AutoEval - A Generic Tool for Automatic Evaluation of Natural Language Applications

AutoEval - ett generiskt verktyg för automatisk utvärdering av språkverktyg

A Master’s Thesis in Computer Science

By

Antoine Solis

m98_sse@m.kth.se

Revision 7, 2003-03-02

Examiner: Professor Stefan Arnborg
NADA, Royal Institute of Technology

Supervisor: Ph. D. Student Johnny Bigert
NADA, Royal Institute of Technology
Abstract

Natural Language Processing Tools are used as important components in many applications. For example, a word processor uses components like the spell and grammar checker. It is important to regularly evaluate the performance of these components and try to localize the errors that they cause, doing so assists further development and enhancements of the tools. The evaluation is normally performed manually and it is a tedious and time-consuming task. Finding a way to automate the evaluation process would be beneficial.

This Master’s thesis will present one solution to the problem: How can one automate evaluation of Natural Language Applications? The solution is given by the development of a versatile computer program that can evaluate Natural Language Processing Applications or parts of them. The focus of this project was evaluation of part of speech taggers (POS-tagger), which are important components of many Natural Language Processing systems.

The report starts by laying out the project goals and setting the restrictions. A short introductory presentation follows, focusing on the background of the subject. Different technologies will be described and the chosen implementation will be presented. The software development phase will just briefly be touched upon, while explaining the structure and features of the developed evaluation tool more in detail. Problems that were present at start and those that arose during the project’s course will be brought forward and their solution presented. Concluding the report, a brief summary will be given, describing whether the project objectives were reached or not. Finally, suggestions to future enhancements and implementation ideas will be given.

The developed program prototype is a scriptable evaluation tool, which can handle various input formats and tries to be as efficient as possible when handling data. It performs well on simple evaluation tasks. A basic set of functions is provided with the program, these functions serve as the basic building blocks for scripting purposes. Due to the flexible nature of the program, it is easily extendable and thus allows future development to be continued without much trouble.

Examples of real life applications for this tool are: evaluation of POS taggers (word class taggers), grammar checkers and correction suggestions from spelling checkers.
**AutoEval - ett generiskt verktyg för automatisk utvärdering av språkverktyg**

**Sammanfattning**

Språkverktyg ingår ofta som viktiga komponenter i många applikationer. Ett exempel är i ordbehandlingsystem, där de finns i form av stavnings- och grammatikkontroll. Det är viktigt att regelbundet utvärdera dessa verktygs prestanda och att försöka hitta fel som de ger upphov till, på så sätt främja vidareutveckling och förbättringar av verktygen. Utvärdering utförs normellt manuellt och det är en monoton och tidskrävande process. Det skulle vara fördelaktigt att finna ett sätt att automatisera utvärderingsprocessen.

Den här examensarbetesrapporten kommer att presentera en lösning på problemet: Hur kan man automatisera utvärderingar av språkverktyg? Lösningen ges i form av utvecklandet av ett mångsidigt datorprogram som kan utvärdera språkprogram eller delar av dem. Tyngdpunkten på det här projektet lades på utvärderingar av ordklasstaggare (POS taggare), som är viktiga delar av många språkverktygssystem.


Exempel på applikationsområden är: utvärdering av ordklasstaggare (POS taggare), grammatikgranskare och rättningsförslag från stavningskontroll.
Foreword

This Master’s thesis describes a Master’s project in Computer Science, commissioned by the human language technology group at NADA, KTH. It was carried out at NADA during a five-month period in the latter part of 2002.

I would like to take the opportunity to thank people that have made this project possible. First, I would like to thank my examiner: Professor Stefan Arnborg for taking his time to examine my work. I am also very grateful for the excellent supervision in form of support and ideas that I have received from my supervisor: Ph.D. student Johnny Bigert.
B. A TYPICAL TEMPLATE FILE .................................................................31
C. USER’S MANUAL ..................................................................................32
   C.1 PROGRAM OPERATION .................................................................32
   C.2 CONFIGURATION FILE .................................................................32
   C.3 SCRIPTING .....................................................................................36
      C.3.1 Templates .................................................................................36
      C.3.2 Data Types ...............................................................................37
      C.3.3 Internal Function List .................................................................37
      C.3.4 External Function List ...............................................................38
   C.4 FILTER FILE ..................................................................................41
   C.5 RESULT FILE ................................................................................41
1 Introduction

The language group at NADA, Royal Institute of Technology has many Natural Language Applications or Natural Language Processing Tools as they are also called. Manual tests are often performed to evaluate the tools’ performances and to determine the errors they cause. In order to avoid manual processing to the largest extent possible an automated evaluation tool is sought after. This tool should be able to process data output originating from Natural Language Processing Tools. Furthermore, the tool should then be able to categorize the data, extract statistics and present a result. Finally, the tool should be able to present examples of different error types. Examples of real life applications for this tool are: evaluation of word class taggers (POS taggers), grammar checkers and correction suggestions from spelling checkers.

2 The Problem

Natural Language Processing Tools are used as important components in many applications. For example, a word processor has such components like the spell and grammar checker. It is important to constantly evaluate these components and try to localize errors; this in turn assists further development and enhancements of the tools.

Manual evaluation of the Natural Language Processing Tools is a tedious and time-consuming task. The data outputs that the tools produce can sometimes be very large and be in different formats. In addition, other factors make manual evaluation difficult, like evaluation of multiple tools at the same time; comparing their data to the reference answer or to another tool’s output, moreover multiple different operations may need to be performed.

An automated tool specially designed for this purpose is needed.

3 Goals

The purpose of this master’s project was to develop a generic evaluation tool.

This evaluation tool should:

- be constructed, to avoid manual post processing to the largest extent possible.
- be as efficient as possible when processing input data.
- be simple to adapt to handle different data input formats.
- be highly configurable via XML-based configuration files.
- be able to perform various evaluation operations.
- be able to understand a scripting language that specifies what evaluation operations shall be performed.
- be able to produce the resulting output in XML.
- be multi-platform and coded in C++.
- be open sourced and developed using open source or free software components.

The tasks that are not included in the project are:

- Cross-platform testing on all platforms.
- Construction of a GUI (graphical user interface) for the program.
- Development of a full-blown scripting language.
4 Project Phases

In this section the project model is described. The project model’s phases are divided into two branches: a development and a documentation branch, these ran in parallel throughout the project’s course.

Development Branch

1. Information-gathering

The first phase of the project was the information-gathering phase. During this phase information on previous field related research was studied. Furthermore, white papers concerning similar tools and different Natural Language Applications were read. Different competing technologies and techniques were evaluated. Finally, information on development tools and third-party development libraries was gathered.

2. Brainstorming

First, the specifications of the project were provided by the supervisor. Thereafter, ideas on the basic program structure were devised and these were discussed with the supervisor, who came with additional suggestions. Sketching and modularization of the program started at an early stage.

3. Software Development with Feedback

After the initial model was drafted, the main software development phase started. Throughout the development course, consultation with the supervisor was carried out, who came with ideas, suggestions, and other types of constructive feedback.

4. Software Testing

When the development phase was completed, the finished program was tested and debugged. Different types of input files were used and different scenarios were laid out. User errors were introduced at various places as to test the program’s exception handling. A final review of the source code concluded this phase.

Documentation Branch

A. Thesis Writing

The documentation process was started at a very early stage and ran in parallel with the development process. Important thoughts and problems were noted and a draft outline was written. This outline defined the fundamental structure of the report. The report was then written progressively and its final touches were laid just after the end of the development phase.

B. Software Documentation

In this final documentation stage, the program’s source code was commented and a user’s instruction manual was written.
5 Pre-study Phase

During the information gathering phase, or pre-study phase, various field related material were read. Information concerning similar projects that have been performed by other researchers was searched for. White papers regarding Natural Language Processing Tools in general and some implementations of them were studied. Specific information on some of these tools was read more thoroughly.

This section will briefly introduce some similar projects, that have been carried out by other researchers.

6 Similar Projects

Described here are two projects that produced tools similar to the one developed in this project.

DIET

The DIET (Diagnostics and Evaluation Tools for Natural Language Applications) project was started in 1997 and was conducted under a 24-month period. It was coordinated by the German Research Center for Artificial Intelligence GmbH (DFKI GmbH). The project had participants from universities all across Europe and was supported by the European Commission. The aim of the project was to develop tools and methods to aid the evaluation of Natural Language Processing Tools or parts of them. The DIET project is the successor of TSNLP (Test Suites for Natural Language Processing) project, which was a similar project done earlier. The main focus of the previous project and the current one was to create test packages that could be used for evaluating Natural Language Tools that were used to process English, French and German language input texts [5].

ELSE

ELSE (Evaluation in Language and Speech Engineering) is another similar project. Like DIET it was also supported by the European Commission. In many aspects it is quite similar to the DIET project. Its goal was to study the possible implementation of comparative evaluation in language engineering in Europe. Comparative evaluation means that users test different systems with similar data and then compare the result of the different systems. The project has released a piece of software that acts as an Evaluation Workbench for POS taggers, which is downloadable from their webpage [9].

7 Natural Language Processing Tools

The Natural Language Processing tools that are to be evaluated initially are mostly of the type: part of speech taggers (POS taggers), which will be performing word class categorizations on Swedish language texts. This section will describe what these taggers are and describe some implementations of them, particularly one: Granska, which was developed in-house at NADA, KTH [4].
Natural Language Processing tools are important parts of many common programs such as word processors, email clients and search engines where they are used in spell and grammar checking components.

The evaluation types that one may want to perform can be of various kinds like evaluating the performance of various POS taggers by comparing their output to a known reference, or to compare one tagger’s output against output from other taggers, other combinations are also possible. Nevertheless, the developed program is not limited to just evaluating POS taggers. The program was developed with versatility in mind and can be used to evaluate other Natural Language Applications as well.

7.1 Part of Speech Taggers

A part of speech tagger is a word class tagger, a piece of software that can determine the correct word class of a given word in a given context. The tagger performs this categorization by utilizing various different methods and is able to produce a result with high accuracy for both words known and unknown to the tagger.

Words in a language may occur in various grammatical categories depending on the context in which they appear. An example is the English word: “train”. “The train was big and black”, in this context the word “train” is a noun. “You will have to train more”, in this context it is a verb. Some words however, only fall into one grammatical category regardless of their textual context like the word “pear” which is always a noun.

A dictionary normally lists all possible grammatical categories for a given word. One has to figure out the correct grammatical category of a given word in context by considering the surrounding words. A part of speech tagger, abbreviated POS tagger is a computer program that attempts to identify and associate words with their correct grammatical categories by analyzing the textual context in which the words appear. The result is given as a tag, a string that contains grammatical information for a specific word. The format of these tags vary from tagger to tagger, each tagger usually has its own format [17].

The POS tagger usually works in three steps

1. Tokenization
2. Lexical lookup
3. Disambiguation

During the first step, tokenization, a piece of text is broken up into words, numbers and punctuations, these pieces are called tokens. Secondly a lexical lookup is performed, during this step a morphological analyzer is used to associate each token with a set of possible tags. The morphological analyzer uses various methods to identify the word-stem of a token, by doing so; it can associate the token with a set of possible tags [15]. Unknown words are also handled during lexical lookup, but by utilizing statistical methods instead of morphological analysis that is used for known words. Finally, disambiguation takes place, which means that one of the tags in the set of possible tags is chosen as the correct one, this selection is usually done with statistical methods like the Hidden Markov Model. The POS tagger then outputs the tag, which it thinks is the correct one [17].
7.2 Granska

Granska is a powerful grammar checking system for the Swedish language, developed at NADA, KTH. Granska can detect and suggest corrections for numerous grammatical errors in Swedish language texts. It is a hybrid system, which uses both probabilistic and rule-based methods to detect grammatical errors [4].

The POS tagger is an important part of Granska. It uses a second order Hidden Markov Model and statistical morphological analysis to assign correct tags to words. By utilizing these powerful methods, it can tag both known and unknown words with a high degree of accuracy [4].

7.3 Sample Output

This section will demonstrate how output from POS taggers may look. A manually tagged text will also be shown; it functions as a correct reference answer. Four different tagged texts will be used as examples. The first three examples are tagged up versions of the same Swedish language text, while the fourth one is a tag up of a different Swedish language text.

Smygrustning nn.utr.sin.ind.nom
av pp
raketvapen nn.neu.plu.ind.nom
Av pp
MATS pm.nom
LUNDEGÅRD pm.nom
DN:s pm.gen
korrespondent nn.utr.sin.ind.nom
. mad

Illustrated above is a sentence from a typical output generated by a POS tagger. The output has two columns, in this case they are separated by a single <tab> character and the second column is terminated by a <tab>. The first column contains a word and the second column contains a tag. The particular output shown above was generated by the POS tagger from Granska.

This combination of a single word and its associated tag can be as a set of data; related pieces of data that are treated as a single unit. This unit will be referred to throughout the report with the name data set. In the example above, there is a data set in each row, totaling it to nine data sets in the entire example.

Smygrustning nn.utr.sin.ind.nom av_pp raketvapen_nn.utr.sin.def.nom
Av pp MATS pm.nom LUNDEGÅRD pm.nom DN:s pm.gen
korrespondent nn.utr.sin.ind.nom . mad

This is the same text but tagged by a different POS tagger; it has the same tag format, but has a different layout. The difference in the layout lies in the difference of the first and second column delimiters and the presence of new line characters. This output was generated by Mxpost: the “Maximum Entropy POS Tagger”, which was developed at the Department of Computer & Information Science, Penn Engineering [18].
Yet again the same text, but this time it is the correct reference answer, this text has been manually tagged by a human. The third column contains the word-stem of the tagged word.

This is a part of different tagged text that was tagged by a POS tagger that produces the output in XML. It uses the same tag format as the previous two taggers, but because it is in XML format, the file has a completely different layout.

The presented outputs from the three taggers all have the same tag format, other POS taggers could however produce tags that conform to completely different formats.

The conclusion that can be drawn is that there can exist various combinations of tag and layout formats. If one wishes to evaluate taggers that have different output formats one will have to design an evaluation tool that can handle this difference and be able to translate between different tag formats.
8 Technologies

This section will describe some of the key technologies that are utilized by the evaluation tool and are essential to it. Listed are also the third-party run-time libraries that provide these functionalities. A short explanation will be given to why these technologies/components were selected to be used in the project.

8.1 XML – the Extensible Markup Language

XML is a widely adopted markup language that can represent structured information in a format that can be easily processed without much effort by a human or a machine. XML was derived from a subset that originated from SGML. Information in XML format can be exchanged across different platforms and applications; both text and binary information can be stored in XML. It is supported by a wide range of applications and development tools, the application areas of XML are very broad, ranging from databases to word processors [10][16].

An XML document is data that contains XML markups and data stored within them. The document can be stored in many different ways, for example: as file on a local file system or in a text field of a database table on a remote server. The XML format also has support for DTD or Schema files, these files define the legal layout of an XML document and are used by parsers to check the validity of a document [10][16].

XML is an Internet standard and is backed up by major industry players like Apple and Microsoft. For instance, the upcoming version of Microsoft Office, version 11 will support the saving of office documents in XML as a secondary format.

Versatility, standardization and industry support were the key deciding factors for choosing XML as the format of choice for the configuration and data output parts of the evaluation tool.

8.2 XML Document Programming Interfaces

If one wishes to interact with XML documents one will need an XML framework, which is typically implemented as development library. The framework’s task is to read and write XML documents and to provide the user with an API (Application Programming Interface). The module of the framework that reads, interprets and in some cases validates XML documents is called an XML parser. Most XML frameworks provide the developer with two standardized APIs, which offer two completely different methods of interacting with XML documents. These APIs are called SAX and DOM. They both have their advantages and drawbacks, and they are intended to be used in different situations.

8.2.1 DOM – the Document Object Model

DOM is an interface (API) for accessing and manipulating data in an XML document. In DOM, the XML document is represented as a tree of linked nodes [19]. The developer interacts with the document by moving between different nodes in the tree and by adding, removing or manipulating the nodes or the data stored in the nodes. Currently the API has three levels: Level 1, Level 2 and Level 3 [7].
DOM’s advantage over SAX is that it is much easier to use and provides an intuitive way of interacting with the XML document. The disadvantage is that for larger files it has a high memory requirement, due to the fact that the entire file has to be read into memory.

The DOM API was chosen to be used in the program whenever it was feasible, when handling XML documents that were not extremely large. The decisive property that made DOM the primary API of choice was; the ease of use, which would facilitate the development cycle.

8.2.2 SAX – Simple API for XML

The SAX API provides another way of accessing and manipulating data in an XML document, but unlike DOM the document is not represented as a tree of nodes, but as a sequence of parse events. When using SAX, one has to write a program module that will receive and interpret parse events. These events will be emitted when the XML parser will “see” a part of the XML document. SAX enables one to perform progressive parsing, collecting data progressively, and having the option to abort parsing before reaching the end of the file. When using SAX the entire documents is not stored in memory like when using DOM, thereby SAX can be used to read large documents or parts of large documents without being a heavy memory burden [13].

The SAX API was selected to be used when the evaluation tool would read XML input data from the Natural Language Processing Tools, these data streams are often very large. In addition to memory conservation, SAX can perform progressive parsing, which was a desired feature.

8.3 Xerces C++ – the XML Framework

Xerces C++ is an XML framework that provides DOM (Level 1, Level 2 and partial Level 3) and SAX/SAX2 APIs. Xerces has support for validating parsers and XML schema [20]. Xerces is written in C++ and is considered portable. It is developed by The Apache Software foundation, which has previously developed other enterprise quality open sourced software projects, like the most famous: the Apache Webserver [1]. The XML framework was developed with the intention to be used as a component in various apache projects that require XML capabilities. Xerces also has a Perl interface which allows access to the APIs from C++ implementation and there is also a version of Xerces available for the Java2™ platform.

The evaluation tool uses the Xerces C++ shared library for all its XML functionality. It uses both the DOM and the SAX API. Xerces was the XML framework of choice because it is generally regarded as a mature implementation and it has more features than other competing implementations.

8.4 Boost Regex – the Regular Expression Library

Boost Regex is a library developed by Dr. John Maddock and it is part of a free peer-reviewed set of C++ libraries called Boost.

The Boost Regex library provides the evaluation tool with regular expression capabilities. A regular expression is a string that contains characters with special meanings. Regular expressions are used to define patterns in data and are used by various programs that perform pattern matching on data. Regular expressions are typically used in search operations or search-and-
replace operations. For example the UNIX utilities: grep, sed and awk use regular expressions. There exists some regular expression functionality in the standard C library. Although, this functionality is limited and the functionality that the Boost regex library provides supersedes the one from the standard C library [12]. The extended functionality and the fact that it has a C++ interface, were the main reasons that Boost Regex was chosen to be used in the project.
9 AutoEval – the Evaluation Tool

This section will explain the structure of the evaluation tool and list its features. This section gives a quick and less technical overview of the tool. The next section “Cornerstones” will give a more in-depth and technical explanation of how the program is structured and how it performs some tasks practically.

The evaluation tool is a Command Line Interface (CLI) program, which can be run from a shell on a UNIX® or UNIX® like system. It can probably run in a “dos window” on Microsoft Windows®, thought this program has neither been compiled nor tested on Microsoft Windows®. The evaluation tool was developed using the GNU GCC 3.1/3.2. It has been compiled and tested under Linux® 2.4.20, Sun Solaris® 9 with the GNU GCC, and Darwin 6.0/Mac OS X 10.2 with the development tools. Because this software was developed with portability in mind, it should compile on most platforms that have a C++ compiler and an implementation of the STL. The evaluation tool however uses external components that are written in C/C++ and it requires these to be compiled and present on the platform where the evaluation tool will be used. It is recommended to use the GNU GCC for compiling purposes, as it is free, available on many platforms, and backed up by major industry players like Apple.

The evaluation tool currently takes two command line parameters: “-c <configurationfile> –o <outputfile>”. The parameter “<configurationfile>” should contain the full path to the configuration file. The –o switch and its associated parameter “<outfile>” are optional, and are used to specify an alternative output file.

9.1 The Configuration File

The format of the configuration file is quite simple. It is an XML format file and it controls nearly all aspects of the evaluation tool. The file is logically divided into several sections.

Using the configuration file, the user can specify the following:

- Defining files: file names, types and file formats
- Defining the data set
- Defining processes that will be performed on the data set
  - Once for each new data set – these will be performed once for each data set.
  - Once for each run – these will be performed once each program run, when all the data sets have been processed.

See appendix A for an example of how configuration file could look like. The user’s manual explains the configuration file more in detail.

9.2 Input Data

One of the main features of the evaluation tool is that it can handle numerous types of input data, thus making it flexible with different input data formats. Presently it can handle data contained in plain text and XML format files. The relevant data; the data set, which should be read and processed by the program is user specifiable. From a user’s point-of-view the tool will read a sequence of data sets from each input file and perform various operations on them. Practically, the extraction of data from plain text input files is done by pattern matching using regular expressions. Regular expressions are one of the most powerful mechanisms to select
specific strings from a set of characters. For structured data like XML format input files the user specifies which part of the XML document contains the data set or which parts of the document contain pieces of the data set.

### 9.2.1 Filters

As discussed earlier the input files could have different layouts and tag formats. The difference between layouts was solved by using plain text and XML file readers combined with regular expressions. The difference between tag formats was solved by filters. Filters are used to substitute a string for another. For example, one tag is substituted for another tag. Filters enable the evaluation of a tagger that has a different tag format than for example the reference answer or another tagger.

The filter data is stored in a separate plain text file, this file has to specified in the configuration file, along with the filter name. One can have an arbitrary amount of filters. The filter data is loaded into memory for faster operation. The format of the filter file is user specifiable.

An example filter file is shown below, first is the string that is to be substituted, separated by an “_” (underscore) is the second string, which will take the first ones place.

```
TagType1FromFormat1_TagType1FromFormat2_
TagType2FromFormat1_TagType2FromFormat2_
...
```

With the particular filter file shown below, the string “AA.BB.CC” would be replaced by the string “XX-YY-ZZ”. When a filter is to be used, the user calls a special filter function, in which the user specifies the name of the filter to use, the format of the filter file and the data string to be filtered.

```
AA.BB.CC_XX-YY-ZZ_
DD.EE.FF HH-FF-TT_
```

The filter feature has enabled the program to become more flexible in its input data use. In the examples presented in this section, the filter is primarily used to translate from one tag format to another, but it is not limited to doing just that, any type of string substitution can be performed.

### 9.3 The Scripting Language

One of the main concepts of this program is the flexibility of the operations that would be performed on a data set. This flexibility was achieved by implementing a simple scripting language. The functions that are called from scripts are either *internal functions* or *external function*. Internal functions are those that are hard-coded into the evaluation tool, while external functions are those that are user-written and mapped by the program.

The internal functions are exclusively used for data acquisition. These functions normally take only one or two arguments and the return type depends on the function, but generally, it is a simple data type or a pointer or reference to a simple data type. The external functions are used for performing different processes; they can have different argument types and counts, but always return a Boolean value. The external functions have made the scripting language dynamic and easily extendable. All that is required from the user is that the user creates a function with arbitrary types and numbers of arguments, but with a return type of a Boolean
value. Then a few simple steps are needed to be performed so that the new external function will be recognized by the program. How this is done practically is described in section 10.9.1. The external functions are located in modules outside of the main program.

It is possible to have two different versions of a function that corresponds to the same script function name. This provides easy function overloading, it is a feature that allows the evaluation tool to automatically choose the correct version of the external function, its decision is based solely by the **number** of arguments.

```plaintext
inc(cnt("words"),3);
filter(filt("filter1"),"_","_",var("tag1"));
```

Above is an example of what scripting functions may look like when written in scripts. The syntax of the scripting functions resembles the syntax of C/C++. Each statement is placed on a single line for clarity and it is terminated by a semicolon. In the example above two external scripting functions are displayed: “inc” and “filter”. The function “inc” takes two arguments, while “filt” takes four arguments. “inc” takes an integer reference and an integer. “filter” takes a reference to a filter, two strings and a reference to a string. The use of internal functions will now be explained. In the given example three different internal functions are used: “cnt”, “filt”, and “var”. Executing ‘Cnt("words")’ will return a reference to an integer variable named “words”. Likewise executing ‘filt("filter1")’ will return a reference to a filter named “filter1” and ‘var("tag1")’ will return a reference to a string variable named “tag1”. The variables mentioned are stored in a special data storage module in the program. The returned values of the internal functions are then used as input arguments for the external functions: “inc” and “filter”.

Scripting functions can be written in two separate places in the configuration file, depending on how often one wants them to execute. The only exception is the “data set specifier function” which must always be written in a special section called “fields”. It is written there because it is essential that it be executed first. Its function is to acquire data sets from the buffers; these data sets form the basic variables that the other scripting functions rely on.

### 9.3.1 Basic Set of External Functions

Included with the program is a basic set of external functions. These functions are provided to give the user a kick-start, so that one will be able to use the program straight away. The basic set consists of functions frequently used in an evaluation environment and some functions that are essential or highly recommended for program functionality. The essential functions must **always** be present, since they provide core functionality to the program. The functions that are highly recommended provide the program with important features; the program could function without them, but with reduced functionality. All these functions can be re-implemented, but their main functionality should be preserved. Failing to do so will create a functionality, which differs from the original concept.
Here is a short description of what types of functions are included in the basic set:

- A function that specifies the data set, it is essential for the program’s functionality and must always be executed first.
- A regular expression function, this function is used to extract parts of data strings; it is normally not called directly, but instead used in a template. This function is considered highly recommended.
- A filtering function, which performs translations between different tag formats, it can also be used for other string data. This function is highly recommended.
- Boolean functions such as: “and”, “or”, and “not”, there are multiple versions of the first two mentioned, which allow more complex expression to be formed.
- A conditional function: “if”, which takes a condition and a function to be executed if the condition is true. Combining the “if” with the Boolean functions allows complex condition statements to be created.
- Comparison functions, for both string and integer data.
- Counter incremental function, which are used to increment numeric counters.
- Various outputting functions, these output different kinds of data or data structures to the result file.
- A random function, which takes a probability in percent, randomizes and returns a Boolean value based on the outcome.
- A majority decision function, which can perform a simple majority decision.

**9.3.2 Templates**

The template feature is used to simplify the creation of scripting functions. Templates resemble macros in many aspects. The template definitions are stored in a separate file, which is specified in the configuration file. The usage of templates is not mandatory, but it facilitates the creation of scripts.

By utilizing templates, the user can group a set of frequently used scripting functions into one special scripting function called a template. To execute the group of scripting functions one would only have to call the name of the template function. A single template function would be expanded into a single or a sequence of scripting functions. When writing a template one has the ability to “fill in” some of the arguments belonging to the scripting functions, while leaving others to be “filled in” by the user. An example demonstrating this fact will be presented shortly. The templates can take an arbitrary numbers of arguments. A template can also contain other templates, thereby forming larger groups of functions. When a template function is created, it is specified which and where arguments supplied to the template will be substituted in the scripting function or functions. The template expansion is performed by a “cut and paste” procedure done by the software. Below is a very simple example of how a template can look like and a demonstration of template expansion is presented.

```xml
<template name="gender" args="2">
    regex(#1,#2,"utr|neu|utr/neu|mas");
</template>
```

In this case, the template is called “gender” and it takes two arguments, as specified by the attributes “name” and “args”. In this simple example, the template only contains one scripting function: “regex”. This template will “fill in” the third argument of “regex” with “utr|neu|utr/neu|mas”. The first two arguments of “regex” will be “cut and pasted” from the first and second arguments passed to “gender”. The “#n” symbol specifies that it will be replaced by the n-th argument passed to the template. Executing the template in a script is done
as with any other script function and templates can be used in the exact same way; the only difference is that templates do not return any values.

\[
gender(arg1, arg2);
\]

This is what the template instruction will look like in a script, it has two arguments, “arg1” and “arg2”.

\[
regex(arg1, arg2, "utr|neu|utr/neu|mas");
\]

When the template will be expanded it will represent the above shown script function.

\[
<\text{template name}=\text{"gender\_and\_wordcl" args}=\text{"3"}>
  gender(#1,#3);
  wordcl(#2,#3);
</\text{template}>
\]

The example above illustrates a more complex template, where “gender” and “wordcl” are not script functions but templates. The given example has templates within templates!

As one can see, these templates can be very powerful if used properly and they reduce the amount of tedious work of writing many functions in a script. This fact is especially true if one repeats a group functions many times in a script. Templates also bring more clarity to a script and make it more structured.

See appendix B for an example of a typical template file. The user’s manual explains the template file in more detail.

### 9.4 The Result File

This file contains the actual result of the processing that the program has made and is an XML format file. The data contained in this file can vary depending on what process instructions have been written in the script.

The data can contain various statistics, which are of interest for the user as:

- String variables
- Integer variables
- Complex variables structures
  - An Error Example – a structure that can be written when two mismatching data sets are found. These data sets can be found when comparing two parser outputs.
10 The Cornerstones

This section will give a more detailed presentation of the program with its different components that serve as its cornerstones. Described here is also how different program modules interact with each other. Furthermore, some programming problems and their solutions are presented. Finally, two simple examples of a configuration and a result file are shown.

10.1 Basicfun – the Basic Set of Scripting Functions

This module contains the basic set of functions that the user can use to process the data with. The functions take different types and numbers of input parameters and they all return a Boolean value. They are provided to the user as a convenience and to facilitate initial evaluation runs, thus enabling an easy start.

Due to the system’s flexibility, it is easy to write custom functions. This is achieved by performing a three-step process. First, one writes the code and prototype for the function in this module or another module, if one wishes to separate the functions. Secondly, one assigns the function to the program’s function map; this is done by calling a special mapping function in the varprocessor module. Finally, the user calls the newly mapped by writing its scripting function name in a scripting section of the configuration file. This scripting function name was chosen when the function was assigned to the function map and is called mapped name.

10.2 Databank – Data Storage Module

This module is responsible for storing data that the evaluation tool uses and it stores data like data sets extracted from input sources, data derived from input data and new data that is created. This databank contains different types of storage containers for different types of data variables.

The different containers are:

- A Input buffer map – this map contains pointers to input buffer objects
- An Output buffer map – same as above, but pointers instead to output buffer objects.
- A Filter map – contains pointers to filter objects.
- Counter maps – contains named unsigned integer counters.
- String maps – contains named string variables.

10.3 EvExp – the Program’s Exception Structure

An exception in C++ is a data structure that is emitted when something unexpected occurs, usually a program error. The exception structure contains error information and is usually intercepted and decoded by some part of the program, where an error message is presented to the user.

EvExp is a data structure containing three data strings. When a program error occurs execution is aborted and this exception type: EvExp, is emitted. Some parts of the evaluation tool use external components that emit their own exception types, these exceptions are caught where emitted. Relevant information is extracted from them and stored with additional information in an EvExp, which is then reemitted. Performing this re-encapsulation makes the er-
ror handling more uniform; all the error information is encapsulated in an EvExp when it reaches its final destination, which is the “main()” function of the program. Here the information is extracted and presented to the user in the form of an error message. As an extra safeguard the main() function also catches unknown exceptions that might rarely be emitted by the currently used external modules and this feature also has the ability to handle exceptions from additional external modules that could possibly be used in the future.

The three strings contained within EvExp that carry error information are:

- **Section** – this string contains the name of the module or the execution stage in which the error occurred.
- **Error** – specifies the type of error that has occurred and sometimes contains a suggestion to the probable cause.
- **Data** – contains additional data regarding the error such as input data, line number, line column, filename, etc.

### 10.4 Evinit – Start and End Tasks

This module contains various initialization and termination tasks that will be performed first and last in the program. These tasks include: starting the program’s configuration stages and the starting and stopping of the Xerces XML framework. Stopping Xerces ensures that all resources dynamically allocated by Xerces are released; this is not mandatory as stated by the Xerces documentation but good practice. Doing this facilitates further development when developed in conjunction with memory leak detectors thereby reducing false alarms.

### 10.5 Evutil – Utilities

Collected in the Evutil module are some miscellaneous functions that are used throughout the evaluation tool in its various modules. Among these functions are those that perform different types of data conversions like string to integer conversions and vice versa. Furthermore, functions that convert to and from the Xerces XML native string format (Unicode), these conversion functions are heavily used, as most of the Xerces API requires string data to be in its native format. Located in the Evutil module is also the EvTokenizer class, which is used to tokenize strings that represent script function calls, these strings are found in the scripting sections of the configuration file. The tokens are further processed and are finally interpreted and executed.

### 10.6 Insrc – Data Input Module

The data input module defines a base abstract interface, a generalization of the data sources. Currently there are two input source implementations; one for normal plain text data and the other for data contained in XML files. These input sources are practically transparent data buffers, which read parts of their associated input files and store this data in a string. A buffer reader from a different program module then requests this string and searches and extracts data from it. If the search fails, a matching pattern cannot be found, the input source is notified and the buffer is enlarged and shown to the buffer reader at request, this procedure is repeated until the desired pattern is found or the maximum buffer size is exceeded. If the buffer size is exceeded, an exception will be emitted, telling this fact. If a buffer reader finds the pattern it wants, it notifies the input source how many bytes it has read from the input buffer. The input sources then discards the read data and refills the buffer with new data from the
input file. In the case of an XML input source, the string buffer will always contain relevant data, because the data in an XML file is structured and the input source knows which data is relevant and which is not. This knowledge of data relevancy was acquired during the configuration stage, based on user specified parameters. If the buffer reader cannot find any for it relevant data in this buffer, the buffer is not enlarged, because a relevant set of data is present in the buffer. An error has clearly occurred, and an exception is immediately emitted.

The plain text input source has compile time adjustable parameters which specific the buffer size, buffer increment step and maximum buffer size.

The XML data input source uses the Xerces SAX2 API. This means that the reader does not parse the entire file in one pass and store it in memory like when using the DOM API. Instead a SAX2XMLReader with progressive parsing is used, which parses the file until it finds a set of relevant data, it then stops parsing and stores the found data in a string buffer. The next time data is needed the parsing will continue from where it last stopped and continue on until a new set of relevant data is found or the end of the input data file is reached.

Using the mentioned memory conservation technique allows the program to handle extremely large input data without taking up much memory. One disadvantage with this technique is that the program will have to perform more I/O operations than if the entire file would be read into memory in one pass.

Letting all input source classes abide to an abstract interface permits the buffer reader to read from different types of input sources without knowing its type. All the buffer reader must know is how to interact with the input source interface: how to request data, report if it has read relevant data and if so, how many characters were read. Utilizing this interface methodology makes it very easy to extend the input source module. For example, one can create input sources that read data strings from structures containing strings, named pipes, over Unix domain sockets, and over TCP/IP connections; maybe an XML document stored in database on a remote server.

10.7 Main – Startup and Cleanup

Due to the modularized nature of the program, the main function contains almost no code. Just some basic startup and cleanup functions are called from main. For example, the initialization and termination functions are called from here. Possible exceptions caused by errors are also caught and presented to the user. This function is also responsible for collecting command line parameters that the user has specified and sends the data contained in them on to configuration module. An information text is displayed to the user if the program is started with no or incorrect number of parameters. This information consists of a help text, version information and the program’s license.

10.8 Outxml – Outputting The Result of The Data Processing

Outxml is the module responsible for producing output from the evaluation tool. The module contains various functions that allow the tool to output some of its variables from the Data-bank. Amongst its functions are those that allow single string or integer values to be written to a user specified section of the output file. In addition, there are functions that can create more complex outputs, like the error example type, which is used to represent parts of a mismatching dataset.
Practically the Outxml module works by creating an XML document using the DOM API that is written to a file (serialized) at the end of the program run. Xerces’ pretty-printing functionality is currently limited, so some manual formatting is done to make the outputted XML document more readable. Beside result data, the result file also contains a date-time stamp.

Part of an example result file is shown below. The root of the document is `<evaloutput>`. The contents of the first two sections `<section1>` and `<section2>` represent output of simple variables. The contents of each section could be mixed, both string and integer variables can be present. The variable and section names are arbitrary, but the name has to be accepted by the XML framework, which currently allows no white spaces in the names. For example `<sectionone>` is allowed, but not `<section one>`. One can divide the result file into arbitrarily many sections under the root of the document, and these sections can contain an arbitrary amount of variables. The third and last section `<examples>` contains a complex variable, which is located under its own subsection and is in this example named `<onevstwo>` . In this case, the complex output is of the type `error example`. The error example contains an attribute and two variables that can have arbitrary names and contain arbitrary data, this complex variable is intended to be used to present an error example. An error example could be outputted when two tags from different data sets mismatch.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<evaloutput date="Mon Dec 24 23:37:35 2002">
  <section1>
    <var name="words">145</var>
  </section1>
  <section2>
    <var name="tag1">nn.neu.plu.ind.nom</var>
  </section2>
  <examples>
    <onevstwo word="raketvapen">
      <granska>nn.neu.plu.ind.nom</granska>
      <mxpost>nn.utr.sin.def.nom</mxpost>
    </onevstwo>
  </examples>
</evaloutput>
```

### 10.9 Process – Scripting, Interpretation and Processing

The process module is the largest, most complex and perhaps the most important part of the evaluation tool. It carries out important functions such as interpretation of scripts, execution of internal and external functions and resolving templates to scripting functions. One of the key features of this project was flexibility and this had to be taken into account when developing the process module.

#### 10.9.1 BaseFunCon – Mapping Script Function Names to External Functions

One of the biggest problems encountered during the project’s course was to find a good way to use `external functions`, these functions would be user-defined and could be located outside of the main program, in a separate module or library. An initial idea was to use function pointers in some way. The external functions could however contain an arbitrary amount of arguments and these arguments could be of an arbitrary type. A different type of function
pointer would be needed for each combination of argument type and argument number. The data container, a map that would contain the script function name mapped to the external function could only use a single data type to represent the external function. Because the function pointers would be of different types, the proposed solution would not be feasible.

Another idea was to encapsulate the function pointers inside some sort of object. This idea could be implemented as a class that would inherit from a generic base class; this base class would serve as a representation of an external function. With the arrival of this idea another problem arose. Each derived class could only contain one pre-defined type of function pointer. This practically meant that one would have to create a base class for each unique function pointer type! This solution was not feasible either.

The final solution to the problem was based on the encapsulation idea, with some additions. It was devised by creating a derived class for each parameter count, a subclass for a external function with one parameter, a subclass for a external function with two parameters and so on. Doing so would limit the amount of derived classes; and this limit would depend on how many arguments the external functions would take. The problem of the external functions having different argument types would be overcome by using the template feature of C++. The template feature enables one to reuse the source code itself [8]. Instead of defining the arguments as fixed types, they would be defined as template types. At compilation-time, the compiler would reuse the source code and automatically creates different versions of the classes, for each combination of arguments. The only limitation that the chosen solution imposes is that the return type of the external functions all has to be the same for all of them. In the current solution, a Boolean was chosen as the return type.

The base class is named BaseFunCon and its derived bases classes are named: FunctorOne, FunctorTwo, and so on. The number in the name specifies how many arguments the function that is being encapsulated takes.

Assigning a script function name to an external function is easy. It is performed by making the call below, appropriately in a function located in the Evinit module.

\[ \text{Varprocessor.addFunc("script\_function\_name",pointer\_to\_function);} \]

### 10.9.2 Data Processing

The process module stores the scripts that perform the calling of functions, it is these functions that extract and process input data. The scripting functions are stored together with their arguments in three sequential containers, one for instructions that acquire data sets, one for instructions that will run more than once and the other for instructions only that will only run once. When an operation is to be performed, the scripting function is read and interpreted, finally the external function mapped to the script function is executed, the arguments passed to script function are passed to the external function. This function mapping has been done during program initialization. It is implemented as an image from script function name to a functor object, where the script function name is the same name that is being used in the script. Because scripting functions can contain arguments that use return values from other functions its arguments are first interpreted. The scripting functions providing the return values will first be interpreted, if these scripting functions have additional functions as arguments, those will also be interpreted and so on recursively.

Some scripting functions are **internal functions**; these are not mapped functions, but instead hard-coded in the process module. They are used to access data in the databank module and their return values are either: a data variable, a reference or pointer to a data variable.
The process module is also responsible of performing template resolution; some scripting functions may be templates instead of functions. These templates are like macros. When a scripting function is added to the sequence container the process module determines if it is a template function or not. If the scripting function is not a template, it is added normally, but if it is a template, it will be resolved first. Resolving the template means that the template function is replaced with its corresponding scripting function or functions. If the template contains other template functions, they will be resolved too and so on recursively. By using the templates feature, the user will be able to write less text in the scripting section of the configuration file. Templates can be used to create complex groups and sequences of scripting functions that are frequently used. The template definitions are stored in a separate XML file. This definition file is then specified in the configuration file, the usage of templates is optional.

Some scripting functions are interpreted and executed whenever a new set of data arrives from the input sources. Examples of these functions are those that perform: data extraction, substitution of data and functions that increment counters. Other functions are only interpreted and executed when all the data sets from the input sources have been processed. These functions are performed only once each program run. Examples of these, are functions that calculate and output statistics.

10.10 XMLconf – XML Configuration

The program configuration is done entirely by a single configuration file. This file is an XML file, which has an associated grammar file, an XML schema file (XSD). The XSD format is the successor of the DTD file, which is a much simpler grammar file format. XML schema has more features that its predecessor. The configuration file’s grammar specifies its syntax; which tags are legal and which are not, the correct combination of tags, where attributes are allowed, data types and much more. The XML schema file is itself an XML file, which has its own special syntax [14]. The XML framework uses the XML schema file, to run the parser in validating mode [20]. When the parser is running in validating mode it can detect any syntax errors in the configuration file. If such an error is found, parsing is immediately interrupted and the user notified. The validation procedure ensures that the configuration file is syntactically correct; during this process, unnecessary white spaces found in the file are ignored. Moreover, validation reduces the amount of error detection that has to be done by the configuration module. Although, errors that are not syntactical ones are not detected. An example of such an error is, if the user has entered a valid file name but the file does not exist, these types of errors will be detected at a later stage.

This configuration module uses the DOM API to read the configuration file. The entire file is parsed and the DOM document tree traversed. During the traversal relevant nodes are search for, when one is encountered, its data contents is packaged inside a map. This map is an internal format representing the data inside a DOM node. The map contains the variable names mapped to their values and other data such as the node’s name and if the node had a text node under it there would be a variable called “#text” with the contents of the text node.

During the interpretation phase of the configuration stage, a function inspects the maps and determines which section of the configuration file it represents. The section relation determines to which function it needs to be forwarded to. When the map reaches the correct function the its contents is read and interpreted, some of the data is extracted and passed along to other functions in other modules that perform the actual configuration of the program. These configuration tasks could be such tasks as: setting up input and output buffers, storing scripting functions or templates and loading filter files.
Depicted above is a skeleton structure of the main configuration file. The “…” represents data that the user will supply. For a complete example of a typical configuration file, see appendix A.

The configuration file consists of six main sections:

- **<root>** – The root of the document, contains various information such as the XML namespace, encoding, and the location of the XML schema file.
- **<libfiles>** – Specifies additional files which contain configuration data, currently only the template definition file.
- **<files>** – Defines all types of input and output files.
- **<fields>** – Defines the data set from each input buffer.
- **<process>** – Contains the script that will be executed whenever there are new data sets available.
- **<processonce>** – Contains the script that will be executed only once, when all the data sets have been processed and there is no data more available.
11 Were the Goals Reached?

This concluding section will discuss whether the project goals were reached or not. The nature of the discussion will depend on whether a goal was reached or not. If a goal was reached, a brief recollection and summary of how it was reached will be given, but if a goal was not reached, an explanation of why it was not reached will be given instead.

The tool should be constructed; to avoid manual post processing to the largest extent possible and be able to produce the resulting output in XML.

All output from the program is in XML format. This enables the result to be presented in both a human readable form and a form that could easily be processed by a machine.

Certain steps are taken to assist the XML engine to make the resulting XML data more human-readable. One of the features of XML is that it can be linked to a XSL (Extensible Style Sheet Language) or a CSS (Cascading Style Sheet). This feature enables a modern web browser to render XML data in a webpage-like presentable form, which could contain tables, and other layouts as specified by the XSL, or CSS file.

The XML data result can also be easily parsed and interpreted by another program for additional processing.

The tool should be as efficient as possible when processing input data.

When the program performs file reading of input data files that can be very large, it does not read the entire file into memory; it just reads a part of the file into memory. Files that can be large are usually those that contain the data sets. When the program has processed the piece of data that was stored in memory, a sequential piece of the file is read into memory, where it overwrites the previous part. The evaluation tool does not store previous data sets, when a data set has been processed the data is no longer needed and it is overwritten by the next data set. The only data that is kept are variables that are created by the user; these could however be overwritten to by the user.

The tool should be simple to adapt to handle different input data formats.

When designing the input system, a base abstract interface was devised. All input modules must conform to this interface. This interface makes it very easy to write additional input modules. Currently the program has only two input sources; they support reading data from plain text and XML files. The present input sources read data from files, but additional input modules are not limited to just that, they can read data from any source as long as they implement the abstract interface. This interface is very simple and contains few functions, thus easy to implement.
The tool should be highly configurable via XML-based configuration files.

The entire configuration is handled by a single configuration file, which has the ability to specify other specific XML configuration files that will also be parsed and interpreted. The configuration file has a grammar files associated with it; in the evaluation tool’s case, it is an XML Schema file (XSD). The grammar file enables validation, which ensures that the configuration file has the correct syntax before its data contents is interpreted by the configuration module. Validation is performed at parse time by the XML framework. The validation process also reduces the amount of exception handling that has to be performed by other modules.

The tool should be able to perform various operations on the result and to be able to understand a scripting language that specifies these operations.

The formula language is simple and extensible. It is composed of both internal functions hard-coded into the software and external functions created by the user. It also has a template feature that simplifies tedious scripting tasks.

Due to the flexibility of the scripting language and its associated external functions, almost any kind of operation can be performed on the data.

The basic set of external functions provided with the program, allow typical evaluation routines to be performed.

The tool should be multi-platform, coded in C++.

The evaluation tool was written entirely in C++. The compiler used was the GNU GCC 3.2 and its associated stdc++ library. The evaluation tool uses a minimum of functions from the old C library. Special care was taken when the STL library was used, features that might not be available on some platforms or other implementations of the STL were not used. For example, hashtables and ropes are not used. The only external runtime libraries used by the evaluation tool are Xerces C++ 2.1 and the Regex library from Boost. Xerces provides the program with XML functionality and the Regex library from Boost provides the program with regular expression functionality.

Be open sourced and developed using free software components.

The evaluation tool is released under the GNU General Public License 2.0 [11]. The compiler and its associated STL library are also governed by this license. The other components used by this program are also covered by other free licenses. Xerces uses the Apache Software License 1.1 and Boost uses Dr John Maddock’s license [2][12]. Other software used such as: text editors, debuggers, and various utilities are also free.
12 Conclusion

As in all other software projects, development takes a long time and requires much work to be done by a software development team. After several intermediate test releases like alphas, betas and release candidates, a final and hopefully bug-free version can be made.

The program is currently in pre-alpha stage. It operates quite well, but has some limitations. It can currently perform simple evaluations of Natural Language Processing Tools.

The feature, which was emphasized the most during this project, was flexibility. The data input module and the processing module are very flexible and easily extendable. This ensures that these modules will serve as a very suitable foundation for further development. Asymmetric input data is currently not handled, but could be implemented in future revisions. Although speed was not a crucial factor for this project, it was coded to be as efficient as possible. During a program test a 1MB XML file was used as test input, even though the file was large the program still performed adequately and without being a memory burden. Files with different layouts were used during test runs, as to test the flexibility of the input system; all the tested files were processed with no problems. The usage of XML technology has greatly facilitated the development process, enhanced performance and introduced a degree of standard into the software. XML played an important role throughout the entire project and is used for such task as: configuration, output, and input. Because the evaluation tool interfaces with the XML framework via the standardized APIs: DOM and SAX. The evaluation tool is not just limited to the XML framework used. The developer can choose any implementation that supports the DOM and SAX APIs.
13 Future Enhancements and Suggestions

During the project’s course some ideas arose to which features are desired be present in future versions of the program. Some of these features would add additional functionality to the program, while others would make it more stable and efficient. Discussed in this section are some of these thoughts on enhancements to the program.

13.1 Synchronization of Input Data

This feature would enable the program to handle such things as: asymmetric input data and different kinds of errors in the input data. An error could be for example when a data set has been repeated several times. At first one can implement a simpler synchronization algorithm that would be able to handle simpler scenarios. Later on, develop a more advanced synchronization method, which could handle more advanced scenarios. Finding an algorithm that could synchronize an arbitrary number of different data sets in the best possible way without impacting program performance in a significant way could turn out to be quite difficult.

One idea is to use the synchronization algorithm from the diff program, which is part of the GNU diffutils. The diff command is traditionally used to show differences between two text files. There also exists a diff3 command, which works in the same way as diff, but can handle three files instead of two. The diff command uses the synchronization algorithm when it performs the file comparison and tries to determine their differences [6].

13.2 External Functions with Arbitrary Return Types

Currently only a Boolean value is supported as a return value from the external functions. If one wishes to output some other types of values from a function, one will have to send a reference to a variable as an argument to the function and let it populate this variable with the result. Currently a return value of Boolean type is sufficient. However, in future versions one may want to extend this functionality, by returning an arbitrary variable type from the external functions. This feature would ease in the creation of more complex functions and eliminate the need for “temporary variables”, which are currently used for returning data. The development needed to enable this feature, would involve modifications to the functor classes their base class. One suggestion is use the template feature of C++, it has been used before to enable arbitrary types of input arguments when developing the process module. Using the C++ template feature on the return value could be a first logical step in an approach.

13.3 Difference in Result between Sessions

Many times, it is interesting to know how the result from the current run differs from the result from the previous session. A session difference feature would enable the evaluation tool to output only the difference in the result from the previous run. An early implementation idea is that during the subsequent run, read the previous result into a DOM document and when creating the new DOM document compare the data with the previous document and only write the data that differed. This new DOM document would be written to a new file or would be used to overwrite the previous result file.
13.4 Input of Compressed Data

Sometimes the input data files can be very large and take up much data space on the media that they occupy. Compressing these files can often save much space. The problem comes when these files are to be read by the evaluation tool, the user will have to manually decompress them and then after the program has finished processing, recompress them again. Luckily, there are two compression libraries, zlib and libbz2. They use the gz respectively bz2 compression algorithms, which provide good compression rations, where bz2’s is a bit higher. They have APIs that resemble the ordinary C/C++ API for reading files, the only difference is that they can read compressed input files and decompress them on the fly while reading them, but leaving the file on disk in a compressed format. Using these libraries would be easy, just like performing ordinary file reading, but calling different function names and using a compressed file as input. The only drawbacks are the file reading times might be a little longer and that program will require one or two additional run-time libraries [3][21].
14 References


Appendices

A. A Typical Configuration File

```xml
<root xmlns="evalcfgfile"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="evalcfgfile cfg.xsd">
  <libfiles>
    <libfile type="template">tmpl.xml</libfile>
  </libfiles>

  <identify>
    <files>
      <file format="plain" type="in" name="datafile1">granska.txt</file>
      <file format="plain" type="in" name="datafile2">mxpost.txt</file>
      <file format="xml" type="in" name="datafile3">qtag.xml</file>
      <file format="xml" type="out" name="outfile1">out.xml</file>
      <file format="plain" type="filter" name="filter1">filter.txt</file>
      <file format="plain" type="filter" name="filter2">filter2.txt</file>
    </files>

    <fields>
      field(file("datafile1"),"\t","\t",var("word1"),var("tag1"));
      field(file("datafile2"),"_","\x20",var("word2"),var("tag2"));
      xfield(file("datafile3"),"contents","w","tag","#txtnode",
            var("tag3"),var("word3"));
    </fields>

  </identify>

  <process>
    filter(filt("filter1"),"_","_",var("tag1"));
    filter(filt("filter2"),"_","_",var("tag2"));
    genwcl(var("gender1"),
           var("wordcl1"),lookup("tag1"));
    gender(var("gender2"),lookup("tag2"));
    gender(var("gender3"),lookup("tag3"));
    wordcl(var("wordcl2"),lookup("tag2"));
    wordcl(var("wordcl3"),lookup("tag3"));
    inc(cnt("words","misc"));
    inc(cnt(lookup("gender1"),"genders"));
  </process>
</root>
```
inc(cnt(lookup("gender2"),"genders"));
inc(cnt(lookup("gender3"),"genders"));
majority(var("majortag"),lookup("tag1"),
lookup("tag2"),lookup("tag3"),lookup("tag1"));
if(not(eqstr(lookup("tag1"),lookup("tag2")));
  inc(cnt("errors","misc"));
if(not(eqstr(lookup("tag1"),lookup("tag2")));
  example(out("outfile1"),"examples",
    "onevstwo","word",lookup("word1"),"granska",
    lookup("tag1"),"mxpost",lookup("tag2"));
</process>

<processonce>
outputintcon(out("outfile1"),
  cntmap("misc"),"misc"); 
outputstr(out("outfile1"),"tag1",
  lookup("tag1"),"strvars");
</processonce>
</root>
B. A Typical Template File

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<root xmlns="evaltmplfile"
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xsi:schemaLocation="evaltmplfile tmpl.xsd">

  <template name="gender" args="2">
    regex(#1,#2,"utr|neu|utr/neu|mas");
  </template>

  <template name="wordcl" args="2">
    regex(#1,#2,"nn|pm|jj|rg|ro|vb|pc|ab|in|ha|dt|hd|ps|hs|pn
     |hp|sn|kn|pp|ie|dl|pl|uo|an");
  </template>

  <template name="genwcl" args="3">
    gender(#1,#3);
    wordcl(#2,#3);
  </template>

</root>
```
C. User’s Manual

This section is intended for the end-user that will be running the evaluation tool. It contains a detailed explanation of the configuration file, template file and the syntax for the scripting language. A listing of the internal functions and the basic set of external functions are also presented.

C.1 Program Operation

The program is a CLI program and is executed from a shell. The syntax for the program is as specified below.

```
autoeval -c <configurationfile> -o <outputfile> [Optional]
```

Where `<configurationfile>` is the full path to the configuration file. The `-o` switch is optional and can be used to specify a global output file without needing to specify it in the configuration file or it can also be used to override the global output buffer in the configuration file. When using this option the output buffer associated with the file is named “global”.

C.2 Configuration File

The configuration file is used define all the program’s options and scripts. The configuration file is divided into several sections. A simplified configuration file will be used as an example.
<?xml version="1.0" encoding="ISO-8859-1"?>
<root xmlns="evalcfgfile"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="evalcfgfile cfg.xsd">

<libfiles>
  <libfile type="template">tmpl.xml</libfile>
</libfiles>

<identify>
  <files>
    <file format="plain" type="in" name="datafile1">tagger1.txt</file>
    <file format="plain" type="in" name="datafile2">tagger2.txt</file>
    <file format="xml" type="in" name="datafile3">tagger3.txt</file>
    <file format="xml" type="out" name="outfile1">out.xml</file>
    <file format="plain" type="filter" name="filter1">filter1.txt</file>
  </files>
  <fields>
    field(file("datafile1"),"\t","\t", var("word1"),var("tag1"));
    field(file("datafile2"),"_"," ", var("word2"),var("tag2"));
    xfield(file("datafile3"),"contents","w", "pos","#txtnode",var("tag3"),var("word3"));
  </fields>
</identify>

<process>
  filter(filt("filter1"),"_","_",var("tag3"));
  gender(var("gender1"),lookup("tag1"));
  gender(var("gender2"),lookup("tag2"));
  gender(var("gender3"),lookup("tag3"));
  wordcl(var("wordcl1"),lookup("tag1"));
  wordcl(var("wordcl2"),lookup("tag2"));
  wordcl(var("wordcl3"),lookup("tag3"));
  inc(cnt("words","stats"));
  majority(var("majorgender"),
   lookup("gender1"),lookup("gender2"),
   lookup("gender3"),lookup("gender1"));
  inc(cnt(lookup("majorgender"),"genders"));
  if(not(eqstr(lookup("tag1"),
   lookup("tag2"))),
   inc(cnt("errors","stats")));
  if(not(eqstr(lookup("tag1"),
   lookup("tag2"))),
   example(out("outfile1"),"examples","onevstwo",
   "word",lookup("word1"),"tagger1",
   lookup("tag1"),"tagger2",lookup("tag2")));
</process>
</root>
First we have the root element of the configuration file it is called <root> and contains some static information that should not be changed.

Secondly, we have the section <libfiles>. This section could contain references to other files, which contain configuration data. Currently only one file type exists, the template definitions file type.

```
<libfiles>
  <libfile type="template">filename</libfile>
</libfiles>
```

Where filetype can currently only be template and filename is the full path of the definition file.

The <libfiles> section is optional and can be omitted if one does not wish to use templates.

Next we have the <identify> section, this section’s only purpose is to serve as separator, it however has important subsections. These subsections are <files> and <fields>.

The subsection <files> contains definitions of I/O files

```
<files>
  <file format="fileformat" type="filetype"
       name="buffername">filename</file>
</files>
```

The attribute fileformat can have the following values:

- plain – for plain text input files.
- xml – for xml input files

The attribute filetype can be either: in, out or filter.

- in – the file is an input file.
- out – the file is a output file.
- filter – the file is an input file which contains a definition of a filter.

When filetype is set to out, fileformat must be set to xml. When filetype is set to filter, fileformat must be set to plain.

The attribute name specifies the buffer/filter name that will be assigned to the file and finally the attribute filename is the full path to the file.
The second subsection, `<fields>` contains script functions that specify and acquire data sets from the buffers. The functions that can perform this task are called “field” for plain text files and “xfield” for XML files, they are part of the basic set of functions.

```plaintext
<fields>
  field(...);
  xfield(...);
  ...
</fields>
```

Finally, we have the two largest sections `<process>` and `<processonce>`. The `<process>` section contains a script that will be executed each time a new data set is fetched, while the script contained in `<processonce>` will be executed at the end of an evaluation run when all the data sets have been processed.

```plaintext
<process>
  function(...);
  ...
  function(...);
</process>

<processonce>
  function(...);
  ...
  function(...);
</processonce>
```
C.3 Scripting

The scripting functions are divided into two categories: internal and external functions. The internal functions are hard-coded into the program, while the external are user-written functions, which are mapped by the program.

The syntax of the scripting language is very straightforward and similar to that of the C/C++ programming language. Return values of internal and external functions can be used as arguments to other functions. The scripting syntax is shown below.

\[
\text{function}(\text{arg1}, \text{arg2}, \ldots, \text{argn})
\]
\[
\text{function1}(\text{function2}(\text{arg1}), \text{arg2})
\]

In the example below, the function “inc” takes an integer reference as an argument, the internal function “cnt” takes a string as argument and returns an integer reference, in this case to an integer named “words”.

\[
\text{inc}(\text{cnt}(\text{"words"}))
\]

The listing below show data types that are used as input arguments to functions and how they are written in scripts.

- **String** – string data, e.g. “words”
- **Integer** – a number, e.g. 123
- **Boolean** – true or false, e.g. true

### C.3.1 Templates

Templates provide an easy way of executing multiple script functions sequentially, by writing only a single line in a script. The templates are defined in a file separate from the configuration file. If one wishes to use a template definition file it must be specified in the configuration file. The template definition file has a root element called `<root>`, which contains static information that should not be altered. The template itself is defined inside `<template>` and `<template>` arguments specify the number of arguments that the template takes. To create a template one writes a script of functions that are desired to be included in the template. The syntax for this script is the same as the syntax used in the scripting sections of the configuration file, with the exception of a new keyword: “#n”. When the template will be resolved, the “#n” will be replaced with the n-th argument supplied to the template.
Below are two templates “gender” and “wordcl”; both these templates are called from the script used in the configuration file in section C.2.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<root xmlns="evaltmplfile"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="evaltmplfile tmpl.xsd">
    <template name="gender" args="2">
        regex(#1,#2,"utr|neu|utr/neu|mas");
    </template>

    <template name="wordcl" args="2">
        regex(#1,#2,"nn|pm|jj|rg|ro|vb|pc|ab|in|ha|dt|hd|ps|hs|pn|hp|sn|kn|pp|ie|dl|pl|uo|an");
    </template>
</root>
```

C.3.2 Data Types

Most data types that are associated with the scripting functions are self-explanatory, but not all. Listed below are some data types worth mentioning. These data types are used in conjunction with the scripting functions, so understanding them is required in order to create scripts.

- CntMap – a map that contains named counters, which are integer variables.
- DataMap – a map that contains named string variables.
- Dsource – an interface for input buffers.
- EvFilter – a filter object.
- Xoutput – an output buffer.

C.3.3 Internal Function List

These functions are meant to be used in conjunction with external functions. The sole purpose of these functions is data acquisition. The table below is a reference list, which contains function names, arguments, return types and short descriptions.

<table>
<thead>
<tr>
<th>Return type</th>
<th>Function name</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned int&amp;</td>
<td>cnt</td>
<td>string countername, string containername=&quot;global&quot;</td>
<td>Returns a reference to an integer variable</td>
</tr>
<tr>
<td>const CntMap&amp;</td>
<td>cntmap</td>
<td>string containername</td>
<td>Returns a reference to the countermap named containername.</td>
</tr>
<tr>
<td>Dsource&amp;</td>
<td>file</td>
<td>string buffername</td>
<td>Returns a reference to a data input buffer.</td>
</tr>
<tr>
<td>EvFilter&amp;</td>
<td>filt</td>
<td>string filtername</td>
<td>Returns a reference to a filter.</td>
</tr>
<tr>
<td>String</td>
<td>lookup</td>
<td>string variablename, string containername=&quot;global&quot;</td>
<td>Returns the value from a named string variable</td>
</tr>
<tr>
<td>Unsigned int</td>
<td>lookupcnt</td>
<td>string countername, string containername=&quot;global&quot;</td>
<td>Returns the value from a counter.</td>
</tr>
<tr>
<td>Xoutput&amp;</td>
<td>out</td>
<td>string stdoutname</td>
<td>Returns a reference to a data output buffer.</td>
</tr>
<tr>
<td>String&amp;</td>
<td>var</td>
<td>string variablename</td>
<td>Returns a reference to a named string variable</td>
</tr>
<tr>
<td>---------</td>
<td>-----</td>
<td>---------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>const CntMap&amp;</td>
<td>varmap</td>
<td>string containername</td>
<td>Returns a reference to the countermap named containername.</td>
</tr>
</tbody>
</table>

### C.3.4 External Function List

The script function name is the name that should be used when writing scripts; the function name is the actual C++ name of the function. Due to function overloading, many functions can be mapped to the same script function name. All the external functions return a Boolean variable.

<table>
<thead>
<tr>
<th>Script-function name</th>
<th>Function name(s)</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>basic_and_2, basic_and_3, basic_and_4</td>
<td>bool logic1, ..., bool logicn Where n=2,3 or 4</td>
<td>Performs a Boolean and-operation on the arguments and returns the result.</td>
</tr>
<tr>
<td>eqstr</td>
<td>basic_eq_str</td>
<td>string str1, string str2</td>
<td>Compares str1 with str2 and returns true if they are the same.</td>
</tr>
<tr>
<td>example</td>
<td>basic_example</td>
<td>Xoutput &amp;buffer, string section, string exName, string kvar, string kvardata, string var1, string var1data, string var2, string var2data</td>
<td>Write an example structure to the output file. To a specified section, with an example name, keyvariable name, keyvariable contents, and two variables, var1 and var2 and their contents.</td>
</tr>
<tr>
<td>field</td>
<td>basic_field</td>
<td>DSource &amp;buffer, string idelim, string edelim, string &amp;var1, string &amp;var2</td>
<td>This function specifies the data set for plain text files. From a specified input buffer. Inter data delimiter string and end-of-data delimiter string. It also takes two references to variables where the data set will be stored</td>
</tr>
<tr>
<td>filter</td>
<td>basic_filter</td>
<td>Evfilter &amp;filter, string idelim, string edelim, string &amp;var</td>
<td>This function substitutes a piece of data for another. It takes reference to a filter. Inter-data delimiter string and end-of-data delimiter string. It also takes a reference to the variable whose data will be substituted.</td>
</tr>
<tr>
<td>if</td>
<td>basic_if_con</td>
<td>bool con, ProOper func</td>
<td>This function takes a Boolean value, if true it will</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inc</td>
<td>basic_increment or basic_increment_byval increases a integer counter by 1 or if a second argument is present it is increased by that value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>majority</td>
<td>basic_majority5 string &amp;major, string var1, string var2, string var3, string draw This will function will perform a majority decision on three data strings, if there is a draw, the data from the draw variable will used instead. Result is written to major.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>basic_not bool logic Returns the inverse of a Boolean value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>basic_or_2, basic_or_3, basic_or_4 bool logic1, ..., logicn Where n=2,3 or 4 Performs a Boolean or-operation on the arguments and returns the result.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outputintcon</td>
<td>basic_output_intmap XOutput &amp;buffer, CntMap &amp;statmap, string section Outputs an entire countermap to a specified section to a specified output buffer.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outputstrcon</td>
<td>basic_output_strmap XOutput &amp;buffer, DataMap &amp;statmap, string section Outputs an entire datamap to a specified section to a specified output buffer.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outputint</td>
<td>basic_out_int XOutput &amp;buffer, string varName, unsigned int varData, string section Outputs an integer variable with name to a specified section and to a specified output buffer.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outputstr</td>
<td>basic_out_str XOutput &amp;buffer, string varName, string varData, string section Outputs a string variable with name to a specified section and to a specified output buffer.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rand</td>
<td>basic_rand_con unsigned int prob Takes a probability in percent (0-100). Randomizes and returns a Boolean value depending on the outcome of the randomization.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>regex</td>
<td>basic_regex string &amp;result, const string searchBuffer, const string rExpr Searches through searchBuffer, tries to match data using a regular expression, and then outputting the result.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xfield</td>
<td>basic_xfield DSource &amp;buffer, string root, string datanode, string data1, string data2, string &amp;var1, This function specifies the data set for XML files. From a specified input buffer. Root is the node above datanode. Datanode is directly above the nodes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string &amp;var2</td>
<td>that contain data set parts. Data1 and Data2 are attributes or text nodes that contain parts of the data set. Data1 and Data2 are directly below datanode. “#txtnode” is used to specify textnode, by using any other name an attribute node is assumed. xfield also takes two references to variables where the data set will be stored.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C.4 Filter File

The filter file is a plain text file that has a user specifiable format. In the example illustrated below, the first column contains the tag to be substituted, the second column delimited by “_” (underscore) is the tag that will take its place, each record is terminated by a “_*”. The format of the filter file is specified by giving arguments to the “filter” function.

Tag1format1_Tag1format2_
... TagNformat1_TagNformat2_

C.5 Result File

The result file could look like this simple example below. This file is what the output would look like if you used the example configuration file from section C.2. This particular result file has three sections: <examples>, <genders> and <other>. The examples section contains a structure generated by the “example” scripting function. The two other sections: <genders> and <other>, contain different string and integer variables, generated by “outputint” and “outputstr”. The result file also contains a time stamp.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<evaloutput date="Thu Jan 16 23:49:14 2003">
  <examples>
    <onevstwo word="raketvapen">
      <tagger1>nn.neu.plu.ind.nom</tagger1>
      <tagger2>nn.utr.sin.def.nom</tagger2>
    </onevstwo>
  </examples>

  <genders>
    <var name="None">1</var>
    <var name="neu">1</var>
    <var name="utr">1</var>
  </genders>

  <other>
    <var name="errors">1</var>
    <var name="words">3</var>
  </other>

</evaloutput>
```