Manual for the AGFL system
version 2.3

January 2004

the AGFL system was developed with support from
the national Dutch Research Organization NWO
and the NLnet Foundation

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.x</td>
<td>January, 1999</td>
<td>Caspar Derksen and Jasper Potjer</td>
<td>obsolete now</td>
</tr>
<tr>
<td>2.0</td>
<td>August, 2001</td>
<td>Erik Verbruggen</td>
<td>revision, hyphen-convention</td>
</tr>
<tr>
<td>2.2</td>
<td>August, 2003</td>
<td>Paul Jones</td>
<td>Best-Only parsing</td>
</tr>
<tr>
<td>2.3</td>
<td>January, 2004</td>
<td>P. Beinema and C.H.A. Koster</td>
<td>system and manual revised</td>
</tr>
</tbody>
</table>
Contents

1 Introduction 3
  1.1 The AGFL system ........................................... 3
    1.1.1 About version 2.3 ........................................... 4
    1.1.2 Copyright and License Information .......................... 4
    1.1.3 Availability and Support .................................... 4
  1.2 Outline of this Manual ....................................... 4

2 Brief overview of AGFL 5
  2.1 Affix Grammars and AGFL ..................................... 5
    2.1.1 The first level ........................................... 5
    2.1.2 The second level ........................................... 6
    2.1.3 Combining the levels ..................................... 6
    2.1.4 Rule type ................................................ 6
  2.2 Affix Grammars over a Finite Lattice ........................ 7
    2.2.1 Powerset domains ......................................... 7
    2.2.2 Affix lattices ............................................. 7
    2.2.3 Flat lattices ............................................. 9
  2.3 AGFL-parsers ................................................. 9
  2.4 The lexicon ................................................ 10
  2.5 Hybrid parsing ............................................... 11
    2.5.1 Penalties ............................................... 11
    2.5.2 Syntactic probabilities .................................. 11
    2.5.3 Lexical probabilities ................................... 12
    2.5.4 Robustness mechanisms ................................... 12
  2.6 Parse trees and transduction .................................. 12

3 Preamble: descriptional formalism 13
  3.1 Notational conventions ...................................... 13
  3.2 Type fonts ................................................ 14
  3.3 Related formalisms .......................................... 14

4 The AGFL formalism 15
  4.1 Overall structure of AGFL's .................................. 15
  4.2 Grammars .................................................. 15
    4.2.1 The grammar interface .................................... 16
    4.2.2 Grammar body ............................................ 17
  4.3 Affix Rules ................................................ 17
  4.4 Specifications ............................................... 19
  4.5 Syntax rules ............................................... 19
    4.5.1 Rule heads .............................................. 20
    4.5.2 Rule bodies .............................................. 21
    4.5.3 Productions .............................................. 22
    4.5.4 Calls .................................................. 22
    4.5.5 Final groups ............................................ 23
    4.5.6 Options ................................................ 23
    4.5.7 Terminal symbols ...................................... 24
Chapter 1

Introduction

The AGFL system is a parser generator for research and applications in Natural Language Processing, based on the AGFL formalism [Koster, 1991] and available under the GNU General Public License\(^1\). The system was developed by the Dept. of Computer Science at the KUN (University of Nijmegen) with support from the national Dutch Research Organization NWO and the Stichting NLnet.

The AGFL formalism for the syntactical description of natural languages belongs to the family of two-level grammars: a context free grammar is augmented with set-valued features for expressing agreement between syntactic categories. The formalism is suited for describing morphological structure and syntactic structure, case distinction and agreement and finite semantics. Grammar rules may be extended with transduction parts, which specify the output. Grammars and lexica may be combined in a modular fashion. Penalties may be used to express syntactical preferences. Frequencies given in the lexicon and the syntax form the basis for hybrid parsing. Mechanisms are provided to handle open classes of words, which enables the construction of robust parsers.

This document describes the AGFL system. It serves as a user manual to the AGFL formalism, the compiler and the generated parsers. Other publications concerning AGFL are available at the AGFL website at [http://www.cs.kun.nl/agfl/](http://www.cs.kun.nl/agfl/).

1.1 The AGFL system

The AGFL system consists of the following components:

- the GEN parser-generator
- the LEXGEN lexicon system
- [the Grammar Workbench GWB]

The GEN parser-generator compiles grammars written in the AGFL formalism into compact and efficient parsers. These parsers are stand-alone executables, reading sentences from a file, or prompting the user for input. Output consists either of decorated parse-trees in one of several formats, or of strings as specified by the transduction.

The LEXGEN lexicon system makes it possible to connect large lexical databases to your grammars in an efficient way. The lexicon system is fully integrated with the parser-generator. When necessary, GEN automatically invokes the lexicon system.

It is our intention to embed the AGFL system into a Grammar Workbench, or GWB for short, an environment for the comfortable development of grammars in the AGFL formalism. Its intended purposes are:

- to serve as a graphical user interface to the parser generator and the parsers generated;
- [ to provide an incremental syntax check in reading a grammar ];
- [ to allow the user to navigate, inspect and modify a grammar, using a dedicated editor ];
- to perform consistency checks on the grammar;

\(^1\)For license-specific information, please refer to appendix A.
• to compute grammar properties;
• [ to perform profiling of a parser ];
• [ to allow incremental tracing of a parser’s execution ];
• [ to generate example sentences ];
• [ to assist in performing grammar transformations ].

Although most of these features have been present in earlier versions of the AGFL system, due to lack of implementation manpower the features between square brackets are currently not available. Volunteers?

1.1.1 About version 2.3

Version 2 of the AGFL system was developed with support from Stichting NLnet in the AGFL/GNU project, and brought under the GNU Public License in January 2002.

The present version 2.3 of AGFL differs in some points from the previous versions:

1. small syntactic differences were needed to make the handling of lexica simpler
2. optimization of positive memoization by tree folding and the best-only heuristic
3. different options and system behaviour.

The most striking improvement is the speed of determining an optimal analysis, allowing probabilistic parsing and making the system more practical for applications.

1.1.2 Copyright and License Information

The copyright on the AGFL system has been transferred from Dept. of Computer Science at the KUN (University of Nijmegen) to the Free Software Foundation. All components of the AGFL system are licensed under the GPL or the LGPL license. The runtime system (the files named \texttt{lib*rts.a}) is licensed under LGPL and the executables are licensed under the GPL. All source files also contain a reference to the appropriate license. The licenses can be found in appendix A, in the files \texttt{COPYING} and \texttt{COPYING.LIB} in the source package, and on \url{http://www.gnu.org/licenses/licenses.html}.

1.1.3 Availability and Support

The AGFL system is presently available for the following architectures:

• Sun-4 (Sparc/Solaris)
• PC (i386 and higher; MS-DOS/W95/NT)
• PC (i386 and higher; Linux)

You can obtain the AGFL system from the AGFL WWW-site at

\url{http://www.cs.kun.nl/agfl/}

or by anonymous FTP from

\url{ftp://ftp.cs.kun.nl/pub/agfl}

At the AGFL WWW-site, you can register yourself as a user. If you do so, you will be informed about new releases of the system and other relevant information. If you experience problems or have any questions, you can send e-mail to \texttt{agfl@cs.kun.nl}. Bug reports should be sent to the same address.

1.2 Outline of this Manual

Chapter 2 provides a short overview of AGFL. Chapter 4 provides the complete syntax of AGFL, annotated with informal semantics and pragmatic remarks. The CF formalism used to describe the AGFL formalism is itself described in chapter 3. Chapter 5 goes into details about the input (lexicalization process, input styles) and output (parse tree representations, transduction). Chapter 6 describes the installation and use of the AGFL system, as well as the use of the generated parsers.
Chapter 2

Brief overview of AGFL

In this chapter, we present a short introduction to the main features of the AGFL formalism. For additional introductory material we refer to the original article [Koster, 1991] and to the AGFL website http://www.cs.kun.nl/agfl. In the next chapter a more systematic and complete description of the AGFL formalism is presented.

Affix grammars can be seen as the formalization of a notion familiar to linguists, namely that of a Context Free (CF) grammar extended with features. Other formalizations of the same notion are to be found in attribute grammars and various more recent feature-based syntactic formalisms. The notation of AGFL is rather like that of PROLOG, because both have a common ancestor: The two-level grammars introduced in 1966 by Aad van Wijngaarden for the formal description of ALGOL 68 [Van Wijngaarden et al, 1972].

AGFL is intended as a practical formalism for describing the syntax of natural languages. The AGFL system generates parsers from grammars and lexica. It supports two kinds of applications:

- **linguistic applications**, where all analyses of a given (possibly ambiguous) utterance are to be found
- applications in Information Retrieval and Natural Language Front ends, where the most likely analysis has to be found for consecutive segments of a running text.

In order to express the notion of likeliness and to obtain robust parsing, a number of mechanisms (penalties and probabilities) have been provided.

2.1 Affix Grammars and AGFL

AGFL (for an explanation of the name, see 2.2.2) is a form of Affix grammars especially developed for describing the syntax of natural languages. Like other kinds of Affix Grammars, an AGFL has a first or lower level consisting of context-free syntax rules, rewriting nonterminals to terminals or to other nonterminals, and a second level which adds parameters (called affixes) to the first-level rules. The characteristic property of AGFL is that the affixes are finite set-valued.

2.1.1 The first level

A syntax rule without a second level is just a CF rule, as can be seen in the following example:

RULE sentence: subject, verb.

This rule states that the nonterminal sentence can be rewritten to a subject followed by a verb. It provides one production for the nonterminal sentence. The rule

RULE subject:
    personal pronoun; noun phrase.

gives two productions for subject. This particular rule has the rule type RULE, which is also the default (so the symbol RULE may be omitted). Other rule types are OPTION and CONDITION (see later). Names of nonterminals may be written in a mixture of small and capital letters, and they may include embedded spaces and underscores.
2.1.2 The second level

The second level is the level where the *affixes* live (also known as *attributes* or *features*). As an example, the affix rule

```
NUMBER:: plural | singular.
```

defines the **affix nonterminal** `NUMBER` as the union (|) of the **terminal affixes** `plural` and `singular`. Names of nonterminal affixes are written in capital letters, those of terminal affixes in small letters (and a few special characters). Embedded spaces are not allowed in affix names. An affix rule states that a nonterminal affix can be rewritten into the union of certain affix-terminals or affix-nonterminals. The **domain** of an affix nonterminal is a **finite set**, the union of all the terminal affixes to which it can be rewritten.

2.1.3 Combining the levels

The two levels are combined by using affixes as parameters to the nonterminals in the syntax rules. According to the **consistent substitution rule**, all occurrences of an affix within one rule obtain the same value. This implies computing the largest intersection as a means for the **unification** of affix values. As an example, the rule

```
RULE sentence: subject (NUMBER), verb (NUMBER).
```

expresses the *agreement* in number between subject and verb.

The affix mechanism is a powerful tool to make a large CF grammar more compact, while at the same time allowing for a linguistically motivated way of describing. We will show this by the following, still very small example.

```
GRAMMAR cfg.
ROOT sentence.

RULE sentence: subject_plural, verb_plural.
RULE sentence: subject_singular, verb_singular.
RULE subject_plural: "we".
RULE subject_singular: "I".
RULE subject_plural: "you".
RULE subject_singular: "you".
RULE verb_plural: "walk".
RULE verb_singular: "walk".
```

Notice the bureaucratic heading specifying grammar name and root. The type **RULE** is optional (see 2.1.4). This one-level (CF) grammar describes the same sentences as the following two-level grammar:

```
GRAMMAR agfl.
ROOT sentence.

RULE sentence: subject (NUMBER), verb (NUMBER).
RULE subject (plural): "we".
RULE subject (singular): "I".
RULE subject (NUMBER): "you".
RULE verb (NUMBER): "walk".
```

They both describe the sentences I walk, we walk, and you walk in which the affix **NUMBER** has the values `singular`, `plural` and `singular|plural`, respectively.

2.1.4 Rule type

A nonterminal symbol may be **productive** to a smaller or larger degree, in the sense that it *always*, *sometimes* or *never* generates terminal symbols. Syntax rules may be typed accordingly as **RULE**, **OPTION** or **CONDITION** (since the ones that never generate a symbol may enforce some condition on its affix values or *predicate* by failing or succeeding depending on those values).
2.2 Affix Grammars over a Finite Lattice

Affix Grammars over a Finite Lattice are a particularly simple form of Affix Grammars: in AGFL the second level consists of restricted context-free affix rules. Each affix-rule allows to rewrite a nonterminal affix into a union of one or more terminal or nonterminal affixes. The (finite) set of terminal affixes into which a nonterminal affix can be rewritten forms its domain.

2.2.1 Powerset domains

In AGFL the affix-rules are restricted to have a finite domain, because recursion is prohibited. This enables reliable consistency checks and the generation of efficient parsers. With this restriction, Affix Grammars describe only Context-Free languages. Without such a restriction, i.e. allowing the affix rules to be CF, all recursively enumerable languages can be defined [Sintzoff 1967]. Since AGFL is intended for the syntactic description of natural languages, CF power should be largely adequate [Gazdar]. The power of the formalism was limited on purpose, to keep grammar writers from abusing larger power to write a parser rather than a grammar.¹

A characteristic property of AGFL is that affixes are set-valued. The only operation allowed on affixes is set intersection (either implicit or indicated by guards). This means that the value of a nonterminal affix is determined by the intersection of the domains of the different instances. In the example above, the intersection for the value of the affix NUMBER is taken. There are two instances of the affix, namely one for subject and one for verb. For the sentence we walk the first instance of NUMBER has the set-value plural and the second instance, for walk has the set-value NUMBER (short for singular|plural). The intersection of these two sets is the singleton value plural, which will be the result.

Finite set-valued affixes can be used to model classical morphological and syntactical features, but also for expressing finite semantical distinctions (such as (in)animateness of nouns and verb subcategorization).

Another characteristic of AGFL is that it describes one parse with the set of all terminal affixes, the maximal intersection, where some other formalisms would describe a separate parse for each affix-value from the intersection. For example, the analysis of you walk with the above grammar would give one parse with the affixvalue NUMBER, and not one analysis with the value plural and another one with the value singular. To do this, AGFL uses the consistent substitution rule, which is similar to the principle of unification. In rewriting a rule, all occurrences of the same affix variable obtain the same value, which is the set of all terminal affixes for which rewriting succeeds.

Since each domain is a finite enumeration of terminal affixes, any affix variable may be replaced by one of its terminal productions. By doing this blow-up in a systematic way for all affix values, a context free grammar is generated, equivalent to the original AGFL. AGFLs are therefore not more expressive than Context-Free grammars, but they are much more compact than an equivalent CF and allow for a more efficient implementation.

2.2.2 Affix lattices

The possible values of an affix variable form a mathematical object called a power-set lattice. An example lattice for the affix defined by

\[
\text{PERSON} ::= \text{first} | \text{second} | \text{third}.
\]

is shown in figure 2.1.

The top element of the lattice can be seen as the union of all possibilities (the value may be first or second or third or any combination). As more information is obtained, the number of possibilities is narrowed down to a particular value, or even further to the bottom element which indicates inconsistency. In that case, no parsing is found.

By means of an additional affix rule

\[
\text{NONFIRST} ::= \text{second} | \text{third}.
\]

a name can be given to the values denoting an intermediate degree of specification like \{second,third\} (figure 2.2).² It is important to bear in mind that affixes do not necessarily need to obtain single terminal values. A nonterminal affix represents a set of values from its domain, yielding an elegant and efficient representation in case of affix ambiguity.

¹But a good FORTRAN programmer can write FORTRAN programs in any language.

²However, these intermediate levels do not require a name, they always exist as potential intersections.
Figure 2.1: Lattice of values for the affix PERSON

\[ \top = \{\text{first, second, third}\} = \text{PERSON} \]

\[ \{\text{first, second}\} \quad \{\text{first, third}\} \quad \{\text{second, third}\} \]

\[ \{\text{first}\} \quad \{\text{second}\} \quad \{\text{third}\} \]

\[ \bot \]

Figure 2.2: Lattice of values for the affix PERSON, with names for intermediate levels

\[ \top = \{\text{first, second, third}\} = \text{PERSON} \]

\[ \text{Nonthird} \quad \text{Nonsecond} \quad \text{Nonfirst} \]

\[ \{\text{first}\} \quad \{\text{second}\} \quad \{\text{third}\} \]

\[ \bot \]
Axes can be used to capture classical linguistic notions like case, time, modality, but they can also be used for other finite semantic distinctions, like the animate/inanimate feature for nouns and the absolute/comparative/superlative feature for adjectives, or for morphological distinctions.

### 2.2.3 Flat lattices

Besides the power-set domains, two flat domains may be used in AGFL. A flat domain is an affix domain of single values, covering only the individual affix terminals and their complete union as possible values, but no other subsets. In AGFL there are two predefined types of affixes that produce lattices without intermediate levels. These two predefined flat domains are the text affixes, which can have any textual value (denoted between quotes), and the integer affixes, which can have any integer value (denoted by numbers) (see figure 2.3).

![Flat lattice of integers](image)

**Figure 2.3: Flat lattice of integers**

Flat domains are to be used when the affix alternatives are mutually exclusive, and powerset domains when combinations can occur. Flat domains are particularly useful for enforcing (long distance) agreements, like that between a (phrasal) verb and a particle in the following example:

```
GRAMMAR flataff.
ROOT s.
X::TEXT.

s: verb form(X), particle(X);
   verb form(X), particle(X).

verb form("goes away"): "goes".
verb form("comes back"): "comes".

particle("goes away"): "away".
particle("comes back"): "back".
```

This grammar recognises the phrases
- goes away
- away goes
- comes back
- back comes
but no others.

The example can be used as a model for the description of phrasal verbs and even idiomatic phrases. Instead of the text-domain, the integer domain might have been used.

### 2.3 AGFL-parsers

The parsers generated by the AGFL-system are top-down recursive backup parsers [Koster, 1974, Meijer, 1986] implementing nondeterministic cancellation parsing [Nederhoff, 1993] using continuations. This parsing method is capable of dealing with (slightly restricted\(^3\)) left-recursive grammars. NOT IMPLE-

---

\(^3\)Rules that are cyclic on the first level are not allowed. A grammar is cyclic if a nonterminal can be rewritten to itself, e.g. `B:[A],B,[C].` allows for the rewriting `B:B.` and therefore it is cyclic.
MENTED IN THIS RELEASE It can produce, on demand, one or all parsings for ambiguous input. The user of the AGFL-system needs to have little knowledge about the mechanism of parsing, and does not need to be aware of transformations and optimizations applied to the grammar; the only thing that counts is the language described by the grammar. Especially when using Best-Only parsing, the generated parsers can be very fast.

Parsing takes place in three steps (phases):

- the prescan phase performs table initialization and robust lexicalization
- the parse phase, which includes the attribute computations
- the print phase for transducing and printing the best analysis (or -es).

When using Best-Only parsing, parsing is so fast that the prescan phase may take most of the time. When there are many analyses, the construction and printing of the parse forest may take much more time than the parsing itself.

In AGFL parsing, affix values are computed on the fly. This means that the value of an affix may be narrowed down during parsing. In this way, an efficient and fast parser is obtained, which computes the affix values economically. Furthermore this has the advantage that affix values may remain unspecified or partially unspecified, i.e. the parse contains the set of all possible terminal affixes.

The on-the-fly computation of affix values during left-to-right parsing can be illustrated by the following example for a fragment of Dutch:

```
GRAMMAR littledutch.
ROOT sentence.

RULE sentence: subject (NUMBER), verb (NUMBER).
RULE subject (NUMBER): "zij". # she/they
RULE verb (singular): "loopt". # walks
RULE verb (plural): "lopen". # walk
```

In the Dutch sentence zij loopt, the value of the affix of the verb loopt determines the value of the affix of the subject zij. It is not clear until the last word that the subject of the sub-clause is singular. Until that moment, the value of NUMBER of the subject zij remains the full set \{singular,plural\}. The rule in the grammar which controls the agreement between the subject and the verb, will accomplish that the number of zij will be singular. This is done by taking the intersection of both set-values, a variant of consistent substitution.

### 2.4 The lexicon

The AGFL system comes with a lexicon system capable of storing terminal symbols together with their part-of-speech (POS), containing entries looking like inverted grammar rules, e.g.

```
"provide" VERBI(with,intr)
"provided" VERBP(with,intr)
"provided" VERBV(with,intr)
"provides" VERBS(with,intr)
"providing" VERBG(with,intr)
"sell" VERBI(to,ditr)
"sells" VERBS(to,ditr)
"sold" VERBV(to,ditr)
```

The syntactic categories defining the part-of-speech (such as VERBI(PREP,TRAN) are called pre-terminals or lexicon nonterminals. Terminal symbols may also be introduced by pre-terminal rules in the grammar, but since these are lexicalized one-by-one during the prescan, it is more efficient to put them into the lexicon.

The lexicon is stored very compactly and access to it is very fast, making it an excellent tool for dealing with large vocabularies - in many cases preferable to the use of a separate tagger or morphological analyser.
2.5 Hybrid parsing

So far, AGFL was presented as a pure rule base system, describing a (usually infinite) set of sentences. In practical usage, such a system suffers from an amazingly high degree of ambiguity. Except for some linguistic applications, we are usually interested in finding the most likely analysis of a sentence rather than the collection of all its possible analyses. We now describe two mechanisms in AGFL that give a ranking to the various analyses that should help in taking the most likely analysis: penalties and probabilities. The same mechanisms play an important role in making AGFL parsers robust against incorrect or unforeseen input.

2.5.1 Penalties

A production may be marked with a penalty (which is a whole number), indicating that every parsing which involves this production should obtain this penalty. The penalty level of an analysis is the sum of its accumulated penalties. As an example, a noun phrase without determiner could be penalized, in order to avoid trivial NP’s:

```
RULE noun phrase:
  determiner, noun part;
$PENALTY(3), noun part.
```

Although the idea of penalties is intuitively appealing, it is hard to give them any formal basis. The main function of penalties is to act as tie breakers in the case of multiple analyses, but in case of a trade-off between various penalized constructions, it is hard to predict what will be the result.

We take the approach that the penalties are related to the negative logarithm of the probabilities of the analyses. In order to make penalties commensurate with probabilities, we interpret a penalty of one as a factor $\frac{1}{2}$ in probability. Thus, the previous rule may be motivated by the fact that we consider a noun phrase with a determiner 8 times as likely (or “acceptable”) than one without.

2.5.2 Syntactic probabilities

The second mechanism NOT IMPLEMENTED IN THIS RELEASE for expressing likelyhood concerns the rewriting probabilities of alternatives: Each alternative (production) of a rule can be marked with a frequency. The default frequency is one. The syntactic probability of the $i$-th alternative of a rule is its frequency divided by the sum of frequencies of all alternatives of the rule.

The frequencies may be given arbitrarily, like the penalties, but they could also be derived from a treebank, i.e. a syntactically analyzed corpus where each sentence has been disambiguated in context. Such treebanks are, at the moment, rather scarce and syntactically not very detailed. In theory, the frequency of an alternative should express the number of times that this alternative has been applied in generating some corpus. An example: The following rule expresses the intuitive idea that any number of adjectives may precede a noun, but that a low number of adjectives is much more likely than a higher number:

```
RULE noun part:
  [25] adjective, noun part;
  [75] noun.
```

In this example the frequencies nicely sum to 100, but this is of course not necessary. By this rule, the probability of having $n$ adjectives preceding a noun decreases geometrically with $n$:

\[
\begin{align*}
n = 0 & \quad 0.75 \\
n = 1 & \quad 0.25 \times 0.75 \\
n = 2 & \quad 0.25 \times 0.25 \times 0.75 \\
\cdots
\end{align*}
\]

The frequencies play an obvious role in generating text with given statistical properties, but they can also be used like the penalties to rank the analyses in order of decreasing probability. The syntactic probability of each alternative applied is transformed to a penalty (taking the logarithm and rounding to the nearest integer).
2.5.3 Lexical probabilities

One of the main sources of the perplexingly high ambiguity of say English sentences is the **lexical ambiguity**: the fact that one word may have more than one **lexical category** or Part-Of-Speech (POS). As an example, the famous shibboleth *time flies like an arrow* makes a play on threefold lexical ambiguity.

- *time* as noun or verb
- *flies* as noun or verb
- *like* as preposition or verb.

Luckily, not all these interpretations are equally likely: *time* occurs much more often as a noun than as a verb.

In *AGFL* every entry in the lexicon (word plus POS) can be given a **lexical frequency** (default is one), and a particular entry obtains a **lexical probability** which is the frequency of this entry divided by the sum of the frequencies of all entries for this word. In a non-ambiguous sentence, the lexical probabilities do not make any difference, but for a lexically ambiguous sentence they act as very effective tie breakers by preferring the most likely POS for every word.

2.5.4 Robustness mechanisms

*AGFL* comes with a number of mechanisms to achieve robust parsing. The general approach is to accept, besides the language to be recognized, also all possible forms of incorrect input and rank the analyses appropriately by means of the previous mechanisms. For this purpose, the following additional mechanisms can be used:

- the *commit operator* `!`, which prevents the execution of further alternatives from a rule if a preceding alternative has been successful. This operator may only be used within a **CONDITION**.

- wildcard matching, matching unknown words of the input against regular expressions, like

  ```
  RULE robust verbform:
  $PENALTY, $MATCH(".*ing");
  $PENALTY, $MATCH(".*ed").
  ```

Making a grammar robust is an art, but *AGFL* provides the right mechanisms.

2.6 Parse trees and transduction

An *agfl* parser may produce as output either

- a parse tree in a simple indented format, suitable for diagnostics
- a parse tree in labelled bracket form, suitable for many applications, or even
- a form of output which is completely controlled by the grammar.

This transduction mechanism is explained in 5, giving many examples.

\[4\text{also on the attachment of the PP like an arrow to the main verb or to the object. Unlike the first three, this is a matter of syntactic rather than lexical probability.}\]
Chapter 3

Preamble: descriptional formalism

This very short chapter is a preparatory chapter for the next chapter, introducing the notation in which the syntax of the AGFL formalism will be described. Readers familiar with W-grammar notation may skip it. Please note it does NOT describe the AGFL formalism itself – that happens in the next chapter.

3.1 Notational conventions

The syntax of AGFL will be described in the next two chapters by a CF grammar in the notation introduced by Van Wijngaarden [Van Wijngaarden et al, 1972]. This notation is very similar to that of AGFL itself, therefore we use different typefonts: sans serif for the syntax of AGFL and courier for examples of AGFL.

Each context-free production rule consists of a left-hand-side and a right-hand-side, separated by a colon :. A left hand side consists of a nonterminal. A right-hand-side consists of a possibly empty series of alternatives separated by semicolons ; and terminated with a period . An alternative consists of a list of members separated by commas. A member is either a nonterminal or a terminal enclosed in double quotes ".

The notation admits a number of shorthand conventions, expressed by suffixes with a general meaning. If \(<\text{nonterminal}>\) stands for a sequence of letters and hyphens which is the name of a nonterminal, we may apply the schema:

\[
<\text{nonterminal}>\text{-list}:
<\text{nonterminal}>;
<\text{nonterminal}>,,<\text{nonterminal}>\text{-list}.
\]

This allows us to use in our description the nonterminal member-list without having to introduce an explicit rule like

\[
\text{member-list}:
\text{member};
\text{member},,\text{member-list}.
\]

The other shorthand conventions are

\[
<\text{nonterminal}>\text{-sequence}:
<\text{nonterminal}>;
<\text{nonterminal}>,<\text{nonterminal}>\text{-sequence}.
\]

\[
<\text{nonterminal}>\text{-series}:
<\text{nonterminal}>;
<\text{nonterminal}>,,,<\text{nonterminal}>\text{-series}.
\]

\[
<\text{nonterminal}>\text{-option}:
<\text{nonterminal}>;
\]

\[
<\text{nonterminal}>\text{-pack}:
"(,<\text{nonterminal}>,")".
\]

Using the above notational conventions, we will describe the AGFL-formalism in the next chapter.
3.2 Type fonts
In the sequel, examples of AGFL will be given in teletype font, like

\texttt{RULE sentence:}
\begin{verbatim}
subject, verb, object.
\end{verbatim}

Examples of terminal production will be given in sans serif font, like

\texttt{time flies like an arrow}

3.3 Related formalisms
AGFL belongs to the family of Two-Level grammars, which includes the following:

- **van Wijngaarden grammars**
The two-level Van Wijngaarden grammars (aka \texttt{W grammars}) were introduced by Aad van Wijn-
gaarden in 1965 [van Wijngaarden, 1965] for the formal description of \texttt{ALGOL 68} [Van Wijngaarden et al, 1972]. They are a “pure” form of two-level grammars, without any admixture of functions, a mechanism for specification without any concessions to implementability. An informal introduction to \texttt{W}-
grammars can be found in [Cleaveland and Uzgalis, 1977].

- **Attribute grammars**; 
  Attribute grammars (\texttt{AtG}) were invented by Don Knuth, hand-in-hand with LR(1)-parsing, in the middle sixties [Knuth, 1968]. An overview of the state-of-the-art in Attribute Grammars can be found in [Alblas and Melichar, 1991].

- **Affix grammars**;
  Affix Grammars (\texttt{AG}) were invented in the early sixties by Lambert Meertens and Kees Koster [Meertens and Koster, 1962] and used in Computer Linguistics [Koster, 1965], as well as in the generation of music by computer (Meertens’ String Quartet received a special mention by the jury at the 1968 IFIP Congress in Edinburgh). They were later applied to programming languages and formally described by Koster [Koster, 1971].

- **Extended Affix grammars**
  Extended Affix Grammars (\texttt{EAG}), defined by David Watt [Watt, 1971], form a bridge between Affix Grammars and \texttt{W}-grammars. They have been used in both Informatics and Computer Linguistics. A survey of some recent research on EAG can be found in [Mejer, 1990].

- **Metamorphosis grammars**
  Defined by Colmerauer in the late sixties for linguistic purposes [Colmerauer, 1978], Metamorphosis Grammars have been the direct precursor of \texttt{PROLOG}.

A number of syntactic formalisms widely used in linguistics are closely related to two-level grammars: various kinds of feature-based and unification-based grammars (e.g. GPSG [Gazdar et al, 1985], Aug-
mented Transition Networks and DCG’s [Pereira and Warren, 1980]). The main difference with two-level grammars is that in those formalisms the metalevel is usually less explicitly grammatical and more like data structures in a programming language than is the case in two-level grammars.

The Compiler Description Language \texttt{CDL3} [Koster and Beney, 1991], used in implementing the \texttt{AGFL} system, is another form of Affix Grammars, using trees over integers and texts for a second level, but because it is intended as an (efficient) systems implementation language it requires the underlying CF grammars to be deterministic.
Chapter 4

The AGFL formalism

This chapter is the reference document for the syntax and the semantics of the AGFL formalism. This is heavy stuff. For its understanding previous knowledge of the descriptional formalism used (see §3) is desirable. This chapter provides important terminology and describes the constituents of grammars and grammar rules and their semantics in a Top-Down order. While this order is reasonable for readers already familiar with AGFL, it is not the best order for didactic purposes, for which an informal introduction (see §2) is advised.

An affix grammar [Koster, 1971] (such as AGFL) is a two-level grammar, consisting of

- an underlying CF grammar, defining CF productions (also known as grammar rules or just rules) for certain nonterminal symbols (aka nonterminals)
- a meta grammar, defining CF production rules (aka affix rules) for certain nonterminal affixes (aka affix nonterminals)
- an annotation of the rules of the underlying grammar with affixes serving as Xs parameter, and
- calls on certain operations embedded in the grammar.

\[
\{ \text{affixes} + \text{domains} \} \\
\{ \text{CFgrammar} + \text{operations} \}
\]

Affix Grammars over a Finite Lattice (AGFL) is a particular form of affix grammars in which the domains are finite and set-valued, and the operations are set intersection and union (called implicitly rather than explicitly).

4.1 Overall structure of AGFL’s

An AGFL grammar consists of a number of files (also called modules):

1. the main grammar, containing the root of the grammar
2. any sub grammars used by it
3. any lexicon files used by it.

A file containing a (sub)grammar has the extension .gra, a lexicon file has the extension .dat.

The AGFL system translates one grammar, the main grammar, together with all its sub-grammars and lexicon files, into a parser.

4.2 Grammars

A grammar has a grammar head identifying the module, an grammar interface describing its relations to other modules, and a grammar body defining the objects of that module.
The name of the grammar should be the same as the name of the file in which it is contained (apart from the extension `.gra`).

### 4.2.1 The grammar interface

The `uses part` of a grammar specifies the sub grammars which must included in it.

```plaintext
uses-part:
  "USES", module-name-list, ".".
```

The root statement in the main grammar indicates which nonterminal serves as root of the system. Only the main grammar should have a root-statement. If no root-statement is given, its first syntax-rule is taken as root. The root must be a nonterminal with arity zero, i.e. without parameters.

```plaintext
root-statement:
  "ROOT", nonterminal-name, ".".
```

The lexicon interface, which may stand just before the root of the grammar, specifies nonterminals which are realized by means of the lexicon rather than by syntax rules, together with their parameters and arity.

```plaintext
lexicon-interface:
  "LEXICON", module-name-list, defines-part.
```

The module-name-list of a lexicon interface lists the lexicon files which it accesses. The defines-part specifies the lexicon nonterminals, implemented by a lexicon rather than by syntax rules.

```plaintext
defines-part:
  "DEFINES", specification-list, ".".
```
All the above elements are shown in the following example:

```
GRAMMAR example. # name of the grammar
USES subj, verb. # grammars used

LEXICON adj, noun, adv, misc, verb # lexicon files
DEFINES
  ADJE(Grad), # lexicon nonterminals
  ADJE(Grad,Prep),
  ADJE_TO(Grad),
  ADVB,
  NOUN(NUMB),
  DIMENSION,
  VERBI(PREP,TRAN),
  VERBS(PREP,TRAN),
  VERBV(PREP,TRAN),
  VERBG(PREP,TRAN),
  VERBP(PREP,TRAN).

ROOT sentence. # root of the grammar
```

### 4.2.2 Grammar body

The body of a grammar consists of specifications and definitions for **objects** (nonterminal symbols and nonterminal affixes). The order of the specifications and definitions is irrelevant, apart from the fact that the order in which parses are found may depend on the order of definitions in the grammar.

```
grammar-body:
  statement-sequence.

statement:
  specification;
  definition.

definition:
  syntax-rule;
  affix-rule.
```

```
RULE subject (NUMBER). # specification
subject (NUMBER): pronoun (NUMBER). # syntax rule
NUMBER :: plural | singular. # affix rule
```

### 4.3 Affix Rules

Affix rules (or **meta rules**) can be seen as CF rules on the meta level, in a slightly different notation, using as delimiters :: instead of the single colon to separate the left- from the right-hand-side, and | instead of ; to separate alternatives. (There is in the meta-level of AGFLno equivalent of the comma composing alternatives). rather than single colon, comma and semicolon.

An affix rule serves to specify the **domain** of an affix nonterminal. This domain is either

- one of the predefined **flat lattices** INT and TEXT (see §2.2.2), or
- a **power-set lattice**, consisting of all subsets of a specified set of **affix terminals**.
[In each power-set-lattice, the number of distinct affix terminals is presently limited to 31, but we hope to drop this restriction.]

affix-rule:
  affix-rule-head, "::", affix-expression, ".".

affix-rule-head:
  affix-nonterminal-list.

Affixes with the same domain are called **affix synonyms**. They may be defined by separate rules, or together in one affix rule:

\[
\begin{align*}
\text{PREP, } P &:: \text{ from } | \text{ to } | \text{ by } | \text{ with } | \text{ of.} \\
\end{align*}
\]

is short for

\[
\begin{align*}
\text{PREP} &:: \text{ from } | \text{ to } | \text{ by } | \text{ with } | \text{ of.} \\
\text{P} &:: \text{ PREP.} \\
\end{align*}
\]

The right-hand-side of an affix rule is an **affix expression**; the same construct can occur as an actual or formal parameter (see §4.5.1).

affix-expression:
  affix-tuple;
  affix-term.

affix-term:
  affix-nonterminal;
  affix-terminal-set;
  predefined-affix;
  text-denotation;
  number.

affix-nonterminal:
  affix-name.

affix-terminal-set:
  affix-terminal;
  affix-terminal, " | ", affix-terminal-set.

predefined-affix:
  "INT";
  "TEXT".

An **affix terminal** is usually a name, written in small letters and some special characters (see section §4.8.1). For the predefined flat domains TEXT and INT it is a text-denotation or a number, respectively. In AGFL recursive affix rules are forbidden, but a hierarchy of affix rules can be described.
4.4 Specifications

Nonterminal symbols can have affixes as parameters. The number of parameters is called the **arity** of the nonterminal. Since a call of a nonterminal with a certain arity can refer only to the definitions for that nonterminal with that arity, a nonterminal is allowed to occur with different arities, each with their own definitions.

A **specification** for a nonterminal with a certain arity serves to establish

- the **type** of a nonterminal (e.g. `RULE`), which must be the same for all its arities, and
- the **domains** of the parameters for this arity, i.e. the set of values which the ith parameter can take on, specified by a nonterminal affix. In this sense, *affixes also act as types*.

**Specifications are optional.** If a nonterminal with a certain arity has no explicit specification, the domains of its parameters are implicitly specified by affix nonterminals which are the **most general unifier** of the affix domains with which the nonterminal with that arity is defined anywhere in the grammar. If there is no such affix nonterminal this is considered an error.

```
specification:
syntax-rule-head, ".".
```

A nonterminal may have more than one specification with different arities. **Specifications may be repeated** as long as they do not contradict one another. All definitions of a nonterminal with a given arity must have a type (of the nonterminal) and domains (of its parameters) agreeing with the specification.

A nonterminal with two different arities:

```
RULE verb (NUMBER, PERSON, TIME). # finite form
RULE verb (TIME). # nonfinite form
```

- the nonterminal `verb` is specified with arities 3 and 1
- since it is specified as a `RULE`, it may not produce empty
  (see §4.5.1)
- the domains of the parameters are specified by affix nonterminals
  `NUMBER`, `PERSON`, etc.

For each call of a nonterminal, it is checked whether the actual parameter domains conform to the specification of the parameters for that arity.

4.5 Syntax rules

A nonterminal of a certain arity is defined by one or more **syntax rules** with that name and arity.

```
syntax-rule:
left-hand-side, ":" , right-hand-side, ".".
```

Nonterminals with the same right-hand-side can be defined together, according to the following convention:

```
left-hand-side: syntax-rule-head-list.
right-hand-side: alternatives.
```

If there are more rules for one nonterminal with the same arity, their alternatives are combined in textual order.
4.5.1 Rule heads

syntax-rule-head:
syntax-rule-type,
nonterminal-name,
formal-parameter-pack-option.

syntax-rule-type:
"RULE"-option;
"OPTION";
"CONDITION".

CONDITION may be shortened to COND. Syntax rules are typed according to whether they are productive in rewriting:

- a RULE must always generate at least one terminal symbol
- an OPTION must be able to generate empty as well as a terminal symbol
- a CONDITION must never generate a terminal symbol (but it may generate empty).

The type RULE is optional (and assumed as default). The type OPTION is used for indicating rules that may produce empty, and therefore never fail (aka nonfalse rules).

An example of typed rules:

RULE postmodifiers:
    postmodifier, maybe postmodifiers.
OPTION maybe postmodifiers:
    postmodifier, maybe postmodifiers; .

The type CONDITION is typically used for expressing a predicate, a test or restriction on affix values, possibly combined with a penalty (see §5.1.7).

Especially in British english, the agreement rules between subject and verb clause are a bit hazy, because certain singular nouns may be interpreted as a collective name (constructio ad sententiam). This may be expressed by the following agreement predicate

TEST agrees (NUMBER, NUMBER):.
TEST agrees (singular, plural): $PENALTY.
TEST agrees (plural, singular): $PENALTY(2).

All definitions for a nonterminal must have the same type, independent of the arity.

The rule types introduce redundancy into the grammar, allowing the system to check the intentions of the grammar writer (expressed in the head of the syntax rule) against his/her deeds (expressed in the body of the syntax rule and in its calls).

formal-parameter:
parameter.

A formal parameter acts as a variable which is bound against the corresponding actual parameter (by set unification). Formal parameters and actual parameters (see §4.5.4) have the same syntactic richness. Everything that may be written as an actual parameter may also be written as a formal parameter.

parameter:
affix-variable;
affix-expression.
In **strict affix grammars**, all parameters are variables, and the domains of formal and actual parameters must be the same. In **extended affix grammars**, like EAG and AGFL affix expressions may occur at parameter positions, and the domain of the formal parameter may be larger or smaller than that of the actual parameter, as long as the intersection is not empty. Having an affix expression as formal parameter implies an *implicit guard* (which is said to be “sugared away”).

An example of this syntactic sugar:

```
RULE to be (NUMBER, second): "you".
```

is equivalent to the (strict) rule

```
RULE to be (NUMBER, PERSON):
{PERSON :: second}, "you".
```

An *affix variable* occurring in a syntax rule must obtain a nonempty value which is no larger than the domain of the corresponding affix-nonterminal. The optional *index* of an affix-variable serves to indicate different instances of an affix-nonterminal in one syntax-rule. Therefore affix-variables with a different index may have different values.

An example of different instances of one affix-nonterminal is:

```
RULE sentence:
  subject (NUMBER1), verb (NUMBER1), object (NUMBER2).
RULE subject (NUMBER) : pronoun (NUMBER).
```

The index is only used to distinguish one instance from another within one alternative, and it has no influence on the use (or name) of the parameter in the rest of the grammar.

### 4.5.2 Rule bodies

Alternatives in a rule may be separated either by a semicolon ; or by the **commit operator** !.

```
alternatives:
  alternative-series,
  commit-operator,
  alternatives;
  alternative-series.

commit-operator:
  "!".
```

During parsing, the alternatives of a rule will be tried in textual order by recursive backtrack. For each successful alternative the continuation of the parsing is performed. However, if one of the alternative preceding a commit-operator has succeeded, then the alternatives following the commit-operator will be skipped.

The commit operator may be used as a robustness device (preferring syntactically correct input but allowing controlled ungrammaticality)
RULE subject (NUMBER):
  pronoun (NUMBER, nom); # [he] walks right there
  NP (NUMBER) ! # [the man] looks at his
  pronoun (NUMBER, acc). # [them] did it

CASE :: nom | gen | dat | acc.

An alternative consists of a production with an optional transduction part.

alternative:
  production,
  transduction-part-option.

4.5.3 Productions

production:
  frequency-part-option,
  syntax-part.

frequency-part:
  "[", number, "]".

The optional frequency parts serve to indicate syntactic probabilities in the grammar. In case no frequency-part is given, a default frequency is taken: 2 for a empty-generating production, and 1 otherwise. This distinction is made in order to be able to generate [later] random examples of terminal productions without going too deeply into recursion.

The syntax part consists of a (possibly empty) list of members, optionally followed by a final group, or it consists of free order members.

syntax-part:
  member-list;
  member-list, ",", final-group;
  success-operator-option;
  free-order-part.

Groups may only appear at the end of a production \(^1\). An empty alternative may be explicitly marked with the success operator +.

member:
  call;
  option;
  guard;
  terminal-symbol.

4.5.4 Calls

Most members are likely to be be calls. A call is the invocation of a nonterminal with an optional list of actual parameters.

call:
  nonterminal-name, actual-parameter-list-pack-option.

This also includes calls of lexicon terminals and quasi-terminals (see §5.1.6).

\(^1\)Enclosed groups may no longer appear among the members. The unrestrained use of groups in non-final position leads to unclear and hard to read programs. They were mainly used to avoid the invention of suitable names for sub-rules. For the time being, they will be syntactically accepted but translated to new rules.
A (very crude) robust rule for recognizing nouns and guessing their number might be

\[
\text{RULE noun (NUMBER):} \\
\begin{align*}
\text{lexnoun (NUMBER);} \\
\text{\$PENALTY (4),} \\
\text{\{NUMBER : plur\};} \\
\text{\{NUMBER : sing\}.}
\end{align*}
\]

actual-parameter:
parameter.

Having an affix expression as actual parameter again implies a restrict guard which is “sugared away”.

The following two lines are equivalent

\[
\begin{align*}
to \text{ be (NUMBER, second)} \\
to \text{ be (NUMBER, PERSON), \{PERSON : second}\}
\end{align*}
\]

4.5.5 Final groups

A final group allows the last member of an alternative to be (recursively) a choice between alternatives.

\[
\text{final-group:} \\
\text{\{"", alternatives, "\"\}}.
\]

Example of an final group:

\[
\text{RULE noun group (NUMBER):} \\
\begin{align*}
noun (NUMBER), \\
\text{\{post adjective;} \\
\text{"of", np (NUMBER1)\}.}
\end{align*}
\]

4.5.6 Options

An option can be seen as a shorthand for an anonymous rule, whose body consists of the given alternative followed by an empty alternative.

\[
\text{option:} \\
\text{\{"", member-list, "\"\}}.
\]

The rule

\[
\text{RULE NP:} \\
\begin{align*}
determiner, \text{[adjectives]}, \text{noun.}
\end{align*}
\]

is short for something like

\[
\text{RULE NP:} \\
\begin{align*}
determiner, \text{Oqcxaaq, noun.} \\
\text{Oqcxaaq:} \\
\text{adjectives; .}
\end{align*}
\]

It would be more elegant to introduce explicitly a rule like

\[
\text{optional adjectives:} \\
\text{adjectives; .}
\]
4.5.7 Terminal symbols

A terminal symbol of the grammar (or terminal for short) is either a syntax terminal or a pre-terminal or a quasi-terminal.

\[
\text{terminal-symbol:} \\
\text{\hspace{1em} syntax-terminal;} \\
\text{\hspace{1em} lexicon-terminal;} \\
\text{\hspace{1em} pre-terminal.}
\]

A terminal is productive, except for some of the pre-terminals like \$PENALTY.

A pre terminal is a (suitably parameterized) call on a lexicon-nonterminal imported from one of the lexica used.

\[
\text{lexicon-terminal:} \\
\text{\hspace{1em} nonterminal-name, actual-parameter-list-pack-option.}
\]

A syntax terminal is a text denotation that is to be matched against (part of) the input. Usually the syntax terminals are defined in a separate lexicon (see §4.7), but they may also be defined in the grammar. However, matching of syntax terminals from the lexicon is much faster than matching syntax terminals from the grammar. Therefore, in a production situation it is advisable to avoid having syntax terminals in the grammar.

\[
\text{syntax-terminal:} \\
\text{\hspace{1em} text-denotation.}
\]

The following rule is a candidate for inclusion as three entries in the lexicon:

\[
\text{RULE perspron (plural): } \text{\"we\" ; \"they\" ; \"them\" .}
\]

A quasi terminal is a (suitably parameterized) call on a built-in function.

\[
\text{quasi-terminal:} \\
\text{\hspace{1em} special-name, actual-parameter-list-pack-option.}
\]

A quasi-terminal serves to specify a (possibly open) class of terminal symbols, whose structure is described by a regular expression given as (one of) its parameter(s) which must be matched against (part of) the input.

4.5.8 Guards

A guard serves to express the restriction of the value of one (or more) affix variables to a certain domain.

\[
\text{guard:} \\
\text{\hspace{1em} \{" , restriction-list, \" \" \} .}
\]

A restriction specifies that the value of the affix-variable is restricted by set-intersection to (a subset of) the set specified by the affix-expression. If this intersection is empty, the guard fails. An affix-expression occurring at a parameter position denotes an implicit restriction.

It is important to realize that restriction is not equality: a guard like \{PERSON :: first\} does not check whether PERSON has the value first, but whether its value includes first; if that is the case, its value is restricted to first.
4.5.9 Free order parts

A free-order part describes in a compact way a construct whose members may occur in any order.

free-order-part:
member, "&", member;
member, "&", free-order-part.

A free-order part recognizes all permutations of its members.

The following complement orders are possible for a Dutch sentence:

\[
\begin{align*}
\text{hij gaf} & \ [\text{het boek}] \ [\text{aan jou}] \ [\text{op woensdag}] \\
\text{he gave} & \ [\text{the book}] \ [\text{to you}] \ [\text{on Wednesday}]
\end{align*}
\]

These can be described by a syntax rule with a free order part as its right-hand-side:

\[
\text{RULE complements:} \\
\text{direct object NP & indirect object PP & time PP .}
\]

4.6 The Transduction part

The default output of an AGFL-parser is a parse tree, which can also be obtained in the form of a labelled bracket tree. However, any other form of output may be specified in the grammar in its stead, by giving a transduction for each syntax rule. The transduction part of a syntax rule specifies this alternative output (see §5.4 for examples).

transduction-part:
" / ", transduction-member-list-option.

The transduction part rewrites to the concatenation of the transductions of its transduction members. A transduction member may be a text denotation, which rewrites to its literal value, or a placeholder referring to an element in the preceding syntax part. The transduction of a placeholder will be its transduction as specified in the rule specifying this placeholder. If no transduction part is specified, the standard transduction will be used.

The standard transduction consists of the concatenation of the transductions of the members in the syntax part of the alternative, in that order.

The following rules have the same meaning:

\[
\begin{align*}
A & : B, C, D. \\
A & : B, C, D / B, C, D.
\end{align*}
\]

The empty transduction may be specified by means of an empty transduction part.

\[
A : B, C, D / .
\]

The transduction results of B, C and D disappear.
transduction-member:
  nonterminal-placeholder;
  quasi-terminal-placeholder;
  ax-placeholder;
  group-placeholder;
  option-placeholder;
  text-denotation.

The form of the placeholders and their transductions are discussed below.

**Placeholder indices**

An index \( n \) following a nonterminal placeholder is used to refer to the \( n^{th} \) instance of that nonterminal in the production; and similarly for the other kinds of placeholders. An index \( n \) used as a placeholder refers to the \( n^{th} \) member of the production.

**Nonterminal placeholders**

\[
\text{nonterminal-placeholder: nonterminal-name, index-option; index.}
\]

A nonterminal placeholder consists of the name of the corresponding nonterminal (without the parameters!). Its transduction consists of the transduction of the corresponding nonterminal in the syntax-part.

\[
\[
\text{RULE sent: subject(N,P), predicate(N,P)/} \\
\quad "[", \text{subject, ",", predicate "]".}
\]

**Affix placeholders**

\[
\text{affix-placeholder: affix-variable.}
\]

The transduction of an affix-placeholder consists of the value of the corresponding affix-variable in the syntax-rule.

\[
\[
\text{PART:: none; PREP.} \\
\text{RULE verbform (N, P, PART).} \\
\text{verbform (N, P, none):} \\
\quad \text{verb (N,P), preposition(PREP)/ verb, preposition.} \\
\text{verbform (N, P, PART):} \\
\quad \text{verb (N,P)/ verb, PART.}
\]

**Quasi-terminal placeholders**

\[
\text{quasi-terminal-placeholder: special-name, index-option.}
\]

As is the case with nonterminals, the index \( n \) of a special name is used to refer to the \( n^{th} \) instance of this special name in the production. The transduction of a quasi-terminal-placeholder is the character sequence in the input matched by that quasi-terminal. If it ended on a terminator character (see §5.1.3) the transduction will include a space at that position.

\[
\[
\text{RULE filter:} \\
\text{word;} \\
\$\text{MATCH(".*")/ "SIKPPED: ", $MATCH.}
\]
Group placeholders

The transduction of a group placeholder is the transduction of the corresponding group.

group-placeholder: "()".

\[
\text{sententia:}
\begin{align*}
\text{subject}(\text{NUMB}, \text{PERS}, \text{GENDER}), \\
(\text{nominal predicate}(\text{NUMB}, \text{PERS}, \text{GENDER}; \\
\text{verbal predicate}(\text{NUMB}, \text{PERS}, \text{GENDER}) / \\
\text{"SUBJ:'}, \text{subject}, \text{"PRED:'}, (), .
\end{align*}
\]

Option placeholders

The transduction of a option placeholder is the transduction of the corresponding option.

\[
\text{group-placeholder: "[]".}
\]

\[
\text{sentence:}
\begin{align*}
[\text{circumstance}], \text{subject}(\text{NUMB}, \text{PERS}), \\
\text{VOC clause}(\text{NUMB}, \text{PERS}), [\text{circumstance}] / \\
["", \text{subject}, ",", \text{VOC clause}, [1], [2], "]
\end{align*}
\]

Text denotation

A text denotation is transduced literally (as in the above examples).

4.7 Lexica

Linguistic applications generally require a parser with a large lexical database. While developing a grammar it may be perfectly adequate to include in it a small or even reasonably large lexicon in the form of grammar rules producing a choice between terminal symbols. But in case of applications with a larger vocabulary it is better to create separate modules for grammar and lexicon. Separating grammar and lexicon has the following advantages:

- faster parser generation;
- faster lexical analysis and parsing;
- separation of concerns by grammar modularization;
- better maintainability of grammar and lexicon.

4.7.1 Lexica

In the AGFL system a lexicon is a module, which can be used from a grammar or another lexicon and will then be automatically included in the parser generated.

\[
\text{lexicon:}
\begin{align*}
\text{lexicon-head-option,} \\
\text{lexicon-interface,} \\
\text{lexicon-body.}
\end{align*}
\]

\[
\text{lexicon-head:}
\begin{align*}
\text{"LEXICON"}, \\
\text{lexicon-name,} \\
\text{""}
\end{align*}
\]
A lexicon may use other lexica, but it may not include grammars.

```
LEXICON irrverb.
DEFINES
  RULE to be (NUMB,PERS),
  RULE to have (NUMB,PERS).

  NUMB :: sing; plur.
  PERS:: first; secnd; third.
```

A lexicon must contain specifications for the affixes and nonterminals occurring in its body. The module using it may (but need not) contain specifications for the objects imported.

4.7.2 Lexicon entries

A lexicon entry is like a (simple) grammar rule written backwards.

```
lexicon-entry:
  terminal-symbol, tab-character, frequency-part-option, left-hand-side.
```

The tab character indicated serves to separate the two sides of the rule in a uniform style, suitable for using grep and similar UNIX utilities. The inverted notation for lexicon rules also makes it easier to maintain lexica. Some utilities for merging and subtracting lexica are included with the AGFL system.

```
"am" to be(sing,first)
"are" to be(plur,first|third)
"are" to be(NUMB,secnd)
"is" to be(sing,third)
"have" to have(sing,first)
"have" to have(plur,first|third)
"have" to have(NUMB,secnd)
"has" to have(sing,third)
```

Notice that the second person is described separately, in order to cope with the fact that English does not distinguish the singular from the plural in the second person pronoun.

An example with explicit lexical frequencies (in the previous example the default value 1 was assumed ) reflects the fact that the words time and times prevalently occur as nouns rather than as verbforms.

```
"time" NNT1 2814
"time" VVI 6
"times" NNT2 345
"times" VVN 1
```
4.8 Lexical conventions

The conventions regarding the form of names, denotations, comments and pragmats are discussed in this section.

4.8.1 Names

The names are formed from letters and some additional characters, the ornaments. AGFL is case-sensitive, but the distinction between upper case and lower case letters has no particular semantics, apart from the following:

- affix-names are in capital letters
- affix-terminals are in small letters
- nonterminal names start with a small letter. They may contain embedded spaces
- module-names are also used as file-names, so they must obey the local restrictions on file names.

Names may also contain certain ornaments like $ and _

\[
\text{nonterminal-name:} \\
\hspace{1em} \text{identifier.}
\]

\[
\text{module-name, grammar-name, lexicon-name:} \\
\hspace{1em} \text{simple-identifier.}
\]

\[
\text{affix-name:} \\
\hspace{1em} \text{capital-identifier.}
\]

\[
\text{special-name:} \\
\hspace{1em} "$", \text{capital-identifier.}
\]

A name has to fit on a single line. Provided the host operating system supports this, characters with nonstandard codes (i.e. ASCII-codes above 127) will also be accepted as letters and can be used for non-Latin characters.

Identifiers

Identifiers may contain embedded spaces. Multiple embedded spaces will always be reduced to one space.

\[
\text{identifier:} \\
\hspace{1em} \text{simple-identifier;} \\
\hspace{2em} \text{simple-identifier, embedded-space, identifier.}
\]

\[
\text{simple-identifier:} \\
\hspace{1em} \text{letter, letment-sequence-option.}
\]

\[
\text{letment:} \\
\hspace{1em} \text{letter;} \\
\hspace{2em} \text{digit;} \\
\hspace{3em} \text{ornament.}
\]

\[
\text{letter:} \\
\hspace{1em} \text{capital-letter;} \\
\hspace{2em} \text{small-letter.}
\]

\[
\text{ornament:} \\
\hspace{1em} "$", \text{\`+\', \`-\', \`\?\', \`@\', \`^\', \`\~\', \`\_\'.}
\]

\[
\text{embedded-space:}
\]
4.8.2 Text denotations

A **text denotation** is a *nonempty* text enclosed between double quotes.

```
text-denotation:
  double-quote, character-sequence, double-quote.
```

Every character within a character-sequence denotes itself, except for certain **operators** and **escaped characters**. Within a text-denotation, a double quote should be denoted as ", the sequence \- denotes the literal hyphen and a backslash is written as \. In addition, the sequence \n denotes the end-of-line character, and \t denotes the tab character. Every other character following a backslash denotes the same character without the backslash.

4.8.3 Numbers

A **number** is denoted by a sequence of decimal digits.

```
number: digit-sequence.
```

There is no way to denote negative numbers (or to perform computations on numbers).

4.8.4 Comments

A comment is a part of the input which should not be translated by the system. A comment starts with the comment-symbol # and comprizes the rest of the input line. It can be used at every place where a linebreak can be inserted.

4.8.5 Pragmats

A **pragmat** is a comment whose meaning is known to the system and which influences the system’s behaviour (although it does not alter the input language). Pragmats may occur at any place among the statements and definitions in a module. Pragmats in the input of the parser generator influence its output and therefore to some extent the behaviour of generated parsers. Pragmats differ from command-line options in that they form part of the input and therefore do not have to be presented at every execution of the system.

```
pragmat:
  " |", directive-list, " |
```

```
directive:
  "segment mode"; "line mode"
  "word terminators", text-denotation;
  "invisible characters", text-denotation;
  "invisible", nonterminal-spec-list.
```

```
nonterminal-spec-list:
  nonterminal name, "/", arity.
```

```
arity:
  number.
```

Below is an example of a grammar written for segment parsing. Furthermore, two grammar rules will not show in the parse tree: the syntax rule with the name xmltags with 0 parameters and the syntax rule agreement predicate with 2 parameters.
| segment mode |
| invisible xmltags/0, agreement predicate/2 |
Chapter 5

Lexicalization and Transduction

In this chapter the lexicalization process and the transduction mechanism are described in more detail, with some (hopefully stimulating) examples.

Parsers generated from AGFL grammars are intended to be used in widely differing applications, in particular in:

- linguistic research, and
- Information Retrieval applications.

Most of the necessary flexibility is provided by the lexical and transduction mechanisms, and a better understanding of those mechanisms will make many little PERL scripts superfluous.

5.1 Lexicalization

By lexicalization we mean the process of analyzing the input into terminal symbols from the grammar and the lexicon. An attempt has been made to make this process so flexible and parametrizable that in most cases no preprocessing of the input is needed:

- No word segmenter (segmenting the input into words) is needed – using the hyphen convention (§5.2) the parser is quite capable of segmenting the unformatted input text into a trellis of (possibly composed) words
- No sentence segmenter (segmenting the input into sentences, while taking abbreviations into account) is needed – in line mode (§5.1.1) the parser itself can split the running text into sentences
- No tagger (assigning POS to words) is needed – the parser with its lexicon is the best tagger
- No separate component for named entity recognition is needed – the syntax of named entities can be described in the grammar, along with other lexical robustness measures.

In the lexicalization phase, the input is split into a trellis of words (§5.1.4) separated by terminator characters (§5.1.3). The lexical graph (§5.1.5) helps in visualizing the result of the lexicalization process.

The two main application areas of AGFL mentioned make different demands on the parsers, therefore two different styles of input are accepted: line mode (next section) and segment mode (§5.1.2).

5.1.1 Line mode

In traditional linguistic applications, e.g. the syntactic analysis of corpora, the input consists of sentences, and the goal of the parser is to find all (or the best $k$) analyses for each sentence. Penalties may be used to indicate a preference for certain analyses over others. In this case, the unit of analysis should be the (complete) sentence, which is indicated by including in the grammar the pragmat line mode.

The input should be presented in a file with one sentence per line. For each line of input, the parser attempts to find the best analyses covering it completely. In case the time for analysing a sentence is larger than the limit (a time break), the result consists in the best parsings found until now.

Line parsing can be effectuated by giving the pragmat
5.1.2 Segment parsing

In Information Retrieval applications, the raw input is not nicely segmented into sentences, and it is to be expected that in many cases the input is not according to the grammar, or contains unknown words. A simple segmenter is only a few lines of Perl, but there are so many complications (like non-standard abbreviations, missing or spurious punctuation marks and unreliable capitalization of words) that even the best segmentation programs achieve little more than 95% precision on raw newspaper prose. The best segmenter is in fact the parser – as long as the input is syntactically correct. In IR applications, we are interested in extracting the largest comprehensible segments in a robust way, which will in many cases not be complete sentences. Furthermore we usually want only one analysis, viz. the best we can achieve.

For this case the segment mode was introduced, which causes the parser to find the best among the longest segments covering a prefix of the input; if no segment is found, one word is skipped. After processing one segment, the parser starts again, on the rest of the input. A single newline in the input is ignored, so that segments may extend over more than one line. A multiple newline is seen as the start of a new segment, and ends the preceding segment. Now the robust recognition intended can be described by

| segment mode |
RULE root: segment.
RULE segment:
  $PENALTY(50), $MATCH (".*");
  $PENALTY(20), fragment;
  $PENALTY(10), noun phrase;
  sentence.

The penalties indicate the quality of the segment. Notice that the order of the productions for segment has been reversed with respect to the previous example: the suboptimal analyses are now found before the optimal ones, so that in case of a time break always at least one suboptimal analysis has been found - thus achieving robustness.

Of course, such a strictly left-to-right segmentation may not be optimal in comparison with a segmentation which may start at any point of the input, but this may be compensated for in the grammar.

5.1.3 Terminators

The word terminators are characters serving to terminate a word: the layout characters and certain other characters enumerated in the terminator string. The default is

' <TAB><NL><CR>,.?;:()!"'

The default may be changed by a pragmat.

Some of the terminators are also invisible characters, meaning that they will be ignored unless they are matched explicitly as (part of) a terminal).

5.1.4 Words

A lexical element may be

1. a word, a string of characters not containing a blank or a hyphen, e.g. "not"
2. a multi word, a string containing a number of words with blanks between them to denote white space, e.g. "software engineering"
3. a part word, a word with a hyphen - at its beginning or end or both.

Since at any position of the input, more than one of these may be applicable, the input is not considered as a sequence of lexical elements, but rather as a lexical graph.
5.1.5 The lexical graph

The result of the lexicalization is the **lexical graph** or **lexical trellis**, a graph whose nodes represent positions in the input and whose arcs are directed and marked with terminal symbols. Three types of terminal symbols may appear in the lexical graph:

1. **terminal symbols** from the grammar, e.g. "not"

2. **quasi terminals** from the grammar, e.g. the argument of $\$MATCH("[a-z]+")

3. **lexical terminals** from the lexica

The lexical graph of the input can be inspected by calling the generated parser with the option -G. An example of a lexical graph, slightly edited to improve the layout, is:

1 "Mary" $\$MATCH("[A-Z][-A-Za-z]*") => 6, "Mary" $\$MATCH(".*") => 6, "mary" NOUN(SING) => 6

6 "had" $\$MATCH(".*") => 10, "had" VERBV(FOR|AS, TRAV) => 10, "had"

   VERBV(NONE,TRAV) => 10, "had" VERBP(FOR|AS,TRAV) => 10, "had"

   VERBP(NONE,TRAV,OFF) => 10, "had" VERBP(NONE,TRAV,ON) => 10, "had"

   VERBP(NONE,TRAV.OUT) => 10, "had" VERBP(NONE,TRAV,OVER) => 10, "had"

   VERBP(NONE,TRAV,UP) => 10, "had" VERBP(NONE,TRAV,IN) => 10, "had"

   VERBP(NONE,TRAV,OFF) => 10, "had" VERBP(NONE,TRAV,ON) => 10, "had"

   VERBP(NONE,TRAV.OUT) => 10, "had" VERBP(NONE,TRAV,OVER) => 10, "had"

   VERBP(NONE,TRAV,UP) => 10, "had" VERBP(NONE,TRAV,IN) => 10, "had"

   VERBP(NONE,TRAV,OFF) => 10, "had" VERBP(NONE,TRAV,ON) => 10, "had"

   VERBP(NONE,TRAV.OUT) => 10, "had" VERBP(NONE,TRAV,OVER) => 10, "had"

   VERBP(NONE,TRAV,UP) => 10, "had" TO HAVE(numb,pers) => 10

10 "a little" ADVB => 19, "a" $\$MATCH(".*") => 12, "a" ART(SING) => 12

12 "little" $\$MATCH(".*") => 19, "little" ADJE(ABSO) => 19, "little"

   ADVB => 19

19 "lamb" $\$MATCH(".*") => 23, "lamb" NOUN(SING) => 23, "lamb"

   VERBI(NONE,TRAV) => 23

23 <EOS>

The example shows a rather high level of lexical ambiguity. Notice the many quasi-terminals ($\$MATCH$).

Example: matching word suffixes

For the grammar

RULE sent:

    "quererlo":
    "querer", lo.

RULE lo: "lo"; "-lo".

the lexical graph for the input "quererlo" shows that the arcs passing a terminator (=>) are distinguished from those not passing a terminator (->):

1 "quererlo" => 9, "querer" -> 7

7 "-lo" => 9

9 <EOS>

Only the latter will be matched against part-words (in this case "-lo").

In the examples, we have not shown the frequencies: every transition has also a frequency associated with it (default = 1).

5.1.6 Quasi-terminals

The **quasi terminals** are built-in rules for dealing with open classes of symbols and penalties. Their names start with a dollar sign and for the rest consist of capital letters, so that they are distinct from nonterminals, but they are used like nonterminals with a special semantics.

Most of the quasi-terminals are parameterized with a quoted regular expression (see §5.1.8) which is to be matched against (a prefix of) the input. A regular expression always matches characters up
to a terminator, unless it ends on a hyphen, in which case the **hyphen convention** (See §5.2) applies. A blank within the regular expression matches a terminator. A regular expression cannot match an empty text. The default regular expression (\.*\) will recognize any token in the input. The following quasi-terminals are available:

\$MATCH ("RE") match the RE against the input
\$SKIP ("RE") match the RTE against the input, but only if NO other symbol matches at this position
\$PENALTY increase the current penalty level by 1
\$PENALTY (INT) increase the current penalty level by that number

### 5.1.7 Penalties

A **penalty** in an alternative indicates that it is less probable or desireable. The parser will produce parsings in order of increasing penalty level, and when limiting the number of parses, the parsing(s) with the lowest number of penalties is (are) preferred.

```
RULE sentence:
  subject, transitive verb, object;  # [he hits] me
  subject, transitive verb, $PENALTY.  # [he hits]
RULE subject : "he"; "they";
  $PENALTY, object.  # [them] likes me
RULE object : "me"; "them".
RULE transitive verb: "hits"; "likes".
```

Penalties may be considered as as rough approximations to (the negative logarithm of) probabilities, and should be used with discretion.

### 5.1.8 Regular expression

The AGFL formalism uses a restricted form of **regular expressions** with a conventional greedy semantics, i.e. the longest possible match is taken, and it will result in at most one match.

```
regular-expression:
  character-expression,
  repetition-operator-option,
  concatenated-expression-option.

character-expression:
  character;
  "\" character.
  ";
  ";
  "[", character-range-sequence, "]";
  "[^", character-range-sequence, "]".

character-range:
  character;
  character, ",", character.
```

A single character in a regular expression matches itself, unless it is a character with a special meaning (an **operator**). The backslash is used as an escape character in order to denote an operator. A dot matches any character except a separator (as given in the terminator string, see §5.1.3). A space matches any separator. A **character range** matches any character in the range, unless the range is preceded with a ^. In the latter case, any character not in the range is matched.

```
repetition-operator:
  "*";
  "+";
  "?".
```
The repetition operators are used for specifying the number of occurrences of the preceding expression. An asterisk * denotes zero or more occurrences, a plus + denotes one or more occurrences, and a question mark ? denotes an optional occurrence.

ORDINAL DIGITS: `$MATCH("([0-9]*\d+st|2nd|3rd|4-90th)")`.

5.2 The hyphen convention

According to the hyphen convention, a part word ending (beginning) with a hyphen does not require the presence of a word terminator following (preceding) it. As an example, either of the sequences

- "querer", "-lo"
- "querer-", "lo"
- "querer-", "-lo"

will match the input “quererlo”, but it will not be matched by

"querer", "lo"

To escape from the hyphen convention, a minus sign at the start or at the end of a word has to be escaped by preceding it with a backslash.

5.2.1 Dealing with prefixes and suffixes

The example of “quererlo” has already demonstrated how to deal with suffixes. The grammar

RULE sent:

```
"lo", "quiero";
"querer", lo.
```

RULE lo: "lo"; "-lo".

does not distinguish between “quererlo” and “querer lo”

Example: negated adjectives

As an example of the recognition of prefixes, consider negated adjectives in English: the three forms “nonexistent”, “non-existent” and “non existent” are presumably equivalent, and the last two should be mapped onto the first. This is achieved by the grammar

```
possibly negated adj:

"non", adj / "non", adj;
"non-", adj / "non", adj;
adj.
```

adj: "existent"; "nonexistent".

which uses transduction (see §5.4) to glue the “non” to the front of the adjective. The following is a trace of the corresponding parser, with the options `-tG` in effect.

```plaintext
>> nonexistent
  1 "non-" -> 4, "nonexistent" => 12
  4 "existent" => 12
  12 <EOS>
nonexistent

nonexistent

>> non existent
  1 "non" => 5
  5 "existent" => 13
  13 <EOS>
nonexistent
```

Notice that the first case was ambiguous (easily mended by a commit in the first rule, by setting the number of analyses to one or by eliminating all adjectives starting with “non” from the lexicon).
Example: Composed nouns

Suppose we want to find out which nouns in the lexicon can be parsed as a compound noun consisting of two nouns. We make a lexicon file (§4.7) containing a copy of all nouns from the lexicon, assigning them to a new class N and adding a hyphen at the end of their representation, e.g.

"table-" N

and write a grammar with as root

word:
  N, word / N1, ",", word !
  N, noun / N, ",", noun!
  noun / !
  $MATCH("\.*") / " NOT A NOUN :", $MATCH.

The lexicon for this grammar is composed of all nouns from the original lexicon plus all Ns obtained from them. The parser (called with -tP 1 to demand transduction of only the first analysis) will recognize words composed of two or more nouns, which will be transduced one-per-line, with a hyphen to separate parts. For non-composed nouns an empty line is produced. Non-nouns give an error message.

By providing it with all nouns from the lexicon as input (one word per line) we will obtain a list of all composed nouns.

5.3 Different character sets

The input may be in different character sets: ASCII, extended ASCII or UNICODE, provided the AGFL system has been suitably installed.

5.3.1 Invisible characters

Some characters like quotes and brackets are hard to deal with in a sensible and robust way. There is a regular way to use them (e.g. as matching brackets around certain syntactical constructions) but they may also occur in an unpredictable way. In those cases they may be skipped explicitly (expressed in the grammar, imposing a suitable penalty).

Any characters marked as an invisible character, will be skipped automatically when not explicitly recognized by the parser. By default, the white space characters (space, tab, new line, carriage return) are invisible, but if needed they may be matched by a space in a regular expression (§5.1.8).

The following example shows how to extend the set of invisible characters.

|invisible characters " \t\n\r\\"\"\\"\"|

5.3.2 Character conversions

In richer character sets, certain characters may form a family of variants with different diacritical marks, like e, e, e, e. In languages like Dutch, these diacritical marks may mostly without problem be replaced by the corresponding letters without diacritics. But to include all diacritical variants of each word in the lexicon appears cumbersome. This particular problem can be solved by mapping all the versions of the letter e onto e itself, and so on, using a suitable alternative character list.

Such a list consists of one or more pairs of decimal numbers, one per line with a blank or tab between the numbers constituting a pair. Such a pair expresses that in the input each character whose code is equal to the first member of a pair should be replaced by that character whose code is the second member. Characters not mentioned in the list remain unchanged.

Example: character conversion

The following alternative character list (rather superfluously) describes the automatic conversion of lowercase characters into the corresponding uppercase ones.
Suppose that the list is called `altchars.dat`; the parser should then be invoked with `-A altchars.dat`.

5.4 Transduction and its many uses

The output from an AGFL parser can be in one of the following forms:

1. an indented parse tree (default)
2. no output (`-O` option)
3. a label-bracketed tree (`-b` option)
4. a transduced form (`-t` option).

In the last case, the form of the output is wholly determined by the transduction specified in the grammar (which is ignored in the other cases).

5.4.1 Example: Fancy output

Using transduction, the output does not have to look like a parse tree, but it may be tailored in great detail to other purposes. An example syntax-rule with a transduction-part, which transduces a beautiful man to (the NP: [ DET/Q-a MOD-beautiful HEAD-man ] has number = SINGULAR)

```
RULE NP (NUMBER):
   determiner (NUMBER) , adjective , noun (NUMBER) /
"(the NP: [ DET/Q- " , determiner, 
   "MOD- " , adjective, 
   "HEAD- " , NOUN, 
   " ] has number = " , NUMBER, " ) " .
```

DETERMINER (singular) : "a".
ADJECTIVE: "beautiful".
NOUN (singular) : "man".

5.4.2 Example: Filtering for unknown words

The transduction mechanism may be used to filter from a given text all words that do not appear in the lexicon. The parser generated from the following grammar can be used as such a filter. It makes use of a lexicon called `wordlex`.

```
GRAMMAR filter.
LEXICON wordlex DEFINES
   NOUN(RESULT),
   VERB(NUMBER,PERS),
   ADJE(RESULT),
   ADVB,
   ART(NUMBER),
   DET(NUMBER),
   PREP,
```

38
POSSPRON,
PERSPRON(NUMB, PERS, CASE),
RELPRON(CASE).

ROOT input.

The parser will use segment mode, so that we can input any text in free format. We allow a rich set of terminators (which will be skipped explicitly).

| segment mode |
| word terminators " -+:,();.!?/%'" |

input: segment, (input! ).

segment:
known / ;
terminator / ;
$MATCH(".*")/ $MATCH.

A segment is either a known word or an unknown one; in the latter case it will be transduced literally. Terminators and known words will be skipped without transduction.

The axes needed by the lexicon are:

NUMB :: sing | plur.
PERS:: first | secnd | third.
CASE :: nom | gen | dat | acc.
GRAD :: abso | comp | supl.

The next rule will recognize all words that occur in the lexicon.

known:
NOUN(NUMB);
VERB(NUMB, PERS);
ADJE(GRAD);
ADVB;
ART(NUMB);
DET(NUMB);
PREP;
POSSPRON;
PERSPRON(NUMB, PERS, CASE);
RELPRON(CASE).

The parser generated outputs the transductions one-per-line, even if they are empty. In order to get rid of spurious empty lines the output can be filtered (under UNIX) by a suitable grep.

./filter -tP 1 | grep -v "\n"

5.4.3 Example: generating a lexicon

As a final example of the use of transduction, we describe a useful technique for generating (part of) the lexicon for an inflected language.

The “traditional” approach in lexical analysis for inflected languages is to construct a morphological grammar describing the structure of word forms in the language, and use a parser generated from this, either preceding the syntax analysis or as part of the syntax analysis. Often a tagger is used instead of a morphological parser,

We shall instead consider a technique where the lexicon is generated from a concise list of stems, declinations and conjugations, taking the nouns of the first and second conjugation in Latin as example. We make a list specifying for each noun the conjugation and one representative form. As output we want a sequence of entries for a lexicon file. For the rest, the example should speak for itself.
GRAMMAR substgen.

# INPUT PATTERNS:
# decl nom
# a  -a
# b  -us/um

# OUTPUT ENTRIES:
# LEXS(NUM,GENUS,CASUS)

NUM :: sg | pl.
GENUS :: fem | masc | ntr.
CASUS :: nom | voc | gen | dat | acc | abl.

ROOT substantivum.

substantivum:
    "a", radix, "a" /
    "",radix,"a"	LEXS(sg,fem,nom|voc|abl)\n",
    "",radix,"ae"	LEXS(sg,fem,gen|dat)\n",
    "",radix,"am"	LEXS(sg,fem,acc)\n",
    "",radix,"ae"	LEXS(pl,fem,NOMVOC)\n",
    "",radix,"arum"	LEXS(pl,fem,gen)\n",
    "",radix,"is"	LEXS(pl,fem,dat|abl)\n",
    "",radix,"as"	LEXS(pl,fem,acc)\n;
    "b", radix, "us" /
    "",radix,"us"	LEXS(sg,masc,nom)\n",
    "",radix,"e"	LEXS(sg,masc,voc)\n",
    "",radix,"i"	LEXS(sg,masc,gen)\n",
    "",radix,"o"	LEXS(sg,masc,dat|abl)\n",
    "",radix,"um"	LEXS(sg,masc,acc)\n",
    "",radix,"i"	LEXS(pl,masc,NOMVOC)\n",
    "",radix,"orum"	LEXS(pl,masc,gen)\n",
    "",radix,"is"	LEXS(pl,masc,dat|abl)\n",
    "",radix,"os"	LEXS(pl,masc,acc)\n;
    "b", radix, "um" /
    "",radix,"um"	LEXS(sg,ntr,nom|acc|voc)\n",
    "",radix,"i"	LEXS(sg,ntr,gen)\n",
    "",radix,"o"	LEXS(sg,ntr,dat|abl)\n",
    "",radix,"a"	LEXS(pl,ntr,nom|acc|voc)\n",
    "",radix,"orum"	LEXS(pl,ntr,gen)\n",
    "",radix,"is"	LEXS(pl,ntr,dat|abl)\n;

words / "ERROR:", words. # to catch incorrect input

words: word, words; .
word: $MATCH(".*").
radix:$MATCH(".*-").

Latinists will note that this is a simplification (some words ending on ”-a” are masculine or even neuter), but the overall approach should be clear enough, and extending this example into a realistic lexicon generator should be straightforward.

In this way it is possible to generate e.g. hundred thousands of verb entries from a few thousand stem times. The sheer size of the generated lexicon presents no problem, since the lexicon system is up to it. And lexicon lookup is much faster than (morphological) parsing.

The generative approach described is prone to overgeneration. But as long as the spuriously generated forms can not be confused with other words this presents no problem. A few words are too irregular to be treated in this way and have to be dealt with separately.
Chapter 6

Installation and Usage

This chapter describes how to install and use the agfl system. The installation is platform specific, the usage is the same for all installations.

6.1 Installation

The agfl system can be installed in several ways. For a number of platforms, binary packages with pre-compiled executables are available. These packages can be found at ftp://ftp.cs.kun.nl/pub/agfl/. If there are no binary packages available for a platform, or if (re-) compilation if preferred, the source package can also be retrieved from the URL mentioned above.

A compiled version of agfl expects to be able to run some programs to generate an executable. The C compiler is used for this step. As such, the agfl system requires the presence of a C compiler during compilation.

6.1.1 Unix flavours

Compiling and installing agfl on Unix and alike systems is quite straight forward. Required are:

- a recent C and C++ compiler, preferably GCC;
- a standard shell (usually /bin/sh suffices);
- the GNU make program;
- tar and GNU’s gzip;
- GLib version 1.2.10 or later.

GLib is used with GNOME, and can be found at any GNOME ftp site. You might also check the GTK+ homepage for more information: http://www.gtk.org/.

The compiler part of the agfl system is written in CDL3. As first step, the CDL3 compiler should be built and installed. The source package for this compiler can be found at the same place as the agfl source packages.

To compile and install CDL3 and agfl, please read the README and INSTALL contained in the source packages.

The usage section below expects the programs, as installed by the make install step, to be in the user’s path.

6.1.2 MS Windows

Compiling agfl for MS Windows is harder than compiling agfl for a Unix flavor. Required are:

- MingW32, downloadable from several places on the Internet;
- the Win32 GLib port (http://user.sgic.fi/ tml/gimp/win32/);
- a program to extract the source tarball (the MingW32 port contains a tar and a gzip port).
The Cygwin suite might also be used instead of MingW32, but this is not tested. To compile AGFL, special makefiles are included. These files are called `Makefile.cygwin` for historical reasons. These makefiles need modifications in order to find various programs.

The compiler part of the AGFL system is written in CDL3. As first step, the CDL3 compiler should be built and installed. The source package for this compiler can be found at the same place as the AGFL source packages.

The AGFL parser generator uses the C preprocessor, the assembler and linker/loader to generate an executable. For MS Windows platforms, it is recommended to download the binary package, which includes the appropriate programs to do this.\(^1\) Copying the contents of the `agfl-binary` directory filled during compilation, to the `agfl` directory from the binary package is sufficient.

The binary package for MS Windows also contains a script called `setagfl.bat`. This script might need modifications to reflect the actual place where the AGFL package is installed. These modifications can be done using any text editor. To use AGFL, start a command prompt and run the `setagfl.bat` script once for each session, or put the contents in the `autoexec.bat` file.

## 6.2 Usage

### 6.2.1 Commandline Options

Both the parser generator `gen` and generated parsers have various commandline options. When invoking `gen` or a generated parser with the `-h` option, all possible commandline options will be shown.

### 6.2.2 A Simple Grammar

For the examples to work, AGFL should be installed and the programs should be in the search path. To verify this, the `gen` command can be invoked:

```
$ gen
```

This will result in the following output:

```
gen: compiler driver for agfl version 2.0
```

The dollar sign ("\$") is used as command prompt in the examples below. For MS Windows systems, this will be `C:\agfl` or something similar.

Writing the various text files can be done by using any text editor which can write plain ASCII text files.

To show the behaviour of the AGFL system and generated parsers, we will use a toy grammar that consists of two files. The grammar file `example.gra` looks as follows:

```
GRAMMAR example.

LEXICON simplelex DEFINES verb(NUMBER, PERSON).

ROOT toy.

NUMBER :: singular | plural.
PERSON :: first | second | third.

toy:
  subject (NUMBER, PERSON),
  verb (NUMBER, PERSON).

subject (singular, first): "I".
subject (NUMBER, second): "you".
subject (singular, third):
  "he";
  "she";
  "it".
```

\(^1\)The binary package contains a stripped-down version of the MingW32 suite: enough to be used by the AGFL parser generator, but not enough to compile the whole AGFL package.
subject (plural, first): "we".
subject (plural, third): "they".

**Important:** there is an intentional error in this file. The second parameter of the first subject rule should be first. This error is put in to show errors generated by the parser generator.

The `simplelex.dat` file looks like this:

```
"walk" verb (singular, first | second)
"walks" verb (singular, third)
"walk" verb (plural, PERSON)
```

After making these two files in the current directory, `GEN` can be invoked:

```
$ gen example
example.gra:12:25: error: affix terminal "frist" does not occur in any domain
stopping: 1 error found...
```

As expected, an error has been found in the file `example.gra`, line 12 at position 25. Correcting this error and restarting `GEN` should give no more error messages and will build an executable named `example` (or `example.exe` for MS Windows).

Parsing text with the generated parser is straight-forward. When the `example` program is started, it shows `>>`, the prompt:

```
$ example
>>
```

On this prompt, a text can be entered. When the enter-key is pressed, the text will be parsed and one or more parse trees will be shown:

```
$ example
>> I walk
toy
 subject(singular, first)
  "I"
 verb(singular, first)
  "walk"
```

The sentence “I walk” has just one parse tree. The sentence “you walk” results in two parse trees.
Bibliography


Index

actual
  parameter, 22
Affix
  grammars, 14
affix, 5, 6
  ambiguity, 7
  expression, 18
  grammar, 15
  integer, 9
  nonterminal, 6, 15
  rule, 6, 7, 15, 17
  set-valued, 7
  synonym, 18
  terminal, 6, 17, 18
  text, 9
  variable, 21
Affix grammar
  Extended, 14
affix grammar
  extended, 21
  strict, 21
AG, 14
agreement, 6
alternative, 13
  character, 37
ambiguity, 11
  affix, 7
  lexical, 12
application
  linguistic, 5, 32
applications
  Information Retrieval, 5
arity, 19
AtG, 14
attachment, 12
Attribute
  grammar, 14
attribute, 6

base
  rule, 11
blow-up, 7
body
  grammar, 15
bottom
  element, 7
bracket tree
  labelled, 25
break

time, 32
breaker
  tie, 11
call, 15, 22
cancellation
  parsing, 9
case
  lower, 29
  upper, 29
category
  lexical, 12
CDL3, 14
CF grammar
  underlying, 15
character
  alternative, 37
  escape, 35
  escaped, 30
  invisible, 33, 37
  range, 35
  set, 37
check
  consistency, 7
commit
  operator, 12, 21
consistency
  check, 7
consistent
  substitution, 10
consistent substitution rule, 6, 7
convention
  hyphen, 35, 36
default
  frequency, 22
denotation
  text, 30
diacritical
  mark, 37
domain, 6, 7, 17, 19
  flat, 9
EAG, 14
efficient
  parser, 7
element
  bottom, 7
top, 7
embedded
Appendix A

Licenses

The components of the AGFL system are licensed as follows:

- the programs (which reside in the `bin` directory) are subject to the GNU GENERAL PUBLIC LICENSE, which can be found in the source package in the file named COPYING, and in appendix A.1. If not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

- the libraries (residing in the `lib` directory) and the include files (residing in the `include` directory) are distributed under the GNU LIBRARY GENERAL PUBLIC LICENSE. You should have received a copy of the GNU Library General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. The Gnu Library General Public License can also be found in appendix A.2.

In addition, each source file has the appropriate license mentioned at the start.

A.1 GNU General Public License

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc. 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.
Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone’s free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”. ) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

(a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

(b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.
Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

(a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients’ exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the
Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and promoting the sharing and reuse of software generally.

**NO WARRANTY**

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
END OF TERMS AND CONDITIONS

A.2 GNU Lesser General Public License

Version 2.1, February 1999

Copyright ©1991, 1999 Free Software Foundation, Inc. 59 Temple Place – Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software—typically libraries—of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author’s reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the Lesser General Public License because it does Less to protect the user’s freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.
For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is less protective of the users’ freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a “work based on the library” and a “work that uses the library”. The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called “this License”). Each licensee is addressed as “you”.

A “library” means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables. The “Library”, below, refers to any such software library or work which has been distributed under these terms. A “work based on the Library” means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term “modification”.)

“Source code” for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

2. You may copy and distribute verbatim copies of the Library’s complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

(a) The modified work must itself be a software library.

(b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
(c) You must cause the whole of the work to be licensed at no charge to all third parties under
the terms of this License.

(d) If a facility in the modified Library refers to a function or a table of data to be supplied by an
application program that uses the facility, other than as an argument passed when the facility
is invoked, then you must make a good faith effort to ensure that, in the event an application
does not supply such function or table, the facility still operates, and performs whatever part
of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-
deﬁned independent of the application. Therefore, Subsection 2d requires that any application-
supplied function or table used by this function must be optional: if the application does not
supply it, the square root function must still compute square roots.)

These requirements apply to the modiﬁed work as a whole. If identiﬁable sections of that work are
not derived from the Library, and can be reasonably considered independent and separate works
in themselves, then this License, and its terms, do not apply to those sections when you distribute
them as separate works. But when you distribute the same sections as part of a whole which is
a work based on the Library, the distribution of the whole must be on the terms of this License,
whose permissions for other licensees extend to the entire whole, and thus to each and every part
regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written
entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or
collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with
a work based on the Library) on a volume of a storage or distribution medium does not bring the
other work under the scope of this License.

4. You may opt to apply the terms of the ordinary GNU General Public License instead of this License
to a given copy of the Library. To do this, you must alter all the notices that refer to this License,
so that they refer to the ordinary GNU General Public License, version 2, instead of to this License.
(If a newer version than version 2 of the ordinary GNU General Public License has appeared, then
you can specify that version instead if you wish.) Do not make any other change in these notices.
Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU
General Public License applies to all subsequent copies and derivative works made from that copy.
This option is useful when you wish to copy part of the code of the Library into a program that is
not a library.

5. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object
code or executable form under the terms of Sections 1 and 2 above provided that you accompany it
with the complete corresponding machine-readable source code, which must be distributed under
the terms of Sections 1 and 2 above on a medium customarily used for software interchange.
If distribution of object code is made by offering access to copy from a designated place, then
offering equivalent access to copy the source code from the same place satisﬁes the requirement to
distribute the source code, even though third parties are not compelled to copy the source along
with the object code.

6. A program that contains no derivative of any portion of the Library, but is designed to work with
the Library by being compiled or linked with it, is called a “work that uses the Library”. Such a
work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of
this License.

However, linking a “work that uses the Library” with the Library creates an executable that is a
derivative of the Library (because it contains portions of the Library), rather than a “work that
uses the library”. The executable is therefore covered by this License. Section 6 states terms for
distribution of such executables.

When a “work that uses the Library” uses material from a header file that is part of the Library,
the object code for the work may be a derivative work of the Library even though the source code
is not. Whether this is true is especially signiﬁcant if the work can be linked without the Library,
or if the work is itself a library. The threshold for this to be true is not precisely deﬁned by law.
If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

7. As an exception to the Sections above, you may also combine or link a “work that uses the Library” with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

(a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable “work that uses the Library”, as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

(b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user’s computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

(c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

(d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

(e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the “work that uses the Library” must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

8. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

(a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
(b) Give prominent notice with the combined library of the fact that part of it is a work based
on the Library, and explaining where to find the accompanying uncombined form of the same
work.

9. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly
provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or
distribute the Library is void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under this License will not have
their licenses terminated so long as such parties remain in full compliance.

10. You are not required to accept this License, since you have not signed it. However, nothing else
grants you permission to modify or distribute the Library or its derivative works. These actions
are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the
Library (or any work based on the Library), you indicate your acceptance of this License to do so,
and all its terms and conditions for copying, distributing or modifying the Library or works based
on it.

11. Each time you redistribute the Library (or any work based on the Library), the recipient auto-
matically receives a license from the original licensor to copy, distribute, link with or modify the
Library subject to these terms and conditions. You may not impose any further restrictions on the
recipients’ exercise of the rights granted herein. You are not responsible for enforcing compliance
by third parties with this License.

12. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason
(not limited to patent issues), conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not excuse you from the conditions
of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may not distribute the
Library at all. For example, if a patent license would not permit royalty-free redistribution of the
Library by all those who receive copies directly or indirectly through you, then the only way you
could satisfy both it and this License would be to refrain entirely from distribution of the Library.
If any portion of this section is held invalid or unenforceable under any particular circumstance,
the balance of the section is intended to apply, and the section as a whole is intended to apply in
other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right
claims or to contest validity of any such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system which is implemented by public license practices.
Many people have made generous contributions to the wide range of software distributed through
that system in reliance on consistent application of that system; it is up to the author/donor to
decide if he or she is willing to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest
of this License.

13. If the distribution and/or use of the Library is restricted in certain countries either by patents or by
copyrighted interfaces, the original copyright holder who places the Library under this License may
add an explicit geographical distribution limitation excluding those countries, so that distribution
is permitted only in or among countries not thus excluded. In such case, this License incorporates
the limitation as if written in the body of this License.

14. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public
License from time to time. Such new versions will be similar in spirit to the present version, but
may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number
of this License which applies to it and “any later version”, you have the option of following the
terms and conditions either of that version or of any later version published by the Free Software
Foundation. If the Library does not specify a license version number, you may choose any version
ever published by the Free Software Foundation.

59
15. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

16. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

17. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS