SICS Technical Report
T99:03
ISSN 1100-3154

Tagging and Morphological Processing in the svensk System

Fredrik Olsson
fredriko@sics.se

March 1998

Abstract: This thesis describes the work of providing separate morphological processing and part-of-speech tagging modules in the svensk system by integrating the Uppsala Chart Processor (UCP) and a Brill tagger into the system. svensk employs GATE (General Architecture for Text Engineering) as the platform in which the components are to be integrated. Two pre-processing modules, a tokeniser and a sentence splitter for Swedish, were developed in order to facilitate the preparation of the texts to be analysed by UCP and the Brill tagger. These four components were then integrated in GATE together with a newly developed viewer for displaying the results produced by UCP.

The thesis introduces the reader to the svensk project, the GATE system and its underlying parts, especially the database architecture which is based on the TIPSTER annotation model. Further, the issues in connection with the development and design of the tokeniser and the sentence splitter for Swedish are elaborated on. The mechanisms behind transformation-based error-driven learning methods as employed by the Brill tagger are introduced as well as the principles of chart processing in general and UCP in particular. The greater part of the thesis is devoted to the process of integrating the natural language (NL) modules in GATE using the Tcl/Tk application programmers interface (API) and a so-called loose coupling.

The results of the integration of the NL modules are very encouraging: it is possible to mix modules written in programming languages from completely different paradigms (in this case the languages are Common LISP, Perl and C) and to have them interact with each other, thus maintaining a high degree of reuse of algorithmical resources. However, the use of Tcl/Tk and the associated API for processing structurally relatively complex data, i.e. the output from UCP, is time consuming and considerably slows the processing in GATE.

Keywords: Reusing Swedish language engineering software, SVENSK

Abstract

This thesis describes the work of providing separate morphological processing and part-of-speech tagging modules in the svensk system by integrating the Uppsala Chart Processor (UCP) and a Brill tagger into the system. svensk employs GATE (General Architecture for Text Engineering) as the platform in which the components are to be integrated. Two pre-processing modules, a tokeniser and a sentence splitter for Swedish, were developed in order to facilitate the preparation of the texts to be analysed by UCP and the Brill tagger. These four components were then integrated in GATE together with a newly developed viewer for displaying the results produced by UCP.

The thesis introduces the reader to the svensk project, the GATE system and its underlying parts, especially the database architecture which is based on the TIPSTER annotation model. Further, the issues in connection with the development and design of the tokeniser and the sentence splitter for Swedish are elaborated on. The mechanisms behind transformation-based error-driven learning methods as employed by the Brill tagger are introduced as well as the principles of chart processing in general and UCP in particular. The greater part of the thesis is devoted to the process of integrating the natural language (NL) modules in GATE using the Tcl/Tk application programmers interface (API) and a so-called loose coupling.

The results of the integration of the NL modules are very encouraging: it is possible to mix modules written in programming languages from completely different paradigms (in this case the languages are Common LISP, Perl and C) and to have them interact with each other, thus maintaining a high degree of reuse of algorithmical resources. However, the use of Tcl/Tk and the associated API for processing structurally relatively complex data, i.e. the output from UCP, is time consuming and considerably slows the processing in GATE.

Acknowledgments

This work has been funded by the Swedish Institute of Computer Science (SICS) and carried out at the Department of Linguistics at Uppsala University.[1] I wish to thank my supervisors Anna Sågvall Hein, Björn Gambäck and Mikael Eriksson for moral and technical support as well as for giving me the opportunity to work with a thesis on an interesting topic. I also wish to thank Hamish Cunningham and Pete Rodgers for quick and invaluable help regarding GATE technicalities. Many thanks to Per Starbäck for setting up our local GATE system. Finally, thank you all who made this possible.

Fredrik Olsson
`fredriko@stp.ling.uu.se`

[1]SICS is a non-profit research foundation. The core of the research is financed by the Swedish National Board for Technical and Industrial Development (NUTEK), Telia AB, Ericsson, CelsiusTech Systems, Försvarets Materielverk (FMV), Statens Järnvägar (SJ), IBM Svenska, Hewlett-Packard Sweden and Sun Microsystems.

Contents

Chapter 1

Introduction

The aim of this work is two-fold: to integrate separate morphological processing and part-of-speech (pos) tagging modules in the svensk system by integrating the UCP and a Brill tagger, trained on Swedish texts, into the platform used and to evaluate the results of the integration. The work has been carried out at the Department of Linguistics, Uppsala University, in collaboration with the Swedish Institute of Computer Science (SICS) during the period April - November in 1997 as a part of the svensk project.

A Brill tagger constitutes the pos-tagging module while the Uppsala Chart Processor (UCP) is used as morphological analyser. The tagger is currently being trained on a Swedish corpus developed at the Department of Linguistics at the Uppsala University within the ETAP project.[1] The UCP has been used within several research projects, most recently as a part of a grammar checker for controlled Swedish in the SCANIA project [Sågvall Hein *et al* 1997]. Two additional natural language (NL) components, a tokeniser and a sentence splitter for Swedish, were developed to enable the integration of the Brill tagger and the UCP into the svensk system. The task of the tokeniser and the splitter is to pre-process texts to make them meet the requirements that the Brill tagger and the UCP pose on the input.[2]

svensk is intended as a tool-box containing natural language processing components and resources for Swedish, primarily aimed at teaching and research. Until the start of this thesis, a number of components had already been successfully integrated in the system by SICS, e.g. a two-level morphology for Swedish (SWETWOL) [Karlsson 1992]; a constraint grammar tagger for Swedish (SWECG) based on the same technique as the one for English [Karlsson *et al* 1995]; and a deep-level unification-based processor (DUP) utilizing a Swedish unification-based grammar [Gambäck 1997] and an LR parser [Samuelsson 1994]. svensk is based on the GATE (General Architecture for Text Engineering) framework which is being developed at the Department of Computer Science at the University of Sheffield. The GATE system in itself is language and domain independent, providing merely a platform in which NL programs may "talk" to each other using a standardised interface (a so called "wrapper") to a database. The strong modularity of GATE gives users the choice of working with supplied NL systems or to create their own, built from integrated modules in a point-and-click fashion. The system is also equipped with a set of "viewers", that is, programs that format and display the contents of the database in a

[1]Etablering och annotering av parallellkorpus för igenkänning av översättningsekvivalenser (Creating and annotating a parallel corpus for the recognition of translation equivalents). Information about ETAP can be found on the Internet at the address: stp.ling.uu.se/~corpora/etap/ .

[2]Henceforth, the tokeniser and the sentence splitter for Swedish are referred to as the tokeniser and the splitter, respectively, unless the context requires more elaborate expressions.

number of different ways depending on the nature of the data under consideration. Apart from the NL components dealt with in this thesis, a viewer called the `single_span_ambiguities`-viewer was implemented and integrated into GATE. The viewer allows for displaying tokens (or words) in the input that are assigned several morphological readings by UCP. It is designed to handle ambiguities in terms of entities in the input that may receive zero or more analyses from some NL program; the intention is that it should be general enough to be used for displaying data similar to the output from UCP produced by any NL program.

Some efforts related to GATE has been made both in the US and in Europe, e.g. the multi-modal Open Agent Architecture^{TM} (OAA) developed by SRI International [Cohen *et al* 1989, Moran *et al* 1997]; the Advanced Language Engineering Platform (ALEP), which is an initiative of the European Commission [Simpkins & Groenendijk 1994, Bredenkamp *et al* 1997]; and the German Verbmobil architecture [Bub & Schwinn 1996]. Although the latter architecture is primarily intended to fit the needs of a specific project, a range of NL modules stemming from different developers as well as from different paradigms of programming languages, were integrated in it. The similarity between these systems and GATE lies in the way the architectures have been designed, that is, they are all open-ended systems facilitating rapid incorporation of NL modules in existing, or as parts of new, systems.

1.1 Some general problems in developing NL systems

Apart from the vast area of unanswered questions and unsolved problems in the field of computational linguistics, language engineering faces some problems of its own (as discussed in [Gaizauskas *et al* 1996a]). Among these, two are of primary importance since they constrain how the field can develop and must therefore be acknowledged and addressed.

- There is no theory of language which is universally accepted, and there is no uncontested computational model of even a small part of any of the theories presented so far.
- To build an intelligent system that reproduces enough language processing capability to be useful, is a large-scale project, that, given the political and economic state of the world today, must rely on the efforts of many small groups rather than a few big ones. This implies that, without having the possibility of reusing algorithmic and data resources of other people's work, these small groups would have to "reinvent the wheel" repeatedly.

Some common problems in developing NL systems are introduced in the next few sections. The problems have all been attended to, one way or another, by the people at the University of Sheffield when designing and developing GATE. An outline of GATE's capabilities in relation to these problems and the requirements they pose on any NL system is to be found in Section 2.2.4.

1.1.1 Reuse of data and algorithmic resources

One of the main problems in research and development (R&D) of NL systems is how to reuse already existing resources of data and algorithms. Intuitively, reuse of data (e.g. corpora) is easier to achieve, one key reason for this being that integration and reuse of existing, algorithmical, components can be a major undertaking. There are repositories for both algorithms (e.g. *The Natural Language Software Registry* which is an initiative of the Association for Computational Linguistics (ACL)) and data (e.g. *The Oxford Text Archive* (OTA)). A survey of electronic corpora can be found in [Edwards 1993, chapter 10].

1.1.2 The toy problem syndrome

The approach to natural language taken within the field of artificial intelligence (AI) is sometimes called the "toy problem syndrome", referring to the trend in this field towards artificial, small-scale applications of the technology under development. Scaling up such a "toy" system domain to a real-world task, e.g. a commercial application, has often shown the technology used in the toy system to be unsuitable for the job. For example, in [Cunningham *et al* 1995, pages 4–5] the authors describe the failure of a large-scale Prolog grammar project that started in 1985. It turned out that the cause of the failure was the total lack of evaluation methods for parsing projects at that time. In other words, when creating a small, theoretically and practically sound system, one might overlook parameters that would make it impossible to enlarge it to a real-world system.

1.1.3 Evaluation of NL systems

A big problem in evaluating NL systems is to determine precisely what the criteria of success should be. Such criteria depend on, of course, for what the system in question is intended to be used. For instance, methods and criteria for evaluating machine translation (MT) systems have been, and are currently being, investigated (for an introduction to this topic, see [Hutchins & Somers 1992, chapter 9]), while the corresponding methods for information retrieval systems are fuzzier.

An introduction to evaluation concepts for various areas of NL systems is given in, for instance, [Hirschman *et al* 1996, Gambäck 1997]. Three kinds of evaluation can be distinguished, appropriate to three different goals. The first is *adequacy evaluation*, which is a determination of the fitness of a system for a purpose — will it do what is required and at what cost, etc. The second kind is *diagnostic evaluation*, which typically involves the production of a system performance profile and is often used by system developers. The third kind is *performance evaluation*, which is a measurement of system performance in one or more specific areas. It is used, for instance, to compare two alternative implementations of a technology, or successive generations of the same implementation.

Participants in the research programmes sponsored by the US government (DARPA), such as the TIPSTER programme [TIPSTER 1996], the Message Understanding Conference (MUC) and the Text Retrieval Conference (TREC) competitions, for example, build NL systems to perform well defined tasks on selected pieces of texts. However, this requires a lot of resources, economic as well as human, and is therefore not always possible to accomplish in a general research project. In MUC and TREC, human analysts are employed to produce correct answers for some set of previously unseen texts, and the systems run to produce machine output for those texts. The machine output is then evaluated (so called *comparative evaluation*) with respect to the output produced by the humans.

1.1.4 Summary of requirements

From the previous sections, one may extract a list of requirements an NL system should be able to meet in order to eliminate/reduce the problems introduced above. Such a system should then (see [Cunningham *et al* 1995]) be able to:

- support collaborative research, e.g. research groups working in the same project should be able to concentrate their efforts to different parts of it, knowing that combining the parts to a whole is not a problem;
- support 'plug-and-play' module interchangeability and reuse of resources, algorithmic as well as data. For instance, in a NL system, it should be easy to replace an existing parser with a new one without having to alter anything in the rest of the system;
- contribute to portability across problem domains and application areas as well as across programming languages and operating systems;
- support (comparative) evaluation at a reasonable cost, that is, the underlying platform should support incorporation of test-suites developed at other sites as well as support creation of such suites for distribution to other research groups. The system should also provide the users with tools for comparing test suites.

Software engineering issues are also more important as the language engineering systems grow bigger. Robustness, quality and efficiency at the software level are areas that must be addressed in the process of transferring LE technology from the research lab to the marketplace.

1.2 Outline of the thesis

This section describes the organisation of the rest of the thesis. Chapter 2 introduces the reader to the svensk project, its goal and future plans (in Section 2.1); the GATE platform, its parts and its relation to the list of requirements outlined above (in Section 2.2); and the TIPSTER annotation model, which is the base for storing information about texts in GATE (in Section 2.3).

Chapter 3 presents two pre-processing modules developed explicitly for this thesis: a tokeniser and a sentence splitter for Swedish. The tokeniser is dealt with in Section 3.1 where some problems in identifying tokens are introduced and exemplified together with an overview of the tokeniser and how it works as a stand alone program as well as in GATE. Section 3.2 presents the sentence splitter by first introducing some problems in finding sentence boundaries followed by an overview of the splitter in the same manner as was given for the tokeniser.

Chapter 4 focuses on the modules for part-of-speech tagging and morphological processing, that is, the Brill tagger for Swedish and the Uppsala Chart Processor (UCP) (Sections 4.1 and 4.2, respectively). The general method used by the tagger to infer rules from a corpora is introduced in Section 4.1.1 and the way it applies to pos tagging in Section 4.1.2. The tagger's interface as a stand alone program as well as with GATE is presented in Section 4.1.3 followed by its limitations in Section 4.1.4. The section about UCP starts off by introducing chart processing in general in Section 4.2.1 and, very briefly, how the morphological competence is realised by means of chart processing in UCP in Section 4.2.2. The interface and limitations are presented in Sections 4.2.3 and 4.2.4, respectively.

Chapter 5 contains the core of this thesis, that is, how the NL modules described in Chapters 3 and 4 were integrated in GATE. The chapter starts with a description of the different types of couplings available between NL modules and the GATE Document Manager (GDM) in Section 5.1 while Sections 5.2 and 5.3 introduce the two templates, `creole_config.tcl` and `moduleName.tcl`, used for creating so-called GATE wrappers. Section 5.4 presents the method used for integrating the NL components in GATE. The integration of the tokeniser is elaborated on in Section 5.5: it is the least complex wrapper implemented in this thesis and its structure is adopted in the wrappers created for the other NL modules. Section 5.6 shows the integration of the sentence splitter, followed by Section 5.7, in which the integration of the Brill tagger is elaborated on. The most complex wrapper is the one for UCP introduced in Section 5.8, the main difference between it and those for the other NL modules is the way it has to process the output produced by the NL module: the UCP wrapper has to read the output cumulatively until enough information has been collected (typically several lines at a time) before recording it in the GDM. This is not the case for the other wrappers. As will be described later on, GATE provides a set of viewers for formatting and displaying the data produced by the different NL modules. Since it turned out that none of the existing viewers were able to display data in the format produced by UCP, a new viewer was designed and implemented (Section 5.8.4).

Chapter 6 concludes the thesis by a discussion of the results, an estimation of the time spent in the various parts of the thesis and some pointers to possible future work.

There are four appendices included in this report. The first of which, Appendix A, contains some practical questions that arose during the process of integrating the NL modules in GATE together with the answers to those. Appendix B briefly introduces the parts of the Tcl scripting language necessary for the work performed in this thesis as well as some additional pointers to where the reader can gain more information on the subject. Appendix C contains a listing of the methods available in the GDM Tcl applications programmers interface (API). Finally, Appendix D contains some of the source code produced. Section D.1 lists the code for the simplest of the wrappers, that is, the tokeniser, while Appendix D.2 lists the code for the UCP wrapper. The intention of these listings is to let the reader compare the wrappers' structure as well as the procedures used in them to order get an idea of what a wrapper might look like, what procedures are needed and how the complexity of the input/output of a NL module affects the complexity of the wrapper. Section D.3 lists the source code for the new viewer, called the single span ambiguities-viewer.

Chapter 2

Background

This chapter introduces the reader to the svensk project in Section 2.1: the goal in Section 2.1.1, and achievements and future plans in Section 2.1.2. Section 2.2 presents the GATE system and its three constituting parts: Section 2.2.1 elaborates on the underlying document management component — the GATE Document Manager (GDM) — which is the most important part of GATE; the GATE Graphical Interface (GGI) is introduced in Section 2.2.2; and the Collection of REusable Objects for LE (CREOLE) in Section 2.2.3. The relation between GATE and the requirements outlined in Section 1.1.4 is elaborated on in Section 2.2.4. The TIPSTER annotation model, which is used in the GDM, is introduced in Section 2.3: Section 2.3.1 presents the notion of attributes, documents and collections used within the annotation model; Sections 2.3.2 and 2.3.3 introduce document annotations and standard document annotations, respectively. Finally, the difference between the TIPSTER architecture and the GDM is outlined in Section 2.3.4

2.1 The svensk Project

The svensk project [Eriksson & Gambäck 1997a, Eriksson & Gambäck 1997b, Olsson *et al* 1998] is carried out at SICS. The project is divided into three phases, the first of which covered the period from spring 1996 to the end of 1996, and the second from the beginning of 1997 to August 1997. The third phase is planned and funded for the period January 1998 to December 1999. svensk has been funded by the Swedish National Board for Industrial and Technical Development (Nutek) and SICS.

2.1.1 Goal of the svensk Project

The goal of the project[1] is to develop a *multi-purpose* language processing system for Swedish based, where possible, on existing components. The generality of the system arises from its adherence to general principles of language engineering, combined with tools which facilitate adaption to specific domains and applications. Each component is integrated into a language engineering development platform which, together with the definition of standard interfaces, ensures interoperability of major components. The system makes it possible to incorporate linguistic resources (such as lexica, tagged corpora, etc) provided by Swedish universities and

[1]As adopted from the project plan available on the Internet at: sics.se/humle/projects/svensk.html

research institutes, so that svensk can be seen as a computational linguist's toolkit for the Swedish language. The components can be tested for robustness and accuracy on test suites from a number of specific application domains. Documentation describes how users can adapt the system to their own domains and applications (such as dialogue systems and machine translation).

2.1.2 Achievements and future plans

When the first phase of the project was concluded at the end of 1996, the main deliverable was the GATE system together with the following, integrated, components:

- SWETWOL – a comprehensive morphology for Swedish [Karlsson 1992] based on the two-level morphology technique presented in [Koskenniemi 1983];
- SWECG – a constraint grammar for Swedish based on the same technique used in the one for English [Karlsson *et al* 1995].

Both SWETWOL and SWECG were originally developed by the Department of General Linguistics, University of Helsinki and then turned into a commercial product by Lingsoft Inc., Helsinki.

- DUP – a deep-level unification-based processor utilizing a Swedish unification-based grammar [Gambäck 1997] and an LR parser [Samuelsson 1994];
- DSP – a domain-specific parser [Sunnehall 1996];
- SWECG2CLE – a conversion module that converts SWECG tags into corresponding CLE (Core Language Engine) features which can serve as input to DUP.

DUP and DSP were previously developed at SICS for other projects; DUP together with Telia and SRI International for the Spoken Language Translator project (SLT) [Rayner *et al* 1993]; DSP as a part of svensk and the Olga project, in which a multi-modal system for information services was developed [Beskow *et al* 1997]. SWECG2CLE was developed explicitly for converting between data sets in svensk.

A technical specification of these components in svensk is available in [Eriksson 1997]. At the end of phase two, the main deliverable included the components introduced above as well as components supplied by Swedish academia.

2.2 The GATE System

GATE is short for *General Architecture for Text Engineering.* It is a system developed at the Department of Computer Science at the University of Sheffield, and funded by the U. K. Engineering and Physical Sciences Research Council (EPSRC) and the U. K. Department of Trade and Industry. The current release of GATE is version 1.5, see [Cunningham *et al* 1995, Cunningham *et al* 1996, Gaizauskas *et al* 1996a]. GATE does not adhere to a particular linguistic theory, but is rather an architecture and a development environment designed to fit the needs of researchers and application developers. It presents users with an environment in

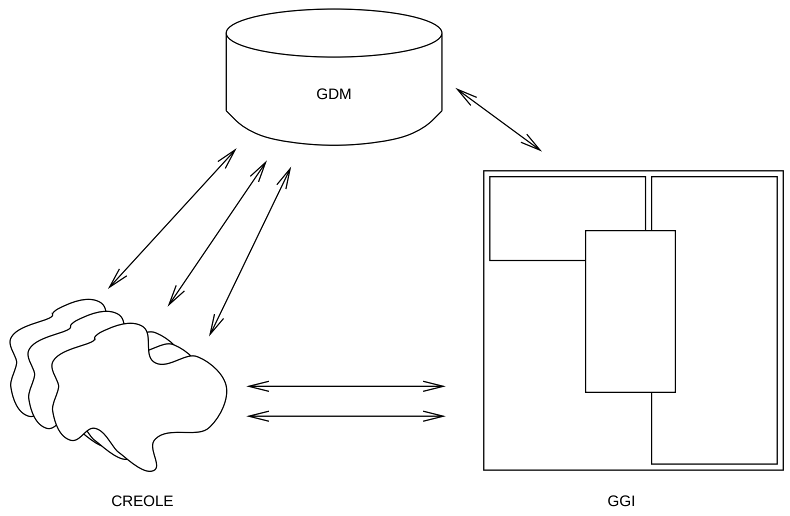


Figure 2.1: The three elements of GATE

which it is easy to use and integrate tools and databases, all accessible through a friendly user interface. As can be seen in Figure 2.1, the GATE consists of three major parts.

- *The GATE Document Manager* (GDM) in which texts and information about them can be stored and retrieved. The GDM is based on an object-oriented database schema called the TIPSTER architecture (see Section 2.3).
- *The GATE Graphical Interface* (GGI) for launching processing tools on data and viewing and evaluating the results.
- *A Collection of REusable Objects for Language Engineering* (CREOLE) which is a collection of wrappers for algorithmic and data resources that inter-operate with the database and the interface.

With a GATE distribution comes, amongst other things, a fully working information extraction system called the *Vanilla Information Extraction system* (VIE) [Humphreys *et al* 1996]. The VIE system is built from language engineering (LE) components such as pos-taggers, syntactic parsers and a discourse interpreter. It is an example of GATE's capabilities as an aid in building large and fully functional systems from existing components. Some of the components used in VIE are at the same time used in other systems, hence the tokeniser, the sentence splitter and the two pos-taggers constitute the components of two separate pos-tagging systems.

2.2.1 The GATE Document Manager — GDM

The GDM is based on the TIPSTER database architecture. It serves as a communication center for the components in an LE system in GATE since it stores all information such a system generates about the texts it processes. The GDM insulates the parts from each other and provides a uniform API for handling the data produced by the system.

The way information is handled by the TIPSTER database model (and thus, by the GDM) is elaborated on in Section 2.3.2. In short, information about a text is stored as annotations associated to sequences of byte offsets in the original text. Each annotation may have several attributes, which in turn may have zero or more values. In Table 2.1 (which is taken from [Cunningham *et al* 1995]), there is, for example, an annotation of the type token associated with the single span 0 and 5, where 0 is the starting byte offset and 5 is the ending one. In the original text, this corresponds to the word *Sarah.* To the token annotation is associated an attribute called pos (short for part of speech) with the value NP.

Text

```
Sarah savored the soup.
0...|5...|10..|15..|20
```

Annotations

Id	Type	Span		Attributes
		Start	End	
1	token	0	5	pos=NP
2	token	6	13	pos=VBD
3	token	14	17	pos=DT
4	token	18	22	pos=NN
5	token	22	23	
6	name	0	5	name_type=person
7	sentence	0	23	

Table 2.1: An example of what an annotation in GDM may look like.

The GDM is organised in such a way that source texts and the information about them are separated. The byte offsets are used as pointers into the original text in order to enable separate storage of the source text and the database holding information associated to it. The opposite approach would be to add information to the source text, as is the case with SGML. See [Cunningham *et al* 1995, appendix A] for a discussion of these two ways of storing information.

2.2.2 The GATE Graphical Interface — GGI

The GGI is a graphical launch-pad which enables interactive testing and building of NL systems within GATE. Various tasks that a user or a developer is likely to face are supported, such as integrating modules (see chapter 5), building systems, launching them and viewing the results [Gaizauskas *et al* 1996b]. The GGI is designed to provide a user with mechanisms to:

- create and access documents and collections and to run NL systems against single documents or entire collections of documents;
- view the results produced by the NL systems in an easy-to-understand fashion;
- alter the parameters of individual NL modules in order to make experiments easy to conduct;
- add modules to the CREOLE set and to create new NL systems from the set available.

The philosophy of the interface, which is implemented in Tcl/Tk, is to provide the user a rich set of tools. There are, for example, six generic viewers for displaying the results of NL systems, ranging from raw annotations to complex parse trees via non-ambiguous output from part-of-speech taggers.

2.2.3 The Collection of REusable Objects for LE — CREOLE

The CREOLE modules/objects[2] in the GATE system are to be thought of as interfaces to resources, either pure data or pure algorithmical or a mixture of both [Humphreys *et al* 1996]. A CREOLE module may be a wrapper around an already existing piece of NL software or it may be an entire NL program developed explicitly for GATE compliance. Either way, it is the CREOLE modules that perform the real work of analysing texts in a GATE NL system. Such a module provides a standardised API to the underlying resources, that is, the GGI and the GDM. There are three different ways by which a CREOLE module can communicate with the rest of GATE: by using a loose, dynamic or tight coupling, each of which is realised by either of two APIs, one in Tcl and one in C++ (see further chapter 5).

CREOLE modules encapsulate information about a program's pre-conditions (what type of annotations or attributes that must be present in the GDM for the program to run) and post-conditions (what data will be the result). The tasks for a CREOLE module involve setting up the environment for the NL program it implements or wraps up, e.g. processing arguments given by the user via the GGI, as well as retrieving information from the GDM, invoking the program, and taking care of the output produced, that is, format it and record it in the GDM.

2.2.4 Fulfillment of the requirements outlined in Section 1.1.4

The relation between the GATE system as such and the list of requirements presented in Section 1.1.4 is strong. GATE supports collaborative research in that it allows for different research groups to supply different parts of a larger NL system. Thus, each group can concentrate its efforts on a single task rather than on tasks that are out of range for their current research, yet necessary and crucial for the final result. Further, GATE supports reuse of resources, data as well as algorithms, since it provides for well defined APIs that are easy to use (see chapter 5). Once a NL module has been integrated in the system, it is very easy to combine it with already existing modules to form new NL systems. GATE contributes to the portability of components in the sense that software written in programming languages stemming from completely different paradigms can be mixed. However, the problem of portability across linguistic domains and hardware platforms must still be addressed on component level rather than by GATE as a system, for example, a parser written in assembler will not be easier to port to a new platform just because it is integrated in GATE. Finally, the last point in the list in Section 1.1.4 concerns (comparative) evaluation of NL systems. In [Cunningham *et al* 1996], the authors say:

> "Licensing restrictions preclude the distribution of MUC scoring tools with GATE, but Sheffield may arrange for evaluation of LE components by other sites. In this way, GATE/VIE will support comparative evaluation of LE components at a lower cost than the ARPA programmes..."

Hence, the GATE system seems to satisfy the requirement of supporting evaluation at a reasonable cost.

[2]The terms CREOLE module and CREOLE object are used interchangeably throughout this thesis unless otherwise stated.

2.3 The TIPSTER Architecture

The TIPSTER project is led by the Defense Advanced Research Projects Agency (DARPA) and it is funded by a number of US Government agencies (e.g. the Central Intelligence Agency, CIA, and the Department of Defense, DoD). The first phase of the project was from 1991 to 1994, the second phase from April 1994 to September 1996. Phase three started in October 1996 and it continues to build on phases one and two. The architecture is developed with US Government agencies with similar text handling requirements in mind. There are two main missions of the TIPSTER project, the first of which is to provide developers and users with an architecture that allows for document detection (information retrieval) in very large texts (several gigabytes). The second mission is to provide an environment for research in document detection and data extraction. The architecture, which is described in [TIPSTER 1996, Grishman *et al* 1997], contains four components: *detection*, *extraction*, *annotation* and *document management*. Detection is the technology which performs text retrieval, extraction is the technology for identifying entities and relations between entities in free text. The annotation component allows for information sharing between the former two components described while the document management component handles files. Currently, the GATE platform supports document management classes but not the information retrieval ones, see [Cunningham *et al* 1996, pages 16–18].

2.3.1 Attributes, documents and collections

The TIPSTER architecture is described in terms of a set of objects. An object class is characterised by a class name, a set of properties and a set of operations. Some of these objects and operations must be implemented by any system conforming to the architecture while some are optional. Since the classes in TIPSTER are well defined (see [Grishman *et al* 1997]), it is possible to define standards for annotations that people may want to use in the communication with (or between) TIPSTER conformant systems, but without requiring *all* such systems to generate these annotations.

An *attribute* is a list of feature-value pairs, where a feature may be an arbitrary string and the value may be any of a number of types in the TIPSTER architecture. There are a number of attribute-related classes described in [Grishman *et al* 1997], e.g. `AttributeValue` which have the operations `GetValue` and `TypeOf`. Attributes are common to several of the classes in the architecture.

A *document* is one of the most central units in TIPSTER. It serves three basic functions: it is the repository of information about a text and it is the atomic unit for building *collections* as well as of retrieval detection operations. Each document is part of one or more collections and it can be accessed only as a members of such. A collection is a compilation of documents. If documents are thought of as chapters, the collection is the book containing them. In general, collections are persistent in the TIPSTER architecture (documents are persistent by virtue of being member of a collection) and thus have names, although the architecture also provides for unnamed collections.

2.3.2 Document annotations

Annotations, along with attributes, provide the way by which information about a document is recorded and transmitted between the modules of a TIPSTER based system. An annotation keeps information about a portion of, or possibly an entire, document. The portion of a document is specified by a set of *spans*. Each span, in turn, consists of a pair of integers denoting the starting and ending points (in byte positions) in the raw document. The current span design is concerned with character-based documents, i.e. not documents including information such as pictures, video and audio. However, this is not a disadvantage in the work on this thesis since the svensk project currently involves texts only.

An annotation associates a *type* with a span. A type may be *token*, *sentence*, *paragraph* or *dateline*, for example. In addition to the types, an annotation may have associated with it one or more attributes. Possible attributes may take a single string as a value, or they may take annotations as values. An example of the former is the type-of-name attribute on a name annotation, which might take on such values as person, country, company, etc. An example of an attribute whose value is another annotation would be a co-reference pointer. Table 2.1 shows an example of annotations in GATE, which conform to the TIPSTER standard.

In most of the cases, an annotation is associated with a single span (and thus, a contiguous portion of the document). However, it is possible for an annotation to be associated with several spans at the same time, providing for a single attribute to describe a sequence of portions of the original text as in the case of discontinuous linguistic elements, e.g. the verb-particle pair in "*Hoppa* inte *på* tåget!" (literarely translated as "*Jump* not *on* the train" and meaning "Don't board the train"). The TIPSTER architecture does not, in its current version, allow annotations to modify the text in such a way that subsequent accesses to the text can see the modified version in place of the original one.

2.3.3 Standard document annotations

A number of standard *structural* and *linguistic* annotation types are defined in TIPSTER. The structural types can be used to distinguish between text, tables and graphics, e.g. *TextSegment* and *GraphicsSegment*. Text segments may consist of one or more languages and information about this can be recorded by annotations of the type *MonolingualTextSegment*, which, in turn, have two attributes: *Language* and *CharacterSet*. Another aspect of structural information is the *header* and *body* annotation types. A document body may be divided into paragraphs that may be divided into sentences which, in turn, may be divided in tokens. According to [Grishman *et al* 1997, page 23], the capability to annotate sentences and tokens will be obligatory for a TIPSTER system.

The names and co-reference tagging as defined for the MUC-6 competition are considered as standard linguistic annotation types. Similarly, the Penn TreeBank part-of-speech tags considered a standard for English. All three linguistic annotation types mentioned here are optional.

2.3.4 Differences between the GDM and TIPSTER

GATE currently supports the document management subset of the TIPSTER specification. The information retrieval classes are not available (see [Cunningham *et al* 1996, pages 16–18]). There are GATE APIs in the C++ and Tcl programming languages while the original TIPSTER model is defined in C (the C language header file is available in [Grishman *et al* 1997, Appendix C]). The GATE C++ API differs from TIPSTER's C API in style but not in functionality.

Chapter 3

A tokeniser and a sentence splitter for Swedish

This chapter presents two pre-processing modules developed explicitly for this thesis: a tokeniser and a sentence splitter for Swedish, both of which are based on modules that come with the VIE system [Humphreys *et al* 1996, chapters 3 and 4]. In the following, the terms tokeniser and sentence splitter refer to the ones developed for Swedish, unless otherwise stated. The tokeniser is dealt with in Section 3.1 where some problems of identifying tokens are introduced and exemplified (in Section 3.1.1), together with an overview of the tokeniser (in Section 3.1.2); an explanation of how it works as a stand alone program as well as in GATE (in Section 3.1.3); and its limitations (in Section 3.1.4). Section 3.2 presents the sentence splitter starting with an introduction of some problems in finding sentence boundaries (in Section 3.2.1), followed by an overview of the splitter (in Section 3.2.2), its interface as a stand alone program and towards GATE (in Section 3.2.3), its processing (in Section 3.2.4), and finally, its limitations (in Section 3.2.5).

3.1 A tokeniser for Swedish

The tokeniser is a fairly naïve approach to the problem of tokenising raw text in that it uses mostly *structural*, and only a limited amount of *linguistic*, information in its aim to recognise tokens. Structural information conveys information about individual characters and the way by which they can be combined to form tokens. Linguistic information says something about entities possibly larger than a single character. For instance, the use of linguistic information may promote the classification of a sequence of characters as a token even though the structure of the sequence will not. The tokeniser for Swedish is used as a pre-processing module for the sentence splitter for Swedish described in Section 3.2 and for the Uppsala Chart Processor (UCP) in Section 4.2.

3.1.1 Problems of tokenisation

The problem of tokenisation is often left aside even though tokens are the basic items in many text processing systems (see e.g. [Karttunen *et al* 1996, Guo 1997, Habert *et al* 1998]). Tokenisation is far from trivial, but still, when it comes to western languages little research has been done in the area. Some, but not all, of the tokens may be described and identified without the use of linguistic knowledge. In [Grefenstette & Tapanainen 1994], the authors consider the tokenisation process as selectively passing the text through a set of modular filters. Their method manages to identify 98.35% of the sentences in the Brown corpus using only non-linguistic knowledge for the identification of tokens.

Numbers (e.g. *123.45*), dates (e.g. *97-10-01*) and acronyms (e.g. *AT&T*) are examples of tokens that may contain ambiguous punctuation. An ambiguous punctuation mark is one which causes trouble in deciding whether the token it is attached to is to be considered as the ending token of a sentence or not. As a step towards disambiguating punctuation marks in tokens, some questions are in place:

1. What characters are to be considered as punctuation marks?
2. In what classes of tokens can the language under consideration be divided?
3. How can tokens from these classes be recognised?

(1) It may be argued that any character in the ISO 8859-1 character set (also known as Latin 1), except for the non-readable control character sequences, alpha-numeric characters and white space characters, are to be considered as punctuation marks in Swedish. This leaves us with characters such as `.!?"#$%&'()*+,/:;<=>?@[\]^_‘{|}~ }`

(2) Two classes of tokens can be identified: the first one consists of tokens which do not have punctuation marks in them, while the second one consists of the tokens that do. The latter class is of primary interest and it can be further divided in tokens that contain ambiguous punctuation marks and those that do not.

(3) The three most frequent sentence delimiters in Swedish are the exclamation mark (!), the question mark (?), and the period (.). The period is by far the delimiter that causes the most problems in terms of ambiguous tokens. To resolve such ambiguities, there are, at least, two different approaches, either using structural information or using linguistic knowledge. The latter may involve a lexicon and a morphological analyser, while the former involves ways of expressing the structure of the strings constituting tokens, e.g. by using regular expressions.

3.1.2 Overview

The tokeniser for Swedish identifies and returns tokens, possibly together with the associated pair of byte offsets, in the input text. The design and implementation of the tokeniser was done with its integration in GATE in mind. However, it is also possible to use it as a stand-alone program. It is written in the C programming language with the aid of `flex` [Levine *et al* 1995], which is a tool for generating finite state lexical scanners. The `flex` tool allows the developer to describe string patterns in terms of regular expressions that it then translates to C code.

The tokeniser uses primarily structural information to recognise tokens, but it is also equipped with a stop list containing frequent abbreviations. By means of this stop list it is able to distinguish the cases of a token containing a period (.), and in which the period should be considered as a sentence delimiter from those cases where it should not. The tokeniser also recognises and signals consecutive newline characters in the input although they are not considered as tokens since it does not make much sense trying to squeeze any linguistic information out of them. Rather, the information about multiple newlines can be used to identify headings in the text as sentences, so this information is passed on to later processing steps for any component to use.[1] The regular expressions passed to `flex` define the following patterns to be tokens;

1. A contiguous character string in which each character is alpha-numeric or a hyphen, e.g. “Uppsala”, “orsak-verkan”, “60åringen”, “100års-jubileum”.
2. Any of the abbreviations specified in the stop list, e.g. “bl.a.”, “t.” and “ex.” from the abbreviation “t. ex.”.
3. Any other characters, except for white space characters, e.g. “.”, “?”, “!”.

When the tokeniser recognises one of the above patterns in the input text, it prints the string corresponding to the pattern together with its byte offsets.

The list of abbreviations mentioned above is obtained from a 300,000 word portion of the Stockholm-Umeå Corpus (SUC). After extraction, the abbreviations were sorted according to frequency, and all those which occurred more than once and had a period in them were picked out. In a subsequent step, all abbreviations containing white space characters were split into parts containing no white space and all those parts which did not have a period in them were removed. In this way, a stop list containing approximately 40 abbreviations was obtained. This may seem too small a number, which it probably is, but it is easy to re-incorporate an extended list into the tokeniser.

3.1.3 Interface

The information in the subsections about GATE pre- and post-conditions apply only when the tokeniser is invoked from within GATE. The subsections about input and output apply any time the tokeniser is invoked; as a stand alone program as well as via GATE.

Input

The tokeniser expects the input to be raw text in Swedish. Currently, it has no capabilities of coping with SGML annotated input (in contrast to the tokeniser in the VIE system [Humphreys *et al* 1996, chapter 2]). It may well be run on other languages than Swedish, but then the stop list of abbreviations will not apply (or, worse, it will apply to the wrong cases).

Output

Depending on the command line options given by the user, the tokeniser will be made to produce output in two different formats, where each line is one of (1) or (2):

1. `<sint> <eint> <token>`
   `multiNl <sint> <eint>`

[1]Headings in the input are perhaps better referred to as sentence fragments rather than as whole sentences, although this is currently not the case.

2. `<token>`

Where `<sint>` and `<eint>` are integers denoting the starting and ending byte offsets, respectively, for the token `<token>` or for the string, `multiNl`, which signals consecutive newline characters. Cases (1) and (2), above, are mutually exclusive. If the user decides to have the tokeniser suppress the byte offsets, the lines signaling consecutive newline characters are also suppressed. Note that the option of omitting the byte offsets and the `multiNL` lines from the output does not apply when the tokeniser is run in GATE since the byte offsets are needed to annotate documents in the GDM.

GATE pre-conditions

The tokeniser for Swedish requires no annotations to be present in the GDM to be able to run.

GATE post-conditions

The tokeniser produces `token` annotations with the attribute `tokenVal`, which takes as value the token string itself. It produces two document level attributes, `upptoken` and `language_swedish`.

3.1.4 Limitations

The tokeniser is domain independent but language dependent in that it uses a list of Swedish abbreviations. It does not try to "de-hyphenate" the input text, i.e. apply a pre-processing step with the aim of getting rid of all hyphens at the end of each line in order to re-join possibly split words. The presence of a well-defined "de-hyphenation" step is likely to increase the accuracy of the tokeniser.

3.2 A sentence splitter for Swedish

The sentence splitter for Swedish is, as in the case of the tokeniser described in Section 3.1, a simple pre-processing module intended for use within the GATE platform. Its primary task is to provide the Brill tagger described in Section 4.1 with sentences and annotations associated with them, but it may also be used to feed other components which take sentences as input.

3.2.1 Some problems in finding sentence boundaries

The problem of splitting a text into sentences is related to that of tokenisation as presented in Section 3.1.1. Finding and, if necessary, disambiguating sentence boundaries is a bit trickier than it might appear at first glance. Consider the following examples;

1. *"Stå inte så tätt isär!" röt Sgt. Verkmestar.*
2. *Lagerlöf blev utnämnd till fil. hedersdr. 1907. 1909 fick hon Nobelpriset.*

(1) is an example of a sentence within a sentence; how can the nesting of the sentence be discovered? (2) is an example of periods attached to the three last tokens of a sentence. How can it be decided which one of the periods constitute the end of the sentence? Possible approaches to solve problems like these may involve a system that uses rules (e.g. a hand-made or induced grammar), statistics (e.g. HMMs), or some sort of neural network. Common to all approaches is that they require one or more pre-processing modules such as a tokeniser, morphological analyser or a program that calculates word frequencies. For instance, in [Palmer & Hearst 1994] the authors present an approach towards disambiguating sentence boundaries using a lexicon with part-of-speech probabilities and a feed-forward neural network. The method is claimed to correctly label over 98.5% of the sentence boundaries in a corpus of more than 27,000 sentence-boundary marks.

The sentence splitter for Swedish implements a rule-based approach using a minimum of information since it is not geared towards any specific corpora (thus assuming a minimum of information to be present in the input). The splitter's performance has not been measured (again, this is due to the fact that it is not tuned to any particular corpora), but it is likely to improve if it can access more information, e.g. a SGML marked-up text. Currently, the information present is mostly structural. The amount of linguistic knowledge is small and implicitly provided by the tokeniser.

3.2.2 Overview

The sentence splitter is designed to be used with the tokeniser described in Section 3.1, but it may function with any tokeniser that meets the requirements the splitter poses on its input. The splitter is implemented in the Perl scripting language.

3.2.3 Interface

The information in the subsections about GATE pre- and post-conditions and parameters apply only when the sentence splitter is invoked within GATE. The subsections about input and output apply any time the splitter is invoked; as a stand alone program as well as via GATE.

Input

The sentence splitter takes as input a sequence of triples separated by new line characters. A triple consists of a token preceded by its byte offsets or by a `multiNl` followed by its byte offsets.

```
<sint> <eint> <token>
multiNl <sint> <eint>
```

where `<sint>` and `<eint>` are integers denoting the starting and ending byte offsets, respectively, for the token `<token>` and for the string representing consecutive newline characters, `multiNl`, generated by the tokeniser (see Section 3.1.3).

Output

Depending on the command line options given by the user when the sentence splitter was invoked, it produces output on two different formats, where each line is one of (1) or (2):

1. <`sint`> <`eint`> <`sentence`>
2. <`sentence`>

where `<sint>` and `<eint>` are integers denoting the starting and ending byte offsets, respectively, for the sentence `<sentence>`.

GATE pre-conditions

The sentence splitter requires the GDM to contain `token` annotations with associated `tokenVal` attributes to run. The value of a `tokenVal` is the string constituting the token itself. It also requires the document attribute `language_swedish`. These pre-conditions are met by the output of the tokeniser (see Section 3.1.3).

GATE post-conditions

The sentence splitter produces `sentence` annotations with `sentenceVal` attributes which takes as value the sentence itself. It also adds the document attribute `uppsplit` to the current document in the GDM.

Parameters

A user may specify two additional parameters to the sentence splitter via the GGI. The options affect the splitter's output to the GDM. Either the user can choose to have the splitter convert the first token in each sentence to lower case, or to have it convert every token to lower case. These options are not mutually exclusive and if both options are given, all tokens are converted to lower case.

3.2.4 Processing

The sentence splitter works with a window of two tokens towards the stream of tokens it processes. It reads and buffers tokens until it encounters one that is defined as a sentence delimiter, currently these are ".", "!" and "?". The splitter then prints the contents of the buffer, including the sentence delimiter. There are a few exceptions to this behaviour. If the splitter encounters a token which signals the presence of consecutive newline characters in the original file, the splitter looks at the next token; if it starts with a lower case letter, the consecutive newlines are ignored and the processing goes on. However, if the first letter in the next token turns out to be uppercase, the splitter considers the contents of the buffer to be a sentence and prints it. The next token is then taken to be the start of a new sentence.

The sentence splitter is quite memory efficient since it keeps at most one sentence of the input file in memory at any one point in time.

3.2.5 Limitations

The sentence splitter is domain independent. It is also language independent as long as the language under consideration employs the same sentence delimiters as the ones used by this program. The sentence splitter will probably perform better if it is extended with linguistic knowledge such as a list of tokens that are exceptions to the rule of which kind of tokens are not allowed to start a sentence.

Chapter 4

Modules for pos-tagging and morphological processing

This chapter introduces the two NL programs used for creating the separate modules for pos-tagging and morphological processing, presented in chapter 1 as being the aim of this thesis. A Brill tagger trained on Swedish constitutes the pos-tagger while UCP is used as morphological processor. The chapter starts off with the Brill tagger for Swedish[1] in Section 4.1. It continues with the basics of transformation-based error-driven-learning and the applicability of the method to pos-tagging in Sections 4.1.1 and 4.1.2, respectively. Section 4.1.3 deals with the tagger's interface, while Section 4.1.4 presents its limitations. The part of the chapter dedicated to UCP, i.e. Section 4.2, starts with a brief introduction to chart processing in general in Section 4.2.1 and continues in Section 4.2.2 with an overview of how to express morphological competence in UCP. The interface and limitations of UCP are presented in Sections 4.2.3 and 4.2.4, respectively.

4.1 A Brill tagger for Swedish

The original Brill tagger [Brill 1992] was developed by Eric Brill. The tagger is public domain software, available from `ftp://ftp.cs.jhu.edu/pub/brill/Programs`. It is written in the C programming language and comes with training scripts written in Perl.

The Department of Linguistics, Uppsala University, currently pursues the training of a Brill tagger for Swedish [Prütz 1997] within the framework of the ETAP project. The criteria for the training corpus are that the texts must be in Swedish, translated to other languages, correct and available. The actual text used for this purpose is, among others, *Regeringsförklaringen* (Eng. *Statement of government policy*) as presented to the Swedish parliament in 1994. The development of the training corpus for the tagger involved processing the selected texts with the UCP and hence obtaining analyses for individual words out of context. UCP assigns multiple attribute-value schemata to each word where appropriate, i.e. when the word is ambiguous, and no schema at all when the word is not present in the lexicon. The texts were then disambiguated manually.

[1]Henceforth, the Brill tagger for Swedish is referred to simply as the Brill tagger or the tagger.

4.1.1 Transformation-based error-driven learning methods

Transformation-based error-driven learning [Brill 1992, Brill 1995] has been applied to a number of natural language problems, including part of speech tagging, speech generation and syntactic parsing. In transformation-based error-driven learning, an un-annotated text is run through an initial-state annotator. Then, the text is compared to the *truth*, which is specified as a manually annotated corpus, to which the results of the steps in the learning process can be compared and evaluated. The comparison results in that the system learns a number of transformation rules which can be applied to the output of the initial state annotator in order to make it better resemble the truth. See Figure 4.1 for a conceptual view of the learning method.

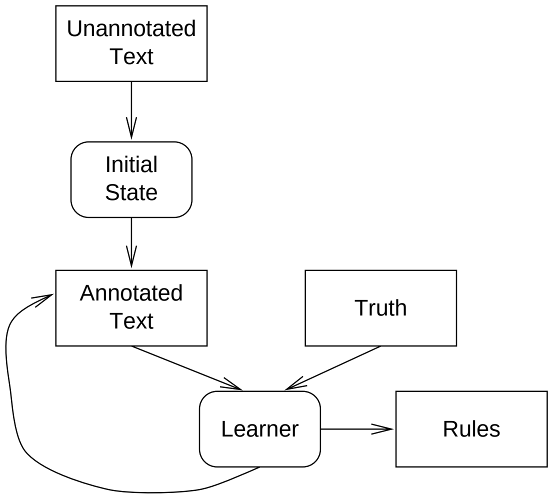


Figure 4.1: The learning method of a Brill tagger

A greedy search is then applied: at each iteration of learning, find the transformation whose application results in the *highest score* with respect to some scoring function. Add that transformation to the ordered list of transformations and update the corpus by applying the learned transformation to it. The following three things must be specified when defining an application of transformation-based learning:

1. The initial state annotator.
2. The space of transformations the learner is allowed to examine.
3. The scoring function for comparing the corpus to the truth and choosing a transformation.

When the list of transformations is learned, new text can be annotated by first running it through the initial-state annotator and then applying the transformations in order.

4.1.2 Pos tagging by means of a Brill tagger

There are by tradition two main approaches to automatic part of speech tagging:[2] one using rules and another using statistics (see [Samuelsson & Voutilainen 1997] for a comparison of a

[2]See [McEnery & Wilson 1996, pages 119-126] for an introduction to part of speech tagging.

rule based and a probabilistic system). The former approach involves hand made rules, often based on the researchers' own linguistic intuitions. The latter approach uses statistical data to capture the structure of the language under consideration. Both approaches have their advantages and disadvantages. The linguistic information used by a rule based system is likely to be easy to read (for a human, that is) while the same linguistic information in a probabilistic tagger is (unreadably) represented implicitly in large tables of statistics, typically as tens of thousands of contextual probabilities. A statistically based system is trainable, which means it is easy to adapt it to, for instance, a new language, while it is much harder to transfer a rule based tagging system to a new language. The Brill pos-tagger is a hybrid system, it is able to gain information from a corpus and to translate that information to a set of rules. This ability facilitates adapting a Brill tagger to new domains and to new languages. The processing and the principles behind transformation-based error-driven learning methods are described in the documentation of the tagger as well as in separate papers, e.g. [Brill 1992, Brill 1995, Brill 1997].

Learning the rules

The training of the Brill tagger is performed in two steps, the first one involves learning rules to predict the most likely tag for unknown words, that is, words not seen in the training corpus. In the second step, the tagger learns contextual rules to improve tagging accuracy.

The tagger tries to predict the tags of unknown words by using a transformation-based approach similar to the one used for known words. To start with, the initial state annotator labels the most likely tag for unknown words as proper noun if the word is capitalised and as common noun otherwise. To learn the actual transformation rules for predicting the tags of unknown words, the tagger has a set of 5 transformation templates.[3] The tagger's efforts in finding transformation rules applicable to unknown words is limited to word types in contrast to the learning of rules for known words which also operates on individual word tokens. Once an unknown word has been assigned a tag, all the occurrences of that word in the corpus are assigned the same tag. For a more in-depth description of a solution to the problem of tagging previously unseen words, see [Brill 1995, section 4.3].

In the second step of training the tagger, the initial state annotator assigns each word in an un-annotated version of the training corpus is most likely tag as indicated in the annotated training corpus. The transformation rules are then inferred by the learner by applying every possible transformation, counting the number of tagging errors after a transformation has been applied and choosing that transformation which results in the greatest error reduction. When no transformations whose application reduces errors beyond a pre-specified threshold are found, the learning stops. The result from this procedure of learning is an ordered list of transformation rules based on observed contextual facts from the training corpus.

Applying the rules

In performing pos-tagging, the Brill tagger first has the initial state annotator assign each word its most likely tag as indicated in the training corpus. The ordered set of rules obtained from the training session is then applied to the input text. For each transformation applied, all environments in the text that can trigger the transformation are found, and the transformation

[3]The set of transformation templates defines the space of transformations the learner, i.e. the Brill tagger, is allowed to examine.

is then carried out on all those environments. In effect, this means that the input text is scanned several times, once for every element in the set of rules.

4.1.3 Interface

All subsections in this section apply when the Brill tagger is invoked via GATE, but only the ones on the tagger's input and output apply when it is run as a stand alone program.

Input

The Brill tagger expects a plain text file as input, formatted with one, tokenised, sentence per line. In this context, tokenisation means that the punctuation marks have been separated from the words in the sentence, e.g.

```
"Stå inte så tätt isär!"
```

becomes

```
" Stå inte så tätt isär !  "
```

Output

The output from the tagger is a version of the input file with appended part-of-speech tags to each token:

```
"/" Stå/VI inte/R så/R tätt/APNSI isär/NNOI !/!  "/"
```

The Brill tagger that comes with the VIE system (Section 2.2) in GATE has a somewhat different output format which consists of the pos-tags only. This approach is due to the fact that the VIE Brill tagger is dedicated to the GATE system and is, thus, not likely to be of any use as a stand alone program.

GATE pre-conditions

The Brill tagger for Swedish requires the GDM to contain `token` and `sentence` annotations with the attribute `sentenceVal` associated to the latter type of annotation. The value of a `sentenceVal` attribute is the string constituting the sentence itself. The tagger also requires the document attribute `language_swedish`. These pre-conditions are met by the sentence splitter for Swedish described in Section 3.2.

GATE post-conditions

The tagger produces `pos` (part of speech) attributes on the existing `token` annotations. It also adds the document attribute `uppbrill` to the current document in the GDM.

Parameters

The set of parameters the Brill tagger for Swedish accepts via the GGI is the same as the VIE Brill tagger does [Humphreys *et al* 1996, chapter 4]. Thus, the tagger takes eight additional parameters, of which the first four have default values: a lexicon; a file containing bigram information; a file with lexical rules; and a file with contextual rules. The other four parameters specify whether the tagger should: use an additional word list; process the input a certain number of lines at a time to save memory; dump intermediate output to a file; and finally, if it should use the start-state tagger only. See the documentation that comes with the distribution of the Brill tagger for further information.

4.1.4 Limitations

The Brill tagger for Swedish is domain dependent in that it is trained on texts from specific domains, e.g. from *Regeringsförklaringen.* The tagger is obviously highly language dependent since the material used for training is in Swedish.

4.2 The Uppsala Chart Processor

The Uppsala Chart Processor was originally developed at the Center for Computational Linguistics at the Uppsala University in the early 80's. The first version, UCP-1, appeared in 1981 and it was written in a dialect of the LISP programming language called Inter LISP [Sågvall Hein 1981]. The current version, UCP-2, is implemented in Common LISP.

Good introductions to chart processing are given in, for instance, [Wirén 1992, chapter 2] and [Gazdar & Mellish 1989, chapter 6]. Chart-parsing originates from research in compiler techniques. Its use as a data structure in NL applications is due to, amongst others, Kay (see e.g. [Kay 1977]).

4.2.1 The essence of chart processing

A chart processor is a program that makes use of a data structure called a *chart* which provides a way to avoid generation of redundant data. The chart is a directed graph, the nodes are often called *vertices* and the arcs *edges* as illustrated in Figure 4.2.

The vertices are typically numbered from 0 to n, where n is an integer, while the edges are labeled with some representation of a portion of the input string. The labels are often realised by "dotted" context free rules on the form $X \rightarrow \alpha \cdot \beta$. Such a rule corresponds to an edge, X, which contains an analysis (a confirmed hypothesis) of a constituent, α, and which seeks the analysis (an unconfirmed hypothesis) of constituent β. If β is empty, the edge labeled with the rule is said to be *inactive* since it represents a fully analysed constituent in the input. Otherwise, if β is non-empty, the edge is *active.*

The following is an example illustrating the use of "dotted" rules for labeling arcs in a chart. If, for instance, $S \rightarrow NP\ VP$ is a rule of the grammar then items (1) to (3) below can be "dotted" grammar rules in the chart:

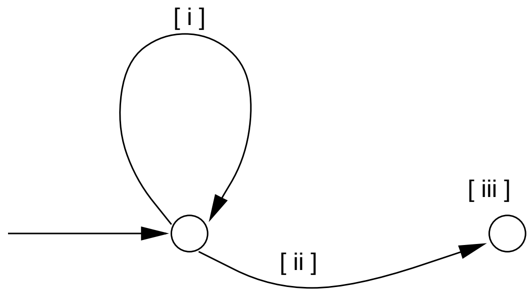


Figure 4.2: A simple chart where [i] represents the only cycle allowed, [ii] is an example of an edge and [iii] is a vertex.

1. $S \to \cdot NP\ VP$
2. $S \to NP\ \cdot VP$
3. $S \to NP\ VP \cdot$

The dots in these labels indicate to what extent the hypothesis that this rule is applicable to has been verified by the processor, see Figure 4.3. The label in (1) will only occur in the kind of arc that cycles back to the vertex it emerged from, denoting the hypothesis that S can be found covering a substring consisting of the sequence *NP VP* starting from the vertex in question. In this case, the hypothesis has not even been partly confirmed. The labels in (2) and (3) denote the same hypothesis as (1), but indicate that the hypothesis has been partially or fully confirmed. In (2), the processor has found the sequence *NP* and is now looking for *VP*. In (3), the processor has verified the hypothesis of the whole sequence *NP VP*, and thus, the label represents an inactive edge. Symbols to the left of a dot in a label represent one or more confirmed hypotheses while the symbols to the right of a dot represent one or more unconfirmed hypotheses.

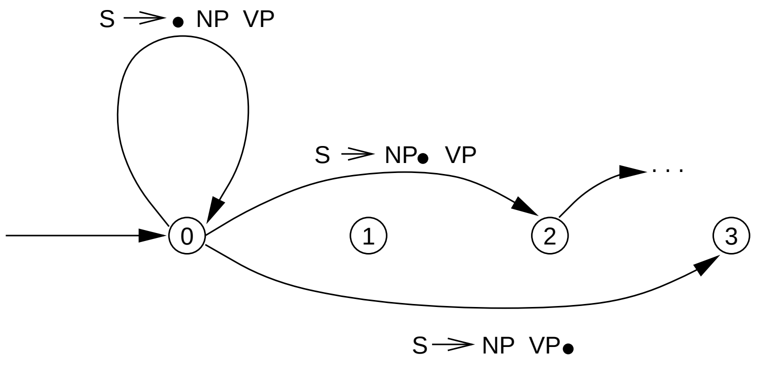


Figure 4.3: A conceptual view of a chart. Active edges are "above the horizon" while inactive are below it.

At this point, enough information has been given to introduce the essence of chart parsing, the *fundamental rule*. This rule describes how active and inactive edges are combined in the chart. An edge can be written as a triple (see [Wirén 1992, page 16] and [Gazdar & Mellish 1989, pages 193–196]):

$$\langle v_{\mathrm{s}}, v_{\mathrm{t}}, X \to \alpha \cdot \beta \rangle$$

In which v_{s} and v_{t} are the starting and ending vertices, respectively, and $X \to \alpha \cdot \beta$ is a rule as explained above. The formal definition of the fundamental rule of chart parsing is then (as cited from [Wirén 1992, page 19]):

The fundamental rule

For each edge of the form $\langle v_{\mathrm{i}}, v_{\mathrm{j}}, X_0 \to \alpha \cdot X_{\mathrm{m}}\beta \rangle$ and each edge of the form $\langle v_{\mathrm{j}}, v_{\mathrm{k}}, Y_0 \to \gamma \rangle$, add edge $\langle v_{\mathrm{i}}, v_{\mathrm{k}}, X_0 \to \alpha X_{\mathrm{m}} \cdot \beta \rangle$ if $X_{\mathrm{m}} = Y_0$.

Informally, the fundamental rule means adding an edge to the chart whenever an active edge meets an inactive edge of the desired category. The new edge should span both the active and inactive edges. Such a combination of edges in a chart may eventually result in more inactive edges, i.e. analyses of (parts of) the input to the processor. In addition to the fundamental rule, a chart parser/processor also needs ways for initialising the chart as well as a rule invocation strategy and a search strategy. Although various combinations of different decisions regarding theses three items have been shown sufficient to cover virtually all possible types of chart parsers [Wirén 1992], the fundamental rule remains, in principle, the same.

4.2.2 Morphological processing in UCP

As many other NL applications of this kind, UCP distinguishes between the linguistic competence, i.e. some description of the language under consideration, and the machinery intended to handle it, i.e. the chart and the chart processing software. In UCP, such a linguistic description consists of a *grammar*, a number of *morph lexicons* and *character*, *lemma*, and *lexeme databases*. Since UCP is quite a complex piece of software, some of the work performed "behind the scenes" is hidden from the user, e.g. spawning new tasks (a task occurs when an active edge encounters an inactive one, that is, when the fundamental rule applies) and managing the wait-list (which handles the backtracking mechanism). Still, the user plays a very important part in how UCP will behave in a given situation since he provides UCP with the description needed to analyse a certain language. To help him, a formalism has been developed, which enables full control of what UCP is doing and yet a comfortable distance to the more difficult and repetitive tasks. The formalism is described in [Dahllöf 1989, Sågvall Hein 1987], to which readers interested in gaining knowledge about how to use UCP as a syntactic analyser are best referred.

When UCP functions as a morphological analyser, incoming words are looked up in the lexicon. Each entry in the lexicon is associated with a inflection pattern which describes the class of inflections that the current word belongs to. In general, an entry in the lexicon is defined on the following format:

```
(define D-entry E #u S;)
```

Where D is the lexicon in which the lexical entry E is to be inserted. E is associated with the UCP statement S, e.g. a rule body and an inflection pattern.

When the word stem has been found, edges corresponding to the suffixes described in the inflection pattern for the word are added to the chart. The edges are then tested using a word class specific rule that adds information about the word. Morphological/morphotactic rules are then employed to categorise the words and to squeeze as much information as possible out of the chart containing them. The format of such a rule is:

(define G-entry R #u S; {#! F_1; {#! F_2;}})

Where G is the name of the grammar in which the rule will be inserted. R is the rule name, S is the body and F_1 and F_2 are optional filters. The filters are called the *inactive filter* and the *active filter* since they operate on active and inactive edges, respectively (see [Dahllöf 1989, pages 2–8]).

To give the reader a feel for what the rules may look like, Figure 4.4 gives examples of a dictionary entry and a morphological (pattern) rule that applies to it. In the figure, a dictionary entry for the word stem *arbet* is defined as being a verb following the inflection pattern of the word *älska*. The grammar rule in the second part of the figure conveys information common to all the word categorised as following the same pattern as *älska*.

```
(define sve.dic-entry "arbet" #u <& lem>:=:'arbeta.vb, pattern.älska;)

(define sve.gram-entry pattern.älska
 #u     assign.infl('älska),
        assign.dicstem,
        advance, <* char> = '%a,
        <& conj>:=:'I,
        <& depon>:=:'-,
        advance,
        (end.of.word,
        <& word.cat> :=:'verb,
        <& diat>:=:'act,
        (<& inff>:=:'inf
        // <& inff>:=:'imp),
        assign.majorprocess('vp),
        assign.same,
        store.single.morph
        / <& voice> :=: '+,
        <& linkb> :=: '%a,
        verb);
)
```

Figure 4.4: A definition of a dictionary entry and a morphology rule in UCP.

Although UCP can be told to use various rule invokation strategies, the most common one is bottom-up (possibly with top-down filtering) using a left to right search of the input, that is, the input is read from the left and the tokens in it are matched against the lexicon and placed in the chart for further processing.

4.2.3 Interface

UCP is primarily intended as a stand-alone program, but it is also possible to run it via the GATE platform.

Input

The UCP operates at many linguistic levels simultaneously (e.g. morphological and syntactical levels). The format of the input expected by UCP depends on what linguistic level the user wants it to act and, since this thesis is concerned with UCP as a morphological analyser, the input should be a file in which each line contains one word.

Output

UCP is able to produce output on a number of different formats. However, the format of interest to this thesis is the one conveying the highest degree of information about the words analysed. UCP gives zero or more analyses for a given word. Thus, it is likely that some of the words in the input will not be analysed at all, while som words will get more than one analysis. This is an example of a word with two readings:

```
i :
 (* = (LEM = I2.AB
       INFL = PATTERN.REDAN
       DIC.STEM = i
       WORD.CAT = ADV))
 (* = (LEM = I1.PP
       INFL = PATTERN.I
       DIC.STEM = i
       WORD.CAT = PREP))
```

Subsequent analyses are separated by a blank line in the output file. The attribute–value pairs in a structure like the one above are arbitrary in that the developer of the rules (grammatical or morphological) may specify any information he/she feels necessary.

GATE pre-conditions

The Uppsala Chart Processor requires the GDM to contain `token` annotations with associated `tokenVal` attributes. The value of a `tokenVal` is the string constituting the token itself. UCP also requires the document attribute `language_swedish`. These requirements are met by the output of the tokeniser for Swedish described in Section 3.1.

GATE post-conditions

UCP produces `morph` attributes on the existing `token` annotations. It also adds the document attribute `uppcp` to the GDM. The value of a `morph` attribute is a list in which each element itself is a list of lists containing an analysis of the current token. If UCP did not provide an analysis for the current token, its `morph` attribute recieves the value of an empty list. The structure of a non-empty value of the `morph` attribute is shown in Figure 4.5.

$$\{structure_1\ structure_2 \ldots structure_{\mathrm{k}}\}$$

where $structure_{\mathrm{j}}$, $1 \leq j \geq k$, is

$$\{\{attribute_1 = value_1\} \ldots \{attribute_{\mathrm{m}} = value_{\mathrm{p}}\}\}$$

$m \geq 1$ and $p \geq 1$

Figure 4.5: The structure of `morph` attribute.

Parameters

The Uppsala Chart Processor allows the user to specify two additional parameters via the GGI. The first, which has a default value, is the path to a file containing a Common LISP memory dump of morphological rules conforming to the UCP formalism [Dahllöf 1989, Sågvall Hein 1987]. The second parameter, which is an optional one, is the name of a file in which to place the output from UCP (that is, a copy of the output before it is recorded in the GDM).

4.2.4 Limitations

The Uppsala Chart Processor is domain as well as language dependent. At least when it comes to the morphological rules used here. The processing-machinery handling the chart is independent of any language or domain.

Chapter 5

Integrating modules in GATE

This chapter contains the core of the thesis, that is, how the NL modules described in Chapters 3 and 4 were integrated in GATE. It assumes the reader to be familiar with the Tcl scripting language (but see Appendix B for a brief overview of Tcl/Tk). The chapter starts with a description of the different types of couplings available between NL modules and the GATE Document Manager (GDM) in Section 5.1. Sections 5.2 and 5.3 introduce the two templates, `creole_config.tcl` and `moduleName.tcl`, used for creating so-called GATE wrappers, and Section 5.4 presents the method used for integrating the NL components in GATE. The integration of the tokeniser is elaborated upon in Section 5.5: it is the least complex wrapper implemented in this thesis and its structure is adopted in the wrappers created for the other NL modules. Section 5.6 shows the integration of the sentence splitter, followed by Section 5.7, in which the integration of the Brill tagger is discussed. The most complex wrapper is the one for UCP introduced in Section 5.8. The main difference between it and those for the other NL modules is the way it has to process the output produced by the NL module: the UCP wrapper has to read the output cumulatively until enough information has been collected (typically several lines at a time) before recording it in the GDM, while this is not the case for the other wrappers. Since it turned out that none of the existing viewers were able to display data on the format produced by UCP, a new viewer was designed and implemented. The viewer is introduced in Section 5.8.4.

The process of integrating modules in GATE (see [Cunningham *et al* 1996, chapter 2]) has been automated to a large degree. Most things can be, and are preferably, done via the GGI. However, some of the vital parts in the process must still be done by hand: for each NL module, two templates generated by GATE are to be filled with code, either in C++ or in Tcl/Tk, depending on which type of coupling should be used between the module and GATE. The templates constitute the "wrapper" for the NL component in question since they wrap the component up in code to make it conform to the GDM API. The ready-made templates are called `creole_config.tcl` and `moduleName.tcl`[1], and they describe the way of which an NL program is to communicate with the GDM. One of the most important things stated in the template files is the degree of coupling between the NL program and the GDM, the type of coupling (as described in the next section) implies the use of either of the two available APIs: Tcl or C++. This thesis is concerned with a loose degree of coupling (and thus the Tcl API), the dynamic and tight couplings are therefore only mentioned briefly.

[1]moduleName should be substituted for by the name of the NL module.

5.1 Different types of couplings

There are three different types of couplings between a CREOLE object and the GDM:

Tight coupling. Any programming language that obeys the C linking conventions can be compiled into GATE directly as a Tcl package. Using wrappers written in such a language is maximally efficient but necessitates re-compilation of GATE when modules change.

Dynamic coupling. On platforms allowing shared libraries, C-based wrappers can be loaded at run-time. This is slightly less efficient than tight coupling since it takes some time to load a wrapper. It allows for change of modules without having to recompile the GATE system.

Loose coupling. Wrappers written in Tcl can also be loaded at run-time. There is a performance penalty in comparison with using the C++-based API but it is the easiest solution of integrating simple cases of modules. The advantage is that a module can be altered and reloaded without leaving GATE.

In the work with the NL components described in this thesis, i.e. the tokeniser and the sentence splitter in Chapter 3 and the Brill tagger and Uppsala Chart Processor in Chapter 4, the type of coupling is set to be loose since the components pre-date the wrappers. One limitation in using a loose coupling is that the GDM API cannot communicate directly with the NL component, but the communication has to take place by creating intermediate results which means that the processing is quite slow in its nature. However, there are a few advantages as well. For instance, the NL module can be a ready-to-run binary, meaning the CREOLE developer[2] does not have to care about how and why the NL module works, and the supplier of the module need not provide the CREOLE developer with the source code of his long-time-and-very-expensive research. Thus, both the CREOLE developer and the NL program supplier have got less to worry about while the degree of reusability of resources, i.e. the NL module, is still high.

5.2 The creole_config.tcl template

The `creole_config.tcl` template specifies the constraints that are put on a module when it comes to its communication with the GDM, i.e. what kind of attributes must be present in the GDM before the module can be executed (pre-conditions), and what is present in the GDM after the module has been executed (post-conditions). Other things specified by the template are, for instance, in what way the data produced by the module will be displayed to the user, and what type of coupling will be used between the module and the GDM. The `creole_config.tcl` is actually nothing more than a Tcl-list, although formatted in a way that makes it easy to read. The file contains six different fields, which in some cases contain fields themselves (i.e. the Tcl-list is actually a list of lists). The fields are introduced and briefly explained below (see [Cunningham *et al* 1996] for further details).

1. `title` This field should contain the name the module has in the GGI system graphs.

[2]A CREOLE developer is a person who integrates NL modules in GATE.

2. `pre_conditions` Annotations and attributes required in the GDM in order to execute the module. The conditions affect three types of TIPSTER objects: `collection_attributes`, `document_attributes` and `annotations`. The latter may be either annotation/attribute pairs or plain annotations.

3. `post_conditions` Annotations and attributes produced by the module. The TIPSTER objects mentioned above are affected here as well.

4. `viewers` There are six generic viewers that can be used to display the output from a module, a viewer may be specified for each post condition. If a viewer is not specified for a post condition, there is no way of viewing the result corresponding to that particular condition except for when using the full annotation viewer found under the `View` menu option in the system window (see [Gaizauskas *et al* 1996b, chapter 5]). The viewers are:

   (a) `raw` Takes an annotation as parameter and uses a standard annotation viewer to display it.

   (b) `single_span` Takes an annotation as parameter and colours the text based on one of the attributes, if any, associated with the annotation. A coloured key bar is produced at the bottom of the viewer. This viewer is used for the VIE Brill tagger.

   (c) `multiple_span` Takes an annotation as parameter and highlights all chains of spans associated with it. The VIE Discourse Interpreter uses this viewer for displaying co-reference chains.

   (d) `parse` Takes an annotation as parameter and produces a parse tree, as in the VIE buChart Parser.

   (e) `text_attr` Takes a `document_attribute` or `collection_attribute` as parameter and displays the attribute value as plain text.

   (f) `text_file` Takes a filename as parameter and displays the file as plain text.

5. `parameters` This field specifies the additional parameters a module might accept via the GGI, e.g. a different set of resource files for a Brill tagger. The parameters are used by GATE to generate a dialogue box that prompts for the corresponding values.

6. `coupling` Can take one of these three values: `loose`, `dynamic` or `tight`.

5.3 The `moduleName.tcl` template

While the `creole_config.tcl` file described a module's way to communicate with the GDM in terms of requirements that must be fulfilled both by the GDM and the module itself to work properly, the `moduleName.tcl` file implements the actual CREOLE object. Informally, a CREOLE object is any NL program that is able to "talk" to GATE via the GDM. To put it more formally, a CREOLE object is a piece of software that implements a function/command called `creole_X` in either C++ or Tcl, where `X` is the name of the current module. This thesis focuses on the use of Tcl for implementing CREOLE objects. The reason why being the fact that all the Uppsala components pre-date integration and, thus, loose coupling is used. For information on the C++ API, see [Cunningham *et al* 1996].

Figure 5.1 shows the only procedure that must be defined by a loosely coupled CREOLE object. The arguments passed by GATE to the procedure are `doc`, which is the currently open GDM document, and `args`, which is a list of additional arguments passed to the module by

```
proc creole_X { doc args } {
  # Process argument in $args, if non-empty, and
  # make TIPSTER calls regarding $doc, which is a
  # document in the spirit of the TIPSTER architecture.
}
```

Figure 5.1: A template for the only procedure required of a loosely coupled CREOLE object.

the user via the GGI. The `X` in the procedure head in Figure 5.1 should be substituted for by the name of the current module. Of course, the `creole_X` module may, in turn, need other procedures to run. Such procedures can be declared within the same file as the `creole_X` or "sourced" from other files (see Section B.6 for further information about the `source` command). All work performed by a wrapper is initiated by GATE calling the `creole_X` procedure, which then is responsible for setting up the environment the current NL module needs to be able to run, as well as invoking the module and taking care of the output.

5.4 Remarks on integrating the NL modules in GATE

In general, the way each `moduleName.tcl` file handles the input and output data of the NL components of concern to this thesis can be described as the following sequence of actions:

1. **Prepare the input.** Retrieve the desired data from the GDM and, if necessary, re-format it to make it correspond to the input format required by the wrapped-up NL module. Put the input data in a temporary file.
2. **Execute the program.** Run the NL module on the file from step 1. Place the output in another temporary file.
3. **Record the output.** Scan the output file and record relevant data in the GDM.

As can be seen from these steps, the wrappers for the components store intermediate results in files rather than in the computer's internal memory. The main reason for this is the way by which the NL programs in question are invoked: they all expect their input to be in files or interactively input at a prompter. Since the latter is not an option in the case of the, supposedly, large texts that the svensk system will handle, the former approach is used. Another reason is that the NL programs pre-date integration. If that would not have been the case, i.e. the modules were designed only to work in GATE, a tight or dynamic coupling together with some look-ahead at design time might result in the ability of the modules to accept input on another format than files, for instance by letting the NL program access the input via direct calls to the GDM using the C++ API.

5.5 Integrating the tokeniser

The integration of the tokeniser for Swedish was quite straightforward since I had access to the GATE wrappers developed for the svensk system [Eriksson 1997]. Those wrappers also make use of a loose coupling (and thus the Tcl API) between the CREOLE object and the GDM.

The organisation of the wrapper for the tokeniser is shown in Figure 5.2. The `creole_upptoken` procedure uses `upptoken_prepareInput` for preparing the input to the tokeniser before invoking it, and `upptoken_recordOutput` to scan and record the data produced by the tokeniser in the GDM.

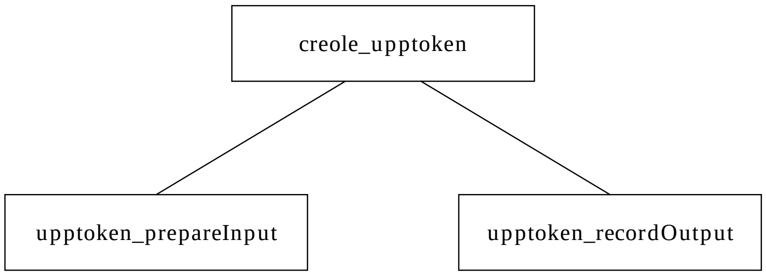


Figure 5.2: The organisation of the tokeniser's wrapper.

5.5.1 Preparing the input

The `upptoken_prepareInput` procedure takes two arguments: `doc` and `file`. The former is the currently open GDM document, and the latter is a path to a file in which the prepared input for the tokeniser should be placed. The vital parts of the procedure are shown in Figure 5.3.[3]

```
 1.     set Text [tip_GetByteSequence $doc]
 2.     set F [open $file w]
 3.     puts $F $Text
```

Figure 5.3: Preparing the input to the tokeniser.

The GDM command `tip_GetByteSequence` in line 1 is used to retrieve the text from `doc` (the tokeniser requires the input to be plain text, see Section 3.1.3). In line 3, the file handle created in line 2 is used for writing the text to the temporary file `file` (see Section B.6 for information about file I/O in Tcl).

5.5.2 Invoking the tokeniser

The tokeniser reads the input from the standard input channel and writes the produced output to the standard output channel. This means that it does not explicitly handle the opening and closing of files. Figure 5.4 shows the Tcl command which invokes the tokeniser.

The variable `$UpptokenDir` holds the path to the directory where the tokeniser resides. The paths to the temporary files used are stored in `$IntermediateFile` and `$FinalFile`.

[3]The numbers to the left in the figures in the rest of this chapter are there for clarity only, they are not part of the code.

```
exec $UpptokenDir/upptoken < $IntermediateFile > $FinalFile
```

Figure 5.4: Invoking the tokeniser.

5.5.3 Recording the output

The procedure that records the output produced by the tokeniser is called `upptoken_recordOutput`. It takes two arguments, `doc` and `file`. The first argument is the currently open GDM document while the second argument is a temporary file holding the output as produced by the tokeniser (the format is described in Section 3.1.3). The procedure reads `file` and splits each line into three parts corresponding to the starting and ending byte offsets, and a string which may be a token or a `multiNl` string signalling consecutive new line characters. The procedure then creates appropriate annotations (that is, `token` or `multiNl` annotations) and records them in the GDM with the associated byte offsets as indices.

```
1.        set TokenAttr [tip_CreateAttribute tokenVal \
2.            [tip_CreateAttributeValue GDM_STRING $Token]]
3.        set Annot [tip_CreateAnnotation token \
4.            [list [tip_CreateSpan $Start $End]] [list $TokenAttr]]
5.        tip_AddAnnotation $doc $Annot
```

Figure 5.5: Recording the output from the tokeniser in the GDM.

Figure 5.5 shows how a `token` annotation with an associated `tokenVal` attribute is created and placed in the GDM. In lines 1 and 2, a `tokenVal` attribute is created and given the value of the current token stored in the variable `$Token`. The commands in lines 3 and 4 create a `token` annotation and associate the `tokenVal` attribute with it. `$Start` and `$End` are the starting and ending byte offsets for the current token. The annotation is added to the GDM in line 5. The `multiNl` annotations are recorded in a similar way, with the exception of not having an attribute.

5.6 Integrating the sentence splitter

The main difference between integrating the tokeniser and the sentence splitter lies in that the former needs the input to be a plain text file, while the latter requires the input to be on a well-defined format. This made the procedure used to prepare the input to the splitter a bit more complex than the corresponding one for the tokeniser. Another difference is the possibility for users to give additional options to the sentence splitter via the GGI. The organisation of the splitter's wrapper, which is similar to that of the tokeniser's, is shown in Figure 5.6. The top-level procedure, `creole_uppsplit`, uses `uppsplit_prepareInput` to format the contents in the GDM to fit the needs of the splitter, and `uppsplit_recordOutput` to record the data produced in the GDM.

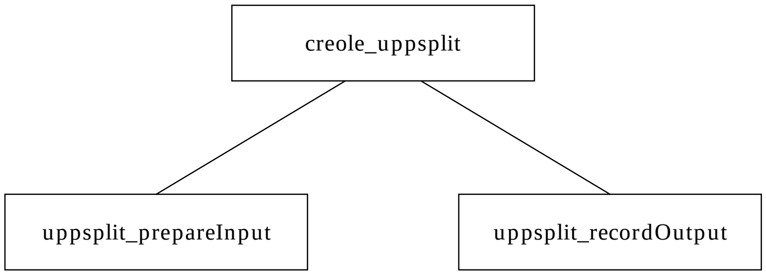


Figure 5.6: The organisation of the sentence splitter's wrapper.

5.6.1 Preparing the input

The `uppsplit_prepareInput` procedure takes two arguments: `doc` and `file`. The former is the currently open GDM document, and the latter is a path to a temporary file in which the prepared input is to be placed.

The task the procedure has to deal with is to make `file` look exactly as the output from the tokeniser would have done if it had been invoked on the text in `doc`, since this is what the splitter expects (see Section 3.2.3). To achieve this, `uppsplit_prepareInput` takes both `token` and `multiNl` annotations into consideration and prints them to `file` while maintaining correct relative order between each unique annotation of the two annotation types. To help out in deciding which of two annotations to print, there are two things to consider. The first is that both `token` and `multiNl` are single span annotation types, meaning that each annotation has only one span associated to it. The other important thing is that the tokeniser *never* returns `multiNl` annotations for which the ending byte offset for the first one is the starting byte offset for the second. That is, if the tokeniser returns `multiNl 0 2`, it will not return `multiNl 2 5`. The essence of these two important things is that, once a `multiNl` annotation has been printed, a `token` annotation is always next (if any annotation at all, that is).

```
1.    set TokenAnnotations [tip_SelectAnnotations $doc token {}]
2.    set MultiNlAnnotations [tip_SelectAnnotations $doc multiNl {}]
3.    set MultiNlCount 0
4.    if {[llength $MultiNlAnnotations] > 1} {
5.        set MultiNlLastIdx [expr [llength $MultiNlAnnotations] - 1]
6.    } else {
7.        set MultiNlLastIdx -1
8.    }
```

Figure 5.7: Retrieving `token` and `multiNl` annotations from the GDM.

Figure 5.7 shows how `token` and `multiNl` annotations are retrieved from the GDM (lines 1 and 2). The annotations are stored in Tcl lists, `TokenAnnotations` and `MultiNlAnnotations`. Since it seems reasonable to assume that the number of `multiNl` annotations will be lower than that of `token` annotations, counters keeping track of the current position in the list with `multiNl` annotations and the upper bound of that list, are initiated in lines 3 and 4 to 7. This approach facilitates for the procedure to iterate over the list of `token` annotations and to consider `multiNl` annotations only when necessary. A negative value on `MultiNlLastIdx` means that there were no `multiNl` annotations present in the GDM for the current document.

At this point, it is time to start processing the lists containing the annotations. The byte offsets of the first `multiNl` annotation, if any, are retrieved. The procedure then considers each `token` annotation, one at a time, comparing the starting byte offsets of the `token` annotations with those of the `multiNl` annotations, see Figure 5.8.

```
  1.    if { $TokenStart <= $MultiNlStart } {
  2.        puts $F "$TokenStart $TokenEnd $TokenAttribute"
  3.    } else {
  4.        puts $F "multiNl $MultiNlStart $MultiNlEnd"
  5.        puts $F "$TokenStart $TokenEnd $TokenAttribute"
  6.        if {$MultiNlCount <= $MultiNlLastIdx} {
  7.            set MultiNlSpan [lindex [tip_GetSpans \
  8.                    [tip_Nth $MultiNlAnnotations $MultiNlCount]] 0]
  9.            set MultiNlStart [tip_GetStart $MultiNlSpan]
 10.            set MultiNlEnd [tip_GetEnd $MultiNlSpan]
 11.            incr MultiNlCount
 12.        }
 13.    }
```

Figure 5.8: Gathering annotations from two sets in one file, maintaining the relative order between their members.

If the starting byte offset of the current `token` annotation is smaller than that of the current `multiNl` annotation (line 1), the `token` annotation is printed to the temporary file (line 2). Otherwise, the `multiNl` annotation is printed to the file, followed by the `token` annotation (lines 4 and 5). The byte offsets for the next `multiNl` annotation are then retrieved (lines 6-11). When both lists with annotations have been exhausted, the input to the sentence splitter is available in the file associated with `$F`.

5.6.2 Invoking the sentence splitter

A somewhat different approach than the one used for the tokeniser had to be taken in order to invoke the sentence splitter. The splitter allows for the user to specify additional parameters via the GGI (see Section 3.2.3). The parameters are passed in a (possibly empty) Tcl list to the wrapper. Since the command line options are not mutually exclusive, the list, `$args`, is converted to a string `$CommandLine` in line 1, see Figure 5.9.

```
1. set CommandLine [join $args " "]
2. eval exec $UppsplitDir/uppsplit $CommandLine < $IntermediateFile > $FinalFile
```

Figure 5.9: Invoking the sentence splitter.

The Tcl command `eval` precedes `exec` in line 2, the reason being that the `$CommandLine` variable would cause the wrapper to abort execution otherwise. `$CommandLine` is not recognised by the splitter as a valid argument since the white space characters in it are considered as a part of one argument rather than separators between several arguments. The `eval` command concatenates all arguments, with white spaces as separators, and then executes the result as a Tcl script (see Section B.6 for information about the `eval` command).

5.6.3 Recording the output

Recording the output from the sentence splitter resembles the task of recording of the output produced by the tokeniser. However, the `uppsplit_recordOutput` procedure, which handles the output from the splitter, does not have to take `multiNl` annotations into consideration since it operates on `sentence` annotations rather than `token` annotations. `uppsplit_recordOutput` takes two arguments, `doc` and `file`. The former is the currently open GDM document and the latter is the file which contains the output produced by the sentence splitter. Each line in `file` is split into three parts, two integers and a string. The string is a sentence and the integers are its starting and ending byte offsets, respectively (the output format is described in Section 3.2.3). The procedure then records `sentence` annotations with `sentenceVal` attributes in the GDM in a way analogous to the one in Figure 5.5.

5.7 Integrating the Brill tagger

The Brill tagger is different from the previous NL modules since it adds attributes to existing annotations. In this case, it is `token` annotations that recieve new attributes, called `pos`. The attributes added to the GDM by the tokeniser and the sentence splitter are added at the same time as the annotations they are associated to, i.e. there is no need to check for correspondence between the annotation and the attribute. The wrapper for the Brill tagger, more precisely, the procedure that records the output from the tagger, has to make sure the sentences in the output have been tagged properly, i.e. that each token has got a pos-tag appended to it. If this is not the case, the tagger notifies the user and aborts the processing.

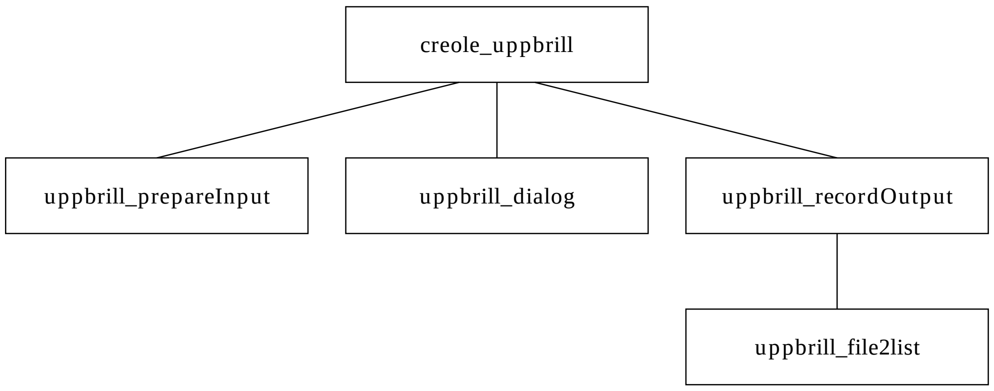


Figure 5.10: The organisation of the Brill tagger's wrapper.

The organisation of the Brill tagger's wrapper is shown in Figure 5.10. The top-level procedure, `creole_uppbrill`, uses procedures for preparing the input to the NL program as well as for recording the output in the GDM. It also makes use of a procedure called `uppbrill_dialog` for generating dialog boxes which issue status/error messages to the user. The `uppbrill_recordOutput` employs a procedure, `uppbrill_file2list`, for converting the file containing the output from the tagger to a Tcl list.

5.7.1 Preparing the input

The `uppbrill_prepareInput` takes two arguments: `doc` and `file`. The former is the currently open GDM document, and the latter is the path to a temporary file in which to place the prepared input for the Brill tagger.

```
 1.    set SentenceAnnotations [tip_SelectAnnotations $doc sentence {}]
 2.    foreach SentenceAnnotation $SentenceAnnotations {
 3.        set SentenceAttribute [tip_GetValue \
 4.                [tip_GetAttribute $SentenceAnnotation sentenceVal]]
 5.        puts $F $SentenceAttribute
 6.    }
```

Figure 5.11: Preparing the input to the Brill tagger.

Figure 5.11 shows how all `sentence` annotations are selected from the GDM (line 1). The `sentenceVal` attributes corresponding to each annotation are then retrieved (lines 3 and 4) and printed to the temporary file (line 5), resulting in a file with one sentence per line. The required input format is described in Section 4.1.3.

5.7.2 Invoking the Brill tagger

The Brill tagger accepts additional (user specified) arguments via the GGI, as in the case of the sentence splitter (see 5.6.2). However, there is a slight difference in the way the wrappers for the two modules handle the list of user specified arguments. There are four default arguments specified for the tagger and they are picked out and stored in variables rather than treated as anonymous parts of a string (mainly because it makes it easier to understand the wrapper code).

```
 1.    set lexicon [lindex $args 0]
 2.    set bigrams [lindex $args 1]
 3.    set lexRules [lindex $args 2]
 4.    set conRules [lindex $args 3]
 5.    set AddCommands " "
 6.    set RestOfArgs [lrange $args 4 end]
 7.    if {[llength $RestOfArgs] > 0} {
 8.        set Commands [join $RestOfArgs " "]
 9.        set AddCommands [format " %s " $Commands]
10.    }
```

Figure 5.12: Collecting default and additional arguments given via the GGI.

The default arguments are picked out in lines 1 to 4 in Figure 5.12. If additional arguments have been given by the user, they are collected in the variable `AddCommands` by the Tcl code in lines 5 to 10. The control of whether the arguments given to the tagger make sense or not is delegated to the tagger itself. Default arguments are accessible to the wrapper as long as the user does not specify new values in the dialog box: they are present in `$args` even if the corresponding fields in the dialog box have been cleared and left empty.

A user who wants to execute a Brill tagger has to change his/her working directory to that of the tagger's. Otherwise, the tagger cannot find two additional programs called start-state-tagger and final-state-tagger which it needs in order to run. In Figure 5.13, there are three lines of Tcl code that stores the current working directory (line 2) before it changes to the new one (line 3).

```
1.    set UppbrillDir /home/staff/gateuser/Brill/Bin
2.    set OldDir [exec pwd]
3.    cd $UppbrillDir
```

Figure 5.13: Changing to the tagger's directory.

Once the wrapper has got the arguments and the directory right, it is time to invoke the Brill tagger. The tagger continuously prints information to the standard error channel about the state of processing it is in. This information has to be dealt with in one of several ways (if left alone, it will cause GATE to abort processing since the exception handling mechanisms will interpret the status information as an error and act accordingly). One way is simply to ignore whatever messages comes from the tagger, i.e. re-direct all messages to some place where they do no harm. However, this does not seem to be a very good idea since the user will not have a clue of what has happened if the tagger refuses to work. The Brill tagger has a large set of status/error messages that helps the user to figure out what is happening during processing. Why not use them? Other ways of dealing with the information are to save it in a log-file for later use or to display it to the user immediately. Currently, the tagger's wrapper implements the latter way, all messages produced by the Brill tagger are shown to the user at once. One problem in all this is that the tagger writes both status and error messages to the same channel, meaning they are hard to tell apart. In Figure 5.14, a Tcl command called `catch` is used to collect status

```
1.    set Code [catch {eval exec tagger \
2.            $lexicon \
3.            $BrillInput \
4.            $bigrams \
5.            $lexRules \
6.            $conRules${AddCommands}> $BrillResult} Msg]
7.    if {$Code == 1} {
8.        uppbrill_dialog .d {Brill Status} $Msg info 0 {Ok}
9.    }
```

Figure 5.14: Invoking the tagger using `catch` to collect status and error messages.

and error messages produced by the tagger (line 1. See Section B.7 for information about the `catch` command). The tagger is invoked (lines 2 to 6) with the arguments from Figure 5.12 and a path to the file `$BrillInput` which was created by `uppbrill_prepareInput` in Figure 5.11. The `catch` command stores the return value from the tagger in `Code` and the message(s) printed by the tagger to the error channel in `Msg` (lines 1 and 6, respectively). The value of `Code` is then used to trigger a procedure, `uppbrill_dialog`, that displays the contents of `Msg` to the user (line 8).

5.7.3 Recording the output

The procedure `uppbrill_recordOutput` records the output from the Brill tagger in the GDM. It takes two arguments, `doc` which is the currently open GDM document, and `file`, which is a file containing the output produced by the tagger.

As already mentioned, the Brill tagger is different from the previously described NL programs in that it adds new GDM attributes to existing annotations. One additional attribute is added to each `token` annotation. The attribute, which is called `pos` for part-of-speech, recieves as value the tag the Brill tagger assigns to every token in its input. Thus, it is important that the `token` annotations present in the GDM matches the output of the Brill tagger so that, after `uppbrill_recordOutput` is done, each token has the right part-of-speech tag associated with it in the GDM. To ensure this, the first thing the `uppbrill_recordOutput` procedure does is to convert the file containing the output from the tagger (the format is described in Section 4.1.3) to a Tcl list in which each element is on the form `word/tag`, where `word` is a token in the input and `tag` is a pos-tag given by the Brill tagger. The `word` part of each element is then used to match `tag` with the right `token` annotation.

```
 1.    set BrillResult [uppbrill_file2list $file]
 2.    set TokenAnnotations [tip_SelectAnnotations $doc token {}]
 3.    set CurrIndex 0
 4.    foreach TokenAnnotation $TokenAnnotations {
 5.        set TokenValue [tip_GetValue \
 6.             [tip_GetAttribute $TokenAnnotation tokenVal]]
 7.        set CurrBrillInput [lindex $BrillResult $CurrIndex]
 8.        regexp {^ *(.+)/(.+) *$} $CurrBrillInput Trash Word PosTag
 9.        if {$Word == $TokenValue} {
10.            set PosAttribute [tip_CreateAttribute pos \
11.                    [tip_CreateAttributeValue GDM_STRING $PosTag]]
12.            tip_AddAnnotation $doc \
13.                    [tip_PutAttribute $TokenAnnotation $PosAttribute]
14.            incr CurrIndex 1
15.        } else {
16.            return  -code error \
17.                    -errorinfo $errorInfo \
18.                    -errorcode $errorCode \
19.                        "The output produced by the Uppsala Brill \
20.                        tagger is corrupt: the character sequence \
21.                        found doesn't match the current token \
22.                        annotation"
23.        }
24.    }
```

Figure 5.15: Recording the output from the Brill tagger in the GDM.

Figure 5.15 contains the Tcl code doing roughly what the previous paragraph talked about. The output from the tagger is converted by a procedure called `uppbrill_file2list` to a list `BrillResult` in line 1. The `token` annotations are retrieved from the GDM in line 2. The current `tokenVal` and the current element, `Word` and `PosTag`, are picked out in lines 5 to 8. If there is a match, i.e. the `tokenVal` is the same as `Word`, then a `pos` attribute is created and added to the GDM (lines 9 to 14). On the other hand, if a mismatch occurs, the procedure returns an error code and an error message (see Section B.7 for information about exceptional events in Tcl).

```
1.    set Code [catch {uppbrill_recordOutput $doc $BrillResult} Msg]
2.    if {$Code == 1} {
3.        uppbrill_dialog .d {Error trying to record Brill output} $Msg \
4.                  error 0 {Ok}
5.        set TokenAnnotations [tip_SelectAnnotations $doc token {}]
6.        foreach token_annotation $TokenAnnotations {
7.           tip_AddAnnotation $doc [tip_RemoveAttribute $TokenAnnotation pos]
8.        }
```

Figure 5.16: Calling `uppbrill_recordOutput`, taking possible errors into consideration.

Figure 5.16 shows how `uppbrill_recordOutput` is invoked from within the `creole_uppbrill` procedure (line 1). Again, the `catch` command is used to catch possible errors that a mismatch between the output as produced by the Brill tagger and the `token` annotations present in the GDM might cause. If an error occurs, the `pos` attributes added to the GDM so far have to be removed in order to keep the database consistent (lines 5 to 8).

5.8 Integrating the Uppsala Chart Processor

The wrapper integrating the Uppsala Chart Processor in GATE is by far the largest one implemented in this thesis (see the listing in Appendix D.2). It differs in some essential aspects from the other wrappers described in the previous sections. As for the preparation of the input to UCP, the wrapper converts the first token in every sentence to lower case (as described in the next section). To achieve this, the wrapper must be able to recognise sentence delimiters constituted either by certain values of the current `tokenVal` attribute or by the presence of a `multiNl` annotation. Thus, the wrapper takes more information into consideration than any of the other wrappers at the same point in time during processing. When it comes to invoking UCP, the wrapper cannot pass the additional arguments on to the UCP as a raw string (as was the case in Sections 5.6.2 and 5.7.2), rather, the arguments are used to build LISP commands that are then passed to UCP. For instance, such a command could tell UCP which lexicon to use and where to place an additional output file. Finally, when recording the output from any of the other modules, the procedure responsible for that could read from a file and record data in the GDM in a line-at-a-time fashion. Since the structures produced by UCP span several lines, the current procedure has to read and process the file containing the output cumulatively, until enough information is gathered and then record it in the GDM.

The organisation of the wrapper in terms of the procedures it uses and the relation between them is shown in Figure 5.17. The procedure `creole_uppcp` employs three procedures: `uppcp_prepareInput` for preparing the input for UCP, `uppcp_dialog` for issuing error messages to the user and `uppcp_recordOutput` for recording the output produced by UCP in the GDM. The latter uses, directly or indirectly, another three procedures: `uppcp_toLowerCase`, for converting upper case characters from the ISO 8859-1 character set to lower case, and `uppcp_file2list` for converting the file containing the output from UCP to a Tcl list, `uppcp_file2list` in turn, uses a procedure called `uppcp_refineList` to clean up partial lists.

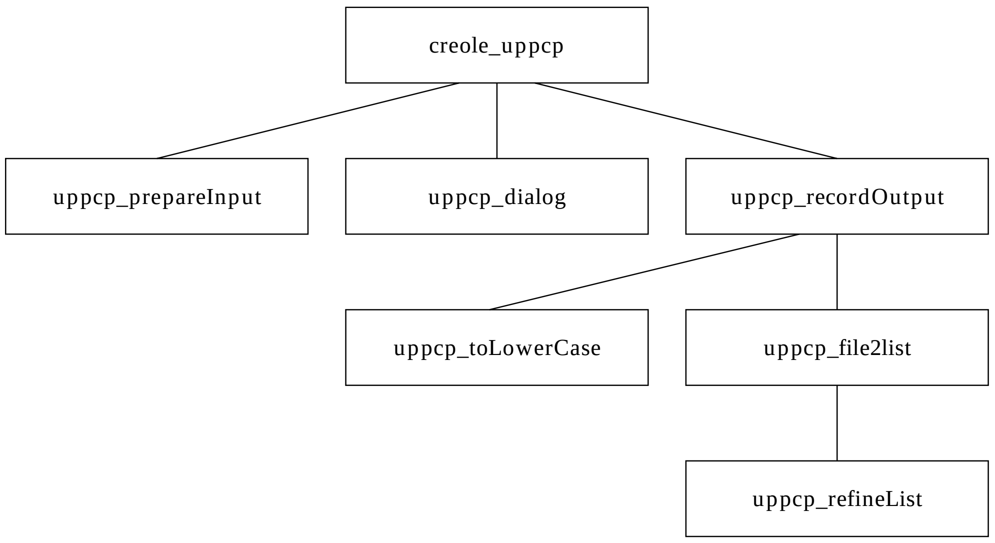


Figure 5.17: The organisation of the Uppsala Chart Processor's wrapper.

5.8.1 Preparing the input

A procedure called `uppcp_prepareInput` handles the preparation of the input to the Uppsala Chart Processor. The procedure takes two arguments: `doc`, which is the currently open GDM document; and `file`, which is a path to a file in which to put the prepared input.

To facilitate for UCP to analyse the words in the input, the wrapper converts the first token in every sentence to lower case. This is because the information encoded in the lexicon and morphological rules used by the UCP is in lower case, with the exception for proper names. Of course, the conversion of tokens employed here is not the best solution possible since proper names starting a sentence will not be assigned correct analyses. A proper name converted to lower case may be interpreted by UCP as something completely different. For example, consider the name *Ester* (i.e. *Esther*), when written in lower case, *ester* can be interpreted as taken from a chemical domain or as Estonians, while the intended analyse as a proper name is not very likely to occur. The `uppcp_prepareInput` procedure was implemented under the assumption that proper names are less frequent at the first position in a sentence than any other word is.

To be able to recognise the first token in the sentences, the wrapper has to decide which tokens are sentence delimiters and which are not. Thus, the work performed by `uppcp_prepareInput` resembles, to some extent, that of the sentence splitter described in Section 3.2. However, `uppcp_prepareInput` only has to recognise the delimiters, it does not have to buffer its input and print it whenever appropriate, which means the procedure still is less complex than the sentence splitter.

`uppcp_prepareInput` uses an approach similar to that described in Figures 5.7 and 5.8: it considers both `token` and `multiNl` annotations, maintaining the relative order between the annotations in the two sets. The basic idea is then to compare byte offsets for the current `token` annotation and the current `multiNl` annotation as well as using information about the previously seen token in order to know when a token should be converted to lower case and printed and when it should be printed as is. Figure 5.18 illustrates how `uppcp_prepareInput` converts the appropriate tokens to lower case. `$TokenStart` and `$MultiNlStart` are the starting byte offsets for the current `token` and `multiNl` annotations, respectively. `$PrevTokenAttribute`

```
 1.    if { $TokenStart <= $MultiNlStart } {
 2.        if {[string match \. $PrevTokenAttribute]} {
 3.            puts $F "[uppcp_toLowerCase $TokenAttribute]"
 4.        } elseif {[string match \! $PrevTokenAttribute]} {
 5.            puts $F "[uppcp_toLowerCase $TokenAttribute]"
 6.        } elseif {[string match \\? $PrevTokenAttribute]} {
 7.            puts $F "[uppcp_toLowerCase $TokenAttribute]"
 8.        } else {
 9.            puts $F "$TokenAttribute"
10.        }
11.        set PreviousAnnotation "token"
12.        set PrevTokenAttribute $TokenAttribute
13.    } else {
14.        puts $F "[uppcp_toLowerCase $TokenAttribute]"
15.        set PreviousAnnotation "multiNl"
16.        set PrevTokenAttribute ""
17.        if {$MultiNlCount <= $MultiNlLastIdx} {
18.            set MultiNlSpan [lindex [tip_GetSpans \
19.                    [tip_Nth $MultiNlAnnotations $MultiNlCount]] 0]
20.            set MultiNlStart [tip_GetStart $MultiNlSpan]
21.            set MultiNlEnd [tip_GetEnd $MultiNlSpan]
22.            incr MultiNlCount
23.        }
24.    }
```

Figure 5.18: Deciding whether a token is the first one in a sentence or not.

holds the value of the `tokenVal` attribute seen before the current one, which is stored in `$TokenAttribute`. `$PreviousAnnotation` is a flag signaling the type of the previous annotation. It can assume the strings `token` or `multiNl` as value. The `if`-statement starting at line 1 in Figure 5.18 applies if the current type of annotation is `token`. The procedure then tries to match the previous `tokenVal` attribute against one of the sentence delimiters (.! and ?), and if successful, convert the attribute to lower case and print it to the temporary file (lines 2 to 7). The procedure used for converting a string to lower case is called `uppcp_toLowerCase` and it is not the same as the built in `string tolower` construct since that does not handle the ISO 8859-1 character set the way needed by the wrapper.

The `else` branch starting in line 13 applies if the current type of annotation is `multiNl`. The procedure then prints the current `tokenVal` attribute to the temporary file (line 14) and proceeds with the next `multiNl` annotation, if any, for comparison with the next `token` annotation (lines 17 to 23).

5.8.2 Invoking UCP

The wrapper integrating UCP in GATE allows for the user to give additional arguments to UCP via the GGI. This was also the case with the sentence splitter and the Brill tagger as described in sections 5.6.2 and 5.7.2. However, in those cases, the contents of the list containing the user specified parameters, `$args`, could more or less be converted to a string which was then passed as is to the NL program in question. UCP expects the additional arguments to be embedded in LISP commands which are then passed to UCP together with a memory dump of the morphological rules at loading time, the reason being that the arguments are originally intended to be input interactively to UCP via the underlying Common LISP interpreter, but a later patch allows a user to specify the arguments on the command line. Figure 5.19 shows how the `$args` list is processed.

```
 1.    set RemoveParseFile 1
 2.    set UcpMorphology [lindex $args 0]
 3.    if {[lindex $args 1] == ""} {
 4.        set LispCommands \
 5.                [format  \
 6.                "(progn (usegd) (try-file \"%s\" :report-style :parses))" \
 7.                $UcpInput]
 8.    } else {
 9.        set LispCommands \
10.                [format \
11.                "(progn (usegd) (try-file \"%s\" \
12.                :report-style :parses :parse-file \"%s\"))" \
13.                $UcpInput [lindex $args 1]]
14.        set UcpParses [lindex $args 1]
15.        set RemoveParseFile 0
16.    }
```

Figure 5.19: Processing arguments given by the user via the GGI.

`RemoveParseFile` (line 1) is a flag telling the wrapper whether the file containing the analyses produced by UCP should be removed or not at the end of processing. `$UcpInput` (line 7) is the path to the file in which the prepared input to UCP resides and `$UcpParses` (line 14) is the path to a file in which the output from UCP is to be stored.

Currently, the UCP wrapper accepts only two user specified arguments, of which the first, a path to a set of morphological rules, has a default value. The second argument is a file in which to save the analyses produced by UCP. If the GGI dialog box is left unchanged, i.e. there is only one argument in `$args`, the wrapper builds the LISP expression at line 6 in Figure 5.19. Otherwise, the expression in lines 11 and 12 is constructed.

5.8.3 Recording the output

A procedure called `uppcp_recordOutput` records the output from UCP in the GDM. The procedure takes two arguments, `doc`, which is the currently open GDM document, and `file`, which is the temporary file used for storage of the output from UCP.

The output produced by UCP is structurally more complex than the output from the other modules in that several lines, rather than just one, must be considered in order to record one attribute in the GDM (the output from UCP is described in Section 4.2.3). First, `uppcp_recordOutput` converts the file containing the analyses produced by UCP to a list. The list is then compared to the token annotations present in the GDM. All `token` annotations present in the GDM are assigned a `morph` attribute. The value of `morph` depends on whether there is a match between the current element in the list of analyses and the current `tokenVal` attribute. If there is, the `morph` attribute is assigned as value a list of lists, in which each element contains an analysis (sequence of attribute–value-pairs) for the word represented as the current `tokenVal` attribute. Otherwise, if there is no match, the `morph` attribute is assigned an empty Tcl list as value. There will most likely not be a one-to-one correspondence between the analyses given by UCP and the `token` annotations present in the GDM since UCP assigns zero or more analyses to any given word (in contrast to the Brill tagger which assigns one pos-tag to every token in the input, see Section 5.7). If the current analysis does not match the current `tokenVal` attribute, the processing goes on with the next attribute. The only mismatch that causes the procedure to abort processing is if there are UCP analyses left when the `token` annotations are exhausted.

Reading the file containing the output from UCP is done in one pass using two procedures: `uppcp_file2list` and `uppcp_refineList`. The former reads the file line by line while building up a list containing the current analysis. Once such a list is built, it is passed to `uppcp_refineList` which, as the name suggests, refines the list, meaning the attribute–value-pairs are extracted and put in lists of their own. The structure of the list resulting from these procedures is shown in Figure 5.20 and the source code of the UCP wrapper in Appendix D.2.

$$\{analyse_{\mathrm{word1}} \;\; analyse_{\mathrm{word2}} \ldots analyse_{\mathrm{wordn}}\}$$

$$\text{where } analyse_{\mathrm{wordi}}\,,\; 1 \leq i \geq n, \text{ is}$$

$$\{\{token_1\} \; structure_1 \ldots structure_{\mathrm{k}}\}$$

$$\text{and } structure_{\mathrm{j}},\; 1 \leq j \geq k, \text{ is}$$

$$\{\{attribute_1 = value_1\} \ldots \{attribute_{\mathrm{m}} = value_{\mathrm{p}}\}\}$$

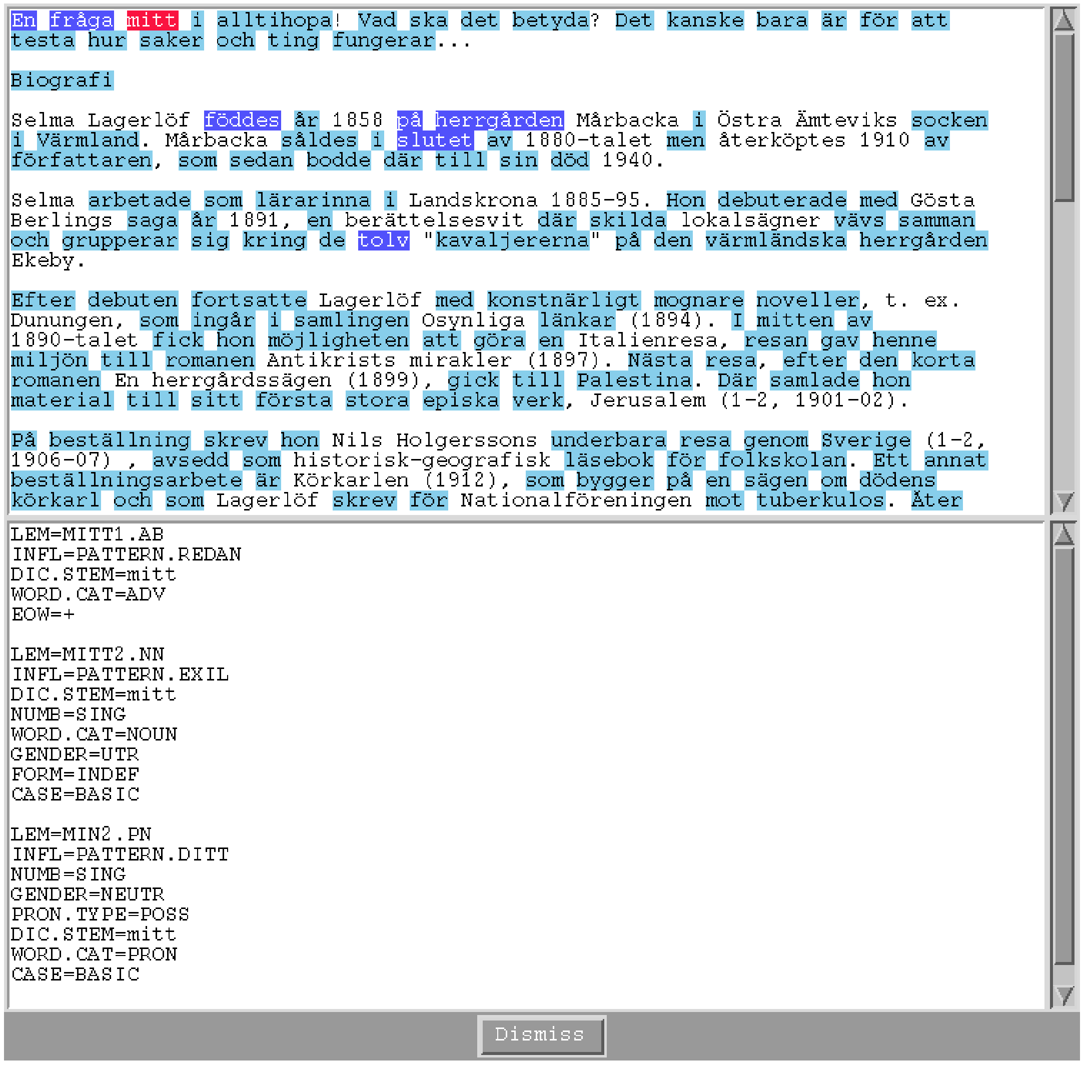
$$m \geq 1 \text{ and } p \geq 1$$

Figure 5.20: The structure of the list of morphological analyses resulting from applying `uppcp_file2list` on a file containing the output from UCP.

5.8.4 Viewing ambiguities — the single_span_ambiguities viewer

None of the viewers available in GATE was suitable for displaying the output of UCP. The ones considered were `raw` and `single_span` (see Section 5.2), but using any of them would surely only confuse the user since, in some cases, the value associated to a `morph` attribute is so big (in terms of numbers of characters, that is) it cannot be displayed all at once. In fact, there is no viewer in GATE able to handle attributes, associated to single span annotations, taking on multiple values like the multiple analyses UCP may assign to a word. This is why the `single_span_ambiguities`-viewer was implemented. It is written in Tcl/Tk with the guidance of source code from available viewers as well as with help from the people at the University of Sheffield who are developing the GATE system. Hopefully, the viewer is general enough to be of use to any CREOLE object which produces attributes that are stored in the same fashion as the `morph` attributes.

Figure 5.21 shows the viewer. The upper window contains a text processed by UCP. Words written in black with a grey background (which is light blue in real life) in the figure have been assigned at least one analysis while the words in black without background colour have not received any at all. When an analysed word is clicked by the user, the corresponding analyses are displayed in the bottom window. The colours of the words that have been clicked are changed to white text on dark blue background. The current word, i.e. the one whose analyses are displayed, is written in white on a red background. In Figure 5.21, the currently activated word is *mitt* (third from the left at the first line in the upper window).



```
En fråga mitt i alltihopa! Vad ska det betyda? Det kanske bara är för att
testa hur saker och ting fungerar...

Biografi

Selma Lagerlöf föddes år 1858 på herrgården Mårbacka i Östra Ämteviks socken
i Värmland. Mårbacka såldes i slutet av 1880-talet men återköptes 1910 av
författaren, som sedan bodde där till sin död 1940.

Selma arbetade som lärarinna i Landskrona 1885-95. Hon debuterade med Gösta
Berlings saga år 1891, en berättelsesvit där skilda lokalsägner vävs samman
och grupperar sig kring de tolv "kavaljererna" på den värmländska herrgården
Ekeby.

Efter debuten fortsatte Lagerlöf med konstnärligt mognare noveller, t. ex.
Dunungen, som ingår i samlingen Osynliga länkar (1894). I mitten av
1890-talet fick hon möjligheten att göra en Italienresa, resan gav henne
miljön till romanen Antikrists mirakler (1897). Nästa resa, efter den korta
romanen En herrgårdssägen (1899), gick till Palestina. Där samlade hon
material till sitt första stora episka verk, Jerusalem (1-2, 1901-02).

På beställning skrev hon Nils Holgerssons underbara resa genom Sverige (1-2,
1906-07) , avsedd som historisk-geografisk läsebok för folkskolan. Ett annat
beställningsarbete är Körkarlen (1912), som bygger på en sägen om dödens
körkarl och som Lagerlöf skrev för Nationalföreningen mot tuberkulos. Åter
```

```
LEM=MITT1.AB
INFL=PATTERN.REDAN
DIC.STEM=mitt
WORD.CAT=ADV
EOW=+

LEM=MITT2.NN
INFL=PATTERN.EXIL
DIC.STEM=mitt
NUMB=SING
WORD.CAT=NOUN
GENDER=UTR
FORM=INDEF
CASE=BASIC

LEM=MIN2.PN
INFL=PATTERN.DITT
NUMB=SING
GENDER=NEUTR
PRON.TYPE=POSS
DIC.STEM=mitt
WORD.CAT=PRON
CASE=BASIC
```

Figure 5.21: The `single_span_ambiguities`-viewer.

When designing a viewer for the GATE system, there are a few things to keep in mind:[4]

- Procedures or global variables used in the viewer should not already be in use by GATE.

[4]Based on personal communication with Pete Rodgers and Hamish Cunningham, the Department of Computer Science at the University of Sheffield.

- The definition of the viewer procedure should be
  `ggi_view_TYPE { doc annotation attribute title }`
  where `doc` is the currently open GDM document, `annotation` is the annotation to be viewed, `attribute` is the specific attribute of the annotation to be viewed and `title` is the name of the viewer to appear in the module viewer window. `TYPE` should be substituted for by a viewer name not already in use.

- The file containing the viewer must have the extension `.gw` to enable integration of the viewer into the GATE system.

The procedures used in the ambiguity viewer and the relations between them are displayed in Figure 5.22. Boxes connected to `ggi_view_single_span_ambiguities` with dashed lines represent three procedures present in the files `ggi.tcl` and `ggi_viewers.tcl`, which constitute parts of the GGI and the viewers available in GATE. The procedure `ggi_wait_message` creates a window telling the user that work is in progress; `ggi_destroy_wait` is used to destroy such a window; `gu_gensym` generates a unique symbol each time it is called, symbols that are used for naming windows so the GGI can tell them apart at updates, deletions etc. The other boxes represent procedures constituting the actual viewer, the source code of which is available in appendix D.3.

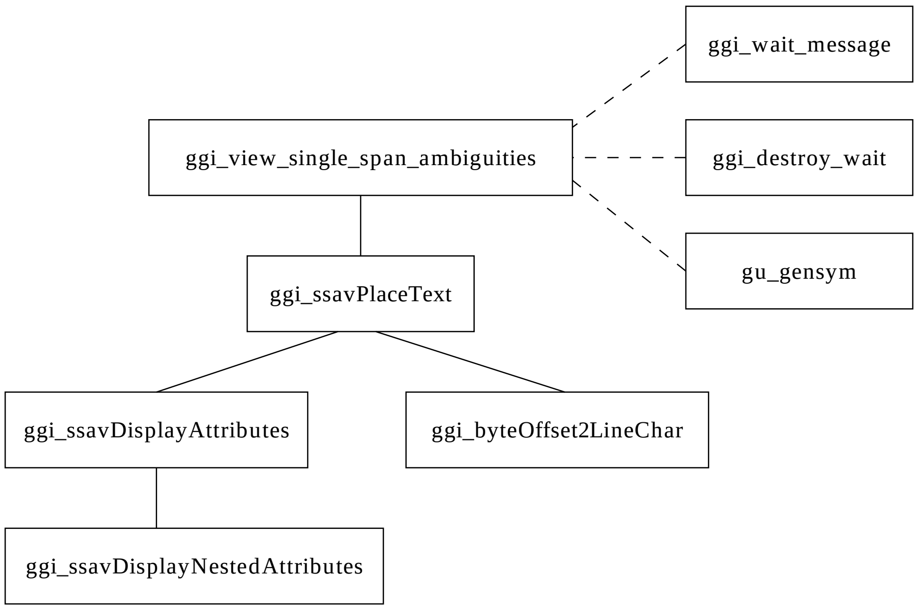

ggi_wait_message
ggi_view_single_span_ambiguities
ggi_destroy_wait
gu_gensym
ggi_ssavPlaceText
ggi_ssavDisplayAttributes
ggi_byteOffset2LineChar
ggi_ssavDisplayNestedAttributes

Figure 5.22: Procedures used in the `single_span_ambiguities`-viewer.

The top-level procedure, `ggi_view_single_span_ambiguities`, sets up the window and fills the upper half with the text of the current GDM document using the `ggi_ssavPlaceText` procedure. The latter places the text in a given window, using `ggi_byteOffset2LineChar` to convert the span corresponding to each relevant GDM annotation to co-ordinates in terms of lines and characters as used by text widgets in Tk. Each span is assigned a unique tag, which is an integer denoting the index of an element in a list containing the attributes in question (in the case of UCP, these are `morph` attributes associated to `token` annotations), and bound to a procedure called `ggi_ssavDisplayAttributes`. When a highlighted portion of text in the upper window is activated, i.e. clicked by the user, the co-ordinates given by the windowing system are used by the `ggi_ssavDisplayAttributes` to retrieve the tag corresponding to the clicked area. This tag, in turn, is used to get the value of the attribute-value associated to the span. The value of the attribute is a list (possibly containing lists of arbitrary nesting depth) which is passed to `ggi_ssavDisplayNestedAttributes` that formats and displays the information in the bottom window. All in all, a hypertext effect is achieved since tags associated to pieces of text in the upper window are used as addresses to the attributes in the GDM which are then displayed in the bottom window.

Chapter 6

Concluding discussion and future work

This chapter concludes the thesis with a discussion of the results of the integration, an estimation of the time spent in the different parts of the thesis and a short note on possible future work.

First of all, the GATE system is easy to work with, although it might take some time to get acquainted to it. I have only come to know parts of it in depth, that is, the Tcl API using a loose coupling between the GDM and the NL modules to be integrated. Also, I have spent more time on integrating the modules than I have on using the system purely from an end-user's point of view. Table 6.1 illustrates the approximate amount of time I have spent on the different phases in the creation of the thesis, it also shows that a person without previous knowledge about GATE and Tcl/Tk can integrate NL programs in GATE after a fairly short learning period.

Time	*Activity*
3	Exploring the GATE system documentation and practical behaviour
2	Learning about UCP and the Brill tagger
1	Basic wrapper design
3	Designing and implementing the tokeniser and the sentence splitter
2	Tcl basics
2	Integrating the tokeniser
2	Integrating the sentence splitter
2	Integrating the Brill tagger
2	Integrating the UCP
1	Tk basics
1	Implementing and integrating the ambiguity viewer
6	Writing a first version of the report (including learning LaTeX)

Table 6.1: Time (in weeks) spent in the different phases of the thesis.

One of the best things about the modularity of the GATE system is that, once a module is integrated into GATE, it is easy to combine it with existing modules to form new NL systems. However, I believe GATE would be even more user and developer friendly if it provided for interaction between the NL modules and the user (or other programs) during the execution of the module in question. For instance, once an NL module has been invoked, the current version of GATE does not allow the user to interrupt the module, and conversely, GATE is unable to prompt the user for information using the existing commands/functions in the GATE core.[1] A high degree of interaction between GATE and a user would be required by, for instance, an NL module for manual disambiguation of the output from UCP. Of course, it is possible for any CREOLE developer to define his own set of commands/functions for solving problems like these, but perhaps the best solution would be to extend the GATE core with a standardised set of commands for handling user interaction, i.e. dialog boxes, exception handling,[2] and other ways to communicate with NL modules that are first and foremost intended to be used interactively.

The process of integration was, with a few exceptions (see Appendix A), straightforward since it was possible to use the same approach in all four wrappers. It is likely that the approach, which is described in Section 5.4, is applicable for integrating any non-interactive NL program in GATE. The Tcl API and the loose coupling are also shown sufficient to successfully integrate NL programs implemented in programming languages from completely different paradigms (in this case the languages are C, Perl and Common LISP). There is, however, one major drawback of the wrappers: the most complex one, i.e. the one for UCP (see Appendix D.2), tends to spend a lot of its processing time to record the output produced. Therefore, I conclude that Tcl is suitable for processing large sets of data (especially lists) only if the developer/user is not concerned with reducing the time of the processing that is spent in the wrapper vs. the time spent by the NL program actually processing texts. For smaller wrappers, like the ones integrating the tokeniser and the sentence splitter, the use of Tcl seems to be no greater disadvantage. As a user, my impression of the time it takes to load the tokeniser or the sentence splitter is that it is about the same as it takes to load the corresponding modules in the VIE system which are integrated in GATE using a tight coupling and the C++ API.

As previously pointed out, it is easy to integrate new NL modules in GATE and, as shown with the `single_span_ambiguities`-viewer, it is also quite easy to incorporate a new viewer in GATE. It is my belief that it is equally easy to extend GATE as a platform with respect to other parts, such as a set of dialogue/interface handling primitives, and that extensions not necessarily need to be provided for by one site only, i.e. the University of Sheffield, but can be designed, implemented and incorporated in GATE by other sites.

When it comes to future work, I would like to see the two APIs appropriately compared in terms of speed and accessibility. Especially, it would be interesting to re-write the UCP wrapper in C++ using a tight coupling and to compare its performance to that of the one implemented in Tcl.

[1]The version of GATE referred to here is 1.0.3.

[2]There are already ways to handle exceptions defined in the GATE documentation, but in my opinion, they are hard to use.

Bibliography

[Beskow *et al* 1997] Jonas Beskow, Kjell Elenius, and Scott MacGlashan. "Olga — A dialogue system with an animated talking agent". In *Proceedings of the 5th European Conference on Speech Communication and Technology*, volume 3, pp. 1651–1654, Rhodes, Greece, 1997.

[Bredenkamp *et al* 1997] A. Bredenkamp, T. Declerck, F. Fouvry, B. Music, and A. Theofilidis. "Linguistic Engineering using ALEP". In *Proceedings of the 2nd International Conference on Recent Advances in Natural Language Processing*, pp. 92–97, Tzigov Chark, Bulgaria, 1997.

[Brill 1992] Eric Brill. "A Simple Rule-Based Part of Speech Tagger". In *Proceedings of the 3rd Conference on Applied Natural Language Processing*, pp. 152–155, Trento, Italy, 1992. ACL.

[Brill 1995] Eric Brill. "Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part of Speech Tagging". *Computational Linguistics*, pp. 543 – 565, 1995.

[Brill 1997] Eric Brill. "Unsupervised Learning of Disambiguation Rules for Part of Speech Tagging". In K. Church, S. Armstrong, P. Isabelle, E. Tzoukermann, and D. Yarowsky, editors, *Natural Language Processing Using Very Large Corpora*. Kluwer Academic Press, Dordrecht, Holland, 1997. in press.

[Bub & Schwinn 1996] Thomas Bub and Johannes Schwinn. "Verbmobil: The Evolution of a Complex Large Speech-to-Speech Translation System". In *Proceedings of the 4th International Conference on Spoken Language Processing*, Philadelphia, Pennsylvania, 1996.

[Cohen *et al* 1989] Philip R. Cohen, Adam J. Cheyer, Michelle Wang, and Soon Choel Baeg. "An Open Agent Architecture". In *AAAI Spring Symposium*, pp. 1–8, Stanford University, California, 1994.

[Cunningham *et al* 1995] Hamish Cunningham, Robert J. Gaizauskas, and Yorick Wilks. "General Architecture for Text Engineering (GATE) – a new approach to Language Engineering R&D". Technical Report CS–95–21, Dept. of Computer Science, University of Sheffield, Sheffield, England, December 1995.

[Cunningham *et al* 1996] Hamish Cunningham, Yorick Wilks, and Robert J. Gaizauskas. "GATE – a General Architecture for Text Engineering". In *Proceedings of the 16th International Conference on Computational Linguistics*, volume 2, pp. 1057–1060, København, Denmark, 1996. ACL.

[Cunningham *et al* 1996] Hamish Cunningham, Kevin Humphreys, Robert J. Gaizauskas, and Martin Stower. *CREOLE Developer's Manual.* Sheffield, England, 1996.

[Dahllöf 1989] Mats Dahllöf. "Satslösning i en lexikonorienterad parser för svenska". Master of Art Thesis, University of Gothenburg, Dept. of Computational Linguistics, Gothenburg, Sweden, 1989. (in Swedish).

[Edwards 1993] Jane A. Edwards. "Survey of Electronic Corpora and Related Resources for Language Researchers". In J.A. Edwards and M.D. Lampert, editors, *Talking Data: Transcription and Coding in Discourse Research*, pp. 263–310. Erlbaum, Hillsdale, New Jersey, 1993.

[Eriksson 1997] Mikael Eriksson. *SVENSK Module Specification.* Kista, Sweden, January 1997.

[Eriksson & Gambäck 1997a] Mikael Eriksson and Björn Gambäck. "SVENSK: A Toolbox of Swedish Language Processing Resources". In *Proceedings of the 2nd International Conference on Recent Advances in Natural Language Processing*, pp. 336–341, Tzigov Chark, Bulgaria, 1997.

[Eriksson & Gambäck 1997b] Mikael Eriksson and Björn Gambäck. "Final Report of SVENSK". Technical report, Swedish Institute of Computer Science, Kista, Sweden, September 1997.

[Gaizauskas *et al* 1996a] Robert Gaizauskas, Hamish Cunningham, Yorick Wilks, Peter Rodgers, and Kevin Humphreys. "GATE: An Environment to Support Research and Development in Natural Language Engineering". In *Proceedings of the 8th IEEE International Conference an Tools with Artificial Intelligence*, Toulouse, France, 1996.

[Gaizauskas *et al* 1996b] Rob Gaizauskas, Pete Rodgers, Hamish Cunningham, and Kevin Humphreys. "GATE User Guide". Technical report, Department of Computer Science and Institute for Language, Speech and Hearing (ILASH), University of Sheffield, UK, 1996.

[Gambäck 1997] Björn Gambäck. *Processing Swedish Sentences: A Unification-Based Grammar and some Applications.* Doctor of Engineering Thesis, The Royal Institute of Technology and Stockholm University, Dept. of Computer and Systems Sciences, Stockholm, Sweden, June 1997. Also available as SICS Dissertation Series 21, Swedish Institute of Computer Science, Kista, Sweden.

[Gazdar & Mellish 1989] Gerald Gazdar and Chris Mellish. *Natural Language Processing in Prolog.* Addison-Wesley, Wokingham, England, 1989.

[Grefenstette & Tapanainen 1994] Gregory Grefenstette and Pasi Tapanainen. "What is a word, What is a sentence? Problems of Tokenization". In *Proceedings of the 3rd Conference on Computational Lexicography and Text Research*, Budapest, Hungary, 1994.

[Grishman *et al* 1997] Ralph Grishman et al. *TIPSTER Text Phase II Architecture Design. Version 2.3.* New York, January 1997.

[Guo 1997] Jin Guo. "Critical Tokenization and its Properties". *Computational Linguistics*, 23(4):569–596, December 1997.

[Habert *et al* 1998] B. Habert, G. Adda, M. Adda-Decker, P. Boula de Marëuil, S. Ferrari, O. Ferret, G. Illouz, and P. Paroubek. "Towards Tokenization Evaluation". In Antonio Rubio et al., editors, *Proceedings of the First International Conference on Language Resources and Evaluation*, volume I, pp. 427–431, Granada, Spain, 1998. ELRA.

[Hirschman *et al* 1996] Lynette Hirschman, Henry S. Thompson, Beth Sundheim, John Hutchins, Ezra Black, Margaret King, David S. Pallet, Adrian Fourcin, Lois C. W. Pols, Sharon Oviatt, Herman J. M. Steeneken, and Junichi Kanai. "Evaluation". In Ronald A. Cole et al., editors, *Survey of the State of the Art in Human Language Technology.* Oregon Graduate Institute of Science and Technology, 1996. The book is available only on the Internet at the address: `www.cse.ogi.edu/CSLU/HLTsurvey/`.

[Humphreys *et al* 1996] Kevin Humphreys, Robert J. Gaizauskas, Hamish Cunningham, and Saliha Azzam. *CREOLE Module Specifications.* Sheffield, England, 1996.

[Humphreys *et al* 1996] Kevin Humphreys, Robert J. Gaizauskas, Hamish Cunningham, and Saliha Azzam. *VIE Technical Specifications.* Sheffield, England, 1996.

[Hutchins & Somers 1992] W. John Hutchins and Harold L. Somers. *An Introduction to Machine Translation.* Academic Press, London, England, 1992.

[Johnson 1997] Ray Johnson. "Tcl Style Guide", August 1997. Available on the Internet at `sunscript.sun.com/techcorner/`.

[Karlsson 1992] Fred Karlsson. "SWETWOL: A Comprehensive Morphological Analyser for Swedish". *Nordic Journal of Linguistics*, 15(1):1–45, 1992.

[Karlsson *et al* 1995] Fred Karlsson, Atro Voutilainen, Juha Heikkilä, and Arto Anttila, editors. *Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text.* Mouton de Gruyter, Berlin, Germany, 1995.

[Karttunen *et al* 1996] Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. "Regular Expressions for Language Engineering". *Natural Language Engineering*, 2(4):305–328, 1996.

[Kay 1977] Martin Kay. "Reversible Grammar: Summary of the Formalism". Technical report, Xerox Palo Alto Research Center, Palo Alto, California, 1977.

[Koskenniemi 1983] Kimmo Koskenniemi. *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production.* Doctor of Philosophy Thesis, University of Helsinki, Dept. of General Linguistics, Helsinki, Finland, 1983.

[Levine *et al* 1995] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reilly & Associates, Sebastopol, California, 1995.

[McEnery & Wilson 1996] Tony McEnery and Andrew Wilson. *Corpus Linguistics.* Edinburgh University Press, 1996.

[Moran *et al* 1997] Douglas B. Moran, Adam J. Cheyer, Luc E. Julia, David L. Martin, and Sangkyu Park. "Multimodal User Interfaces in the Open Agent Architecture". In *Proceedings of the International Conference on Intelligent User Interfaces*, pp. 61–68, Orlando, Florida, 1997. ACM.

[Olsson *et al* 1998] Fredrik Olsson, Björn Gambäck, and Mikael Eriksson. "Reusing Swedish Language Processing Resources in SVENSK". In *Workshop on Minimizing the Effort for Language Resource Acquisition*, Granada, Spain, 1998. European Language Resources Association.

[Ousterhout 1994] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[Ousterhout 1997] John K. Ousterhout. "Scripting: Higher Level Programming for the 21st Century". Whitepaper, Sun Microsystems Laboratories, 1997. This paper is a draft dated March 28, 1997.

[Palmer & Hearst 1994] David D. Palmer and Marti A. Hearst. "Adaptive Sentence Boundary Disambiguation". In *Proceedings of the 4th Conference on Applied Natural Language Processing*, Stuttgart, Germany, 1994. ACL.

[Prütz 1997] Klas Prütz. "Sammanställning av en träningskorpus på svenska för träning av ett automatiskt ordklasstaggningssystem". Technical report, Dept. of Linguistics, Uppsala University, Uppsala, Sweden, 1997. (in Swedish).

[Rayner *et al* 1993] Manny Rayner, Hiyan Alshawi, Ivan Bretan, David M. Carter, Vassilios Digalakis, Björn Gambäck, Jaan Kaja, Jussi Karlgren, Bertil Lyberg, Stephen G. Pulman, Patti Price, and Christer Samuelsson. "A Speech to Speech Translation System Built from Standard Components". In *Proceedings of the Workshop on Human Language Technology*, Princeton, New Jersey, 1993. ARPA, Morgan Kaufmann. Also available as SRI International Technical Report CRC-031, Cambridge, England.

[Samuelsson 1994] Christer Samuelsson. "Notes on LR Parser Design". In *Proceedings of the 15th International Conference on Computational Linguistics*, volume 1, pp. 386–390, Kyoto, Japan, 1994. ACL.

[Samuelsson & Voutilainen 1997] Christer Samuelsson and Atro Voutilainen. "Comparing a Linguistic and a Stochastic Tagger". In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics, ACL*, pp. 246–253, 1997.

[Simpkins & Groenendijk 1994] N. Simpkins and M. Groenendijk. "The ALEP Project". Technical report, Cray Systems/CEC, Luxembourg, 1994.

[Sunnehall 1996] Joel Sunnehall. "Robust Parsing Using Dependency with Constraints and Preference". Master of Art Thesis, Uppsala University, Uppsala, Sweden, September 1996.

[Sågvall Hein 1981] Anna Sågvall Hein. "An Overview of the Uppsala Chart Parser Version 1 (UCP-1)". Technical report, Center for Computational Linguistics, Uppsala University, Uppsala, Sweden, 1981.

[Sågvall Hein 1987] Anna Sågvall Hein. "Parsing by Means of the Uppsala Chart Processor (UCP)". In Leonard Bolc, editor, *Natural Language Parsing Systems*, pp. 203–266. Springer-Verlag, Berlin, Germany, 1987.

[Sågvall Hein *et al* 1997] Anna Sågvall Hein, Ingrid Almqvist, and Per Starbäck. "Scania Swedish – A Basis for Multilingual Machine Translation". In *Proceedings of the 19th Conference on Translating and the Computer*, London, England, 1997. ASLIB.

[TIPSTER 1996] Architecture Committe for the TIPSTER Text Phase II Program. *TIPSTER Text Phase II Architecture Concept.* New York, New York, 1996.

[Wirén 1992] Mats Wirén. *Studies in Incremental Natural-Language Analysis.* Doctor of Philosophy Thesis, Linköping University, Dept. of Computer and Information Science, Linköping, Sweden, December 1992.

Appendices

Appendix A

Trouble-shooting — 4 questions and answers about integration

This appendix deals with some practical questions that came up during the integration of the NL programs introduced in chapters 3 and 4.

What about using additional Tcl procedures, that is, procedures used by `creole_X` in the `moduleName.tcl` file?

When developing CREOLE objects, the (possible) additional procedures defined at the same level as the `creole_X` procedure in any `moduleName.tcl` file must have different names! Otherwise, there will be unreported name clashes when GATE tries to execute a system, which will result in a behaviour that may appear random to the user: the wrapper works fine one moment but not at all the next. The reason being that all procedures at the top-level in the wrappers shares the same name space in GATE. As a consequence, if two procedures have the same name, only the definition of the last loaded one will be used by GATE. However, such behaviour can be prevented by either using unique names for the procedures or by defining the additional procedures within the scope of the `creole_X` procedure (things defined within a the scope of a procedure cannot be accessed from outside that procedure).

GATE refuses to start — what's wrong?

What is wrong if GATE refuses to start after the last module was loaded, but instead gives an error message similar to this?

```
user@computer ~$> This is GATE, version 1.0.3
command returned bad code: 537861148
    while executing
"gate_startup"
    (file "/usr/local/lib/gate-1.0.3/Build/gate" line 567)
```

The reason for this particular error is that there was a mistyping in the head of the procedure `creole_X`, e.g. `pric` instead of `proc`, and the module was then loaded into GATE before the typo was discovered. At next start-up, GATE gave the above message. Behaviour like this is due to syntactical errors in the Tcl code making up the wrapper.

To attend to such errors, the directory holding the wrapper must be renamed and the module reintegrated and loaded from a "fresh" directory. The erroneous wrapper code may then be taken care of before it is copied to and loaded as the new module.

For instance, if the module `test` is situated in a user's `creole` directory such that GATE can find it in `~user/creole/test`, `creole` may be renamed to, say, `creole_tmp`, preserving the contents of the `creole`. The original `creole` directory may then be emptied. Now, launch GATE and create a new module named `test` in the empty `creole` directory. Then take care of the erroneous code in the version of `test` located in `creole_tmp`, copy the corrected files to the new `~user/creole/test` directory and reload the module.

How can I make a system I've created available to all users?

Suppose you are logged on as root on your local system, and that you have integrated some new modules in the `$GATE/creole` directory (where `$GATE` is the path to the top-level of the local GATE system). After having created a new system containing the newly integrated modules, the question is how to make the new system available to all users, and not only to the root user? The answer lies in the `.gate.props` file, which is a property file that controls the appearance of GATE. A user must have a copy of the `.gate.props` file in his/her home directory that contains information about the new system. To achieve this, the user could copy root's `.gate.props` file, i.e. the file `~root/.gate.props` generated by GATE when the new system was created. As root, you can replace the default file, which resides in `$GATE/Build/gate.props` with `~root/.gate.props` and the delete the users' `.gate.props` files. However, this approach is only applicable as long as no user has created their own, locally available, system. If the property file is deleted, the local system will no longer be available to the user, and in that case, it is probably better to edit the user's `.gate.props` file so the local system is kept intact while the user still gains access to the new, global, system.

How can I get my new viewer into the local GATE system?

To integrate a new viewer into an already successfully installed GATE system, the following steps should be taken (henceforth, `$GATE` refers to the top-level directory of your local GATE system and `viewer.gw` refers to the file containing the viewer):

1. Copy `viewer.gw` to the directory `$GATE/Build/`
2. Two lines in the file `$GATE/gui/Makefile.inc` has to be changed:
   - Add `viewer.gw` to the `SCRIPTS:=` –line.
   - Add `viewer.gw` file to the `gui` line under `Application targets`.
3. In the file `$GATE/gui/ggi.tcl`, add the line
   `source @APPL_HOME@/viewer.gw`
   at the top of the file, nearby the other `source`...–lines.
4. Change to the `$GATE/Build` directory and type
   `make gui`⟨*enter*⟩
   After a while, GATE should answer:
   `******* made gui`

Appendix B

A brief overview of Tcl

This chapter introduces, very briefly, the parts of the Tcl scripting language that were used throughout the thesis. It will serve as a pointer to further reading as well as a good start in integrating NL modules, using a loose coupling, in the GATE system. Section B.1 introduces the reader to the syntax of Tcl as well as to the notion of substitutions, Section B.2 elaborates on simple variables and associative arrays, while mechanisms for controlling the flow of execution in a Tcl program is introduced in Section B.3. Next, how procedures are declared and used is presented in Section B.4. Section B.5 elaborates on Tcl lists and strings, two of the most important features of the language when it comes to creating GATE wrappers. Another thing needed is the ability to communicate with other programs, i.e. spawning external processes and handling file I/O, introduced in Section B.6. Finally, errors, exceptions and ways to handle those are presented in Section B.7.

There is a big difference between the paradigms of languages known as system programming languages and scripting languages [Ousterhout 1997]. The former are languages such as C, C++, Ada, Pascal and the latter such as Visual Basic, Perl and Tcl/Tk. The thing a user might notice is the difference in abstraction from the underlying platform (operating system and hardware) and degree of typing[1]. System programming languages are primarily suited for creating fast and efficient programs while the scripting languages are used to combine those programs into larger systems. Typically, system-programming languages have a higher degree of typing than scripting languages. Of course, there are advantages with both paradigms and it all boils down to being a trade-off between control and abstraction. For instance, the number of computer actions taken from one single statement in Tcl code might range from 100 to 1000, while the same figure for a language like C is likely to be in the interval of 5 to 50. Tcl provides a developer with a high level of abstraction and is thus easy to learn and to use. It is well suited to be used in applications such as GATE wrappers and viewers.

The rest of this appendix pretty much follows the structure of [Ousterhout 1994] and most of the examples are adopted from there. A good aid in Tcl programming style is given in [Johnson 1997]. There are also numerous on-line sources on the Internet. The one of most use to the work in this thesis was `sunscript.sun.com/`.

The text in the examples and figures used throughout the rest of this appendix follows some typographical rules: text typed in to the computer by a user is written in `typewriter` style and whatever the computer responds to those particular Tcl commands is written in *italic typewriter* style preceded by an arrow, $\Rightarrow$, unless an error occurred, then the answer

[1]The term "typing" refers to the degree to which the meaning of information is specified in advance of its use.

is preceded by ⊘. In some figures, the response from the computer is left out, and line numbers are introduced in the input to emphasise what is important (e.g. the equivalence of the two Tcl scripts in Figure B.1).

B.1 Syntax

A *Tcl script* is made up of one or more *commands* which, in turn, consists of one or more *words*. Each command is separated from the others by new lines or semicolons. For instance, in Figure B.1 the script in lines 1a and 2a is equivalent to that in line 1b.[2] Words are separated by whitespace characters (spaces or tabs). The first command in each of the scripts in Figure B.1 has three words while the second command has two. A word is an arbitrary string which does not contain any white space character.

```
1a.    set Var "SVENSK"
2a.    puts $Var

1b.    set Var "SVENSK"; puts $Var
```

Figure B.1: Commands and words in Tcl.

There are three forms of substitution in Tcl. When a substitution occurs, some of the original characters in a word are replaced by a new value. The three substitutions provided for are *variable-*, *command-* and *backslash* substitution:

Variable substitution is triggered by the dollar ($) sign. It causes the value of a variable to be inserted into a word. The command in line 1 in Figure B.2 sets the value of the variable `seconds` to 60. The command in line 2 calculates how many seconds there are in five minutes. Variable substitution is used to replace `$seconds` with the value of the variable `seconds`, what is actually calculated is `60*5`.

```
1.    set seconds 60
2.    expr $seconds*5
```

Figure B.2: Variable substitution.

Command substitution causes a word (or a part thereof) to be replaced by the result of a Tcl command. This kind of substitution is triggered by brackets ([and]), enclosing a valid Tcl script. Command substitution occurs in line 2 in Figure B.3. It causes the variable `nbrSeconds` to be set to the result of the Tcl script within the brackets: `[expr $seconds*5]` is replaced by 300 and assigned to `nbrSeconds`.

Backslash substitution is used to insert special characters such as $, [and newlines into Tcl commands. This is achieved by inserting a backslash (\) before the special character.

[2] The numbers to the left in the figures in the rest of this appendix are there for clarity only, they are not part of the Tcl code.

```
set seconds 60
set nbrSeconds [expr $seconds*5]
```

Figure B.3: Command substitution.

```
set str Time\ is\ \$\nWork\ hard!
```

Figure B.4: Backslash substitution.

The variable `str` in Figure B.4 would be printed as:

```
Time is $
Work hard
```

There are a number of ways to prevent the Tcl interpreter from regarding characters such as $, new lines and semicolons as special characters. The use of this technique is called quoting. Backslash substitution is one quoting technique, the use of double quotes (") is another and braces ({}) is a third.

Characters following a # character up to the next newline are considered as comments in a Tcl script.

B.2 Variables

In Tcl, there are two kinds of variables, *simple variables* and *associative arrays*. A variable has a name and a value, both of which may be an arbitrary string. Variable names are case sensitive, i.e. `svensk` is different from `Svensk`. Variables are always stored as strings in Tcl, even though they may appear to be characters, integers, reals, lists etc.. This is possible because Tcl is designed to be "typeless", i.e. variables do not have types.

The `set` command is used to create, read and modify variables. It takes either one or two arguments of which the first is the name of a variable and the second, if present, is the new value for that variable. In Figure B.5 the workings of the `set` command is illustrated. The first command sets the variable `Var` to a string, the second command is used to query for the value of `Var` and the third command modifies the value. In all cases, the `set` command returns the value of the variable.

The `incr` command is used to increase the value of a variable which has an integer string as value, as illustrated in Figure B.6. It takes either one or two arguments, the first being the variable whose value is to be increased and the second, if present, is an integer string denoting the increment. The first command in Figure B.6 sets the variable `Var` to 10. The second command increases the value by one, which is the default used if there is no second argument to `incr`. The third command increases the value of `Var` by 3. The `incr` command returns the new value as well as stores it in the variable.

The third and last Tcl command concerning variables introduced in this section is the `unset` command, illustrated in B.7. It is used to delete variables and it takes an arbitrary number

```
set Var "SVENSK employs GATE"
⇒ SVENSK employs GATE
set Var
⇒ SVENSK employs GATE
set Var 4711
⇒ 4711
```

Figure B.5: The `set` command.

```
set Var 10
⇒ 10
incr Var
⇒ 11
incr Var 3
⇒ 14
```

Figure B.6: The `incr` command.

```
set Var SVENSK
⇒ SVENSK
unset Var
set Var
⊘ can't read "Var": no such variable
```

Figure B.7: The `unset` command.

of arguments. The first `set` command in the figure sets the variable `Var` to the value `SVENSK`, which is then returned. The `unset` command in the second line unsets `Var`. Note that it seems as if nothing is returned but, actually, the empty string is. Finally, the unsetting of the variable is verified with another `set` command, which results in an error message indicating that the variable does not exist.

The arrays provided for by Tcl are called associative arrays since the indices to an array may be an arbitrary string. An array is a collection of elements of which each is a variable itself. The first line in Figure B.8 illustrates how an associative array is created (unless, of course, it already exists, in which case the figure illustrates how such an array is modified). The name of an array may be any string, in this case it is `theArray`. The two elements in the figure, `Var1` and `Var2`, has the values `10` and `SVENSK`, respectively. The commands introduced earlier in this section, `set`, `incr` and `unset`, applies to arrays in the same way as they do to simple variables.

B.3 Control flow

Tcl provides a number of different control mechanisms for controlling the flow of execution in a script, i.e. the conditional construct `if` and the looping commands `while`, `for` and `foreach`. Two commands for loop control are available, `break` and `continue`.

```
set theArray(Var1) 10
⇒ 10
set theArray(Var2) SVENSK
⇒ SVENSK
set theArray(Var1)
⇒ 10
```

Figure B.8: Associative arrays in Tcl.

The `if` command tests an expression and executes one of two scripts based on the result of the test. The syntax of the `if` command is shown in Figure B.9. If `<test1>` evaluates to a value other than zero, `<body1>` is executed. Otherwise, if `<test2>` is evaluated to non-zero, `<body2>` is executed. If none of `<test1>` or `<test2>` evaluates to non-zero, `<body3>` is executed. The tests are evaluated as expressions and the bodies as Tcl scripts. The `if` command returns as result the value obtained from executing one of the bodies. Either, or both, of the `elseif` in line 3 and `else` in line 5, with associated bodies, can be omitted. Due to the design of the Tcl interpreter (valid Tcl syntax is described by a set of procedures rather than, as might be expected, by a set of grammar rules, see [Ousterhout 1994, chapter 2]), each open brace ({), must be on the same line as the preceding word. Otherwise, the newline between the brace and the word will be treated as a command separator. Figure B.10 illustrates how *not* to use the `if` command. The newline character between lines 1 and 2 will cause the interpreter to split the code in Figure B.10 up in two commands, which is most probably not the intention of the programmer.

```
1.      if { <test1> } {
2.       <body1>
3.      } elseif { <test2> } {
4.       <body2>
5.      } else {
6.       <body3>
7.      }
```

Figure B.9: The syntax of the `if` command.

```
1.      if { $X < 0 }
2.      {
3.         set X 0
4.      }
```

Figure B.10: A non-valid `if` command.

There are three commands for looping in Tcl: `while`, `for` and `foreach`. The syntax for each command is illustrated in Figure B.11, Figure B.12 and Figure B.13, respectively. The `while` command in Figure B.11 evaluates the `<test>` expression and if the result is other than zero the Tcl script `<body>` is executed. This is repeated until `<test>` evaluates to zero. The `while` command then terminates and returns an empty string.

```
while { <test> } {
    <body>
}
```

Figure B.11: The syntax of the `while` command.

The `for` command in Figure B.12 takes four arguments: `<init>` which is a Tcl script for initialising relevant variables; `<test>` which is an expression that, if evaluated to non-zero, causes `<body>` to be executed; `<reinit>` which is a Tcl script for reinitialisation of variables and which is executed after `<body>` is run; and `<body>` which is a Tcl script. The execution of `<body>` is repeated until `<test>` becomes zero, the `for` command then returns an empty string.

```
for { <init> } { <test> } { <reinit> } {
    <body>
}
```

Figure B.12: The syntax of the `for` command.

The `foreach` command in Figure B.13 iterates over all elements in a valid Tcl list. It sets `<variable>` to each element in `<list>`, in order, and executes `<body>` for each value of `<variable>`. The `foreach` command returns an empty string.

```
foreach <variable> <list> {
    <body>
}
```

Figure B.13: The syntax of the `foreach` command.

There are two commands for loop control: `break` and `continue`. The `break` command causes the loop from which it is called to terminate immediately. Execution control then returns to the part of the Tcl script which initiated the loop. The `continue` command causes the current iteration of the loop from which the command was called to terminate. Looping then continues at next iteration. None of the looping control commands takes any arguments and they do not have return values.

B.4 Procedures

A *Tcl procedure* implements a *Tcl command.* Procedures may be defined within the scope of other procedures, making the newly defined procedure available only to procedures defined at the same level. However, there are ways to reach procedures defined outside the scope of the procedure in which the caller is defined (see the `upvar` and `uplevel` commands in [Ousterhout 1994, chapter 8]). This section introduces the Tcl commands `proc`, `return` and `global`.

The `proc` command in Figure B.14 is used to define Tcl procedures. It takes three arguments: `<name>` which is the name of the procedure, `<arglist>` which is a list in which each element is

```
proc <name> <arglist> <body>
return <options> <value>
global <var1> <var2> ... <varN>
```

Figure B.14: The syntax of the `proc`, `return` and `global` commands.

an argument to the new procedure, and `<body>` which is a valid Tcl script. The `proc` command returns an empty string. One important thing to notice is that the procedure name, `<name>`, overrides any existing procedures with the same name. Figure B.15 gives an example of the definition of a procedure called `printList` which takes one argument, `theList`, and prints each element in it.

```
proc printList { theList } {
    foreach Element $theList {
        puts $Element
    }
    return
}
```

Figure B.15: An example of a procedure definition.

The `return` command takes two optional arguments, see Figure B.14. It returns from the innermost nested procedure or `source` command (see Section B.6). The `<options>` argument can be used to trigger exceptions (see Section B.7) and `<value>` is the result of the procedure from which `return` returns. The default for `<value>` is an empty string, but it may be anything.

Variables used within a procedure are not the same as the ones the calling procedure has. Variables in a procedure are *local* to that procedure: they are available only for the procedure in which they are declared and they are deleted (deallocated) when the procedure terminates. Variables referenced outside a procedure definition are said to be *global*. Initially, when a procedure is about to start processing, i.e. just after it has been called, the only local variables with values are those which are given as arguments to the procedure. The `global` command, introduced in Figure B.14, is used to enable a procedure to access global variables not given as arguments.

Consider the example in Figure B.16. In line 1, two variables, `a` and `b`, are set to 4 and 5, respectively. In lines 2 to 5, a procedure called `addGlobals` is defined, the purpose of which is to add the values of `a` and `b` and to return the result of the operation to the caller. To provide for the procedure to access `a` and `b`, the `global` command is used (line 3). Command substitution (Section B.1) is used together with the `expr` command (see [Ousterhout 1994, chapter 5]) to calculate and return the sum of `a` and `b`. The result from invoking the procedure in Figure B.16 can be seen in Figure B.17.

```
set a 4; set b 5
proc addGlobals {} {
    global a b
    return [expr $a + $b]
}
```

Figure B.16: An example of the `global` and the `return` commands.

```
addGlobals
⇒ 9
```

Figure B.17: Invoking the procedure from Figure B.16.

B.5 List and string manipulation

The basic structure of a *list* is illustrated in Figure B.18. In its simplest form, a Tcl list is just a string containing any number of items separated by whitespace characters such as spaces or tabs. The list in Figure B.18 contains three elements, `SVENSK`, `GATE` and `Tcl`.

```
SVENSK GATE Tcl
```

Figure B.18: A simple list structure.

The `list` and `concat` commands can be used for creating lists. The `list` command in Figure B.19 takes any number of arguments and it joins them together so that each argument becomes a distinct element in the list that is returned by `list`.

```
list <value1> <value2> ... <valueN>
concat <list1> <list2> ... <listN>
llength <list>
lappend <var> <value1> <value2> ... <valueN>
split <string> <splitchars>
join <list> <joinstring>
```

Figure B.19: The `list`, `concat`, `llength`, `lappend`, `split` and `join` commands.

Figure B.20 illustrates how a list is created using the `list` command. As can be seen from the figure, the elements in the returned list are as distinct as they were when `list` was called.

The syntax of the `concat` command is illustrated in Figure B.19. Unlike `list`, the `concat` command does not treat the arguments as being distinct elements of a list, but rather expects the arguments to be well formed Tcl lists. Hence, it joins all of the arguments together into one large list as illustrated in Figure B.21.

```
list {SVENSK GATE} Tcl {TIPSTER}
⇒ {SVENSK GATE} Tcl TIPSTER
```

Figure B.20: Creating a list using the `list` command.

```
concat {SVENSK GATE} Tcl {TIPSTER}
⇒ SVENSK GATE Tcl TIPSTER
```

Figure B.21: Creating a list using the `concat` command.

```
llength {SVENSK GATE} Tcl {TIPSTER}
⇒ 3
llength S
⇒ 1
llength {}
⇒ 0
```

Figure B.22: The `llength` command.

```
set A [list a b {c d}]
⇒ a b {c d}
lappend A {b c} e
⇒ a b {c d} {b c} e
```

Figure B.23: The use of the `lappend` command.

The `llength` command (see Figure B.19) takes a Tcl list as argument and returns the the number of elements present in it. As illustrated in Figure B.22, a string one character long is a well formed Tcl list containing one element. The empty string corresponds to the empty list.

There are a number of commands for modifying lists (i.e. `linsert`, `lreplace` and `lrange`, see [Ousterhout 1994, chapter 6]), but this section only introduces the `lappend` command, see Figure B.19. It appends `<value1> - <valueN>` to the variable `<var>` as list elements and returns the new value of `<var>`, which is created if it does not already exist. Figure B.23 shows an example of `lappend`.

There are two commands for converting between strings and lists: `split` and `join`, see Figure B.19. The `split` command takes a string and a set of split-characters as arguments and returns the string as a list in which the elements are the the pieces of the string after breaking it up over the split-characters. How the `split` command works is illustrated in Figure B.24. Note that the split-characters themselves are discarded. The `join` command works the opposite way, it forms a string by joining the elements of a list together. It takes two arguments, a list and a

```
set Var x/y/z
⇒ x/y/z
split $Var /
⇒ x y z
```

Figure B.24: The use of the `split` command.

```
set L [list a b c d]
⇒ a b c d
join $L +
⇒ a+b+c+d
```

Figure B.25: The use of the `join` command.

string which is used to join the elements together, and returns the new string as illustrated by the example in Figure B.25.

As in the case of Tcl lists described above, there are a number of Tcl commands for manipulating strings. Among other things, there are two kinds of pattern matching available: glob-style and regular expressions, of which the latter is introduced in this section (see [Ousterhout 1994, chapter 9] for information about glob-style pattern matching). The syntax of the `regexp` com-

```
regexp -indices -nocase -- <exp> <str> <mVar> <sVar1> ... <sVarN>
regsub -all -nocase -- <expression> <string> <subSpec> <var>
format <formatString> <var1> <var2> ... <varN>
```

Figure B.26: The `regexp`, `regsub` and `format` commands.

mand, which is used for regular expressions, is illustrated in Figure B.26. The options `-indices` and `-nocase` are used in order to get the answer from the operation in terms of indices in the string, `<str>`, and to allow the matching to be case insensitive, respectively. The `--` option is used to tell the Tcl interpreter that the next argument should be treated as being a `<exp>`, even if it starts with a `-`. The `regexp` command determines whether the regular expression, `<exp>`, matches part of or all of string, `<str>` and it returns 0 if there is no match and 1 otherwise. The portions of the string, `<str>`, that matches the expression, `<exp>`, are placed in the variables `<mVar>` and `<sVar1>` - `<sVarN>`, if present. As an example of the use of `regexp`, a Tcl command used for extracting the starting and ending byte offsets as well as the sentence from a line in the output from the sentence splitter is shown in Figure B.27. The notion of regular expressions is almost a science of its own and is not elaborated on here, see [Ousterhout 1994, chapter 9] for more information. In the example in Figure B.27, `$Line` is a string conforming to the format that the sentence splitter produces (see Section 3.2.3). The variables `Trash`, `Start`, `End` and `Sentence` are used to store the parts of `$Line` that matched the regular expression { `*([0-9]+) *([0-9]+) *(.+)`}. When the `regexp` command has been executed, `Trash` stores the whole line, `Start` and `End` holds the starting and ending byte offsets, respectively, and the `Sentence` variable holds the part of `Line` constituting the sentence itself.

```
set Line "0 14 This is a test"
⇒ 0 14 This is a test
regexp { *([0-9]+) *([0-9]+) *(.+)} $Line Trash Start End Sentence
⇒ 1
puts $Trash
⇒ 0 14 This is a test
puts $Start
⇒ 0
puts $End
⇒ 14
puts $Sentence
⇒ This is a test
```

Figure B.27: The use of the `regexp` command.

Regular expressions can be used for substituting a whole string or a part thereof. The command implementing this feature is `regsub`. It resembles the `regexp` command in its arguments structure, illustrated in Figure B.26. The `-nocase` option have the same function as in the case of the `regexp` command while `-all` is used if all parts of the string, `<string>`, that matches the regular expression, `<expression>`, are to be substituted by the string, `<subSpec>`, before the result is copied as a string to `<var>`. The `--` argument has the same function as in the case of the `regexp` command. The example in Figure B.28 shows a `regsub` command for imploding consecutive whitespace characters in a string. The command is taken from the wrapper code for integrating the Brill tagger for Swedish described in chapter 4.

```
set Line "This    is a   test"
⇒ This    is a   test
regsub -all { +} $InLine " " OutLine
⇒ 3
puts $OutLine
⇒ This is a test
```

Figure B.28: A `regsub` command for imploding consecutive whitespace characters in a string.

Strings can be generated using the `format` command (and they can be parsed with the `scan` command, see [Ousterhout 1994, chapter 9]). The syntax of `format` is illustrated in Figure B.26. The command returns a result equal to the string, `<formatString>`, when the values of the variables, `<var1>` - `<varN>`, have been substituted in place of % -sequences in `<formatString>`. An example of how the `format` command can be used is given in Figure B.29.

B.6 File I/O and external processes

This section introduces the basics of how to read from and write to files as well as how to invoke external processes.

```
set Name Jonna
⇒ Jonna
set Age 1
⇒ 1
set Msg [format "%s is %d year old" $Name $Age]
⇒ Jonna is 1 year old
```

Figure B.29: Using the `format` command to generate a string.

```
open <name> <access>
close <fileId>
gets <fileId> <var>
puts -nonewline <fileId> <string>
```

Figure B.30: The `open`, `close`, `gets` and `puts` commands.

Files are opened and closed with the `open` and `close` commands, respectively. The syntax of `open` is illustrated in Figure B.30, where `<name>` is the path to the file to be opened and `<access>` is the method of access, e.g. that the file is opened for reading only. There are a number of different access methods (see [Ousterhout 1994, chapter 10]), of which the `r` and `w` methods are the only ones employed by the wrappers implemented in this thesis. The `r` access method is used to open a file for reading only (which is the default for `open`), the file to be opened must already exist. The `w` access method is used to open a file for writing only. The file is truncated (overwritten) if it already exists and created otherwise. The `open` command returns a file identifier to be used in conjunction with other file handling commands such as `close`, `gets` and `puts`. Figure B.31 shows how the file `fredriksFile` is opened for writing only and how `open` returns the identifier `file4` as a so called file handle. The syntax of the `close` command is shown in Figure B.30. It takes a file identifier as a single argument and returns an empty string.

```
set File [open fredriksFile w]
⇒ file4
```

Figure B.31: Opening the file `fredriksFile` for writing only.

The `gets` command is used to read from files, the syntax is illustrated in Figure B.30. It takes two arguments, a file identifier, `<file>`, and a variable name, `<var>`, which is optional. The `gets` command reads the next line from `file`, discarding the terminating newline character. If `<var>` is specified, then the current line is stored in it and the command returns the number of characters present in the line (or -1 for end of file). Otherwise, if `<var>` is not present, the `gets` command returns the line (or an empty string for end of file).

The `puts` command is used to write to a file one line at a time. The syntax is illustrated in Figure B.30. It takes three arguments, `-nonewline`, `<fileId>` and `<string>`, of which the latter is the only compulsory one. `puts` writes `<string>` to the file associated to `<fileId>`, if specified (the standard output channel is the default value), appending a newline character to the line unless the `-nonewline` option has been used. The return value is an empty string.

```
exec -keepnewline -- <arg1> <arg2> ... <argN>
eval <arg1> ... <argN>
source <file>
```

Figure B.32: The `exec`, `eval` and `source` commands.

New processes can be created by using the `exec` command, which syntax is illustrated in Figure B.32. It executes the commands specified in `<arg1>` - `<argN>` and waits until they have completed processing before returning the results to the standard output, or to the caller as an empty string if the output has been redirected. The trailing newline character is discarded unless the `-keepnewline` option is specified. The `--` argument has the same function as in the case of the `regexp` command in Section B.5. For information about how to redirect the output and create background processes, see [Ousterhout 1994, chapter 11]. An example of the `exec` command is given in Figure B.33 where a word count facility, `wc`, is invoked by `exec` on the file `fredriksFile`. The return value from the `wc` command is returned to the caller by `exec`.

```
exec wc fredriksFile
⇒ 4     38     210 fredriksFile
```

Figure B.33: Using `exec` to invoke a word count facility as an external process.

A command that may come in handy when external processes is to be created by the `exec` command is `eval`. It is a building block for creating and executing Tcl scripts. The syntax is illustrated in Figure B.32; it takes any number of arguments, concatenates them with white spaces as separators and then evaluates the result as a Tcl script. The result of the evaluation is returned by `eval`.

The `source` command reads a file and executes it as a Tcl script. It takes a single argument, as can be seen in Figure B.32, and returns the value returned from the Tcl script in the file, `<file>`. Also, `eval` allows for procedures in the file making up the Tcl script to use the `return` command to terminate the processing of the file.

B.7 Errors and exceptions

An error in a Tcl script is a special case of a set of events called exceptions. The cause of an exception may be a number of things, for instance a call to a non-existing Tcl command in a Tcl script, or an attempt to index out of bounds in an array. To facilitate for error and exception handling in Tcl, there are commands for generating errors as well as for trapping them. The commands referred to are `return`, `error` and `catch`, see Figure B.34 (also, see [Ousterhout 1994, chapter 12] for more information about exceptional events in Tcl).

The `error` command should be used only in situations where the only correct action is to abort the script being executed. If this is not the case, the commands `return` and `catch` should be used instead. How these commands function is best illustrated with the help of examples. The procedure `uppbrill_recordOutput` from the wrapper integrating the Brill tagger for Swedish in GATE (described in chapter 4) uses the Tcl code shown in Figure B.35 to return an error to

```
return -code <code> -errorinfo <info> -errorcode <code> <string>
catch <command> <var>
error <message> <info> <code>
```

Figure B.34: The `return`, `catch` and `error` commands.

its caller if the processing of the output produced by the Brill tagger fails. The `-code` option can assume one of several values: `ok`, `error`, `return`, `break`, `continue` or an integer. The `-errorinfo` and `-errorcode` options are used to initialise global variables called `errorInfo` and `errorCode`, respectively, which are used to store information about the state of the processing the script was in when the error occurred.

```
1.      return  -code error\
2.          -errorinfo $errorInfo\
3.          -errorcode $errorCode\
4.          "There wasn't enough data produced by the\
5.          Uppsala Brill Tagger to create attributes\
6.          for the existing token annotations."
```

Figure B.35: Using the `return` and `error` commands to generate and return an error to the calling procedure.

```
set Code [catch {uppbrill_recordOutput $doc $BrillResult} Msg]
```

Figure B.36: Using the `catch` command to trap possible errors from returned from the `uppbrill_recordOutput` procedure.

Figure B.36 illustrates how `catch` is used in the wrapper integrating the Brill tagger to catch whatever is returned by the `uppbrill_recordOutput` procedure. The variable `Code` in the figure will hold the value returned from the called procedure and the variable `Msg` will contain the message returned in lines 4 to 6 in Figure B.35. `Code` can then be used to test the result of the processing performed by `uppbrill_recordOutput`, and if an error is discovered, the contents of `Msg` can be issued to the user as an error message.

Appendix C

The GDM Tcl API

This appendix contains a listing (see Table C.1) of all the TIPSTER methods as employed by and implemented in the GDM. Note that the implemented methods may vary from version to version of GATE. This particular listing is extracted from version 1.0.3. Refer to [Grishman *et al* 1997, appendix B] for information about argument structures and return values for these commands.

`tip_AddAnnotation`	`tip_GetName`
`tip_Annotate`	`tip_GetOwner`
`tip_AnnotateCollection`	`tip_GetParent`
`tip_AnnotationsAt`	`tip_GetRawData`
`tip_Close`	`tip_GetSpans`
`tip_CreateAnnotation`	`tip_GetStart`
`tip_CreateAnnotationSet`	`tip_GetType`
`tip_CreateAttribute`	`tip_GetValue`
`tip_CreateAttributeValue`	`tip_Length`
`tip_CreateCollection`	`tip_MergeAnnotations`
`tip_CreateDocument`	`tip_NextAnnotations`
`tip_CreateSpan`	`tip_NextDocument`
`tip_DeleteAnnotations`	`tip_Nth`
`tip_Destroy`	`tip_OpenCollection`
`tip_FirstDocument`	`tip_PutAttribute`
`tip_GetAnnotations`	`tip_ReadSGML`
`tip_GetAttribute`	`tip_RemoveAnnotation`
`tip_GetAttributes`	`tip_RemoveAttribute`
`tip_GetByExternalId`	`tip_SelectAnnotations`
`tip_GetByteSequence`	`tip_SetExternalId`
`tip_GetDocument`	`tip_SetOwner`
`tip_GetEnd`	`tip_Sync`
`tip_GetExternalId`	`tip_test`
`tip_GetId`	`tip_WriteSGML`

Table C.1: The TIPSTER methods available in the GDM Tcl API in GATE 1.0.3.

Appendix D

Source code

This appendix contains the source code of two of the wrappers[1] for the NL components dealt with in this thesis — the one for the tokeniser (in Section D.1) and the one for UCP (in Section D.2) — together with the source code for the `single_span_ambiguities`-viewer (in Section D.3). The purpose of the wrapper listings is to illustrate two extremes in terms of wrapper length and complexity: the wrapper integrating the tokeniser is short and not very complex, while the one for UCP is an example of the opposite.

D.1 The wrapper for the tokeniser for Swedish

```
# upptoken.tcl --
#
#       This file implements the GATE 'wrapper' code for the Uppsala Tokenizer.
#
# This CREOLE module was created by Fredrik Olsson, fredriko@stp.ling.uu.se, as
# a part of his M.A. thesis "Tagging and Morphological Processing in the SVENSK
# System", which was carried out within the framework of the SVENSK Project at
# the Swedish Institute of Computer Science (SICS) and at the department of
# linguistics, university of Uppsala during the preiod April - October, 1997.


################################################################################
#
# upptoken.tcl - Mon Jun 16 11:11:01 DFT 1997
#
# To create a new module provide some code in the function
# creole_upptoken below.
#
# $Id: template_wrapper.tcl,v 1.4 1996/11/12 12:22:23 hamish Exp $
#
################################################################################

# Location of this wrapper

set upptoken_home /home/staff/gateuser/creole/upptoken
```

[1] Wrapper in this context corresponds to the contents of the moduleName.tcl template introduced in Section 5.3.

```
# upptoken_prepareInput --
#
#       Retrieves information from the GDM and formats it to meet the
#       requirements the Uppsala Tokenizer poses on its input.
#
# Arguments:
#       doc   The currently open GDM document.
#       file  The file in which the formatted information from doc will be
#             placed.
#
# Results:
#        The procedure creates a file which contains input data to the Uppsala
#        Tokenizer. The procedure does not return anything.

proc upptoken_prepareInput { doc file } {
    set Text [tip_GetByteSequence $doc]
    set F [open $file w]
    puts $F $Text
    close $F
    return
}

# upptoken_recordOutput --
#
#       Formats the output produced by the Uppsala Tokenizer and records the
#       information in the GDM.
#
# Arguments:
#       doc   The currently open GDM document.
#       file  The file in which the output produced by the Uppsala Tokenizer
#             resides.
#
# Results:
#       Information created by the Uppsala Tokenizer is recorded in the GDM.
#       The procedure does not return anything.

proc upptoken_recordOutput { doc file } {

    # Read the file line by line and extract the byte offsets from it. Then
    # create token annotations and add them to the GDM.

    set F [open $file r]
    while {[gets $F Line] >= 0} {

# The format of the input line depends on whether the tokenizer has
# found a sequence of multiple newline characters or not.

# If the line is on the format <int> <int> <string> then <string>
# is a token and should be recorded as such in the GDM.

if {[regexp { *([0-9]+) +([0-9]+) +(.+)} \
$Line Trash Start End Token]} {

    # Create a token attribute, tokenVal, which has as value the token
    # string itself. Use the attribute to create a token annotation,
    # which is then added to the GDM.

    set Token_attr [tip_CreateAttribute tokenVal \
    [tip_CreateAttributeValue GDM_STRING $Token]]
    set Annot [tip_CreateAnnotation token \
    [list [tip_CreateSpan $Start $End]] [list $Token_attr]]
    tip_AddAnnotation $doc $Annot

    # If the line is on the format <multipleNewline> <int> <int> then
    # the tokenizer has found consecutive newline characters in its
    # input. Record this in the GDM as an annotation without an
    # attribute.

} elseif {[regexp { *(multiNl) +([0-9]+) +([0-9]+)} \
$Line Trash String Start End]} {
    set MnAnnot [tip_CreateAnnotation multiNl \
    [list [tip_CreateSpan $Start $End]] {}]
    tip_AddAnnotation $doc $MnAnnot
}
    }

    close $F
    return
}

# creole_upptoken --
#
#       Implements the Tcl procedure that serves as a GATE wrapper for the
#       Uppsala Tokenizer.
#
# Arguments:
#       doc   The currently open GDM document.
#       args  User specified arguments specified by the user via the GGI. Empty
#             in the case of this particular procedure.
#
# Results:
#       The result is that of the Uppsala Tokenizer being run taking the
#       currently open GDM document into consideration and the information
#       produced by the tokenizer being recorded in the GDM. The procedure does
#       not return anything.

proc creole_upptoken { doc args } {

    # Use the env-array to get the path to the home directory of the current
    # user.

    global env
    set UserHome $env(HOME)

    # The path to the directory which holds the executable program.

    set UpptokenDir /home/staff/gateuser/Upptoken

    # Temporary files.

    set IntermediateFile ${UserHome}/upptokenIntermediate.tmp
    set FinalFile ${UserHome}/upptokenFinal.tmp

    # Prepare the input, run the tokenizer and record its output.

    upptoken_prepareInput $doc $IntermediateFile
    exec $UpptokenDir/upptoken < $IntermediateFile > $FinalFile
    upptoken_recordOutput $doc $FinalFile

    # Remove temporary files.

    eval exec rm $IntermediateFile $FinalFile

    # Record the fact that we ran and exit normally. Add a document attribute
    # to signal that the document was treated as if it was in Swedish.

    tip_PutAttribute $doc {upptoken {GDM_STRING "language_swedish"}}
    return
}

################################################################################
#
# $Log: template_wrapper.tcl,v $
# Revision 1.4  1996/11/12 12:22:23  hamish
# added module paths
#
# Revision 1.3  1996/11/05  12:15:09  hamish
# CVS keywords in (again)
#
################################################################################

#
# End of upptoken.tcl
#
```

D.2 The wrapper for UCP

```
# uppcp.tcl --
#
#       This file implements the GATE 'wrapper' code for the Uppsala Chart
#       Processor.
#
# This CREOLE module was created by Fredrik Olsson, fredriko@stp.ling.uu.se, as
# a part of his M.A. thesis "Tagging and Morphological Processing in the SVENSK
# System", carried out within the framework of the SVENSK Project at the Swedish
# Institute of Computer Science (SICS) and at the department of linguistics at
# Uppsala University during the period April - October, 1997.

################################################################################
#
#        uppcp.tcl - Tue Aug 12 17:01:55 DFT 1997
#
#        To create a new module provide some code in the function
#        creole_uppcp below.
#
#        $Id: template_wrapper.tcl,v 1.4 1996/11/12 12:22:23 hamish Exp $
#
################################################################################

# Location of this wrapper.

set uppcp_home /home/staff/gateuser/creole/uppcp

# uppcp_prepareInput --
#
#       Retrieves information from the GDM and formats it to meet the
#       requirements the Uppsala Chart Processor poses on its input. Any values
#       on the tokenVal attributes associated with token annotations that
#       follows .! or ? in the original input text, are converted to lower case
#       since the lexicon of the UCP does not include capitalized words. The
#       procedure also takes into account any multiNl annotations that might be
#       present in the GDM in order to decide on whether a token should be
#       convertwed to lower case or not.
#
# Arguments:
#       doc   The currently open GDM document.
#       file  The file in which the formatted information from doc will be
#             placed.
#
# Results:
#      The procedure creates a file which contains input data to the Uppsala
#      Chart Processor. The procedure does not return anything.

proc uppcp_prepareInput { doc file } {

    set F [open $file w]

    # Get the set of token annotations and the (possibly empty) set of
    # multiNl annotations (mulitNl = multiple newline characters).

    set TokenAnnotations [tip_SelectAnnotations $doc token {}]
set MultiNlAnnotations [tip_SelectAnnotations $doc multiNl {}]

# Decide on upper bound for indexing in the set of multiNl annotations. A
# negative value is used if there are no annotations.

set MultiNlCount 0
if {[llength $MultiNlAnnotations] > 1} {
    set MultiNlLastIdx [expr [llength $MultiNlAnnotations] - 1]
} else {
    set MultiNlLastIdx -1
}

# Get the value of the starting and ending byte offsets for the current
# mulitNl annotation. If no multiNl annotations exists for the current
# document, make sure the values used on MultiNL-variables do not interfere
# with the ones of the token annotations.

if {$MultiNlCount <= $MultiNlLastIdx} {
    set MultiNlSpan [lindex [tip_GetSpans \
            [tip_Nth $MultiNlAnnotations $MultiNlCount]] 0]
    set MultiNlStart [tip_GetStart $MultiNlSpan]
    set MultiNlEnd [tip_GetEnd $MultiNlSpan]
    incr MultiNlCount
} else {
    set LastTokenAnnotation [tip_Nth $TokenAnnotations \
            [expr [llength $TokenAnnotations] - 1]]
    set LastTokenSpan [lindex [tip_GetSpans $LastTokenAnnotation] 0]

    # This way, the starting and ending byte offsets for the (non-existant)
    # set of multiNl annotations get larger values than those of the last
    # token annotation.

    set MultiNlStart [expr [tip_GetStart $LastTokenSpan] + 1]
    set MultiNlEnd [expr [tip_GetEnd $LastTokenSpan] + 1]
}

# We need to keep track of the type of the previously seen annotation as
# well as of the value of the previous token attribute.

set PreviousAnnotation ""
set PrevTokenAttribute ""

# The basic idea is to compare starting values for the current token
# annotation and the current multiNl annotation as well as using information
# about the previously seen token in order to know when a token should be
# converted to lower case and printed and when it is just to be printed.

foreach TokenAnnotation $TokenAnnotations {

    set TokenSpan [lindex [tip_GetSpans $TokenAnnotation] 0]
    set TokenStart [tip_GetStart $TokenSpan]
    set TokenEnd [tip_GetEnd $TokenSpan]
    set TokenAttribute [tip_GetValue \
            [tip_GetAttribute $TokenAnnotation tokenVal]]

    # This case only applies if the current token annotation is the first
# one; it is retrieved and converted to lower case, since it is assumed
# to be the starting token of a sentence.

if {[expr {$PreviousAnnotation == ""} \
        && {$PrevTokenAttribute == ""}]} {
    puts $F "[uppcp_toLowerCase $TokenAttribute]"
    set PreviousAnnotation "token"
    set PrevTokenAttribute $TokenAttribute
    continue;
}

# The current type of annotation is token.

if { $TokenStart <= $MultiNlStart } {

    # If any sentence delimiter is matched, convert the current value
    # of the token to lower case and print it to the output file.

    if {[string match \. $PrevTokenAttribute]} {
        puts $F "[uppcp_toLowerCase $TokenAttribute]"
    } elseif {[string match \! $PrevTokenAttribute]} {
        puts $F "[uppcp_toLowerCase $TokenAttribute]"
    } elseif {[string match \\? $PrevTokenAttribute]} {
        puts $F "[uppcp_toLowerCase $TokenAttribute]"
    } else {
        puts $F "$TokenAttribute"
    }

    # Update the variables holding information about the previously seen
    # annotaion and the value of the previous token attibute.

    set PreviousAnnotation "token"
    set PrevTokenAttribute $TokenAttribute

# The current type of annotation is a multiNl.

} else {

    # Convert whatever follows after one or more empty lines in the
    # original input file to lower case.

    puts $F "[uppcp_toLowerCase $TokenAttribute]"

    # Update the variables holding information about the previously seen
    # annotaion and the value of the previous token attibute.

    set PreviousAnnotation "multiNl"
    set PrevTokenAttribute ""

    # Proceed with the next multiNl annotation.

    if {$MultiNlCount <= $MultiNlLastIdx} {
        set MultiNlSpan [lindex [tip_GetSpans \
                [tip_Nth $MultiNlAnnotations $MultiNlCount]] 0]
        set MultiNlStart [tip_GetStart $MultiNlSpan]
        set MultiNlEnd [tip_GetEnd $MultiNlSpan]
```

```
                incr MultiNlCount
            }
        }
    }
    close $F
    return
}

# uppcp_toLowerCase --
#
#       The procedure converts a string to lower case. This procedure handles
#       the cases which the built in 'string tolower" cannot handle, that is,
#       the characters corresponding to those in the ASCII table for the ISO
#       8859-1 character set that have a decimal value between (and including)
#       224 and 255.
#
# Arguments:
#       str  Is the string to be converted.
#
# Results:
#       The procedure returns a string which is the input string where upper
#       case characters have been converted to lower case.

proc uppcp_toLowerCase { str } {

    # toASCII --
    #
    #       This procedure is private to the uppcp_toLowerCase-procedure.
    #       It converts a character to its corresponding decimal value.
    #
    # Arguments:
    #       char  Is the character to be converted.
    #
    # Restults:
    #       The procedure returns an integer corresponding to the decminal value
    #       of the characters position in the ASCII table.

    proc toASCII { char } {
        scan $char %c value
        return $value
    }


    # toChar --
    #
    #       This procedure is private to the uppcp_toLowerCase-procedure.
    #       It converts an integer to a character. No control is made to
    #       ensure that the value corresponds to a printable ASCII character.
    #
    # Arguments:
    #       value  Is the value to be converted to a character.
    #
    # Results:
    #       The procedure returns a character corresponding to the input value.

    proc toChar { value } {
        return [format %c $value]
    }

    set Result {}

    # Convert the string to a list and check each element in it. As long as
    # an element has a decimal value corresponding to an upper case letter,
    # 32 is added to the value before it is converted back to a character.

    foreach Element [split $str {}] {
        set Value [toASCII $Element]
        if {($Value >= 65 && $Value <= 90) || \
                ($Value >= 192 && $Value <= 221)} {
            lappend Result [toChar [expr $Value + 32]]
        } else {
            lappend Result $Element
        }
    }

    # Return the result as a string rather than as a list.

    return [join $Result {}]
}

# uppcp_dialog --
#
#       The procedure generates various dialog boxes based on its input data. It
#       is originally named "dialog" and is taken from chapter 27 in the book
#       "Tcl and the Tk Toolkit" by John K. Ousterhout.
#
# Arguments:
#       w        Is the path to the window.
#       title    Is the title shown at the top of the window.
#       text     Is the message displayed in the window.
#       bitmap   Is the name of, or path to, a bitmapped picture to display. If
#                left empty, no picture appears in the window.
#       default  Is an integer representing the default button in the window. 0
#                denotes the left-most button and a negative value indicates that
#                no default button is desired.
#       args     Is one or more text strings which are used as labels for the
#                buttons in the window.
#
# Results:
#       The procedure returns the name of the button clicked by the user.

proc uppcp_dialog {w title text bitmap default args} {
    global button

    # 1. Create the top-level window and divide it into top and bottom parts.

    toplevel $w -class Dialog
    wm title $w $title
    wm iconname $w Dialog
    frame $w.top -relief raised -bd 1
    pack $w.top -side top -fill both
    frame $w.bot -relief raised -bd 1
    pack $w.bot -side bottom -fill both

    # 2. Fill the top part with the bitmap and message.

    message $w.top.msg -width 3i -text $text \
            -font -Adobe-Times-Medium-R-Normal-*-180-*
    pack $w.top.msg -side right -expand 1 -fill both -padx 3m -pady 3m
    if {$bitmap !=""} {
        label $w.top.bitmap -bitmap $bitmap
        pack $w.top.bitmap -side left -padx 3m -pady 3m
    }

    # 3. Create a row of buttons at the bottom of the dialog.

    set i 0
    foreach but $args {
        button $w.bot.button$i -text $but -command "set button $i"
        if {$i == $default} {
            frame $w.bot.default -relief sunken -bd 1
            raise $w.bot.button$i
            pack $w.bot.default -side left -expand 1 -padx 3m -pady 2m
            pack $w.bot.button$i -in $w.bot.default -side left \
                    -padx 2m -pady 2m -ipadx 2m -ipady 1m
        } else {
            pack $w.bot.button$i -side left -expand 1 \
                    -padx 3m -pady 3m -ipadx 2m -ipady 1m
        }
        incr i
    }

    # 4. Set up a binding for <Return>, if there's a default, set a grab, and
    # claim the focus too.

    if {$default >= 0} {
        bind $w <Return> "$w.bot.button$default flash; set button $default"
    }
    set oldFocus [focus]
    grab set $w
    focus $w

    # 5. Wait for the user to respond, then restore the focus and return the
    # index of the selected button.

    tkwait variable button
    destroy $w
    focus $oldFocus
    return $button
}

# uppcp_refineList --
#
#       This procedure is used by the procedure uppcp_file2list. It converts a
#       list containing a flattened UCP attribute-value structure to a list of
#       lists.
#
# Arguments:
```

```
#       list  Is a list which contains a flattened UCP attribute-value strucutre.
#
# Results:
#       The procedure returns a list where the first element is a list containing
#       the the word being analyzed and where the rest of the elements are lists,
#       each of which are holding an attribute-value pair.

proc uppcp_refineList { list } {

    # InteriorLinesList holds the attribute-value pairs found in the elements in
    # list representing the innermost lines in an UCP attribute-value structure.
    # BodyList holds the attribute-value pairs in InteriorLinesList and those
    # found at the first and last rows of an UCP attribute-value structure.

    set InteriorLinesList {}
    set BodyList {}

    # One of the following cases should apply to each element in the input list
    # in order discard unwanted syntactic items.

    foreach Element $list {

        # Get the word which appears on the form "word :"

        if {[regexp {^(.+):$} $Element Trash Word]} {

            # Get rid of the "(* = ( "construct.

        } elseif {[regexp {^\(\*=\((.+)$} $Element Trash FirstLine]} {

            # Get rid of the "))" construct. It's time to start preparing the
            # final list.

        } elseif {[regexp {^(.+)\)\)$} $Element Trash LastLine]} {

            lappend BodyList [concat $FirstLine $InteriorLinesList $LastLine]
            set FirstLine {}
            set InteriorLinesList {}
            set LastLine {}

            # This case serves as a default case; all "body" lines in an UCP
            # attribute-value structure is caught here.

        } else {
            lappend InteriorLinesList $Element
        }
    }
    return [concat [list $Word] $BodyList]
}

# uppcp_file2list --
#
#       The procedure converts a file containing morphological analyzes produced
#       by the Uppsala Chart Processor to a list of lists.
#
# Arguments:
```

```
#       file  Is a file holding morphological analyzes produced by the Uppsala
#             Chart Processor.
#
# Results:
#       The procedure returns a list of lists, where each list in turn consists
#       of lists containing the attribute-value pairs extracted from the output
#       from the Uppsala Chart Processor when invoked with morphological rules.
#       Thus, the list have three levels of nesting.

proc uppcp_file2list { file } {

    set F [open $file r]
    set FinalList {}
    set AttrValList_1 {}
    set AttrValList_2 {}

    while {[gets $F Line] >= 0} {

        # If we see an empty line and if we, at the same time, have a non-empty
        # list containing an UCP attribute-value strucutre, it's time to refine
        # that list and add it to the final output.

        if {[regexp {^ *$} $Line] && [expr [llength $AttrValList_1] > 0]} {
            set AttrValList_2 [uppcp_refineList $AttrValList_1]
            lappend FinalList $AttrValList_2
            set AttrValList_1 {}
            set AttrValList_2 {}

            # This case is just to skip all but the first of consecutive empty
            # lines (originating from the input file).

        } elseif {[regexp {^ *$} $Line] \
                && [expr [llength $AttrValList_1] <= 0]} {

            # All non-empty lines are added to a buffer list which is later
            # processed in the first if-statement.

        } else {
            regsub -all { } $Line {} refinedLine_1
            lappend AttrValList_1 $refinedLine_1
            }
    }
    close $F
    return $FinalList
}

# uppcp_recordOutput --
#
#       Formats the output produced by the Uppsala Chart Processor and records
#       the information in the GDM.
#
# Arguments:
#       doc   The currently open GDM document.
#       file  The file in which the information to be recorded in the GDM
#             resides.
#
```

```
# Results:
#       Information provided by the Uppsala Chart Processor is recorded in the
#       GDM. The procedure returns an error if something went wrong, otherwise
#       it returns nothing.

proc uppcp_recordOutput { doc file } {

    global errorInfo errorCode

    # Get a list containing the UCP analyzes.

    set UcpResult [uppcp_file2list $file]

    set TokenAnnotations [tip_SelectAnnotations $doc token {}]

    # Initiate counters, the first points to the current
    # position in the $UcpResult-list and the second one
    # points to the end of the same list.
    #

    # Counters for keeping track of the current position in the list of
    # analyzes produced by the UCP.

    set CurrIndex 0
    set LastIndex [llength $UcpResult]

    # The basic idea is to check for a match between the current token and the
    # first element in the current list of analyzes. If there is a match, add
    # some information to the GDM, otherwise, add an empty value and move on
    # to the next token value.

    foreach TokenAnnotation $TokenAnnotations {

        set TokenValue [tip_GetValue \
                [tip_GetAttribute $TokenAnnotation tokenVal]]
        set CurrUcpInput [lindex $UcpResult $CurrIndex]
        set Word [lindex $CurrUcpInput 0]
        set MorphValue [lrange $CurrUcpInput 1 end]

        # Check for a match between the current token and the current UCP
        # analyze.

        set LowWord [uppcp_toLowerCase $Word]
        set LowTokenValue [uppcp_toLowerCase $TokenValue]

        if {$LowWord == $LowTokenValue} {
            set MorphAttribute [tip_CreateAttribute morph \
                    [tip_CreateAttributeValue GDM_STRING $MorphValue]]
            tip_AddAnnotation $doc \
                    [tip_PutAttribute $TokenAnnotation $MorphAttribute]
            incr CurrIndex

        } else {
            set MorphAttribute [tip_CreateAttribute morph \
                    [tip_CreateAttributeValue GDM_STRING ""]]
            tip_AddAnnotation $doc \
                    [tip_PutAttribute $TokenAnnotation $MorphAttribute]
        }
    }

    # If there are analyses left when all token annotations have been taken into
    # consideration, an error has occurred.

    if {$CurrIndex < $LastIndex} {
        return -code error\
                -errorinfo $errorInfo\
                -errorcode $errorCode\
                "The output produced by the Uppsala Chart\
                Processor is corrupt OR the annotations\
                for $doc in the GDM are defect: UCP output\
                remains after all token annotations have been\
                treated"
    }
    return
}

# creole_uppcp --
#
#       Implements the Tcl prcedure that serves as a GATE wrapper for the Uppsala
#       Chart Processor.
#
# Arguments:
#       doc   The currently open GDM document.
#       args  User specified arguments (some with default values) passed to the
#             procedure via the GGI. In the case of the Uppsala Chart Processor,
#             the following arguments can be present in the args-list:
#
#                    ucpMorphology  Is the path to a memory dump produced by
#                                   clisp of a set of morphology rules written
#                                   in the UCP formalism. Default value exists.
#
#             The final argument that may appear is a path to a file in which
#             the raw UCP analyzes are stored.
#
# Results:
#       The result is that of the Uppsala Chart Processor being run taking the
#       currently open GDM document into consideration and the information
#       produced by the processor being recorded in the GDM. The procedure does
#       not return anything.

proc creole_uppcp { doc args } {

    # Use the env-array to get the path to the home directory of the current
    # user.

    global env
    set user_home $env(HOME)

    # Temporary files.

    set UcpInput ${user_home}/ucpInput.tmp
    set UcpFailures ${user_home}/ucpInput.0
set UcpParses ${user_home}/ucpInput.parses

# The path to the directory which holds the executable program.

set UcpDir /usr/local/ucp

# This is a flag used to decide whether the output file prdoced by the UCP
# should be removed or not. Default is to remove it.

set RemoveParseFile 1

# Get the (default) value for the path to the UCP morphology from the
# args-list.

set UcpMorphology [lindex $args 0]

# An additional value in the args-list affects the looks of the final command
# line to be passed to the UCP. In any case, the command line should have the
# load a dictionary and tell the UCP on what format the output should be.

if {[lindex $args 1] == ""} {

    # The user have not specified an output file.

    set LispCommands \
            [format  \
            "(progn (usegd) (try-file \"%s\" :report-style :parses))" \
            $UcpInput]
} else {

    # The user have specified an output file. Propagate this to the rest of
    # the program via the RemoveParseFile-flag.

    set LispCommands \
            [format \
            "(progn (usegd) (try-file \"%s\" \
            :report-style :parses :parse-file \"%s\"))" \
            $UcpInput [lindex $args 1]]
    set UcpParses [lindex $args 1]
    set RemoveParseFile 0
}

# This is where the actual processing starts. Retrieve
# information from the GDM and place it, suitably formatted,
# in a file.
#

# Prepare the input to the processor and place the result in a file.

uppcp_prepareInput $doc $UcpInput

# Additional information to the user in case of an error.

set ErrorString "No morph annotations present in the GDM."

# Invoke the ucp Perl-script with relevant parameters. Catch errors that the
# UCP may generate.

set Code [catch {eval exec \
        {${UcpDir}/ucp -m $UcpMorphology -x "$LispCommands"}} msg]

# If there was an error, issue an error message to the user and return to
# the caller.

if {$Code == 1} {
    set ErrorMessage [format "%s %s" $msg $ErrorString]
    uppcp_dialog .d {UCP Status} $ErrorMessage error 0 {Ok}
    return
}

# Record the output produced by the Uppsala Chart Processor in the GDM. The
# procedure uppcp_recordOutput throws an exception which we catch here.

set Code [catch {uppcp_recordOutput $doc $UcpParses} Msg]

# If an error has occurred, issue an error message to the user and remove
# the morph attributes present on the token annotations to keep the database
# consistent.

if {$Code == 1} {
    set ErrorMessage [format "%s %s" $Msg $ErrorString]
    uppcp_dialog .d {Error trying to record UCP output}\
            $ErrorMessage error 0 {Ok}
    set TokenAnnotations [tip_SelectAnnotations $doc token {}]
    foreach TokenAnnotation $TokenAnnotations {
        tip_AddAnnotation $doc [tip_RemoveAttribute $TokenAnnotation morph]
    }

    # Remove temporary files. Remove the raw UCP data file only if it is
    # flagged as unwanted.

    eval exec rm $UcpInput $UcpFailures
    if {$RemoveParseFile} {
        eval exec rm $UcpParses
    }

    # We didn't quite succeed in annotating the current $doc, but we'll
    # return to the caller anyway (for now). Without adding *any*
    # annotations to the GDM!!

    return
}

# Remove temporary files. Remove the raw UCP data file only if it is
# flagged as unwanted.

eval exec rm $UcpInput $UcpFailures
if {$RemoveParseFile} {
    eval exec rm $UcpParses
}

# Record the fact that we ran and exit normally. Add a document attribute
```

```
    # to signal that the document was treated as if it was in Swedish.

    tip_PutAttribute $doc {uppcp {GDM_STRING "language_swedish"}}
    return
}

################################################################################
#
# $Log: template_wrapper.tcl,v $
# Revision 1.4  1996/11/12 12:22:23  hamish
# added module paths
#
# Revision 1.3  1996/11/05  12:15:09  hamish
# CVS keywords in (again)
#
################################################################################

#
# End of uppcp.tcl
#
```

D.3 Source code for the single_span_ambiguities-viewer

```
# ggi_AmbiguityViewer.gw --
#
#       This file implements the Tcl code for a single span ambiguity viewer
#       intended to be used for displaying ambiguous attributes created by
#       CREOLE objects invoked from the GATE platform.
#
# This viewer was created by Fredrik Olsson, fredriko@stp.ling.uu.se, as a part
# of his M.A. thesis "Tagging and Morphological Processing in the SVENSK System",
# carried out within the framework of the SVENSK Project at the Swedish
# Institute of Computer Science (SICS) and at the department of linguistics at
# Uppsala University during the period April - October, 1997.
#
# The first version was created during the period 1997-09-01 to 1997-09-09.
#
# Henceforth, the string 'ssav' is an abbreviation for 'single span ambiguity
# viewer'. Any procedure name denoting a procedure specific to the single span
# ambiguity viewer is on the form "ggi_ssavXXX", where XXX is a text string.
# Other procedures, which could be of use in a more general perspective have
# names on the form "ggi_XXX".

# Global variables
#
#       All global variables used by this program are gathered in one ass-
#       ociative array, ggi_ssavGlobals. For clarity, all elements in the
#       array are set in the next few lines.

set ggi_ssavGlobals(passiveCheckedBg) blue
set ggi_ssavGlobals(passiveCheckedFg) white
set ggi_ssavGlobals(passiveBg) skyBlue
set ggi_ssavGlobals(passiveFg) black
set ggi_ssavGlobals(activeBg) red
set ggi_ssavGlobals(activeFg) white
set ggi_ssavGlobals(staticBg) ""
set ggi_ssavGlobals(staticFg) ""
set ggi_ssavGlobals(prevClickedAnnotation) ""
set ggi_ssavGlobals(annotationSet) ""
set ggi_ssavGlobals(attribute) ""
set ggi_ssavGlobals(sourceWin) ""
set ggi_ssavGlobals(targetWin) ""


# ggi_view_single_span_ambiguities --
#
#       Displays ambiguous GDM attributes using a hyper text approach. The
#       source document is displayed in one window and, when a piece of text is
#       clicked on, the attributes corresponding to that piece are displayed in
#       another window.
#
# Arguments:
#       document    The currently open GDM document.
#       annotation  The annotation type to be displyed.
#       attribute   The attribute which may have ambiguous (0 or more) values.
#       title       The title of the window holding the viewer.
#
```

```
# Results:
#       The result is that of the values of the specified attribute for the
#       specified annotation type being displayed to the user. The procedure
#       returns the path name of the window constituing the single span ambiguity
#       viewer.

proc ggi_view_single_span_ambiguities { document annotation attribute title } {

    global ggi_ssavGlobals

    # ExitSsav --
    #
    #       This procedure is private to "ggi_view_single_span_ambiguities".
    #       It resets some variables before exiting the ambiguity viewer.
    #
    # Arguments:
    #       win  Is the path to the ambiguity viewer window.
    #
    # Results:
    #       The variable holding the value of the last clicked annotation is
    #       reset and the window is destroyed.

    proc ExitSsav { win } {
        global ggi_ssavGlobals
        set ggi_ssavGlobals(prevClickedAnnotation) ""
        destroy $win
        return
    }

    set WaitWin [ggi_wait_message "$annotation annotations being loaded"]
    update

    set AnnotationSet [$document SelectAnnotations $annotation {}]
    set TextStr [$document GetByteSequence]

    # Store the set of annotations and the current kind of attribute as global
    # variables for later use.

    set ggi_ssavGlobals(annotationSet) $AnnotationSet
    set ggi_ssavGlobals(attribute) $attribute

    # Create a top level window containing three frames; two text widgets and a
    # button. Use a unique integer in combination with the window name.

    set win [gu_gensym .w]
    toplevel $win
    wm title $win $title
    wm iconname $win $title

    frame $win.buttons
    pack $win.buttons -side bottom
    button $win.buttons.dismiss -text "Dismiss" \
            -command [list ExitSsav $win]
    pack $win.buttons.dismiss -padx 2 -pady 2

    frame $win.topFrame
    pack $win.topFrame -fill both -expand true
    text $win.topFrame.text -borderwidth 2 -relief sunken -wrap word \
            -setgrid true -yscrollcommand [list $win.topFrame.scroll set] \
            -width 80 -height 25

    scrollbar $win.topFrame.scroll -command [list $win.topFrame.text yview]
    pack $win.topFrame.scroll -side right -fill y
    pack $win.topFrame.text -side left -fill both -expand true

    frame $win.bottomFrame
    pack $win.bottomFrame -fill both -expand true
    text $win.bottomFrame.text -borderwidth 2 -relief sunken -wrap word \
            -setgrid true -yscrollcommand [list $win.bottomFrame.scroll set] \
            -width 80 -height 25
    scrollbar $win.bottomFrame.scroll -command [list $win.bottomFrame.text yview]
    pack $win.bottomFrame.scroll -side right -fill y
    pack $win.bottomFrame.text -side left -fill both -expand true

    # If the elements staticFg and/or staticBg are specified as global variables;
    # use those values to colour the background and foreground of the two text
    # widgets, otherwise use the defaults.

    set staticFg $ggi_ssavGlobals(staticFg)
    set staticBg $ggi_ssavGlobals(staticBg)
    if {$staticFg != ""} {
        $win.topFrame.text configure -fg $staticFg
        $win.bottomFrame.text configure -fg $staticFg
    }
    if {$staticBg != ""} {
        $win.topFrame.text configure -bg $staticBg
        $win.bottomFrame.text configure -bg $staticBg
    }

    # Store the paths to the two text widgets as global variables.

    set ggi_ssavGlobals(sourceWin) $win.topFrame.text
    set ggi_ssavGlobals(targetWin) $win.bottomFrame.text

    # Place the text in the top text widget. See the documentation of the
    # ggi_ssavPlaceText-procedure below!

    ggi_ssavPlaceText $win.topFrame.text $TextStr $AnnotationSet \
            $attribute $ggi_ssavGlobals(passiveBg) $ggi_ssavGlobals(passiveFg)

    ggi_destroy_wait $WaitWin

    # Insertion/deletion in any of the text widgets is not allowed.

    $win.topFrame.text configure -state disabled
    $win.bottomFrame.text configure -state disabled

    return $win

}

# ggi_ssavPlaceText --
```

```
#
#       Places a text in a text widget, also marks the span corresponding each
#       GDM annotation with a unique tag and binds it to an event (procedure).
#
# Arguments:
#       srcWin   Is the window in which the source text is displayed. The
#                variable must contain a path to a text widget!
#       textStr  Is the text string to be displayed in dstWin.
#       annSet   Is the set of annotations from which the values of the
#                attributes is extracted.
#       attr     Is the type of attribute the user wished to view.
#       tagBg    Is the background colour of the tagged areas in srcWin.
#       tagFg    Is the foreground colour of the tagged areas in srcWin.
#
# Results:
#       The result is that of the text being placed in a window and then
#       processed in a way necessary for the rest of the program. The procedure
#       returns nothing.

proc ggi_ssavPlaceText { srcWin textStr annSet attr tagBg tagFg } {

    $srcWin delete 1.0 end
    $srcWin insert end $textStr

    # The value of this counter will serve as a tag for the annotations' span
    # in the text widget.

    set AnnCounter 0

    # Tag each annotation with a unique number (which will later be used as an
    # index in the list of annotations).

    foreach Annotation $annSet {


        # Make sure we only tag the appropriate spans, i.e. those belonging to
        # annotations which have non-emtpy values on the attribute in question.

        set AttrVal [tip_GetValue [tip_GetAttribute $Annotation $attr]]
        if {$AttrVal == ""} {
            incr AnnCounter
            continue
        }

        set Start [tip_GetStart [lindex [tip_GetSpans $Annotation] 0]]
        set End [tip_GetEnd [lindex [tip_GetSpans $Annotation] 0]]

        # Convert the current byte offsets to text widget coordinates (i.e. to
        # line.character format).

        set LineCharList [ggi_byteOffset2LineChar $srcWin $Start $End]


        # Tag the current span with the value of the annotation counter.

        $srcWin tag add $AnnCounter [lindex $LineCharList 0] \
                [lindex $LineCharList 1]

        # Raise the priority of $AnnCounter tag to the highest possible to
        # ensure that the tag is the last in a list resulting from a "tag names"-
        # command performed later.

        $srcWin tag raise $AnnCounter

        # Bind the tag to a procedure.

        $srcWin tag bind $AnnCounter <Button-1> {
            ggi_ssavDisplayAttributes %x %y
        }

        # Set the appearance of the tagged span.

        $srcWin tag configure $AnnCounter -background $tagBg \
                -foreground $tagFg
        incr AnnCounter
    }
    return
}

# ggi_ssavDisplayAttributes --
#
#       The procedure displays the values of the specified attribute for the
#       current (clicked) annotation. Note that the passing of information to
#       procedure is done via global variables.
#
# Arguments:
#       x  Is the x-coordinate given by the user when clicking a tagged area
#          in the source window (top text widget).
#       y  Is the y-coordinate given by the user at the same time as the x-
#          coordinate.
#
# Results:
#       The result is that of all the values for a specific GDM attribute being
#       displayed in the bottom text widget. The procedure returns nothing.

proc ggi_ssavDisplayAttributes { x y } {

    global ggi_ssavGlobals

    set PreviousAnn $ggi_ssavGlobals(prevClickedAnnotation)

    set SrcWin $ggi_ssavGlobals(sourceWin)
    set DstWin $ggi_ssavGlobals(targetWin)

    $DstWin delete 1.0 end

    # Since the priority of the tag added to each span is the highest possible,
    # it should appear last in the list of tags produced by the "tag names"-
    # command. That way we get a hold of the value if the tag, which is also an
    # index to the list of attibutes relevant to this annotation set.

    set CurrTags [$SrcWin tag names [$SrcWin index @${x},${y}]]
    set CurrTag [lindex $CurrTags [expr [llength $CurrTags] - 1]]

    # Reconfigure the tag corresponding to the previously clicked area in the
    # source text as well as the appearance of the currently clicked one.

    if {$PreviousAnn != ""} {
        $SrcWin tag configure $PreviousAnn \
                -background $ggi_ssavGlobals(passiveCheckedBg)
        $SrcWin tag configure $PreviousAnn \
                -foreground $ggi_ssavGlobals(passiveCheckedFg)
    }

    $SrcWin tag configure $CurrTag -background $ggi_ssavGlobals(activeBg)
    $SrcWin tag configure $CurrTag -foreground $ggi_ssavGlobals(activeFg)

    # Update the global variable that holds the information about the previously
    # clicked annotation to point to the one just clicked.

    set ggi_ssavGlobals(prevClickedAnnotation) $CurrTag

    set AnnotationSet $ggi_ssavGlobals(annotationSet)
    set Attribute $ggi_ssavGlobals(attribute)


    # Get a list of relevant attribute values.

    set Ann [tip_Nth $AnnotationSet $CurrTag]
    set Attr [tip_GetAttribute $Ann $Attribute]
    set AttrVal [tip_GetValue $Attr]

    # Toggle the state of the bottom text widget to enable the program to output
    # information in it. The state is toggled once again when the information is
    # in place.

    $DstWin configure -state normal

    $DstWin  delete 1.0 end

    # Display the list of attribute-values in a human readable way. See the
    # documentation of the ggi_ssavDisplayNestedAttributes-procedure below!

    ggi_ssavDisplayNestedAttributes $DstWin $AttrVal

    $DstWin configure -state disabled

    return
}

# ggi_ssavDisplayNestedAttributes --
#
#       Displays attribute values that might be nested, i.e. list of lists, to
#       an arbitrary level. Note that the nesting of any element may not be more
#       than one level deeper than that of the first element in the (sub) list.
#       Otherwise, the appearance of the contents of the list will be obscured by
#       curly braces.
#
```

```
# Arguments:
#       win            Is a path to the text widget in which the annotationSet is
#                      to be displayed.
#       annotationSet  Is the annotation set to be displayed.
#
# Results:
#       The result is that if the annotations being displayed to the user in a
#       text widget. The procedure returns nothing.

proc ggi_ssavDisplayNestedAttributes { win annotationSet }  {

    # ListOrString --
    #
    #       The procedure determines whether its argument's first element is a
    #       list or a string. It is private to the
    #       ggi_ssavDisplayNestedAttributes-procedure.
    #
    # Arguments:
    #       Var  Is a list or a string.
    #
    # Results:
    #       The procedure returns 0 if its argument is a list and 1 otherwise.

    proc ListOrString { Var } {
        if {[string index $Var 0] == "\{"} {
            return 0
        } else {
            return 1
        }
    }

    foreach Annotation $annotationSet {

        # If the current list element doesn't start with a list but with a
        # string, print all elements regardless of their level of nesting
        # relative to the first element. The number of elements may be one or
        # more.

        if {[ListOrString $Annotation]} {
            foreach element $Annotation {
                $win insert end $element\n
            }
            $win insert end \n
        } else {

            # Otherwise, proceed recursively with the current element.

            ggi_printNestedList $Annotation
        }
    }
    return
}

# ggi_byteOffset2LineChar --
#
#       Converts a pair of byte offsets to the corresponding line-character
#       coordinates in a text widget.
#
# Arguments:
#       win    Is the path to the text widget for which the conversion will take
#              place.
#       start  Is the first integer denoting a byte offset.
#       end    Is the second integer denoting a byte offset.
#
# Results:
#       The procedure returns a list containing two elements; the first is
#       "start" converted to line-character format (i.e. X.Y, where X is the line
#       number, starting from 1, and Y is the character number, starting from 0
#       on each line) and the second element is "end" converted to the same
#       format.

proc ggi_byteOffset2LineChar { win start end } {
    return [list [$win index "1.0 + $start chars"] \
            [$win index "1.0 + $end chars"]]
}

#
# End of ambiguity viewer.
#
```