

Malaga 7.12

User's and Programmer's Manual

Björn Beutel

Copyright © 1995 Björn Beutel.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Table of Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 2 | Malaga’s Grammar Formalism | 2 |
| 3 | The Malaga Programs | 4 |
| 3.1 | Projects | 4 |
| 3.2 | The Malaga Profiles ‘.malagarc’ or ‘malaga.ini’ | 6 |
| 3.3 | The Program malaga | 8 |
| 3.4 | The Program mallex | 9 |
| 3.5 | The Program malmake | 10 |
| 3.6 | The Program malrul | 10 |
| 3.7 | The Program malsym | 11 |
| 4 | The Commands of malaga and mallex | 12 |
| 4.1 | The Command backtrace | 12 |
| 4.2 | The Command break | 12 |
| 4.3 | The Command clear-cache (malaga) | 13 |
| 4.4 | The Command continue | 13 |
| 4.5 | The Command debug-ga (mallex) | 14 |
| 4.6 | The Command debug-ga-file (mallex) | 14 |
| 4.7 | The Command debug-ga-line (mallex) | 14 |
| 4.8 | The Command debug-ma (malaga) | 15 |
| 4.9 | The Command debug-ma-line (malaga) | 15 |
| 4.10 | The Command debug-sa (malaga) | 15 |
| 4.11 | The Command debug-sa-line (malaga) | 15 |
| 4.12 | The Command debug-state (malaga) | 15 |
| 4.13 | The Command delete | 16 |
| 4.14 | The Command down | 16 |
| 4.15 | The Command finish | 16 |
| 4.16 | The Command frame | 16 |
| 4.17 | The Command ga (mallex) | 16 |
| 4.18 | The Command ga-file (mallex) | 16 |
| 4.19 | The Command ga-line (mallex) | 17 |
| 4.20 | The Command get | 17 |
| 4.21 | The Command help | 17 |
| 4.22 | The Command info (malaga) | 17 |
| 4.23 | The Command list | 17 |
| 4.24 | The Command ma (malaga) | 18 |
| 4.25 | The Command ma-file (malaga) | 18 |
| 4.26 | The Command ma-line (malaga) | 19 |
| 4.27 | The Command mg (malaga) | 19 |
| 4.28 | The Command next | 19 |

| | | |
|----------|---|-----------|
| 4.29 | The Command <code>print</code> | 19 |
| 4.30 | The Command <code>quit</code> | 20 |
| 4.31 | The Command <code>read-constants</code> (mallex)..... | 20 |
| 4.32 | The Command <code>result</code> | 21 |
| 4.33 | The Command <code>run</code> | 21 |
| 4.34 | The Command <code>sa</code> (malaga)..... | 21 |
| 4.35 | The Command <code>sa-file</code> (malaga)..... | 22 |
| 4.36 | The Command <code>sa-line</code> (malaga)..... | 22 |
| 4.37 | The Command <code>set</code> | 23 |
| 4.38 | The Command <code>sg</code> (malaga)..... | 23 |
| 4.39 | The Command <code>step</code> | 23 |
| 4.40 | The Command <code>transmit</code> | 23 |
| 4.41 | The Command <code>tree</code> (malaga)..... | 23 |
| 4.42 | The Command <code>up</code> | 25 |
| 4.43 | The Command <code>variables</code> | 25 |
| 4.44 | The Command <code>walk</code> | 26 |
| 4.45 | The Command <code>where</code> | 26 |
| 5 | The Options of malaga and mallex..... | 28 |
| 5.1 | The Option <code>alias</code> | 28 |
| 5.2 | The Option <code>allo-format</code> (mallex)..... | 28 |
| 5.3 | The Option <code>auto-tree</code> (malaga)..... | 28 |
| 5.4 | The Option <code>auto-variables</code> | 28 |
| 5.5 | The Option <code>cache-size</code> (malaga)..... | 29 |
| 5.6 | The Option <code>display-cmd</code> | 29 |
| 5.7 | The Option <code>error-format</code> (malaga)..... | 29 |
| 5.8 | The Option <code>hidden</code> | 29 |
| 5.9 | The Option <code>mor-incomplete</code> (malaga)..... | 30 |
| 5.10 | The Option <code>mor-out-filter</code> (malaga)..... | 30 |
| 5.11 | The Option <code>mor-pruning</code> (malaga)..... | 30 |
| 5.12 | The Option <code>result-format</code> (malaga)..... | 30 |
| 5.13 | The Option <code>result-list</code> (malaga)..... | 31 |
| 5.14 | The Option <code>robust-rule</code> (malaga)..... | 31 |
| 5.15 | The Option <code>roman-hangul</code> | 31 |
| 5.16 | The Option <code>sort-records</code> | 31 |
| 5.17 | The Option <code>switch</code> | 32 |
| 5.18 | The Option <code>syn-incomplete</code> (malaga)..... | 32 |
| 5.19 | The Option <code>syn-in-filter</code> (malaga)..... | 32 |
| 5.20 | The Option <code>syn-out-filter</code> (malaga)..... | 32 |
| 5.21 | The Option <code>syn-pruning</code> (malaga)..... | 32 |
| 5.22 | The Option <code>transmit-cmd</code> | 33 |
| 5.23 | The Option <code>unknown-format</code> (malaga)..... | 33 |
| 5.24 | The Option <code>use-display</code> | 33 |

| | | |
|----------|--|-----------|
| 6 | The Programming Language Malaga | 34 |
| 6.1 | Characterisation of Malaga | 34 |
| 6.2 | Malaga Source Texts | 34 |
| 6.2.1 | Comments | 35 |
| 6.2.2 | The <code>include</code> Statement | 35 |
| 6.2.3 | Identifiers | 35 |
| 6.3 | Values | 35 |
| 6.3.1 | Symbols | 35 |
| 6.3.2 | Numbers | 35 |
| 6.3.3 | Strings | 36 |
| 6.3.4 | Lists | 36 |
| 6.3.5 | Records | 36 |
| 6.4 | Expressions | 36 |
| 6.4.1 | Variables | 37 |
| 6.4.2 | Constants | 37 |
| 6.4.3 | Subrule Invocations | 37 |
| 6.4.4 | The Function <code>atoms</code> | 37 |
| 6.4.5 | The Function <code>capital</code> | 37 |
| 6.4.6 | The Function <code>floor</code> | 37 |
| 6.4.7 | The Function <code>length</code> | 37 |
| 6.4.8 | The Function <code>multi</code> | 37 |
| 6.4.9 | The Function <code>set</code> | 38 |
| 6.4.10 | The Function <code>substring</code> | 38 |
| 6.4.11 | The Function <code>switch</code> | 38 |
| 6.4.12 | The Function <code>transmit</code> | 38 |
| 6.4.13 | The Function <code>value_string</code> | 38 |
| 6.4.14 | The Function <code>value_type</code> | 38 |
| 6.4.15 | The <code>if</code> Expression | 38 |
| 6.4.16 | Unary <code>'-'</code> | 39 |
| 6.4.17 | The Operator <code>'.'</code> | 39 |
| 6.4.18 | The Operator <code>'+'</code> | 39 |
| 6.4.19 | The Operator <code>'-'</code> | 39 |
| 6.4.20 | The Operator <code>'*'</code> | 40 |
| 6.4.21 | The Operator <code>'/'</code> | 40 |
| 6.5 | Conditions | 41 |
| 6.5.1 | The Operators <code>'='</code> and <code>'/='</code> | 41 |
| 6.5.2 | The Operators <code>less</code> , <code>less_equal</code> , <code>greater</code> , <code>greater_equal</code> | 41 |
| 6.5.3 | The Operators <code>'~'</code> and <code>'/~'</code> | 42 |
| 6.5.4 | The Operator <code>in</code> | 42 |
| 6.5.5 | The <code>matches</code> Condition (Regular Expressions) | 42 |
| 6.6 | The Operators <code>not</code> , <code>and</code> , and <code>or</code> | 43 |
| 6.7 | The Symbol Table | 44 |
| 6.8 | The Initial State | 44 |
| 6.9 | The Constant Definition | 44 |
| 6.10 | Rules | 45 |
| 6.11 | Statements | 46 |
| 6.11.1 | The <code>assert</code> Statement | 47 |

| | | |
|--------------------|-------------------------------------|-----------|
| 6.11.2 | The Assignment | 47 |
| 6.11.3 | The break Statement | 47 |
| 6.11.4 | The choose Statement | 47 |
| 6.11.5 | The continue Statement | 48 |
| 6.11.6 | The define Statement | 48 |
| 6.11.7 | The error Statement | 48 |
| 6.11.8 | The foreach Statement | 48 |
| 6.11.9 | The if Statement | 49 |
| 6.11.10 | The repeat Statement | 49 |
| 6.11.11 | The require Statement | 50 |
| 6.11.12 | The result Statement | 50 |
| 6.11.13 | The return Statement | 51 |
| 6.11.14 | The select Statement | 51 |
| 6.11.15 | The stop Statement | 52 |
| 6.12 | Files | 52 |
| 6.12.1 | The Symbol File | 52 |
| 6.12.2 | The Extended Symbol File | 52 |
| 6.12.3 | The Lexicon File | 52 |
| 6.12.4 | The Allomorph Rule File | 52 |
| 6.12.5 | The Combi-Rule Files | 52 |
| 6.13 | Summary of the Malaga Syntax | 53 |
| Index | | 57 |

1 Introduction

The Name “Malaga” is used with two different meanings: on the one hand, it is the name of a special purpose programming language, namely a language to implement grammars for natural languages. On the other hand, it is the name of a program package for development of Malaga Grammars and testing them by analysing words and sentences. “Malaga” is an acronym for “**M**erely **a** **L**eft-**A**ssociative **G**rammar **A**pplication”.

The program package “Malaga” has been developed by Björn Beutel. Gerald Schüller has implemented parts of the debugger, parts of the Emacs Malaga mode, and the original Tree and Variable output via TCL/Tk.

So far, morphology grammars for several natural languages have been developed with Malaga, including the Albanian, Bulgarian, English, Finnish, German, Italian, Korean and Spanish language.

2 Malaga's Grammar Formalism

A formal grammar for a natural language can be used to check whether a sentence or a word form is grammatically well-formed (a word form is a special inflectional form of a word, so "book" and "books" are two different word forms of the word "book"). Furthermore, a grammar can describe the structure and meaning of a sentence or a word form by a data structure that has been constructed during the analysis process.

Malaga is using a formalism that is derived of the Left-Associative Grammar (LAG), which has been developed by Roland Hausser. An LAG analyses a sentence (or a word form) step by step: its parts are concatenated from the left to the right, hence the name "Left-Associative Grammar". A single LAG rule can only join two parts to a bigger one: it concatenates the state part (which is the beginning of the sentence or word form that has already been analysed) and the link part (which is the next word form or the next allomorph). In contrast to LAG, Malaga's formalism already reads in the first part of a word form or of a sentence by applying a rule. Take a look at the following sentence:

Shakespeare liked writing comedies.

The sentence is being analysed by five rule applications:

" " + "Shakespeare"
 "Shakespeare" + "liked"
 "Shakespeare liked" + "writing"
 "Shakespeare liked writing" + "comedies"
 "Shakespeare liked writing comedies" + "."

To apply a rule it's not sufficient to know the spelling of a word or an allomorph. A rule also requires morphological and syntactic information, such as word class, gender, meaning of a suffix etc. This information, which is associated with an element of an utterance, like a sentence, a word form or an allomorph, is called its *feature structure*. The analysis of a sentence or a word form returns such a feature structure as result.

Now let us take a closer look at how a sentence is analysed.

1. Before we can start to analyse a sentence, the analysis automaton must be in an *initial state*. The initial state includes:
 - a feature structure for that state, and
 - the *combination rule* checking whether it is allowed to start with a specific word form. This rule also builds the feature structure of the successor state (whose surface consists of the first word form).
2. The next word form that is going to be added is read and analysed morphologically. If there is no valid word form, the analysis process aborts.
3. The feature structure that morphology assigns to this word form is called the link's feature structure. The feature structure of the input that has been analysed syntactically so far is called the state's feature structure.
4. The active combination rule checks whether it is allowed to combine the state's surface (which may be empty if the rule is operating on the initial state) with the link, i.e., the next word form. The combination rule takes the feature structures of the state and of the link as parameters. They can be compared by logical tests, and finally the feature

structure of the successor state (whose surface includes the word form that has been read), is constructed by the rule. The rule also specifies which *successor rule* is active in the successor state. Execution then continues at step 2.

Instead of specifying a successor rule, a rule can also *accept* the analysed sentence. In that case, the feature structure of the successor state will be used as the feature structure of the complete analysed sentence.

Morphological analysis operates analogously, except that a word form, composed from allomorphs, is being analysed. The link (step 2) is found in the allomorph lexicon.

This sketch is of course simplified. There can be ambiguities in an analysis, induced by several causes:

- The initial state may contain several rules to analyse the first word form or allomorph.
- A rule may have multiple successor rules.
- In morphology, the continuation of the input may match several trie entries.
- In syntax analysis, the link may be assigned several feature structures by morphology.

These ambiguities are coped with by dividing the analysis into several subanalyses: if there are two lexicon entries for a word form, for example, the analysis continues using the first entry (and its feature structure) as well as the second one. You can compare this with a branching path. The analyses will be continued independently of each other. So, one analysis path can accept the input while the other fails. Each analysis path can divide repeatedly when other ambiguities are met. If several analysis paths are continued until they accept, the analysis process returns more than one result.

3 The Malaga Programs

The Malaga programs are all started in a similar manner: either you give the name of a *project file* as argument (this is not possible if you start `malrul` or `malsym`), or you give the name of the files that are needed by the program (for `malmake` and `malaga`, you have to give the project file as argument). The file type is recognised by the file name ending.

Assume you've written a grammar that consists of a symbol file '`english.sym`', an allomorph rule file '`english.all`', a lexicon file '`english.lex`' and a morphology rule file '`english.mor`', and you have also written a project file '`english.pro`'. You first have to create binary files from these files:

```
malmake english.pro
```

The source files must be in the Unicode UTF-8 format, which is also used for input and output by the Malaga programs.

The binary files have the same name as their source counterparts, but have a '`_l`' (for little endian processors like x86), a '`_b`' (for big endian processors like HPPA) or a '`_c`' (for other architectures) appended. Now you can start the program `malaga` by entering the following command line: `malaga english.pro`.

The names of the grammar files will be read from the project file.

If you want to know about the command line arguments of a Malaga program, you can get help by using the option '`-help`' or '`-h`', like `mallelex -help`

If you just want to know which version of a Malaga program you are using, you can get the version number by using the option '`-version`' or '`-v`', like `malrul -version`

The program just emits a few lines with information about its version number and about using and copying it.

3.1 Projects

A couple of files, taken together, form a Malaga grammar:

The *lexicon file* ('`.lex`')

A lexicon of base forms.

The *prelex file* ('`.prelex`', optional)

A precompiled lexicon in binary format.

The *allomorph rule file* ('`.all`')

A file with rules which generate the allomorphs of the base forms.

The *morphology rule file* ('`.mor`')

A file with rules which combine allomorphs to word forms.

The *symbol file* ('`.sym`')

A file with the symbols that may be used in rules and feature structures.

The *syntax rule file* ('`.syn`', optional)

A file with rules that combine word forms to sentences.

The *extended symbol file* ('`.esym`', optional)

A file with additional symbols that may only be used in a syntax rule file.

You can group these files together to a *project*. To do this, you have to write a project file, with a name ending in `.pro`, in which you list the names of the several files, each one behind a keyword (each file type in a line on its own). Imagine you have written a grammar that consists of the files `standard.sym`, `webster.lex`, `english.all`, `english.mor`, and `english.syn`. The project file for this grammar will look like this:

```
sym: standard.sym
lex: webster.lex
all: english.all
mor: english.mor
syn: english.syn
```

In your source files, you can include further source files by using the `include` statement; so a binary file of your grammar may be dependent on several source files. The program `malmake` uses the information in the project file to check for dependencies between source files and binaries, so the project file must contain the name of all source files for a specific binary. Relative path names are always relative to the directory of the project file.

Assume, you've got a lexicon file `webster.lex` that looks like this:

```
include "suffixes.lex";
include "verbs.lex";
include "adjectives.lex";
include "nouns.lex";
include "particles.lex";
include "abbreviations.lex";
include "names.lex";
include "numbers.lex";
```

In this case, you must write the names of all these files in the `lex:` line of your project file behind the name of the real lexicon file:

```
lex: webster.lex suffixes.lex verbs.lex adjectives.lex
lex: nouns.lex particles.lex abbreviations.lex names.lex numbers.lex
```

Since there is a number of files in this example, the `lex:` line has been divided into two lines, each line starting with `lex:`.

If you want to extend an existing project (for example, you might want to add a syntax rule file to a morphology grammar), you can include the project file of the morphology grammar in the project file of your syntax grammar by using a line starting with `include:`:

```
include: /projects/grammars/english/english.pro
syn: english-syntax.syn
```

The file entries in the project file of the morphology are treated as if they would replace the `include:` line. Relative paths in the included file are relative to the *included* directory, not the *including* directory.

The programs `malaga` and `mallelex` can set options like `hidden` or `robust` from the project file, so you do not need to set these options each time you start `malaga`. Each line in the project file that starts with `malaga:` and `mallelex:`, respectively, will be executed when `malaga` and `mallelex`, respectively, has been started, but you may only use the `set` command, so you can only set options in the project file. Here is an example:

```
...
```

```
malaga: set hidden +semantics
malaga: set robust-rule on
mallex: set hidden +semantics +syntax
...
```

When you start `malaga`, the commands `set hidden +semantics` and `set robust-rule on` will be executed; when you start `mallex`, the command `set hidden +semantics +syntax` will be executed.

Options in project files that are read in by ‘`include:`’ lines in other project files will be executed as if they were in place of the ‘`include:`’ line.

Lines in project files that start with ‘`info:`’ contain information about the grammar. In `malaga`, you get this information if you use the command `info`. Example:

```
info: =====
info: Deutsche Malaga Morphologie 3.0
info: written by Oliver Lorenz, 11.04.1997
info: =====
```

The Korean writing system, Hangul, needs special treatment, because the characters it uses are syllables that must be split up into individual letters for morphological analysis. Such a conversion is built-in into `malaga`. To activate this Hangul support, insert the following line into your project file:

```
split-hangul-syllables: yes
```

If Hangul support has been switch on, you may also enter Hangul text in a Latin transcription that is based on the Yale transcription. Transcribed text must be contained in curly brackets, and each syllable must start with a dot, for example ‘`{.cwess.ta}`’. `Malaga` can also display Hangul text in Latin transcription if Hangul support has been activated. This can be controlled by the option `roman-hangul`.

When `Malaga` splits Hangul syllables, you must be aware that string operations work with Hangul letters, even if you have entered syllables in you grammar source code:

- In a pattern, a character class that contains a syllable will match each single Hangul letter that is part of that syllable. Only use single letters in character classes. If you want to select between different syllables, use alternatives, separated by vertical bars.
- In a pattern, when a postfix operator (‘`*`’, ‘`?`’, or ‘`+`’) follows a Hangul syllable, it will only operate on the last letter of that syllable. If you want the operator to work on the whole syllable, put the syllable in parentheses.
- The functions `substring` and `length` will count single characters, not syllables.

3.2 The Malaga Profiles ‘`.malagarc`’ or ‘`malaga.ini`’

If you prefer some options that you want to use with every `Malaga` project, you may create a personal profile. On POSIX systems, it is located in your home directory and is called ‘`.malagarc`’. In Microsoft Windows (NT based) systems, it is located in your user profile directory and is called ‘`malaga.ini`’. In Microsoft Windows (DOS based) systems, it is located in the root directory of your system drive and is also called ‘`malaga.ini`’. You can enter `malaga` and `mallex` options in the same manner as you do in the project file:

```
malaga: set display-cmd "malshow"
malaga: set use-display yes
```

```
mallex: set display-cmd "malshow"
mallex: set use-display yes
```

The settings in your personal profile override the settings in the project file.

You can set some attributes of the graphical user interface `malshow`, like the position, the size, and the font size of each window that is opened by `malshow`. The following attributes are available:

***_geometry:**

Defines the size and/or position of a window. The “*” must be replaced by the name of the window, which may be `allomorphs`, `path`, `result`, `tree`, `variables`, or `expressions`. The attribute must be followed by an expression like ‘628x480+640+512’. The first two numbers (‘628x480’) define the width and the height of the window in pixels, the last two numbers (‘+640+512’) define the position of its upper left corner.

font: The attribute must be followed by the name of the font family to use.

font_size:

The attribute must be followed by an integer font size, given in points. The available font sizes are 8, 10, 12, 14, 18, and 24 points.

show_indexes:

The attribute must be followed by `yes` or `no`, which determines whether state indexes are shown in the Tree and Path windows.

hanging_style:

The attribute must be followed by `yes` or `no`, which determines whether horizontally adjacent complex values are aligned at their top lines (*hanging style*) or at their bottom lines (*non-hanging style*).

inline_path:

The attribute must be followed by `yes` or `no`, which determines whether the components of a state or a link in a path will be arranged horizontally or vertically. For small feature-structures, e.g. in formal grammars, horizontal arrangement is better readable, while full-blown natural language grammar paths look better in vertical arrangement.

show_tree:

A three-valued attribute that determines which states of a tree are shown. Possible values are: `full`, `no_dead_ends` and `result_paths`.

Here is an example which sets every option available:

```
allomorphs_geometry: 628x480+640+0
path_geometry: 628x480+640+0
result_geometry: 628x480+640+0
tree_geometry: 628x480+640+512
variables_geometry: 628x480+640+512
expressions_geometry: 628x480+640+0

font: helvetica
font_size: 12
```

```

show_indexes: yes
hanging_style: yes
inline_path: yes
show_tree: no_dead_ends

```

3.3 The Program malaga

The program `malaga` is the user interface for analysing word forms and sentences, displaying the results and finding bugs in a grammar. Start `malaga` with the name of a project file as argument:

```
malaga english.pro
```

When `malaga` has been started, it loads the symbol file, the lexicon file and the morphology rule file, and the syntax rule file, if there is one. After loading, the *prompt* appears. Then `malaga` is ready to execute your commands:

```

$ malaga english.pro
This is malaga, version 7.12.
Copyright (C) 1995 Bjoern Beutel.
This program is part of Malaga, a system for Natural Language Analysis.
You can distribute it under the terms of the GNU General Public License.
malaga>

```

You can now enter any `malaga` command. If you are not sure about the name of a command, use the command `help` to get an overview of all `malaga` commands.

If you want to quit `malaga`, enter the command `quit`.

You can use the following command line options when you start `malaga`:

`'-morphology'` or `'-m'`

Starts `malaga` in *morphology mode*. That is, word forms are being read in from the standard input stream and analysed (one word form per line). The analysis result is being written to the standard output stream.

`'-syntax'` or `'-s'`

Starts `malaga` in *syntax mode*. That is, sentences are being read in from the standard input stream and analysed (one sentence per line). The analysis result is being written to the standard output stream.

`'-quoted'` or `'-q'`

When `malaga` has been started in syntax or morphology mode, and the option `'-quoted'` has been used, then each input line must be enclosed in double quotes which are removed prior to analysis. Within the double quotes there may be any combination of printable characters except the backslash `'\'` and the double quotes. These characters must be preceded by a `'\'` (escape character).

`'-input'` or `'-i'`

Starts `malaga` in *argument analysis mode*. That is, the argument following the `'-input'` is being analysed. Either the `'-morphology'` or the `'-syntax'` option

must also be given. The analysis result is being sent to the standard output stream in a structured format.

3.4 The Program `malle`

By using `malle`, you can make the allomorph rules process the entries of a base form lexicon.

You can start `malle` either with the name of a project file or with the names of the needed grammar files:

```
malle english.pro
```

or

```
malle english.sym english.all english.lex
```

If you are not using a project file, you must give

- the name of the symbol file (`.sym`),
- the name of the allomorph rule file (`.all`),
- the name of the lexicon file (`.lex`, in batch mode), and
- the name of the prelex file (`.prelex`, in batch mode, optional).

Normally, `malle` runs interactively: it loads the symbol file and the allomorph rule file. Then the *prompt* appears:

```
$ malle english.pro
This is malle, version 7.12.
Copyright (C) 1995 Bjoern Beutel.
This program is part of Malaga, a system for Natural Language Analysis.
You can distribute it under the terms of the GNU General Public License.
malle>
```

You can now enter any `malle` command. If you do not remember the command names, you can use the command `help` to see an overview of the `malle` commands.

If you want to quit `malle`, enter the command `quit`.

If you have started `malle` by using the option `-binary` or `-b`, it creates the run time lexicon file from the base form lexicon file and the optional prelex file. If the lexicons are very big or the allomorph rules are very complex, this can take some time. After creation, `malle` exits.

If you have started `malle` by using the option `-prelex` or `-p`, it creates a precompiled lexicon file from the source lexicon file and the optional prelex file and exits.

You can use the following command line options when you start `malle`:

`-binary` or `-b`

Runs `malle` in batch mode and creates the run-time lexicon.

`-readable` or `-r`

Runs `malle` in batch mode and outputs the allomorph lexicon in readable form on the standard output stream.

`‘-prelex’` or `‘-p’`

Runs `mallex` in batch mode, but doesn’t apply the allomorph filter yet. Outputs the allomorph lexicon as a `‘.prelex’` binary file.

3.5 The Program `malmake`

The program `malmake` reads a project file, checks if all grammar files needed do exist, and translates all grammar files that have not yet been translated or whose source files have changed since they have been translated. `malmake` itself calls the programs `malsym`, `mallex` and `malrul` if needed. An example: assume you have written a morphology grammar whose grammar files are bundled in a project file `‘english.pro’`:

```
sym: rules/english.sym
all: rules/english.all
lex: rules/english.lex lex/adjectives.lex
lex: lex/particles.lex lex/suffixes.lex lex/verbs.lex
lex: lex/nouns.lex lex/abbreviations.lex lex/numbers.lex
mor: rules/english.mor
mallex: set hidden +semantics +syntax
malaga: set hidden +semantics
```

When executing `malmake dmm.pro` for the first time, the symbol file, the rule files and the lexicon file will be translated:

```
$ malmake dmm.pro
compiling "dmm.sym"
compiling "dmm.all"
compiling "dmm.mor"
compiling "dmm.lex"
project is up to date
$
```

If you want all files to be recompiled on all accounts, use the option `‘-new’` or `‘-n’`.

The translation of a big lexicon can take some minutes, since the allomorph rules have to be executed for each lexicon entry.

3.6 The Program `malrul`

The program `malrul` translates Malaga rule files, i.e. files that have the endings `‘.all’`, `‘.mor’` or `‘.syn’`. The compiled file gets the suffix `‘_l’`, `‘_b’`, or `‘_c’`, depending on the endianness of your processor. Give the following arguments if you are starting `malrul`:

- the name of the rule file that is to be translated, and
- the name of the associated symbol file (`‘.sym’` or `‘.esym’`).

The order of the arguments is arbitrary. Here is an example:

```
malrul english.mor english.sym
```

3.7 The Program `malsym`

`malsym` can translate Malaga symbol files, i.e. files having the ending `.sym` or `.esym`. The translated file gets the suffix `_l`, `_b`, or `_c`, depending on the endianness of your processor.

For example:

```
malsym english.sym
```

If you are translating an extended symbol file with the ending `.esym`, enter the name of the compiled symbol file after the command line option `-use` or `-u`:

```
malsym english.esym -use english.sym
```

This argument is needed since extended symbol files are extensions of ordinary symbol files.

If you use the command line option `-split-hangul-syllables` when starting `malsym`, the symbol file and all the Malaga files that use it will split up Hangul syllables in individual letters internally. This option is invoked by `malmake` if the project file contains the line `split-hangul-syllables: yes`.

4 The Commands of `malaga` and `malleX`

Since the user interfaces of `malaga` and `malleX` are very similar and since they have a bunch of commands in common, I will describe them in a common chapter. Commands that can be used in `malaga` or in `malleX` only, are marked by the name of the program in which they can be used.

4.1 The Command `backtrace`

If you are executing your rules in debug mode or the rules were interrupted by an error, this command shows where rule execution currently stopped. If it stopped in a subrule, all calling rules are also shown. The currently examined rule is marked with a `*`:

```
debug> backtrace
*2: "dmm.mor", line 1218, rule "deletePOS"
  1: "dmm.mor", line 31, rule "Start"
debug>
```

This means, rule execution stopped in frame 2, line 1218 of `'dmm.mor'`, in rule `deletePOS`. This subrule was called from frame 1, line 31 in `'dmm.mor'`, in rule `Start`.

4.2 The Command `break`

If you want to stop the rules at a specific point, for example to take a look at the variables, you can use the command `break` to set *breakpoints*. A breakpoint is a point in the rule source text where rule execution is interrupted, so you can enter commands in debug mode. Breakpoints are only active in debug mode, this means you have started rule execution by a debug command or you have continued rule execution by one of the commands `step`, `next`, `walk`, or `continue`.

Behind the command name, `break`, you can give one of the following arguments:

A line number.

A breakpoint is set at this line in the current source file. If there is no statement starting at this line, the breakpoint will be set at the nearest line where a statement starts. You can, for example, set a breakpoint at line 245 in the current source file by entering the command

```
break 245
```

A file name and a line number.

A breakpoint is set at this line in this file. If there is no statement starting at this line, the breakpoint will be set at the nearest line where a statement starts.

An example:

```
break english.syn 59
```

A rule name.

A breakpoint is set at the first statement in this rule. An example:

```
break final_rule
```

If the rule name or the file name is ambiguous, you can insert an abbreviation for the rule system you refer to. Put it in front of the rule name or the file name. The following abbreviations are used:

- ‘all’ for allomorph rules,
- ‘mor’ for morphology rules,
- ‘syn’ for syntax rules,

If you omit any argument, the breakpoint is set on the current line in the current file (this is helpful in debug mode).

Every breakpoint gets a unique number once it has been set, so you can delete it later, when you do not need it any longer.

You can list the breakpoints using the command `list` and delete them using `delete`.

4.3 The Command `clear-cache` (`malaga`)

If you have changed your settings so that the wordform cache is no longer valid, you can clear the cache using `clear-cache`. This can be necessary if you have turned on/off input or output filters or modified switches.

4.4 The Command `continue`

This command can only be executed in debug mode. It resumes rule execution and may be followed by:

Nothing. Rule execution is continued until a breakpoint is met or the rules have been executed completely.

A line number.

Rule execution is continued until a breakpoint is met, the rules have been executed completely or the given line in the current source file is met. If there is no statement starting at this line, execution will be stopped at the nearest line where a statement starts. You can, for example, continue execution until line 245 in the current source file is met by entering the command

```
continue 245
```

A file name and a line number.

Rule execution is continued until a breakpoint is met, the rules have been executed completely or the given line in the given file is met. If there is no statement starting at this line, execution will be stopped at the nearest line where a statement starts. An example:

```
continue english.syn 59
```

A rule name.

Rule execution is continued until a breakpoint is met, the rules have been executed completely or the first statement of the given rule is met. An example:

```
continue final_rule
```

A comparison.

The comparison must be of the form `variable = value`, where *variable* may be any variable name, maybe followed by a path, and *value* may be any Malaga

value. Rule execution is continued until a breakpoint is met, the rules have been executed completely or until *variable* is defined and its value is *value*.

4.5 The Command `debug-ga` (`mallelex`)

Use `debug-ga` to find errors in your allomorph rules. This command works like `ga`, but the allomorph generation will be stopped before the first statement of the first rule is executed:

```
mallelex> debug-ga [surface: "john", class: name]
at rule "irregular_verb"
debug>
```

The prompt ‘`debug>`’ that appears instead of ‘`mallelex>`’ indicates that `mallelex` is currently executing the allomorph rules but has been interrupted. Since this ability has been developed to support the *debugging* of Malaga rules, this mode is called *debug mode*.

When `mallelex` arrives at the start of a new rule in debug mode (as in the example above), the name of this rule is displayed. When in debug mode, you can always get the name of the current rule using the command `rule`.

If you’re running `mallelex` from Emacs, another Emacs window will display the source file. An arrow is used to show to the statement that will be executed next.

```
...
allo_rule irregular_verb ($entry):
=>? $entry.class = verb;
...
```

In debug mode, you can, for example, get the variables that are currently defined (using `variable` or `print`), and you can execute statements (using `step`, `next`, `walk`, `continue`, or `run`). If you want to quit the debug mode, just enter `run`. The remaining statements for generation will then be executed without interruption.

4.6 The Command `debug-ga-file` (`mallelex`)

Use the command `debug-ga-file` to make the allomorph rules work on a lexicon file in debug mode. Assume you have written a lexicon file ‘`mini.lex`’:

```
[surface: "m{a}n", class: noun];
[surface: "table", class: noun];
[surface: "wise", class: adjective];
```

To let the rules process this lexicon in debug mode, enter:

```
debug-ga-file mini.lex
```

4.7 The Command `debug-ga-line` (`mallelex`)

Use the command `debug-ga-line` to make the allomorph rules generate allomorphs for a single lexicon entry in debug mode. Assume you want to test the second line in the lexicon file ‘`mini.lex`’:

```
[surface: "m{a}n", class: noun];
[surface: "table", class: noun];
[surface: "wise", class: adjective];
```

Enter the following line:

```
debug-ga-line mini.lex 2
```

Then `mallelex` stops in debug mode at the entry of the first allomorph rule that is being executed for the lexicon entry

```
[surface: "table", class:noun];
```

If there is no lexicon entry at this line, the subsequent lexicon entry will be taken.

4.8 The Command `debug-ma` (malaga)

Use the command `debug-ma` to find errors in your morphology combination rules. This command analyses the rest of the command line morphologically and executes the morphology combination rules in debug mode. Debug mode is explained for the command `debug-ga`.

4.9 The Command `debug-ma-line` (malaga)

Use the command `debug-ma-line` to find errors in your morphology combination rules. This command analyses the rest of the command line morphologically and executes the morphology combination rules in debug mode. Debug mode is explained for the command `debug-ga`.

4.10 The Command `debug-sa` (malaga)

Use the command `debug-sa` to find errors in your syntax combination rules. This command analyses the rest of the command line syntactically and executes the syntax combination rules in debug mode. Debug mode is explained for the command `debug-ga`.

4.11 The Command `debug-sa-line` (malaga)

Use the command `debug-sa-line` to find errors in your syntax combination rules. This command analyses the rest of the command line morphologically and executes the morphology combination rules in debug mode. Debug mode is explained for the command `debug-ga`.

4.12 The Command `debug-state` (malaga)

Use the command `debug-state` to execute the successor rules of a specific LAG state in debug mode. Previously, you must have already analysed a word or a sentence, respectively. Let `malaga` display the analysis tree by entering `tree`, move the mouse pointer over the state you want to debug, and press the left mouse button. A window opens in which this state's feature structure is shown. The window's title line contains the index of the state. Use this number as argument for `debug-state`. The last analysis input will be analysed again, and analysis stops when reaching the first successor rule of the specified state and `malaga` switches to debug mode. Debug mode is explained for the command `debug-ga`.

4.13 The Command `delete`

If you want to delete a breakpoint, use the command `delete` with the number of the breakpoints as argument.

Enter `'delete all'` to delete all breakpoints.

4.14 The Command `down`

If you want to look at the source and the variables of the (sub)rule that is currently being called by the current subrule, you can do this by entering `down`. You can list the frames via `backtrace`.

4.15 The Command `finish`

This command can only be executed in debug mode. The rule execution will be resumed and continues until a `return` statement is met or until the current rule path will be terminated.

4.16 The Command `frame`

If you want to look at the source and the variables of a (sub)rule that has called the current subrule, directly or indirectly, you can do this by typing `frame` and the number of the frame you want to examine. You can list the frames via `backtrace`.

4.17 The Command `ga` (`mallelex`)

Use the command `ga` (short for *generate allomorphs*) to generate allomorphs. This is useful for testing allomorph generation from within `mallelex`. When you enter the command, give a lexicon entry as argument. All allomorphs that are generated from this entry by the allomorph rules, are displayed on screen. For example:

```
mallelex> ga [Lemma: "!", POS: Punctuation, Type: ExclamationMark]
"!=": [POS: <Punctuation>,
      Punctuation: <[Allomorph: "!",
                    BaseForm: "!",
                    concatStem: no,
                    concatSx: no,
                    POS: Punctuation,
                    Type: ExclamationMark,
                    terminal: yes]>,
      Surface: "!="]
mallelex>
```

If the rules create multiple allomorphs from an entry, they are displayed one after another.

4.18 The Command `ga-file` (`mallelex`)

Use the command `ga-file` to make the allomorph rules generate allomorphs for a lexicon file. Assume you have written a lexicon file `'mini.lex'`:

```
[surface: "m{a}n", class: noun];
```

```
[surface: "table", class: noun];
[surface: "wise", class: adjective];
```

To generate the allomorphs for this lexicon, enter `'ga-file mini.lex'`.

This will produce a readable allomorph file whose name ends in `'.out'`; for `'mini.lex'` its name will be `'mini.lex.out'`:

```
"man": [class: noun, syn: singular]
"men": [class: noun, syn: plural]
"table": [class: noun]
"wise": [class: adjective, restr: complete]
"wis": [class: adjective, restr: inflect]
```

4.19 The Command ga-line (mallex)

Use the command `ga-line` to make the allomorph rules generate allomorphs for a single lexicon entry. Assume you want to test the second line in the lexicon file `'mini.lex'`:

```
[surface: "m{a}n", class: noun];
[surface: "table", class: noun];
[surface: "wise", class: adjective];
```

Enter the following line:

```
ga-line mini.lex 2
```

Then mallex generates allomorphs for `[surface: "table", class:noun];`.

If there is no lexicon entry at this line, the subsequent lexicon entry will be taken.

4.20 The Command get

This command is used to query settings of malaga or mallex. Enter it together with the name of the option whose setting you want to know. The possible options are described in the next chapter. If you just enter `'get'`, all settings will be shown.

4.21 The Command help

Use this command to get a list of the commands you can use. If you give the name of a command or an option as argument, a short explanation of this item will be displayed. If a name represents a command as well as an option, prepend `'command'` or `'option'` to it.

4.22 The Command info (malaga)

This command gives you information about the grammar you are using. It takes no argument.

4.23 The Command list

If you enter the command `list`, all breakpoints are listed. For each breakpoint, its number, the name of the source file and the source line is shown.

4.24 The Command `ma` (`malaga`)

The command `ma` (for *morphological analysis*) starts a word form analysis. Give the word form that you want to be analysed as argument:

```
ma house
```

`Malaga` will show the results automatically, and it will also show the analysis tree automatically if you specified it using the `auto-tree` option. You can look at the results using `result` or at the entire analysis tree using `tree`.

If you do not enter a word form behind the command `ma`, `malaga` re-analyses the last input.

4.25 The Command `ma-file` (`malaga`)

The command `ma-file` can be used to analyse files that contain word lists. A word list consists of a number of word forms, each word form on a line on its own. There may be empty lines in a word list. The following example is a word list called ‘`word-list`’:

```
table
men's
blue
handicap
```

To analyse this word list, enter:

```
ma-file word-list result
```

This will produce a file ‘`result`’ that contains the analysis results. If the second argument is missing, the result will be written to a file whose name ends in ‘`.out`’; for ‘`word-list`’, its name will be ‘`word-list.out`’:

```
1: "table": [class: noun, ...]
2: "men's": [class: noun, ...]
3: "blue": [class: noun, ...]
3: "blue": [class: adjective, ...]
3: "blue": [class: name, ...]
4: "handicap: unknown
```

The number at the line start represents the line number of the analysed original word form. The output format can be changed by using the options `result-format` and `unknown-format`.

If a runtime error occurs during the analysis of a word, the line will be output in the format given by the option `error-format`.

After the analysis, some statistics will be displayed:

- The number of analysed word forms.
- The number of recognised word forms.
- The number of word forms recognised by combi-rules and end-rules.
- The number of word forms recognised by robust-rules.
- The number of word forms whose analyses produced errors.
- The average number of results per word form.
- The analysis run time.
- The average number of word forms that have been analysed per second.

- The number of cache accesses.
- The number of cache hits.

4.26 The Command `ma-line` (`malaga`)

You can use this command to analyse a single line in a text file morphologically. Assume you want to analyse the word in the third line in the file `'words'`. Then enter the following command:

```
ma-line words 3
```

Malaga will show the results automatically, and it will also show the analysis tree automatically if you specified it using the `auto-tree` option. You can look at the results using `result` or at the entire analysis tree using `tree`.

4.27 The Command `mg` (`malaga`)

Use the command `mg` to generate all word forms that consist of a specified set of allomorphs. For example, the command

```
mg 3 un able believe
```

This generates all word forms that consist of up to three allomorphs, where only the specified allomorphs (`'un'`, `'able'`, and `'believe'`) are used. The word forms are numbered from 1 onward, but different analyses of the same word form get the same index. The output will look like this:

```
malaga> mg 3 un able believe
1: "able"
2: "believe"
3: "unable"
4: "unbelievable"
malaga>
```

Please note that generation does not know of filters, pruning rules and default rules.

4.28 The Command `next`

This command can only be executed in debug mode. The rule execution will be resumed and continues until a different source line is met, a different path is going to be executed since the old one has terminated, or until the rules have been executed completely. It is like `step`, but subrules will be executed without interruption. If you specify a number as argument, the command will be repeated as often as specified.

4.29 The Command `print`

The command `print` is used to display the current values of Malaga variables or named constants, or parts of them. You can specify any variable or constant names (including the `'$'` or `'@'`) as arguments to this command; you can also specify a path of attributes and/or indexes (with suffix `'L'` or `'R'`) behind each of the variable or constant names. In that case, only the values of the specified paths are displayed:

```

debug> print $word
$word = [class: pronoun,
        result: S2]
debug> print $word.class
$word.class = pronoun
debug> print @plan.1L.name
$plan.1L = declarative
debug>

```

If the option `use-display` is on and `malshow` is used as `display-cmd`, the expressions will be displayed in window on their own. If the **Expressions** window is not open yet, it will open now. If there is an open **Expressions** window, the new expressions and their values will be displayed in this window.

You can left-click on an expression to make its value disappear or appear again. You can middle-click or right-click on an expression to erase it.

The **Expressions** window has a menu with some commands:

Window

Export Postscript...

Export the displayed expressions as an Embedded Postscript file. Currently, only ASCII, Latin-1 Supplement, Hangul Compatibility Jamo and Hangul Syllables can be converted to Postscript.

Close

Close the **Expressions** window.

Style

Font Size ...

Select an item to adjust the font size.

Hanging

Normally, all values and subvalues are aligned at their bottom. If this option is active, records are “hanging down”: they are aligned at their top.

Expressions

Clear All

Clear all expressions.

Show All

Display the values of all expressions currently displayed.

Hide All

Suppress the values of all expressions currently displayed.

4.30 The Command quit

Use this command to leave malaga or mallex.

4.31 The Command read-constants (mallex)

If you want to parse lexicon entries that use Malaga constants (prefixed by ‘@’), these constants can be read in using the command ‘`read-constants lexicon-file`’. It parses *lexicon-file* and memorizes all constant definitions in it.

4.32 The Command `result`

If you have previously analysed a word form or a sentence using `ma`, `ma-line`, `sa`, or `sa-line` (in `malaga`), or you have generated allomorphs using `ga` or `ga-line` (in `mallelex`), you can display the results with `result`.

`use-display` is off: The results will be sent to standard output.

`use-display` is on and `malshow` is used as `display-cmd`: The results will show in a window on their own which is called **Results** for `malaga` and **Allomorphs** for `mallelex`. They are numbered from 1 onward.

If you are executing the command `result` for the first time, or if you have closed a **Results/Allomorphs** window that you'd opened before, a window will open, displaying the values of all results/allomorphs of the last analysis/generation.

If there is a **Results/Allomorphs** window currently opened, the new results/allomorphs will be displayed in this window.

The **Result/Allomorphs** window has a menu with some commands:

Window

Export Postscript...

Export the displayed results as an Embedded Postscript file. Currently, only ASCII, Latin-1 Supplement, Hangul Compatibility Jamo and Hangul Syllables can be converted to Postscript.

Close

Close the **Result/Allomorphs** window.

Style

Font Size ...

Select an item to adjust the font size.

Hanging

Normally, all values and subvalues are aligned at their bottom. If this option is active, records are “hanging down”: they are aligned at their top.

4.33 The Command `run`

This command can only be used in debug mode. The rule execution will be resumed, and the rules will be executed completely without any interruption.

If you have invoked debug mode by the command `debug-node`, rule execution will be stopped again when another link is going to be analysed.

4.34 The Command `sa` (`malaga`)

If you have started `malaga` with a syntax file in your command line or in the project file, you can start syntactic analyses using the command `sa` (short for *syntactic analysis*). Put the sentence you want to be analysed as argument behind the command name:

```
sa The man is in town.
```

`Malaga` will show the results automatically, and it will also show the analysis tree automatically if you specified it using the `tree` option. You can look at the results using `result` or at the entire analysis tree using `tree`.

If you do not enter a sentence behind the command `sa`, `malaga` re-analyses the last input.

4.35 The Command `sa-file` (`malaga`)

Using the command `sa-file`, you can analyse files that contain sentence lists. In a sentence list, each sentence stands in a line on its own; empty lines are permitted. Here is an example, a sentence list named `'sentence-list'`:

```
He sleeps.
He slept.
He has slept.
He had slept.
```

To analyse this sentence list, enter:

```
sa-file sentence-list result
```

This will produce a file `'result'` that contains the analysis results. If the second argument is missing, the result will be written to a file whose name ends in `'.out'`; for `'sentence-list'`, its name will be `'sentence-list.out'`.

```
1: "He sleeps.": [functor: [syn: <S3>, sem: <"sleep">]]
2: "He slept.": [functor: [syn: <S3>, sem: <"sleep">]]
3: "He has slept.": [functor: [syn: <S3>, sem: <"have", "sleep">]]
4: "He had slept.": [functor: [syn: <S3>, sem: <"have", "sleep">]]
```

The number at the line start represents the line number of the analysed original sentence. The output format can be changed by using the options `result-format` and `unknown-format`.

If a runtime error occurs during the analysis of a sentence, the line's output will be in the format given by the option `error-format`.

After the analysis, some statistics will be displayed:

- The number of analysed sentences.
- The number of recognised sentences.
- The number of sentences recognised by combi-rules and end-rules.
- The number of sentences recognised by robust-rules.
- The number of sentences whose analyses produced errors.
- The average number of results per sentence.
- The analysis run time.
- The average number of sentences that have been analysed per second.
- The number of cache accesses.
- The number of cache hits.

4.36 The Command `sa-line` (`malaga`)

If you have started `malaga` with a syntax file in your command line or in the project file, you can start syntactic analyses using the command `sa-line` (short for *syntactic analysis*). Assume you want to analyse the sentence in the third line in the file `'sentences'`. Then enter the following command:

```
sa-line sentences 3
```

Malaga will show the results automatically, and it will also show the analysis tree automatically if you specified it using the `auto-tree` option. You can look at the results using `result` or at the entire analysis tree using `tree`.

4.37 The Command `set`

This command is used to change the settings of `malaga` or `malle`. The command line ‘`set option argument`’ changes `option` to `argument`. If you want to get the current state of an option, use the command `get`. Options can also be set in the project file. The possible options are described in the next chapter.

4.38 The Command `sg` (`malaga`)

Use `sg` to generate sentences that are composed of a specified set of word forms. For example, enter:

```
sg 3 . ? he she sleeps
```

All sentences that consist of up to three word forms, where only the specified word forms (“.”, “?”, “he”, “she”, and “sleeps”) are used. The sentences are numbered from 1 onward, but different analyses of the same sentence get the same index. The output looks like this:

```
malaga> sg 3 . ? he she sleeps
1: "he sleeps ."
2: "he sleeps ?"
3: "she sleeps ."
4: "she sleeps ?"
malaga>
```

Please note that generation does not know of filters, pruning rules and default rules.

4.39 The Command `step`

This command can only be executed in debug mode. The rule execution will be resumed and continues until a different source line is met, a different path is going to be executed since the old one has terminated, or until the rules have been executed completely.

4.40 The Command `transmit`

If you have specified a `transmit` command line (to do this, use the option `transmit-cmd`), you can send a command to it:

```
malaga> set transmit-cmd cat
malaga> transmit [surf: "go", POS: verb];
[POS: verb,
 surf: "go"]
malaga>
```

4.41 The Command `tree` (`malaga`)

If you’ve started a grammatical analysis using one of the commands `ma` or `sa` (or their debug variants), you can make `malaga` display the result by entering

```
tree
```

If the analysis has not yet finished (in debug mode or in case of an error), a partial tree will be shown.

If you're executing the command `tree` for the first time, or if you've closed the `Tree` window before, a new tree window will open in which the current analysis tree will be displayed.

If there is already a `Tree` window open, the new analysis tree will be displayed in this window.

In the upper left corner of the `Tree` window, you will see the sentence or the word form that has been analysed. Below, the analysis tree is displayed. An analysis path always follows the edges from the left to the right.

A circle node stands for a LAG state, a two-circle node stands for an end state. A crossed circle stands for a LAG state that has been removed by a pruning-rule, and a crossed two-circle node stands for an end state that is invalid because it has some remaining input still remaining. A box node is not a state, but a *dead end*, which means that no rule has created a state at this position.

Above each edge, the link's surface of the corresponding rule application is displayed. Below the edge, you'll see the name of the applied rule.

You can click on a node using the left mouse button. Then another window will open, namely the `Path` window. The `Path` window displays the surface, the feature structure and the successor rules of the state you've clicked on. The node will be highlighted by a red border.

If you press the right mouse button while the mouse is on a node, a pop-up menu will appear. You can then either select that this node is the first node of the path to be displayed, or you can select it to be the last one. All rule applications, from the first node up to the last node in the path, will be displayed in the `Path` window. The corresponding path will be highlighted in the `Tree` window.

If you're clicking on a link surface using any mouse button, the surface and its feature structure will be displayed in the `Path` window.

You can also click on rule names using any mouse button. Then the corresponding rule application will be displayed in the `Path` window, i.e. the surfaces and feature structures of the original state, the link, and the successor state, and the successor rules.

There are some commands that can be initiated from the `Tree` menu bar:

Window

Export Postscript...

Export the displayed analysis tree as an Embedded Postscript file. Currently, only ASCII, Latin-1 Supplement, Hangul Compatibility Jamo and Hangul Syllables can be converted to Postscript.

Close Close the `Tree` window.

Style Select an item in this menu to adjust the font size.

Tree Specify which nodes of the analysis tree are actually displayed and whether state indexes are shown.

Full Tree All analysis states are displayed, and also boxes for rule applications that did not succeed (dead ends).

No Dead Ends

All analysis states are displayed.

Complete paths

Only the nodes that are part of a complete analysis are displayed.

Show State Indexes

Toggles the display of the state's indexes.

End States

Select an end state to display in the `Path` window.

Show First

Display the first end state.

Show Previous

If there is an end state displayed in the `Path` window, jump to the previous one.

Show Next If there is an end state displayed in the `Path` window, jump to the next one.

Show Last Display the last end state.

The `Path` window has got its own menu bar which contains the menus `Window`, `Style`, and `End States` with the same menu items as the corresponding menus in the `Tree` window, and two additional options in `Style`:

Hanging Normally, all values and subvalues are aligned at their bottom. If this option is active, records are “hanging down”: they are aligned at their top.

Inline Normally, a state is displayed with surface, feature structure and rule set stacked. If this option is active, they are displayed aligned on one line.

The `Path` window also has a menu `Path`, in which you can specify whether state indexes are shown:

Show State Indexes

Toggles the display of the state's indexes.

4.42 The Command `up`

If you want to look at the source and the variables of the (sub)rule that has called the current subrule, you can do this by entering `up`. You can list the frames via `backtrace`.

4.43 The Command `variables`

If you invoke `variables`, you get the values of all Malaga variables that are currently defined. The variables will be shown in the order of their definitions. You can only use the command `variables` in debug mode or if the previous analysis has stopped with an error in the combination rules.

If the option `use-display` is off, the variables will be sent to standard output:

```
malaga> sa-debug You are so beautiful.
entering rule "Noun", surf: "", link: "You", state: 1
debug> variables
$sentence = [class: main_clause,
             parts: <>]
$word = [class: pronoun,
         result: S2]
debug>
```

If the option `use-display` is on and `malshow` is used as `display-cmd`, the variables will be displayed in window on their own. If the `Variables` window is not open yet, it will open now. If there is an open `Variables` window, the new variable contents will be displayed in this window.

You can left-click on a variable name to make its value disappear or appear again.

The `Variables` window has a menu with some commands:

Window

- Export Postscript...
Export the displayed variables as an Embedded Postscript file. Currently, only ASCII, Latin-1 Supplement, Hangul Compatibility Jamo and Hangul Syllables can be converted to Postscript.
- Close Close the `Variables` window.

Style

- Font Size ...
Select an item to adjust the font size.
- Hanging Normally, all values and subvalues are aligned at their bottom. If this option is active, records are “hanging down”: they are aligned at their top.

Variables

- Show All Display the values of all variables currently defined.
- Hide All Suppress the values of all variables currently defined.

4.44 The Command `walk`

This command works in debug mode only. The rule execution will be continued and stopped again as soon as a new rule is executed, a breakpoint is met or there are no more rules to execute.

4.45 The Command `where`

This command can only be used in debugger mode or after rule execution has been stopped by an error. It displays the name of the rule that has been executed; additionally, the surfaces of state and link are displayed in `malaga`. For example:

```
debug> where  
at rule "flexion", surf: "hous", link: "es", state: 2  
debug>
```

5 The Options of `malaga` and `malleX`

The programs `malaga` and `malleX` share some of their options, so I will describe them in a common chapter. Options can be set using the command `set`, and you can get the current value of an option using `get`. Options that can be used in `malaga` or in `malleX` only, are marked by the name of the program in which they can be used.

5.1 The Option `alias`

With `alias`, you can define abbreviations for longer command lines. As arguments, give an alias name and an expansion (a command line which the name will stand for). If the expansion contains spaces, enclose it in double quotes. Use `set alias name` to delete alias `name`.

If you type the name of an alias at your command line, its expansion will be executed. The character sequence `'%a'` in your alias definition will be replaced by what follows the alias name in the command line.

Aliases cannot be nested.

5.2 The Option `allo-format` (`malleX`)

With `allo-format`, you can change the output format for the generated allomorphs. Enter a format string as argument. If the format string contains spaces, enclose it in double quotes. If the argument is an empty string (`""`), no allomorphs will be shown.

In the format string, the following sequences have a special meaning:

- `'%c'` Will be replaced by the allomorph's feature structure.
- `'%n'` Will be replaced by the allomorph's number.
- `'%s'` Will be replaced by the allomorph's surface.

5.3 The Option `auto-tree` (`malaga`)

You can use `auto-tree` to make `malaga` execute the `tree` command each time when you invoked an analysis by `ma` or `sa`. Set it in one of the following ways:

```
set auto-tree yes
```

The `tree` command will be executed after each analysis.

```
set auto-tree no
```

The `tree` command will not be executed automatically.

5.4 The Option `auto-variables`

When `malaga` or `malleX` stops in debug mode while executing a `malaga` rule, they can automatically show the defined variables at this point. Use the option `auto-variables` to set this behaviour.

```
set auto-variables yes
```

The `variables` command will be executed each time when `malaga` or `malleX` stops in debug mode.

```
set auto-variables no
```

The `variables` command will not be executed automatically.

5.5 The Option `cache-size` (`malaga`)

Malaga has a cache for word forms. You can set the cache size, i.e. the maximum number of words in the cache, to n with `set cache-size n`. If you set the cache size to 0, the cache will be deactivated.

When `malaga` analyses a word form or sentence, it tries to get a word form from the cache before it uses the morphology combination rules. Therefore, `malaga` separates the first word form from the remaining input. It uses spacing characters as separators; so if a word-form contains a space or does not end with a space, caching will not work.

5.6 The Option `display-cmd`

The programs `malaga` and `mallelex` normally use the program `malshow` for GUI-based display of Malaga trees, results or variables. If you want to use a different display program, set the command line that starts this program with the `display` option, like this:

```
set display-cmd "java -classpath /opt/malaga/amalgam Amalgam"
```

5.7 The Option `error-format` (`malaga`)

With `error-format`, you can change the output format for items that produced an analysis error. Enter a format string as argument. If the format string contains spaces, enclose it in double quotes. If the argument is an empty string (`"`), no forms that produced an error will be shown.

In the format string, the following sequences have a special meaning:

- '%e' Will be replaced by the error message for the analysed form.
- '%l' Will be replaced by the line number of the analysed form.
- '%n' Will be replaced by the number of analysis states for this form.
- '%s' Will be replaced by the surface.

5.8 The Option `hidden`

Some grammars can produce very large feature structures, so it can be useful not to show the values of some specified attributes. To achieve this, use the option `hidden`. You can give any number of arguments to this option. The following arguments are available:

```
'+attribute-name'
```

The specified attribute name will be put in parentheses if it occurs in a value; the attribute value will not be shown.

```
'-attribute-name'
```

The specified attribute will be shown completely again in the future.

```
'none'
```

All attributes will be shown completely again in the future.

5.9 The Option `mor-incomplete` (`malaga`)

If you want to get morphological analysis results not only for the whole input line, but for any grammatically well-formed prefix of the input line, you can use the option `mor-incomplete`:

```
set mor-incomplete yes
    Accept words that have been incompletely parsed.
```

```
set mor-incomplete no
    Only accept words that have been completely parsed.
```

Note that this option has no effect in subordinate morphological analyses that are needed by syntactic analysis.

5.10 The Option `mor-out-filter` (`malaga`)

Use the option `mor-out-filter` to switch the morphology output-filter on or off:

```
set mor-out-filter yes
    Activate the filter.
```

```
set mor-out-filter no
    Deactivate the filter.
```

5.11 The Option `mor-pruning` (`malaga`)

In your morphology rules, you may have specified a pruning rule that can prune the morphology analysis tree, i.e. it can reduce the number of parallel paths. If you want this pruning rule to be executed, use the option `mor-pruning`. Use one of the following arguments:

```
set mor-pruning n
    Call the morphology pruning rule whenever at least n states have consumed the
    same amount of input, for  $n > 0$ .
```

```
set mor-pruning 0
    Deactivate the morphology pruning rule.
```

5.12 The Option `result-format` (`malaga`)

With `result-format`, you can change the output format for analysed items that have been recognised. Enter a format string as argument. If the format string contains spaces, enclose it in double quotes. If the argument is an empty string (`""`), no recognised forms will be shown.

In the format string, the following sequences have a special meaning:

| | |
|-------------------|---|
| <code>'%c'</code> | Will be replaced by the result feature structure of the analysis. |
| <code>'%l'</code> | Will be replaced by the line number of the analysed form. |
| <code>'%n'</code> | Will be replaced by the number of analysis states for this form. |
| <code>'%r'</code> | Will be replaced by the reading index (the results for a form are indexed from 1 to the number of results). |
| <code>'%s'</code> | Will be replaced by the surface. |

5.13 The Option `result-list` (`malaga`)

With this command, you can specify whether you want `malaga` to pack all analysis results into a single list. This option only has an impact in filter mode or when a file is being analysed. Even results of different lengths are combined; this could not be achieved by an output-filter. Results of different lengths can occur when the option `mor-incomplete` or `syn-incomplete` is active.

`set result-list yes`
Combine results into a single list.

`set result-list no`
Leave results unchanged.

5.14 The Option `robust-rule` (`malaga`)

With this command, you can specify if you want to run a robust-rule for the word forms that could not be recognised by LAG rules. The robust-rule gets the surface of an unknown word form as parameter and it can create one or more results by executing the `result` statement.

`set robust-rule yes`
Enable the robust rule.

`set robust-rule no`
Disable the robust rule.

5.15 The Option `roman-hangul`

If you are using the option `'split-hangul-syllables: yes'` in your project file, `Malaga` can transcribe Hangul using Latin letters, basing on the Yale system. The transcribed text is enclosed in curly braces, and each syllable starts with a dot.

`set roman-hangul yes`
Display Hangul using Latin transcription.

`set roman-hangul no`
Display Hangul directly.

5.16 The Option `sort-records`

There are different ways to determine the order in which the attributes of a record are displayed. With `sort-records`, you can choose between three order schemes:

`set sort-records internal`
The attributes will be displayed in the order they have internally.

`set sort-records alphabetic`
The attributes will be ordered alphabetically by their names.

`set sort-records definition`
The attributes will be ordered by their names; the order is the same as in the symbol table.

5.17 The Option `switch`

Malaga rules can query simple Malaga values (*switches*) that you can change during run time. Use the option `switch` to change the values:

```
set switch name value
```

Set the switch *name*, which must be a symbol, to *value*, which can be any Malaga value.

5.18 The Option `syn-incomplete` (`malaga`)

If you want to get syntactic analysis results not only for the whole input line, but for any grammatically well-formed prefix of the sentence, you can use the option `syn-incomplete`:

```
set syn-incomplete yes
```

Accept sentences that have been incompletely parsed.

```
set syn-incomplete no
```

Only accept sentences that have been completely parsed.

5.19 The Option `syn-in-filter` (`malaga`)

Use the option `syn-in-filter` to switch the syntax input-filter on or off:

```
set syn-in-filter yes
```

Activate the filter.

```
set syn-in-filter no
```

Deactivate the filter.

5.20 The Option `syn-out-filter` (`malaga`)

Use the option `syn-out-filter` to switch the syntax output-filter on or off:

```
set syn-out-filter yes
```

Activate the filter.

```
set syn-out-filter no
```

Deactivate the filter.

5.21 The Option `syn-pruning` (`malaga`)

In your syntax rules, you may have specified a pruning rule that can prune the syntax analysis tree, i.e. it can reduce the number of parallel paths. If you want this pruning rule to be executed, use the option `syn-pruning`. Use one of the following arguments:

```
set syn-pruning n
```

Call the syntax pruning rule whenever at least *n* states have consumed the same amount of input, for $n > 0$.

```
set syn-pruning 0
```

Deactivate the syntax pruning rule.

5.22 The Option `transmit-cmd`

If you want to use the `transmit` function in `malaga` or `mallex`, you have to set a command line that starts the transmit process using the `transmit-cmd` option. Here is an example:

```
set transmit-cmd "perl my-transmit-program.pl"
```

5.23 The Option `unknown-format` (`malaga`)

With `unknown-format`, you can change the output format for analysed items that have not been recognised. Enter a format string as argument. If the format string contains spaces, enclose it in double quotes. If the argument is an empty string (`""`), no unrecognised forms will be shown.

In the format string, the following sequences have a special meaning:

- '%l' Will be replaced by the line number of the analysed form.
- '%n' Will be replaced by the number of analysis states for this form.
- '%s' Will be replaced by the surface.

5.24 The Option `use-display`

If you want the output of the commands `result` and `variables` to be shown by the `Display` process, use the option `use-display`:

```
set use-display yes
    Use the Display process to show the output of result and variables.
```

```
set use-display no
    Send the output of result and variables to your terminal.
```

6 The Programming Language Malaga

6.1 Characterisation of Malaga

A malaga rule file resembles much in programming languages like Pascal or C (of course, those languages do not have a Left-Associative Grammar formalism built in). A malaga source file must be translated before execution, this is the same as for compiler languages. But the generated Malaga code is not a machine code, but an *intermediate code* and has to be executed (*interpreted*) by an analysis program. Malaga may be characterised as follows, as far as programming structures and data structures are concerned:

structured values:

The basic values in Malaga are symbols (names that can be used e.g. for categories or subcategories), numbers (floating point numbers), and strings. Values can be combined to ordered lists or records (also known as attribute-value matrixes). A value in a list or a record can be a list or a record itself. An “ambiguous” symbol like `singular_plural` can be assigned a list of symbols like `<singular, plural>`; such a symbol is called a *multi-symbol*.

structured statements:

In Malaga, the concept of statement blocks is implemented in a similar way as it is in the programming language Pascal. There are structured control statements to select or repeat a statement sequence. A variable is always defined *locally*, i.e. it only exists from the point where it has been defined up to the end of the statement sequence in which it has been defined.

no type restrictions:

Any value can be assigned to a variable and the programmer can freely define the structure of values.

no side effects:

Malaga is, unlike programming languages like Pascal or C, free of side effects. If a variable gets a value, no other variable will be changed. Analysis paths are independent of each other.

termination:

A Malaga grammar that contains no recursive subrules and no `repeat` statements is guaranteed to terminate, i.e. it can never hang in a loop.

variables: In a `define` statement, a variable is defined and gets an initial value. Use an assignment to set a variable that has already been defined to a new value.

operators: Many generative grammar theories or linguistical programming languages use the concept of unification of feature structures. Malaga does not use unification, but it offers some operators to build feature structures explicitly. Since Malaga does without unification, analyses are much faster.

6.2 Malaga Source Texts

Source texts in Malaga must be in the Unicode UTF-8 format. They are format-free; this means that between lexical symbols (strings, identifiers, keywords, numerals and symbols

such as ‘+’, ‘~’ or ‘:=’) there may be blanks or newlines (whitespaces) or comments. Between two identifiers or two keywords there *must* be at least one whitespace to separate them syntactically.

6.2.1 Comments

A comment may be inserted everywhere where a whitespace may be inserted. A comment begins with the symbol ‘#’ and extends to the end of the line. Comments are being ignored.

6.2.2 The include Statement

A Malaga file may contain the statement

```
include "filename";
```

In a rule file, it can stand everywhere a rule can stand. In lexicon files, it can stand in place of a value; in symbol files, it can replace a symbol definition. The text of the included file is inserted verbatim at the very location where the `include` statement occurs. The file name has to be stated relatively to the directory of the file which contains the `include` statement.

6.2.3 Identifiers

In Malaga, names for variables, constants, symbols, and rules, and (see below for explanation) are called *identifiers*. An identifier may consist of uppercase and lowercase characters, the underscore ‘_’, the ampersand ‘&’, the vertical bar ‘|’, and, from the second character on, also of digits. Uppercase and lowercase characters are not distinguished, i.e., Malaga is *not* case-sensitive. Malaga keywords must not be used as identifiers. A variable name must start with a ‘\$’, a constant name must start with a ‘@’. The same identifier may be used as variable name, constant name, symbol name, or rule name independently. Malaga can distinguish them by the context in which they occur.

Valid identifiers would be ‘Noun’, ‘noun’ (the same as the first), ‘R2D2’, ‘Vb_aux’, ‘A|G|D’, ‘_INF’. Identifiers like ‘2Noun’, ‘Verb.Frame’, ‘OK?’, ‘_~INF’ are *not* valid.

6.3 Values

Malaga expressions can have values with very complex structures. To describe how those values can be composed from simple values a few rules suffice. Simple values in Malaga are *symbols*, *numbers*, and *strings*, which can be composed to form *records* and *lists*.

6.3.1 Symbols

The central data type in Malaga is the symbol. It is used for describing syntactic or semantic properties of an allomorph, a word, or a sentence. A symbol is an identifier like ‘Verb’, ‘reflexive’, ‘Sing_1’. The symbols ‘nil’, ‘yes’, ‘no’, ‘symbol’, ‘string’, ‘number’, ‘list’, and ‘record’ are predefined and have special meanings.

6.3.2 Numbers

A number in Malaga consists of an integer part, an optional fractional part and an optional exponent of the form ‘E[+|-]n’. There must be a dot between the integer part and the fractional part. Examples: ‘0’, ‘1’, ‘1.0’, ‘13.75’, ‘1.2E-5’.

Alternatively, a number may consist of an integer number followed by ‘L’, indicating that the number is intended as a list index counting from the *left* border), or by ‘R’, indicating

that the number is intended as a list index counting from the *right* border. Examples: 5L = 5, 12R = -12.

6.3.3 Strings

A string may consist of any number of characters (it may also be empty). It must be enclosed in double quotes and must not extend over more than one line. Within the double quotes there may be any combination of printable characters except the backslash ‘\’ and the double quotes. When part of a string, these characters must be preceded by a ‘\’ (escape character). Examples: "Hello", "He says: \"Great\"".

6.3.4 Lists

A list is an ordered sequence of values. The values are separated by commas and enclosed in angle brackets:

```
<element1, element2, ...>
```

A list may as well be empty. The elements in a list may be arbitrarily complex; they may also be lists or records.

6.3.5 Records

A record is a collection of attributes. An *attribute* consists of a symbol, the *attribute name*, and an associated *attribute value*, which can be an arbitrary Malaga value. The attribute name serves as an access key for the attribute value, so all attributes in a record must have distinct names.

Records are noted down as follows:

```
[name1: value1, name2: value2, ...]
```

where *name i* denotes an attribute name and *value i* the associated attribute value. Example: [Class: Verb, Reg: Reg, Val: dirObj].

A record with no attributes, [], is called *empty record*.

6.4 Expressions

An expression is the form in which a value is used in Malaga. Values can be written as follows:

```
[Surf: "he", Class: Pron, Case&Number: S3]
```

Variables (these are placeholders for values within a rule) can as well be used as expressions:

```
$Pron
```

Furthermore, constants (placeholders for values in a rule file) can be used as expressions:

```
@combination_table
```

All three forms can be mixed:

```
[Surf: "he", Class: Pron, Case&Number: $result]
```

Furthermore, there are operators which modify values or combine two values to form a new value. Complex values can be composed using those operators. All operators have a priority assigned. An operator with higher priority is applied before an operator with lower priority. If two operators have the same priority, they are applied from the left to the right. The order in which the operators are to be applied can be changed by bracketing with round parentheses ‘()’.

| | |
|-----------|--------------------|
| unary ‘-’ | very high priority |
| ‘.’ | high priority |
| ‘*’, ‘/’ | middle priority |
| ‘+’, ‘-’ | low priority |

6.4.1 Variables

A variable is marked by a ‘\$’ preceding its name. The name may be any valid identifier. A variable is defined by the `define` statement; it receives a value and may from this point on be used in all expressions within the statement sequence. In such a statement sequence (and all subordinated statement sequences) a variable with the same name must not be defined again.

6.4.2 Constants

A constant is marked by a ‘@’ preceding its name. The name may be any valid identifier. A constant is defined by a constant definition in a rule file, outside a rule. It is assigned a value and can be used in subsequent rules and constant definitions in that rule file.

6.4.3 Subrule Invocations

A subrule is invoked when an expression `subrule(value1, value2, ...)` is evaluated. The expression yields the value that is returned by the `return` statement in the subrule. The number of parameters in a subrule invocation must match the number of parameters in the subrule definition.

There is a number of default subrules which are predefined. They are called *functions*.

6.4.4 The Function atoms

The expression `atoms(symbol)` yields the list of atomic symbols for *symbol*. If *symbol* is not a multi-symbol, it yields the list `<symbol>`.

6.4.5 The Function capital

The expression `capital(string)` yields `yes` if the first character of *string* is a capital letter, else it yields `no`.

6.4.6 The Function floor

The expression `floor(number)` yields the largest integer number that is not greater than *number*.

6.4.7 The Function length

The expression `length(list)` yields the number of elements in *list*.

The expression `length(string)` yields the number of characters in *string*.

6.4.8 The Function multi

The expression `multi(list)` where *list* is a list of symbols, yields the multi-symbol whose atomic list corresponds to *list*. If *list* contains a single atomic symbol, this symbol will be yield by the expression.

6.4.9 The Function `set`

The expression `set(list)` yields a list which contains each element of *list*, but only once. That means, the list is converted to a set.

6.4.10 The Function `substring`

The expression `substring(string, start_index, end_index)` yields the substring of *string* that starts at *start_index* and ends at *end_index*, both inclusive. A positive index counts from the string start: 1L is the index of the first character; a negative index counts from the string end: 1R is the index of the last character. If *end_index* is omitted, it is assumed to be the same as *start_index*, so `substring(string, index)` yields the character at *index* in *string*. If *end_index* is less than *start_index*, the function yields an empty string.

6.4.11 The Function `switch`

The expression `switch(symbol)` yields the current value of the switch associated to *symbol*. Use the option `switch` to change this value.

6.4.12 The Function `transmit`

The expression `transmit(value)` writes *value*, converted to text format, to the transmit process via pipe and reads a value in text format from the transmit process via pipe. The answer is converted to the internal Malaga value format and returned as the result of the expression.

When this function is evaluated, the transmit process is started if it is not running. The command line of the transmit process is specified by the option `transmit`.

6.4.13 The Function `value_string`

The expression `value_string(value)` returns *value* converted to text format as a string.

6.4.14 The Function `value_type`

The expression `value_type(value)` yields the type of *value*. The type information is coded as one of the symbols `symbol`, `string`, `number`, `list`, or `record`.

6.4.15 The if Expression

An if expression has the following form:

```

if condition1 then
    expression1
elseif condition2 then
    expression2
else
    expression3
end if

```

The `elseif` part may be repeated unrestrictedly (including zero times).

First, *condition1* is evaluated. If it is satisfied, the expression *expression1* is evaluated and yields the value of the if expression.

If *condition1* is not satisfied, each condition following an **elseif** keyword is evaluated in turn, until a condition is found that is satisfied. The expression that follows this condition will be evaluated and yields the value of the **if** expression.

If the **if** condition and **elseif** conditions all fail, the expression *expression3* will be evaluated and yields the value of the **if** expression.

The **if** after the **end** may be omitted.

6.4.16 Unary ‘-’

A ‘-’ in front of a value of type **number** negates that value.

6.4.17 The Operator ‘.’

This operator may only be used in the following ways:

record.symbol

This yields the attribute value of the attribute of *record* whose name is *symbol*. If there is no attribute in *record* whose name is *symbol*, the expression yields the special symbol **nil**.

list.number

This yields the element of *list* at position *number*. If there is no element at position *number* in *list*, the expression yields the special symbol **nil**.

value.list

Here, *list* must be a list $\langle e1, e2, \dots \rangle$ of symbols and/or numbers. This expression serves as an abbreviation for *value.e1.e2...*

6.4.18 The Operator ‘+’

This operator may only be used in the following ways:

string1 + string2

This yields the concatenation of *string1* and *string2*.

list1 + list2

This yields the concatenation of *list1* and *list2*.

number1 + number2

This yields the sum of *number1* and *number2*.

record1 + record2

This yields a record which consists of all attributes of *record1* and *record2*. If *record1* and *record2* have a common attribute names, the corresponding attributes in the result record will have the attribute values from *record2*, in contrast to the operator ‘*’.

6.4.19 The Operator ‘-’

This operator may only be used in the following ways:

record - symbol

This yields *record* without the attribute named *symbol*, if *symbol* is an attribute name in *record*. If not, the expression yields *record*.

record - list

Here, *list* must be a list of symbols. This yields *record* without the attributes in *list*.

list - number

This yields *list* without the element at index *number*. If this element does not exist, the expression yields *list*.

list1 - list2

This yields the multi-set difference of the two lists *list1* and *list2*. This means, it yields the list *list1*, but the first *n* appearances of each element will be deleted, if that element appears *n* times in *list2*.

number1 - number2

This yields the difference of *number1* and *number2*.

6.4.20 The Operator ‘*’

This operator may only be used in the following ways:

record * symbol

This yields the record which only contains the attribute of *record* whose name is *symbol*.

record1 * record2

This yields a record which consists of all attributes of *record1* and *record2*. If *record1* and *record2* have a common attribute names, the corresponding attributes in the result record will have the attribute values from *record1*, in contrast to the operator ‘+’.

record * list

Here, *list* must be a list of symbols. This yields the record which only contains the attributes of *record* whose names are in *list*.

list1 * list2

This yields the *intersection* of the lists interpreted as multi-sets; if an element is *m* times contained in *list1* and *n* times contained in *list2*, it will be $\min(m, n)$ times contained in the result.

number1 * number2

This yields the product of *number1* and *number2*.

6.4.21 The Operator ‘/’

This operator may only be used in the following ways:

list1 / list2

This yields the list which contains all elements of *list1* which are not elements of *list2*.

list / number

This yields the list which contains all elements of *list* without the leftmost *number* elements, if *number* is positive, or without the rightmost *-number* elements, if *number* is negative.

number1 / *number2*

Here, *number2* must not be 0. This yields the quotient of *number1* and *number2*.

6.5 Conditions

A condition can either be true or false, as in **Verb = Verb** or **Verb = Noun**, respectively. An expression that is evaluated to any of the symbols **yes** or **no** is a valid condition.

A condition can be used in all places where a non-constant value is needed. It will evaluate to **yes** or **no**. In this case, the condition must be surrounded by parentheses.

6.5.1 The Operators '=' and '/='

The condition *expr1* = *expr2* tests whether the expressions *expr1* and *expr2* are equal. Depending on the types of *expr1* and *expr2*, equality is defined as follows:

expr1 and *expr2* are both symbols or both numbers.

In this case *expr1* and *expr2* must be identical.

expr1 and *expr2* are strings.

In this case *expr1* and *expr2* must be the same, but the test is case-insensitive.

expr1 and *expr2* are lists.

In this case *expr1* and *expr2* must have the same length, and, for each *i*, the *i*-th element of *expr1* must be equal to the *i*-th element of *expr2*.

expr1 and *expr2* are records.

In this case *expr1* and *expr2* must contain the same attribute names, though not necessarily in the same order. For each attribute name, the attribute value of *expr1* and the attribute value of *expr2* must be equal.

If *expr1* and *expr2* do not have the same type and are both different from the symbol **nil**, the test results in an error; the symbol **nil** can be compared to any value without error message.

The test *expr1* /= *expr2* holds if and only if the test *expr1* = *expr2* does not hold.

6.5.2 The Operators less, less_equal, greater, greater_equal

A condition of type *expr1 operator expr2* compares two numbers. Here, *operator* can have the following values:

less The condition holds if *expr1* has a smaller value than *expr2*.

less_equal

The condition holds if *expr1* has a smaller value than *expr2* or both numbers are equal.

greater The condition holds if *expr1* has a bigger value than *expr2*

greater_equal

The condition holds if *expr1* has a bigger value than *expr2* or both numbers are equal.

If either *expr1* or *expr2* is no number, an error will be reported.

6.5.3 The Operators ‘~’ and ‘/~’

The operator ‘~’ can be used in the following ways:

list1 ~ *list2*

This tests whether *list1* and *list2* do *congruate*, this means, whether they have at least one element in common.

symbol1 ~ *symbol2*

This tests if `atoms(symbol1)` and `atoms(symbol2)`, the lists of their atomic symbols, do congruate.

The comparison *expr1* /~ *expr2* holds if and only if the comparison *expr1* ~ *expr2* does not hold.

6.5.4 The Operator in

The operator `in` can be only used in the following ways:

symbol in *record*

This condition holds if and only if *record* contains an attribute named *symbol*.

value in *list*

This condition holds if and only if *value* is an element of *list*.

6.5.5 The matches Condition (Regular Expressions)

The condition *expr* matches *pattern* or *expr* matches (*pattern*) interprets *pattern* as a pattern (a regular expression) and tests whether *expr* matches *pattern*. Patterns are defined as follows:

pattern ::= *alternative* {‘|’ *alternative*}

The string must be identical with one of the alternatives.

alternative ::= {*atom* [* | ? | +]}

An alternative is a (possibly empty) sequence of atoms. An atom in a pattern corresponds to a character in a string. By using an optional postfix operator it is possible to specify for any atom how often it may be repeated within the string at that location: zero times or once (‘?’), at least once (‘+’), or arbitrarily often, including zero times (‘*’).

Normally, these operators are *greedy*, i.e. they try to match as much as possible. If you put a ‘?’ behind a postfix operator, it will try to match as few characters as possible. This can make a difference if you’re assigning variables in your pattern.

atom ::= ‘(*pattern*)’

A pattern may be grouped by parentheses.

atom ::= ‘[[‘^’] *range* {*range*}]’

A character class. It represents exactly one character from one of the ranges. If the symbol ‘^’ is the first one in the class, the expression represents exactly one character that is *not* contained in one of the ranges.

atom ::= ‘.’

Represents any character.

atom ::= *character*

Represents the character itself.

range ::= *character1* [*'-* *character2*]

The range contains any character with a code at least as big as the code of *character1* and not bigger than the code of *character2*. The code of *character2* must be at least as big as the code of *character1*. If *character2* is omitted, the range only contains *character1*.

character ::= Any character except *'*?+[]^-.\|()'*

To use one of the characters *'*?+[]^-.\|()'*, it must be preceded by a *'\'* (pattern escape). To insert the pattern escape itself, you have to double it: *'\\'*.

You can divide the pattern into segments:

```
$surf matches ("un|in|im|ir|il", ".*", "(en)?")
```

is is the same as

```
$surf matches ("(un|in|im|ir|il).*(en)?")
```

A section of the string can be stored in a variable by suffixing the respective pattern with *': variable_name'*, as in

```
$surf matches ("un|in|im|ir|il": $a, ".*")
```

For backwards compatibility, you may also prefix the pattern with the variable name, as in

```
$surf matches $a: "un|in|im|ir|il", ".*"
```

The variables defined by pattern matching are only defined in the statement sequence which is being executed if the pattern matching is successful. A *matches* condition may not have variable definitions in it if it is

- contained in a disjunction (an *or* condition),
- contained in a negation (a *not* condition), or
- used as a truth value (e.g. in an assignment).

6.6 The Operators *not*, *and*, *or*

Conditions can be combined logically:

not cond This is true if condition *cond* is false.

cond1 and cond2 and cond3 and ...

This is true if all conditions *cond1*, *cond2*, *cond3*, ... are true. The conditions are tested one by one from left to right until one of them is false. This is called *short-cut evaluation*.

cond1 or cond2 or cond3 or ...

This is true if at least one of the conditions *cond1*, *cond2*, *cond3*, ... is true. The conditions are tested one by one from left to right until one of them is true. This is also a form of short-cut evaluation.

The operator *not* takes exactly one argument. If its argument contains another logical operator, put it in parentheses *'()'*, as in *not (cond1 or cond2)*.

The operators `and` and `or` may not be mixed as in `cond1 and cond2 or cond3`; here the order of evaluation would be ambiguous. Use parentheses ‘()’ to indicate in which order the condition is to be evaluated, as in `(cond1 and cond2) or cond3`.

6.7 The Symbol Table

Every symbol used in a grammar has to be defined at least once in the *symbol table*. Every symbol must be followed by a semicolon: `verb; noun; adjective;`

Symbols that are being defined that way are called *atoms*. A symbol can also be defined as a *molecule*. Then the entry for this symbol has the following format:

```
symbol := list;
```

The *list* for this symbol must consist of at least two atoms; no atom may occur more than once in the list. This list will be used by the operators ‘`~`’ and ‘`/~`’, `atoms`, and `multi`. The lists in the symbol table must be different from each other; it does not suffice that they only differ in the order of their elements. If a symbol is defined more than once in the symbol table, the definitions must all match: Either the symbol must always be defined atomic or it must always be molecular with the same atom-list.

6.8 The Initial State

The initial state in a combination rule file is defined as follows:

```
initial value, rules rule1, rule2, ...;
```

The initial state of a combi rule file specifies a feature structure and a list of rules (behind the keyword `rules`). Each of the rules will be applied to read in the first allomorph (in morphology) or word form (in syntax). The list may be enclosed in parentheses.

A combi rule or an end rule is successful if it creates at least one new state, otherwise it fails. If you want rules to be executed only if all other rules failed, you can put their names behind the other rules’ names and write an `else` in front of them:

```
initial value, rules rule1, rule2 else
rule3, rule4 else ...;
```

If both rules `rule1` and `rule2` fail, `rule3` and `rule4` are executed. If these rules also fail, the next rules are executed, and so on.

6.9 The Constant Definition

A constant definition is of the form

```
define @constant := expr;
```

The constant expression `expr` will be evaluated and the constant `@constant` will be defined to have this value. The constant must not have been defined previously. The constant is valid from this definition up to the end of the rule file. If you use the keyword `default` instead of `define`, you provide a default value for `@constant`. This means, the value is only preliminary and may be changed by a normal constant definition. After a constant has been used in an expression, its value may not be changed any more.

6.10 Rules

A rule is a sequence of statements that is executed as a unit:

```
combi_rule name($param1, $param2, ...):
    statement1
    statement2
    ...
end name;
```

A rule has to begin with one of the keywords `allo_rule`, `combi_rule`, `end_rule`, `pruning_rule`, `robust_rule`, `input_filter`, `output_filter` or `subrule`. It is followed by its *parameter list*, a list of variable names. The variables will be assigned the parameter values when the rule is executed. The number of parameters depends on the rule type. The rule names have the following meanings:

`allo_rule($lex_entry)`

An allo-rule must occur exactly once in an allomorph rule file. It analyses a lexical entry and must generate one or more allomorph entries via `result`. An allomorph rule has one parameter, namely the lexicon entry.

`combi_rule($state, $link, $surf, $index)`

Any number of combi-rules may occur in a combi-rule file. Before processing such a rule, the *link* is read in, which is either the word form or the allomorph that follows the state's surface. The first parameter of the rule is the state's feature structure, the second is the link's feature structure, the third is the link's surface, and the fourth is the link's index. The third and the fourth parameter are optional. A combi-rule may state a successor rule set or accept the analysed input (both via `result`).

`end_rule($state, $remain_input)`

Any number of end-rules may occur in a combi-rule file. The first parameter is the state's feature structure, the second, which is optional, is the remaining input. If the rule takes only one parameter, it is only called if the remaining input is empty or begins with a space. An end rule may accept the analysed input via `result`.

`pruning_rule($list)`

A pruning-rule may occur at most once in a combi-rule file. During analysis, it can decide which states are still valid and which are to be deleted. The parameter is a list of feature structures of the states that have consumed the same input so far. The pruning-rule must execute a `return` statement with a list of the symbols `yes` and/or `no`. Each state in *\$list* corresponds to a symbol in the result list. If the symbol is `yes`, the corresponding state is preserved. If the symbol is `no`, the state is abandoned.

`robust_rule($surface, $remain_input)`

A robust-rule can only appear at most once a morphology rule file. If robust analysis has been switched on by the `robust` command, and a word form could not be recognised by the combi-rules, the robust-rule is executed with the surface of the next word form as its first parameter. The next word form is defined

as the remaining input up to (but excluding) the next space. The optional second parameter contains the whole remaining input. A robust-rule can accept any prefix of the remaining input via **result**.

input_filter(\$feature_structure_list)

An input-filter may occur at most once in a syntax rule file. The input-filter is called after a word form has been analysed. It gets one parameter, namely the list of the analysis results, and it transforms it to one or more filtered results (via **result**).

output_filter(\$feature_structure_list)

An output-filter may occur at most once in any rule file.

In allo-rule files:

The output-filter is called after all lexicon entry have been processed by the allo-rules. The filter is called for every allomorph surface. It gets one parameter, namely the list of the generated feature structures with that surface, and it transforms it to one or more filtered allomorph feature structures (via **result**).

In combi-rule files:

The output-filter is called after an item has been analysed. It gets one parameter, namely the list of the analysis results, and it transforms it to one or more filtered results (via **result**).

subrule(\$param1, \$param2, ...)

Any number of subrules may occur in any rule file. A subrule can be invoked from other rules and it must return a value to this rule via **return**. It can have any number of parameters (at least one).

If a rule is executed, all statements in the rule are processed sequentially. After that, the rule execution is terminated. Thereby, the **if** statement, the **foreach** statement, and the **select** statement may change the processing order. Special conditions apply if:

1. A condition in a **require** statement does not hold. In this case the processing of the current rule path is terminated. This is not an error.
2. The **stop** statement was executed. In this case the processing of the current rule path is terminated. This is not an error.
3. An **assert** condition does not hold. In this case the processing of the whole grammar is terminated and an error message is displayed. This rule termination can be used to find bugs in the rule system or in the lexicon.
4. The **error** statement was executed. In this case the processing of the whole grammar is terminated and an error message is displayed.
5. The **return** statement was executed in a subrule or in a pruning rule. In a subrule, this terminates the subrule into the current rule path and immediately returns to the calling rule. In a pruning rule, this terminates the pruning rule.

6.11 Statements

A rule body contains a sequence of statements.

The statements are the assignment and the statements beginning with `assert`, `choose`, `define`, `error`, `foreach`, `if`, `repeat`, `require`, `result`, `return`, `select`, and `stop`.

6.11.1 The `assert` Statement

The statement `assert condition`; or `! condition`; tests whether *condition* holds. If this is not the case, an error message with the line number in the source code is displayed and the processing of *all* paths is terminated.

The `assert` statement should be used to check whether there are structural flaws in the lexicon or the rule system.

6.11.2 The Assignment

To set the value of an already defined variable to a different value, use a statement of the following form:

```
$var := expr;
```

The expression *expr* is evaluated and the result is assigned to the variable *\$var*. The variable must have already been defined.

You can assign the elements of a list value to multiple variables at once:

```
<$var1, $var2, ... > := expr;
```

The first, second, ... element of *expr*, which must be a list, is assigned to variable *\$var1*, *\$var2*, ... respectively. Any of these variables may be followed by a path. The number of variables must match the length of the list value.

You can optionally specify a path behind the variable that is to be set by an assignment:

```
$var.part1.part2 := value;
```

In this case, only the value of *\$var.part1.part2* will be set to *value*; the remainder of the variable *\$var* will be unchanged. Each *part* must be an expression that evaluates to a symbol, a number or a list of symbols and numbers.

You can also use one of four other assignment operators instead of the operator `:=`: The statement `$var :=+ value`; is a shorthand for `$var := $var + value`;. The same holds for the assignment operators `:= -`, `:= *`, and `:= /`. Here, *\$var* may be followed by a path again.

6.11.3 The `break` Statement

The `break` statement leaves the `foreach` loop with *Label*.

```
break Label;
```

If the label is omitted, the `break` statement leaves the innermost `foreach` loop it is contained in. The statement must be situated in the body of the `foreach` loop it wants to leave.

6.11.4 The `choose` Statement

The `choose` statement chooses an element of a list. Its format is:

```
choose $var in expr;
```

For every element in the list *expr* a rule path is created; in this rule path the element is stored in the variable *\$var*. Thus the number of rule paths can multiply. If, for example, *expr* has the value `<A, B, C>`, the currently processed rule path has three continuations: In

the first one *\$var* has the value A, in the second one it has the value B and in the third one it has the value C. The three paths behave independently from now on.

The `choose` statement can also be used for records. In that case, the variable *\$var* gets a different attribute name of the record *expr* in each path.

The `choose` statement also works for numbers:

- If *expr* is a positive number *n*, the variable *\$var* is assigned the numbers 1, 2, ..., *n*, respectively, in each path.
- If *expr* is a negative number *-n*, the variable *\$var* is assigned the numbers -1, -2, ..., *-n*, respectively, in each path.

6.11.5 The continue Statement

The `continue` statement terminates the current pass of the `foreach` loop with *Label* and starts the next pass. If the current pass is the last one, the loop will be left.

```
continue Label ;
```

If the label is omitted, the statement affects the innermost `foreach` loop it is contained in. The statement must be situated in the body of the `foreach` loop it wants to affect.

6.11.6 The define Statement

A `define` statement is of the form

```
define $var := expr ;
```

The expression *expr* is evaluated and the result is assigned to the variable *\$var*. The variable may not be defined before this statement; it is defined by the statement and only exists until the statement sequence in which the assignment is situated has been processed fully.

You can assign the elements of a list value to multiple variables at once:

```
define <$var1, $var2, ... > := expr ;
```

The first, second, ... element of *expr*, which must be a list, is assigned to the new variable *\$var1*, *\$var2*, ... respectively. The number of variables must match the length of the list value.

6.11.7 The error Statement

The statement `error` terminates the execution of *all* paths and displays the given expression, which must be a string, and the line of the source text:

```
error message ;
```

6.11.8 The foreach Statement

You may wish to manipulate all elements of a list or a record *sequentially* in *one* rule path. For this purpose, the `foreach` statement was introduced. It has the following format:

```
foreach $var in expr :
  statements
end foreach ;
```

Sequentially, *\$var* is assigned a number of values, depending on the type of *expr*, and the statement sequence *statements* is executed for each of those assignments. Every time the *statements* are being walked through, the variable *\$var* is defined again. Its scope is the block *statements*.

- If *expr* is a list, *\$var* is assigned the first, second, third, ... element of *expr*.
- If *expr* is a record, *\$var* is assigned the first, second, ... attribute name of *expr*.
- If *expr* is a positive number *n*, the variable *\$var* is assigned the numbers 1, 2, ..., *n* sequentially.
- If *expr* is a negative number *n*, the variable *\$var* is assigned the numbers -1, -2, ..., -*n* sequentially.
- If *expr* is an empty list, an empty record or the number 0, the foreach loop is terminated immediately.

6.11.9 The if Statement

An if statement has the following form:

```

if condition1 then
    statements1
elseif condition2 then
    statements2
else
    statements3
end if;

```

The `elseif` part may be repeated unrestrictedly (including zero times), the `else` part may be omitted.

First, *condition1* is evaluated. If it is satisfied, the statement sequence *statements1* is executed.

If the first condition is not satisfied, *condition2* is evaluated; if the result is true, *statements2* is executed. This procedure is repeated for every `elseif` part until a condition is satisfied.

If the `if` condition and `elseif` conditions fail, the statement sequence *statements3* is executed (if it exists).

After the `if` statement has been processed, the following statement is executed.

The `if` after the `end` may be omitted.

6.11.10 The repeat Statement

You may wish to repeat a sequence of statements while a specific condition holds. This can be realised by the `repeat` loop. It has the following form:

```

repeat
    statements1
while condition;
    statements2
end repeat;

```

The statements *statements1* are executed. Then, *condition* is tested. If it holds, the *statements2* are executed and the `repeat` statement is executed again. If *condition* does not hold, execution proceeds after the `repeat` statement.

If *statements1* is empty, the `repeat` loop is equivalent to a while loop in C:

```

repeat while condition;
    statements

```

```
end repeat;
```

If *statements2* is empty, the **repeat** loop is equivalent to a do-while loop in C:

```
repeat
  statements
while condition;
end repeat;
```

6.11.11 The require Statement

A statement of the form

```
require condition;
```

or

```
? condition;
```

tests whether *condition* is true. If this is not the case the rule path is terminated *without* error message. Test statements should be used to decide whether the combination of a state and a link is grammatical.

6.11.12 The result Statement

In combi rules:

The statement

```
result expr, rules rule1, rule2, ...;
```

specifies the Result feature structure of the rule and the successor rules. The value *expr* is the Result feature structure. Behind the keyword **rules** the names of all successor rules are enumerated. For every successor rule that is being executed a new rule path will be created. The rule set may be enclosed in parentheses.

If you want successor rules to be executed only if no other rule has been successful, you can put their names behind the other rules' names and write an **else** in front of them:

```
result expr,
rules rule1, rule2 else rule3, rule4 else ...;
```

If none of the normal rules (here: *rule1* and *rule2*) has been successful, *rule3* and *rule4* are executed. If these rule also fail, the next rules are executed, and so on. A rule has been successful if at least one **result** statement has been executed.

In combi-rules and end-rules:

If the input is to be accepted by the **result** statement (and therefore no successor rules are to be called) the following format has to be used:

```
result expr, accept;
```

If this statement is reached in a rule path, the input is accepted as grammatically well-formed. The value *expr* is returned as the result of the morphological or syntactic analysis.

In filters: The format of a **result** statement in a filter or robust-rule is

```
result expr;
```

If this statement is reached, the value *expr* is used as a result of the executed rule.

In robust-rules:

The format of a **result** statement in a robust-rule:

```
result feature_structure;
```

or

```
result surface, feature_structure;
```

The word form *surface* with feature structure *feature_structure* is used as a result of the robust-rule. *surface* must be a prefix of the input that has not been parsed yet. If it is omitted, the input up to, but excluding, the first space is taken.

In allo-rules:

The format of the **result** statement in an allo rule is:

```
result surface, feature_structure;
```

It creates an entry in the allomorph lexicon. The allomorph surface *surface* must be a string; *feature_structure* is the feature structure of the allomorph.

6.11.13 The return Statement

In a subrule, the **return** statement is of the following form:

```
return expr;
```

The value of *expr* is returned to the rule that invoked this subrule and the subrule execution is finished.

In a pruning rule, the **return** statement is of the same form. Here, *expr* must be a list a list of the symbols **yes** and/or **no**. Each state in the feature structure list, which is the pruning rule parameter, corresponds to a symbol in the result list. If the symbol is **yes**, the corresponding state is preserved. If the symbol is **no**, the state is abandoned.

6.11.14 The select Statement

By using the **select** statement, more than one continuation of an analysis path can be generated. Its format is:

```
select
  statements1
or
  statements2
or
  statements3
...
end select;
```

This creates as many rule paths as there are statement sequences. In the first rule path, *statements1* are executed, in the second one *statements2* are executed, etc. Each rule path continues by executing the statements following the **select** statement.

The keyword **select** behind the **end** can be omitted.

6.11.15 The stop Statement

The `stop` statement terminates the current rule path. Its format is:

```
stop;
```

6.12 Files

A Malaga grammar system comprises several files: a symbol file, a lexicon file, an allomorph rule file, a morphology rule file, an extended symbol file (optional), and a syntax rule file (optional). The type of a file can be seen by the ending of the file name. A grammar for the English language may consist of the files `'english.sym'`, `'english.lex'`, `'english.all'`, `'english.mor'` and `'english.syn'`.

6.12.1 The Symbol File

A symbol file has the suffix `' .sym'`. It contains the symbol table.

6.12.2 The Extended Symbol File

An extended symbol file has the suffix `' .esym'`. It contains an additional symbol table that contains symbols which may only be used in the syntax rule file.

6.12.3 The Lexicon File

A lexicon file has the suffix `' .lex'`. It consists of any number of values and constant definitions, each terminated by a semicolon. Each value stands for a lexical entry. A value may contain named constants and the operators `'.'`, `'+'`, `'-'`, `'*'`, and `'/'`. values, the lexical entries; The format of the lexical entries is free, although it should be consistent with the conception of the whole rule system.

6.12.4 The Allomorph Rule File

The allomorph lexicon is generated from the base form lexicon by applying the allo-rule on the base form entries. The allomorph generation rule file has the suffix `' .all'` and consists of one allo-rule, an optional output-filter, and any number of subrules and constant definitions. For every lexical entry, the allo-rule is executed with the value of the lexicon entry as parameter. The allo-rule can generate allomorphs using the `result` statement.

After all allomorphs have been produced, the output-filter is executed once for each surface in the (intermediate) allomorph lexicon. As parameter, the output-filter gets the list of feature structures that share that surface. An entry in the final allomorph lexicon is created everytime the `result` statement is executed. The surface cannot be changed by the output-filter.

6.12.5 The Combi-Rule Files

A grammar system includes up to two combination rules files: one for morphological combination with the suffix `' .mor'` and (optionally) one for syntactic combination with the suffix `' .syn'`.

A combination rule file consists of an initial state and any number of combi-rules, subrules, and constant definitions. A syntax rule file may contain one optional pruning-rule, one optional input-filter and one optional output-filter; a morphology rule file may contain one optional robust-rule, one optional pruning-rule and one optional output-filter.

Beginning with the rules listed up in the initial state, the rules and their successors are processed until a **result** statement with the keyword **accept** is encountered in every path. A path dies if there is no more input (from the lexicon or from the morphology) that can be processed.

In morphology, if analysis has created no result and robust analysis has been switched on, the robust-rule will be called with the analysis surface and can create a result.

In syntax, when a new wordform has been imported from morphology, the input-filter can take a look at its feature structures and create new result feature structures.

If a pruning-rule is present, pruning has been activated, and the number of current LAG states is not less than **mor-pruning** (in morphology) or **syn-pruning** (in syntax), the concatenation of the next allomorph (in morphology) or word form (in syntax) is preceded by the following step: The feature structures of all current LAG states are merged into a list, which is the parameter of the pruning rule. The pruning-rule must execute a **return** statement with a list of the symbols **yes** and **no**. Each state in the feature structure list corresponds to a symbol in the result list. If the symbol is **yes**, the corresponding state is preserved. If the symbol is **no**, the state is abandoned.

After analysis has completed, the output-filter can take a look at all result feature structures and create new result feature structures. This can be used to merge similar feature structures or drop some results.

6.13 Summary of the Malaga Syntax

The syntax of Malaga source texts is defined formally by a sort of EBNF notation:

- Terminals like **assert** and **‘:=’** stand for themselves.
- Nonterminals like *assignment* are defined by *productions*.
- A bar **‘|’** separates alternatives.
- Brackets **‘[]’** enclose optional parts.
- Curly braces **‘{ }’** enclose parts that are repeated zero times, one time, or multiple times.
- Parentheses **‘()’** are used for grouping.

The start productions for Malaga source texts are *lexicon-file*, *rule-file*, and *symbol-file*. A nonterminal marked with **‘*’** in its definition is a lexical symbol.

assert-statement:

```
(assert | ‘!’) condition ‘;’
```

assignment:

```
path (‘:=’ | ‘:=+’ | ‘:=−’ | ‘:=*’ | ‘:=/’) expression ‘;’ | ‘<’ path ‘,’ path  
‘>’ ‘:=’ expression ‘;’
```

break-statement:

```
break [label] ‘;’
```

choose-statement:

```
choose variable in expression ‘;’
```

*comment**:

```
‘#’ {printing-char}
```

comparison:
 [not] (expression [comparison-operator expression] | match-comparison)

comparison-operator:
 '=' | '/=' | '~' | '/~' | in | less | greater | less_equal | greater_equal

condition: comparison ({and comparison} | {or comparison})

*constant**: '@' identifier

constant-definition:
 (define | default) constant ':=' constant-expression ';'

constant-expression:
 expression

continue-statement:
 continue [label] ';'

define-statement:
 define variable ':=' expression ';' | define '<' variable {',' variable} '>' ':='
 expression ';'

error-statement:
 error expression ';'

expression:
 term {'+' | '-'} term

factor: value {'.'} value

foreach-statement:
 [label ':'] foreach variable in expression ':' statements end [foreach] ';'

*identifier**:
 (letter | '_' | '&') {letter | digit | '_' | '&'}

if-statement:
 if condition then statements {elseif condition then statements} [else state-
 ments] end [if] ';'

if-expression:
 if condition then expression {elseif condition then expression} else expres-
 sion end [if]

include: include string ';'

initial: initial constant-expression ',' rule-set ';'

label: identifier

lexicon-file:
 {constant-definition | constant-expression ';' }

list: '<' {expression {','} expression} '>'

match: constant-expression [':' variable] | variable ':' constant-expression

match-comparison:
expression matches ('(' *match* {',' *match*} ')' | *match* {',' *match*})

number:* *digit* {*digit*} ('L' | 'R' | ['. ' *digit* {*digit*}] ['E' *digit* {*digit*}])

path: *variable* {'. ' *value*}

record: '[' {*symbol-value-pair* {',' *symbol-value-pair*}} ']'

repeat-statement:
repeat statements while condition ';' *statements end* [*repeat*] ';' ;'

require-statement:
(*require* | '?') *condition* ';' ;'

result-statement:
result expression [',' (*rule-set* | *accept*)] ';' ;'

return-statement:
return expression ';' ;'

rule: *rule-type rule-name* '(' *variable* {',' *variable*} ')' ':' *statements end* [*rule-type*] [*rule-name*] ';' ;'

rule-file: {*rule* | *constant-definition* | *initial* | *include*}

rule-name:
identifier

rule-set: *rules* (*rules* {*else rules*} | '(' *rules* {*else rules*} ')')

rule-type: *allo_rule* | *combi_rule* | *end_rule* | *pruning_rule* | *robust_rule* | *input_filter* | *output_filter* | *subrule*

rules: *rule-name* {',' *rule-name*}

select-statement:
select statements {*or statements*} *end* [*select*] ';' ;'

statements:
{*assert-statement* | *assignment* | *break-statement* | *choose-statement* | *continue-statement* | *define-statement* | *error-statement* | *foreach-statement* | *if-statement* | *select-statement* | *repeat-statement* | *require-statement* | *result-statement* | *return-statement* | *stop-statement*}

stop-statement:
stop ';' ;'

string:* '"' {*char-except-double-quotes* | '\ "' | '\ \' } '"' ;'

subrule-invocation:
rule-name '(' *expression* {',' *expression*}

symbol: *identifier*

symbol-definition:
symbol [':=' '<' *symbol* {',' *symbol*} '>'] ';' ;'

symbol-file:

{symbol-definition | include}

symbol-value-pair:

expression ':' expression

term: *factor* *{('*' | '/') factor}*

value: *[-'] (symbol | string | number | list | record | constant | subrule-invocation | variable | '(' condition ')') | if-expression*

variable:* *'\$' identifier*

Index

- ***
- * (operator) 40
- +**
- + (operator) 39
-
- (operator) 39
- .**
- .malagarc (file) 6
- /**
- / (operator) 40
- /= (operator) 41
- /~ (operator) 42
- =**
- = (operator) 41
- ~**
- ~ (operator) 42
- A**
- accept (keyword) 50
- alias (option) 28
- allo-format (mallex option) 28
- allo_rule (rule) 45
- allomorph rule files 52
- and (operator) 43
- assignment 47
- atoms 44
- atoms (function) 37
- attribute order 31
- attributes 36
- auto-tree (malaga option) 28
- auto-variables (option) 28
- B**
- backtrace (command) 12
- Beutel, Björn 1
- binary (command line option) 9
- boolean operators 43
- break (command) 12
- break (statement) 47
- breakpoints 12
- C**
- cache-size (malaga option) 29
- capital (function) 37
- choose (statement) 47
- clear-cache (malaga command) 13
- combi-rule files 52
- combi_rule (rule) 45
- commands 12
- comments 35
- conditions 41
- constant definition 44
- constants 37
- continue (command) 13
- continue (statement) 48
- D**
- debug-ga (mallex command) 14
- debug-ga-file (mallex command) 14
- debug-ga-line (mallex command) 14
- debug-ma (malaga command) 15
- debug-ma-line (malaga command) 15
- debug-sa (malaga command) 15
- debug-sa-line (malaga command) 15
- debug-state (malaga command) 15
- define (statement) 48
- definition, constant 44
- delete (command) 16
- display-cmd (option) 29
- down (command) 16
- E**
- else (keyword) 38, 49
- elseif (keyword) 38, 49
- empty record 36
- end_rule (rule) 45
- error (statement) 48
- error-format (malaga option) 29
- escape character ('\\') 36
- expressions 36
- expressions, regular 42
- extended symbol files 52
- F**
- failing rule 44
- files 52
- files, allomorph rule 52
- files, combi-rule 52
- files, extended symbol 52

- files, lexicon 52
 - files, morphology rule 52
 - files, symbol 52
 - files, syntax rule 52
 - finish** (command) 16
 - floor** (function) 37
 - font family 7
 - font size 7
 - foreach** (statement) 48
 - Formalism 2
 - frame** (command) 16
 - functions 37
- G**
- ga** (mallex command) 16
 - ga-file** (mallex command) 16
 - ga-line** (mallex command) 17
 - geometry 7
 - get** (command) 17
 - greater** (operator) 41
 - greater_equal** (operator) 41
- H**
- Hangul 6
 - Hangul, transcribed 31
 - help** (command line option) 4
 - help** (command) 17
 - hidden** (option) 29
- I**
- identifiers 35
 - if** (expression) 38
 - if** (statement) 49
 - in** (operator) 42
 - include** (statement) 35
 - indexes, state 7
 - info** (malaga command) 17
 - initial state 44
 - input** (command line option) 8
 - input_filter** (rule) 45
- L**
- LAG 2
 - length** (function) 37
 - less** (operator) 41
 - less_equal** (operator) 41
 - lexicon files 52
 - list** (command) 17
 - list assignment 47
 - lists 36
- M**
- ma** (malaga command) 18
 - ma-file** (malaga command) 18
 - ma-line** (malaga command) 19
 - malaga** (program) 8
 - Malaga, programming language 34
 - malaga.ini** (file) 6
 - mallex** (program) 9
 - malmake** (program) 10
 - matches** (operator) 42
 - mg** (malaga command) 19
 - molecules 44
 - mor-incomplete** (malaga option) 30
 - mor-out-filter** (malaga option) 30
 - mor-pruning** (malaga option) 30
 - morphology** (command line option) 8
 - morphology rule files 52
 - multi** (function) 37
- N**
- next** (command) 19
 - no** (symbol) 41
 - not** (operator) 43
 - numbers 35
- O**
- operator priority 36
 - options 28
 - or** (operator) 43
 - order, attribute 31
 - output_filter** (rule) 45
- P**
- patterns, string 42
 - prelex** (command line option) 10
 - print** (command) 19
 - priority, operator 36
 - profile 6
 - projects 4
 - Pruning 30, 32
 - pruning_rule** (rule) 45
- Q**
- quit** (command) 20
 - quoted** (command line option) 8
- R**
- read-constants** (mallex command) 20
 - readable** (command line option) 9
 - record, empty 36
 - records 36
 - regular expressions 42
 - repeat** (statement) 49
 - require** (statement) 50

- result** (command) 21
result (statement) 50
result-format (malaga option) 30
result-list (malaga option) 31
return (statement) 37, 51
robust-rule (malaga option) 31
robust_rule (rule) 45
roman-hangul (option) 31
rule files, allomorph 52
rule files, morphology 52
rule files, syntax 52
rule, failing 44
rule, successful 44
rules 45
run (command) 21
- S**
- sa** (malaga command) 21
sa-file (malaga command) 22
sa-line (malaga command) 22
Schüller, Gerald 1
select (statement) 51
set (command) 23
set (function) 38
sg (malaga command) 23
sort-records (option) 31
split-hangul-syllables 6
state indexes 7
state, initial 44
statements 46
step (command) 23
stop (statement) 52
string patterns 42
strings 36
subrule (rule) 45
subrules, calling 37
substring (function) 38
successful rule 44
switch (function) 38
switch (option) 32
symbol definition 44
symbol files 52
- symbol files, extended 52
symbol table 44
symbols 35
syn-in-filter (malaga option) 32
syn-incomplete (malaga option) 32
syn-out-filter (malaga option) 32
syn-pruning (malaga option) 32
syntax (command line option) 8
syntax rule files 52
syntax, Malaga 53
- T**
- transcribed Hangul 31
transmit (command) 23
transmit (function) 38
transmit-cmd (option) 33
tree (malaga command) 23
- U**
- unknown-format** (malaga option) 33
up (command) 25
use-display (option) 33
- V**
- value_string** (function) 38
value_type (function) 38
values 35
variables (command) 25
version (command line option) 4
- W**
- walk** (command) 26
where (command) 26
window geometry 7
- Y**
- yes** (symbol) 41