node lex specs &aux dict_data)
  (let* ((dict_name (first specs))
         (key       (second specs))
         (dict_data (mgn_get-dict dict_name key f_node lex)))
    (if dict_data
        ;;Fetch root from dict.
        (let* ((root_position (first (third specs)))
               (inflection_position (second (third specs)))
               (root
                (mgn_get_root f_node root_position)))

          ;;Inflect and return the fetched root
          (mgn_apply-dict_rule
           f_node root dict_data inflection_position))

        ;;else if no dict data, use lex of f_node as default
        (mgn_get-lex_value f_node) )))
```

```
*****************************************************************
mgn_get_root (f_node root_position)
*****************************************************************
```

Fetches a root string for the current node, given a position in a dictionary entry.

- The system function MG-FETCH returns a master-entry in the following format:

  ```
  (*MASTER* (0 "verstehen" VERB (:ALL "versteh" "versteh" "verstand"
       "verstuend" "verstanden" H NIL E ST T EN T Z EST Z EN ET)))
  ```

- If the root_position is 0, the string after 0 is accessed. Otherwise, a position is accessed in the sublist whose first element is a lexid symbol (usually :ALL).
- If the result is a string, it is returned; otherwise, an error message is printed.

```
(defun mgn_get_root (f_node root_position)
  (let* ((lex (mgn_get-lex_value f_node))
         (master-entry (mg-fetch lex))
         root)
    (if (equal 0 root_position)
        ;;refer to format of master entry above
        (setq root (second (second master-entry)))
        (setq root (nth-position root_position (fourth (second master-entry)))))
    (if (stringp root)
        root
        (format_msg "~%ERROR: fetched root ~S is not a string" root))))
```

The file `net-macro-patch.lisp` also contains a few auxiliary functions.

2.4 Limitations of the Present Approach

The `:inflectroot` action was intended to enable German inflection with few changes to the existing English morphology functions. For this reason, this action continues to use numerical positions as indexes into a dictionary entry. This approach is not satisfactory, however — for either German or English — because it is difficult to debug and to modify. (When programming, one must continually refer to a format chart; if the dictionary format changes, new numbers must replace old ones in the morphology network; and so on.) If the German morphology system is upgraded, symbolic reference to dictionary entries should be enabled. For example, the action statement `(:inflectroot :nadine91 :lex-cat-id (2 9))` should become `(:inflectroot :nadine91 :lex-cat-id (base-stem ich-pres))`. A lookup table could enable translation to (and hide) the position numbers.

The numbering system itself may be confusing. Numbers do not refer to ordering in the entire dictionary entry. Instead, they refer to ordering within the contained sublist whose first element is a lexid symbol (usually :ALL). For example, here is the format for noun entries:

```
(sing-stem syn-cat  (:ALL  plur-stem  sing-gen  plur-dat   other))
      0       NA      1       2          3         4         5
```

The number 0 can be treated as a reference to the first element of the dictionary entry, here sing-stem. (From the implementation viewpoint, 0 is actually a direct reference to the value of the `lex` feature of the current node, but both strings must be identical.)

We should also mention the possibility of a more radical change of design: terminal leaves of the syntax tree are now *words*; but they could be *morphemes* instead, if the syntactic component were appropriately augmented. In this case, to inflect a verb (for example), we would no longer make two dictionary fetches via the present `:inflectroot` action, one for the root and one for the ending. Instead, we would handle the stem and suffix as separate nodes, with at most one dictionary fetch for each. To determine the proper morphophonemic forms for endings, we would often have to exploit the network's ability to examine not only the current node but its neighbors. This approach would probably be more satisfying from a linguistic point of view; the present design, however, stays much closer to the existing English design in both syntax and morphology.

Chapter 3

Dictionary Creation

We now discuss the procedures used to compile a German morphological dictionary and the functions written to facilitate this work.

The lexical work can be divided into two stages:

- surveying the German words needed to translate the "mset" corpus
- preparing dictionary entries for these words

3.1 Surveying the Corpus

We began with an alphabetized concordance of inflected German words, taken programmatically directly from our corpus. (The results are in the file `ger-words-orig`.)

```
(
 "ab"
 "Abendessen"
 "abends"
 "aber"
 "Abmeldung"
 "acht"
 "achten"
 "Achttausend"
 "Adresse"
 "Akira"
 ...
 )
```

We assigned cats (parts of speech, or pos) by hand. Multiple assignments could be made by creating new lines. (The results are in the file `ger-words-with-cats`.)

```
(
("ab" p  )
("Abendessen"  n )
("abends"  adv )
("aber"  comp )
```

```
("Abmeldung"  n )
("acht"  num )
("achten"  num )
("Achttausend"  num )
("Adresse"  n )
("Akira"  n ) ;jap
...)
```

We sorted words according to the assigned cats (parts of speech), using the function SORT-WORDS-INTO-POS. (Results are in the file `ger-words-sorted-by-pos`.)

```
(
("besonderes" A)
("fachliche" A)
("genau" A)
("gut" A)
("Internationalen" A)
("lange" A)
...
("abends" ADV)
("Also" ADV)
 ...)
```

We then created a sublist for each category, alphabetizing each sublist. The function SORT-WITHIN-POS was used. (Results are in the file `ger-words-sorted-within-pos`.)

```
((("besonderes" A)
  ("fachliche" A)
  ("genau" A)
  ("gut" A)
  ("Internationalen" A)
   ...
  ("wahr" A)
  ("weiter" A)
  ("weites" A))
 (("abends" ADV)
  ("Also" ADV)
  ("anders" ADV)
  ("dann" ADV)
  ("Diesmal" ADV)
   ...))
```

Then, by hand, we put each word entry into dictionary entry form.

We eliminated the entries which would not require dictionary entries:

- irregular forms (e.g. aux verbs and pronps) which are handled directly by the morphology network
- cats which are not inflected

Cardinal numerals are not inflected, and were omitted; ordinal numerals (*zweite*, etc.) are inflected like adjectives, and were retained. (Results are in the file `ger-words-reg-all`.)

```
((("besondere" A)
```

```
  ("fachlich" A)
  ("genau" A)
  ("gut" A)
  ("international" A)
  ...
  ("wahr" A)
  ("weit" A)
 )
  ...
 (("Abendessen" N)
  ("Abmeldung" N)
  ("Adresse" N)
  ("Akira" N)
  ("Ankuendigung" N)
  ("Anmeldeformular" N)
   ...))
```

The survey part of our lexical work was then complete. Individual words were occasionally added when our translation target was adjusted.

We have demonstrated the extraction of word lists from local corpora. Of course, future development could instead use ready-made word lists from other corpora or from reference books. For instance, we have already taken a list of strong verb stems from a standard German grammar and have created a corresponding file `strong-verbs-wo-endings` of dictionary entries. These entries could at any time be integrated into the master dictionary using the procedures described below. For now, however, we have kept them separate to simplify debugging.

3.2 Creation of Dictionary Entries

The creation of dictionary entries can itself be divided into two stages: the creation of lexical input files containing partial lexical entries for each word; and the programmatic completion of those lexical entries.

3.2.1 Making Partial Dictionary Entries

We used the survey information to create a *lexical input file* for each inflected cat:

```
nouns-wo-endings
verbs-wo-endings
adjs-wo-endings
num-wo-endings
```

Each lexical input file contains a list of *partial dictionary entries.* Most partial entries are created by hand using a text editor; the partial entries for weak verbs, however, can be created programatically (see next section). Full dictionary entries are created by automatically completing the partial entries.

The partial entries contain data which we could not compute (plurals for nouns, stems for strong verbs, comparative and superlative forms for adjectives) plus a sublist header containing any additional data required for computing endings.

A partial noun entry, for example, contains a singular and plural stem, and a header list showing (a) the gender and (b) an indication of whether the noun was derived from another category. ":Ubernachtungsm:oglichkeit", for example, is a feminine (F) and non-derived (ND) noun whose plural stem is ":Ubernachtungsm:oglichkeiten".

```
((F ND) ":Ubernachtungsm:oglichkeit" N
        (:ALL ":Ubernachtungsm:oglichkeiten")
```

Endings must be added at the end of the sublist whose first element is a lexid symbol (usually :ALL). For this entry, they will all be Z, or zero.

```
(":Ubernachtungsm:oglichkeit" N
       (:ALL ":Ubernachtungsm:oglichkeiten" Z Z Z))
```

Some additional partial entries for nouns:

```
(((F ND) ":Ubernachtungsm:oglichkeit"
         N
         (:ALL ":Ubernachtungsm:oglichkeiten"))
  ((F ND) ":Ubersetzung"
          N
          (:ALL ":Ubersetzungen"))
  ((N ND) "Abendessen"
          N
          (:ALL "Abendessen"))
  ((F ND) "Abmeldung"
          N
          (:ALL "Abmeldungen"))
 ...
 ((M D) "Vortragend"  ;a derived noun
         N
         (:ALL "**no plural**"))
  ...)
```

Verbs have a header indicating whether the verb is strong or weak.

```
(
((s) "abhalten" V (:ALL "abhalt" "abh:alt" "abhielt" "abhielt" "abgehalten" h nil ))
((w) "akzeptieren" V (:ALL "akzeptier" ... "akzeptier" "akzeptier" "akzeptiert" h nil ))
((s) "angeben" V (:ALL "angeb" "angib" "angab" "ang:ab" "angegeben" h nil ))
((w) "anmelden" V (:ALL "anmeld" "anmeld" "anmeld" "anmeld" "angemeldet" h nil ))
((w) "beantworten" V (:ALL "beantwort" ... "beantwort" "beantwort" "beantwortet" h nil ))
((w) "begutachten" V (:ALL "begutacht" ... "begutacht" "begutacht" "begutachtet" h nil ))
...)
```

No extra information is required for adjectives:

```
(
("besondere" A (:ALL "**no comp**" "**no super**" ))
("fachlich" A (:ALL "**no comp**" "**no super**" ))
("genau" A (:ALL "genauer" "genauest" ))
("gut" A (:ALL "besser" "best" ))
("international" A (:ALL "**no comp**" "**no super**" ))
("lang" A (:ALL "l:anger" "l:angst" ))
("maschinell" A (:ALL "**no comp**" "**no super**" ))
("m:oglich" A (:ALL "**no comp**" "**no super**" ))
("nah" A (:ALL "n:aher" "n:achst" ))
...)
```

or ordinal numerals:

```
(
("acht" NUM  (:ALL "**no comp**" "**no super**" ))
("drei:sigst" NUM (:ALL "**no comp**" "**no super**" ))
("f:unft" NUM (:ALL "**no comp**" "**no super**" ))
("f:unfundzwanzigst" NUM (:ALL "**no comp**" "**no super**" ))
("siebenundzwanzigst" NUM (:ALL "**no comp**" "**no super**" ))
("viert" NUM (:ALL "**no comp**" "**no super**" ))
...)
```

The following auxiliary functions are used to access information in header lists. Of course, the header lists are discarded when complete dictionary entries are built.

```
get-noun-gender (noun-entry)
get-noun-derivation-information (noun-entry)
get-verb-strength (verb-entry)
```

3.2.2 Completing Dictionary Entries and Making a Dictionary

To (re)make a morphological dictionary, we first invoke these functions (described in some detail below):

```
add-noun-endings (input-file output-file)
add-verb-endings (input-file output-file)
add-adj-endings (input-file output-file) ;for both adjectives and ordinal numerals
```

using these input files, containing lists of partial entries as input,

```
nouns-wo-endings
verbs-wo-endings
adjs-wo-endings
num-wo-endings
```

to create the following output files, containing lists of complete entries:

```
nouns-w-endings
verbs-w-endings
adjs-w-endings
num-w-endings
```

Files for two closed-class inflected cats — for d(eterminers) and q(uantifiers) — are prepared by hand.

```
dets-w-endings
q-w-endings
```

```
*****************************************************************************
make-master-dict (path a-file d-file n-file num-file q-file v-file output-file)
*****************************************************************************
```

Finally, we invoke MAKE-MASTER-DICT, which simply appends the six -w-endings files and writes the result as the finished dictionary. (Note: The input files are Lisp lists, but the dictionary is written as a stream, with no enclosing parens.)

```
(defun make-master-dict (path a-file d-file n-file num-file q-file v-file output-file)
  (let* ((a (read-from-file (format nil "~a~a" path a-file)))
         (d (read-from-file (format nil "~a~a" path d-file)))
         (n (read-from-file (format nil "~a~a" path n-file)))
         (num (read-from-file (format nil "~a~a" path num-file)))
         (q (read-from-file (format nil "~a~a" path q-file)))
         (v (read-from-file (format nil "~a~a" path v-file)))
         (all (append a d n num q v)))
    (write-loop all (format nil "~a~a" path output-file))))
```

```
*****************************************************************************
remake-dict ()
*****************************************************************************
```

A hardcoded function with no arguments, REMAKE-DICT, was provided to avoid typing paths and filenames at remake time when these are stable. This version also copies the new dictionary to a second file.

```
(defun remake-dict ()
  (add-noun-endings "~/generation-ger/lex/nouns-wo-endings"
                    "~/generation-ger/lex/nouns-w-endings" )
  (add-verb-endings "~/generation-ger/lex/verbs-wo-endings"
                    "~/generation-ger/lex/verbs-w-endings" )
  (add-adj-endings "~/generation-ger/lex/adjs-wo-endings"
                   "~/generation-ger/lex/adjs-w-endings" )
  (add-adj-endings "~/generation-ger/lex/num-wo-endings"
                   "~/generation-ger/lex/num-w-endings" )

  (make-master-dict "~/generation-ger/lex/" "adjs-w-endings"
                    "dets-w-endings" "nouns-w-endings" "num-w-endings"
                    "q-w-endings" "verbs-w-endings" "new.dict")

  (copy-loop "~/generation-ger/lex/new.dict"
             "~/generation-ger/morph/dict/current.dict"))
```

Two additional simple functions are useful for handling and integrating dictionary entries:

```
*************************************************************
alphabetize-dict-entries (input-file &optional output-file)
*************************************************************
```

- Takes a list of dictionary entries from input-file, and alphabetizes it using the first list element as a key.
- Then writes the sorted list to the same file (by default) or to output-file if specified.

```
*****************************************************************************************
integrate-and-alphabetize-dict-entries (input-file1 input-file2 &optional output-file)
*****************************************************************************************
```

- Combines current input-file1, e.g. nouns-w-endings, with input-file2, e.g. new-nouns-w-endings, containing new items (nouns, verbs, adjectives, or numerals) to be added to dictionary.

- Alphabetizes, using the first list element as key.
- Writes the combined, sorted list to a file. By default, output *overwrites* input-file1, giving e.g. a new version of `nouns-w-endings`; or an optional output-file can be specified.

When there are new nouns, verbs, adjectives, or numerals to be added to the morphological dictionary, the usual steps are to add endings to new items (using `add-verb-endings`, etc.) in a separate file and then to combine with the master file of full entries for the relevant cat (`verbs-w-endings`, etc.) using this function. Alternatively, one can integrate items without endings, and then add endings to the integrated file.

3.3 Automating Dictionary Production

We have given an overview of dictionary creation. We now further discuss our attempts to partly automate this creation. We give further details concerning two automatic processes already mentioned (computing stems for weak verbs, automatically adding word endings), and describe for the first time a third automatic process (automatic merging of dictionary entries).

3.3.1 Computing Stems for Weak Verbs

As mentioned, from a handmade list of weak verbs, we can compute partial verb entries, including all stems and past participles. The function MAKE-WEAK-VERB-ENTRIES is used. It returns a list of partial verb entries.

```
make-weak-verb-entries (verb-list)
```

For instance, given an input list including "geh:oren", it returns

```
(("geh:oren" V (:ALL "geh:or" "geh:or" "geh:or" "geh:or" "geh:ort" H NIL))
 ...)
```

3.3.2 Automatically Adding Endings

As mentioned, we automatically complete partial entries for nouns, verbs, adjectives, and ordinal numerals. That is, for these cats, we compute all endings. (An ending is the symbolic name of a rule which can be used to augment and/or modify a stem string.) For example, given an input list containing the above partial entry, ADD-VERB-ENDINGS returns

```
(("geh:oren" V (:ALL "geh:or" "geh:or" "geh:or" "geh:or" "geh:ort" H NIL
                E ST T EN T TE TEST TE TEN TET))
 ...)
```

Endings are computed for nouns, verbs, adjectives, and ordinal numerals, based on one or two information points entered for each word by hand (e.g. noun gender, verb strength). These functions are used:

```
add-noun-endings (input-file output-file)
add-verb-endings (input-file output-file)
add-adj-endings (input-file output-file)
```

Note that ADD-ADJ-ENDINGS is used for ordinal numerals as well as adjectives.

We now describe these functions and their subroutines. We display Lisp code when this may be helpful. For further details, see the complete, commented code in the file `lexfns.lisp`.

```
;;;*****************************************************************
;;; add-noun-endings (input-file output-file)
;;;*****************************************************************
```

- Input a list of partial noun entries from input-file.
- Loop through the partial noun entries. For each partial entry:
  - if the noun is derived, treat it as an adjective: construct a full noun entry by assigning an invariant set of adjective endings (ER E ES EN EM)
  - if the noun is not derived, construct a full noun entry using several subroutines: compute the sing-gen ending using function GET-SING-GEN, compute the "other" ending using function GET-OTHER, and compute the pl-dat ending using function GET-PL-DAT
- once all noun entries are complete (once all the endings have been added) invoke MERGE-ENTRIES-WITH-SAME-LEX
- write the resulting set of complete, merged noun entries to an output-file

We next examine the functions which compute the proper endings for non-derived nouns: GET-SING-GEN, GET-PL-DAT, and GET-OTHER, and their auxiliary functions and global variables.

```
;;;*****************************************************************
;;; get-sing-gen (noun-entry)
;;;*****************************************************************
```

- Determines the singular genitive ending for nouns, according to their gender, stem endings, etc. "Weak" masculine nouns are recognized by referring to lists and by checking for endings which usually indicate weak nouns of foreign origin.
- The sing-gen ending is
  - Z (zero) for fem nouns
  - S or ES for neuter and non-gender nouns marked (x) (The choice between S and ES depends on the final letters of the stem. The function S-OR-ES implements the decision.)
  - For masculine nouns:
    * N for *Herr* (instead of EN, as the morphophonemic rule would normally require)
    * ENS for *Herz*
    * NS for eight exceptional masculine nouns like *Name*, *Gedanke*, etc.
    * N or EN for other weak masc nouns like *Mensch*, *Affe* (The choice between N and EN depends on the final letters of the stem. The function N-OR-EN implements the decision. Weak masculine nouns are identified by the predicate IS-WEAK-MASC-NOUN?)
    * S or ES for all other masc nouns (The choice depends on the final letters of the stem. The function S-OR-ES implements the decision.)

```
(defun get-sing-gen (noun-entry)
  (let ((gender (get-noun-gender noun-entry))
        (stem (second noun-entry)))
    (if (member stem *nouns-wo-singular* :test #'string=)
        'z
```

```
        (case gender
          ((f) 'z)
          ((n x) (s-or-es stem))
          ((m)
           (cond ((equal stem "Herr")
                  'n)
                 ((equal stem "Herz")
                  'ens)
                 ((member stem *8-exceptional-masc-nouns* :test #'string=)
                  'ns)
                 ((is-weak-masc-noun? stem)
                  (n-or-en stem))
                 (t (s-or-es stem))))))))


;;;*****************************************************************
;;; s-or-es (stem)
;;;*****************************************************************
```

Roughly determines the morphophonemics of gen-sing (S or ES?) according to the final letters of the stem. A list of exceptions (words which should take S according to these rules but in fact take ES for reasons of rhythm, style, etc.) is in *EXCEPTIONAL-ES-ENDINGS*.

```
(defun s-or-es (stem)
  (if
   (or
    (is-last-n-chars? stem 1 "s")
    (is-last-n-chars? stem 3 "sch")
    (is-last-n-chars? stem 2 "ss")
    (is-last-n-chars? stem 2 ":s")
    (is-last-n-chars? stem 2 "st")
    (is-last-n-chars? stem 1 "z")
    (member stem *exceptional-es-endings* :test #'string=))
   'es
   's))


;;;*****************************************************************
;;; global variable: *exceptional-es-endings*
;;;*****************************************************************

(setq *exceptional-es-endings* '("Jahr"))



;;;*****************************************************************
;;; n-or-en (stem)
;;;*****************************************************************
```

Roughly determines the morphophonemics of dat-pl: N or EN? Warning! These criteria are oversimplified. Consult a German grammar for exceptions.

```
(defun n-or-en (stem)
  (cond
    ((is-last-n-chars? stem 3 ":ar")
     'en)
    ((or
```

```
      (is-last-n-chars? stem 2 "el")
      (is-last-n-chars? stem 1 "e")
      (is-last-n-chars? stem 2 "er"))
     'n)
    (t 'en)))
```

```
****************************************************************
is-weak-masc-noun? (stem)
****************************************************************
```

Determines whether a noun already known to be masculine is "weak". Warning! These criteria are oversimplified: the endings "ant", etc. are good but imperfect evidence of weakness.

```
(defun is-weak-masc-noun? (stem)
  (or
    (member stem *weak-masc-nouns* :test #'string=)
    (member stem *8-exceptional-masc-nouns* :test #'string=)
    (is-last-n-chars? stem 3 "ant")
    (is-last-n-chars? stem 3 "aph")
    (is-last-n-chars? stem 4 "arch")
    (is-last-n-chars? stem 2 "at")
    (is-last-n-chars? stem 3 "ent")
    (is-last-n-chars? stem 2 "et")
    (is-last-n-chars? stem 3 "ist")
    (is-last-n-chars? stem 4 "krat")
    (is-last-n-chars? stem 3 "log")
    (is-last-n-chars? stem 3 "nom")
    (is-last-n-chars? stem 3 "mon")))
```

```
****************************************************************
global variable: *weak-masc-nouns*
****************************************************************
```

Contains an incomplete list of common weak masculine nouns, grouped by type. As weak masc nouns are added to the lexicon, they should be listed here.

```
(setq *weak-masc-nouns* '(
"Affe" "Bote" "Franzose" "Schwabe"

"Barbar" "Chirurg" "Kamerad" "Katholik" "Tyrann"

"B:ar" "Bauer" "Bayer" "Bub" "Bursche" "Fink" "F:urst" "Graf"
"Held" "Herr" "Hirt" "Mensch" "Nachbar" "Narr" "Oberst" "Ochs"
"Papagei" "Pfau" "Spatz" "Tor" "Untertan"
))
```

```
****************************************************************
global variable: *8-exceptional-masc-nouns*
****************************************************************
```

These are masculine nouns which take NS in the genitive singular. They are considered weak for the purposes of the "other" ending.

```
(setq *8-exceptional-masc-nouns* '("Buchstabe" "Friede" "Funke" "Gedanke" "Glaube"
                                   "Name" "Same" "Wille" ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
get-pl-dat (noun-entry)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Determines the plural dative ending for nouns.

- Pl-dat ending is Z (zero) for non-weak nouns with pl-stem ending in "n" or "s" and for nouns whose dictionary entry contains the string "**no plural**" in the pl-stem position.
- N or EN for all other nouns (including all weak masculine nouns). (The choice is implemented by the function N-OR-EN. Note: we consider that the plural stem is the same as the singular stem (pl-stem = sing-stem) for weak masculine nouns.)

```
(defun get-pl-dat (noun-entry)
  (let ((gender (get-noun-gender noun-entry))
        (pl-stem (second (fourth noun-entry))))
    (cond ((and
            (not (is-weak-masc-noun? pl-stem))
            (or
             ;;plural stem ends in n or s
             (or (is-last-n-chars? pl-stem 1 "n")
                 (is-last-n-chars? pl-stem 1 "s"))
             (string= "**no plural**" pl-stem)))
           'z)
          ;;including all weak nouns
          (t (n-or-en pl-stem)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
get-other (noun-entry)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Determines the "other" ending for nouns: the ending for number-case combinations other than sing-nom, sing-gen, and pl-dat — singular nouns in the accusative and dative (sing-acc and sing-dat) and plural nouns in the nominative, accusative, and genitive (pl-nom, pl-acc and pl-gen).

The ending is N or EN for weak or "exceptional" masc nouns (the choice is implemented by function N-OR-EN) and Z (zero) for all other nouns.

```
(defun get-other (noun-entry)
  (let ((gender (get-noun-gender noun-entry))
        (stem (second noun-entry)))
    (case gender
      ((f n x) 'z)
      ((m)
          (if (is-weak-masc-noun? stem)
              (n-or-en stem)
              'z)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
add-verb-endings (input-file output-file)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

- Loop through partial verb entries taken from input-file.
- For each partial entry, construct a full verb entry, using these ending patterns:

```
weak ending in d or t   (e est et en et ete etest ete eten etet)
other weak              (e st t en t te test te ten tet)
strong                  (e st t en t z st z en t)
```

  (Noun strength information is contained in a header list, later discarded, in each partial entry.)
- Once endings have been added, invoke the function MODIFY-IF-EL-STEM as a post-check. It modifies a verb entry in which the present stem ends in "el", e.g. for *klingel(n)*: instead of the unmarked first person ending adding E, it substitutes the ending rule LE, giving "klingle" instead of "*klingele".
- When the list of full entries is complete (once all the endings have been added) invoke MERGE-ENTRIES-WITH-SAME-LEX
- Write the resulting set of complete, merged verb entries to an output-file.

```
(defun add-verb-endings (input-file output-file)
  (let (output
        strength
        (verbs (read-from-file input-file))
        (weak '(e st t en t te test te ten tet))
        (weakdt '(e est et en et ete etest ete eten etet))
        (strong '(e st t en t z st z en t)))
    (loop for v in verbs do
          (setq strength (get-verb-strength v))
          (let ((2nd (second v))
                (3rd (third v))
                (4th (fourth v)))
            (case strength
              ((w)
               (if;;stem ends in d or t
                (or (is-last-n-chars? (second 4th) 1 "d")
                    (is-last-n-chars? (second 4th) 1 "t"))
                (setq v (list 2nd 3rd (append 4th weakdt)))
                (setq v (list 2nd 3rd (append 4th weak)))))
              ((s) (setq v (list 2nd 3rd (append 4th strong))))
              (t (setq v (list 2nd 3rd (append 4th '(**error**)))))))
          (setq v (modify-if-el-stem v))
          (setq output (cons v output)))
    (setq output (merge-entries-with-same-lex output))
    (write-to-file-no-pp (reverse output) output-file)))
```

```
****************************************************************
modify-if-el-stem (v-entry)
****************************************************************
```

Modifies a verb entry in which the present stem ends in el, e.g. klingel(n). Instead of unmarked first person ending adding E, substitute ending rule LE, giving *klingle* instead of *klingele*.

```
(defun modify-if-el-stem (v-entry)
  (let ((stem (second (third v-entry))))
    (if (is-last-n-chars? stem 2 "el")
        (subst 'le 'e v-entry)
        v-entry)))
```

```
****************************************************************
add-adj-endings (input-file output-file)
****************************************************************
```

Loop through partial adj (or num) entries taken from input-file.

- For each partial entry, construct a full entry, using this invariant ending pattern: ER E ES EN EM
- When list of full entries is complete (once all the endings have been added) invoke MERGE-ENTRIES-WITH-SAME-LEX
- write the resulting set of complete, merged adj (or num) entries to output-file

```
****************************************************************
global variable: *adj-endings*
****************************************************************

(setq *adj-endings* '(er e es en em))
```

3.3.3 Automatic Merging of Dictionary Entries

We now discuss a third automation process for the first time: automatic merging of dictionary entries.

Sometimes separate words have the same surface dictionary form string (`lex`) and the same `cat`. In our corpus, for instance, it is necessary to distinguish two words with `lex` "Deutsch" and `cat` N: one is seen as derived from an adjective and is inflected as an adjective (*ins Deutsche*); while the second is seen as non-derived, and receives normal noun inflection (*auf Deutsch*). (Such situations are actually rare in our corpus — we observe only *Deutsch*, *Englisch*, and *Japanisch* — but might be common in a larger one.)

System conventions require that entries sharing the same *lex* should appear in a merged dictionary entry. Below, "Deutsch" is the shared lex, while `DEUTSCH-1` and `DEUTSCH-2` are lexids. Note the different ending patterns.

```
("Deutsch" N
           (DEUTSCH-2 "**no comp**" "**no super**" ER E ES EN EM)
           (DEUTSCH-1 "**no plural**" ES Z Z))
```

Our partial entry file contains two separate entries. They are automatically merged when the complete entries are made, via the function MERGE-ENTRIES-WITH-SAME-LEX.

3.4 Compiling the Dictionary

The complete master dictionary is not used in its original form during actual generation. Instead, a compiled version is prepared and used. The function which performs the compilation is

```
****************************************************************
mg-make-m-dictionary (file)
****************************************************************
```

Given a file `dictionary.dict`, it creates two files, `dictionary.data` and `dictionary.index`. This function must be invoked each time the master morphological dictionary is changed.

Warning! Further, when the master dictionary is updated, (mg-quit) should be explicitly invoked, and all morphological elements of the generation system should be reloaded. These precautions are necessary because of a *known bug*: the function MG-START does not call MG-QUIT, so important variables, including *EMD_MASTER_STREAM*, the dictionary stream, may be left over from a previous session.

Chapter 4

Morphological Rules

Morphological rules are used to add endings to, and sometimes also to modify, strings representing word stems. Each rule is created by a call to def-mg-rule, with these arguments:

```
*********************************************************************************
def-mg-rule (rulename  no-of-chars-to-remove  string-to-attach  double-final-char?)
*********************************************************************************
```

(The last argument is obsolete, a survival of the earlier English morphology system.)

For example, the function call

```
(def-mg-rule le 2 "le" nil)
```

makes a rule named LE which can be referenced in the dictionary entries of verbs. (A rule name is a Lisp symbol; upper or lower case is not significant.) The rule adds the present tense ich-ending "e" to verbs like *klingeln*, *laecheln* or *segeln*, and at the same time modifies the stem, giving "klingle", "laechle", or "segle" as output. (If the ending were added without modifying the stem, the result would instead be "klingele", etc.) It receives a stem (e.g. "klingel"); cuts off the final two characters (giving "kling"); and then adds the string "le". All of the German morphological rules are listed below.

Note! In German, different cats (parts of speech) sometimes add the same phonological ending. Both verbs and nouns, for instance, can have EN endings. Of course, these endings have different functions and thus are morphologically distinct. However, from the phonological point of view, no distinction is necessary; and so only a single morphological rule is defined and used in such cases. For easy reference, a rule with multiple uses is listed under every relevant category (but repeated listings are commented out).

```
;;;verb rules

(def-mg-rule e 0 "e" nil)      ;ich trinke
(def-mg-rule le 2 "le" nil)    ;ich klingle, laechle, segle
(def-mg-rule st 0 "st" nil)    ;du trinkst
(def-mg-rule est 0 "est" nil) ;du arbeitest, verwendest
(def-mg-rule t 0 "t" nil)      ;er/sie/ihr trinkt --- also du gruesst, setzt
(def-mg-rule et 0 "et" nil)    ;er/sie/ihr arbeitet, verwendet
(def-mg-rule en 0 "en" nil)    ;wir/Sie/sie trinken
(def-mg-rule n 0 "n" nil)      ;wir/Sie/sie tun, knien, schrein, wandern, klingeln
(def-mg-rule z 0 "" nil)       ;ZERO ending, ich sah
```

```
;;;past tense for weak verbs

(def-mg-rule te 0 "te" nil)        ;ich/er/sie fragte
(def-mg-rule ete 0 "ete" nil)      ;ich/er/sie arbeitete
(def-mg-rule test 0 "test" nil)    ;ich/er/sie fragtest
(def-mg-rule etest 0 "etest" nil) ;ich/er/sie arbeitetest
(def-mg-rule ten 0 "ten" nil)      ;Sie/sie/wir fragten
(def-mg-rule eten 0 "eten" nil)    ;Sie/sie/wir arbeiteten
(def-mg-rule tet 0 "tet" nil)      ;ihr fragtet
(def-mg-rule etet 0 "etet" nil)    ;ihr arbeitetet

;;;noun rules

;;;(def-mg-rule z 0 "" nil)        ;ZERO ending, die Muetter
;;;(def-mg-rule n 0 "n" nil)       ;(des) Herrn
;;;(def-mg-rule en 0 "en" nil)     ;(des) Menschen
(def-mg-rule s 0 "s" nil)          ;(des) Bahnhoffs
(def-mg-rule es 0 "es" nil)        ;(des) Busches
(def-mg-rule ns 0 "ns" nil)        ;(des) Namens
(def-mg-rule ens 0 "ens" nil)      ;(des) Herzens

;;;det rules

;;;(def-mg-rule z 0 "" nil)        ;ZERO ending, ein
(def-mg-rule er 0 "er" nil)        ;einer
(def-mg-rule $$er 2 "rer" nil)     ;unsrer, eurer
;;;(def-mg-rule e 0 "e" nil)       ;eine
;;;(def-mg-rule es 0 "es" nil)     ;eines
;;;(def-mg-rule en 0 "en" nil)     ;einen
(def-mg-rule em 0 "em" nil)        ;einem

;;;adj rules

;;;(def-mg-rule z 0 "" nil)        ;ZERO ending, gut
;;;(def-mg-rule er 0 "er" nil)     ;guter
;;;(def-mg-rule e 0 "e" nil)       ;gute
;;;(def-mg-rule es 0 "es" nil)     ;gutes
;;;(def-mg-rule en 0 "en" nil)     ;guten
;;;(def-mg-rule em 0 "em" nil)     ;gutem
```

Chapter 5

Postprocessing

When all of the words (leaves) of the syntactic tree have been processed, the result is a Lisp list of strings, e.g. (" Harald" " weint" "." "," " aber" " Helmut" " lacht" "."). The string includes punctuation marks, since these are included as terminal elements in the syntactic tree. *Postprocessing* then creates a single string which will become the final result of German generation: "Harald weint, aber Helmut lacht."

The strings in the list cannot simply be concatenated, however, since punctuation and capitalization can create special problems. For example, the punctuation of an embedding construction can override the default punctuation of an embedded construction. (The medial punctuation for a compound sentence, for instance, normally overrides the default final punctuation of the embedded clauses.) Further, since the German grammar can deliver multiple sentences, capitalization after a full stop sometimes becomes necessary.

Since the earlier postprocessing functions for English did not support such punctuation override or multiple sentence output, it became necessary to produce new versions. The new functions are loaded as patches after the normal generation system load is complete.

5.1 Postprocessing: the Specification

We now describe the behavior of the postprocessing routines. In the next section, we will discuss some of the functions which implement this behavior.

Postprocessing is used to make punctuation from embedding and embedded constructions combine properly; to manage capitalization when there are multiple sentences in the output; to suppress default commas, e.g. from relative clauses, immediately before sentence-final punctuation; and to avoid the appearance of multiple commas in constructions (especially addresses) allowing null elements.

5.1.1 Default Punctuation

Punctuation is supplied by syntactic rules associated with illocutionary force types (IFT's). Simple IFT's include INFORM, QUESTIONIF, QUESTIONCONF(irmation), etc., and the syntactic rules which express them provide period or question mark. When these are joined under COMPOUND-IFT (supplied by Japanese-German transfer) the resulting clauses are linked by medial punctuation (,) or separated by final punctuation (. ? !) coming from the compound construction..

When explicit punctuation is supplied by a syntactic construction expressing a compound IFT, the default punctuation from the lower IFT is normally deleted. Thus any punctuation from a compound IFT dominates. (Or, if no compound IFT punctuation is explicit, a lone simple IFT punctuation is deleted. In effect, a null

punctuation mark overrides the default punctuation.) Alternatively, by explicit request, simple (default) IFT punctuation can be allowed to remain in place.

We now give examples using output string lists.

Later punctuation normally overrides earlier in any position:

```
(" a" "." "," " b" ".") --> ("A" "," " b" ".")
(" John" " laughed" "." "," " and" " Mary" " cried" ".")
    ---> ("John" " laughed" "," "and" "Mary" "cried" ".")

(" a" "." "." " b" ".") --> ("a" "." "b" ".")
(" John" " laughed" "." "." " and" " Mary" " cried" ".")
    ---> ("John" " laughed" "." " And" " Mary" " cried" ".")
```

However, earlier punctuation overrides later iff the later punctuation is the reserved symbol ^. This convention permits the default punctuation to remain in place if desired.

```
(" a" "?" "^" " b" ".") --> ("A" "?" " B" ".")
(" did" "John" " laugh" "?" "^" " and" " did" " Mary" " cry" "?")
     --> (" Did" "John" " laugh" "?" " And" " did" " Mary" " cry" "?")
```

Lone final punctuation is deleted except at the very end of the list. In effect, null punctuation from the compound IFT overrides the default punctuation from the simple IFT.

```
(" a" "." " b" ".") --> ("A" " b" ".")
(" John" " laughed" "." " and" " Mary" " cried" ".")
    ---> (" John" " laughed" " and" " Mary" " cried" ".")
```

5.1.2 Capitalization

A word following a surviving final punctuation is capitalized.

```
(" John" " laughed" "." "." " and" " Mary" " cried" ".")
    ---> ("John" " laughed" "." " And" " Mary" " cried" ".")
```

5.1.3 Penultimate (Just-Before-Final) Comma

We delete a comma, e.g. the default comma from a relative clause, in penultimate (just-before-final) position. Otherwise, lone commas are not disturbed.

```
(" this" " is" " John" "," " who" " lives" " in" " California" "," ".")
  -->  ("This" " is" " John" "," " who" " lives" " in" " California" ".")
```

5.1.4 Comma Before Comma

Certain syntactic constructions, especially for addresses, permit null elements. The commas surrounding the null elements remain until post-processing, so that several consecutive commas may appear in the output string. For such multi-comma sequences, only the last comma is retained.

```
("mein" "Adresse" "sein" "<trace>"  "," "," "dreiundzwanzig" ","
          "Tschaja-matschi" "," "Kitta-ku" "," "O:saka" "<trace>" "." )

 -->  ("mein" "Adresse" "sein" "<trace>"  "," "dreiundzwanzig" ","
          "Tschaja-matschi" "," "Kitta-ku" "," "O:saka" "<trace>" "." )
```

Note! The above example shows a minor problem in the current syntax/morphology interface. Actually, *both* commas should be deleted after "Mein Adresse ist ...": the single remaining comma is a minor punctuation error. On the other hand, at least one comma should survive between later address elements. Postprocessing could not distinguish these contexts, however, unless structural clues were given. For now, we keep the extra bad comma, recognizing that later revision of the syntactic rules is desirable.

5.2 Postprocessing Functions

We now give some implementation details concerning postprocessing. Commented functions for enhanced postprocessing are in the file `mgen-main-patch.lisp`.

Two system functions are overwritten: MGEN_MAIN and PUNCTUATION-MARKS-P.

The new versions:

```
*****************************************************************
mgen_main (f_node &aux morphemes last_elem)
*****************************************************************
```

Postprocess a list *morphemes* of morphological output strings:

- tag some punctuation marks (using @) for later deletion;
- delete tagged punctuation marks;
- capitalize after surviving final punctuation;
- capitalize the first word of the list;
- concatenate the surviving strings into a single output string.

```
(defun mgen_main  (f_node &aux morphemes last_elem)
  (setq morphemes (mgen_make-morpheme_list f_node))

  ;;Note DESTRUCTIVE operations
  (mark-puncts-for-deletion morphemes)
  (delete-marked-puncts morphemes)
  (capitalize-after-final-punct morphemes)

  (when (consp morphemes)
    ;;capitalize first word
    (setf (car morphemes)
          (mgen_string_upcase
           (string-left-trim '(#\space)(car morphemes))))

    ;;concat list of strings into single string
    (format nil "~{~a~}~a"
            (nreverse (cdr (reverse morphemes)))
            (car (last morphemes)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
punctuation-marks-p (string)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

@ is used to tag some punctuation for later deletion. @ should be recognized as a punctuation mark.

```
(defun punctuation-marks-p (string)
  (and (= 1 (length string))
       (member (char string 0)
               '(#\. #\? #\, #\, #\: #\; #\! #\@))))
```

As seen above, the important new functions called by the new version of MGEN_MAIN are:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
mark-puncts-for-deletion (morpheme-list)
delete-marked-puncts (morpheme-list)
capitalize-after-final-punct (morpheme-list)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Several auxiliary predicates are also defined: for instance, the following function recognizes the final punctuation marks ., ?, and !.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
final-punctuation-marks-p (string)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Appendix A

References

Kikui, Gen-ichiro. 1993. *Morphological Synthesis: Reference Manual.* ATR Technical Report, TR-I-0361.