CHALMERS

Multilingual Grammar-Based Language Training: Computational Methods and Tools

Master of Science Thesis in Software Engineering and Technology

ELNAZ ABOLAHRAR

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Division of Language Technology
Göteborg, Sweden, September 2011

Multilingual Grammar-Based Language Training: Computational Methods and Tools

ELNAZ ABOLAHRAR

Examiner: AARNE RANTA (Prof.)

Department of Computer Science and Engineering

Chalmers University of Technology

SE-412 96 Göteborg

Sweden

Telephone + 46 (0) 31 - 772 1000

Cover: Multilingual Grammar-Based Language Training: Computational Methods and Tools

Department of Computer Science and Engineering

Göteborg, Sweden September 2011

Acknowledgment

I would like to express my special thanks to Aarne Ranta my supervisor for his invaluable help and support and also Thomas Hallgren for his generous assistance with the JavaScript part as well as Krasimir Angelov and Ramona Enache for all their help. I also appreciate Olga Caprotti's helpful comments and advice on the report which highly improved it. Last but not least, I want to thank all friends and family who have helped and supported me by all means to accomplish this work.

Abstract

Living in the communication era we face an ever-increasing demand for communication. Considering the human languages role as the major means of communication, has inspired the author to do this thesis work, which we believe is an effort towards solving the challenge of human communication. The main goal of this thesis is introducing a teaching/learning aid - which we have called the *GF Translation Quiz* or in short the *Quiz* - to be used in training human languages in general and specifically targeting their grammatical aspects. The *Quiz* is designed as a web application, that makes it also available on mobile phone platforms - such as iPhone and Android. It may be applied either as an aid in a language learning methodology or as a stand alone exercise tool for self training. The *Quiz* is intended to help teachers and education supervisors by reducing the burden of creating exercises and exam questions, and at the same time help learners by providing them with instructive feedback.

About the background many efforts have been done in this mutual area between computer science and linguistics, among which we have focused on *GF (Grammatical Framework)* and its unique conceptual approach towards translation and multilinguality. Thus this thesis work relies heavily on *GF* from many aspects, i.e. most characteristics of the *Quiz* application, i.e. grammatical precision, support for multilinguality, and coverage of natural languages in small fragments are provided by *GF*. From another point of view the *Quiz* application works as an interface that makes *GF* capabilities available to ordinary non-specialist users.

List of Abbreviations

- GF : Grammatical Framework
- PGF: Portable Grammar Format
- MT: Machine Translation
- BLEU: BiLingual Evaluation Understudy
- JS : JavaScript
- CSS: Cascading Style Sheets
- HTML: Hyper Text Markup Language
- DOM: Document Object Model
- CEFR: Common European Framework of Reference

Contents

1 Introduction

As we are in the communication era today, human beings have realized the importance of communication more than ever. Nowadays being able to communicate with other nations in far or near countries is not considered a luxury anymore as it might have been a few decades ago, instead it is an essential demand which is growing at a rapid rate. In this regards human languages - as the major means of communication - have gained an unconventionally high significance over time. The actual problem with human languages is the fact that there are so many of them which are all unique regardless of probable similarities. There exist roughly 6500 spoken languages in the world today, of which 4500 have more than 1000 speakers [1]. In order to be able to communicate, we need to find some way to understand each other's languages. Many efforts have been undertaken towards solving this problem, and although great achievements have been gained, the process is still ongoing and seems to be an everlasting challenge. Among the works done in this area - that is mutual between computer science and linguistics -, we have focused on *GF (Grammatical Framework)* and its unique and conceptual approach toward translation and multilinguality. Thus *GF* is the major source of inspiration for this work and actually forms the basis that this thesis is built upon.

In brief the aim of this project is introducing a teaching/learning aid to be used in training the lexical, morphological and syntactic aspects of human languages, by discussing computational methods as well as implementing a tool for this purpose. The tool is designed as a web application, that makes it also available on different mobile phone platforms - such as iPhone and Android. From this point on we call this web application, the *GF Translation Quiz* or in short the *Quiz*. You can find more details about the application itself in Chapter 2, Solution Description.

This chapter explains the objectives of the project, state of the art, and also the background and inspiration behind this thesis work, which the author believes is somewhere in the path towards solving the challenge of human communication. Later on in Chapter 2, we explain our solution - that is the resulted application -, by describing first the applied methods and technologies, and then the application itself including implementation and technical details. In the 3rd Chapter, the outcomes of the project are evaluated and both advantages and drawbacks are discussed. In the following Chapter, some future work possibilities are suggested. Afterwards we have concluded the whole thesis work in the final Chapter, which is followed by three Appendices: A, B and C. Appendix A explains *Quiz'* features and functionalities and how they work

with an intention for end users, while Appendix B is mostly aimed at teachers or education supervisors who would like to use this application as a means of training and/or evaluating their students language skills. Finally in Appendix C some information is provided for users at a higher level; e.g. programmers/developers who might want to apply the *Quiz*'s code within their own applications.

1.1 Objectives

This section focuses on the main objectives of this project; each objective describes one advantageous aspect of the project.

1.1.1 Multilinguality

One important goal in this project has been introducing a multilingual application, and by multilingual we mean that the system is available in many languages that the user can choose from, and it does not necessarily mean that all languages are applied at the same time [2]. The generated exercises are in the preliminary form of sentences in a certain language which need to be translated to another one. In other words the application is intended to be general in terms of languages and give the users many language choices. For this purpose *GF*'s Resource Grammar Library have been applied as the constructive base of the project. Furthermore the possibility to extend the system by adding new languages is highly intended, which is a major achievement in *GF*.

1.1.2 Grammatical Precision and Language Coverage

The question to be answered here is that how much grammatical precision we actually need in the *Quiz* application and how big chunk of each language should it cover at the same time. "According to an old wisdom in natural language processing coverage and precision cannot be maximized at the same time, thus there is a trade off between coverage and precision" [2]. Therefore if we put aside a hard coded application which cannot go wrong but has not much to offer as well, there remains a choice between the so called data-driven and grammar-driven translation systems that the *Quiz* application could rely on. Where data-driven methods (e.g. Google translator [3]) mostly based on statistical methods provide a much larger coverage of the language - compared to grammar-driven ones-, but of course with no warranty for grammatical accuracy [2]. However in our case making the decision is not difficult at all, for

the purpose of the application - which is teaching languages - enforces grammatical precision as an inevitable priority. Evidently the error tolerance of the system should be absolute zero. Who could imagine a reliable learning tool that generates wrong output, which is totally self-contradicting!

Moreover it is more effective to expose learners to a new language in a gradual manner rather than throwing them into the entire language all at once. The same holds for the *Quiz* application, therefore we do not need large language coverage. Conversely, we would rather have languages in small fragments. That is what we can get from the *Controlled Languages* concept available in *GF*. See more about this in Subsection 2.1.1.

1.1.3 Stand Alone Exercise Tool

Although the *Quiz* application can be applied as a very helpful aid inside a language learning methodology, it is initially intended to be a stand alone exercise tool for self training. Therefore providing learners with helpful guidance and instructive feedback is a high priority. This purpose is covered by a combination of features and functionalities in the application, namely *Check Answer*, *Hint* and *History*. Specially the *Hint,* which analyzes the input answers and gives accurate feedback on wrong answers. This makes the application easy to use by the learners without the aid of a teacher or other help resources. A simplified version of BLEU [4], which is an automatic machine translation evaluation method, combined with the Mastermind's game strategy has been applied to implement this feature. See more details about the *Quiz* functionalities in Section 2.2.3.

1.2 State of the art

In this section we discuss briefly some existing computerized language training and exercise generating applications, as listed below:

1.2.1 Moodle

- What is Moodle

- Moodle (abbreviation for Modular Object-Oriented Dynamic Learning Environment) is a Course Management System (CMS), also known as a Learning Management System (LMS) or a Virtual

Learning Environment (VLE). It is a free and open-source e-learning web application that educators can use to create effective on-line learning sites. Its distribution is via standard core packages, which the users need to install on their own web servers which can be nearly any platform (currently available on Mac, Windows and Linux) where PHP and a database are already installed [5].

- Advantages

- Moodle is a general framework for education and is highly customizable, so it can meet diverse demands of a vast range of users. It is open source and therefore available to all.
- It is more than a mere application; it is a growing community of users, developers and literally anyone who is interested which adds great potential for support and further development.

- Drawbacks

- In order to create a quiz in Moodle the user should first create the questions, however he/she can also save them in a question bank for later use. Therefore in Moodle the burden of designing and creating quiz questions remains unresolved.
- The question banks are reusable of course but still there stand two major difficulties: Firstly, to get a well designed quiz the user still needs to check the drawn question manually to make sure that they meet his/her requirements. Secondly as this method is rather hard coded, the diversity of the questions is limited to the number of existing questions in the bank which is a static number.
- The interesting point is that Moodle's major advantage which is being general is at the same time a drawback, because it cannot cover specific needs of all subjects at the same time.

1.2.2 EngOnline

- What is EngOnline [6]

- Provided by the Center of Language and Communication at department of Applied IT, which is mutual between Chalmers University of Technology and Gothenburg University, EngOnline is an on-line

service that provides the students with an on-line grammar book and also lots of exercises and quizzes exclusively designed for the English language learners.

- Advantages

- The teaching material and the assessing procedures are smoothly combined.
- Quizzes and exercise sets are generated automatically by drawing questions randomly from a question data base. Thus various combinations are readily available for different purposes like homework, exams and so on.

- Drawbacks

- As stated above exercises are drawing from a question data base. The same problems with Moodle about limited diversity and need for manual supervision hold here as well.
- It is only targeting one language, namely English.

1.2.3 Rivstart

- What is Rivstart

- Rivstart is the name of a method for teaching Swedish language published by Natur & Kultur. Its main materials consist of a text book (Textbok) and a CD containing mp3 files, accompanied by an exercise book (Övningsbok). Rivstart comes in four levels: A1, A2, B1, B2 according the CEFR standard levels [7]. The part that we are most interested in about this method is its website which provides lots of on-line exercises for each chapter in the book. The exercises are of various types like multiple choice questions, phrase matching, blank filling and etc. Again the exercises we are most interested in are the blank filling ones which have a very precise feedback system which catches the user's errors up to letter's scale. Below are some sample screen-shots of these exercises and the system's feedback. For more information about Rivstart and its on-line exercises see [8].

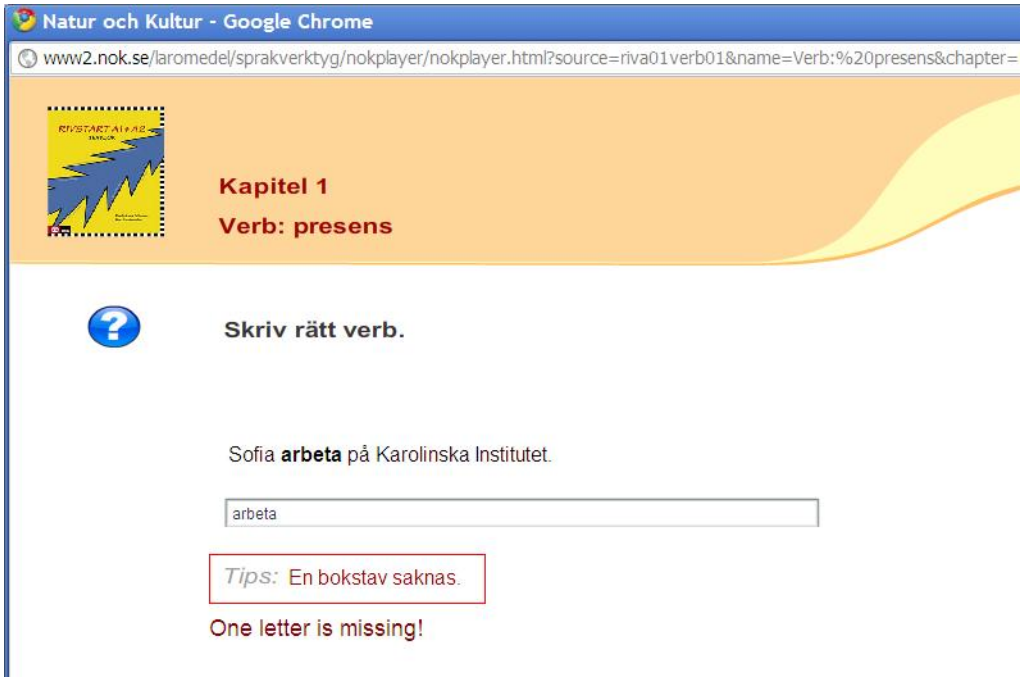

Figure 1: Rivstart samples: missing letter

Figure 2: Rivstart samples: wrong letter

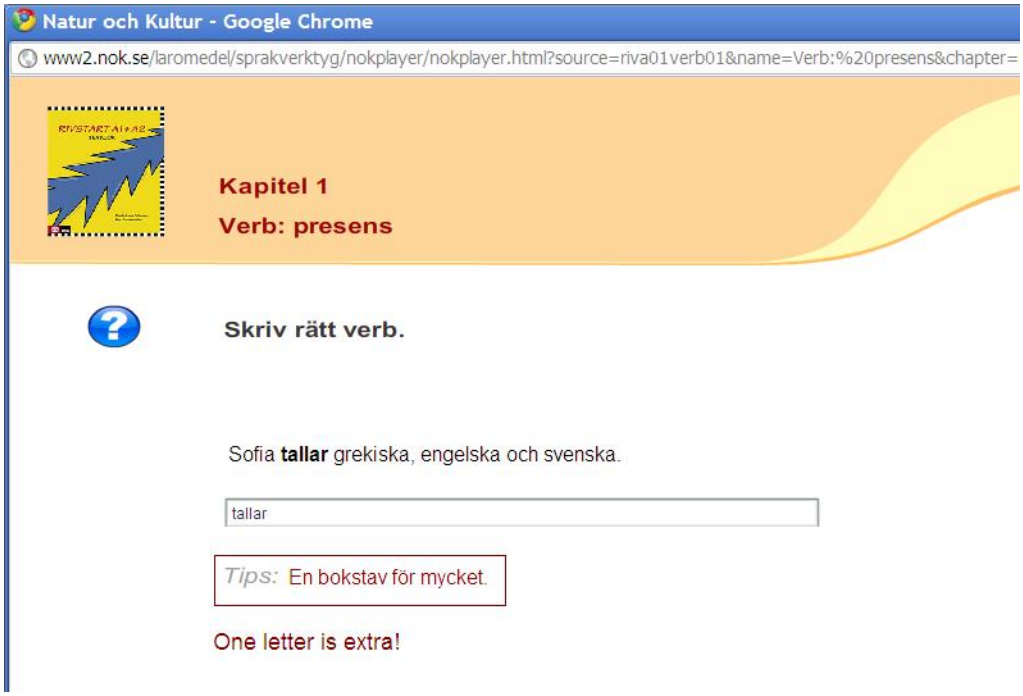

Figure 3: Rivstart samples: extra letter

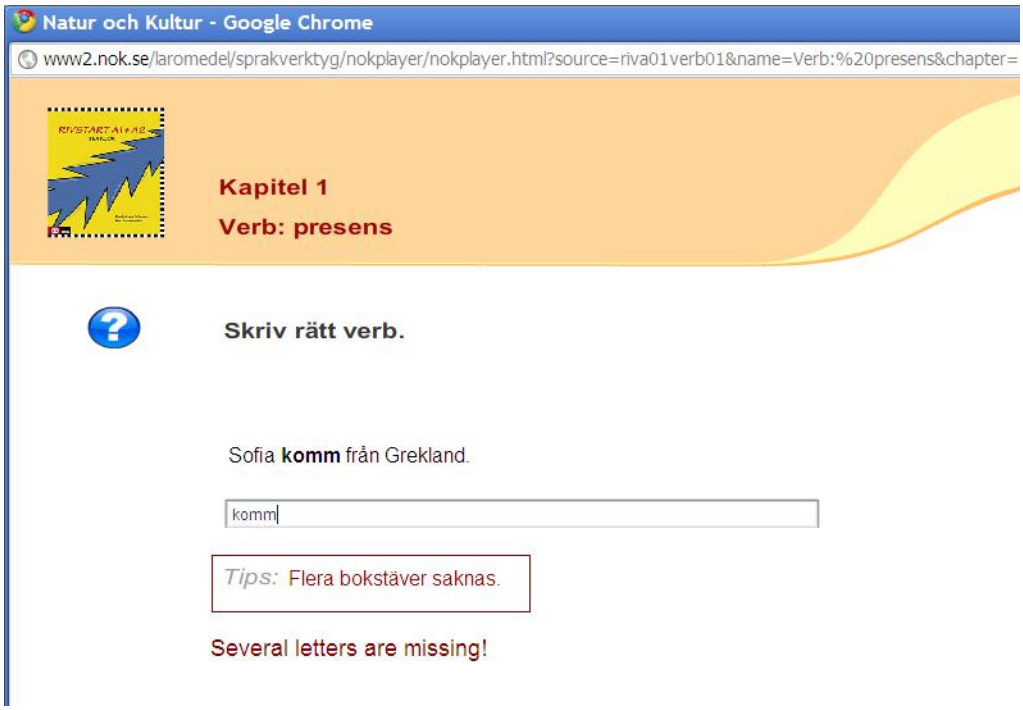

Figure 4: Rivstart samples: several missing letters

- Advantages

- The exercises are designed specifically to support a method and are therefore very coherent with the rest of teaching material and thus benefit a lot from their support at the same time, for example learners can always refer to the book for examples and explanations.
- Feedback is very precise, namely up to the letter's level, and therefore very useful for the learners. This also provides the possibility to train independently.

- Drawbacks

- Having static exercises, the same problems with Moodle and EngOnline about limited diversity and need for more manual work hold here as well.
- It is only targeting one language, namely Swedish.

1.3 Background and Inspiration

This subsection is about the reasons and motivations behind this work. First we discuss the essence of language learning, then we continue with describing the desired role of computers in the language learning process. This leads us to introduce *GF* and its capabilities which is our major source of inspiration and actually the *Quiz* is built upon its strength. We will discuss the *GF* system itself in detail in Section 2.1.1.

1.3.1 Why learning a foreign language matters at all

As we mentioned earlier people in different parts of the world speak different languages, thus in order to be able to communicate we need to find some way to understand each other. The simplest yet most effective way is to get to know the other languages. Of course it is not possible for everyone to learn all those foreign languages out there but it is very common that some people will need to learn one or more foreign languages in their life time.

On the other hand, languages as an advanced complex means of communication are known to be an exclusive ability which has made the intellectual human being, distinctive from other creatures on earth. Therefore the ability of applying languages for communication is known as one of

the mind's most important properties used to describe humans intelligence. Thus learning languages is not only looked upon as a requirement but also as a brain exercise for intelligence training, and as most intelligence related topics, its fun factor for many should not be neglected as well.

1.3.2 How can computers help us in the language learning process in general and GF's specific role

Learning a language is not possible without training and exercising even for the most talented people. On the other hand as professionals in language education usually state creating exercises and exam questions is one of the most difficult, dull and time consuming tasks in the whole educating procedure. It is difficult because it has to be correct and precise grammatically and new and creative at the same time, and the main trouble is that this difficult process has to be repeated many times over and over as time passes by. There are two specific reasons for this need: being up to date and also creative rather than sticking to old clichés which may end up in a mere memorizing activity.

As computers were created to take the burden of exactly the same type of jobs - dull, full of repetitions and demanding high accuracy at the same time- off human shoulders, it seems that it is worth the effort to computerize the exercise generation process.

To achieve this goal, we have focused on *GF* and its unique conceptual approach toward translation and multilinguality. Here we only mention those *GF's* characteristics that the *Quiz* application has benefited from. We will discuss the *GF* system itself in detail in Section 2.1.1.

We can summarize a list of linguistic requirements of the *GF Translation Quiz* as follows:

- High grammatical precision as the first priority
- Support for multilinguality
- Coverage of natural languages in small fragments

GF addresses all these requirements: First of all it provides a high-level grammar formalism which allows writing multilingual domain specific application grammars (also called semantic grammars) at minimum expense. Moreover having a resource grammar library along with the *Controlled Languages* concept defined in *GF,* makes it even easier to write

application grammars which cover similar fragments in several natural languages at the same time. The so called *Controlled Languages* are fragments of natural languages, designed to be clear and unambiguous, so that they become mechanically processable. Thus they get the benefits of formal languages, and since they are actually fragments of natural languages, they perfectly fit the demands of natural language training applications such as our *Translation Quiz*. Thus *GF* saves us a considerable amount of time and effort required to add new languages to and/or extend the existing grammars [9].

A detailed example on how grammars actually work in *GF* is discussed in Sections 2.1.1 and 2.1.2.

1.3.3 `Translation_Quiz` command in GF shell

This thesis project was initiated by an attempt to improve the *GF* Multilingual Grammar Tour which is a multilingual web document designed for teaching the grammar of supported languages in *GF* by A. Ranta. There are a lot of examples and exercises in this document which rely on the *GF* shell for execution. Among these exercises there is a shell command called **`translation_quiz`**, that was a starting point for the *Quiz* application. We borrowed the idea and developed it into our current web-based *Quiz* application with its many features which will be discussed in detail in Section 2.2. Here I briefly describe how this command works in the *GF* shell:

The command format is as shown in the following example, where the `from` option determines the source language we want to translate from and the `to` option sets the target language. Finally the `cat` option determines the starting category for the random tree generation action, for example `A` for adjective, `V` for verb and `N` for noun.

```
> translation_quiz -from=DemoEng -to=DemoSwe -cat=N
```

Here is an screen-shot of the **`translation_quiz`** command in the *GF* shell.

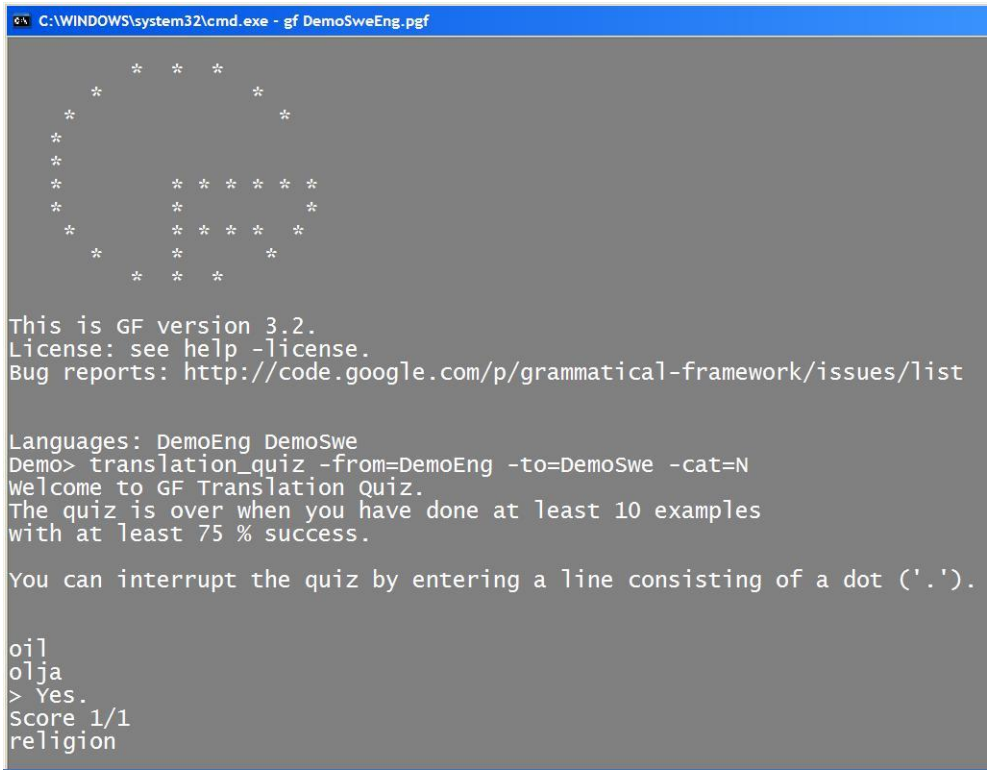

Figure 5: **`translation_quiz`** in *GF* shell

However before the users can use this command, they should perform some initialization tasks:

First of all they should have *GF* installed on their system to be able to run *GF* at the first place. For practical installation instructions, please refer to the *GF* official website at :

http://www.grammaticalframework.org/

Then they have to get - e.g. download - the required *GF* grammar files (with **`.gf`** suffix) and make the appropriate *PGF* file which contains all the **`from`** and **`to`** languages' related *GF* grammar files. *PGF (Portable Grammar Format)* files , are the portable format for grammars written in *GF*. The following command is required for this purpose [2]:

```
> gf -make --name=<name of pgf file> <concrete grammars .gf>
```

In the above command 1 to n grammars may be applied at the same time. The only condition is that all these concrete grammars should share the same abstract syntax grammar. The resulted *PGF* file is a binary file

which contains a pre-compiled and optimized multilingual grammar as defined in the applied concrete grammars [2]. Note that you should be in the directory that contains the grammar files. For example here we have the demo grammar for Swedish and English located in `gf\lib\src\demo`, and we have named the *PGF* file as `DemoSweEng`:

```
> gf -make --name=DemoSweEng DemoSwe.gf DemoEng.gf
```

And then they can start *GF* by calling the name of the *PGF* file to import the grammars, like this in our example:

```
> gf DemoSweEng.pgf
```

As one can see, it is not a trivial task to work with *GF* directly from a system's shell, specially for those who are not much familiar with computing, let alone *GF*. The greatest achievement by the web-based *Quiz* application is bringing *GF* capabilities to users with no prior acquaintance with computers or linguistic background. Accordingly, one of our goals has been to design the application in a way that even school children and retired people find it easy to use. Now the only knowledge required to use the *Quiz* application is how to open a browser and of course a link to the *Quiz*'s web page.

Note: The *GF* shell command (**`translation_quiz )`**, that we explained in this section, has only been a source of inspiration to our current web-based *Quiz* application - which is a part of this thesis work -, and has not been used in the application development by any means.

2 Solution Description

The implemented tool is a teaching/learning aid aimed at training the lexical, morphological and syntactic aspects of human languages. The tool is designed as a web application- that we have named the *GF Translation Quiz* or in short the *Quiz* . This design decision makes the *Quiz* also available on mobile phone platforms. Basically the *Quiz* application - as it is evident from its name - generates quiz questions by randomly constructing phrases from the input *GF* grammar. The generated questions are in the preliminary form of phrases in a certain language which need to be translated to another one. The *Quiz* is intended to support grammatical precision and multilinguality. In order to meet these requirements, it relies on *GF* and its *Resource Grammar Library*, which also makes it very easy and systematic to extend by adding new languages to and/or extending the existing grammars.

The *Quiz* application may be applied either as an aid in a language learning methodology or as a stand alone exercise tool for self training. In both cases and specially in the later one providing the user with helpful feedback is an important functionality of the application. This is handled by the *Hint* feature of the *Quiz,* which analyzes the user input and gives instructive feedback on wrong answers. The application is generally planned to be easy to use for a wide range of users without the help of a teacher or other help resources required. In this chapter we will discuss the methods and technology applied, the *Quiz* application details, as well as known application's limitations.

2.1 Methods and Technology

2.1.1 GF

GF (Grammatical Framework) has more than one description based on different angles from which we look at it: It can be described as "a special-purpose functional language for defining grammars" [10], and also as "a grammar formalism which is based on constructive type theory to express the semantics of natural languages for multilingual grammars and their applications" [10]. In both senses, *GF* uses a Logical Framework (LF) [11] for a description of an abstract syntax module accompanied by any number of concrete syntax modules. The abstract syntax is a tree-like representation that captures the semantically relevant structure of the language fragment, while the concrete syntaxes map the tree structure with linear text representations of target languages. Although *GF*

grammars themselves are purely declarative, they can be used both for linearizing abstract syntax trees and for parsing strings. Additionally both formal and natural languages can be described in *GF*. The key notion of this description is a grammatical object, which in fact is a record that contains all information on inflection and inherent grammatical features such as number and gender in natural languages, or precedence in the case of formal languages [10, 2].

Here we demonstrate how these *GF* concepts are applied by quoting an example grammar from the book *"Grammatical Framework: Programming with Multilingual Grammars"* by A. Ranta [2]. This is a rather simple grammar named the Foods grammar which describes short comments on food. Here are some examples:

- This cheese is very expensive.

- Those fish are fresh .

- That wine is very very delicious.

As stated before, *GF* grammars consist of two module types: **abstract syntax** and **concrete syntax**. Abstract syntax module contains the syntactic functions which are applied for building abstract syntax trees in that grammar, while concrete syntax modules explain how the abstract syntax trees are linearized to actual phrases (strings). First let's take a look at the abstract syntax module of the Foods grammar [2]:

```
abstract Foods = {
flags startcat = Comment ;

    cat

       Comment ; Item ; Kind ; Quality ;
    fun

       Pred : Item -> Quality -> Comment ;
       This, That, These, Those : Kind -> Item ;
       Mod : Quality -> Kind -> Kind ;
       Wine, Cheese, Fish, Pizza : Kind ;
       Very : Quality -> Quality ;
       Fresh, Warm, Italian,
        Expensive, Delicious, Boring : Quality ;
    }
```

Figure 6: The abstract syntax module of Foods grammar [2]

The first line is the module header which contains the module type (abstract) as well as the module name (Foods). The module body consists of different judgment forms as we explain below [2]:

- flags: **flag definitions**, e.g. the one in our example states that `Comment` is the start category by default.
- cat: **category declarations**, telling what categories (types of abstract trees) exist in the grammar.
- fun: **function declarations**, telling what tree building functions there are in the grammar.

Now we discuss the concrete syntax modules of Foods grammar for English and Italian languages. The English concrete syntax is given on page 17. Again the header indicates the module type which is a concrete syntax named `FoodsEng` for the abstract syntax of Foods grammar. The judgments applied here are [2]:

- lincat: **linearization type definitions**, indicating what are the type of produced objects resulted from linearizing trees of each category . (Note: Linearization means the procedure of converting trees to strings.)
- lin: **linearization** rules, telling how trees are linearized .

In order to understand the the concrete module we need to explain some *GF* concepts first [2]:

- **Parameters:** By means of the `param` judgment in *GF* we can define new types, exactly the same as is done in functional languages like *Haskell*. Here is an example which defines the type `Number` which has two values, Singular and Plural:

  ```
  param Number = Sg | Pl
  ```
  Another example is the `Gender` type which is either Feminine or Masculine:
  ```
  param Gender = Masc | Fem
  ```

- **Tables:** are applied in *GF* to formalize inflection tables, where each value of a certain parameter type is assigned a string (the corresponding inflection form). Below is an example of a table type and its definition in *GF*:

```
Number => String
table { Sg => "mouse" ; Pl => "mice" }
```

– **Records and record types:** The same as their general meaning in computer science, records in *GF* are data structures that gather objects of different types together. Here are two examples in *GF*. The first record contains a string with label s and a **Number** with label n, and the second one contains a table from **Number**s to strings labeled s and also a **Gender** with label g:

```
{s= "mouse" ; n = Sg }
{s= table { Sg => "mouse" ; Pl => "mice" } ; g = Fem }
```

– **Selection Operator (!) :** is applied for accessing a value in a table, for example,

```
table { Sg => "mouse" ; Pl => "mice" } ! Sg
```
will give the string **"mouse"** as its result.

– **Projection Operator (.) :** is used to get access to a value inside a record under given label. For example, the following projection will result in the number **Sg** :

```
{s= "mouse" ; n = Sg } . n
```

– **Operation definition:** In order to avoid copy and paste programming which is a significant rule in the functional programming paradigm, functions are introduced under **oper** judgments. Note: This functions are called operations to avoid confusion with **fun** functions in abstract syntax modules. The following example is an operation defining the regular inflection of nouns in English:

```
oper regNoun : Str -> {s: Number => Str } =
  \word -> {s = table {Sg => word ; Pl => word + "s" }};
```
Which is very similar to function definition in any functional programming language.

– **Appending operator (++) vs. Gluing operator (+) :** The (++) operator combines two lists of tokens into one list, while the (+) operator combines two tokens into one token, for example:

```
"foo" + "bar" = "foobar"
```
but
```
"foo" ++ "bar" = "foo bar"
```

```
concrete FoodsEng of Foods = {
lincat
    Comment, Quality = {s : Str} ;
    Kind = {s : Number => Str} ;
    Item = {s : Str ; n : Number} ;
lin
    Pred item quality =
          {s = item.s ++ copula ! item.n ++ quality.s} ;
    This  = det Sg "this" ;
    That  = det Sg "that" ;
    These = det Pl "these" ;
    Those = det Pl "those" ;
    Mod quality kind =
          {s = \\n => quality.s ++ kind.s ! n} ;
    Wine = regNoun "wine" ;
    Cheese = regNoun "cheese" ;
    Fish = noun "fish" "fish" ;
    Pizza = regNoun "pizza" ;
    Very a = {s = "very" ++ a.s} ;
    Fresh = adj "fresh" ;
    Warm = adj "warm" ;
    Italian = adj "Italian" ;
    Expensive = adj "expensive" ;
    Delicious = adj "delicious" ;
    Boring = adj "boring" ;
param
    Number = Sg | Pl ;
oper
    det : Number -> Str ->
           {s : Number => Str} -> {s : Str ; n : Number} =
               \n,det,noun -> {s = det ++ noun.s ! n ; n = n} ;
    noun : Str -> Str -> {s : Number => Str} =
           \man,men -> {s = table {Sg => man ; Pl => men}} ;
    regNoun : Str -> {s : Number => Str} =
           \car -> noun car (car + "s") ;
    adj : Str -> {s : Str} =
           \cold -> {s = cold} ;
    copula : Number => Str =
           table {Sg => "is" ; Pl => "are"} ;
    }
```

Next to discuss is the Italian concrete module of the Foods grammar, that is shown below. The Italian version is different from the English one in a few aspects. First of all you might ask where did all **param** types and **oper** definitions go! To explain this we should note that these parts are completely independent of any abstract syntax and actually they may be applied in many concrete syntaxes, for different abstract syntaxes and some times even for different languages. Therefore to increase reusability, *GF* provides a module type called **resource**, which can contain **oper** and **param** judgments. Then it is enough that the the **concrete** module opens the **resource** module to be able to use its definitions [2].

```
concrete FoodsIta of Foods = open ResIta in {
lincat

    Comment = {s : Str} ;
    Kind = {s : Number => Str ; g: Gender} ;
    Item = {s : Str ; g: Gender ; n : Number } ;
    Quality = {s: Gender => Number => Str } ;

lin

    Pred item quality =
          {s = item.s ++ copula ! item.n ++
               quality.s ! item.g ! item.n} ;
    This  = det Sg "questo" "questa" ;
    That  = det Sg "quel"   "quella" ;
    These = det Pl "questi" "queste" ;
    Those = det Pl "quei"   "quelle" ;
    Mod quality kind =
          {s = table {n => kind.s ! n ++ quality.s ! kind.g ! n ;
              g = kind.g } } ;
    Wine = noun "vino" "vini" Masc ;
    Cheese = noun "formaggio" "formaggi" Masc ;
    Fish = noun "pesce" "pesci" Masc ;
    Pizza = noun "pizza" "pizze" Fem ;
    Very a = {s = table {g,n => "molto" ++ qual.s ! g ! n}} ;
    Fresh = adjective "fresco" "fresca" "freschi" "fresche" ;
    Warm = regAdj "caldo" ;
    Italian = regAdj "italiano"" ;
    Expensive = regAdj "caro" ;
    Delicious = regAdj "delizioso" ;
    Boring = regAdj "noioso" ;
    }
```

Here is the resource module called **ResIta** that is used by the **FoodsIta** concrete syntax [2].

```
resource ResIta = open Prelude in {
param

    Number = Sg | Pl ;
    Gender = Masc | Fem ;

oper
   NounPhrase : Type =
      {s : Str ; g : Gender ; n : Number} ;
   Noun : Type = {s : Number => Str ; g : Gender} ;
   Adjective : Type = {s : Gender => Number => Str} ;


    det : Number -> Str -> Str -> Noun -> NounPhrase =

      \n,masculine,feminine,commonNoun -> {


    s = table {Masc => masculine ; Fem => feminine} ! commonNoun.g
         ++ commonNoun.s ! n ;

       g = commonNoun.g ;
       n = n
      } ;

    noun : Str -> Str -> Gender -> Noun =
           \vino,vini,g -> {
            s = table {Sg => vino ; Pl => vini} ;
            g = g
          } ;
    adjective : (nero,nera,neri,nere : Str) -> Adjective =
          \nero,nera,neri,nere -> {
            s = table {
               Masc => table {Sg => nero ; Pl => neri} ;
               Fem => table {Sg => nera ; Pl => nere}
              }
            } ;
    regAdj : Str -> Adjective = \nero ->
          let ner : Str = init nero
           in
          adjective nero (ner+"a") (ner+"i") (ner+"e") ;
    copula : Number => Str =
          table {Sg => "è" ; Pl => "sono"} ;
    }
```

As you might have noticed another difference from the English version is the introduction of a new form in the type of **`adjective`** operation, which is called **function types with variables**. The type

```
(nero,nera,neri,nere : Str) -> Adjective
```

is the same as

```
Str -> Str -> Str -> Str -> Adjective
```

just that the use of variables make it possible to share the argument type. In this case we could have used wild-cards (-) as well because the the variables make no semantic difference. However using mnemonic variables help the user of the function to give the proper forms of arguments [2]. See other examples of this form in Section 2.1.2, under inflection paradigms needed for the Foods grammar on page 24.

The last difference is that in **`ResIta`** we have used **type synonyms** for the linearization type of noun phrases, nouns and adjectives. This is all for the purpose of cleaner and more structured code as well as enhanced data abstraction [2].

In the following subsection we describe the *GF Resource Grammar Library* and how it can be used to write application grammars. Then we rewrite the same Foods example shown in this section, this time by means of the *Resource Grammar Library*.

2.1.2 GF Resource Grammar Library

Writing multilingual domain specific application grammars (also called semantic grammars) in *GF* - as we saw an example of in the previous section -, becomes even simpler by applying the *GF Resource Grammar Library*. Such grammars cover similar semantic fragments in several natural languages, and therefore are extremely redundant. Thus applying the *Resource Grammar Library* also saves us a considerable amount of time and effort [9].

The great benefit we get from *GF Resource Grammar Library* is that it makes writing multilingual domain specific grammars - also called application grammars in *GF* -

The *GF Resource Grammar Library* is a set of natural language grammars implemented in *GF*. These grammars share a common abstract

syntax, that is in the format of a tree structure. Each language is then obtained via compositional mappings from abstract syntax trees to a concrete syntax specifically written for that language. The grammar defines, for each language, a complete set of morphological paradigms and a syntax fragment. The *GF Resource Grammar Library* plays the role of the standard software library much similar to the Standard Template Library of C++ or the Java API [12].

The library is available as open-source software under the GNU LGPL License. Currently it covers 25 languages including but not limited to: Bulgarian, Catalan, Danish, Dutch, English, Finnish, French, German, Italian, Nepali, Norwegian (bokmål), Persian, Polish, Punjabi, Romanian, Russian, Spanish, Swedish, and Urdu while more languages are under construction [12]. For an up-to-date list of available languages, please refer to the *GF* official website at:

http://www.grammaticalframework.org/

The *Resource Grammar Library* has manifold language processing applications such as translation, multilingual generation, software localization, natural language interfaces, and spoken dialogue systems. In this thesis work, we have applied it as the resource for getting grammatically precise translations between the *From* (source) and *To* (target) languages in the *Quiz* application.

Here we will rewrite our example, the Foods grammar from the previous section, by applying the *GF Resource Grammar Library*. Just as a reminder, the whole examples and explanations in Sections 2.1.1 and 2.1.2 are quoted or summarized from the "*Grammatical Framework: Programming with Multilingual Grammars*" by A. Ranta [2].

The abstract syntax module remains unchanged, and is the same as in Figure 6. Before going to the Foods grammar example we need to give some preparatory explanations here:

In the *Resource Grammar Library* the goal is to achieve grammatically correct combinations of words, regardless of their meaning. This makes it possible to cover a much larger subset of languages compared to semantic grammars. Based on this approach, a *Resource Grammar* has two kinds of categories and two corresponding type of rules [2]:

- lexical:

  – lexical categories to classify words

  – lexical rules to define words and their properties

- phrasal (also known as combinational or syntactic):

  – phrasal categories to classify phrases of arbitrary size

  – phrasal rules to combine phrases into larger phrases

In an abstract syntax the lexical rules are **fun** functions that take no arguments while phrasal rules are functions that do take arguments [2].

There is also another classification inside the lexical categories and that is **open** vs. **closed** categories. In general closed categories contain structural words (also known as function words). Here are examples of the closed category words [2]:

```
Det (stands for determiner, e.g. this, that)
AdA (stands for ad-adjective, e.g. very)
```

And below are some examples of the open category words [2]:

```
N (stands for noun, e.g. wine, pizza)
A (stands for adjective, e.g. delicious, warm)
```

The major benefit we get from defining such classification is reusability for the words in closed categories can be listed once and for ever in a library. For the Foods grammar case we can use the following structural words from the **SyntaxEng** module existing in the library [2]:

```
this_Det, that_Det, these_Det, those_Det : Det ;
very_AdA : AdA ;
```

For the linearizations of the open words used in the Foods grammar we can apply the morphological paradigm library called **ParadigmsEng** , for example [2]:

```
lin Wine =  mkN "wine" ;
```

This will make a noun out of the string “wine”.

The *Resource Grammar* API is divided into language-specific and language-independent parts [2]:

– The syntax API is language-independent, i.e. has the same types and functions for all languages. It is called **`SyntaxL`** for each language **`L`**.

– The morphology API is language-specific, i.e. has partly different types and functions for different languages. It is called **`ParadigmsL`** for each language **`L`**.

In Figure 7 we see a table of all phrasal, categories and rules and also the structural words we require for the Foods grammar example, which is a small fragment of the whole *Resource* API [2].

Category	Explanation	Example
`Cl`	clause (sentence), with all tenses	*she looks at this*
`AP`	adjectival phrase	*very warm*
`CN`	common noun (without determiner)	*red house*
`NP`	noun phrase (subject or object)	*the red house*
`AdA`	adjective-modifying adverb,	*very*
`Det`	determiner	*this*
`A`	one-place adjective	*warm*
`N`	common noun	*house*

Function	Type	Example
`mkCl`	`NP -> AP -> Cl`	*John is very old*
`mkNP`	`Det -> CN -> NP`	*this old man*
`mkCN`	`N -> CN`	*house*
`mkCN`	`AP -> CN -> CN`	*very big blue house*
`mkAP`	`A -> AP`	*old*
`mkAP`	`AdA -> AP -> AP`	*very very old*

Function	Type	In English
`this_Det`	`Det`	*this*
`that_Det`	`Det`	*that*
`these_Det`	`Det`	*this*
`those_Det`	`Det`	*that*
`very_AdA`	`AdA`	*very*

Figure 7: *Resource Grammar* API presentations for phrasal, categories and rules as well as the structural words required for the Foods grammar [2]

Moreover we need some inflection paradigms for each language. The paradigms needed for English and Italian are shown in Figure 8 [2].

English:

Function	Type
mkN	(dog : Str) -> N
mkN	(man,men : Str) -> N
mkA	(cold : Str) -> A

Italian:

Function	Type
mkN	(vino : Str) -> N
mkA	(caro : Str) -> A

Figure 8: Inflection paradigms needed for the Foods grammar [2]

As mentioned earlier **mkN** makes a noun out of an input string, and **mkA** makes an adjective from its input string. In the above paradigms, we also see application of **function types with variables**.For example in **(dog : Str) -> N**, the word "dog" is just a representative of the input type - which should be a noun here - and could be any other noun. As another example, in **(man,men : Str) -> N**, the words "man" and "men" are used just to say that the function takes two input strings which are nouns and the second one is the irregular plural form of the first one. This form was explained in more formal terms, in Section 2.1.1, under the **resource** module for Italian called **ResIta** on page 20.

Now we are ready to write the concrete syntax modules. We begin with English. The concrete syntax opens **SyntaxEng** and **ParadigmsEng** in order to get access to the required resource libraries [2]:

```
concrete FoodsEng of Foods =
     open SyntaxEng,ParadigmsEng in {
lincat
    Comment = Cl ;
    Item = NP ;
    Kind = CN ;
    Quality = AP ;
lin
    Pred item quality = mkCl item quality ;
    This kind = mkNP this_Det kind ;
    That kind = mkNP that_Det kind ;
    These kind = mkNP this_Det plNum kind ;
    Those kind = mkNP that_Det plNum kind ;
    Mod quality kind = mkCN quality kind ;
    Wine = mkCN (mkN "wine") ;
    Pizza = mkCN (mkN "pizza") ;
    Cheese = mkCN (mkN "cheese") ;
    Fish = mkCN (mkN "fish" "fish") ;
    Very quality = mkAP very_AdA quality ;
    Fresh = mkAP (mkA "fresh") ;
    Warm = mkAP (mkA "warm") ;
    Italian = mkAP (mkA "Italian") ;
    Expensive = mkAP (mkA "expensive") ;
    Delicious = mkAP (mkA "delicious") ;
    Boring = mkAP (mkA "boring") ;
}
```

Different languages tend to use syntactic structures in similar ways to express the same meanings. We also know that all languages in *GF Resource Grammar Library* implement the same syntactic structures. Therefore in most cases we only need to rewrite the lexical parts of a concrete syntax for a new language. Thus we again encounter copy-and-paste programing which is not wise to do specially in a functional programming paradigm. In order to solve this problem we introduce **functors**. A functor is a function that operates on modules. In *GF* a functor is a module that **`opens`** one or more **interfaces**. An interface is a module similar to a **`resource`**, but it only contains the types of **`opers`** and not their definitions. The definitions are then given in **instances** of this interfaces. Thus a functor is a module-level function taking instances as arguments and producing modules as values. Here is the functor implementation of the Foods grammar. The module header uses the keyword incomplete to indicate that FoodsI is a functor [2]:

```
incomplete concrete FoodsI of Foods =
     open Syntax, LexFoods in {
  lincat
    Comment = Cl ;
    Item = NP ;
    Kind = CN ;
    Quality = AP ;
  lin
    Pred item quality = mkCl item quality ;
    This kind = mkNP this_Det kind ;
    That kind = mkNP that_Det kind ;
    These kind = mkNP these_Det kind ;
    Those kind = mkNP those_Det kind ;
    Mod quality kind = mkCN quality kind ;
    Very quality = mkAP very_AdA quality ;
    Wine = mkCN wine_N ;
    Pizza = mkCN pizza_N ;
    Cheese = mkCN cheese_N ;
    Fish = mkCN fish_N ;
    Fresh = mkAP fresh_A ;
    Warm = mkAP warm_A ;
    Italian = mkAP italian_A ;
    Expensive = mkAP expensive_A ;
    Delicious = mkAP delicious_A ;
    Boring = mkAP boring_A ;
}
```

To obtain a complete concrete syntax this function (FoodsI functor) needs to take two instances of the interfaces **`Syntax`** and **`LexFoods`** as its arguments. This action is called a **functor instantiation** [2]. Accordingly the English concrete syntax can be written as shown in Figure 9 :

```
concrete FoodsEng of Foods = FoodsI with
  (Syntax = SyntaxEng),
  (LexFoods = LexFoodsEng) ;
```

Figure 9: The English concrete syntax by a functor [2]

It remains to show how interfaces and their instances actually look like. Figure 10 shows the **`LexFoods`** interface and, an English instantiation of it is shown in Figure 11 [2].

```
interface LexFoods = open Syntax in {
  oper
    wine_N : N ;
    pizza_N : N ;
    cheese_N : N ;
    fish_N : N ;
    fresh_A : A ;
    warm_A : A ;
    italian_A : A ;
    expensive_A : A ;
    delicious_A : A ;
    boring_A : A ;
}
```

Figure 10: A lexicon interface for Foods grammar [2]

```
instance LexFoodsEng of LexFoods =
    open SyntaxEng, ParadigmsEng, in {
  oper
    wine_N = mkN "wine" ;
    pizza_N = mkN "pizza" ;
    cheese_N = mkN "cheese" ;
    fish_N = mkN "fish" "fish" ;
    fresh_A = mkA "fresh" ;
    warm_A = mkA "warm" ;
    italian_A = mkA "Italian" ;
    expensive_A = mkA "expensive" ;
    delicious_A = mkA "delicious" ;
    boring_A = mkA "boring" ;
}
```

Figure 11: An English instance of the lexicon interface for Foods grammar [2]

Finally we see an instance of the lexicon interface for Italian in Figure 12 and the concrete syntax for Italian in Figure 13, which has now become very straightforward to write [2].

```
instance LexFoodsIta of LexFoods =
     open SyntaxIta, ParadigmsIta in {
  oper
    wine_N = mkN "vino" ;
    pizza_N = mkN "pizza" ;
    cheese_N = mkN "formaggio" ;
    fish_N = mkN "pesce" ;
    fresh_A = mkA "fresco" ;
    warm_A = mkA "caldo" ;
    italian_A = mkA "italiano" ;
    expensive_A = mkA "caro" ;
    delicious_A = mkA "delizioso" ;
    boring_A = mkA "noioso" ;
}
```

Figure 12: An Italian instance of the lexicon interface for Foods grammar [2]

```
concrete FoodsIta of Foods = FoodsI with
  (Syntax = SyntaxIta),
  (LexFoods = LexFoodsIta) ;
```

Figure 13: The Italian concrete syntax by a functor [2]

See also Section 2.3.1 for more information about how to write your own grammar for the *Quiz*.

2.1.3 GF Web Service API

Some of the functionalities available in the *GF* shell are also available via the *GF* Web Services API [13], which is a small application that exposes the *PGF* API in form of a web service. The application uses FastCGI [14] as communication protocol to talk with the web server, and the data protocol in use is JSON [15][16].

We can list the most important tasks in the *Quiz* application as follows:

- Generating the question

- Evaluating the user answer

- Analyzing the user answer and providing instructive feedback

In all these phases we need to call the *PGF* server which is handled by the *GF* Web Services API [13]. Moreover as the *Quiz* application is written in JavaScript, we have used an additional interface for JavaScript provided by Thomas Hallgren which is available in `pgf_online.js` [17, 18].

Here is a table containing calls to the *PGF* server that we used directly in the *Quiz*, in the format they appear in the JavaScript API defined in `pgf_online.js` [17, 18]. We have only mentioned the input and output elements that were used in the *Quiz* and not all existing ones. More details about how the calls were applied in the *Quiz* can be found in Section 2.2.3, Functionalities.

Call as in JavaScript API	Input	Output
server.get_random	-	the generated abstract syntax tree
server.linearize	the abstract syntax tree to linearize, language to use in the linearization	the linearization result text
server.linearizeAll	the abstract syntax tree to linearize, language to use in the linearization	a list of all possible linearizations text
server.parse	the string to be parsed	a list of abstract syntax trees as plain strings

Table 1: Calls to the *PGF* server defined in the JavaScript API in `pgf_online.js`

The following calls were used indirectly in the *Quiz* for they exist in the minibar application [17]. We have slightly modified minibar - to meet our *Quiz* requirements - and integrated it into the *Quiz* application as a component:

- options.grammars_url

- server.get_grammarlist

- server.switch_grammar

- server.get_languages

For a list of all supported calls by the *GF* Web Service API please refer to the *GF* Wiki page available at [13]:

http://code.google.com/p/grammatical-framework/wiki/GFWebServiceAPI

Also in another guide by Thomas Hallgren, you will find these calls illustrated by examples, in addition to a detailed reference on how to make these calls from JavaScript using the API defined in `pgf_online.js` [17, 18]. This guide is available at [18]:

http://www.grammaticalframework.org/demos/minibar/gf-web-api-examples.html

More information about how to run a web server and launch the *Quiz* application with your own grammar is available in Appendix B: 6.2.

2.1.4 HTML and DOM

Hyper Text Markup Language abbreviated as HTML is the predominant markup language for web pages, where a markup language is a set of markup tags designed to describe web pages. The web page content consists of HTML elements defined by "tags" surrounded by angle brackets (like <html>). HTML tags normally come in pairs like and . The first tag in a pair is the start tag, the second tag is the end tag (they are also called opening tags and closing tags). The role of a web browser is to use the tags to interpret the content of the page to be able to read HTML documents and display them as web pages [19].

The base of the *Quiz* application is a web page written in HTML, where HTML elements form its building blocks. It serves as a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, links, forms, quotes and other items. All the *Quiz* functionalities are added in JavaScript to accompany this HTML base [19].

The HTML base and the JavaScript component have evolved in parallel and at the same time in an interactive way through the application development history. One example of this interaction is transferring to a DOM (Document Object Model) [20] in order to interact dynamically with the HTML elements through JavaScript rather than static elements in the HTML file. Here we explain DOM and how this change was done in the *Quiz*:

The Document Object Model (DOM) is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML [21] and XML documents [22]. Aspects of the DOM (such as its "Elements") may be addressed and manipulated within the syntax of the programming language in use, which is JavaScript in our case. DOM is likely to be best suited for applications where the document must be accessed repeatedly or out of sequence order, which is the exact case with our *Quiz* application. DOM has been applied to make

it possible to inspect or modify a web page dynamically by JavaScript scripts. In other words, DOM exposes its containing HTML page and browser state to JavaScript [20].

Getting to know the Minibar application - as we are using it in the *Easy Study Mode* of the *Quiz* to make word magnets available for the user to be used in constructing answers - and already facing problems with static elements in the *Quiz*, the design applied in the Minibar was an inspiration to change the *Quiz* approach to use DOM as well. For this purpose, functions available in support.js from the Minibar application [17], have been applied to access and modify the HTML elements from the JavaScript code dynamically. See Section 2.2.1 for a detailed example on how applying DOM benefits the *Quiz* application.

2.1.5 CSS

Another change that took place in the *Quiz* development time was using a CSS external sheet to include the entire application's styling and layout instead of the inefficient style attributes in HTML tags for the purpose of controlling the layout and style of the web page.

CSS stands for Cascading Style Sheets is a style sheet language used to describe the presentation semantics (the look and formatting) of a document written in a markup language. Its most common application is to style web pages written in HTML and XHTML [21], but the language can also be applied to any kind of XML document [22] [23].

From the historical point HTML was never intended to contain tags for formatting a document but only to define the content of a document. Therefore when tags like <font>, and color attributes were added to the HTML 3.2 specification, it started a nightmare for web developers. Development of large web sites, where fonts and color information were added to every single page, became a long and expensive process. To solve this problem, World Wide Web Consortium (W3C) created and added CSS to HTML 4.0. [24]. Accordingly CSS is designed primarily to enable the separation of document content (e.g. written in HTML or a similar markup language) from document presentation, including elements such as the layout, colors, and fonts. This separation can improve content accessibility, provide more flexibility and control in the specification of presentation characteristics, enable multiple pages to share formatting, and reduce complexity and repetition in the structural content. Thus External Style Sheets stored in .css files can save a lot of work both in the development and maintenance phases [23].

2.1.6 JavaScript

Officially managed by Mozilla Foundation, JavaScript is a scripting (light weight programming) language that was primarily designed to add interactivity to HTML pages by providing enhanced user interfaces and dynamically. JavaScript is usually embedded directly into HTML pages as it is an interpreted language (means that scripts execute without preliminary compilation) [25]. However as the amount of the JavaScript code is considerably large in the *Quiz* application - about one kilo lines of code without taking the minibar application code into account -, we have organized it in modules which are in the from of separate external .js files that accompany the .html page. These modules are available in detail in Appendix C: Section 6.3 of this document.

In the *Quiz* application we have used JavaScript in the form of client-side JavaScript - as JavaScript's primary use -, which is known to work in all major browsers, such as Internet Explorer [26], Firefox [27], Chrome [28], Opera [29], and Safari [30] [25]. Some other characteristics of JavaScript which are worth mentioning here are: It is a prototype-based, object-oriented scripting language which also supports the structured programming paradigm. It is dynamic, weakly typed as types are associated with values and not with variables. It has first-class functions which are objects themselves. So they have properties and methods, and they can be assigned to variables, passed as arguments, returned by other functions, and manipulated like any other object. JavaScript is also considered a functional programming language like Scheme [31] and OCaml [32] because it has *closure* and supports higher-order functions. By *closure*, JavaScript allows a combination of code that can be executed outside the scope in which it is defined, with its own scope to be used during that execution [33]. With all its diverse properties, JavaScript as all other programming languages has many advantages and also some drawbacks. Here we will briefly mention those we encountered. Let's start with its advantages:

- Advantages

- It is open source, and therefore accessible to all without the need to purchase a license.
- JavaScript supports regular expressions, which provide a concise and powerful syntax for text manipulation that is more sophisticated than the built-in string functions [33]. We have used regular expressions in preprocessing the raw user input, as we will explain

in detail in Section 2.2.3, under Preprocessing and Evaluating the user's answer (*Check Answer*).

- Being very popular today, JavaScript has abundant freely available supporting material and communities in the Internet.

- Drawbacks

- The characteristic of being weakly typed, makes the programming very easy indeed but some times has unexpected consequences which are difficult to avoid and costs more effort to be controlled eventually.
- I encountered the same traditional problems that exist mostly in structured programming paradigm, especially as the code was growing larger. It was very likely to end up in a redundant, spaghetti code. However enforcing a modular design - manually by the programmer - can help a lot in this case. This solution in return, requires a thoughtful design to make it also easy to keep track of the flow control with all those modules.

2.1.7 BLEU an Automatic MT Evaluation Method

In order to be able to give instructive feedback, we need to analyze and evaluate the answer given by the user. For this purpose a simplified version of MT (Machine Translation) evaluation methods has been applied. Although it might seem a totally different problem, our purpose has a lot in common with MT evaluation process. In both cases, the input is a given translation which needs to be evaluated by being compared against one or more reliable reference translations. Therefore we found it very useful to use the same techniques. The main technique applied here has a lot in common with the famous method known as BLEU (BiLingual Evaluation Understudy) [4]. Here we describe briefly how BLEU works, then will continue with the method applied in the *Quiz*. More general, end-user information can be found in Section 2.2.3 under , Analyzing the user's answer and Providing Instructive Feedback (*Hint*).

First of all let's see how Blue answers this fundamental question: How can we judge a translation? "The closer a machine translation is to a professional human translation, the better it is". This is the central idea behind the BLEU method. "To judge the quality of a machine translation, one measures its closeness by a numerical metric to one or more

reference human translations". Therefore, BLEU requires two ingredients: first a numerical "translation closeness" metric, and second one or more good quality reference translations [4].

In our application we already have the second part, i.e. at least one grammatically accurate, one hundred percent correct translation, namely linearizations of all possible abstract trees to the target language by the *PGF* server which we will use as our reliable reference. For the first part in the BLEU method "The main idea is to use a weighted average of variable length phrase matches, against the reference translations " [4, 34]. We should note that, our requirements is different from judging and deciding which translation is a better one; our goal is to give accurate feedback. We also have free variations explicitly existing in our reference translations, so we do not need to care about that either. Because of these differences the metric applied in the *Quiz* is also different, i.e. it is simplified so that it only considers words as phrase elements. Moreover it keeps track of the position of words in the phrase. It works like this: the user answer is compared with all possible correct answers, and the one with the most common words in the same positions as in the user's answer, is selected as the reference.

For the final evaluation, I have applied a strategy very similar to the Mastermind's game [35]. Words in green stand for correct words in their correct place in the sentence, and yellow words mean these words are correct and are part of the right answer but they are misplaced, while red words stand for words which do not exist in the right answer and are either misspelled or totally wrong lexically or grammatically. The *Quiz* also displays blanks for missing words.

2.2 Solution Details

In this section solution details including the design, implementation and testing phases are discussed in a general sense, for more technical details refer to Appendix C: Section 6.3, Developer's Guide.

2.2.1 User Interface

The user interface which is in form of a web page has been designed to be extremely compact and narrow in order to fit in a mobile phone screen, Figure 14 shows its initial state. The layout and styling issues has been mostly handled in a CSS external file as explained in Section 2.1 Methods and Technology in Subsection 2.1.5. Also most HTML elements like the *Menubar* and other buttons are added dynamically from within

the JavaScript code applying DOM instead of being statically defined in the HTML code. This makes the application more flexible and gives the programmer a better control of the application and also makes the maintenance process much easier. For this purpose functions in support.js from the Minibar application [36] have been applied.

To demonstrate the strength and flexibility provided by this method, we give a small example. The *"Check Answer"* button the user sees in the *Easy Study Mode* of the *Quiz* (Figure 15) looks the same as it does in other *Quiz Modes*, however it is a totally different button, i.e. it performs completely differently in this mode. At application start-up, it is defined as a submit button so that the Enter key will be its shortcut. Unfortunately this useful shortcut will become a programmer's nightmare in the *Easy Study Mode* where the Enter key has another role, i.e. to take the typed text by the user and add it to the answer being constructed. This causes a serious conflict with the role of submitting the user's answer for checking. In order to avoid this disaster and keep the benefit of checking the answer by the Enter key, the *"Check Answer"* button is removed and replaced with a none-submit one while changing to the *Easy Study Mode* and vice versa. See more details about DOM in Section 2.1.4.

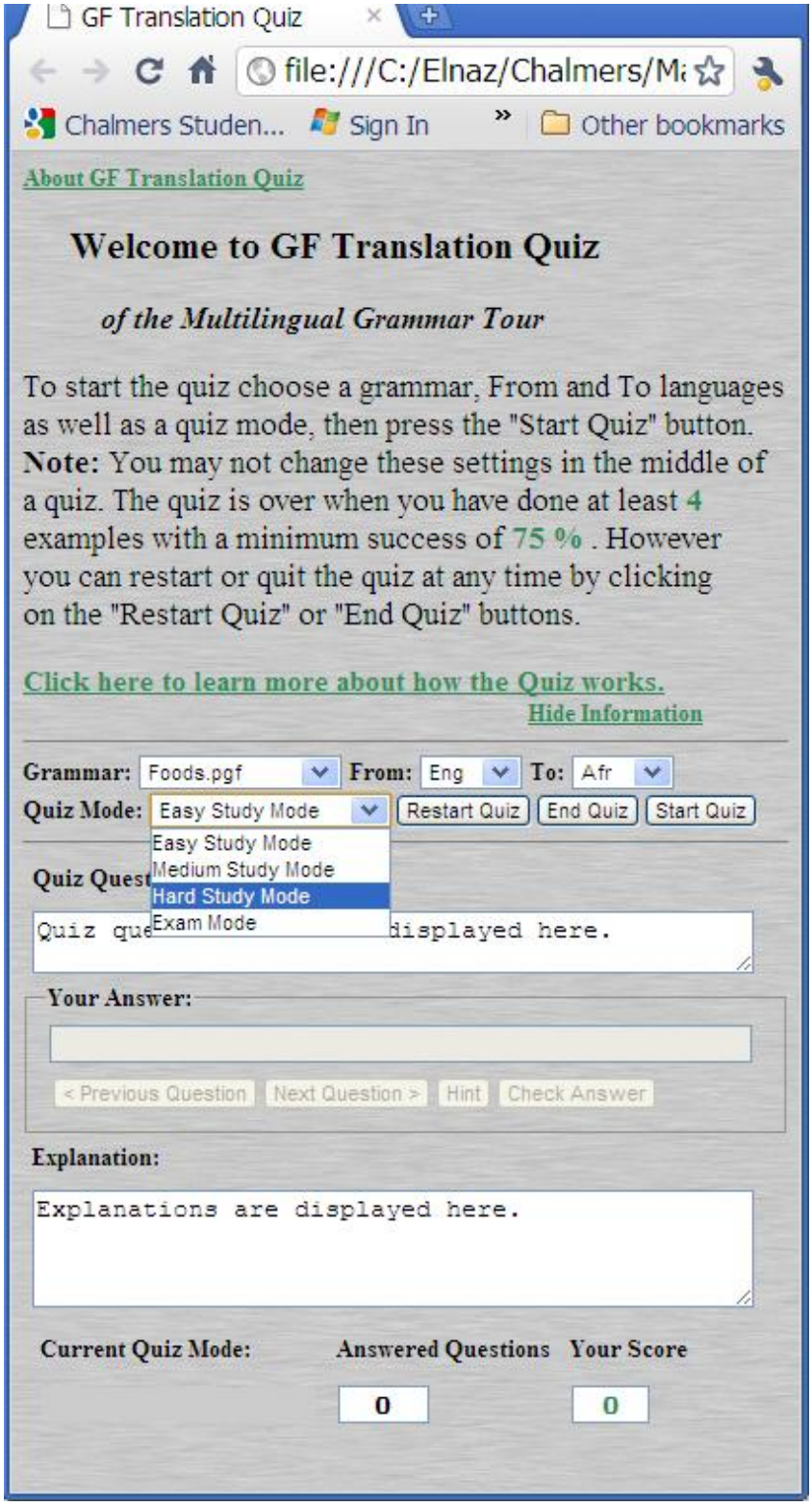

Figure 14: The initial state

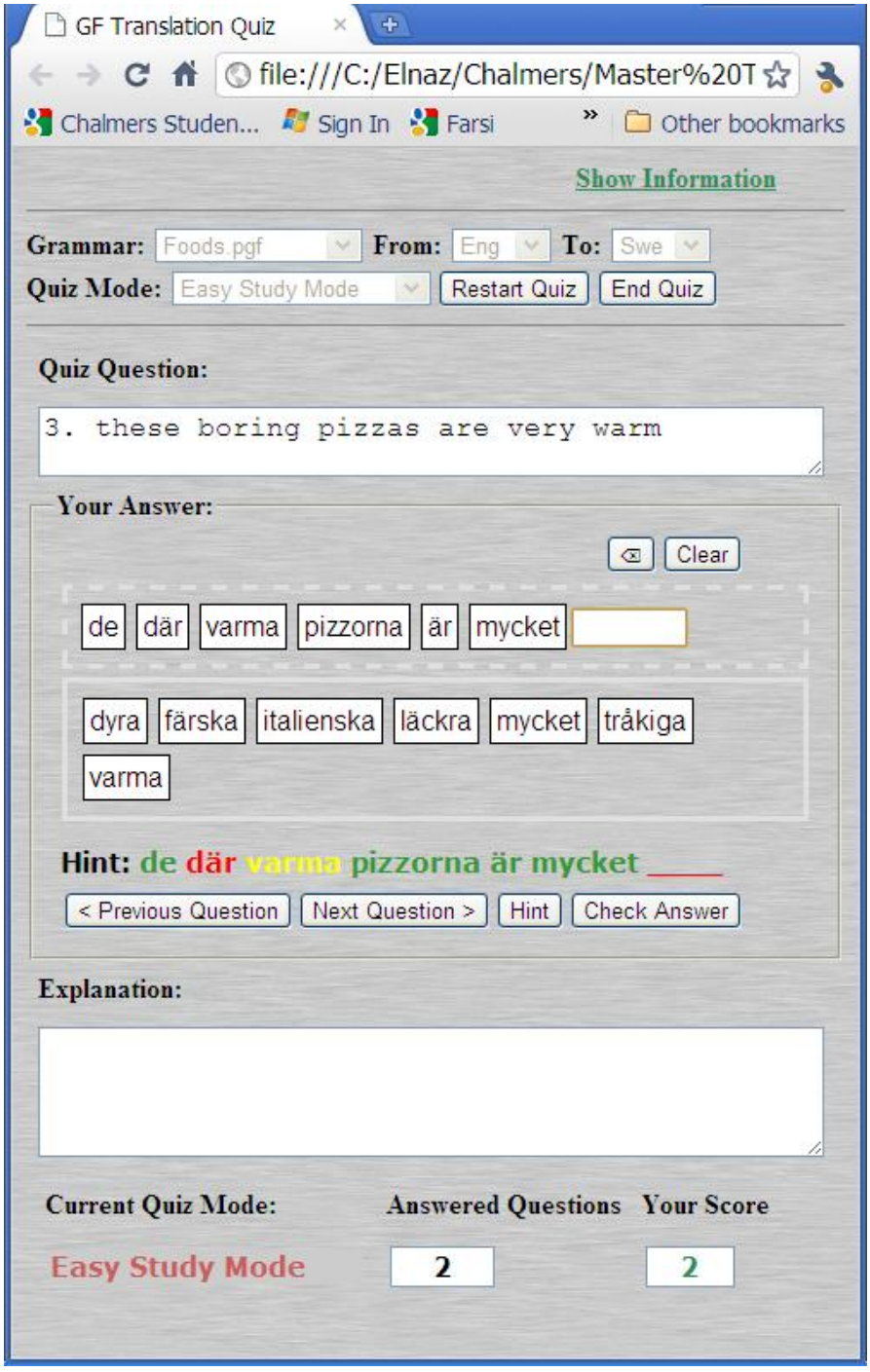

Figure 15: The *Easy Study Mode*

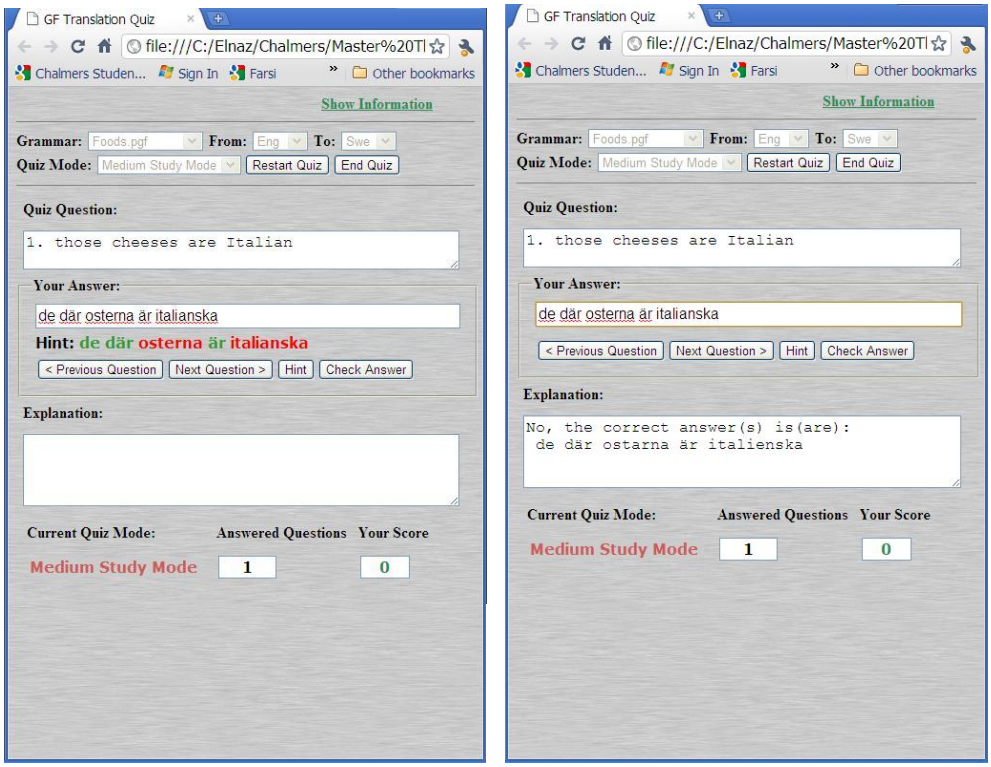

Figure 16: *Hint* vs. *Check Answer*

2.2.2 Configurations

The application has three different ways to be fine-tuned. Through *User Customizations* designed in the interface for end users, *Configuration Variables and Mode Settings* found within the code - which are mostly aimed at instructors and teaching supervisors -, and *Server Configurations* which are related to the *GF* web services and are mostly aimed at developers and partly to instructors and teaching supervisors.

- User Customizations They are handled from the user interface and are designed to be as easy as possible. From the *Menubar*, users can customize the quiz to their needs, i.e. they can choose their desired *Grammar*, *From* (source) and *To* (target) languages and also a *Quiz Mode*:

- ***Easy Study Mode*:** In this mode word magnets are available to help users; they can type and/or click on them freely to construct their answer. "*Delete last*" and "*Clear*" buttons can be used to delete the last word or the whole answer in case required. These

word suggestions come from the selected grammar, and therefore it keeps users from misspelling words and even making grammatically wrong sentences. Also, users have unlimited *Hint* which is automatically updated as the answer is being modified. It is also possible to go to one previous question at a time; and of course, users can check their answer's correctness.

- ***Medium Study Mode*:** In this mode users can use the *"Hint"* button for a maximum of 3 times for each question, and they can go to one previous question at a time as well as checking their answer's correctness.
- ***Hard Study Mode*:** In this mode users can use the *"Hint"* button only once for each question, and they may not use the *"Previous Question"* button, but they can still check their answer's correctness.
- ***Exam Mode*:** In this mode users cannot use the *"Hint", "Previous Question"* or *"Check Answer"* buttons, and they will not see their score until the end of the exam.

This piece of information is also available in the application through a link just above the *Quiz Menubar*.

- Configuration Variables and Mode Settings There are some global variables defined in the code that control the application's general settings. We have divided these variables in two groups: *Configuration Variables,* which affect the whole quiz; and *Mode Settings* that define the *Quiz Modes* differences, and thus makes it possible to change the *Modes* behavior as desired. Generally these configurations and settings are aimed at instructors and teaching supervisors, and they are only modifiable from within the code. They are explained in more detail in Appendix B: Sections 6.2.1 and 6.2.2.

- *Configuration Variables*: which consist of minimum percentage of right answers (score/counter) required to pass the *Quiz*, minimum number of questions required to pass the *Quiz*, number of questions in the *Exam Mode*, which is a fixed number, and finally maximum times the user may answer the question to increase his/her score.
- *Mode Settings*: which makes it possible to modify inner settings for all mode properties, including: access to the Minibar magnet words, *"Previous Question*", *"Hint"* and *"Check Answer"* buttons, etc.

- The Server Configurations They define the *PGF* server the *Quiz* needs to communicate with, and they are modifiable only from within the code. It is possible to choose between a predefined server accessible over the Internet, or a local server. Below is an example code; the variable in charge of this setting is called **`online_options`**. More details about the *GF* Web Service API can be found in Section 2.1.3. Also in Appendix B: Section 6.2.3, you can find information on how to launch this web service on your own server.

```
var online_options ={
grammars_url:"http://www.grammaticalframework.org/grammars/"
//grammars_url:"http://tournesol.cs.chalmers.se:41296/grammars",
//grammars_url: "http://localhost:41296/grammars",
//grammar_list: ["Foods.pgf"],

    }
```

2.2.3 Functionalities

Here is a list of the *Quiz'* main functionalities, that we will discuss its items individually afterwards. We will also mention the methods and technologies behind their implementation very briefly. More details about the methods and technologies themselves can be found in Section 2.1.

- **Generating a new question** (*"Next Question"*)
- **Preprocessing and Evaluating the user's answer** (*"Check Answer"*)
- **Analyzing the user's answer and Providing Instructive Feedback** (*"Hint"*)
- **Possibility to see the whole Quiz** (*"Show Quiz History"*)

- Generating a new question (*"Next Question"*)**:**

Triggered by the *"Next Question"* button, the `generate_question()` function from the JavaScript is called, which in turn sends a random tree generation request to the *PGF* server by this command `server.get_random()`, finally the `server.linearize()` will linearize the generated random abstract tree to the selected source language, and display the result to the user.

- Preprocessing and Evaluating the user's answer (*"Check Answer"*):

Users can check whether their answer is right by either clicking on this button or pressing the Enter/Return key of their keyboard. What happens from a user's point of view is that he/she will receive a basic feedback in this phase which is either approval: "Yes, that was the correct answer." or denial "No, the correct answer(s) is(are): ..." which reveals the correct answer(s). Users have to be careful because checking the answer also submits their answer, and for each question they have only one submission chance to increase their score; further submissions don't have any impact on their score. Therefore it is recommended to use the *Hint* first, to make sure they have got the right answer; of course if *Hint* is available (see below). Moreover, following the "learning from your mistakes" approach, it is more effective for the learners to first find where they were wrong rather than seeing the right answer all at once. Here we explain the underlying procedure in a simplified step by step approach:

1- The *"Check Answer"* button triggers the `check_answer_quiz()` function in the JavaScript code which in return calls the `make_all_answers()` function if it is the first time this question is being checked and *Hint* has not been called before. Otherwise it calls the `continue_checking()` in Step 3.

2- Here we explain what happens in `make_all_answers()` function. There are cases where a single word in the source (*From*) language translates to two or more different words with distinct meanings and applications in the target (*To*) language. For example the word "you" in English becomes "du" (singular you) or "ni" (plural you) in Swedish. Therefore there are more than one correct answers in these cases. Moreover *GF* grammars may include "free variations, which are alternative concrete syntax objects that map to the same abstract syntax tree" [2]. This means that a single word in a certain language may have more than one translation equivalents. For example the verb "run" in English may have "springer" and "löper" as its equivalent translations in Swedish concrete syntax. Taking care of these possibilities, the function `make_all_answers()` produces an array containing all possible correct answers by following these steps, also illustrated in Figure 17 :

- 2-1- The random abstract tree generated by `generate_question()` function in the question generation phase, is linearized to the source language.

– 2-2- The result text is then parsed to the source language. This may result in additional trees to the one we started with, thus we can cover the first case that we explained in the above example.

– 2-3- The resulted abstract tree(s) are then linearized to the target language separately, for this step the linearizeAll command is used instead of ordinary linearize in order to get all translation alternatives including free variation.

– 2-4- Then it will continue with `continue_checking()` in Step 3.

3- `continue_checking()` will continue running the checking process in three branches:

– 3-1- `check_answer_exam2()` is called if the quiz is in the *Exam Mode*, which behaves differently in terms of showing immediate reaction to user's actions. We will discuss it later.

– 3-2- `show_hint2()`is called in case the *"Hint"* button was clicked, which we will discuss later in the following functionality called *Hint*.

– 3-3- `check_answer2()`is called if the two previous conditions fail. The first action taken in `check_answer2()` is preprocessing the raw user answer. For this purpose we have applied regular expressions:

3-3-1- First, all unwanted characters including digits, punctuation marks and in brief all none-letter characters are replaced by a space character; apostrophe is exceptionally allowed for its semantic role in the many grammars. These characters are defined by the following regular expression in terms of ranges of Unicodes:

```
/[\u0021-\u0026 \u0028-\u0040 \u005b-\u0060 \u007b-\u007e]+/g
```

3-3-2- Then all extra spaces defined in this regular expression: `/^\s+|\s+$/g` , are removed and finally all white-space characters stated by the regular expression: `/\s+/g` , are replaced with a single space which is required in the analyzing phase taking place in the next phase, namely the *Hint* feature of the *Quiz*. This approach makes the application more robust and flexible towards the user's subtle mistypings and even towards a careful high standard input including punctuations that might not be supported in the grammars.

3-3-3- `check_answer2()` then compares the processed user answer to all possible correct answers computed in Step 2 by `make_all_answers()` function, and finally displays the result of this phase in the explanation part of the *Quiz* interface.

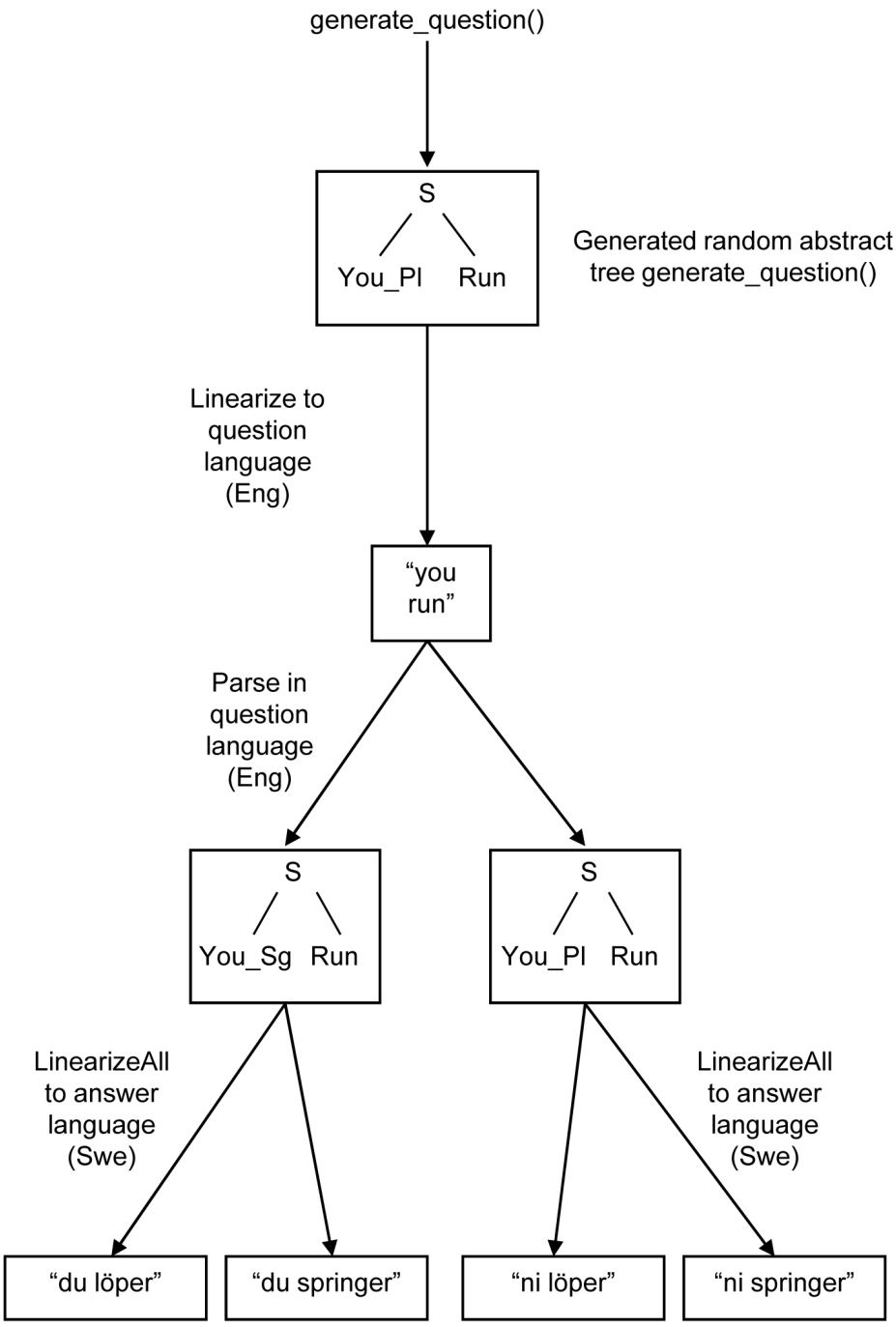

Figure 17: How all possible correct answers are generated

- Analyzing the user's answer and Providing Instructive Feedback (*"Hint"*):

Besides the basic feedback provided in the *Check Answer* (see previous functionality), there exists a more sophisticated feature in the *Quiz,* called the *Hint*. This feature is created to fulfill one of the main goals of this project, namely providing users with instructive and helpful feedback. The *"Hint"* button is available in all *Quiz Modes* but the *Exam Mode*. With automatic update upon input change and no usage restriction, it is designed to be extremely handy in the *Easy Study Mode*. Actually with the help of this feature it is very hard for the users to go wrong. The evaluation process combines a simplified version of the MT evaluation method called BLEU, together with a strategy very similar to the Mastermind's game [35]. For technical details of the applied methods in the *Quiz Hint* see Section 2.1.7. Although there is some interoperability between *Check Answer* and *Hint,* the *Hint* mechanism is designed to work completely independent of the *Check Answer.* As we did for the *Check Answer* phase, here we explain the *Hint* procedure in a simplified step by step approach:

1- The *"Hint"* button triggers the `show_hint()` function in the JavaScript code which in return calls the `make_all_answers()` function if it is the first time *Hint* is called and this question has not been checked before. Otherwise it calls the `continue_checking()` in Step 3.

2- What happens here in the `make_all_answers()` function, we have already explained in Step 2 of *Check Answer* which we avoid repeating.

3- `continue_checking()` will continue running the checking process in three branches:

- 3-1- `check_answer_exam2()` is called if the quiz is in the *Exam Mode*, which follows similar steps as the *Check Answer*, but behaves differently in terms of showing immediate reaction to user's actions, e.g. it reveals the final score only at the end of the exam.
- 3-2- `check_answer2()` is called which we have discussed, in the previous phase. Please see Step 3-3- in *Check Answer.*
- 3-3- `show_hint2()` is called in case the *"Hint"* button was clicked. The first action taken in `show_hint2()` is preprocessing the raw user answer, which we have explained in the previous functionality, *Check Answer*.

3-3-1- Please see Step 3-3-1- in *Check Answer*.

3-3-2- Please see Step 3-3-1- in *Check Answer*.

3-3-3- `show_hint2()` then compares the processed user answer to all possible correct answers computed in Step 2 by `make_all_answers()` function, and displays the result as explanation below:

Words in green stand for correct words in their correct place in the sentence, and yellow words mean these words are correct words and are part of the right answer but they are misplaced, while red words stand for words which do not exist in the right answer and are either misspelled or totally wrong lexically or grammatically. *Hint* also displays a blank for each missing word.

- Possibility to have the whole Quiz (*"Show Quiz History"*):

The application keeps a record of the quiz questions, user's answers and the explanations as a quiz is running. At the end of the quiz, which is either invoked manually by the user clicking on the *"End Quiz"* button, or by the application itself (e.g. when the pass condition occurs or an exam ends), a *"Show Quiz History"* button will be displayed. Clicking on this button will reveal all quiz questions together with user's answers and explanations from the application in a separate window, where the user has the possibility to review, save a copy and also print the History.

2.3 Extra Features

2.3.1 How to make your own quiz

This part is mostly aimed at teachers and education supervisors who wish to make their own desired quizzes. This simply requires writing their own grammar and applying it in the current web interface of the *Quiz*. Apart from the ordinary way of writing a *GF* grammar and by help of *GF Resource Grammar Library* - that we saw in Sections 2.1.1 and 2.1.2 - there are two additional possibilities for writing a *GF* grammars. Although these methods do not cover the complete *GF* grammar notation until this moment, but they are very much easier than the previously mentioned ways, and still meet the needs of non-GF-grammarian users. Here we discuss them briefly:

- GF online editor for simple multilingual grammars For the means of *GF* grammar writing a recent feature has been introduced which makes writing grammars much easier specially for non-specialist users. It is called *GF online editor for simple multilingual grammars.* This online editor, written by Thomas Hallgren, is available at:

http://www.grammaticalframework.org/demos/gfse/

"Traditionally, *GF* grammars are created in a text editor and tested in the GF shell. Text editors know very little (if anything) about the syntax of *GF* grammars, and thus provide little guidance for novice *GF* users. Also, the grammar author has to download and install the *GF* software on his/her own computer. On the contrary in the online editor all that is needed is a reasonably modern web browser" [37].

"The editor also guides the grammar author by showing a skeleton grammar file and hinting how the parts should be filled in. When a new part is added to the grammar, it is immediately checked for errors. Last but not least in spite of its name, the editor runs entirely in the web browser, so once you have opened the web page, you can continue editing grammars even while you are offline" [37]. Thus users can write their grammar with the online editor in an easy interactive way, and test them in the *Minibar* [17] or *Quiz* frameworks afterwards.

- The *gfm* format Another approach to grammar writing which is even easier is writing them in a very simple and straight forward format called the *gfm* format. In this format a multilingual grammar is a simple text file with a .gfm suffix which contains language elements, including words, phrases and whole sentences in all destined languages in a multi-column style so that they could be easily mapped to one another.
Here is an example which relates some English, Italian and Swedish nouns and noun phrases.

```
langs Eng Ita Swe
man      ; uomo      ; man
woman    ; donna     ; kvinna
boy      ; ragazzo   ; pojke , kille
girl     ; ragazza   ; flicka, tjej
a good man  ; un buon uomo   ; en god man
a good woman; una buona donna; en god kvinna
these girls ; queste ragazze ; dessa flickor, de här flickorna
these boys  ; questi ragazzi ; dessa pojkar , de här pojkarna
```

The text is then saved in a file named `Foods.gfm`. The compiler recognizes the suffix `.gfm` and creates these four files: `Foods.gf`, `FoodsEng.gf`, `FoodsIta.gf` and `FoodsSwe.gf`.

The first line gives the list of applied language names. Multi-word expressions and free variations are allowed. Words of different languages are separated by a '**;**' while free variants within a language are separated by a '**,**'. As a grammar grows larger - as in the above example - the benefits of the modular grammar writing are sensed better. However this method works very well for small grammars, with simple and short phrases.

2.3.2 How the *Translation Quiz* handles morphology questions

Aarne Ranta has suggested a simple and splendid idea that makes it possible to apply the *Quiz* for training morphology inside a specific language as well. This is possible by defining a grammar in a way that the abstract syntax module contains the morphology forms while one concrete syntax maps these forms to their descriptions and another concrete grammar maps them to their actual forms. Thus a translation from descriptions to real forms simply makes a morphology quiz. One example grammar of this kind is available on the grammatical framework server: it is called **`MorphoQuizVerbsEng.pgf`** and is written by A. Ranta for means of training English irregular verb forms. It gives the form(s) description(s) and expects the real form(s) as an answer. The rest is quite the same as with the ordinary grammars. This feature adds another dimension to the *Quiz* and again more specific grammars can be written to enhance this usage of the *Quiz* application.

3 Evaluation

This section is organized in two parts: First we discuss Limitations and Drawbacks, and then Advantages. They contain the author's observations and concerns and also some users' points of view.

3.1 Limitations and Drawbacks

Here is a list of limitations and some undesirable aspects of the application:

- It requires a high speed internet connection, otherwise the latency in receiving responses to each server call will be quite annoying from a user's perspective specially in the case of remote users.

- The auto-fill feature which is set as default nearly in all browsers is not so desirable in the *Quiz* because it keeps everything the user has typed and suggests them while they want to write a new answer. This is both distracting from a learner's point of view and also might be considered as cheating from a teacher's point of view in case of a serious exam.

- It lacks a long term history mechanism, for example a user account system, so that the learners can keep track of their training in the long run.

3.2 Advantages

- Easy to use, which covers a large range of users including school children.

- Being based on *GF* makes it possible to consider and take care of complex language concepts like, free variations (where translation alternatives are possible) and the reverse situation of ambiguity (where a single word has more than one translations with distinct meanings in another language).

- Extendable by adding grammars and languages.

4 Future Work

Here is a list of possible future work some of which come directly as solutions to the problems stated in the Limitations and Drawbacks in Section 3.1:

- Designing specific grammars in order to fulfill the *Quiz* demands.

- Controlling and directing the random generation of phrases and sentences.

- Adding an initial test for determining user's knowledge level can be added. This can help users to know where to start (i.e. which level).

- Defining and making the *Quiz* configurable for different levels of language knowledge, by means of designing specific grammars and defining equivalent levels adapting to standards like the Common European Framework of Reference for Languages' Learning, Teaching and Assessment; abbreviated as CEFR [7].

- Introducing an offline version of the *Quiz* which does not require a constant internet connection and is much faster in the case of low speed connections or remote users.

- Introducing a user account system with capability of keeping the records of learners' training so that they can keep track of their overall progress and find their weak and strong points. This information can be useful for teachers or supervisors the same way. Thus the training can be more organized and fruitful.

- The above history system can also provide some useful feedback for the *Quiz* itself, for example the questions that the user got wrong could appear again soon or the ones he got right may not appear again and so on.

- Enhance the *Hint* so that it can catch user errors in letters scale in addition to words scale. For example it can detect wrong or misplaced letters in a word. Maybe something similar to what is done in Rivstart web-page. For more details see Rivstart in Section 1.2.3.

5 Conclusion

In brief the aim of this thesis work is introducing an automated exercise generator to be used in training grammars in general as well as the lexical and syntactic aspects of different human languages. This tool is designed and implemented as a web application - which we have called the *GF Translation Quiz* or in short the *Quiz* - that makes it also available on mobile phone platforms such as iPhone and Android. We have a very good reason to call it thus: *GF (Grammatical Framework)* and its mostly unique and conceptual approach toward translation and multilinguality is the major source of inspiration to this work and forms the basis that this thesis is built upon its strength. Therefore most characteristics of the *Quiz* application, i.e. grammatical precision, support for multilinguality, and coverage of natural languages in small fragments are provided by *GF*. From another point of view, the *Quiz* application plays the role of an interface, so that ordinary users with no prior computer knowledge can also benefit from *GF*'s capabilities for the purpose of language learning.

The *GF Translation Quiz* provides an extendable frame work from two different dimensions:

- Creating new exercises by defining new abstract syntax grammars
- Adding new languages by writing the corresponding concrete syntax grammar

However although the current frame work is quite accomplished and robust but there are a lot of possible future work which can enhance the application further. Some are as follows:

- Creating a user account system for keeping a long term history for each user which allows users to keep track of their training in the long run
- Designing specific grammars in order to fulfill the *Quiz* demands
- Controlling and directing the random generation of phrases and sentences
- and so on ...

For more future work suggestions please see Section 4.

6 Appendices

6.1 Appendix A: User manual

This chapter explains all the *Translation Quiz*'s features and functionalities and how they work with an intention for end users. See also Figures 14 and 15 for sample states of the user interface .

6.1.1 Customization:

From the *Menubar*, users can customize the quiz to their needs, i.e. they can choose their desired *Grammar*, *From* (source) and *To* (target) languages and also a *Quiz Mode*:

- ***Easy Study Mode*:** In this mode word magnets are available to help users; they can type and/or click on them freely to construct their answer. *"Delete last"* and *"Clear"* buttons can be used to delete the last word or the whole answer in case required. These word suggestions come from the selected grammar, and therefore it keeps users from misspelling words and even making grammatically wrong sentences. Also, users have unlimited *Hint* which is automatically updated as the answer is being modified. It is also possible to go to one previous question at a time; and of course, users can check their answer's correctness.
- ***Medium Study Mode*:** In this mode users can use the *"Hint"* button for a maximum of 3 times for each question, and they can go to one previous question at a time as well as checking their answer's correctness.
- ***Hard Study Mode*:** In this mode users can use the *"Hint"* button only once for each question, and they may not use the *"Previous Question"* button, but they can still check their answer's correctness.
- ***Exam Mode*:** In this mode users cannot use the *"Hint"*, *"Previous Question"* or *"Check Answer"* buttons, and they will not see their score until the end of the exam.

This piece of information is also available in the application through a link just above the *Quiz Menubar*.

6.1.2 Functionalities:

As most of the *Quiz*'s functionalities are quit self-explanatory, in this part only the none evident functionalities are explained to make them clear and prevent probable misinterpretations:

- ***Next Question*:** As it is expected to, it will bring up the next question, and it will prompt you before doing so if you have not checked your answer to the current question so that you don't miss a question by mistake. The only matter you should know about it is that if your answer to the previous question has been correct, *"Check Answer"* moves you automatically to the next question so that you won't need an extra click on the *"Next Question"*.
- ***Check Answer*:** You can check whether your answer is right by either clicking on this button or pressing the Enter/Return key of your keyboard. (Exception: In the *Easy Study Mode* you cannot use the Enter/Return key for checking your answer, because it has another functionality which is adding typed words to your answer.) Please note that by checking your answer you also submit it, and for each question you have only one submission chance to increase your score - further submissions don't have any impact on your score. Use the *Hint* if you are not sure and of course if it is available. Finally please note that *"Check Answer"* moves automatically to the next question if your answer is right.
- ***Hint*:** You will find the *"Hint"* button very handy in all *Quiz Modes*. In fact it works very much similar to the famous Mastermind game [35]. Words in green stand for correct words in their correct place in the sentence, and yellow words mean these words are correct words and are part of the right answer but they are misplaced, while red words stand for words which do not exist in the right answer and are either misspelled or totally wrong lexically or grammatically.
- ***Show Quiz History*:** The application keeps a record of the questions, user answers and the explanations as a quiz is running. At the end of the quiz, which is either invoked manually by the user clicking on the *"End Quiz"* button, or by the application itself (e.g. when the pass condition occurs or an exam ends), a *"Show Quiz History"* button will be displayed. Clicking on this button will reveal all quiz questions together with user's answers and explanations from the application in a separate window, where the user has the possibility to review, save a copy and print the History.

6.1.3 Technical Concerns

The *Quiz* application has been tested on different browsers like Google Chrome [28], Mozilla Firefox [27] and IE (Windows Internet Explorer) [26]. Due to technical problems with IE and previous graphical issues with Chrome the author recommends Firefox over all tested browsers as the most consistent browser for the *Quiz* application. However presently no problem has been detected with Chrome anymore.

6.2 Appendix B: Teachers/ Supervisor's Guide

This section is mostly aimed at teachers or education supervisors who would like to use this application as a means of training and/or evaluating their students language skills. At this level you might want to run the *Quiz* with other *GF* grammars, and so will need to run a server locally which will be discussed later on this chapter. Moreover the quiz can be tailored to meet the specific requirements of different users with different perspectives. For this purpose there exist some configurable parts in the JavaScript code, which is explained as follows:

6.2.1 Configuration Variables

Here is the list of configuration variables available in the beginning of the **`translation_quiz.js`** file, a JavaScript component of the *Quiz*:

- The minimum percentage of right answers (score/counter) required to pass the *Quiz* with current default value of 0.75 which equals to 75 %.

  ```
  var pass_percentage = 0.75;
  ```

- The minimum number of questions required to pass the *Quiz* with current default value of 10.

  ```
  var min_no_questions = 10;
  ```

- The number of questions in the *Exam Mode*, which is a fixed number with current default value of 20.

  ```
  var exam_quesNo = 20;
  ```

– The maximum times the user may answer the question to increase his/her score, with current default value of 1. However with the current *Check Answer* mechanism which reveals the right answer, it does not make much sense to change this variable.

```
var max_answer_times = 1;
```

6.2.2 Modes Settings

Here are the modes settings with their default values available under the same title (*Modes Settings*) in the **`set_mode()`** function in the **`translation_quiz.js`** file, a JavaScript component of the *Quiz*:

– **have_minibar :** Determines whether the *Quiz* has the word magnets available from the Minibar application [17]. Currently the default value is true only for the *Easy Study Mode* and false for all other modes.

– **have_prevQuestion :** Determines whether the user can use the *"Previous Question"* button. Currently the default value is true for *Easy Study Mode* and *Medium Study Mode* and false for *Hard Study Mode* as well as *Exam Mode*.

– **have_checkAns :** Determines whether the user can use the *"Check Answer"* button. Currently the default value is true for all modes but the *Exam Mode*.

– **max_hint_times :** Determines how many times the user can use the *"Hint"* button for each question. Currently the default value is unlimited times for *Easy Study Mode*, three times for *Medium Study Mode*, only once for *Hard Study Mode* and evidently zero times for *Exam Mode*.

6.2.3 Technical Concerns

– **Which Browser to use:** The *Quiz* application has been tested on different browsers like Google Chrome [28], Mozilla Firefox [27] and IE (Windows Internet Explorer) [26]. Due to technical problems with IE and previous graphical issues with Chrome the author recommends Firefox over all tested browsers as the most consistent browser for the *Quiz* application. However presently no problem has been detected with Chrome.

– **How to make your own quiz:** A detailed discussion about this can be found in Section 2.3.1.

– **How to install and run a PGF server locally:** For a detailed up to date tutorial please refer to the Grammatical Framework Wiki page available at :
http://code.google.com/p/grammatical-framework/wiki/LaunchWebDemos
Before that, you need to install *GF* itself for this step refer to the following page :
http://code.google.com/p/grammatical-framework/wiki/DevelopersPage

– **How to apply a new grammar:** For a detailed up to date tutorial please refer to the Grammatical Framework Wiki page available at:
http://code.google.com/p/grammatical-framework/wiki/GFWebServiceAPI

6.3 Appendix C: Developer's Guide

Finally in this section some information is provided for users at a higher level than end users; e.g. programmers/developers who might want to apply the *Quiz*'s code within their own applications.

6.3.1 Code Organization and Modules

The code is organized as follows:

– **`translation_quiz.html`,** which contains the main HTML elements of the *Quiz* in its general state.

– **`translation_quiz.js`**, the main JavaScript code which contains the global variables and different functions of the *Quiz* can be found here.

– **`quiz_pre_start.js`**, which prepares and sets the variables prior to page load.

– **`quiz_support.js`**, it handles the lower level functions applied in **`translation_quiz.js`**.

– **`minibar_quiz.js`**, the JavaScript code that is in charge of the word magnets, in the *Easy Study Mode* of the *Quiz*. It is based on the original **`minibar.js`** from the Minibar application [17] which is slightly modified in order to serve the *Quiz*'s requirements.

– **support.js**, which is the original file from the Minibar application [17]. It handles the lower level functions used in **minibar_quiz.js** and **translation_quiz.js**.

– **minibar_quiz.css** and **brushed-metal.png**, imported from the Minibar application [17] with slight changes in the CSS file, handle the styling and graphics in the *Quiz* application.

– **pgf_online.js**, which is the original file from the Minibar application [17]. It creates the server object with desired parameters.

6.3.2 Technical Concerns

The *Quiz* application has been designed and implemented to be compatible with Google Chrome (version 7.0.517.44) [28] as default browser. It has also been tested and verified with Mozilla Firefox (version 3.6.12) [27]. However it does not work with IE (version 6.0.29) [26] due to JavaScript restrictions.

Another important issue is that occasionally some disorder in displaying the word magnets has been observed in Chrome browser, which seems to be a rendering bug in Chrome. However this problem has been bypassed from within the Minibar application [36] in its released version on Oct 26th of 2010 and after that, and thus the same holds for the *Quiz* application.

References

[1] Spoken languages in the world, http://www.infoplease.com/askeds/many-spoken-languages.html [Accessed 2 November 2010]

[2] Ranta A., *"Grammatical Framework: Programming with Multilingual Grammars*", CSLI Publications, Stanford, 2011

[3] Google translator, http://translate.google.com/ [Accessed 1 November 2010]

[4] Papineni K., Roukos S., Ward T. and Zhu W.J., *"Bleu: a Method for Automatic Evaluation of Machine Translation"*, IBM Research Report, RC22176 (W0109-022), September 17, 2001.

[5] Moodle, http://moodle.org/ [Accessed 2 November 2010]

[6] EngOnline, https://learning.portal.chalmers.se/ [Accessed 2 November 2010]

[7] CEFR (Common European Framework of Reference for Languages), http://www.coe.int/t/dg4/linguistic/cadre_en.asp [Accessed 14 December 2010]

[8] Rivstart, http://www.nok.se/rivstart [Accessed 21 November 2010]

[9] Ranta A. and Angelov K., *"Implementing Controlled Languages in GF"*, CNL-2009, CEUR Workshop Proceedings, vol. 448, 2009

[10] Ranta A. *, "Grammatical Framework : A Type Theoretical Grammar Formalism",* The Journal of Functional Programming*,* 14(2), 145-189 , 2004

[11] Bart Jacobs, *"Categorical Logic and Type Theory"*, Elsevier, 2001, ISBN 9780444508539

[12] Ranta A., *"The GF Resource Grammar Library. Linguistic Issues in Language Technology"*, CSLI Publications, Volume 2, Issue 2, 2009

[13] *Grammatical Framework* Wiki page: *GF* Web Service API. http://code.google.com/p/grammatical-framework/wiki /GFWebServiceAPI [Accessed 2 November 2010]

[14] FastCGI, http://www.fastcgi.com/devkit/doc/fcgi-spec.html [Accessed 19 February 2011]

[15] JSON (JavaScript Object Notation), http://www.json.org/ [Accessed 14 February 2011]

[16] GF Wiki page: Launching the Web applications, http://code.google.com/p/grammatical-framework/wiki /LaunchWebDemos [Accessed 2 November 2010]

[17] Minibar Application, http://www.grammaticalframework.org/ demos/minibar/minibar.html [Accessed 2 November 2010]

[18] GF web services API examples, http://www.grammaticalframework.org/demos/minibar/gf-web-api-examples.html [Accessed 5 November 2010]

[19] HTML (Hyper Text Markup Language), http://www.w3.org/TR/html401/ [Accessed 10 November 2010]

[20] DOM (Document Object Model), http://www.w3.org/DOM/ [Accessed 11 November 2010]

[21] XHTML (Extensible Hyper Text Markup Language), http://www.w3.org/TR/xhtml1/ [Accessed 3 March 2011]

[22] XML (Extensible Markup Language), http://www.w3.org/TR/xml/ [Accessed 7 March 2011]

[23] CSS (Cascading Style Sheets), http://www.w3.org/TR/CSS2/ [Accessed 3 December 2010]

[24] CSS (Cascading Style Sheets) web tutorial, Available from: http://www.w3schools.com/css/css_intro.asp [Accessed 3 December 2010]

[25] w3schools.com, http://www.w3schools.com/js/js_intro.asp [Accessed 2 November 2010]

[26] Internet Explorer, http://www.microsoft.com/windows/internet-explorer/ [Accessed 2 November 2010]

[27] Mozilla Firefox, http://www.mozilla.com/en-US/firefox/personal.html [Accessed 2 November 2010]

[28] Google Chrome, http://www.google.com/chrome/intl/en/ landing_chrome.html?hl=en [Accessed 2 November 2010]

[29] Opera, http://www.opera.com/docs/specs/ [Accessed 15 November 2010]

[30] Safari, http://www.apple.com/safari/ [Accessed 15 November 2010]

[31] Scheme, http://www.schemers.org/Documents/Standards/R5RS/ [Accessed 20 March 2011]

[32] OCaml, http://dries.ulyssis.org/apt/packages/ocaml/ocaml-spec.html [Accessed 20 March 2011]

[33] JavaScript, https://developer.mozilla.org/en/JavaScript_Language_Resources [Accessed 1 November 2010]

[34] Marquez L. and Gimenez J., *"Automatic Evaluation in Machine Translation Towards Similarity Measures Based on Multiple Linguistic Layers"*, Presented in MOLTO workshop: *GF* meets SMT, Gothenburg, November 5, 2010. Available from: http://www.molto-project.eu/sites/default/files/MT-evaluation-seminar.pdf [Accessed 2 December 2010]

[35] Mastermind, Available from: http://en.wikipedia.org/wiki/Mastermind_(board_game) [Accessed 2 November 2010]

[36] Minibar Documentation. Available from: http://www.grammaticalframework.org:41296/minibar/about.html [Accessed 2 November 2010]

[37] GF online editor for simple multilingual grammars, http://www.grammaticalframework.org/demos/gfse/about.html [Accessed 20 February 2011]