Intelligent Writing Support for Second Language Learners of Icelandic Using Web Services

Guðmundur Örn Leifsson

Faculty of Industrial-, Mechanical Engineering, and Computer Science
University of Iceland
2013

INTELLIGENT WRITING SUPPORT FOR SECOND LANGUAGE LEARNERS OF ICELANDIC USING WEB SERVICES

Guðmundur Örn Leifsson

60 ECTS thesis submitted in partial fulfillment of a
Magister Scientiarum degree in Computer Science

Advisor
Hrafn Loftsson

Faculty Representative
Hjálmtýr Hafsteinsson

M.Sc. committee
Eiríkur Rögnvaldsson

Faculty of Industrial-, Mechanical Engineering, and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, August 2013

Intelligent Writing Support for Second Language Learners of Icelandic Using Web Services
Intelligent Writing Support for L2 Learners
60 ECTS thesis submitted in partial fulfillment of a M.Sc. degree in Computer Science

Faculty of Industrial-, Mechanical Engineering, and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Hjardarhagi 2-6
107, Reykjavik, Reykjavik
Iceland

Telephone: 525 4000

Bibliographic information:
Guðmundur Örn Leifsson, 2013, Intelligent Writing Support for Second Language Learners of Icelandic Using Web Services, M.Sc. thesis, Faculty of Industrial-, Mechanical Engineering, and Computer Science, University of Iceland.

Printing: Háskólaprent, Fálkagata 2, 107 Reykjavík
Reykjavik, Iceland, August 2013

Abstract

There are no ICALL (Intelligent Computer-Assisted Language Learning) applications for second language learners of Icelandic. This project aims to build a web service that facilitates ICALL applications and a website that highlights grammatical errors in free written texts. The website, Writing Support, was created for this purpose. Writing Support utilises a module in the web service for analysing texts and marking grammatical errors in them. It analyses and annotates Icelandic texts with the IceNLP toolkit and sends a TCF (Text Corpus Format) document to Writing Support which uses it to display a web page that highlights grammatical errors. Grammatical error detection in IceNLP was improved, and new types of error detection implemented. Two evaluations of the grammatical error detection were conducted. Participants wrote sentences describing two pictures. Participants in the first evaluation were 13 intermediate learners of Icelandic and in the second evaluation 26 beginners of Icelandic language. The first evaluation resulted in 76% precision and recall, and in the second evaluation the precision was 64.5% and recall 43%. Bad sentence structure of second language learners and spelling errors were probable causes of the poor accuracy of the grammatical error detection. The participants thought that Writing Support helped them in writing Icelandic, but the current precision and recall is too low to be utilised in second language teaching.

Útdráttur

Það eru ekki til nein greind forrit sem styðja við tungumálanám (e. Intelligent Computer-Assisted Language Learning, ICALL) hjá þeim sem eru að læra íslensku sem annað tungumál. Verkefnið snýst um að búa til vefþjónustu sem styður greind tungumálanámsforrit og vefsíðu sem merkir málfræðivillur í frjálsum texta. Í þeim tilgangi var vefsíðan Rithjálp (e. Writing Support) búin til. Rithjálp notar einingu í vefþjónustunni til að greina texta og merkja málfræðivillur í honum. Hún greinir og merkir íslenskan texta með IceNLP-hugbúnaðinum og sendir TCF (e. Text Corpus Format) skrá á Rithjálpina sem notar hana til að mynda vefsíðu þar sem bent er á málfræðivillur. Málfræðivillugreining IceNLP var bætt og greiningar á nýjum villum búnar til. Tvær kannanir voru gerðar til að meta málfræðivillugreininguna. Í þeim lýstu þátttakendur tveim myndum í texta. Í fyrri könnuninni voru 13 meðalgóðir íslenskunemendur, en 26 byrjendur í þeirri seinni. Niðurstaða fyrri könnunarinnar sýndi 76% nákvæmni og griphlutfall, en í þeirri seinni var nákvæmni 64% og griphlutfall 43%. Slæm uppbygging setninga hjá þeim sem læra íslensku sem sitt annað mál og stafsetningavillur voru líklegir orsakavaldar ónákvæmni við villugreiningu. Þátt-

takendum fannst Rithjálp hjálpa þeim að skrifa íslensku, en nákvæmni og griphlutfall er of lágt í núverandi mynd til þess að unt sé að nota við kennslu í íslensku fyrir útlendinga.

Contents

1 Introduction

Language learning has mainly been focused on drill-and-practice exercises for decades. This is true both for language learning with and without assistance of computers. With the rise of artificial intelligence, linguistic processing tools now can facilitate exercises that comprise more varied tasks than drill-and-practice. They can for instance help students with free written texts. Software that detects grammatical errors in free written texts does not exist for second language learners of Icelandic. The project described in this thesis is about making an Intelligent Computer-Assisted Language Learning (ICALL) website that facilitates second language learning of Icelandic by highlighting particular grammatical errors in free written texts. This website uses web service that carries out the linguistic analysis needed.

Icelandic is a morphologically rich language. That makes detecting the morphological properties of words, and annotating phrases and syntactic functions difficult. How well errors can be detected was unknown using the current state of Icelandic Natural Language Processing (NLP) tools, i.e. the IceNLP toolkit.

The website and the web service are made as part of the project 'A System Architecture for Intelligent Computer-Assisted Language Learning' (funded by NordPlus Sprog 2011-2013) to develop an architecture and a language learning program using it. The project is a collaboration between Reykjavik University, University of Iceland, and University of Gothenburg.

1.1 Project Goal

The aim of this project is twofold. First, building a platform based on web services that is capable of facilitating many distinct language learning programmes; and second, to use that platform to develop an intelligent writing support which highlights grammatical errors for second language learners of Icelandic. This thesis investigates the feasibility of creating such a platform and error detection.

A detection of grammatical errors is implemented as part of IceNLP. A web service is created containing IceNLP to allow requests to be made to do lexical analysis, including marking grammatical errors. A website is created for second language learners. The system is evaluated for accuracy of the grammatical detection, and how the language learners react to the system.

The thesis investigates whether it is possible to create a system that highlights grammatical errors in texts of second language learners of Icelandic that facilitates their learning in a productive manner, using a platform that supports various language learning methods.

1.2 Structure of the Thesis

Chapter 2 discusses language learning and how computers have been used in relation to it. It mentions how NLP started, and how NLP affected language learning with examples. Section 2.4 goes in details into various parts of IceNLP which are important and are used in this project for detecting grammatical errors. The project is described in chapter 3 without technical details. The technical details of the products of this project are discussed in chapter 4. The error detection mentioned above is explained in Section 4.2, including how errors are detected, coded, and marked. The web service that provides the error detection is detailed in the same chapter, going through all of its functionality and how it is used. The website that allows second language learners to enter free written texts and submit them to the grammatical error detection is discussed in Section 4.4, named Writing Support after the name of the website. In the following chapter it is shown how the web service can be used in the same way to mimic the functionality of the old IceNLP website. The website utilises functionalities of the web services, but has its own methods and functions.

Chapter 5 is about evaluation of the web services, how accurate the grammatical error detection is, and how participants in the evaluation perceived it. Two evaluations were conducted with second language students, first in 2012 and then again in 2013. Each part is discussed in separate sections, both having a conclusion subsection. The last chapter, chapter 6 Discussion and Conclusion, is contains over all conclusions of the thesis in broader terms. It also discusses the problems, solutions, and future work of the project in regard to error detection, improve student learning, and possible improvements to the web service that facilitates the language learning website that we built.

2 Background

2.1 Computer-Assisted Language Learning

Any form of language learning which is carried out using a computer, is called Computer-Assisted Language Learning (CALL). The material of CALL is often student-centred and interactive which facilitates independent learning of the student. It can be in the form of drill-and-practice programs or websites, with corpora[1] or concordances[2], and a varying degree of interactivity (?). These drill-and-practice programs are, as the word indicates, a repetitive practice of specific skills and helps memorisation of the learning material.

CALL has been accused of embodying old ways of understanding teaching and learning which is caused by lack of interaction between different disciplines, such as software developers, psychology, pedagogy, and other academic disciplines (?). Both online and offline CALL material has been created for facilitating children's first language acquisition of Icelandic, such as the CD-ROM "Stafakarlarnir" and number of web browser based sites hosted by The National Centre for Educational Materials (NCEM). For second language acquisition of Icelandic there are few available options of which Icelandic Online (see Section 2.2) is the most prominent.

One of the first notable CALL applications was made in 1959 by the University of Illinois with Control Data Corporation. It was called Programmed Logic/Learning for Automated Teaching Operation (PLATO) and was designed to teach Russian. PLATO used drill-and-practise exercises which included vocabulary and translation tests. Early CALL applications were influenced greatly by the learning material that was made for schools at the time, not utilising the flexibility which software was capable of. Those applications were made with exercises as they were in the textbooks (?). This means that the questions and answers were pre-programmed. If there were too few questions the students could memorise the questions and answers, learning the program but not the language skill.

[1]Corpora is a large set of text, usually machine-readable to enable computerised searchability, as discussed in ?.

[2]Concordance is an alphabetised list of critical words with their immediate context such as a list of words that are practised in vocabulary exercises.

Drill-and-practice CALL applications dominated the period around 1960-1980, but then less restricted CALL applications, such as games and simulations, started to emerge. Today those CALL applications coexist in education systems that rely more on books for language learning than CALL, and the same type of exercises are in CALL applications as are in the textbooks (?), but with more pleasing graphical interfaces.

2.2 Icelandic Online

The main internet resource for learning Icelandic for second language learners has been Icelandic Online (IOL; http://icelandiconline.is) since it opened its first web course in 2004. Today about 90,000 registered users have access to the web site (?). IOL has four courses, numbered from 1 to 4, which are open for anyone free of charge on their web site. However, students must register before accessing the learning material.

The web site is a pedagogically driven CALL application. The first two courses introduce the student to the structure and lexicon of the Icelandic language by the means of 40 pre-programmed objects of learning. Therefore, the courses are limited to vocabulary and grammatical exercises. With the launch of courses number 3 and 4, which were targeted towards immigrants, authentic videos, texts and interactivity were introduced onto the web site (?).

In one of the courses, the second language learners of Icelandic in IOL send teachers short written texts, and are given feedback by teachers. The teachers go through all texts and mark all errors (i.e. spelling errors, feature agreement errors, case errors in objects of verbs, etc.) by hand with special codes, which is highly labour intensive (?).

2.3 Intelligent Computer-Assisted Language Learning

The integration of NLP (see Section 2.4) tools into CALL has added a new intelligence to CALL application. This combination of NLP and CALL is called Intelligent Computer-Assisted Language Learning, or ICALL for short. It allows for more dynamic content for the learner than pre-programmed finite number of exercises.

With the inception of Artificial Intelligence (AI), a phrase coined by John McCarthy in 1956, the need for NLP arose for the need of human interaction with computers (?). The start of ICALL, however, was when AI was combined with CALL, and was called computer-assisted language instruction, incorporating AI techniques (CALI-AI). The main goal of CALI-AI was to enrich teaching software with enough AI to replicate how a teacher would teach languages. CALI-AI introduced NLP to CALL which enabled the software to check syntax of student texts for errors, and it provided more sophisticated feedback for drill-and-practice exercises (?), which is what ICALL is. ICALL is therefore the combination of various AI tools such as NLP tools with CALL. Those tools are the intelligent part of ICALL, which is a term which was coined at the end of 1980s while ICALL still was a relatively small field with few software implementations (?).

The NLP tools from AI allow those static pre-programmed exercises, from the earlier era of CALL, to be more dynamic. The tools would then in turn be able to generate both questions and answers for fill-in-the-gap and other forms of drill-and-practice exercises. An example of this is a tool that selects a sentence based on the learner's abilities from a corpora, and creates a question out of it. The answer to that question would then be generated by an appropriate NLP tool.

The reason why pre-programmed exercises are still being developed today could be that it is simple to make fill-in-the-gap and multiple choice exercises. If the developer is a teacher those drill-and-practice type of exercises are simple to make from a programming perspective. If the developer is a competent programmer, but not a competent teacher, it is difficult to think of other types of learning material. In the drill-and-practice exercises grammar, synonyms, syntax, and extra unexpected do not cause problems. However, when students are allowed to write relatively freely about a given subject those problems are a problem, making drill-and-practice exercise an irresistible choice (?). Furthermore, the field of computer science has shown lack of interest in joining CALL research projects, leaving CALL researchers to do the work of computer scientists, which can result in an unsatisfactory outcome. The reason appears to be a lack of challenge from the perspective of programming (?).

Here are few examples of modern ICALL applications.

- Tagarela[3] is an intelligent web based workbook for second language learners of Portuguese according to ?. Tagarela offers individualised feedback for incorrect sentences written by students in production (sentence building) exercises. This feedback tells the student the probable cause of the error and possibly even which word caused the error. This is achived by processing the text by various NLP-tools.

[3]http://sifnos.sfs.uni-tuebingen.de/tagarela/index.py/main

- E-Tutor[4] facilitates second language learning of German since its first implementation in 1999. It is a non-commercial German tutor on the web, consisting of a suite of open source tools written in the PHP server-side scripting language. E-Tutor offers the student sentence building, reading comprehension, and essay writing, where the student's progress is tracked and used for feedback. The feedback system prioritises which errors are displayed, depending on the student's past performance, and as well as the activity type which the student engages in (**?**).

- Robo-Sensei is a proprietary online application for learning Japanese. It consists of introduction, grammar, dialogue, vocabulary, Kana/Kanji, and tutor modules. Tutor is an NLP driven stand-alone program where students are allowed to produce their own sentences, receive comprehensive feedback on the nature of students' errors, and get points for writing correct Japanese sentences. If a student fails to write a sentence correctly three times, it produces the correct answer if the student wants to see the answer (**?**).

  The tutor module originated in 1997 when the project was named Banzai. Its processing is done with the NLP-tools listed by **?**, "a lexicon, a morphological generator, a word segmentor, a morphological parser, a syntactic parser, an error detector, and a feedback generator."

2.4 Natural Language Processing

Natural Language Processing (NLP) is a field of computer science which deals with the processing and analysing of natural languages. The main purpose of NLP is to detect characteristics of words, such as part-of-speech, and to determine the structure of texts (**?**).

NLP was originated by people like **?**, who recognised that it was possible to characterise languages which can be generated by grammars with state machines, and others who worked on formal language theory in 1950s. In 1960s the first functional NLP systems arose as part of AI, and in 1970s computer emulated robot acquired natural language understanding of simple instructions. Data-driven methods for part-of-speech tagging, parsing and attachment ambiguities, and semantics came in 1980s which became the dominant method of those tasks in 1990s (**?**).

Most NLP toolkits include a parser, a part-of-speech tagger, a text segmentator, and a morphological analyser. Text processing is often accomplished by first carrying

[4]www.e-tutor.org

out morphological analysis and part-of-speech tagging followed by a grammatical or syntactical analysis. After a text has been processed with NLP tools, a program can take its output and perform further specialised processing. Those tasks can vary, and NLP can be found in online automated assistants for customer service, speech recognition, artificial intelligence, grammar checkers, machine translation, automatic summarisation, sentiment analysis of online texts, and more applications.

2.4.1 IceNLP

IceNLP[5] is an NLP toolkit for the Icelandic language. It consists of different components for processing and analysing Icelandic, such as a tokeniser, a part-of-speech tagger, a morphological analyser, and a syntactic analyser (**?**). IceNLP is written in Java and is an open source project, licensed under the GNU Library or Lesser General Public License (LGPL). IceNLP is in three forms: a core, a daemon (**?**), and web service. The core includes the modules which are described below and can be run as a client on a local computer. The daemon is in two parts, a server and a client, where the server uses the processing done by the core of IceNLP to transmit results to the client. The web service is an Apache Tomcat[6] web application, which utilises IceNLP's core functions to generate feedback for the user.

2.4.2 Tokeniser

The first tool in IceNLP, which most texts will go through, is the tokeniser. It can receive a plain text and outputs tokens separated with white spaces, where a token is the smallest useful semantic unit or element of a sentence. Consider what we get from inserting "Hér kl. 14:30 voru sýndir munir frá 20. öld, þótti það 'kúl'!" ('Here at 14:30 o'clock were shown exhibitis from 20th century, it was considered 'cool'!') into the tokeniser.

```
Hér kl. 14:30 voru sýndir munir frá 20. öld , þótti það ' kúl ' !
```

2.4.3 IceTagger

The annotation of words with part-of-speech (POS) tags is the function of a tagger. A POS tag is an encoding of the word's class and morphological features such as case, number, and gender. POS taggers can be rule-based, can use probabilistic, or

[5]http://sourceforge.net/projects/icenlp/
[6]http://tomcat.apache.org/

even neural network models to decide POS tags of words (?). All of those models tackle the problem of ambiguity of words, where a word can have multiple possible POS tags, and disambiguate the words to select the most appropriate tags according to context. Because of this, solving ambiguity is the most problematic task in POS tagging.

The input which is to be tagged by IceTagger has to be a tokenised text. First IceTagger looks up the possible tags for each token in a lexicon, generated from the Icelandic Frequency Dictionary (IFD) corpus (?). IFD contains close to 600,000 words from various sources, which have been POS tagged mechanically and then corrected manually. If the token is found, a string of possible tags is assigned to it, otherwise it is marked as unknown. Those unknown tokens will be assigned a set of tags from IceMorphy (see Section 2.4.5), which analyses the suffixes of the unknown tokens and assigns appropriate tags accordingly. This means that a word could have been assigned multiple tags, which then are systematically removed, one by one by local rules with a window of two words on each side of the target word. This disambiguation uses 175 local rules initially and then applies heuristic global rules after that. The last step, global heuristics, performs grammatical function analysis, forcing feature agreement, and guesses prepositional phrases, removing inappropriate tags from each word. If there still is more than one POS tag left, there is a default rule to select the word's most frequent tag to annotate the word with (?).

The Icelandic POS tagset is large, as the language is morphologically rich. According to ? the IDF corpus has about 700 distinct tags, making accurate POS tagging harder for languages with smaller tagsets. The tag output by IceTagger consists of letters, each representing a different morphological feature in Icelandic. The first letter of the tag determines the word class which is followed by a string of letters, each representing the word's morphosyntax which is unique to each word class. Consider "smái hesturinn drekkur vatn" (the small horse drinks water) as an example of a tokenised string. The tagged output from IceTagger for this string is :

```
  Smái lkenvf  hesturinn nkeng  drekkur sfg3en  vatn nheo
```

Smái has the POS tag "lkenvf" where the letters represent l-adjective, k-masculine, e-singular, n-nominative, v-weak declension, and f-positive form. The tag for *hesturinn* "nkeng" translates to n-noun, k-masculine, e-singular, n-nominative, and g-suffixed definite article. The verb *drekkur* has the following tags: s-verb, f-indicative mood, g-active voice; 3-third person, e-singular, and n-present tense. *Vatn* is tagged n-noun, h-neuter, e-singular, and o-accusative.

Every letter in the POS tag represents a single morphological feature within each word class, but the letters may represent the same or different features for each word class. This can be seen when comparing the POS tags of the adjective *smái* and the verb *drekkur* in the example above. The POS tags of *smái* and *drekkur*, lkenvf and sfg3en, share three common letters, e, n and f. The letter e has a shared meaning in those tags, both words are singular. The n signals nominative case for *smái*, but present tense for *drekkur*. When word classes share a morphological feature the representation remains the same for both. But when word class does not share a morphological feature with others, then any letter can represent that feature. This can be seen in the example where the f has a distinct meaning for both words. In the POS tag for the word *smái* it represents a positive form, but for *drekkur* it represents that the mood of the word is indicative mood.

2.4.4 HMM+ICE+HMM

HMM+ICE+HMM is a combination of IceTagger and TriTagger. TriTagger is a re-implementation by **?** of the Trigrams'n'Tags (TnT) tagger developed by **?**, which is based on a Hidden Markov Model (HMM). HMM is a data-driven method used in POS tagging. The name Trigrams'n'Tags is derived from that TnT is of a second order HMM, that is a trigram, and is a tagger. TriTagger uses IceMorphy (see Section 2.4.5) to disambiguate unknown words in this re-implementation. TriTagger can be applied before and after IceTagger attempts to disambiguate the input further. This combination is called HMM+ICE, ICE+HMM, or HMM+ICE+HMM, indicating various combinations of IceTagger (ICE) and TriTagger (HMM). When HMM+ICE is performed IceTagger creates a tag profile for words and TriTagger eliminates the tags which imply an incorrect word class, then a standard IceTagger is run in order to select the most probable tag. This renders IceTagger to have a reduced tag profile for some words, making it more probable a correct POS tag will be selected. This improvement showed accuracy of 92.19%, an improvement from using IceTagger solely which had an accuracy of 91.59%. ICE+HMM uses IceTagger initially for populating POS tag profiles and then TriTagger disambiguates the words which IceTagger does not fully disambiguate. The combination of those two methods is HMM+ICE+HMM, and it has an accuracy of 92.31% (**?**). In the IceNLP toolkit, IceTagger can be used with or without any combination with TriTagger as explained above.

2.4.5 IceMorphy

Some words are not in the lexicon IceTagger uses, and hence their tags need to be guessed. IceMorphy deals with unknown words by three different methods, morphological analysis, compound analysis, and ending analysis (**?**). IceMorphy contains rules on inflection, and to which morphological classes each of the inflection belongs. These rules are used by the morphological analyser to look up endings of words and to assign the appropriate morphological class to it.

2.4.6 IceParser

IceNLP's shallow parser, IceParser, annotates the input text in two modules, phrase structure, such as noun and verb phrases, and syntactic functions such as subjects and objects (**?**). IceParser's input has to be of a special format. It has to be of the same format IceTagger outputs as described above, where every word is followed by its POS-tag. To differentiate the POS tags from words in the string they are prefixed and suffixed by ˆ (caret sign) and $ (dollar sign) respectively. This is done only for internal processing within IceParser's transducers.

Transducers

IceParser consists of 22 main finite-state transducers where each transducer's output serves as the input to the next one. Those transducers are written in JFlex[7], which is an open source lexical analyser generator. The transducers are written as rules in a form of regular expressions in JFlex.

```
NounTag = {encodeOpen}n{Gender}{Number}{Case}{ArticleChar}?
          {encodeClose}{WhiteSpace}+
```

The example above shows a description of noun tags using a regular expression. Tags are open and closed by ˆ ({encodeOpen}) and $ ({encodeClose}) respectively, which means a string like "ˆnken$" is a valid noun tag. POS tags are made up from individual letters that signal, word class, gender, number, and case in the same order as it is in the tag. The POS-tag-code's individual letters have to be strictly in the same order as they are defined in the rule, otherwise it is not accepted as a noun tag. Those variables in the expression above, with { and } brackets around them, have been defined to accept an appropriate letter in a POS-tag code. The tag in the example does not define an article, but the rule marks {ArticleChar} with a question mark, signalling it need not to be in the code. Hence, "ˆnken$" is a valid noun tag.

```
Noun = {WordSpaces}{NounTag}
```

The rule for what is a noun, is defined as any word that is followed by a valid noun POS-tag as explained above. {WordSpaces} is defined as any word followed by any number of white spaces.

[7]http://jflex.de/

With those two rules a transducer written in JFlex can manipulate the string that matches a noun as defined by the rule above. This manipulation is coded in Java which is the programming language IceNLP is written in. In order for IceNLP to be able to run the transducers they have to be compiled. JFlex compiles the rules, stored/written in flex files, and builds a non-deterministic state machine from them, which then is transformed into a deterministic state machine. In order for a Java application to be able to effectively execute the state machine, it transforms it into a Java class, which then can be compiled by a Java compiler into an executable program. This effectively allows a Java program to do complex string matching and manipulation using less resources than a pure Java code would use.

There are two main modules that annotate the input text, phrase structure module, and syntactic function module. Phrases are a single word or a group of words in a sentence that form a single meaningful unit in the sentence's syntax. If a phrase is made up by more than one word it can contain more phrases. Every word in a sentence belongs to a phrase within that sentence, and hence, every word in the sentence is tagged by the phrase structure module. The tag is made up of two letters, e.g. NP is a noun phrase and AP is an adjective phrase. The tag opens and closes on each side of the target phrase with a [sign opening the phrase on its left side, and] sign closing it on its right side.

The Icelandic sentence "Stóri maðurinn er glaður" ('The big man is happy') contains a single noun phrase, "Stóri maðurinn" which is made up from a noun and an adjective. This sentence is tagged by the phrase structure module accordingly:

```
[NP [AP Stóri AP] maðurinn NP]
[VPb er VPb]
[AP glaður AP]
```

Functions or syntactic functions, are the functional relationships the phrases have to each other in a sentence. A common example of this is a subject–verb–object relationship, as was seen in an example on page 8, "Smái hesturinn drekkur vatn" ('The small horse drinks water'). Similarly, in the example above "Stóri maðurinn er glaður" ('The big man is happy') which is a subject-verb-complement. Complement completes the meaning of a sentence, or complements a subject of a sentence. Therefore, "glaður" ('happy') is the complement of "Stóri maðurinn" ('The big man') in the example.

Functions are tagged in a different manner than phrases. Each type of function is coded with letters up to four in length (i.e. OBJ, SUBJ, and COMP), and is prefixed with an asterisk (*) sign to mark it being a function. In the same way phrases are marked to open and close on left and right side of the target word or words, so are the functions, but with { and } signs around the target phrase. Hence, the example "Stóri maðurinn er glaður" ('The big man is happy') will have two phrases marked by functions:

```
{*SUBJ [NP [AP Stóri AP] maðurinn NP] *SUBJ}
[VPb er VPb]
{*COMP [AP glaður AP] *COMP}
```

The phrase structure module consists of 14 transducers, the syntactic function module 8 (**?**), and 6 others which encode POS tags and clean the code.

The sequence of transducers is important in order to detect phrases and functions accurately according to the programmed rules. Consider the dependency chain of *Subject 1* transducer. Subjects in Icelandic are noun phrases in nominative case, therefore we must previosly have run the transducer *Noun phrase case. Noun phrase case* depends on, both *Noun phrase 1* and *Noun phrase 2*, their duty is to detect all noun phrases. Those two transducers depend on *Tag encoder* in order to identify the word class of a noun phrase, and *Noun phrase case* to identify its case. The order in which those transducers are carried out is therefore vital.

All transducers of IceParser are listed here below, with the name of the transducers followed by its file name in parenthesis. They are briefly described as they were at the start of this project, ordered in the same sequence as they are carried out in. Preprocess:

- Tag encoder
  Prefixes POS tags with a ˆ sign and suffixes them with a $ sign.
- Preprocess
  Adds a white space after the last tag if needed.

Phrase module's transducers:

- Foreign words (Phrase_FOREIGN.flex)
  Marks words, whose POS tag is 'e', with FRW or FRWs. The 'e' stands for "erlent orð" ('foreign word').
- Multi word expressions (Phrase_MWE.flex)
  Contains a lexicon to detect certain expressions like "allt í einu" ('all of a sudden') and "fyrst og fremst" ('first and foremost'). Expressions are marked with MWE_AdvP, MWE_AP, MWE_CP depending on whether the expressions are adjectives, adverbs, conjunctions, respectively.
- Multi word expressions 1 (Phrase_MWEP1.flex)
  Contains a lexicon to detect and mark multi word preposition expressions with MWE_PP that start with "fyrir", as in "fyrir handan" ('on the other side of').

- Multi word expressions 2 (Phrase_MWEP1.flex)
  Detects and marks the expressions that Multi word expressions 1 did not mark, like "þrátt fyrir" ('even though').

- Adverb Phrase (Phrase_AdvP.flex)
  Detects adverbs, conjunctions, subordinating conjunctions, and interjections and marks them with AdvP, CP, SCP, or InjP respectively.

- Adjective phrase (Phrase_AP.flex)
  Finds an adjective phrase, that can include a previously marked adverb phrase in front of an adjective, in the annotated text and marks the adjective phrase with AP.

- Adjective phrase case (Case_AP.flex)
  Adds the case of adjectives onto their AP annotations, such as APn for adverb in nominative case.

- Adjective phrases (Phrase_APs.flex)
  Groups and annotates two adjectives together with APs if there is a conjunction between them.

- Noun phrase 1 (Phrase_NP.flex)
  Finds and marks the same phrases as are done in Noun phrase 2 that are accompanied by additional words or phrases. As an example of this are words that have a POS tag that indicates a personal pronoun with or without a reflexive pronoun immediately after it, and a word tagged as a possessive pronoun followed by either adjective phrases and a noun, or both in the preceding order.

- Noun phrase 2 (Phrase_NP2.flex)
  All words with the following POS tags are annotated with NP: noun, proper noun, article, title, numeral, personal pronoun, possessive pronoun, indefinite pronoun, demonstrative pronoun, reflexive pronoun, and inter pronoun tags. In addition words such as "hver" ('who') and "hvaða" ('what') are also marked with NP.

- Verb phrase (Phrase_VP.flex)
  Marks verbs with VP, VPi, VPb, VPs, VPp, and VPg. VP are all finite verbs, which potentially can be followed by a marked adverb phrase and a supine. VPi is an infinite verb with or without an infinitive marker before it and a supine after it. VPs is a word tagged as a supine. If the last word for any of the three last markings is a form of "vera" ('to be') and "orðið" ('been'), then it will be marked with VPb instead of VP, VPi, or VPs. Words with POS tags that state the word is a verb that is in past participle are marked with VPp. VPg are all words that have the POS tag 'slg', which stands for a verb in present participle with an active voice.

- Noun phrase case (Case_NP.flex)
  Finds marked noun phrases and suffixes NP with a letter to indicate the case, where n stands for nominative case, a for accusative case, d for dative case, and g for genitive case. If the noun phrase contains an adjective, it has already had its case marked by Adjective phrase case, the noun phrase mark will have the same case suffix added. Otherwise the case is determined by the POS tag in the same way Adjective phrase case did, but with a noun.
- Noun phrases (Phrase_NPs.flex)
  If there is one or more proper noun followed by a noun phrase, where both have the same case, then the phrases are surrounded by NPs markings. That is also done to the aforementioned phrases only if they have either a noun phrase marked with NPg or a pair of noun phrases where the first phrase is NPn, NPa, or NPd and the latter noun phrase with NPg.
- Preposition phrase (Phrase_PP.flex)
  Marks preposition phrases.

Cleaning before function module:

- Clean 1 (Clean1.flex)
  Has three functions. The first function looks for a sequence of adverbial phrases, and combines them within a single adverbial phrase marked with AdvP. The second function searches for dative noun phrases that contain an adjective phrase of a nominative case. The function moves the adjective phrase outside of the noun phrase markings. The third function finds noun phrases that contain two proper nouns, where the second noun is a qualifier, and splits them into two noun phrases, one for each proper noun.

Functions module's transducers:

- Time expression (Func_TIMEX.flex)
  Finds and marks temporal expressions with TIMEX markings.
- Qualifier (Func_QUAL.flex)
  All one or more noun phrases in a sequence that has been marked to be of a genitive case are marked to be a qualifier.
- Subject 1 (Func_SUBJ.flex)
  Marks noun phrases of nominative case with the code SUBJ indicating a subject.
- Complement (Func_COMP.flex)
  Uses a detected subject to locate a nominative noun or adjectival phrase which is the subject's compliment.

- Object 1 (Func_OBJ.flex)
  Marks noun and adjective phrases with OBJ, if they are of accusative, dative or genitive case, and if they are related to a marked subject or stand with a verb that acts upon the object.
- Object 2 (Func_OBJ2.flex)
  Marks undetected objects, such as the ones followed by a verb and a previously detected object, where the object and the noun phrase are not of the same dative or accusative case. Also uses previously marked complements that include verb past participle phrases to mark any none nominative case noun phrases as objects. Previously marked subjects along verb phrases are used to detect nominative objects. It also uses previously marked subjects followed by a verb to mark complements if the following word is a nominative adjective.
- Object 3 (Func_OBJ3.flex)
  Marks dative noun phrases as objects if they stand with a marked complement that holds an adjective.
- Subject 2 (Func_SUBJ2.flex)
  Marks all nominative noun phrases as subjects.

Post process:

- Clean 2 (Clean2.flex)
  Removes cases of phrasal codes that have a case letter assigned to them.
- Phrase per line (Phrase_Per_Line.flex)
  Writes the output out one phrase per line.
- Tag decoder (TagDecoder.flex)
  Removes the ˆ and $ signs from POS-tags.

Each of the transducers for phrase modules and function modules surrounds the word it detects with an appropriate tag. For instance, when the *Noun phrase 1* transducer detects a word and immediately after it comes a POS-tag of a noun, starting with 'n', it surrounds the word and its POS-tag with "[NP" and "NP]" as can be seen in this example for "borðar fisk" ('eats fish'):

```
borðar ^sfg3en$ [NP fisk ^nkeo$ NP] . .
```

The case detector of noun phrases is then run and a letter is put at the end of a noun phrase opening code, representing which case the noun phrase is. This is done by examining the tag of the noun phrase, extracting the case, and injecting the appropriate letter which represent the case in hand. Case is always the fourth letter of a noun phrase's POS-tag. In the POS-tag of *fisk nkeo* from the example above, accusative case is coded as 'o' for "þolfall" in Icelandic, as POS-tags are coded in

Icelandic. The coding of phrases is in English, so when 'o' is found in a noun phrase the coding will suffix 'a' for accusative case as seen here:

```
borðar ^sfg3en$ [NPa fisk ^nkeo$ NP] . .
```

Functions such as subjects and objects are detected after the phrase module has been executed. Functions' code starts with {* then the type of function is indicated which can be SUBJ, OBJ, QUAL, or COMP. Subjects and objects indicate the position of verbs in a subject-verb-object phrase. In these kinds of relations, the position of the verb is indicated on the function's code via < or >, depending on which side the verb is to that function. In the example below "borðar fisk" ('eats fish'), the object's code includes a less than sign indicating the verb is on its left site.

```
borðar ^sfg3en$ {*OBJ< [NPa fisk ^nkeo$ NP] *OBJ<} . .
```

The rules, by which this is accomplished, are as follows. The NPAcc gets an 'a' suffixed on the noun phrase code as seen above. NPAcc is defined as "OpenNPa CloseNP" where open and closing of NP is "[NP" and "NP]" respectively. NP is defined as "OpenNP[adg] CloseNP", which means a, d, or g is suffixed on a noun phrase opening code, indicating accusative, dative or genitive case. Absence of a case letter is also a valid option. This NP is used when object is detected. Objects are defined to be matched to "(FuncQualifierWhiteSpace+)? (NP|NPs|AP|APs) (WhiteSpace+FuncQualifier)?" That is a qualifier may or may not be in front or at the end of a noun phrase (NP) or an adjective phrase (AP). Similar rules are in all transducers within function and phrase modules.

Error marking

Before this project started, IceParser had a simple error detection of noun phrase agreement. In that old version errors are marked by suffixing a question mark to either the noun phrase or subject. The detection of the noun phrase agreement is conducted within the *Noun phrase 1* and *Subject 1* transducers. Both transducers detect disagreement with a function that compared case, gender, and number of every word within noun phrases or subjects. If there is any disagreement the phrase or function is marked as seen here:

```
{*SUBJ> [NP? [AP litlu lhfnvf AP] barnið nheng NP] *SUBJ>}
```

Outputs

IceParser has four output formats: plain text, plain text with one phrase per line, XML and JSON. The last two are part of the daemon version of IceNLP, created for Apertium-is-en (?) which translates Icelandic into English. Plain text output is compiled throughout IceParser's process. XML and JSON output formats are built from this plain text format via a special output generator.

There are two ways in which IceNLP generates output from IceParser's output. The former method is to take the plain text output and return it, with the possibility to insert newline character after every period sign.

Plain:

```
{*SUBJ> [NP [AP Litla lhenvf ] barnið nheng ] } [VP
borðar sfg3en ] {*OBJ< [NP fisk nkeo ] } . .
```

Phrase per line:

```
{*SUBJ> [NP [AP Litla lhenvf ] barnið nheng ] }
[VP borðar sfg3en ]
{*OBJ< [NP fisk nkeo ] }
. .
```

The second method to generate output is via a class named OutputFormatter, which is part of IceNLP's daemon. It takes in plain text output from the parser and creates a tree out of it from each function, phrase, word and tag. That tree is then used to make an XML or JSON output.

Example of XML output:

```
<SENTENCE>
  <FUNC> {*SUBJ&gt;
    <PHRASE> [NP
      <PHRASE> [AP
        <WORDS>
          <WORD> Litla
            <TAG>lhenvf</TAG>
          </WORD>
        </WORDS>
      </PHRASE>
      <WORDS>
        <WORD> barnið
          <TAG>nheng</TAG>
        </WORD>
      </WORDS>
    </PHRASE>
  </FUNC>
  <PHRASE> [VP
    <WORDS>
      <WORD> borðar
        <TAG>sfg3en</TAG>
      </WORD>
    </WORDS>
  </PHRASE>
  <FUNC> {*OBJ&lt;
    <PHRASE> [NP
      <WORDS>
        <WORD> fisk
          <TAG>nkeo</TAG>
        </WORD>
      </WORDS>
    </PHRASE>
  </FUNC>
  <WORDS>
    <WORD> .
      <TAG>.</TAG>
    </WORD>
  </WORDS>
</SENTENCE>
```

Example of JSON output:

```
"Parsed Text":{
  "Sentence":{
    "{*SUBJ>":{
      "[NP":{
        "[AP":{
          "WORDS":[{
            "Litla": "lhenvf"
          }]
        },
        "WORDS":[{
          "barnið": "nheng"
        }]
      }
    },
    "[VP":{
      "WORDS":[{
        "borðar": "sfg3en"
      }]
    },
    "{*OBJ<":{
      "[NP":{
        "WORDS":[{
          "fisk": "nkeo"
        }]
      }
    },
    "WORDS":[{
      ".": "."
    }]
  }
}
```

2.5 Text Corpus Format

Text Corpus Format (TCF) is an XML-based format made to simplify the communication within the WebLicht[8] (E. Hinrichs, M. Hinrichs, and Zastow, 2010) tool chain, where each tool adds a new layer. TCF contains annotations, such as part-of-speech tags, lemmas, and tokens, in separate layers made by each tool (**?**). Most formats such as TEI[9], MAF, and TIGER-XML[10] are difficult to extend, but because of the layer structure of TCF, new types of annotations can easily be added as new layers (**?**). To extract each type of annotation, it can be found inside of a single layer within a TCF document. Each layer is marked in XML style open and close boundaries, making it only necessary to parse part of the document in order to extract the relevant data from it. These boundaries can be seen in the example below where a TCF input of an English tokeniser, containing the sentence "The small horse drinks water" has <text> and </text> around it:

```
<?xml version="1.0" encoding="UTF-8"?>
<D-Spin xmlns="http://www.dspin.de/data" version="0.4">
  <MetaData xmlns="http://www.dspin.de/data/metadata">
    <source>RU, Reykjavik University</source>
  </MetaData>
  <TextCorpus xmlns="http://www.dspin.de/data/textcorpus" lang="en">
```

[8]https://weblicht.sfs.uni-tuebingen.de
[9]http://www.tei-c.org/
[10]http://www.ims.uni-stuttgart.de/forschung/ressourcen/werkzeuge/TIGERSearch/doc/html/TigerXML.html

```
    <text>The small horse drinks water</text>

  </TextCorpus>
</D-Spin>
```

The English tokeniser could create a new layer and add it to the input in the output as seen below:

```
<?xml version="1.0" encoding="UTF-8"?>
<D-Spin xmlns="http://www.dspin.de/data" version="0.4">
  <MetaData xmlns="http://www.dspin.de/data/metadata">
    <source>RU, Reykjavik University</source>
  </MetaData>
  <TextCorpus xmlns="http://www.dspin.de/data/textcorpus" lang="en">

    <text>The small horse drinks water</text>

    <tokens>
      <token ID="t_0">The</token>
      <token ID="t_1">small</token>
      <token ID="t_2">horse</token>
      <token ID="t_3">drinks</token>
      <token ID="t_4">water</token>
    </tokens>

  </TextCorpus>
</D-Spin>
```

A TCF document which would be created after running the above example through a whole NLP tool chain might look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<D-Spin xmlns="http://www.dspin.de/data" version="0.4">
  <MetaData xmlns="http://www.dspin.de/data/metadata">
    <source>RU, Reykjavik University</source>
  </MetaData>
  <TextCorpus xmlns="http://www.dspin.de/data/textcorpus" lang="en">

    <text>The small horse drinks water</text>

    <tokens>
      <token ID="t_0">The</token>
  <token ID="t_1">small</token>
  <token ID="t_2">horse</token>
  <token ID="t_3">drinks</token>
  <token ID="t_4">water</token>
</tokens>

<sentences>
  <sentence ID="s_0" tokenIDs="t_0 t_1 t_2 t_3 t_4"></sentence>
</sentences>

<POStags tagset="stanford pos tagger">
  <tag ID="pt_0" tokenIDs="t_0">DT</tag>
  <tag ID="pt_1" tokenIDs="t_1">JJ</tag>
  <tag ID="pt_2" tokenIDs="t_2">NN</tag>
  <tag ID="pt_3" tokenIDs="t_3">NNS</tag>
  <tag ID="pt_4" tokenIDs="t_4">NN</tag>
</POStags>

<parsing tagset="stanford parser">
  <parse>
    <constituent cat="ROOT" ID="c_0">
      <constituent cat="NP" ID="c_1">
        <constituent cat="NP" ID="c_2">
          <constituent cat="DT" ID="c_3" tokenIDs="t_0"/>
          <constituent cat="JJ" ID="c_4" tokenIDs="t_1"/>
          <constituent cat="NN" ID="c_5" tokenIDs="t_2"/>
        </constituent>
        <constituent cat="NP" ID="c_6">
          <constituent cat="NNS" ID="c_7" tokenIDs="t_3">
        </constituent>
        <constituent cat="NP" ID="c_8">
          <constituent cat="NN" ID="c_9" tokenIDs="t_4">
        </constituent>
      </constituent>
    </constituent>
  </parse>
</parsing>

<depparsing tagset="stanford depparser" emptytoks="false"
 multigovs="false">
  <parse ID="d_0">
    <dependency govIDs="t_2" depIDs="t_0" func="det"/>
    <dependency govIDs="t_2" depIDs="t_1" func="amod"/>
    <dependency depIDs="t_2" func="root"/>
```

```
        <dependency govIDs="t_2" depIDs="t_3" func="dep"/>
        <dependency govIDs="t_2" depIDs="t_4" func="dep"/>
      </parse>
    </depparsing>

  </TextCorpus>
</D-Spin>
```

3 Project Description

The project, described in this thesis, is a part of the project 'A System Architecture for Intelligent Computer-Assisted Language Learning', whose aim is to develop an open source architecture for ICALL applications, in order for future projects to be able to reuse existing NLP tools and resources. Web service is to be created in this project that makes the NLP tools readily available on requests (**?**). The architecture has to be language independent when it comes to the languages the NLP tools are designed for, and the NLP tools have to be able to be plugged into the architecture. Additionally, it attempts to build a platform which encourages reusability of NLP-tools. The project is sponsored in 2011-2013 by NordPlus Sprog, and is a collaboration between Reykjavik University, University of Iceland, and University of Gothenburg.

The architecture is based on web services, on the principles of Service Oriented Architecture (SOA), where the user requests a text to be analysed or to provide other services. IceNLP Web Service is an ICALL platform which was built by the Icelandic team. IceNLP Web Service receives requests from users to perform analysis of Icelandic texts utilising IceNLP. The Swedish team developed LÄR språket via KorpusAnalys (Lärka). Lärka[1] is a web service that facilitates CALL (**?**).

Distinct applications for language learners are constructed to test and demonstrate the designed system architecture. The Swedish team provided Lärka. It has three main features. Two of which are POS and syntactic relations training for linguists, and one for language learners which trains vocabulary by multiple choice items. The Icelandic team created Writing Support. Writing Support's main purpose is in facilitating second language learning by reducing the workload of teachers. Writing Support is a website where second language learners of Icelandic can submit their Icelandic texts to be error marked. The website submits their texts to IceNLP Web Service for error marking and displays to the student the same text marked with certain common errors made by second language learners (**?**).

This project tries to answer the question if it is possible to build an automated error marking mechanism for the Icelandic language, in such a way that is useful to language learners. By that, the workload of teachers hand marking written texts in

[1]http://spraakbanken.gu.se/ws/larka

IOL is hopefully reduced.

Writing Support, through IceNLP Web Service, relies on the IceNLP toolkit in order to detect certain grammatical errors in Icelandic texts. Some errors are readily detectable in IceParser. An error detector analyses a parsed text to check for grammatical errors. Therefore, to avoid duplication of work, injecting the error detector into IceParser allows the error detector to detect and mark error simultaneously to IceParser's processing. Second language learners of Icelandic must be able to submit and receive feedback with certain errors marked on a website in a format easily understood by students.

The contribution of this thesis to the project is to provide the aforementioned architecture and a system that finds grammatical errors of second language learners of Icelandic in free writing exercises. The architecture of the platform required it to be easily accessible and support ICALL for any language. This platform has functions to analyse text with IceNLP, give human readable explanations of annotations that IceNLP outputs in plain text formats, and perform English language POS tagging. In order to highlight grammatical errors IceNLP had to be improved. A grammatical error detection and marking was implemented for this purpose, in addition to error codes to distinguish the error types. As part of this project the ability for IceNLP to output TCF was added. It is important in communication with external applications.

A new application called Writing Support is built to facilitate second language learning of Icelandic, by highlighting grammatical errors in free written texts. This application is constructed in the form of a website accessible to anyone with internet connection. In addition, the former IceNLP website was changed from being a server side JavaServer Pages, which creates the website, to be a website which makes requests to IceNLP Web Service for analysing texts.

The error detection is evaluated by having second language learners of Icelandic write short essays about a given subject, utilising this new Writing Support system to help them to avoid common grammatical errors. The accuracy of the grammatical error detection is a vital part of the evaluation.

4 System Development

A web service that receives requests was created with the ability to analyse Icelandic and English. This web service, called IceNLP Web Service, contains IceNLP. Enhanced abilities in detecting and marking grammatical errors were added to IceNLP. Writing Support, a website that facilitates second language learning of Icelandic, was created and uses IceNLP Web Service for detecting grammatical errors. This website reads TCF documents and converts them into a web page with a Perl script, creating an HTML and Java script code. Tools to read and write TCF were created in addition to making a new layer inside TCF documents in order to hold information about grammatical errors.

The TCF generator is the first subject of this chapter. Section 4.2 is about detection, coding, and marking of errors. It includes subsections describing how different errors are detected within IceNLP. Section 4.3 discusses the structure of IceNLP Web Service, and how its modules allow requests to be made on the internet, e.g. to analyse and detect grammatical errors in Icelandic texts. This web service can be used by websites such as those in Sections 4.4 and 4.5. The first section describes how the Writing Support website allows second language learners to enter free written texts and submit them to grammatical error detection in IceNLP Web Service, and the usage of TCF to highlight errors. The other section demonstrates how IceNLP Web Service is used to mimic the functionality of the old IceNLP website. The website utilises functionalities of the web services, but has its own methods and functions. The IceNLP daemon is described in Section 4.6. It was used in one of the two evaluations of the error detection of Writing Support before being replaced by IceNLP Web Service.

4.1 Text Corpus Format Generator

A common data format is vital for communication between different modules that could potentially be replaced. TCF was chosen for communication between client and server, and within the modules on the server. IceNLP uses a plain text format for its tool chain.

Multiple data models that are used for communication of web services and between NLP tools exist. The reason why TCF is chosen rather than TEI, Tiger or JSON is that TCF is simple and has every annotation needed for the project in its specification, with the exception of error annotation. An error marking layer could be introduced to TCF by adding a new layer for errors. TCF could easily be incorporated into IceNLP as IceParser has an XML output that could be utilised to build a TCF output.

In order to add a TCF output to IceParser's existing type of outputs there were two options, either to use the Java libraries from the WebLicht website, or write the TCF writer from scratch. We chose the second option since a custom built code is easier to adapt to requirements which are not known in advance, and is easy to create utilising IceParser's existing functions. The plan was to mark errors which is a new type of annotation for TCF.

The XML output of IceParser is generated from its plain text output. The plain text output is taken and split up where white spaces and new line characters are found. This is turned into a list of words, tags, and annotations which the XML generator uses to create layers. Each element in the list is a pair of content and an identifier. The content is a word, tag, or annotation, but the identifier will tell the XML generator which type of content it is: a word, words, a POS tag, a phrase, a function, or a sentence. The XML generator recursively makes a tree of those layers, creating lists, where each list contains data content as described and potentially a sublist, as seen in the example of XML given in the output Section of 2.4.6.

The new TCF generator utilises everything the XML generator uses to create output. The TCF generator goes recursively through the word lists, identifying each entry in the word list and stores: functions, phrases, POS tags, and words. The data collected from the lists are compiled in a variable as it will be represented in the TCF layer it belongs to.

Token layer

We have a storage called token variable, which will hold the contents of the token layer as seen in the example below. The example shows the contents of the token variable. When a word is encountered the token variable will add a new line with number of indents in accordance with the depth of the word in the tree. An identification number, prefixed by a *t*, is written in the token variable, along with the token in a syntax according to the rest of the TCF output. The identification number comes from a counter which keeps track of how many tokens have been stored. In the example below the contents of the token variable after encountering "Litla barnið" ('The small child') is shown:

```
<token ID="t1">Litla</token>
<token ID="t2">barnið</token>
```

Text layer

In the same way as the token layer has a variable which holds its contents, the text layer has its own text variable. When a token is found it is added to the text layer, which adds the encountered token to the text variable. The words are separated with a single white space between them as seen in the example below. The example shows what has been added to the text variable for the same input as in the previous example.

```
Litla barnið
```

Part-of-speech layer

The POS layer is compiled in the same fashion as the token layer. There is a POS variable which holds what will come in the POS layer as seen in the example below. A line is added to the POS variable when a POS tag is encountered. The line contains an identification number of the token and its POS tag. An identification number indicates which token the POS tag belongs. It is the same token number as is controlled by the function which forms the token layer. Here is an example in accordance with the examples above:

```
<tag tokenIDs="t1">lhenvf</tag>
<tag tokenIDs="t2">nheng</tag>
```

Constituent layer

As with the above layers, the main bulk of the constituent layer is added one line at a time in an appropriate string variable. The constituent layer differs from the others in one regard. The TCF generator writes a new line into the variable in a similar manner as with the POS tagging layer and the token layer, but as an open XML tag. When it encounters any constituent it marks it with an identification number, writing down its type, and the token identification number if it was a word. The identification number for constituents is prefixed with a *c* followed by a number from its own counter, holding the number of constituents it has encountered. This line is written as an open XML tag, but before it can be closed the rest of the tree has to be gone through to see what is written when it opens and closes. This can be seen in the example below where the three dots represent where additional constituents could potentially appear within the current constituent.

```
<constituent ID="c4" cat="AP">
  ...
</constituent>
```

Words are the other type of constituents that are written into the variable by the TCF generator. When a word is encountered it writes a closed XML tag, which includes the constituent identifier, token identifier, and phrase type. As an example of this we can use the example above and insert a word within the adjective phrase.

```
<constituent ID="c4" cat="AP">
  <constituent ID="c5" cat="AP" tokenIDs="t1"/>
</constituent>
```

Every time the TCF generator encounters a phrase it looks through its children in the tree to determine if they contain a word.

Error layer

In order to be able to return a text with highlighted grammatical errors in the text which the user inputs, it is essential to introduce an error layer to TCF. The errors have their own identification number prefixed with *e*, and are associated with a constituent in the constituent layer by the constituent's identification number code. The error type is written in code described in Section 4.2. An example of a line from the contents of the error layer is the following:

```
<e ID="e1" const="c3" type="Nn" />
```

Text Corpus Format example

After the list, described in section 4.1 above, has been fully gone through, the TCF document is compiled from the previously stored variables described above. The document is written out as seen in the example below, with each layer as described above with appropriate tags opening and closing that layer.

```
<?xml version="1.0" encoding="utf-8"?>
<D-Spin xmlns="http://www.dspin.de/data" version="0.4">
 <MetaData xmlns="http://www.dspin.de/data/metadata">
  <source>RU, Reykjavik University</source>
 </MetaData>
 <TextCorpus xmlns="http://www.dspin.de/data/textcorpus" lang="is">
  <text>Smáu hesturinn drekkur vatn</text>

  <tokens>
   <token ID="t1">Smáu</token>
   <token ID="t2">hesturinn</token>
   <token ID="t3">drekkur</token>
   <token ID="t4">vatn</token>
  </tokens>

  <POStags tagset="ifd">
   <tag tokenIDs="t1">lkfnvf</tag>
   <tag tokenIDs="t2">nkeng</tag>
   <tag tokenIDs="t3">sfg3en</tag>
   <tag tokenIDs="t4">nheo</tag>
  </POStags>

  <parsing tagset="ifd">
   <parse>
    <constituent ID="c1" cat="SENTENCE">
     <constituent ID="c2" cat="*SUBJ&gt;">
      <constituent ID="c3" cat="NP">
       <constituent ID="c4" cat="AP">
        <constituent ID="c5" cat="AP" tokenIDs="t1"/>
       </constituent>
       <constituent ID="c6" cat="NP" tokenIDs="t2"/>
      </constituent>
     </constituent>
     <constituent ID="c7" cat="VP">
      <constituent ID="c8" cat="VP" tokenIDs="t3"/>
     </constituent>
     <constituent ID="c9" cat="*OBJ&lt;">
      <constituent ID="c10" cat="NP">
       <constituent ID="c11" cat="NP" tokenIDs="t4"/>
      </constituent>
     </constituent>
    </constituent>
   </parse>
  </parsing>
  <errors>
   <e ID="e1" const="c3" type="Nn" />
  </errors>

 </TextCorpus>
</D-Spin>
```

As explained in Section 2.5 the TCF document contains layers in which different type of information is collected. Within the <text> </text> tags is the analysed text with a single white space between tokens. The text in the example is "Smáu hesturinn drekkur vatn" ('The small horse drinks water') where the word "smáu" ('small') is plural but the word "hesturinn" ('the horse') is singular. The IceParser output for this sentence is the following:

```
  {*SUBJ> [NP?Nn? [AP Smáu lkfnvf ] hesturinn nkeng ] }
  [VP drekkur sfg3en ]
  {*OBJ< [NP vatn nheo ] }
```

The tokens are all identified by a code. The word "hesturinn" is identified with 't2'. This will allow every layer to be able to refer to individual tokens. The following layer is the POS layer indicated by the *<POStags tagset="ifd"> </POStags>* tags. As seen here the tagset is based on the IFD (Icelandic Frequency Dictionary). The POS tag "nkeng" belongs to token with the identity code 't2', that is "hesturinn". Every POS tag is paired up with a token in this manner.

The next layer is the constituent layer, where every constituent has its own identity code. The constituents are both the structure and the words of the sentence, but not their tags. Constituent 'c6' contains a noun phrase, as the category of the constituent is marked to be 'NP'. This noun of the noun phrase is "hesturinn" as the token identity code is 't2'. The entire noun phrase can be found in the constituent layer marked 'c3'. The noun phrase opens and contains more constituents within it, 'c4', 'c5', and 'c6'. 'c4' is an adjective phrase indicated by being of category 'AP', which holds the constituent 'c5'. 'c5' is the adjective "Smáu" identified by being token 't1'.

The disagreement within the noun phrase "Smáu hesturinn" has been marked as an error in the layer below in the TCF document, the error layer. The error with code 'e1' is placed within 'c3', that is the noun phrase "Smáu hesturinn", to be of error type 'Nn' (see Section 4.2 for error codes). This error code refers to a noun phrase disagreement in number, were the upper case letter stands for the error being disagreement in a noun phrase and the lower case letter stands for an error in number.

4.2 Error Detection

Before the project started, IceNLP solely marked disagreement within noun phrases with a single question mark. The error detection has now been extended from that to marking five different error types with an error code. Icelandic Online (IO) provided a list of the ten most frequent elements in Icelandic which second language learners make errors in:

- Noun phrase agreement.
- Agreement between a subject and its complement.
- Agreement between a subject and a verb.
- Case of an object governed by a preposition.
- Case of an object governed by a verb.
- Topicalisation and permutation.
- Subject of an impersonal verb.
- Subjunctive mood in independent clauses and dependent clauses.
- Case in participles.
- Compound verb phrase with full or bare infinitive or participle.

The first five from this list were chosen to be implemented. Below are given examples for each of the five errors, where the words in grey colour are irrelevant to the error type. These errors will be discussed separately in this section. Noun phrase agreement is when all words within a noun phrase agree in gender, number, and case. This can be seen in the first example "litla barnið er svangt" ('the little child is hungry'), where "litla barnið" is the noun phrase. The second example is about agreement in gender and number, between a subject and its compliment. The example shows the same noun phrase as before and the complement "svangt" ('hungry'). An example of a disagreement of gender could be "litla barnið er svangur" where the noun phrase is neutral but the complement is masculine. The third type requires agreement in gender and number of the noun phrase and the verb "borðar" ('eat') in the example. The fourth error type addresses the case of a noun affected by a preposition, where the preposition requires the noun to be of a certain case. In the example "litla barnið borðar í eldhúsinu" ('the little child eats in the kitchen') "í" is the preposition and "eldhúsinu" is the noun affected by it. The fifth error type is about case of an object being affected by a verb. The example shows the verb "borðar" ('eats') preceding the object "fisk" ('fish'), where "fisk" is in the correct case, governed by the verb.

1. Noun phrase agreement
   Litla barnið er svangt.
2. Agreement between a subject and its complement.
   Litla barnið er svangt.
3. Agreement between a subject and a verb.
   Litla barnið borðar fisk.
4. Case of an object governed by a preposition.
   Litla barnið borðar í eldhúsinu.
5. Case of an object governed by a verb.
   Litla barnið borðar fisk.

An error is marked by appending the error code at the end of the phrase bound by question marks on each side. The code within the question marks a pair. The first part of the pair tells which type of error it was, and the second part tells what grammatical categories the error lies in. The first part, the error type, is a single upper case letter N, C, V, P, and A which stand for the first five errors in the list above in the same order as they appear. The second part of the code is a string of lower case letters g, n, c, and p which stand for gender, number, case, and person respectively. Here is an example of an error in gender agreement within the noun phrase "Litli barnið" ('The little child', where the case of 'little' is masculine).

```
[NP?Ng? [AP Litli lkenvf ] barnið nheng ]
```

In the occurrence of multiple errors within a phrase, each error is represented in a sequence between question marks. This can be seen by modifying the example from above. By introducing a disagreement in number in addition to gender "Litli börnin" ('Little children', where the case of 'little' is singular), we get the sequence of NgNn, which signifies a disagreement within a noun phrase in gender and number, as seen in the example:

```
{*SUBJ [NP?NgNn? [AP Litli lkenvf ] börnin nhfng ] }
```

At the end of the process of creating error detection for the five target errors, another type of error connected with the use of the auxiliary verbs "geta" ('can') and "hafa" ('have'), looked feasible to detect. The Icelandic verbs "geta", "hafa", and "fá" have restrictions in usage that will be explained below where the detection of the auxiliary verb error is described. In the error code the 'h' does not signal a grammatical category in Gh, but the first letter in "hafa", and the capital G stands for "geta". The entire list of possible error codes IceParser generates is:

- Ng, Nn, Nc, Np (noun phrase agreement)
- Vn, Vp (agreement between a subject and a verb)
- Oc (case of an object governed by a verb)

- Cg, Cn (agreement between a subject and its complement)
- Pc (case of an object governed by a preposition)
- Gh (auxiliary verb error)

IceParser's transducers detect grammatical patterns and place grammatical tags (such as NP and SUBJ) around phrases and functions. Within the transducers a particular sentence structure is matched by a regular expression and marked by the appropriate grammatical tag. Part of that process is to indicate whether there are grammatical errors or not.

Marking phrases or functions, to indicate an error, is accomplished by concatenating a list of errors, and then suffixing the NP tag with the error. In the example in Section 4.2.1 about Noun phrase agreement, the error list is 'NgNn'. A question mark is added in front and back of the error code to indicate that the following is an error code. This will in turn make the error code '?NgNn?'. This error code is then suffixed to the noun phrase tag as seen here:

```
[NP?NgNn?
```

Detection of errors is carried out at the same time as IceParser conducts parsing of tagged text. As explained in Section 2.4.6 the transducers detect the different syntactical categories and grammatical functions, and the transducers contain functions that tag them appropriately. This can be seen in the *Adjective phrase* transducer where an {AdjectivePhrase} is detected with a regular expression. When a match is found, the opening and closing tags for adjective phrase is put around the matched text, and the whole text is written out. Here in the example below AdjectivePhrase is the regular expression, and will be discussed further. out.write will write out everything that is within the parentheses. APOpen and APClose are the opening and closing tags of Adjectives, but "text" is the text that matched the pattern of the regular expressions.

```
AdjectivePhrase  { out.write(APOpen + text + APClose) }
```

The regular expression of AdjevtivePhrase in the example above is the following. An adjective phrase contains an optimal adverb phrase followed by an adjective. An adjective is any word accompanied by an adjective tag provided by IceTagger. Adverb phrases have already been marked with opening and closing adverb tags by the *Adverb Phrase* transducer, as seen on the list of IceParser's transducers in Section 2.4.6 starting on page 12.

```
AdjectivePhrase = {AdverbPhrase}? {Adjective}
Adjective       = {Word} {AdjectiveTag}
AdverbPhrase    = {OpenAdvP} ~ {AdverbTag} {CloseAdvP}
```

4.2.1 Noun phrase agreement

The detection of noun phrase agreement is conducted within the *Noun Phrase 1* transducer, and the detection of noun phrases is accomplished in a similar manner as is with adjective phrases. In addition to the capability to detect noun phrases, the *Noun Phrase 1* transducer had a notable additional capability to detect grammatical errors. Before this project had started, IceNLP was capable of detecting and marking disagreement within noun phrases and subjects. The method that was used to detect the error was to check gender, number, and case of every word within the noun phrase and compare each one to another, as long as there are two or more tags. If a mismatch was found, the noun phrase labelled brackets were suffixed with a single question mark as seen below in the example were "Lítill hjólið" ('The small bicycle' where the case of 'small' and 'bicycle' are masculine and neuter, respectively) has been annotated:

```
[NP? [AP Lítill lkensf AP] hjólið nheng NP?]
```

The programming functions that detected the disagreement and marked the error was located within the transducers *Noun phrase 1* (Phrase_NP.flex) and *Subject 1* (Func_SUBJ.flex). Each of those transducers contained copies of code to detect and mark the disagreement errors.

The current project expanded the error detection to detect more types of grammatical errors and code to represent each type of error. The transducers detect a desired pattern, and when found execute a function contained outside of the transducer, not within it as was done before. The code was moved to a special error detection and marking module. Minor changes were made. Instead of comparing every single tag to one another the list of tags is only carried out once, skipping comparison of tags if either of them does not have gender, number, or case. The list of tags in the noun phrase is gone through one tag at a time, and the first tag which has gender, number, and case is selected. Then after that the tag is compared to any following tags which is encountered within the noun phrase. Each time a mismatch is found in gender, number, or case, the error is stored. At the end of the tag list the errors are added to the noun phrase code.

The noun phrase agreement error detection is not conducted in *Subject 1* any more, only in *Noun phrase 1*. Limiting the detection to *Noun phrase 1* does not reduce the number of noun phrases checked for errors. In the transducer *Noun phrases 1* all noun phrases that potentially have agreement errors are found. In order for subjects to be detected, noun phrases have to have been marked. The list in Section 2.4.6, beginning on page 12, shows the order in which the transducers are carried out. There it can be seen that *Noun phrase 1* coming before *Subject 1*, discovers all noun phrases. *Noun phrase 2* detects single word noun phrases which *Noun*

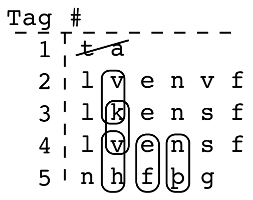


Figure 4.1: The noun phrase agreement checking five POS tags from the noun phrase "3ji litla skemmtilegur falleg börnunum".

phrase 1 has not labelled to be noun phrases. Those noun phrases can therefore not contain disagreement. Hence, removing the error detection out of *Subject 1* does not reduce the scope of error detection. The example shown in Figure 4.1 holds five POS tags from the noun phrase "3ji litla skemmtilegur falleg börnunum" ('the 3rd litle fun beautiful children'). This noun phrase is not consistent in gender, number, nor case. The tags 'ta', 'lvenvf', 'lkensf', 'lvensf', and 'nhfþg' are compared in pairs, tag number 2 and 3, 3 and 4, 4 and 5. The first tag does not hold gender, number, or case and hence the first comparison pair is skipped. With this method, in contrast to the original method of comparing every single POS tag, the number of comparisons are less. For this example the old method would do six comparisons, $(\frac{n(n+1)}{2})$, but the new method only n, that is four comparisons.

Let us consider what happens if the *Noun Phrase 1* transducer encounters a noun phrase that contains both disagreement in gender and number like "Litlu ljótu hesturinn" ('Little ugly horse'). In this example "litlu" is plural and of neuter gender, "ljótu" of same number and gender as "litlu", but "hesturinn" is singular and of masculine gender. The POS tags of the words which the error detector would get are then '^nhfn$', '^nhfn$', and '^nken$', with the tag encoding in front and back of the actual POS tag. The error detector extracts the POS tags by identifying the tag encoding, then it compares the first tag to the second tag, and then the tag to the third one, etc., as seen in the example below. In the comparison gender, number, and case is extracted from two consecutive POS tags. If a mismatch is found in gender, number, or case 'Ng', 'Nn', or 'Nc' is added to the list of errors for this phrase (see page 33 in section 4.2 for error codes). This list will be put together and suffixed to the NP opening tag as an error code. The detector first compares the first two POS tags, which are identical. Then the second and the third one '^nhfn$', and '^nken$'. The letters that represents gender ('h' and 'k') and number ('f' and 'e') do not match between the two POS tags. From this first 'Ng' is put on the list of errors, and then 'Nn' added to it, forming the sequence 'NgNn'. Finally the error code is added to the NP tag with a question mark between as seen in the example below after the transducer has finished.

```
[NP?NgNn] Litlu ^nhfn$ ljótu ^nhfn$ hesturinn ^nken$ [NP]
```

A difficult sentence for the noun phrase disagreement detection is for example "Barnið á blá skyrta" ('The child owns a blue shirt') where 'a blue shirt' is nominative but should be accusative. "Blá skyrta" agrees in gender, number, and case.

```
{*SUBJ [NP barnið nheng ] }
[PP?Pc? á ao [NP?NgNnNc? [AP blá lhfosf ] skyrta nven ] ]
```

The word "á" can either be the verb 'to own' or the preposition 'on'. In the example it was incorrectly detected as the preposition 'on'. The preposition dictates that the noun that follows it is accusative. The adjective "blá" that follows "á" can either be neuter, plural, and accusative or feminine, singular, and nominative. Therefore the tagger chooses the former of those two set of tags. The second word in the noun phrase cannot be tagged the same way and hence a disagreement in gender, number, and case is detected.

4.2.2 Agreement between a subject and its complement.

When detecting a disagreement between a subject and its complement, both the subject and complement must either have been discovered or being annotated by IceParser. There are two transducer which fulfil that requirement. In the transducer Complement, a subject and a complement next to each other, with or without a verb and an adverb, or a subject-verb-complement sequence are detected. Complement also annotates a complement that follows a subject following a verb like "Verður hesturinn þyrstur?" ('Will the horse become thirsty?') which is a verb-subject-complement relationship as it can be seen in the output of the transducer below. The second transducer that annotates complements following a subject is Object 2.

```
[VPb Verður ^sfg3en$ VPb]
{*SUBJ< [NP hesturinn ^nkeng$ NP] *SUBJ<}
{*COMP< [AP þyrstur ^lkensf$ AP] *COMP<}
? ^?$
```

When the complement in those transducers is found, the input is sent to the error module to be analysed. There the tags from the subject and complement are put in two separate lists. Then the gender of the subject and the complement are compared, if they disagree a *Cg* is added to the error. The gender is determined by the first noun in the phrase, if there is not a noun in the phrase the first tag which holds a gender will be select. The same is done with number.

If the subject has multiple nouns, as in the sentence "Maðurinn og konan eru falleg" ('The man and the woman are beautiful'), the gender of the complement can be neuter. When two or more words are in a noun phrase, which are of feminine gender, they are referred to by the complement in plural number and feminine gender. The same holds for masculine and neuter gender. When the nouns within the noun phrase disagree on gender, they will be referred to with a neuter gender. There is a separate mode for subjects with a single noun and multiple nouns. Noun phrases having more than one noun are denoted by [NPs and NPs], but noun phrases with a single noun is denoted by [NP and NP]. Detection of which mode it is of is therefore determined by the noun phrase labels NP and NPs.

```
{*SUBJ> [NPs [NP Maðurinn ^nkeng$ NP] [CP og ^c$ CP]
  [NP konan ^nveng$ NP] NPs] *SUBJ>}
[VPb eru ^sfg3fn$ VPb]
{*COMP<?CgCn? [AP falleg ^lvensf$ AP] *COMP<}
```

When NPs is encountered as in the example above it sees two different genders, and therefore demands a neuter gender of the complement as explained. If the compliment is not of the gender that was expected, then an error, Cg, is stored as explained above.

When a sentence has been incorrectly tagged or parsed the error detection can fail. The example below shows how the subject can be incorrectly detected for the erroneous sentence "það er ljóshærð kona talar við ungur drengur og hún er brosandi" ('there is blond woman talks to young boy and she is smiling'). The correct subject is "hún" in "hún er brosandi", but "ungur drengur og hún" has been erroneously marked the subject.

```
{*SUBJ> [NPs [NP [AP ungur lkensf ] drengur nken ]
  [CP og c ] [NP hún fpven ] ] }
[VPb?Vn? er sfg3en ]
{*COMP<?CgCn? [AP brosandi lvenof ] }
```

The example shows "brosandi" marked to disagree in gender and number with the subject. If the subject had been correctly marked, the error detection would correctly not mark the complement. The subject was incorrectly marked is "ungur drengur" in the sentence, as it should have been "ungan dreng". That is, "ungur drengur" was incorrectly written in nominative but not accusative case. Subjects are in the nominative case, just as "ungur drengur" and "hún" are. Therefore, "ungur drengur og hún" were marked to be the subject, meaning the compliment was expected to be third person and plural.

4.2.3 Agreement between a subject and a verb

The disagreement between a subject and a verb is detected within the *Subject 1* transducer as by then verbs have been marked, and within *Subject 1* the patterns verb-subject and subject-verb are discovered. In the error detector all POS tags from the subject and the verb are listed. If the subject holds multiple nouns, marked NPs, then the verb must be plural. If the subject is not marked NPs, then the number of the subject and verb are compared to see if they match. If they do not match 'Vn' is added to a variable that lists detected errors. Same applies for when a verb is singular but the subject is NPs. Person of subject and verb are compared for both NP and NPs. If a mismatch is found between the POS tags in person, then 'Vp' is added to the list of detected errors. After both number and person have been checked then the detected errors are added to the verb phrase annotation as seen here in the example "ég eru stór" ('I are big') where the subject is singular and first person, but the verb plural and third person. The string for the verb is displayed here:

```
[VPb?VnVp? eru ^sfg3fn$ VPb]
```

As in the previous section the detection can be wrong when the subject is incorrectly detected by the parser. In the example below "tré" is the correct subject that governs the morphological features of the verb "er". The example contains the first part of the parsed output of the sentence "við hliðina á tré er það en stöðuvatn sem er stór" ('next to a tree is that but lake that is big').

```
[PP Við ao [NP hliðina nveog ] ]
[PP á aþ [NP tré nheþ ] ]
[VPb?Vn? er sfg3en ]
{*SUBJ< [NPs [NP það fphen ] [CP en c ] [NP stöðuvatn nhen ] ] }
```

The error detection found a disagreement in the example between the verb "er" and the subject "það en stöðuvatn". The subject has incorrectly grouped together "það en stöðuvatn" detecting "en" as a conjunction connecting the two nouns, which are both of the nominative case. This makes the detection expect the verb to be plural, not singular. Therefore the verb is marked to to indicate a disagreement in number.

4.2.4 Case of an object governed by a preposition

When a preposition is detected and annotated within the *Preposition Phrase* transducer the error detector is called to go through the detected phrases. An example for an input is " við ao [NP vegur nken NP] " for the sentence "Maðurinn stóð við vegur" ('The man stood by a road') where "vegur" ('road') is in nominative case, as opposed to the correct accusative form "veg". The function removes everything except for POS tags. In the example POS tags would be 'ao' and 'nken'.

The first preposition tag which is found is taken and its case is extracted. In the example the letter which represents the case is 'o' which means the preposition governs the accusative. Then the case of the last tag is compared with the case of the first. If a mismatch is found the error marking of '?Pc?' is added to the preposition phrase markings as seen here:

```
[PP?Pc? við ^ao$ [NP vegur ^nken$ NP] PP]
```

Prepositions can be incorrectly tagged to be of other word classes. "Á" ('on') can be incorrectly marked as the verb "á" ('to own'). "Á" has been correctly tagged and marked in the sentence, written by a second language learner of Icelandic, "strákur situr í fanginu á mamman og þeir brosa til hver" ('a boy sits in the mother lap and they smile to each'), but the preposition "til" was incorrectly detected as an adverb. "Hver" in "til hver" is nominative but not genitive, the case governed by "til". Therefore "til" was incorrectly tagged not to be a preposition as then the following word would be genitive.

```
{*SUBJ> [NP Strákur nken ] }
[VP situr sfg3en ]
[PP í aþ [NP fanginu nheþg ] ]
[PP?Pc? á
 [NPs [NP mamman nveng ] [CP og c ] [NP þeir fpkfn ] ] ]
[VP brosa sfg3fn ]
[AdvP til aa ]
{*SUBJ< [NP hver fsken ] }
```

4.2.5 Case of an object governed by a verb

This detection is carried out in the *Object 1* transducer. By then verbs have been marked making the subject-verb-object pattern easily detectable. The object is either a noun or an adjective phrase with or without qualifiers, and the verbs and the subjects have been annotated.

Let us consider an example where a user sends the sentence "Drengurinn sparkaði bolti" ('The boy kicked a ball'). The object "bolti" is in the nominative case but should be "bolta" in dative case. The error detection function receives input in two parts, subject-verb and the object. The two parts for the example can be seen here:

```
1: {*SUBJ> [NP Drengurinn ^nkeng$ NP] *SUBJ>}
     [VP sparkaði ^sfg3eþ$ VP]
2: [NP bolta ^nkeþ$ NP]
```

First the verb phrase is isolated and the word extracted from the location of the end of where '[VP' occurs in the text and to the beginning of its POS tag marked with ˆ. The tag is also extracted, from the ˆ sign to the $. The case of the object is also found.

The verb is lemmatised with Lemmald, the lemmatiser module of IceNLP (?). The lemma of the verb is then looked up in a hash table. This hash table is generated from a file containing verbs and cases the verbs governs. For instance, "sparka" dictates that the case of "bolti" is dative, and that no other cases are grammatically correct. The hash table creates a list of possible cases after "sparka", which in our example is only dative. The expected case of the object is compared to the actual case. The expected case is dative but the actual detected case is nominative. When there is a mismatch an error is marked on the noun phrase with '?Oc?' and the output created with the noun phrase marked as an objective.

```
{*SUBJ> [NP Drengurinn nkeng NP] *SUBJ>}
[VP sparkaði sfg3eþ ]
{*OBJ< [NP?Oc? bolti nken NP] *OBJ<} . .
```

The first version of looking up what cases the verb governs was done utilising an SQL database. This required an external SQL database to be set up, and requests to be sent for every verb-object pattern detected. It was found simpler to have internal hash table to look up the verbs instead of an external service as an SQL database.

When the error detector encounters a sentence such as "konan á hvít skyrta og barnið á blá skyrta" ('the woman owns a white shirt and the child owns a blue shirt') the first verb "á" has been tagged with the correct POS. As with the example above the case of the object is correctly detected to be incorrect. The second "á" is incorrectly tagged as a preposition. This is caused by the word "blá" to be of a wrong case, it should be "bláa" in the accusative case but not the nominative case. Because of this the verb "á" is expected to be of the same case, leaving the more appropriate word class for the word to be preposition and not a verb.

```
{*SUBJ> [NP Konan nveng ] }
[VP á sfg3en ]
{*OBJ< [NP?Oc? [AP hvít lvensf ] skyrta nven ] }
```

```
[CP og c ]
[NP barnið nheog ]
[PP?Pc? á ao [NP?NgNnNc? [AP blá lhfosf ] skyrta nven ] ]
```

4.2.6 Auxiliary verb error

A new transducer was made to detect this pattern. The transducer is carried out in the post process between the *Clean 2* and the *Phrase per Line* transducers. This transducer looks for two verb phrases next to each other, potentially with an adverb between them. The first verb phrase should only hold the auxilary verbs "geta", "hafa", "fá". The second verb should be marked "VPb" or "VPi". "VPb" are any forms of the word "vera" ('to be'), such as "orðið" which is its past participle. Infinitive verb phrases are marked with "VPi". When a match is found to this error pattern in the transducer, a function is called to find the "[VPi" and modify into "[VPi?Gh?". If "[VPi" cannot be found nothing is done. An example of this can be seen below.

The idea was that the combination of the two verb phrases could be something a second language learner might write, and it would be discovered in testing whether the detection is useful. An example of such a sentence would be "drengurinn getur að borða" ('the boy can to eat'), as opposed to the correct "drengurinn getur borðað" ('the boy can eat').

```
{*SUBJ> [NP Drengurinn nkeng ] } [VP getur sfg3en ]
[VPi?Gh? að cn borða sng ] . .
```

This is the only error detection that does not utilise a special error detection function, but relies on the transducer itself. This makes the error detection faster in processing than the detection of the other errors that are detected.

4.2.7 Spelling mistakes

A spell checker is not part of grammatical error highlighting and should be handled before a request is sent for grammatical error detection. Therefore this feature has been removed, but was temperately part of the error detection as there was a problem with spelling mistakes skewing the accuracy of IceNLP. This made the error detection problematic. Some browsers and operating systems come with spell checker that would point out spelling mistakes. People that do not have spell checkers in their browsers need additional help. For the first evaluation GNU Aspell[1] was used for spell checking. The dictionaries used were a combination of two sources. The first source is the Big Icelandic Wordlist[2]. The second source is a list of word forms extracted from the Database of Modern Icelandic Inflection (**?**). The dictionaries were merged and tested on news reports. Additional words were added when discovered missing.

Aspell was called from the output generator of IceParser as a temporary solution which was part of the first user evaluation of Writing Support. When the TCF output creator, described in Section 4.1, encountered a token, Aspell was carried out as an external program with the token as standard input, the pipe and *dont-suggest* flags as parameters. A regular expression went through each output from Aspell to see if the token is found in a dictionary or not. The token was found in the dictionary when the asterisk sign (*) appeared as the first character of a line. When a token was not found then the output contains a number sign (#) followed by the token and the number of words found to be similar according to Aspell. The spelling errors were added to the TCF document to the error layer with a new code "Xx" for spelling mistakes. Therefore, the participants of the first evaluation received their spelling errors highlighted only after submitting their texts, allowing the spelling mistakes to skew the output of IceNLP and the error detection.

As stated previously, spelling error detection should be carried out before the grammatical error detection. In the second evaluation, in 2013, the aim was to have access to a new Icelandic context sensitive spell checker named Skrambi developed by **?**. In addition to looking up similar words to those that do not appear in Skrambi's lexicon, it also disambiguates certain words that have identical spelling but different meaning. Those words are distinguished by context utilising IceNLP and corrected accordingly. This context sensitive spelling error detection might be less accurate in erroneous texts of second language learners than in grammatically correct texts. Skrambi was not ready to be used in the evaluation of 2013 and hence spelling errors were not handled. This spell checker operates on websites so that all spelling errors would have been fixed before the user would submit text for grammatical error detection.

4.3 IceNLP Web Service

The architecture for the ICALL system built in this project, allows the user to send a request to a number of different language processing tools and select which one they

[1]http://aspell.net/

[2]http://helgafell.rhi.hi.is/pub/aspell-is/aspell-is-0.51.1-0.tar.bz2

want with a special parameter. IceNLP Web Service keeps two components of the old IceNLP website, the request capabilities from users and how IceNLP is handled. Utilising a web service to handle Writing Support's error detection (see section 4.4) detaches the web site from the analysis, leaving the front end and the analysis part of the website independent from each other. Furthermore, it allows other web sites and programs to have access the processing abilities of Writing Support. Because of the flexibility of the IceNLP module (discussed in 4.3.4) other websites, such as the updated IceNLP website, can make requests that fit their own requirements. When accuracy of text analysis is improved on the web service, all websites and programs that utilises it immediately benefit from it without having to implement it specially for each of those websites and programs.

The old website of IceNLP was hosted on a Apache Tomcat server as an HTML document. The result page of the website was dynamically generated through a Java Servlet. This Java Servlet received HTTP POST and GET requests. When a request was made the Servlet generated a result page for the user. Results were made with IceNLP in accordance to the request. The Servlet contained IceNLP, and kept it loaded in memory.

IceNLP Web Service builds on the foundation of the Java Servlet of the old IceNLP website, accepting POST and GET requests containing parameters that dictate how text analysis or other functions are conducted. A simple example of that can be seen here:

```
http://nlp.cs.ru.is/IceNLPWebService?query=Hann er ágætur
```

The layout of the platform holds three main parts which incoming request go through: a switch, filters, and execution of the requests. As seen in Figure 4.2, the switch directs the request to the appropriate module: IceNLP, DeCode (see Section 4.3.5), or Stanford POS Tagger (see Section 4.3.6). In the modules the request's parameters are filtered and set up before carrying out their main functionality.

This section discusses the main structures of the IceNLP Web Service. The first part is the request input to the server, called request on Figure 4.2. Section 4.3.2 describes the switch where requests are directed to modules where they will be processed. Section 4.3.3 describes the filter, which helps the modules set up the request for processing. The next sections, IceNLP Lexical Analysis, Decode tags, and Stanford Part-Of-Speech tagger are about the three modules that have been implemented. Those are the functions that users can request IceNLP Web Service to carry out and receive the results from.

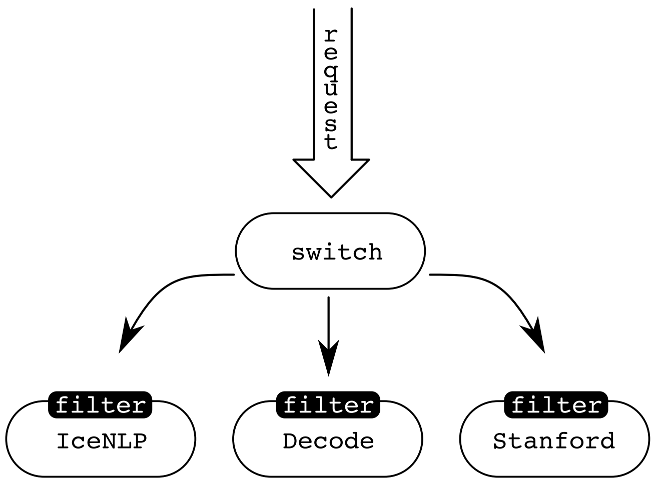


Figure 4.2: IceNLP Web Service's structure.

4.3.1 Input Requests

When deciding upon the architecture, the way in which the platform would receive requests were made. Two main methods were the most feasible. Simple Object Access Protocol[3] (SOAP) is a widely used method to send a requests to web services. It sends an XML information set that contains all information needed for the service to carry out the request. The other method was to use simple HTTP GET and POST[4] requests as suggested by the REST web service model. GET requests are easy to accomplish by any user. When a user enters a URL into a web browser with certain parameters, he is in fact carrying out a GET request. POST requests can be performed from most programs. The user can send a POST request through a form with a submit button on a website. The content of the form acts as the parameters in the request. Because of the simplicity of GET and POST requests, and the fact that they can be carried out by users solely by typing URL into a browser, it was chosen rather than SOAP.

4.3.2 Switch

The switch layer checks which function is to be carried out. The current version of IceNLP Web Service contains three functions: Icelandic text analysis through IceNLP, IceNLP tag decoding, and English language POS tagging with the Stanford Tagger.

[3]http://www.w3.org/TR/soap12-part1/
[4]http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

A request comes in to IceNLP Web Service switch function, which checks the 'mode' parameter. The mode parameter tells which module to execute: icenlp, stanford, or decodetag. If the mode parameter is not set, then by default 'icenlp' is assumed. When a match is found the appropriate module is carried out. This can be seen in the example below on how the Stanford POS tagger is called from the main IceNLP Web Service class with two variables, the parameters from the request and an object that will create a response to the user.

```
if (request.getParameter("mode").equalsIgnoreCase("Stanford"))
{
  stanbi.run(request, response);
}
```

This has to be done for all three modules, and it allows additional utilities such as loggers to be carried out before or after the module. Inside the module the parameters are set up with the filter as explained previously. Finally, with the parameters set up, the module can carry out its components, as will be explained below.

In case of IceNLP, logging of the incoming message is accomplished before the IceNLP module is carried out, and then the output is logged afterwards. Two different loggers were used, the older one was placed within the IceNLP module. Both loggers will be explained in Section 4.4.2.

4.3.3 Filter

A document with the configuration of the filter has been constructed with the rules about which parameters are allowed, which values the parameters can take, and what are their default value if none is specified. So that when a request with a parameter that is not specified in the configuration document is found, that parameter is not passed on to execution of a module. But if a parameter that exists in the configuration has a value that is not in the document, then a default value is given for that parameter. Here is an example of few lines from the configuration document:

```
name=stanford
function=built-in
query=$
tagger=wsjbi,wsjleft3,bi,left3, bidirectional,wsjbidirectional
```

This is the configuration of the filter for the Stanford POS tagger. The first line expresses that the following lines belong to the Stanford module. The next line states that the Stanford module is built into the Java code, i.e. that this is a function within IceNLP Web Service, but not an external source. External sources were not needed for IceNLP Web Service but will be further explained in section 6. Those two first lines have to appear in this order for each module with configuration rules of parameters, but after that the order of parameters rules is free.

The last line, in the example above, tells that the parameter 'tagger' allows the values 'wsjbi', 'wsjleft3', 'bi', 'left3', 'bidirectional', and 'wsjbidirectional'. All other values that are not specified and replaced with a default value are rejected. The default value is the first value after the equality sign. In the case of the 'tagger' parameter the default is hence 'wsjbi'. The other parameter that the stanford module takes is 'query', but its value can be any string. This is indicated with a dollar sign ($), but a number sign (#) indicates any integer number. No default value is specified for the 'query' parameter, but it can be assigned as seen in the second line in the example below. The default value in this example below for 'query' of the IceNLP module is "Þetta er staðlað inntak" ('This is a standard input').

```
name=icenlp
query=Þetta er staðlað inntak,$
tokenize=false,true // accepts true or false, default is false.
```

The third line shows a message that is commented out and does not affect the configuration. Everything that is written on the right hand side of double slash characters is not read by the filter. The parameter 'tokenize' is taken from the part of the document that specifies the configuration about IceNLP's parameters, but both the modules for Stanford and IceNLP have 'query' parameter. This is because the name of the module and the parameter name are joined together to become a unique key in a hash table. Two modules can have a parameter with the same name.

When IceNLP is initialised the hash table is created from the document that dictates the filter's configuration. The document is read line by line, and entries to a hash table are added appropriately. Each entry is a pair of a key and its value. In the case of 'tokenize' in the example above its key would be "icenlp_tokenize" and its value "false,true".

Keys in hash tables are used to look up and retrieve their values. The function that retrieves default values takes in the module name and a parameter name. It then puts it together to form a combination such as "icenlp_tokenize" as seen above. Then the function searches the hash table for a matching key and retrieves its value. In the example of "icenlp_tokenize" the value is "false,true". The function extracts and returns the string that is on the left hand side of the first comma. Therefore the function will detect "false" to be the default value.

The function that checks if the parameter's value is valid works in a similar way. It fetches the value based on the combination of a module name and a parameter name. Then it checks if the incoming value matches any of the values described in the configuration document. But as mentioned before, if the allowed value is indicated with a number or a dollar sign, then all integers or any strings are allowed in the incoming variable respectively.

```
boolean showparsing = Utils.strToBool(
  Config.makeValue(request, mode, "parsing") );
```

Before a module executes its functionality the input is filtered and variables which the function uses are set up. The variables are set by using the function described above. This function is called 'makeValue' and can be seen applied above. In this function validity is first to be checked, and if it is valid then the incoming value is set. If it is not found then the default value is set. Values that are integer or boolean, such as '6' and 'true', are converted from a string to the appropriate type of variable. In the case of the boolean value, if a string matches 'true' then a TRUE value is assigned. The boolean converting utility was shown in the example above. A function such as parseInt from the Java Integer class, which is used in the filter, is sufficient to convert a string number into an integer value.

If the filter allows a value that the module does not recognise, then a translation has to occur. In the case of the IceNLP module it allows many different values of the parameter which tells IceTagger if and how to use the HMM models. For only using IceTagger without HMM either 'icetagger' or 'none' is used. Both are accepted by IceNLP Web Service but solely 'none' is accepted by IceNLP. Therefore a conversion needs to be carried out that converts 'icetagger' into 'none'.

```
String model = Config.makeValue(request, mode,"tagger");
if (model.equals("icetagger"))
{
  model = "none";
}
```

Moreover, IceNLP does not accept a string value, but an object. This object has to be created and the value of the string dictates the usage of HMM in IceTagger accordingly. This is shown here:

```
IceTagger.HmmModelType modelType =
  IceTagger.HmmModelType.valueOf(model);
```

4.3.4 IceNLP Lexical Analysis

For lexical analysis of Icelandic, IceNLP Web Service relies on IceNLP as the name indicates. IceNLP is loaded into memory, as the old website's Java Servlet did, to reduce processing time of requests. IceNLP lexical analysis module can return output of IceNLP's tokeniser, IceTagger, and IceParser. Those three can be displayed or not displayed by a request of the user such as this one:

```
http://nlp.cs.ru.is/IceNLPWebService/?mode=icenlp
&parsing=true&tagging=true&tokenize=true
&query=Banani er gulur
```

This GET request will send the user the output of the tokeniser, IceTagger, and IceParser for the input of "Banani er gulur" ('A banana is yellow'). The value of the parameter: tokenize, tagging, and parsing can be true or false, to show or not show the appropriate output. If none of those three parameters are defined in the request, the default is to display the output from the tokeniser, to give the user at least some feedback. Those three different parts of IceNLP are executed separately for each part of the output. All output parts are separated by empty lines. The output is collected and sent back to the user as seen here below:

```
Banani er gulur

Banani nken
er sfg3en
gulur lkensf

[NP Banani nken ] [VPb er sfg3en ] [AP gulur lkensf ]
```

In the first evaluation of 2012 (see Section 5.1) a version of IceNLP was used as a server, which was a daemon. During the evaluation a problem was discovered. The daemon did not accept concurring requests. This resulted in an evaluation of how much load IceNLP Web Service could handle when IceNLP Web Service was made. The result of the evaluation (see Section 5.2) was that there was a potential problem. The problem was solved by making the main function synchronized, adding the synchronized keyword to its declaration in the code. By making the function, that tokenises, POS tags, and parses synchronized, a request will not interfere with another request.

4.3.5 Decoding tags

A function within IceNLP decodes tags and annotations into English or Icelandic. The module receives an input which a regular expression checks if matches a POS tag from IceNLP. If the input is a POS tag, then the decode function is carried out. The function assembles a reply which then is sent to the user.

If the regular expression does not recognise the input is a POS tag, but it has either { or [brackets, then the phrase function of the decoder is carried out to identify strings like "[NP".

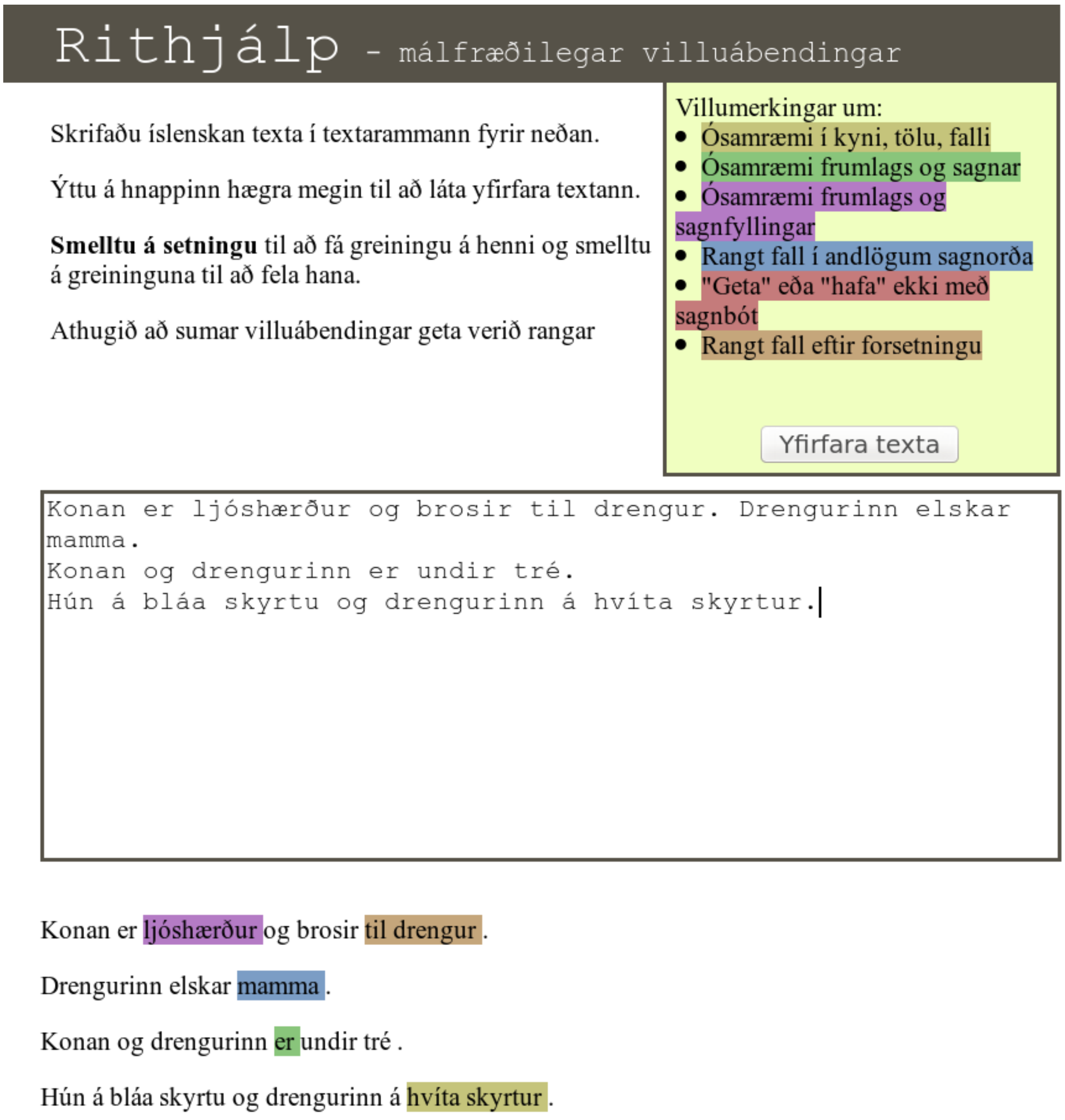
The decoding tags module accepts two parameters: the query string, and which language the user want the output to be in. The language can either be English or Icelandic. The decoder uses a utility function of IceNLP which takes each character of the POS tag and writes down a word or words to explain the tag. Explanations of each character is separated in the output by a semicolon. The POS tag of "Drengurinn" is "nkeng" which output from the decoder is "Noun; Masculine; Singular; Nominative; Suffixed article".

4.3.6 Stanford Part-Of-Speech tagger

This module allows POS tagging of English to be accomplished utilising the Stanford Log-linear Part-Of-Speech Tagger (**?**). It is an open source software licensed under the GNU General Public License. IceNLP Web Service is able to handle the Stanford module, as it is not specially geared towards Icelandic, but is language independent.

When the Stanford module's main function is called by IceNLP Web Service it does the same as the other modules and uses the filter. The module only utilises the 'tagger' and "query" parameters. The 'tagger' parameter allows two values in the filter to represent the same model which the tagger uses. This model is initiated and loaded into memory for each request, unless it has already been loaded in memory. Because of memory restrictions it was decided not to allow more than one model to be loaded into memory. If a new model has been selected to be used it is initiated and the functionality of the tagger is carried out. Because of those memory restrictions the Stanford module is disabled.

The tagging function first tokenises the query and puts each token in a list. Each sentence is put into a separate list, making this a list within a list where each sublist holds a sentence. Each list of tokens is taken and POS tagging is carried out, one sentence list at a time. The result is a list of word and POS tag pairs. To create an output, each element of the result list is output to a string, separating each element


Rithjálp - málfræðilegar villuábendingar

Skrifaðu íslenskan texta í textarammann fyrir neðan.

Ýttu á hnappinn hægra megin til að láta yfirfara textann.

Smelltu á setningu til að fá greiningu á henni og smelltu á greininguna til að fela hana.

Athugið að sumar villuábendingar geta verið rangar

Villumerkingar um:
- Ósamræmi í kyni, tölu, falli
- Ósamræmi frumlags og sagnar
- Ósamræmi frumlags og sagnfyllingar
- Rangt fall í andlögum sagnorða
- "Geta" eða "hafa" ekki með sagnbót
- Rangt fall eftir forsetningu

```
Konan er ljóshærður og brosir til drengur. Drengurinn elskar
mamma.
Konan og drengurinn er undir tré.
Hún á bláa skyrtu og drengurinn á hvíta skyrtur.
```

Konan er ljóshærður og brosir til drengur .

Drengurinn elskar mamma .

Konan og drengurinn er undir tré .

Hún á bláa skyrtu og drengurinn á hvíta skyrtur .

Figure 4.3: The Writing Support website with grammatical errors highlighted.

of the list with a new line character.

4.4 Writing Support Website

In order to demonstrate the abilities of the web service, a website was created to facilitate second language learning. The website was named Writing Support, or Rithjálp in Icelandic, and was hosted at http://nlp.cs.ru.is/rithjalp/. The content of the website includes a title, instructions in Icelandic on its usage, input text box, a submit button, and an iframe[5] where results are displayed.

There are five files making up the website, index.php, outputTextFeedback.php, TCFtoHTML.pl, and two Cascading Style Sheets (CSS) files named divSetup.css and errorHighlighting.css. The main file, index.php, generates the look of the website with help of divSetup.css, and submits text for processing to outputTextFeedback.php which inhabits an iframe on bottom of the website. When outputTextFeedback.php receives an input it prepares the input and sends it to the IceNLP web service. When it receives an answer from the webservice in the form of TCF it converts it into HTML and JavaScript, via a Perl script named TCFtoHTML.pl, which will be human readable when displayed in a webbrowser as seen at the bottom of Figure 4.3.

In the first user evaluation the website was able to highlight four grammatical errors. In addition to the error of a wrong case of an object governed by a verb there were three disagreements: within a noun phrase, between a subject and its compliment, and between a subject and a verb.

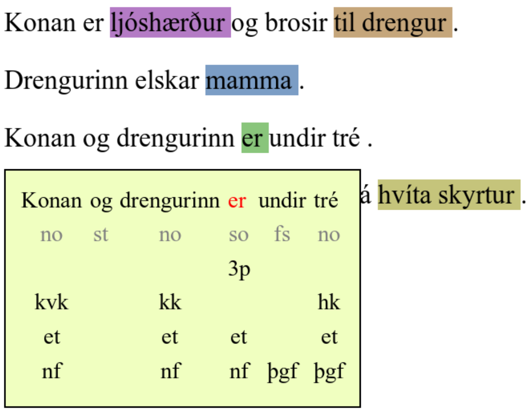
4.4.1 Text Corpus Format to HTML

Second language learners who tested the demo website needed to understand the error markings submitted by the web service. Because of that, it was decided to highlight the detected errors in different colours depending on the error type. Plain text output was not sufficient, as cryptic tags are created in the parsing process. Students had to receive plain text with the errors highlighted to know where the errors lie. The students also needed to have access to POS information about the words to help them to decide how to correct their grammatical mistakes. The TCF contains everything that is needed: the original text, the errors, and POS tags of the words.

A converter was created that changes TCF output from the web service into an HTML format. The Perl programming language was chosen because of its ease of use with regular expressions, which is a vital part of parsing the TCF input. The conversion process can be described in three steps.

- Reading the TCF document, extracting information from each layer and putting into separate arrays.
- Constructing a main array containing positions of where phrases and functions start and end.

[5] Iframe is a nested web page on another web page.



Konan er ljóshærður og brosir til drengur .

Drengurinn elskar mamma .

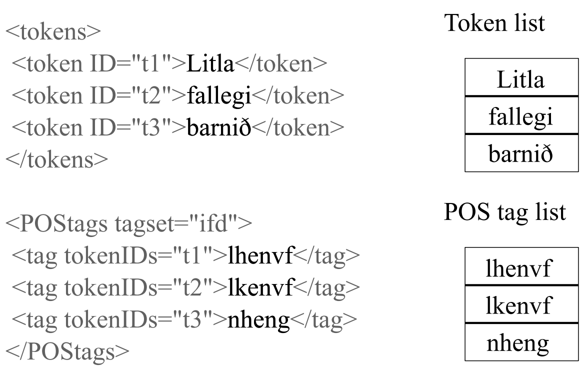
Konan og drengurinn er undir tré .

Konan	og	drengurinn	er	undir	tré
no	st	no	so	fs	no
			3p		
kvk		kk			hk
et		et	et		et
nf		nf	nf	þgf	þgf

á hvíta skyrtur .

Figure 4.4: Web page created from Text Corpus Format data with lexical data about a sentence the user selected to display.

- Compiling an HTML text output from the main array, marking every phrase with spans.



```
<tokens>
 <token ID="t1">Litla</token>
 <token ID="t2">fallegi</token>
 <token ID="t3">barnið</token>
</tokens>

<POStags tagset="ifd">
 <tag tokenIDs="t1">lhenvf</tag>
 <tag tokenIDs="t2">lkenvf</tag>
 <tag tokenIDs="t3">nheng</tag>
</POStags>
```

Figure 4.5: Tokens and POS tags are extracted from TCF and put into lists.

The TCF document is parsed one line at a time. The contents of token, POS tag, constituent, and error layers are put into several lists. When a token is encountered, recognised by the opening and closing of <token> tags, the token is added to the token list. Same is done for the POS tag list when <tag> is found. This is demonstrated in figure 4.5.

Constituents have four lists, which hold constituent identification number, category, token identification number, or type. Type is: a token, a constituent opening that contains other constituents, or the closing of a constituent. The four lists can be seen in Figure 4.6. Category of a constituent can be root or sentence, phrases, or functions. Type of a constituent structures the tree which the constituent layer forms. The constituent layer contains a tree of constituents, were tokens are the leafs. A type can be one of the following: a token, an opening constituent marking that holds more constituents within it, or a closing constituent markings.

constID	constCat	constTokenID	constType
1-c1	1-SENTENCE	1-	1-open
2-c2	2-*SUBJ	2-	2-open
3-c3	3-NP	3-	3-open
4-c4	4-AP	4-	4-open
5-c5	5-AP	5-	5-open
6-c6	6-AP	6-t1	6-token
7-	7-	7-	7-close
8-c7	8-AP	8-	8-open
9-c8	9-AP	9-t2	9-token
10-	10-	10-	10-close
11-	11-	11-	11-close
12-c9	12-NP	12-t3	12-token
13-	13-	13-	13-close
14-	14-	14-	14-close
15-	15-	15-	15-close

Figure 4.6: Tokens and POS tags are extracted from TCF and put into lists from the example in Figure 4.5

A list is created that holds the positions of constituents, the final output will be created from this list. For each line in the constituent layer, an entry is made in this list. When a constituent tag that opens is found, 'open' is written to the list named *constType*. When a constituent closes 'close' is written to the list. When a constituent tag is found that is neither an opening nor a closing tag, but holds a token identification code, then 'token' is written to the list as seen in Figure 4.6.

When an error is found the error code is put on the *errorType* list as seen in figure 4.7 and the constituent it belongs to in *errorConst*. With all errors known and *constType* showing the structure of the text, *mappedErrors* is created. The error codes are placed in the appropriate location within *mappedErrors* according to what constituent the errors belong to. The lists *constType* and *mappedErrors* are gone through one element at a time in parallel. A counter counts how many 'open' are encountered after seeing an error. The counter is decremented by one when 'close' is encountered, and writes 0 in *mappedErrors*. When the counter reaches zero an error in a constituent is closed. This is marked with 1 in *mappedErrors*. The start of an error is marked with the error code, and its end is marked with a 1 sign. The positions of opening, closing, and errors can be seen on Figure 4.7 where position 3 is where the error starts and position 13 where it ends.

The output is created with help of *constType*. The output is made putting HTML "<font>" around each constituent, and putting the appropriate class to its error if there is an error, and tokens. The list is gone through one element in the list at a time, keeping track of the position in a counter. When 'token' is encountered the appropriate token from the token list is put to the output. The position of the token is kept in *constTokenID* in a position in accordance to the number of the counter.

errorType	errorConst	mappedErrors
1-Ng	1-c3	1-
		2-
		3-Ng
		4-
		5-
		6-
		7-0
		8-
		9-
		10-0
		11-0
		12-
		13-1
		14-0
		15-0

```
1  2      3   4
   {*SUBJ [NP [APs
     5   6            7  8    9               10 11 12          13 14  15
     [AP Litla lhenvf ] [AP fallegi lkenvf ] ] barnið nheng ] }
```

Figure 4.7: Tokens and POS tags are extracted from TCF and put into lists.

If 'close' is encountered then "</font>" is written to the output. When 'open' is encountered "<font class="$error">" is added to the output, where $error is the error code matching the appropriate CSS class.

A Java Script is also generated to create the pop-up box seen on Figure 4.4 on page 52. An HTML paragraph tag is placed around each sentence in the output containing an onclick event calling a function that displays this box. POS tags from each sentence are put into separate arrays. Person, gender, number, and case are extracted from the POS tags. This information about each POS tag is written down for each token in the token list as seen on Figure 4.4.

4.4.2 Loggers

Two loggers were made for the evaluations, one implemented in Java and the other in Perl. The Perl logger was based on TCFtoHTML.pl (see Section 4.4.1), receiving a TCF file and compiling the original plain text format out of its layers before storing it in a log file. The Perl script was called by the IceNLP daemon when a request was made. Having external program as one of the platform's utilities was not the best option. This Perl logger was not used in the evaluations of Writing Support. Instead a Java implementation of this script was made as part of IceNLP daemon. This logger was part of the first evaluation (see Section 5.1) and was carried out for all error detection requests.

A modified version of the logger was constructed in order to log the input and output of IceNLP module in IceNLP Web Service during the second evaluation (see Section 5.3). Each user got a unique identity code to identify from which user requests are made during the testing session. The information needed for the log was a unique ID, the time of submitted text, the text it self, and the text returned to the user.

In the switch described in Section 4.3.2, the parameters are taken and sent to a utility function named logger before the IceNLP module is carried out in IceNLP Web Service. First a time stamp is created and written out to the standard output. Then the logger also prints out the string in the 'query' parameter, along with a user identity code fetched from the 'user' parameter of the request. There was a problem with identifying users on IceNLP Web Service, because the Writing Support website sends the request and therefore is the user. Hence, all users will be given the same identity code. The solution was to create an identity code on the Writing Support website with PHP commands to get the client's IP number and session ID. Those two numbers are combined and sent to IceNLP Web Service as a parameter. The query is written to standard output along with user ID and a time stamp. The standard output is directed into a file named catalina.out by Apache Tomcat.

The output of IceParser is logged inside of the IceNLP module directly after the input has been analysed. If the output format is anything else than TCF it is written to catalina.out directly. But if it is TCF a utility function named TCFtoHTML is called. It works in the same way as the TCF to HTML function described in Section 4.4.1. First it creates lists of tokens, POS tags, and constituents. Then it uses the constituent list to create the output with the same markings, 'open' for start of a phrase like '[NP', 'close' to close a phrase ']', and then tokens and POS tags. This process will transform the TCF into a plain text output format as seen below, and then it will be put onto to the standard output to be written into the catalina.out file.

```
{*SUBJ> [NP [AP Litla lhenvf ] barnið nheng ] }
[VP borðar sfg3en ] {*OBJ< [NP fisk nkeo ] } . .
```

4.5 IceNLP Website

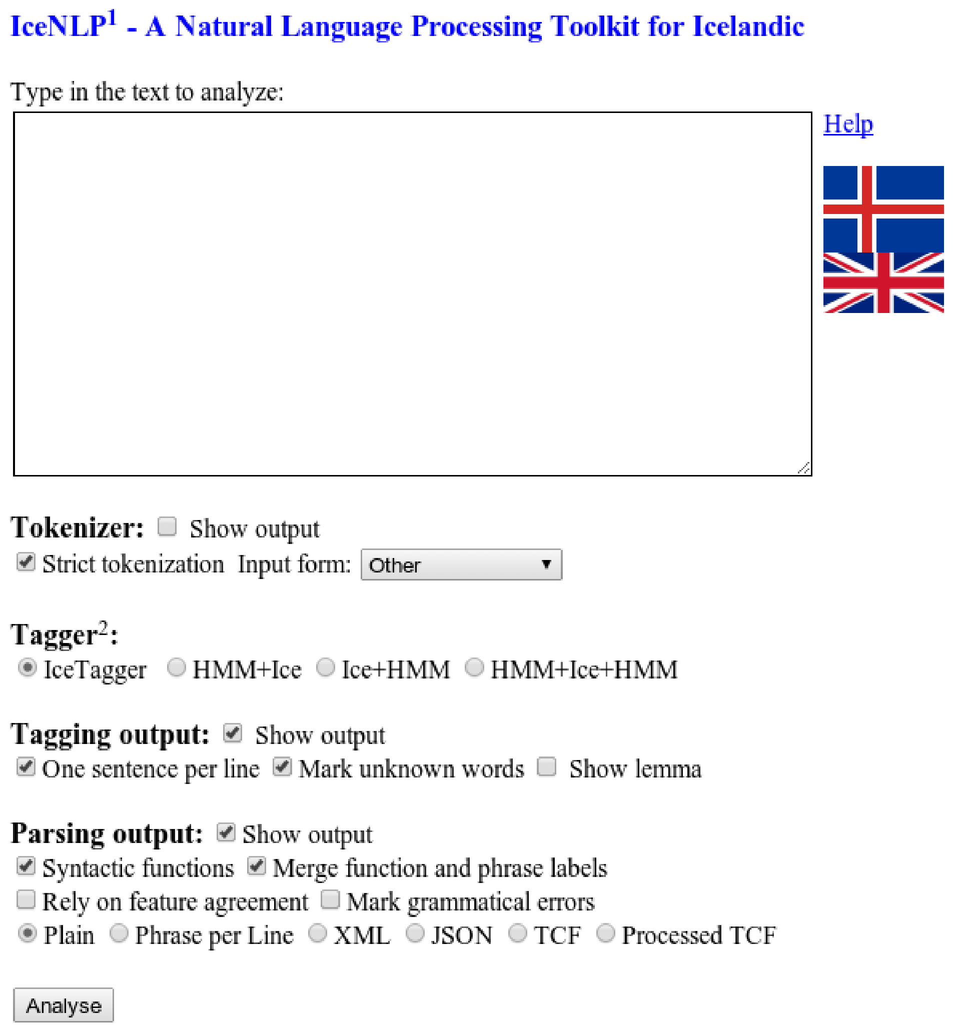
Figure 4.8: IceNLP's shortcut icon.

The original IceNLP website at http://nlp.cs.ru.is/ was on an Apache Tomcat server with a static HTML page as a main page and a JSP (JavaServer Pages) web application for a result page. The application generates HTML code of the result page for each request. IceNLP's tokeniser, tagger, and parser are part of the application. When a visitor requests text to be analysed, the application uses IceNLP to generate the results and displays them. With IceNLP Web Service it is possible to separate the generation of the website and the NLP tools.

A new website was constructed with the same appearance and functionality as the old website had. This website has no lexical analysis capabilities on its own, instead the website requests the IceNLP Web Service (see section 4.3) for all such processing. The website consists of two main files, an index page (see Figure 4.9 on page 57) and a result page. Both pages were represented with an icon shown in Figure 4.8. The index page has all the options of how the analysis is to be conducted. When the user clicks on the analyse button, all of the options which the user selected are directed to the result page. This is done by sending the content of the form with the POST request method. The result page takes those options and forms another POST request to the IceNLP Web Service. When feedback is received, the result page displays the results from the IceNLP Web Service. The two servers, an Apache HTTP server and Apache Tomcat, are hosted on the same computer. The locations of servers are irrelevant when everyone has equal access to do requests to IceNLP Web Service.

Additional options were added to the new website that did not exist in the old one. Now there is given a choice to show outputs from individual parts of IceNLP, the tokeniser, the tagger, and the parser. On the old website the user was shown all three outputs in the results. In addition to the two output formats of plain text format and phrase per line, four new formats were introduced : XML, JSON, TCF, and processed TCF. Text in the three basic formats is written on the web page without extra formatting as it is returned from the IceNLP Web Service, but not the processed TCF. The processed TCF is generated by a Perl script (further explained in Section 4.4.1) that extracts the text, and marks the errors. Furthermore, the script also gives the user the option to click on a sentence and see the words' class, person, gender, number, and case.

The old version of the website allowed the user to hover the mouse cursor over a POS tag, on the results page, and display an explanation of it in English or Icelandic. This functionality has been implemented for both plain text and one phrase per line outputs. A function was made that processes either output type, and receives text from IceNLP Web Service and inserts HTML span tags around all lexical annotations of the input text. This is accomplished by sending every word in the received text to IceNLP Web Service as a decode tag request (see Section 4.3.5). If the result is not a question mark it is an explanation of an annotation that needs to be put into spans. The explanation can be seen when the user hovers the mouse over IceNLP's annotations as seen in Figure 4.10. The function requests IceNLP Web Service decode function to analyse each word, where a word is any string between white spaces. When decode tag request is replied to with an explanation of an annotation



IceNLP[1] - A Natural Language Processing Toolkit for Icelandic

Type in the text to analyze:

Help

Tokenizer: ☐ Show output
[✓ Strict tokenization] Input form: Other

Tagger[2]:
[✓ IceTagger] HMM+Ice Ice+HMM HMM+Ice+HMM

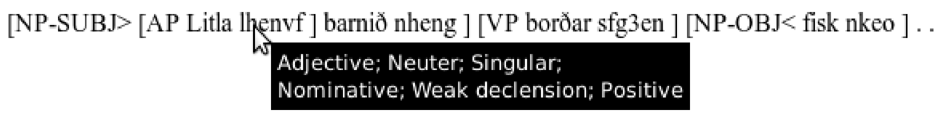
Tagging output: [✓ Show output]
[✓ One sentence per line] [✓ Mark unknown words] ☐ Show lemma

Parsing output: [✓ Show output]
[✓ Syntactic functions] [✓ Merge function and phrase labels]
☐ Rely on feature agreement ☐ Mark grammatical errors
[✓ Plain] Phrase per Line XML JSON TCF Processed TCF

Analyse

Figure 4.9: IceNLP's index site.

a span, containing the explanation, is created around the word.



[NP-SUBJ> [AP Litla lhenvf] barnið nheng] [VP borðar sfg3en] [NP-OBJ< fisk nkeo] . .

Adjective; Neuter; Singular;
Nominative; Weak declension; Positive

Figure 4.10: IceNLP's result site POS tag decode.

4.6 IceNLP Daemon

Before the work of this project started there was a daemon version of IceNLP as briefly described in Section 2.4.1 on page 7. It has a server part and a client part which functions in the following way. When the server part is executed it initiates in the same way IceNLP does in IceNLP Web Service. There all IceNLP functions are present with the addition of socket communication. The server accepts requests through a socket defined in a configuration file. The client sends requests to this socket, the server would then parse the text with IceParser and send back to the client which would write the results to the user's screen.

4.6.1 Output Format Requests

This part of IceNLP was considered to be used for a server and was part of the first evaluation (see Section 5.1 on page 61). It was made to generate TCF, XML, and JSON formats in addition to plain text and phrase per line outputs. A feature was added so the user could request which formatis chosen for the output. The user can enter a string, as shown below, into the standard input of the client. The server then checks what is inside the brackets and sets the output accordingly.

```
[tcf]Litla barnið borðar fisk.
```

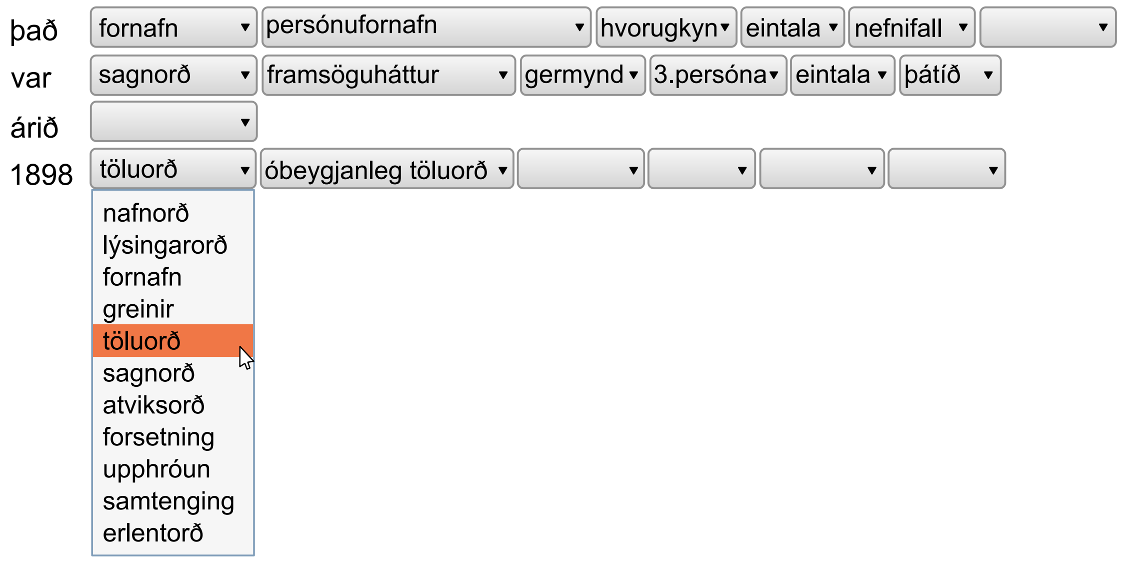
When the server is initiated it reads a configuration file for how to set up the output. After the initiation of the server the output could not be changed. In order to be able to request different outputs, without reinitiating the server, a new setting in the configuration file was added. When the server sees ALT, standing for alternating output, as the output type, it allows output to be requested as seen in the example above. The brackets and what comes between them is removed from the text before it is parsed by IceParser.

```
[txt][errorHighlight]Litla barnið borðar fisk.
```

IceParser can be requested to mark grammatical errors by adding "[errorHighlight]" in front of the input text as seen in the example above. It can also receive "[tag]", in which IceParser will not carry out the function of the transducers and return the POS tag sentences in plain text format.

4.6.2 Exercise Generator

The Exercise Generator was a suggestion on how it would be possible to utilise the platform to create drill-and-practice exercises. The idea is to display a sentence and let students select the morphological features for each word, generating a new sentence only when the student has identified the correct morphological features, or the website is refreshed. An exercise is generated from a sentence and displayed as a web page, an example is shown in Figure 4.11. The sentence is fetched from a corpus according to a certain aspect, e.g. if the student is supposed to identify the case of nouns a sentence with multiple nouns is selected in accordance with the student's abilities. Sentences are given a score to select appropriately complex exercises for students.



það	fornafn	persónufornafn	hvorugkyn	eintala	nefnifall		
var	sagnorð	framsöguháttur	germynd	3.persóna	eintala	þátíð	
árið							
1898	töluorð	óbeygjanleg töluorð					

Figure 4.11: Example of an exercise created by the Exercise Generator.

A website was created for testing this feature. When a user connects to that website, a script creates a request to the IceNLP daemon by having the client send the string "[makeExercise]" to the server. The server runs the Perl program Exercise Generator, that searches a corpus for a suitable sentence. It creates a string for every word in the sentence in brackets, as seen in the example below, where the information about each word is separated by a newline character. The text is then sent to another Perl script that uses this text to create a web page containing HTML and JavaScript as seen in Figure 4.11. The JavaScript manipulates the web page. When a student selects a word class, the appropriate drop-down lists appear to the right of that word.

```
[word=var; lemma=vera; ordflokkur=sagnorð; hattur=framsöguháttur;
  mynd=germynd; persona=3.persóna; tala=eintala; tid=þátíð]\n
```

The IMS Open Corpus Workbench (CWB) is used to search the corpus. CWB has a query processor that allows external programs to make queries (?). Exercise Generator uses the CWB in this way to fetch sentences. This feature was not implemented as a feature in IceNLP Web Service, but shows that there are possibilities to create exercises designed for every student using a platform that has access to NLP tools.

4.6.3 Remove Plain Text Brackets

The plain text and phrase per line outputs have been modified. In the output of IceParser the annotation was on both sides of the target phrase or function. The raw plain text output by IceParser for "Litla barnið borðar" ('The little child eats') is shown here:

```
{*SUBJ> [NP [AP Litla lhenvf AP] barnið nheng NP] *SUBJ>}
  [VP borðar sfg3en VP]
```

The convention is to mark the closing on right hand side only with a bracket. IceParser's transducers require opening and closing annotation that match. Therefore the internal workings of IceParser were left untouched in regard to the annotation. A function strips off the right hand side markings from the output of the transducers and leaves only the brackets. This can be seen below for the same example as above:

```
{*SUBJ> [NP [AP Litla lhenvf ] barnið nheng ] } [VP borðar sfg3en ]
```

This is also implemented for IceNLP Web Service and the IceNLP core.

5 Evaluation

This chapter deals with the evaluation of the grammatical error highlighting of Writing Support, the performance of grammatical error detection, and the user interface of Writing Support. During the first evaluation of Writing Support (section 5.1) the performance of IceNLP Web Server affected the feedback that each user received. It gave incorrect results as evident in Section 5.1.4. Hence it was important to evaluate the performance of IceNLP Web Service for this issue. In the second evaluation of Writing Support (section 5.3) the performance issue had been resolved, and new grammatical errors introduced.

5.1 Writing Support: First Evaluation

An evaluation of the current status of error detection was made in the summer of 2012. The main purpose of the evaluation was to find out if second language learners of Icelandic feel that they learn from using a grammar error highlighter, if they are open to using it, how accurately the detected errors perform, and how users react to the interface of the website.

5.1.1 Participants

The participants were 13 second language learners, mainly from English speaking or other West Germanic speaking countries. They were considered to have some comprehension of written Icelandic, but not spoken Icelandic. Some of the participants had experience of Old Icelandic. The students had been learning Icelandic, as a summer course, at Icelandic Online for a couple of months prior to the experiment.

A teacher from Icelandic Online assisted the students. All students had access to computers connected to the internet.

5.1.2 Materials

Writing Support was hosted temporarily on a remote server for this user test. The status of the website was close to that described on page 50 and as can be seen in Figure 4.4. The main differences were that only four grammatical errors were detected: wrong case of an object governed by a verb, disagreement in a noun phrase, disagreement between a subject and its complement, and disagreement between a subject and a verb. Additionally spelling errors were highlighted in a dark grey colour with white letters. For identifying grammatical errors IceNLP's daemon was utilised. Instead of the website making HTTP requests, the client part of IceNLP's server was executed with users' text piped into it.

Two forms were given to the participants. Both forms were in Icelandic and English. The first form had instructions for participants on what they were supposed to do. Participants were asked to describe two photos and then write a narrative about them which included words from a given word list. There were two photos. The first photo was of a boy sitting on the lap of a woman. They were sitting on a bench with a bicycle, palm trees, and a lake seen in the background. The second photo was of a woman and a boy washing a dog with a water hose. Along each picture a list of relevant words followed, such as: "drengur" ('boy'), "hjól" ('bike'), "stórt tré" ('large tree'), and "ungur" ('young'), both in Icelandic and English. On the form the participants were given an IP address which they were supposed to open. The IP address leads to Writing Support where they have to write at least 10-12 sentences. Instructions were given on how to use the Writing Support website with its Java Script pop-up window, detailed at the end of Section 4.4.1.

The second form had questions about the participant, the interface of Writing Support, and if the error highlighting of grammatical errors helped. Questions about the participant were with regard to Icelandic proficiency, native language, and various other questions as seen below.

The first question asks the participant if her proficiency of Icelandic is beginner, intermediate, or advanced. The next four questions ask the participant how much they agree with the statements, choosing between: agree, agree somewhat, neither agree nor disagree, disagree somewhat, and disagree.

Q1. General guidelines about how to use the system are clear.
Q2. Explanations of errors are clear.
Q3. Having some errors in my writing identified helps me write.
Q4. The system (overall) is helpful in writing Icelandic.

The next questions allowed the participants to reply in sentences.

Q5. Are there other errors you would like identified?

Q6. Did you learn anything new about Icelandic morphology as a result of using this system?

Q7. Can you think of a different way to provide automatic feedback in writing and if so what?

Q8. How can we improve this system?

Q9. How could we improve the interface or look of the system?

Participants brought their own computers to do the assignment outlined for them.

5.1.3 Procedure

The participants had fifteen minutes to prepare before undertaking the task, e.g. opening the Writing Support website on a web browser. When the task started they had half an hour to write whole sentences in Writing Support. The text was supposed to describe the two photos on the first form handed to them, and they were also supposed to make a narrative about what happened before or after the photos were taken. Then they were to let the Writing Support website analyse the text for grammatical errors. After that they had ten minutes to modify the text and fix grammatical errors before submitting it to the teacher. Finally fifteen minutes were given to fill out the questionnaire on the second form handed out to the participants.

A log with all input texts from participants was collected. The first submissions from students were analysed manually, marking all occurrences of patterns that are checked by the error detector for each of the four errors. Unmarked occurrences are categorised by if they are correctly left unhighlighted or if they should have been highlighted. In the same fashion, errors that have been highlighted are categorised if they have been correctly highlighted or not. From counting each group the number of errors in the text is the sum of correctly highlighted errors and correctly not highlighted occurrences of the pattern. Of those errors the number of undetected errors is the difference between correctly and not correctly unhighlighted occurrences of the pattern. The equation to find out recall is: $\frac{Detected\ Errors}{Errors} = Recall$.

5.1.4 Results

Texts from all the participants were contained in the logs, but only nine questionnaires were returned. The first submitted texts were gathered for analysis from each participant. From those first texts 47 spelling errors were found. The grammar error detector highlighted 10 instances of noun phrases out of 23. Noun phrase disagreement is the column marked with *N* in table 5.1. Eight of them were correctly highlighted, but in two of the cases where noun phrases were highlighted there was no error. Recall is 100% as all errors were detected.

Two instances occurred of the error, where the case of an object governed by a verb was wrong. One instance was found, and the other one was not detected out of the 38 times the pattern in which such error can occur. There were also very few errors detected where there was disagreement between a subject and its verb. There were only three errors correctly highlighted out of the seven errors that were in the text, rendering the recall only 42.9%. There were four errors incorrectly highlighted out of the 92 instances of subject-verb patterns.

There were about as many subject-complement patterns as verb-object, and its accuracy was about the same, 97.2%. That is in accordance with the overall accuracy of the four errors ranging between 95% and 100%. Eight errors were made in agreement between a subject and its complement, seven of which were detected. That makes the recall 87.5% as Table 5.1 shows.

O =	Case of an object governed by a verb is incorrect.
N =	Disagreement in a noun phrase.
C =	Disagreement between a subject and its complement.
V =	Disagreement between a subject and a verb.

	O	N	C	V
Highlighted errors	1	10	7	7
Correctly highlighted	1	8	7	3
Precision %	100.0	80.0	100.0	42.9
Errors in text	2	8	8	7
Correctly highlighted	1	8	7	3
Recall (%)	50.0	100.0	87.5	42.9
Not highlighted	38	23	36	92
Correctly not highlighted	37	23	35	88
Accuracy (%)	97.4	100.0	97.2	95.7
Undetected errors	1	0	1	4

Table 5.1: Precision and recall of error detection.

As seen in Table 5.2, participants answered that they were either of intermediate or advanced proficiency of Icelandic, with seven of them claiming to be intermediate and two advanced. Results of question 3 showed that the participants thought that having grammatical errors highlighted would help them with only one participant disagreeing with the statement. The following is a list of replies to the questionnaire.

Q0	Q1	Q2	Q3	Q4
i	1	1	1	1
i	2	2	5	5
a	5	5	5	5
i	4	5	5	3
i	4	1	5	4
i	4	3	5	4
a	4	4	5	4
i	2	2	4	4
i	4	2	5	4
Average:	3.33	2.78	4.44	3.78

Table 5.2: Answers to first five questions and averages.

The participants were frustrated with the performance of the system. The submitted texts got scrambled up with each other when sent simultaneously, so that the participants got more than their own texts back. They also complained about sluggishness of the system and that texts were incorrectly highlighted, where there were no errors in that text. The grey highlight of spelling mistakes did not show white colour but black. Writing Support website was difficult to use on tablet computers.

The way errors are pointed out was thought to suit language learners well, but that they need to be given additional hints on how to correct the errors they make. There was a suggestion to include the declension tables of a morphological database of **?** called BÍN ('Database of Modern Icelandic Inflection'). None of the participants claimed to have learned something about Icelandic morphology from using the system.

Only one participant suggested that the interface should be changed. The major changes the interface needed, according to the questionnaire, is to change error highlighting and text on the website. A nicer looking URL is suggested. The answers of the last five questions of the questionnaire can be seen below.

- Q5: Are there other errors you would like identified?

  The system does not work when there are more students simultaneous on the system. Also, it is not BYOD (Bring Your Own Device) friendly because it is difficult to use on a tablet.

  One time, after I clicked on "Yfirfara texta" ('Analyse text'), I was given *someone else's text back with their errors highlighted. My text was still in the box to edit, though.*

When several users are logged in simultaneously the correction system sometimes ceases to work proPerly.

The original text I wrote was accurately reproduced in the correction window. I then corrected the original version and pressed "Yfirfara texta" ('Analyse text'). Unfortunately the text was scrambled and also words not in my original or corrected version appeared in the text!

Some of the errors identified are not correct. When more than one user is on the system it is very slow. Occasionally you see the work of other users on the system. It is difficult to use on an iPad.

The system is slow. Improvement of the interface.

- Q6: Did you learn anything new about Icelandic morphology as a result of using this system?

  No.

- Q7: Can you think of a different way to provide automatic feedback in writing and if so what?

  No, I think this is very good. Technology just can't cope with every variation in language yet.

  No, the basic idea is good. However, further development needs to be done on the system to remove errors. Also: the boxes which are intended to show what the corrections are, are themselves confusing. They provide no useful information. They simply tell you that you have made a mistake which you already know.

  The system is an excellent idea. There are some bugs in the system which need to be sorted but overall it is a very quick and easy way to highlight grammatical areas. If it could be combined with declension tables that would be useful, or even better if you could see a declension table when clicking on a highlighted mistake that would be very useful.

- Q8: How can we improve this system?

  If the error messages could provide suggestions for correction, that might be nice.

  Make explanations of errors in English as well as Icelandic.

  It would be good if the system would be more stable. It did not always work, and was very slow sometimes.

  Better speed → a lot of people online together. Improve it visually.

  There was a problem with the server, the system seemed to be overloaded, so

I could not really try out the program. My sentences were mixed up with those of other participants. When it finally worked, I had no more time. Basically, I like the idea of this program a lot and I would certainly use it!

- Q9: How could we improve the interface or look of the system?

  People who have trouble seeing things in low contrast might find it difficult to see the black-on-grey of "Orð ekki í orðabók" ('Word not in dictionary').

  I do not think the look of the system does need improving.

  No need to improve this.

  The URL could be something simpler and logical.

  A bit of web design.

5.1.5 Conclusion

With regard to the error detection, there was a severe problem with performance when two ore more participants had submitted their texts at the same time. The way Aspell was executed made the output generation of IceParser slow. This increased the chance of participants sending texts while other participant's text was being processed. Making the IceNLP daemon able to handle concurring requests must be implemented. The spell checking did not help the participants with their first submission. The spell checking must be done before text is submitted. Perhaps spelling mistakes can be detected in the web browser of the user, either by their browser or the Writing Support website.

The over all precision was 76% and recall 76%, but the error detection of disagreement between a subject and a verb was 42.9% for both precision and recall. A subject-verb sentence such as "konan og maðurinn borða fisk" ('the woman and the man eat fish') should not be highlighted. What was needed is to detect that the subject, "konan og maðurinn" ('the woman and the man'), should be treated as a whole, being plural and neuter, and expecting the verb to be as well. There were many of those errors in the texts due to the two photos picturing a boy and a woman on both pictures.

The Writing Support website must get a new URL that represents it better than an IP address. As suggested by Icelandic Online on the design, the website should be in Icelandic and not correct errors or give the student the answers. But it needs to have some additional information about the errors for the students to be able to help themselves. Connecting Writing Support with BÍN as suggested by a participant could be accomplished, it needed to be investigated to see if it can be implemented without a total redesign of the website. BÍN is a databse of inflection that could be used to help students to correct highlighted errors.

The major issue is with the error detection which needed to be addressed. The precision and recall are considerably high. This can be caused by a small sample size due to few participants, and the fact that the students have intermediate comprehension of Icelandic. Further testing must be done with more participants for more accurate results.

5.2 Load Test

The first evaluation of Writing Support showed a problem with concurrency. The platform which facilitates Writing Support was to be a Java Servlet, not using the daemon any longer. IceNLP's Java Servlet was transformed and then tested to see how many requests it could handle per second. Would the Servlet behave like the daemon had done in the first evaluation as mentioned in Section 5.1? If so then there is a lack of the ability to handle concurrent requests which needs to be addressed. When evaluating how well IceNLP Web Service would handle Load, there were conducted two tests where IceNLP Web Server handled concurrent requests, and where IceNLP Web Server did not handle concurrent requests. That is, the main function that calls the IceNLP tokenise, tagging, and parsing functions is declared with and without synchronized keyword.

5.2.1 Materials

A script was written in JavaScript that sends POST requests to the IceNLPWebServer and writes the results. The script carried out both the requests and writing of the results on screen 1000 times in a succession. Each request contained a grammatically erroneous Icelandic text (see the example texts A and B below), and requested from IceNLPWebServer to conduct tokenisation, POS tagging, and parsing of the text. Parsing included error detection. Each request has identity code with 'CH', 'OP', or 'FF' depending on which browser it was sent from, then the number of the request counting from 0, followed by six random letters and numbers. By this each request can be tracked in the log with the third request from Chromium being identified with 'CH4' and then a random number code.

```
A:Í fyrstu myndunni sé ég litla stelpu sem er að hjóla .
  Hún er að hlæja .
  Í myndinni er líka maður .
  Hann er dökkhærður .
  Maðurinn hjalpar stelpunni að hjóla .
  (Ung stelpa á hjól , en kann ekki hjóla .
  Hún vill læra að hjóla .
  Getirðu hjálpað mér , pabbi ?

B:Hildur er ung stelpa , sex árs gömul .
  Hún er að læra að hjóla .
  Pabbi er að hjálpa henni , en pabbi segir Hildu er svo lítill .
  Pabbi kan ekki einbera sér , af því pabbi sér falleg kona .
  það finnst Hildi ekki gaman .
  Hundurinn var hvítur , en núna er hundurinn brúnn ,
```

An early version of IceNLP Web Service was tested, having only the IceNLP mode. It was hosted on the Apache Tomcat server on localhost. The website holding the scripts, from which the requests were made, was hosted on Apache HTTP Server on the same computer with another port. Three web browsers were used: Chromium version 18.0, Opera version 12.02, and Mozilla Firefox version 15.0.

5.2.2 Method

The script was set up on the website to send requests with certain parameters. The three browsers were initiated. The browsers went at the same time to the address of the website holding the script. Logs were collected.

5.2.3 Results without handling concurring requests

Opera made its 17th requests when Chromium made its first request to the server, and then Opera stopped making new requests.

```
[exiting]  45876 id=OP16XYMAOl
[entering] 45877 id=OP17hyofrP
[entering] 45879 id=CH0smqrSNM
```

The 18th request was being processed while Chromium's first request arrived in IceNLP Web Server as seen in the text above. Chromium continued making requests until Firefox made a request. Firefox's first request was received by IceNLP Web Server while it was processing the 27th request, numbered 26. The processing started working on the 28th request, but never finished. Instead Firefox's first request is finished processing and Firefox finishes all of its 1000 requests.

```
[exiting]  45905 id=CH25jTd6BJ
[entering] 45906 id=CH26ITmyLZ
```

```
[entering] 45907 id=FF0fnDuAqM
[exiting]  45906 id=CH26ITmyLZ
[entering] 45908 id=CH27F1OXXV
[exiting]  45907 id=FF0fnDuAqM
[entering] 45909 id=FF1dfcUdnM
```

Opera sent text A shown in the material section above. Chromium sent requests with the B text. IceParser received strange inputs when the first input from Chromium interfered with Opera's 18th. The first five lines are from text A as expected, but after that the start of text B takes over, with the first letters, "Hildur er u", of the first sentence of text B missing.

```
0.  Í fyrstu myndunni sé ég litla stelpu sem er að hjóla .
1.  Hún er að hlæja .
2.  Í myndinni er líka maður .
3.  Hann er dökkhærður .
4.  Maðurinn hjalpar stelpunni að hjóla .
5.  ng stelpa , sex árs gömul .
6.  Hún er að læra að hjóla .
7.  Pabbi er að hjálpa henni , en pabbi segir Hildu er svo lítill .
8.  Pabbi kan ekki einbera sér , af því pabbi sér falleg kona .
9.  það finnst Hildi ekki gaman .
10. Hundurinn var hvítur , en núna er hundurinn brúnn ,
```

5.2.4 Results while handling concurrent requests

The three browsers all finished their 1000 requests. The output for every request did not differ from what was expected for a single request with no interfering requests. As seen in the example below from the coloured log, multiple requests could be handled at the same time. The example shows when the tagger and parser are entered and exited, and also when the synchronized 'analyse' function that carries out the functions of the tokeniser, IceTagger, and IceParser.

```
enter - analyse id=FF268
exit  - tagger  id=CH407
enter - parser  id=CH407
enter - tagger  id=OP372
exit  - parser  id=CH407
enter - analyse id=CH408
exit  - tagger  id=OP372
enter - tagger  id=CH408
enter - parser  id=OP372
```

```
exit  - tagger  id=CH408
enter - parser  id=CH408
enter - tagger  id=FF268
exit  - parser  id=OP372
enter - analyse id=OP373
exit  - parser  id=CH408
enter - analyse id=CH409
exit  - tagger  id=FF268
enter - parser  id=FF268
enter - tagger  id=CH409
exit  - tagger  id=CH409
enter - parser  id=CH409
enter - tagger  id=OP373
```

The time it took all three browsers to finish 1000 requests of 64 tokens was similar when each batch of requests was made separately or simultaneously as seen in Table 5.3. The difference was 14.4 seconds more on average for the three browsers compared to when the requests were made one at a time. IceNLP Web Service analysed 3.48 requests per second on average when the browsers carried out the script separately, and 3.32 requests per second when the browsers made requests at the same time.

Time in seconds	Firefox	Opera	Chromium
One at a time	346.06	260.90	254.60
All together	365.47	277.42	261.87
Difference	19.41	16.52	7.27

Table 5.3: Precision and recall of error detection.

5.2.5 Conclusion

The load test showed that IceNLP Web Service could not handle simultaneous requests without the concurrency implementation. With that implementation IceNLP Web Service can service simultaneous requests. The time it takes to process multiple requests is increased by a fraction of a second when multiple requests are sent at the same time. This is highly unlikely to occur unless the processing time of the requests is long.

5.3 Writing Support: Second Evaluation

The study was conducted during the summer of 2013 with more participants than in the 2012 experiment. All five of the grammatical errors that were set out to be implemented were tested as had been done with a subset of them earlier. One extra error that had not been planned was to be tested as well. The same procedures were followed as have been detailed in Section 5.1.3, but with slightly different material as is explained in 5.3.2.

5.3.1 Participants

There were 26 second language learners. All had started to study Icelandic at Icelandic Online two months prior to the experiment. Two teachers from Icelandic Online assisted the students on site.

5.3.2 Materials

The same instructions were given to participants as were given in the 2012 experiment, described in Section 5.1.2, but with changed URL to the Writing Support website. The URL to the website changed to http://nlp.cs.ru.is/rithjalp. Writing Support did not use the IceNLP daemon any longer for error detection, but utilised the new IceNLP Web Service for that functionality. The daemon had provided spelling error detection which was not implemented in IceNLP Web Service. The Writing Support website was left largely untouched and therefore did not provide any spelling error detection either. There was only minor change to the questionnaire, rewriting the statements from 2012 into questions.

Q1. Are the directions for using the writing feedback system clear?
Q2. Are the explanations about the types of errors clear?
Q3. Does it help your Icelandic writing to receive suggestions about some errors but not all?
Q4. Did the system help you (in general) with your Icelandic writing?
Q5. Would you have wanted suggestions for correcting other errors in your text? If so, which ones?
Q6. Did you learn anything new about the Icelandic case system for this feedback?
Q7. Can you think of another way to give effective automatic feedback on writing?
Q8. Do you have suggestions about how we can improve the automatic feedback system?

Q9. Could we improve the "look" of the system?

IceNLP Web Service had replaced the IceNLP daemon for grammatical error detection, which was hoped to make the performance of the server stable and to handle an increased amount of participants better than the IceNLP daemon did the year before. Two new grammatical error detections were added: Incorrect case of an object governed by a preposition and the auxiliary verb error. The spelling error detection was removed. Grammatical detection had been improved, and detection of disagreement between a subject and a verb now changed gender and number if the subject contained something like 'man and woman'.

5.3.3 Results

O	=	Case of an object governed by a verb is incorrect.
N	=	Disagreement in a noun phrase.
C	=	Disagreement between a subject and its complement.
V	=	Disagreement between a subject and a verb.
P	=	Case of an object governed by a preposition.
G	=	Auxiliary verb error.

	O	N	C	V	P	G
Highlighted errors	4	44	6	28	42	0
Correctly highlighted	2	24	2	12	40	0
Precision (%)	50.0	54.5	33.3	42.9	95.2	0.0
Errors in text	18	31	6	43	86	2
Correctly highlighted	2	24	2	12	40	0
Recall (%)	11.1	77.4	33.1	27.9	46.5	0.0
Not highlighted	40	146	55	257	167	9
Correctly not highlighted	24	139	51	226	121	7
Accuracy (%)	60.0	95.2	92.7	87.9	72.5	77.8
Undetected errors	16	7	4	31	46	2

Table 5.4: Precision and recall of error detection.

There was a very low recall for errors when the case of an object governed by a verb, 11.1% as seen in Table 5.4. The recall was also very low for all other errors except for noun phrase agreement. Disagreement in noun phrases had recall of 77.4%, but suffered in precision with only 54.5%, of 44 noun phrases that were highlighted only

24 of had disagreement. From the new errors, the auxiliary verb error was never detected but two instances of the error occurred. Further more, the detection of an incorrect case of an object according to a preposition was considerably good with precision of 95.2%, even though its recall only had 40 out of 86 errors correctly highlighted.

The precision and recall varied greatly between the grammatical error types. Both precision and recall was high for the noun phrase disagreement error detection compared to the other error types. This is a result of noun phrases not being contingent on word order, nor does the order of the words within the noun phrase interfere with the error detection. Other error type detections such as the case of an object governed by a verb can suffer greatly from word order where the verb usually comes before the object. When there are more complex, unusual, or incorrect sentence structure, then the error detections that are sensitive to word order suffer. The error of wrong case of an object governed by a preposition had high precision but low recall. The high precision is caused by that when a preposition has been correctly tagged it causes few wrong detections to compare the preposition and the expected case of the following noun phrase. The low recall is caused by the prepositions incorrectly be tagged of a different word class than a preposition as explained in section 4.2.4, incorrect case of the object can cause incorrect tagging that leads to the error detection to not find the error.

Out of the 26 participants, 24 questionnaires were received. Majority of the participants claimed to be beginners. None of them thought the directions on using Writing Support were unclear. Majority of the participants thought the system helped them in writing Icelandic as seen on Table 5.5. The table also shows that on average the participants where more positive than negative in regard to explanations of the error types being clear and whether it helped them writing Icelandic to receive indication of some errors.

The participants wanted more information or suggestions on how to correct their errors, even to receive a corrected text. The error highlighting helped some participants to learn something new about Icelandic, specially the highlighting of noun phrases that are in wrong case governed by a verb. Nonetheless the participants found the system useful and thought it was a good way to get feedback from a computer on their grammar.

Many participants agreed that the website needs to be in English, specially the explanations of what each error type is. Other suggestions about the websites were to change the highlighting colour scheme to facilitate colour-blind people. The colours and design of the website should also be updated according to some participant, but some of them thought it did not need to change.

Q0	Q1	Q2	Q3	Q4
b	4	2	3	4
b	5	4	4	4
b	5	3	5	4
b	5	3	3	4
b	4	4	5	5
b	4	4	4	4
i	3	2	5	5
i	4	3	3	4
b	4	3	4	3
b	4	4	3	3
b	5	5	5	5
b	4	2	4	2
b	4	3	4	4
b	3	2	3	3
b	4	3	2	4
b	5	4	4	5
b	4	5	3	2
b	3	2	4	4
b	5	4	4	4
b	4	2	3	3
b	5	5	4	4
b	5	4	5	5
i	4	4	4	5
b	5	4	4	5
Average:	4.25	3.38	3.83	3.96

Table 5.5: Answers to first first five questions and averages.

- Q5: Would you have wanted suggestions for correcting other errors in your text? If so, which ones?

  Yes, it didn't correct cases.

  There were some grammatical errors that the program did not find.

  Maybe suggestions for other phrases.

  Wrong spelling.

  Endings ... maybe.

  The program didn't pick up when I had the wrong cases.

  Yes, right answer on click on certain button.

  Yes, all of them if possible.

  It would be helpful to get suggestions for all errors, because I don't always *understand what's wrong with my text.*

Yes, the system doesn't correct errors like "þeir" ('them' of a masculine gender) for a woman and a man ("þau") ('them' neuter).

- Q6: Did you learn anything new about the Icelandic case system for this feedback?

  Nothing new that it was a good [practice] for learning the declension of the cases.

  Something. New words, and about grammar.

  Helped in past tense writing.

  I don't know it well enough yet =)

  The case followed by certain verbs.

  The use of different cases.

  I learned new word phrases in which the accusative form of the word is required.

  Which case to use after which signal word, but have to look it up on Árnastofnun.

  I learn: "Hún er 'et' hvít skyrta".

- Q7: Can you think of another way to give effective automatic feedback on writing?

  No.

  This system works well.

  Only through teacher/student interaction. I think it is very difficult to create a program that effectively and accurately gives feedback. Especially in Icelandic.

  Don't know.

  Give examples on words that would fit better.

  This seems to be quite a good way to get feedback, but automatic feedback system does not recognize all mistakes yet. That's why it's better to get feedback from teachers instead.

  System based on Internet browsing.

  I find this system very useful.

  Maybe when there is an error of "sense" a thing like "maybe you would like to say something different" or "This makes no sense".

- Q8: Do you have suggestions about how we can improve the automatic feedback system?

  Maybe the instructions also in English.

It missed some simple things, so it did not seem to work 100%.

Show the case of a word that is written in the wrong case.

Give "Villimerkingar um" ('Error highlights' information) in English as well.

Better explanations about expressions and cases.

Give the correct form when one clicks on a return button. English translation of error message on click.

The program could offer different options?

Maybe by translating the meaning of the colours for the mistakes in English!

- Q9: Could we improve the "look" of the system?

It could be more "professional" looking. Now it looks quite simple.

Not necessary.

Something more contemporary, use the whole space of the web page.

There is too much file in front page.

Times New Roman - fonts look very home-made. Why is the box Easter-yellow? =)

Make it fill the whole page, use 3d effects, use cleaner colours.

Maybe.

Different background colour in comparison to text field, move elements to the centre of the screen. Change colours to "ósamrœmi í kyni.." ('disagreement in gender..' of noun phrases) and "ósamfrœmi frumlags" ('disagreement of subject' and a verb): I am red-green colour blind and it is difficult to tell them apart.

Brighter colours :)

Of course the look of it could be more modern, but if it's made to be too fancy the program may not work as efficiently as it is working now.

Nicer typesetting.

With more colours.

I like that different errors are marked with different colours. The program looks just a bit old. :)

Maybe but it's a system to learn language not a social network, so I think it's fine =)

5.3.4 Conclusion

The IceNLP Web Service handled the requests from users perfectly, giving none of the bad performance seen in the testing of 2012 when the service failed. But improvements to grammatical detection did not translate into better results. In 2012 the disagreement between a subject and a verb error had both precision and recall of 42.9%, but in 2013 recall dropped to 27.9% even if precision remained 42.9%. Detection of other grammatical errors declined as well with over all precision of 64.5% and recall of 43%. This was not explained by the lack of a spell checker. The big change from the result in 2012 might be that the participants in 2012 were of intermediate proficiency of Icelandic, but the group in 2013 thought of themselves as beginners. Many of the participants wrote texts with extra words, with incorrect word order, and unfamiliar sentence structure, such as "bak við eru það lítið hjólið og stór græn tré" ('behind are it little the bicycle and big green tree'). By forming the sentence in an unexpected manner IceNLP fails to correctly annotate the text. Therefore a tool which helps with sentence structure is needed for students. The tool might only accept common and simple sentence structures known in Icelandic, where the structure is gathered from a corpus.

There were errors discovered that need to be addressed with regard to detecting errors. A sentence like "þetta eru menn" ('those are men'), "þetta" is actually 'this', a singular word. Because of this the next word "eru" ('are') is incorrectly highlighted. More errors were discovered such as a disagreement in noun phrases in sentences with numbers, such as "ég er fimm ára" ('I am five years old'). "Fimm" is nominative and "ára" is genitive. This occurred few times in this year's participant's texts, but did not in last year's evaluation.

The evaluation shows that language learners are open to using a tool such as this website to facilitate their language learning. The questionnaire showed that the website needs a new design, not only to look fancy, but also for the colour blind. The website might need to be both in Icelandic and English, or made in such a way that language is not a barrier to understand the type of errors and how to correct them.

A teacher from Icelandic Online was asked her views on Writing Support. She claimed that the task of describing the pictures required more complex sentences, grammar, and vocabulary than the participants needed in previous exercises in their learning. Writing Support could become a useful tool for learning Icelandic after further improvements on the detection, not all grammatical errors were detected.

6 Discussion and Conclusion

The purpose of the project, described in this thesis, is to build an intelligent writing support system for second language learners of Icelandic. A website was built, named Writing Support, where second language learners can write sentences and have certain grammatical errors highlighted. Writing Support utilises the IceNLP module in IceNLP Web Service for lexical analysis and to mark grammatical errors. IceNLP Web Service has NLP modules that can be used to facilitate language learning. This web service receives requests from users, such as websites, to carry out analysis and to return results. Two evaluations of the error detection of Writing Support were conducted, one in 2012 and the other in 2013. The results of Writing Support's evaluation shows that precision and recall are rather low, but that participants in the evaluation agreed that this was a good way to learn Icelandic.

The reason for low precision and recall is twofold. First, spelling errors make it difficult for IceTagger to recognise words and make IceTagger guess the morphological features of words in accordance to their surrounding words. When IceTagger encounters an unknown word that might be incorrectly written, the word is tagged according to its context rather than its morphological features, as morphological features of unknown words are unknown. The tag selected to be the correct one applies to the word as it should be written correctly. IceNLP is good at tagging and parsing grammatically correct Icelandic texts. The simpler the sentence the better. IceNLP is not good at analysing erroneous and complex sentences. Second, strange sentence structure and multiple grammatical errors need to be addressed before any grammatical error detection.

Sentence structure is annotated by IceParser. Therefore only after IceParser has finished annotating phrases and syntactical functions can sentence structure error highlighting take place. This can be accomplished inside the post process phase between the *Clean 2* and *Phrase Per Line* transducers in the form of a new transducer. It is possible to analyse a tagged corpora to detect the most common sentence structures in Icelandic. With rules of most common known sentence structures it is possible to warn a user if rare or awkward sentence structure is detected. Students could be given a list of sentence structures that would guide the students. The list could be generated in accordance to the words which students use. In future releases error detection should be moved to its own transducer. That will result in longer processing time, but the error detector would have access to the whole annotated string, instead of partially annotated string.

The questionnaire showed that the Writing Support website needs a redesign. Many participants of the second evaluation wanted a spell checker. As discussed before, this is important. Many options are for spell checkers online, GNU Aspell, discussed on page 41, or Skrambi. Skrambi can easily be used as web service when it is released. Participants also wanted features that could help them fix the errors that were highlighted, one participant suggested connecting BÍN (?; see page 67) to Writing Support. This could be done for all words within highlighted noun phrases. In a similar way a list of possible cases following verbs could be shown for highlighted error between a verb and an object. A help page explaining how to correct each error type might help language learners.

Part of this project was also to create an architecture for a platform that facilitates ICALL. This platform was built using web services, and the implementation of that platform used by Writing Support was named IceNLP Web Service. As seen in Section 5.2 about load testing IceNLP Web Service, it can handle multiple requests at the same time. The platform is not language specific. Stanford POS Tagger was implemented allowing IceNLP Service to tag English, in addition to Icelandic. The Stanford module in IceNLP Web Service can only handle one model in memory at a time, due to memory restrictions, in its current implementation. When a user requests a model that is not the one that is in memory, the new model must be initiated, causing delays in processing of the request. This shows that there is a need of a monitor that directs requests, if needed, to another resource. That resource is another IceNLP Web Service that already has that other model initiated by Stanford Tagger. This can also be done if the primary IceNLP Web Service is busy but one of its resources, another IceNLP Web Service, is idle. The primary IceNLP Web Service would track status of its resources and get signals from them indicating what type of requests they are able to accept.

As mentioned in Section 4.3.3 about the filter, the functions that were implemented in IceNLP Web Service were all built-in. A POST request function was constructed but not used. But this has to be expanded further to include external programs, databases, and corpora.

In the first evaluation the precision and recall were considerably high, 76% for both precision and recall, compared to the second evaluation which had 64.5% precision and 43% recall. The participants in the first group had better comprehension of Icelandic than the group in the second evaluation, so their sentence structure was better. In the first evaluation most of the participants considered themselves to be intermediate learners of Icelandic, but the participants of the second evaluation in 2013 thought of themselves as beginners, having only studied the language for a couple of months. The number of participants was less in the first evaluation than in the second one, 13 and 26 respectively. Recall and precision of the error detection for Writing Support was low, but participants of the second evaluation thought the system helped them writing Icelandic. The system highlighted many errors, and participants liked the idea of Writing Support. Error detection worked well on the simplest sentences. This shows that there is potential for this type of system for facilitating second language learning of Icelandic.