UNIVERSITY OF GOTHENBURG

# Grammar-based suggestion engine with keyword search

*Master of Science Thesis in Computer Science*

MARTIN AGFJORD

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, August 2014

**Grammar-based suggestion engine with keyword search**

Examiner: KRASIMIR ANGELOV

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2014

# ABSTRACT

In this thesis we investigate how we can develop an application which can translate sentences formulated in natural languages (English and Swedish) into a query language. We also build a suggestion engine which offers suggestions to a user based on a partial or invalid sentence. The purpose of the suggestion engine is to help the user to find valid sentences that the application can translate.

We implement the translation by using a computational grammar. The grammar is developed by using *Grammatical Framework (GF)*, which is a development platform for building natural language grammars. We take two approaches on building the natural language parts of the grammar. The first is concatenation of strings and the second is by using the *GF Resource Grammar Library*. The query part is implemented with concatenation of strings.

The results show that it is more suitable to develop the natural language parts of the grammar by concatenating strings but only if the developer has good knowledge of the natural language. By concatenating strings, we can map all sorts of ungrammatical sentences to a grammatical sentence which is not possible with the GF Resource grammar library. This mapping makes the suggestion engine more powerful.

**Keywords:** Grammar, Grammatical Framework, GF, Natural language, Query language, Translation, Suggestion engine, Apache, Solr, Lucene, Tomcat, Maven, Java EE, Functional programming

A demo of the application and the source code can be found at thesis.agfjord.se

*We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better.*

— Donald E. Knuth [1]

# ACKNOWLEDGMENTS

I want to thank my supervisor Krasimir Angelov for his guidance throughout the whole project. He has always showed great interest in my work from the very beginning until the end.

I also want to thank the whole Findwise organization, where I did most of the programming work. In particular, a special thanks to Svetoslav Marinov who came up with the project idea. Also a special thanks to Per Fredelius who was my advisor at Findwise. Per inspired me with many ideas to the project.

As this thesis ends the academic part of my life, I also want to thank other people who have helped me during my time as a student.

I want to thank my friends in Monaden[1] and in Hilbert[2]. Their company and help have been invaluable.

Finally, I want to thank my family and my girlfriend Nellie for always supporting and believing in me. I wouldn't be where I am now if it weren't for them.

**Thank you!**

—

Martin Agfjord
Gothenburg, August 26, 2014

[1] The lunchroom for computer science students at University of Gothenburg
[2] The lunchroom for physics students at University of Gothenburg

# CONTENTS

# ACRONYMS

| | |
|---|---|
| GF | Grammatical Framework |
| RGL | Resource Grammar Library |
| Java EE | Java Enterprise Edition |
| PGF | Portable Grammar Format |

This page is intentionally left (almost) blank.

# 1 INTRODUCTION

## 1.1 A DEMAND FOR A NEW USER INTERFACE

It is complex for an average person to retrieve data by using query languages. Many applications make use of specifically designed graphical elements in order to facilitate for the end user to create queries. However, as data on the web is constantly growing it is increasingly harder to design such elements to cover the whole data set [2][p. 5].

Another approach of designing a user interface is by allowing the user to formulate instructions in a natural language. There exists evidence that this type of user interface is more satisfactory by end users than the traditional approach [3].

*A natural language is a language that humans use to communicate with each other.*

The beauty of writing instructions in a natural language is that there is no limitation of how a user can express herself, assuming that the machine which interprets the natural language instructions can extract the semantics from the instructions she writes.

## 1.2 A NATURAL LANGUAGE INTERFACE

We will in this thesis investigate how we can create a user interface which allows us to execute queries in a query language by expressing instructions in a natural language. In other words, we investigate how we can translate from a natural language into a query language.

*A query language is a computer language which is used to query a database or index.*

## 1.3 PROBLEM DESCRIPTION

How can one retrieve information from a computer by writing instructions in a natural language? The inspiration for this thesis came from Facebook graph search[1], which is a service that allows users to search for entities by asking Facebook's social graph for information in a natural language [4].

In this project, we have chosen to examine how a similar service can be realized. We have limited the project to handle instructions that can occur naturally in the intranet of a software development company. We assume that there exists a database with information about employees, customers and projects. A typical instruction in this environment could be

[1] https://www.facebook.com/about/graphsearch

```
people who know Java
```

The answer would be a list of all employees in the database who have some degree of expertise of the programming language Java. However, when using search engines, expert users do not use instructions as the one above. They simply rely purely on keywords [5]. The following instruction is more suited for expert users

```
people java
```

How can we create a user interface that is sufficient for both regular and expert users? How can we translate these instructions into machine readable queries?

### 1.4 A PROPOSED SOLUTION

Query languages require precise syntax, we therefore need precise translation from a natural language into a query language. Since we have a limited scope of instructions, we know all instructions that the program shall support and we know how their machine readable representation shall look like. We only need a tool which we can use to make the mapping between natural language and query language.

We will in this thesis use a computational grammar to extract the semantics from a natural language sentence. We are then going to use the semantics to produce a query string in a query language.

There exists different grammar formalisms where attribute grammars [6] and context-free grammars [7][pp. 77-106] (along with the Backus Naur Form (BNF) [8] notation) are the two most well-known. These two are mostly used for formally describing programming languages.

In this thesis, we will use the *Grammatical Framework (GF)* which is another grammar formalism [9] based on Martin-Löf's type theory [10]. GF is specifically designed for building grammars for natural languages.

A grammar defined by GF is a set of structural rules which decide how words can be created and be combined into clauses and phrases. By expressing how words can be combined into an instruction in one language one can also use the same logic to express how the same instruction can be produced in another language. A *multilingual* grammar is a special type of grammar which can translate between two or more languages. We will describe GF more in detail in Section 1.5.

## 1.5 RELATED WORK

This section presents two important projects that this project has been based on.

### 1.5.1 Facebook graph search

Facebook graph search [4] is a search engine which consists of a user interface where the user can formulate an instruction in a natural language as a string. The semantics of a natural language instruction is extracted while parsing the string.

The natural language that can be understood by Facebook Graph Search is represented by a weighted context free grammar (WCFG) [11]. The grammar consists of a set of production rules which are used to extract one or more semantic parse trees from a natural language sentence. The parse tree(s) represent the *meaning* of the sentence in a semantic way. This tree can be sent to Unicorn, which is a software for retrieving information from Facebook's social graph [4].

#### Entity recognition

Facebook's grammar also supports entity recognition, which means that the grammar tries to find the suitable *type* of a word if it thinks it represents an object in the social graph. For example, if the user types *people who live in San Fransisco* then the grammar can with high confidence express that *San Fransisco* is an object of the type *Location*. This is achieved by using n-gram based language models in order to obtain the type with the highest probability.

#### Lexical analysis

Synonyms are supported by the grammar. Synonyms could be words or phrases. For example the phrase *people who like surfing* has the synonyms *people who surf* and *surfers*. They are defined to have equivalent semantics.

Since computers normally only accept perfectly correct input when dealing with machine instructions, Facebook have added support of grammatically incorrect sentences to the grammar. It can therefore map the sentence *people who **works** at facebook* into *people who work at Facebook*.

### 1.5.2 Grammatical Framework

Natural languages contain a lot ambiguities and can often differ a lot on a linguistic level. Those properties makes it very hard and exhausting to develop accurate natural language interpreters [3]. In order to make use of previous research in the field, we will make us of Grammatical Framework (GF), which is an open source functional programming language for creating grammars that can interpret natural languages [12, p. 1]. GF features a strong type system, it adopts abstract and concrete syntax rules and it offers reusable libraries to facilitate design of grammars [13]. For a reader with a background within compilers, one can see that GF is very much based on the theory of programming languages as they also make use of abstract and concrete syntaxes [14, pp. 69-70].

Abstract syntax is a tree representation which captures the *meaning* (i.e. the semantics) of a sentence, and leaves out anything irrelevant. The concrete syntax describes how an abstract syntax tree is represented as a string in a language.

When designing abstract and concrete syntaxes one make us of functions. The functions are defined in the abstract syntax and designs how the a tree can be built by combining values from the functions. The concrete syntax purpose is to add rules to the functions which are used to extract the semantics of strings to build abstract syntax trees. Conversely, if one has an abstract syntax tree, one can use the functions to create a sentence.

With both abstract and concrete syntaxes, GF is able to create a *parser* and a *linearizer* for all given concrete languages. The parser translates a string into abstract syntax trees and the linearizer translates abstract syntax trees into a string representations for a specified concrete syntax. In addition, GF also offers a *generator* for abstract syntax trees that can generate all possible abstract syntax trees.

Because GF separates between abstract and concrete syntax, one can easily add a new concrete syntax (a new language) to an existing abstract syntax. This advantage makes it easy to parse a string in one language and obtain an abstract syntax tree which can be linearized into many concrete syntaxes.

GF's translation approach is different from previous translation approaches and allows translation between languages that are not closely related from a structural point of view [15][p. 9].

# 2 A SIMPLE GRAMMAR

This chapter presents an example of how GF can be used to create a grammar that can translate the sentence *people who know Java* into Apache Solr query language and vice verca. Apache Solr is a search platform based on Apache Lucene [16].

## 2.1 ABSTRACT SYNTAX

To model the meaning of a sentence, GF adopts the use of functions and *categories*. A category (`cat`) in GF is the same as a data type. We start by listing the categories we need, as seen in Figure 1 on lines 3-7. We then define the values that our data types can take. This is achieved by using functions. The functions in an abstract syntax are usually not implemented, we can therefore only see the function declarations. The reason is because we only want to model the semantics at the abstract level. How the semantics are implemented in a specific language is irrelevant, because we want to keep the abstract syntax as language independent as possible in order to make it easier to develop concrete syntaxes.

We define a function `Java : Object` which means that `Java` is a constant and returns a value of type `Object`. `Know` takes one argument of the type `Object` and returns a value of type `Relation`.

*A function without arguments is called a constant in lazy functional programming languages.*

An instruction can be created by obtaining a value of the type `Instruction`. Only `MkInstruction` returns the desired type and it takes two arguments, one of type `Subject` the other of type `Relation`.

```
abstract Instrucs = {
  flags startcat = Instruction;
  cat
    Instruction ; -- An Instruction
    Subject ;     -- The subject of an instruction
    Relation ;    -- A verb phrase
    Object ;      -- an object

  fun
    MkInstruction : Subject -> Relation -> Instruction ;
    People : Subject ;
    Know : Object -> Relation ;
    Java : Object ;
}
```

Figure 1: Abstract syntax

## 2.2 CONCRETE SYNTAX

We are now going to implement the linearizations of the function declarations we just defined in the abstract syntax. This implementation makes it possible to linearize abstract syntax trees into concrete syntax. We will start by defining the concrete syntax for English.

### 2.2.1 *English concrete syntax*

Figure 2 shows the implementation of the concrete syntax for English. Categories are linearized by the keyword `lincat`, which literally means the linearization of categories. A category is linearized by assigning a data type to it. Here we assign all categories to be of the type string, which means that all our linearization functions also must return a string. The functions are linearized by using the keyword `lin`. We linearize `Java` by returning the string `"Java"`, as it is a constant function. Analogously, `"people"` is returned by `People`. The function `Know` takes one parameter. This parameter is appended on the string `"know"`. Finally, `MkInstruction` takes two arguments, where `subject` is prepended and `relation` is appended on `"who"`. One can easily see how these functions can be used to construct the sentence *people who know Java*.

```
concrete InstrucsEng of Instrucs = {
  lincat
    Instruction = Str ;
    Subject = Str ;
    Relation = Str ;
    Object = Str ;
  lin
    MkInstruction subject relation = subject ++ "who" ++ relation ;
    People = "people" ;
    Know object = "know" ++ object ;
    Java = "Java" ;
}
```

Figure 2: English concrete syntax

### 2.2.2 *Solr concrete syntax*

The final step in this example is to linearize the same abstract syntax into Solr concrete syntax. As Figure 3 shows, the categories are strings as in English. The function linearizations are however different. `People` returns `"object_type : Person"`, we assume that the Solr-schema has a field with the name `object_type` which represents the type of a document. Similarly, we make another assumption about `Know`. `MkInstruction` is also implemented differently, here we can see that the result is going to be a query string by looking at the first part `"q="` which is prepended on the subject. We then append `"AND"` together with `relation` in order to create a valid Solr query.

*A query string is a part of a URL, e.g. foo.com?**q=name***

```
concrete InstrucsSolr of Instrucs = {
    lincat
      Instruction = Str ;
      Subject = Str ;
      Relation = Str ;
      Object = Str ;

    lin
      MkInstruction subject relation = "q=" ++ subject ++ "AND" ++ relation ;
      People = "object_type : Person" ;
      Know object = "expertise : " ++ object ;
      Java = "Java" ;
}
```

Figure 3: Solr concrete syntax

## 2.3 translation

In order to make any translations, we need to use the GF runtime system. The runtime system we will use in this section is the shell application, which allows us to load our GF source code and use parsers, linearizers and generators. In addition to the shell application, there also exists programming libraries for GF in C, Haskell, Java and Python. These libraries can be used to build a translation application which does not require the user to have GF installed.

```
$ gf InstrucsEng.gf InstrucsSolr.gf

       *  *  *
    *           *
  *               *
 *
 *
 *        * * * * * *
 *        *         *
  *       * * * *  *
    *     *      *
       *  *  *

This is GF version 3.5.12-darcs.
No detailed version info available
Built on linux/x86_64 with ghc-7.6, flags: interrupt server
License: see help -license.
Bug reports: http://code.google.com/p/grammatical-framework/issues/list
 - compiling Instrucs.gf...   write file Instrucs.gfo
- compiling InstrucsEng.gf...   write file InstrucsEng.gfo
- compiling InstrucsSolr.gf...   write file InstrucsSolr.gfo
linking ... OK
 Languages: InstrucsEng InstrucsSolr
Instrucs>
```

Figure 4: GF shell prompt

A string can be parsed into an abstract syntax tree.

```
Instrucs> parse -lang=InstrucsEng "people who know Java"
MkInstruction People (Know Java)
```

Figure 5: Parse a string

Abstract syntax trees can be linearized into concrete syntaxes, here we linearize one abstract syntax tree into all known concrete syntaxes.

```
Instrucs> linearize MkInstruction People (Know Java)
people who know Java
q= object_type : Person AND expertise : Java
```

Figure 6: Linearize an abstract syntax tree

Finally, a string can be translated from one concrete syntax into another. Here we translate from `InstrucsEng` into `InstrucsSolr`. We use a *pipeline* to pass the result of the parsing as an argument to the linearizing function. Note how we use `p` instead of `parse` and `l` instead of `linearize`. They are just shorthands of their longer representations.

```
Instrucs> p -lang=InstrucsEng "people who know Java" | l -lang=InstrucsSolr
q= object_type : Person AND expertise : Java
```

Figure 7: Translate between concrete syntaxes

## 2.4 GF RESOURCE GRAMMAR LIBRARY

The previous example is fairly easy to understand, but it also make use of English, a well-known natural language. It is much harder to create a concrete syntax that implements a lesser known natural language by using concatenation. Even though a user might know correct translation of the individual words to use, she might not know how to use them in a grammatically correct sentence. It is often the case that if one directly translates a sentence, i.e. just translate each word by word, one ends up in a grammatically incorrect sentence.

The GF Resource Grammar Library (RGL) [13] is a set of grammars which implements the morphology and basic syntax of currently 29 languages [17]. In other words, it contains functions and categories which describes the structure of natural languages. One can therefore create values of specific types from the words of a sentence and then combine the words by using functions in order to create a grammatically correct phrase or sentence. We say that a developer does only need to have knowledge of her *domain*, i.e. the individual words to use, and does not have any specific linguistic knowledge of the natural language.

### *Example usage of the RGL in a grammar*

In this section, we will present how the previous concrete syntax for English can be implemented by using the RGL. We will also show how this grammar can be further generalized into an incomplete concrete syntax which can be used by both English and Swedish.

Figure 8 shows the concrete syntax for English by using the RGL. The categories are now set to be types that exists in the RGL and the functions are now using RGL-functions in order to create values of the correct types.

The most simple function in this case is `People`, which shall return a noun. A noun can be created by using the *operation* `mkN`. We create a noun which has the singular form `"person"` and plural form `"people"`, we will never use the singular form in this grammar, but it will become handy later in the thesis to use both singular and plural forms.

*An operation in GF is a function which can be called by linearization functions.*

`Java` returns a `noun phrase` which is created by the function `mkNP`, however, we create a noun phrase by converting a proper name which is initialized as *Java*. `Know` returns a relative sentence. A relative sentence can for example be *who know Java*. A relative sentence is constructed by first creating a *verb phrase* from a verb and an object. This verb phrase is then used together with the constant `which_RP` to create a *relative clause*. Finally, we convert the relative clause into a relative sentence. This is achieved by using a self made operation named `mkRS'`, the purpose of this operation is to make the code easier to read and also in the future reuse code.

```
concrete InstrucsEng of Instrucs = open SyntaxEng, ParadigmsEng in {
  lincat
    Instruction = Utt ;
    Subject = N ;
    Relation = RS ;
    Object = NP ;

  lin
    MkInstruction subject relation = mkUtt
                                      (mkNP aPl_Det (mkCN subject relation)) ;
    People = mkN "person" "people" ;
    Know object = mkRS' (mkVP (mkV2 (mkV "know") object)) ;
    Java = mkNP (mkPN "Java") ;

   oper
      mkRS' : VP -> RS = \vp -> mkRS (mkRCl which_RP vp) ;
}
```

Figure 8: English concrete syntax using the RGL

The only thing that is left is to combine a noun with a relative sentence, e.g. combine *people* with *who know Java*. This is done by using the operation `mkCN` to create a common noun. As common nouns do not have any determiners, we have to construct a noun phrase together with the determiner `aPl_Det` in order to select only the plural forms. Lastly we convert the noun phrase into an utterance in order to only allow the nominative form of the sentence (we would otherwise end up with with multiple equal abstract syntax trees when parsing a sentence).

## 2.5 GENERALIZING THE CONCRETE SYNTAX

This section describes how the concrete syntax can be generalized into an *incomplete concerete syntax* and then be instantiated by two concrete syntaxes, one for English and one for Swedish.

### *An incomplete concrete syntax*

As we already have designed the concrete syntax for English, we can fairly easy convert it to a generalised version. The incomplete concrete syntax can be seen in Figure 9. We no longer have any strings defined, as we want to keep the syntax generalised. Constants are used in place of strings, and they are imported from the lexicon interface `LexInstrucs`.

```
incomplete concrete InstrucsI of Instrucs = open Syntax, LexInstrucs in {
  lincat
    Instruction = Utt ;
    Subject = N ;
    Relation = RS ;
    Object = NP ;

  lin
    MkInstruction subject relation = mkUtt
                                      (mkNP aPl_Det (mkCN subject relation)) ;
    People = person_N ;
    Know object = mkRS' (mkVP know_V2 object) ;
    Java = java_NP ;

   oper
      mkRS' : VP -> RS = \vp -> mkRS (mkRCl which_RP vp) ;
}
```

Figure 9: Incomplete concrete syntax

`LexInstrucs` is an *interface*, which means that it only provides declarations. Figure 10 shows that we have one operation declaration for each word we want to use in the concrete syntax. Because we do not implement the operations, it is possible to create multiple instances of the lexicon where each one can implement the lexicon differently.

```
interface LexInstrucs = open Syntax in {
   oper
     person_N : N ;
     know_V2  : V2 ;
     java_NP  : NP ;
}
```

Figure 10: Lexicon interface

Figure 11 shows how the operations defined in `LexInstrucs` are implemented in `LexInstrucsEng`. We represent the words in the same way as in the old version of the concrete syntax for English.

```
instance LexInstrucsEng of LexInstrucs = open SyntaxEng, ParadigmsEng in {
    oper
      person_N = mkN "person" "people" ;
      know_V2 = mkV2 (mkV "know") ;
      java_NP = mkNP (mkPN "Java");
}
```

Figure 11: Lexicon instantiation of English

Figure 12 shows another instance of `LexInstrucs`, the lexicon for Swedish.

*The definition of* `know_V2` *is taken from StructuralSwe.gf in the RGL*

```
instance LexInstrucsSwe of LexInstrucs = open SyntaxSwe, ParadigmsSwe in {
    oper
      person_N = mkN "person" "personer" ;
      know_V2 = mkV2 (mkV "kunna" "kan" "kunn" "kunde" "kunnat" "kunnen") ;
      java_NP = mkNP (mkPN "Java");
}
```

Figure 12: Lexicon instantiation of Swedish

We are now ready to instantiate the incomplete concrete syntax. The code below describes how `InstrucsI` is instantiated as `InstrucsEng`. Note how we override `Syntax` with `SyntaxEng` and `LexInstrucs` with `LexInstrucsEng`.

```
concrete InstrucsEng of Instrucs = InstrucsI with
                                         (Syntax = SyntaxEng),
                                         (LexInstrucs = LexInstrucsEng)
                                         ** open ParadigmsEng in {}
```

Figure 13: English instantiation of the incomplete concrete syntax

Analogously, we create an instance for Swedish concrete syntax by instantiating InstrucsI and overriding with different files.

```
concrete InstrucsSwe of Instrucs = InstrucsI with
                                             (Syntax = SyntaxSwe),
                                             (LexInstrucs = LexInstrucsSwe)
                                             ** open ParadigmsSwe in {}
```

Figure 14: Swedish instantiation of the incomplete concrete syntax

If we load the GF-shell with InstrucsEng.gf and InstrucsSwe.gf we can make the following translation from English to Swedish.

```
Instrucs> p -lang=InstrucsEng "people who know Java" | l -lang=InstrucsSwe
personer som kan Java
```

Figure 15: Swedish instantiation of the incomplete concrete syntax

Whats really interesting is that we now can go from both English and Swedish into abstract syntax, and by extension, also to Solr-syntax.

# 3

# APPLICATION DEVELOPMENT

## 3.1 BRIEF DESCRIPTION OF THE APPLICATION

We begin by describing what the different parts of the application and the why we need them.

### 3.1.1 *Generation of mock data*

As described in Section 1.3, we want to develop an application which can translate natural language questions that refers to entities in a database or index owned by a software development company. This project has been made with strong collaboration with Findwise, a company with focus on search driven solutions. Findwise has an index with information about employees, projects and customers. However, it is not possible to use their information because it is confidential and cannot be published in a master thesis. A different approach to get hold of relevant data is to generate mock data that is inspired by Findwise's data model. Mock data in this project is simply generated data from files that can be used to simulate a real world example application.

### 3.1.2 *Grammar development*

The grammar in Chapter 2 can only translate the instruction *people who know Java* in English and Swedish into Solr query language. The grammar needs to be extended to handle *any* programming language that exists in the mock data, not only *Java*. The grammar also needs to support more instructions in order to make a more realistic use case. We have chosen to support the following instructions:

```
English                         Solr query language
people who know Java            q= object_type : Person AND KNOWS : Java
people who work in London       q= object_type : Person AND WORKS_IN : London
people who work with Unicef     q= object_type : Person AND WORKS_WITH : Unicef
customers who use Solr          q= object_type : Customer AND USES : Solr
projects who use Solr           q= object_type : Project AND USES : Solr
```

Figure 16: All sentences supported by the application

Two more cases have been added to instructions regarding *people*. In addition, two new type of instructions has been added, translations about *customers* and *projects*. Note that Figure 16 only shows specific instances of instructions. These instances are based on data in the mock-database, where we assume that the words *Java*, *London*, *Unicef* and *Solr* exists in the database. It should be possible to express every word available in the database in an instruction. For example if *Paris* is a city in the database we can create the instruction *people who work in Paris*.

### *3.1.3 Suggestions*

If a user has no idea of which instructions the application can translate, how can she use the application? This thesis uses a narrow application grammar, which means that it only covers specified sentences. Therefore, if a sentence has one character in the wrong place, GF will not be able to translate anything. This problem can be solved by using a wide coverage grammar, an example of an application that adopts this technique is the GF android app [18, p. 41]. This project however, does not use this technique due to lack of documentation of how it is accomplished.

GF has the power to suggest valid words of an incomplete sentence by using incremental parsing [19]. This means that even though a user do not know what to type, the application can suggest valid words to use. If the user chooses to add one of those words, the suggestion engine can show a new list of words that will match the new partial sentence.

However, this method does not support the use of only keywords, since one cannot start a sentence with for example the word *Java*. It is also inflexible as it does not support the use of words outside the grammar. For example, in the sentence `all people who know Java`, the parser would not be able to parse the word *all*.

This thesis takes a different approach on a suggestion engine. Instead of suggesting one word at a time, one can suggest a whole sentence based on what the user has typed so far. This is achieved by generating all possible sentences that the application can translate (up to a certain size) and index them in Apache Solr. This makes it very easy to search on matching sentences, we also gain powerful techniques such as approximate string matching [20][p. 22].

By using this approach, if a user types a sentence in the application, it will search in the index on instructions related to the string and will retrieve the most relevant instructions.

As the suggestion engine uses a search platform, it is possible to type any word that exists in a sentence and get suggestions, even only keywords like `'people java'` will suggest instructions that can be formulated with these two words.

### 3.1.4 *Runtime environment*

The chosen programming language for this project is Java. The main reason is because it is Findwise's primary programming language. It is also very well known among many companies in the world. Many professional Java-developers adopt a specific development platform, Java Enterprise Edition (Java EE) [21]. This platform provides many libraries that can be scaled to work in an enterprise environment. This project also adopts Java EE.

### 3.1.5 *Handling dependencies*

A typical Java EE project makes use of several libraries, in computer science terms we say that a project can have other libraries as *dependencies*. It is not unusual that these libraries also have their own dependencies. Larger projects can therefore have a lot of dependencies, so many that it becomes hard to keep track of them. This project makes use of an open source tool called Apache Maven [22] to handle dependencies. One simply lists all libraries the project shall have access to, then Maven will automatically fetch them and their dependencies. This also makes the application more flexible, as it do not have to include the needed libraries in the application.

### 3.1.6 *Input and output presentation*

Besides handling translation and suggestions, the application also needs to handle input and present its results in some way. This application takes input and presents output by using a web graphical user interface (web gui).

### 3.1.7 *Information flow*

This section aims to describe how the information flows in the application when a user starts to type words in the input field and then obtains relevant data.

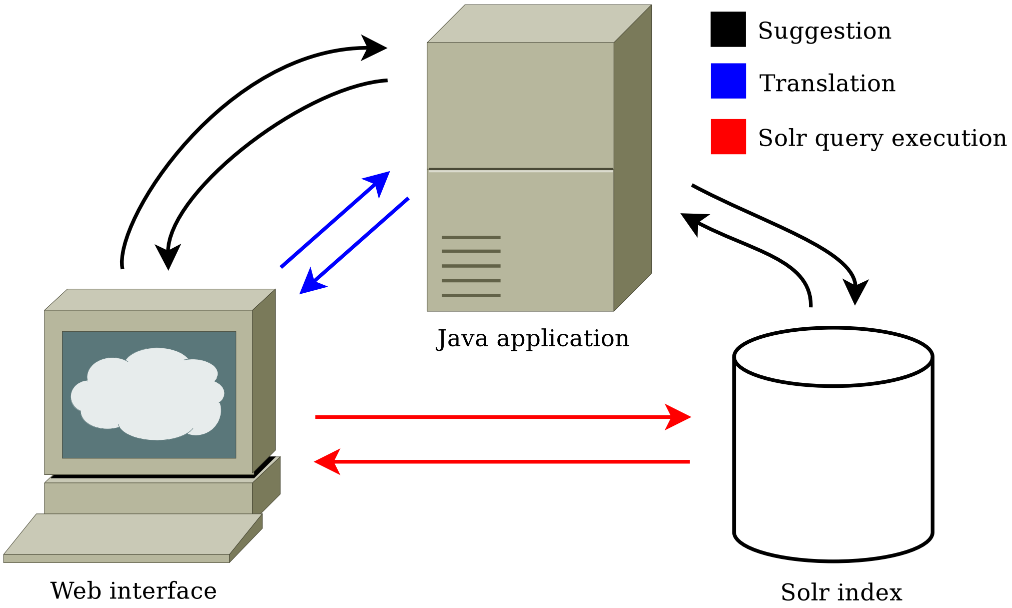


Figure 17: Information flow in the application

A JavaScript-listener is active on the input field. When a user types letters into the input field, the listener will send the input to a the Java-application which will run the suggestion algorithm on the input. The suggestion algorithm is making multiple requests to the Solr index in order to obtain suggestions. The suggestions are sent back to the user and are presented by using JavaScript.

If a user chooses to translate a sentence (by pushing enter while focusing on the input field), another JavaScript function will be executed which sends the phrase or sentence to the Java application. This sentence is parsed into one or multiple abstract syntax trees by using PGF [23][p. 14]. The resulting abstract syntax tree(s) are linearized into all possible concrete syntaxes and returned to the web application. The web application present the result to the user.

In addition to presenting the result, it also creates a hyperlink of each Solr linearization. This hyperlink leads to the Solr index and is an executable version of the Solr linearization.

### 3.1.8 *Running the application*

Web applications built in Java usually have the WAR file format. It is a special JAR-file which includes classes, dependencies and web pages. This project uses an open source web server called Apache Tomcat [24] to host a web application by exporting our application as a WAR-file. Apache Tomcat will make the application available by using HTTP-requests and spawn a new thread for each request.

Details about the runtime environment can be found in Appendix A and Appendix B.

## 3.2 GRAMMAR DEVELOPMENT WITH THE RGL

This section continues the work on the grammar introduced in Chapter 2.

### 3.2.1 *Supported instructions*

The example grammar could only translate one instruction. This instruction in English is *people who know Java*. It is easy to extend this grammar to support more programming languages, for example, to support *Python* one can add a function `Python : Object` in the abstract syntax and implement it as `Python = "python"` in the concrete syntaxes. However, this approach makes the grammar inflexible because we need to extend the grammar every time we want to add a new programming language.

### 3.2.2 *Names*

Defining a new function for each programming language is not a good idea, as described in the previous section. A better solution would be to make one function that could be used by any programming language.

One intuitive approach to solve this problem is to create a function `MkObject : String -> Object`. The implementation for this function would be

```
-- Abstract syntax
MkObject : String -> Object ;
-- RGL implementation
MkObject str = mkNP (mkPN str.s) ; -- PN = Proper Name
-- Solr implementation
MkObject str = str.s ;
```

Figure 18: Intuitive approach on names

The GF-code compiles, and the parsing and linearization by using Solr query language works. Unfortunately, this approach does not work with the RGL, because the `mkPN`-operation directly tries to manipulate the string which of course cannot be done when it is arbitrary.

Fortunately, there exists a built in category which can be used for exactly these situations. We use the category `Symb`, along with the function `MkSymb :`

`String -> Symb` to represent arbitrary strings. We can then use the function `SymbPN` to create a proper name and finally create a noun phrase.

```
-- Abstract syntax
MkObject : Symb -> Object ;
-- RGL implementation
MkObject symb = mkNP (SymbPN symb) ; -- PN = Proper Name
-- Solr implementation
MkObject symb = symb.s ; -- Symb has the type { s : Str }
```

Figure 19: Working approach on names

By using this solution, we can translate the sentence *people who know Foo*, where *Foo* can be anything.

### 3.2.2 *Recognition of names*

We said that *Foo* can be anything in the previous example, but we can also replace it with *Foo Bar Baz...* and the words could continue forever as long as the first letter in each word is in uppercase. This is how the Java runtime for GF recognizes names by default, but one can also use a customized definition.

### 3.2.3 *Extending the grammar*

It is not trivial to extend the grammar to support the instructions described in Section 3.1. One has to take into account that it shall not be possible to translate invalid instructions like *projects who work in London*. We will in this section first extend the abstract syntax to support the new instructions and then extend the concrete syntaxes.

#### *Abstract syntax*

We begin by removing the category `Subject` and replacing it with three new categories: `Internal`, `External` and `Resource`. The function `People` will return a value of the type `Internal` and `Customer` and `Project` will return values of the types `External` and `Resource` respectively.

```
-- Instructions.gf
cat
  Internal ;
  External ;
  Resource ;
fun
  People   : Internal ;
  Customer : External ;
  Project  : Resource ;
```

Figure 20: Abstract syntax with new categories and functions for subjects

In addition to adding new subject categories, three new categories for relations are also introduced: `InternalRelation`, `ExternalRelation` and `ResourceRelation` (`Relation` is removed). The idea is to link subject types to the correct relation types. For instance, we link a value of the type `Internal` with a value of type `InternalRelation`.

All relation functions are changed to return the correct type. For example, `Know` is changed to return a value of the type `InternalRelation`. This means that only `People` can be used together with `Know`, as desired. The new relation implementations can be seen in Figure 21.

```
cat
  InternalRelation ;
  ExternalRelation ;
  ResourceRelation ;
fun
  Know     : Object -> InternalRelation ;
  UseExt   : Object -> ExternalRelation ;
  UseRes   : Object -> ResourceRelation ;
  WorkIn   : Object -> InternalRelation ;
  WorkWith : Object -> InternalRelation ;
```

Figure 21: Abstract syntax with new categories and functions for relations

The last thing to modify is how subjects and relations are combined. In Figure 22, the function `MkInstruction` is replaced by three new functions: `InstrucInternal`, `InstrucExternal` and `InstrucResource`. However, as we do not need to make a distinction between different type of instructions at this level, all three functions returns a value of the type `Instruction`.

```
cat
  Instruction ;
fun
  InstrucInternal : Internal -> InternalRelation -> Instruction ;
  InstrucExternal : External -> ExternalRelation -> Instruction ;
  InstrucResource : Resource -> ResourceRelation -> Instruction ;
```

Figure 22: Abstract syntax with new categories and functions for instructions

### 3.2.3 *Concrete syntax using the RGL*

As the abstract syntax has changed, the concrete syntaxes have to be modifed as well. This section explains how the generalised concrete syntax which uses the RGL is implemeneted.

Figure 23 shows how the categories has been implemented. The new categories are implemented in the same way as the previous.

```
lincat
  Instruction = Utt ;
  Internal, External, Resource = N ;
  InternalRelation, ExternalRelation, ResourceRelation = VP ;
```

Figure 23: Concrete syntax using the RGL with new category implementations

The new subject functions are implemented in the same way as `People`.

```
lin
  People = person_N ;
  Customer = customer_N ;
  Project = project_N ;
```

Figure 24: RGL concrete syntax with new subject implementations

Four new relation functions are added. Line 5-6 in Figure 25 shows how we use the verb `work_V` together with two prepositions, `in_Prep` and `with_Prep` in order correctly linearize into *work in Foo* and *work with Foo* respectively (*Foo* is the value of `object`. The relation implementations make use of an operation `mkRS'` to reuse code.

```
lin
  Know object = mkRS’ (mkVP know_V2 object) ;
  UseExt object = mkRS’ (mkVP use_V2 object) ;
  UseRes object = mkRS’ (mkVP use_V2 object) ;
  WorkIn object = mkRS’ (mkVP (mkV2 work_V in_Prep) object) ;
  WorkWith object = mkRS’ (mkVP (mkV2 work_V with_Prep) object) ;

oper
  -- Make a relative sentence
  mkRS’ : VP -> RS = \vp -> mkRS (mkRCl which_RP vp) ;
```

Figure 25: Concrete syntax using the RGL with new relation implementations

Subjects and relations are combined as before, but as this solution has three functions instead of one, a new operation mkI has been defined in order to reuse code.

```
lin
  InstrucInternal internal relation = mkI internal relation ;
  InstrucExternal external relation = mkI external relation ;
  InstrucResource resource’ relation = mkI resource’ relation ;

oper
  mkI : N -> RS -> Utt = \noun,rs -> mkUtt (mkNP aPl_Det (mkCN noun rs)) ;
```

Figure 26: Concrete syntax using the RGL with new instruction implementations

### *Concrete syntax for Solr*

This section describes how the concrete syntax for Solr is modified to work with the new abstract syntax.

The new categories are all defined as strings.

```
lincat
  Instruction = Str ;
  Internal, External, Resource = Str ;
  InternalRelation, ExternalRelation, ResourceRelation = Str ;
  Object = Str ;
```

Figure 27: Solr concrete syntax with new implementation of categories

Subject types are hard coded into strings. We assume that these strings exists in the Solr index.

```
lin
  People = "Person" ;
  Customer = "Organization" ;
  Project = "Project" ;
```

Figure 28: Solr concrete syntax with new subject implementations

We also make an assumption about how the relations are defined in the Solr index.

```
lin
  Know obj = "KNOWS" ++ ":" ++ obj ;
  UseExt obj = "USES" ++ ":" ++ obj ;
  UseRes obj = "USES" ++ ":" ++ obj ;
  WorkWith obj = "WORKS_WITH" ++ ":" ++ obj ;
  WorkIn obj = "WORKS_IN" ++ ":" ++ obj ;
```

Figure 29: Solr concrete syntax with new relation implementations

As in the concrete syntax using the RGL, also an operation is defined and used by the three functions.

```
lin
  InstrucInternal internal relation = select internal relation ;
  InstrucExternal external relation = select external relation ;
  InstrucResource resource’ relation = select resource’ relation;

oper
  select : Str -> Str -> Str = \subj,relation ->
                  "select?q=*:*&wt=json&fq=" ++ "object_type :"
                  ++ subj ++ "AND" ++ relation ;
```

Figure 30: Solr concrete syntax with new instruction implementations

### 3.2.4 Boolean operators

The grammar is now powerful enough to translate a variety of questions. To make it even more powerful, one could use boolean operators in order to combine relations. For example, an instruction that could be useful is *people who know Java and Python*. Another useful instruction is *people who know Java and work in Gothenburg*. This section explains how the grammar can be extended to support these kind of instructions.

In addition to the previous example with the boolean operator *and*, we will also add support for the boolean operator *or*. We begin by adding functionality to support boolean operators to combine values of the type `Object`. As seen in

Figure 31, two new functions are defined in the abstract syntax to handle these cases, one for each operator.

```
fun
  And_O : Object -> Object -> Object ;
  Or_O : Object -> Object -> Object ;
```

Figure 31: Abstract syntax for boolean operators and objects

The RGL implementation is shown in Figure 32.

```
lin
  And_O o1 o2 = mkNP and_Conj o1 o2 ;
  Or_O o1 o2 = mkNP or_Conj o1 o2 ;
```

Figure 32: Concrete syntax using the RGL for boolean operators and objects

The Solr implementation is shown in Figure 33. We add `AND` or `OR` between the two objects.

```
lin
  And_O o1 o2 = "(" ++ o1 ++ "AND" ++ o2 ++ ")" ;
  Or_O o1 o2 = "(" ++ o1 ++ "OR" ++ o2 ++ ")" ;
```

Figure 33: Solr concrete syntax for boolean operators and objects

It is now possible to express *people who know Java and Python*. In order to use boolean operators with whole relations like *people who know Java and work in Gothenburg*, the grammar has to be further extended.

We must also take into account that it shall only be possible to combine relation that are possible to express in the current sentence. Therefore, we need to define the boolean logic three times, as we have three different types of instructions.

```
fun
  InternalAnd : InternalRelation -> InternalRelation -> InternalRelation ;
  InternalOr : InternalRelation -> InternalRelation -> InternalRelation ;

  ExternalAnd : ExternalRelation -> ExternalRelation -> ExternalRelation ;
  ExternalOr : ExternalRelation -> ExternalRelation -> ExternalRelation ;

  ResourceAnd : ResourceRelation -> ResourceRelation -> ResourceRelation ;
  ResourceOr : ResourceRelation -> ResourceRelation -> ResourceRelation ;
```

Figure 34: Abstract syntax for boolean operators and relations

Instead of combining noun phrases as in Figure 32, here we combine relative sentences in the RGL implementation.

```
lin
  InternalAnd rs1 rs2 = mkRS and_Conj rs1 rs2 ;
  InternalOr rs1 rs2 = mkRS or_Conj rs1 rs2 ;

  ExternalAnd rs1 rs2 = mkRS and_Conj rs1 rs2 ;
  ExternalAnd rs1 rs2 = mkRS or_Conj rs1 rs2 ;

  ResourceAnd rs1 rs2 = mkRS and_Conj rs1 rs2 ;
  ResourceOr rs1 rs2 = mkRS or_Conj rs1 rs2 ;
```

Figure 35: Concrete syntax using the RGL for boolean operators and relations

The Solr implementation is fairly straight forward, similarly as with values of the type `Object`, we also add `AND` or `OR` between the strings. The only difference is that we do it three times as we have three different subject types.

```
lin
  InternalAnd s1 s2 = "(" ++ s1 ++ "AND" ++ s2 ++ ")";
  InternalOr s1 s2 = "(" ++ s1 ++ " OR " ++ s2 ++ ")";

  ExternalAnd s1 s2 = "(" ++ s1 ++ "AND" ++ s2 ++ ")";
  ExternalOr s1 s2 = "(" ++ s1 ++ " OR " ++ s2 ++ ")";

  ResourceAnd s1 s2 = "(" ++ s1 ++ "AND" ++ s2 ++ ")";
  ResourceOr s1 s2 = "(" ++ s1 ++ " OR " ++ s2 ++ ")";
```

Figure 36: Solr concrete syntax for boolean operators and relations

### 3.2.4 *Boolean operators and ambiguity*

Our definition of boolean operations can create ambiguous instructions. An ambiguous instruction can be interpreted by the program in more than one way. For example *people who know Java or Python and Haskell* could be interpreted as `'people who know (Java or Python) and Haskell'` or it can be interpreted as `'people who know Java or (Python and Haskell)'`. GF automatically detects ambiguity and will return several abstract syntax trees while parsing an instruction which is ambiguous.

### 3.2.4 *Infinitely many instructions*

The boolean operators we just defined are recursively defined, because they take two arguments of a certain type, and they use those two arguments to create a value of the same type. This means that the resulting value can actually be used in the same function to create another value, and this goes on forever.

## 3.3 SUGGESTION ENGINE

The suggestion engine shall generate all possible instructions in all natural languages and store them in an Apache Solr index. It is suitable to use the *generator* which GF offers to generate abstract syntax trees and linearize them into the specified concrete languages (English and Swedish in this case).

The generator can be accessed through the GF-shell. Figure 37 shows how the function `generate_trees` is executed to generate all trees with the *depth* 4. By depth, we mean the maximum number edges between a leaf and the root element.

```
Instrucs> generate_trees
InstrucExternal Customer (UseExt (MkObject (MkSymb "Foo")))
InstrucInternal People (Know (MkObject (MkSymb "Foo")))
InstrucInternal People (WorkIn (MkObject (MkSymb "Foo")))
InstrucInternal People (WorkWith (MkObject (MkSymb "Foo")))
InstrucResource Project (UseRes (MkObject (MkSymb "Foo")))
```

Figure 37: `generate_trees` is used to create all abstract syntax trees with max depth 4

Figure 37 shows 5 trees, but there exists many more trees. The reason GF only generates 5 trees is because the default depth setting is 4. If we increase the depth we will obtain more trees, as it then will include trees containing boolean operators. By increasing the max depth to 5, GF will generate 36 trees. With depth 6, GF will generate 321 trees.

*Actually, there exists infinitely many trees, as we have recursive functions in the abstract syntax.*

It is often good to visualize trees to understand them better. Figure 38 shows two abstract syntax trees, the first one with depth 4 and the second with depth 5. One can easily see that the former has maximum 4 edges between root and leaf by counting the edges between `InstrucInternal` and `"Foo"`, and the latter has maximum 5 edges between root and leaf.

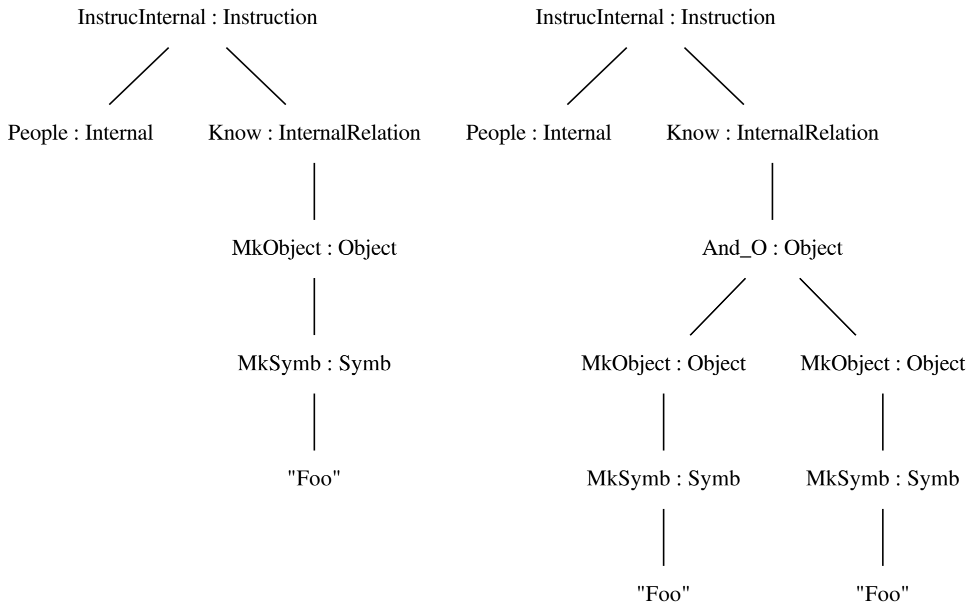


Figure 38: Visualization of abstract syntax trees with depth 4 and 5

### 3.3.1 *Populating the Solr index*

It is now time to linearize the generated trees and store them in a Solr index that is dedicated to store linearizations. By doing so, we will be able to search on instructions by using words that exists in the instructions. What we cannot do is to search on names, because the names does not occur in the instructions, instead they only contain a placeholder for a name (*"Foo"*). It will therefore only suggest instructions like *people who know Foo and Foo* which is a useless suggestion.

We want to be able to suggest relevant names. If the user database contains a person which knows Java, then we also want to suggest instructions based on that name. This requirement forces us to change the application once more.

A naïve solution would be to fetch all distinct names from the database and create all possible instructions that a user shall be able to express with these names. For an instruction like *people who know Foo or Foo and works in Foo* and if the database contains 10 programming languages and 5 cities we would then have to generate $10 * 10 * 5 = 500$ instructions. This is clearly not suitable, as GF generates 321 trees (with depth 6).

A better approach on the problem is to store all distinct names in a separate index. When a user begins to type an instruction, the application checks each word the user has typed. If a word exists in the name-index, then treat it as a name and replace it with `Foo`, then query the linearizations index. Retrieve the results, and change back Foo to the original name. However, as the application do not make any distinction between different type of names, we could end up with suggestions like *people who work in Python*, because *Python* was replaced by *Foo*. Luckily, this problem can be resolved by introducing a distinction between different types of names in the grammar.

### 3.3.2 *Introducing name types*

This application uses four different kind of names. Programming languages are used together with the Know relation. Organizations are used with the WorkWith relation. Locations are used with the WorkIn relation. Lastly, modules are used with the Use relation. We extend the grammar to support these new name types.

Figure 39 shows the new abstract syntax. Line 3 defines new name types. Line 7-10 defines how the names are instantiated. Lines 13-23 defines how names can be combined by using boolean operators. Note how we use the type Skill for programming languages.

```
1  cat
2    -- Names
3    Skill ; Organization ; Location ; Module ;
4
5  fun
6    -- Create unknown names
7    MkSkill : Symb -> Skill ;
8    MkOrganization : Symb -> Organization ;
9    MkModule: Symb -> Module ;
10   MkLocation : Symb -> Location ;
11
12   -- Boolean operators for Organizations
13   And_S : Skill -> Skill -> Skill ;
14   Or_S : Skill -> Skill -> Skill ;
15
16   And_O : Organization -> Organization -> Organization ;
17   Or_O : Organization -> Organization -> Organization ;
18
19   And_L : Location -> Location -> Location ;
20   Or_L : Location -> Location -> Location ;
21
22   And_M : Module -> Module -> Module ;
23   Or_M : Module -> Module -> Module ;
```

Figure 39: Abstract syntax with new name types

The concrete syntax for these new functions are implemented in the same way as the ones we removed (category Object and functions MkObject, And_O and Or_O). We have omitted the new concrete syntaxes from the thesis as they do not contribute to anything new.

As the grammar has changed, also the abstract syntax trees has changed. Figure 40 shows how GF now generates all trees with depth 4.

```
Instrucs> generate_trees
InstrucExternal Customer (UseExt (MkModule (MkSymb "Foo")))
InstrucInternal People (Know (MkSkill (MkSymb "Foo")))
InstrucInternal People (WorkIn (MkLocation (MkSymb "Foo")))
InstrucInternal People (WorkWith (MkOrganization (MkSymb "Foo")))
InstrucResource Project (UseRes (MkModule (MkSymb "Foo")))
```

Figure 40: Abstract syntax trees with name types

As the index stores the linearizations of each tree it is important to post process each linearization in order to preserve the type information. Currently, the type of each name would be lost when linearized, for example `MkSkill (MkSymb "Foo")` is linearized into just `"Foo"` if using the english concrete syntax.

To preserve type information, we replace each `"Foo"` with the current name type + an index. Figure 41 shows an example.

```
InstrucInternal People (InternalAnd (Know
            (And_S (MkSkill (MkSymb "Skill0")) (MkSkill (MkSymb "Skill1"))))
            (WorkIn (MkLocation (MkSymb "Location0"))))
```

Figure 41: An abstract syntax tree with name types and changed names

The previous example linearizes into *people who know Skill*0 *and Skill*1 *and who work in Location*0 by using the concrete syntax for English. We generate all trees again, linearize them into all natural languages and store them in the linearizations index.

In addition, as we store each name in a separate index, we associate each name with its type. By doing so, we can easily find out if a word is a name and then find out what type that name has.

The grammar and the Solr indices are now complete. The application can now suggest relevant sentences even though the user only has typed names (provided that the names exists in the name index). Figure 42 shows the pseudocode for the suggestion algorithm with informative comments. This algorithm is executed by the Java-application when a user has typed a partial question.

```
suggestions(sentence) {
    // sentence = "persons who knew java or python and work in London"
    // names[] = {Java, Python, London}
    names[] = extractNames(sentence);

    // "persons who knew java or python and work in London" ===>
    // "persons who knew Skill0 or Skill1 and work in Location0"
    sentence = replaceNamesWithTypes(sentence, names);

    // suggestions =
    //   { "people who know Skill0 or Skill1 and who work in Location0", ... }
    suggestions[] = findSentences(sentence);

    foreach suggestion in suggestions {
        // "people who know Skill0 or Skill1 and who work in Location0" ===>
        // "people who know Java or Python and who work in London"
        suggestion = restoreNames(names, suggestion);
    }
    return suggestions;
}
```

Figure 42: Pseudocode for the suggestion algorithm

### 3.3.2 *Supporting more sentences*

This section aims to add better accuracy to the suggestion engine by linearizing different variations of a sentence. In other words, we make it possible to linearize an abstract syntax tree into multiple strings.

When normally linearizing an abstract syntax tree in the GF-shell always gets just one result. However, there exists a parameter `-list` which can be used to obtain all possible ways of representing the abstract syntax tree.

GF adopts the pipe symbol to express that two elements are semantically equivalent. The following sections describes how the pipe symbol can be used to further extend the grammar.

We extend the grammar to treat a subject in singular form the same as a subject in plural form. I.e. the sentences *a person which knows Java* is the same as *people who know Java*. As we treat them equivalent, they will have the same abstract syntax tree, and this tree will linearize to only one Solr query, hence if we translate *a person who knows Java* into Solr, we will obtain a query that retrieves multiple results. The implementation of this extension is achieved by modifying the operation `mkI` (defined in Figure 26). The modification can be seen in Figure 43. The reason why the plural version is chosen as the preferable sentence is because `aPl_Det` is listed before `aSg_Det`.

```
oper
  mkI : N -> RS -> Utt = \noun_N,rs_RS -> mkUtt (mkNP (aPl_Det | aSg_Det)
                             (mkCN noun_N rs_RS)) ;
```

Figure 43: Treating singular form equivalent to plural form

The second extension is to add support of the word *that* so we can express sentences like *people that know Java*. This is achieved by moving the relative pronoun from the concrete syntax to the lexicon and adding support of multiple relative pronouns by using the variance function. Figure 44 shows how we define a new constant operation which will provide the concrete syntaxes with a relative pronoun.

```
interface LexInstrucs = open Syntax in {
  oper
    ...
    who_RP : RP ;
}
```

Figure 44: Adding a new relative pronoun to the lexicon interface

We then implement the constant operation we just defined. In English, we use the variance function to allow both `which_RP` and `that_RP`.

```
instance LexInstrucsEng of LexInstrucs = open Prelude, SyntaxEng,
                                      ParadigmsEng, ExtraEng in {
  oper
    ...
    who_RP = which_RP | that_RP ;
}
```

Figure 45: Defining the new relative pronoun in English

It is not possible to express the relative pronoun in more than one way in Swedish, so we let the new operation be implemented to return `which_RP`.

```
instance LexInstrucsSwe of LexInstrucs = open SyntaxSwe, ParadigmsSwe in {
  oper
    ...
    who_RP = which_RP ;
}
```

Figure 46: Defining the new relative pronoun in Swedish

Finally we change the operation `mkRS'` to use the new constant operation we defined in the lexicon.

```
oper
  mkRS' : VP -> RS = \vp -> mkRS (mkRCl who_RP vp) ;
```

Figure 47: Usage of the new relative pronoun

The third extension is to treat the *progressive form* of a verb as the same as the regular form. A progressive form of a verb indicates that something is happening right now or was happening or will be happening. The grammar in this project only supports present tense, so it will only cover progressive verbs which expresses that something is happening right now. This is achieved by using the variance function again.

```
oper
  mkRS' : VP -> RS = \vp -> mkRS (mkRCl who_RP (vp | progressiveVP vp)) ;
```

Figure 48: Treating progressive verb equivalent normal form

However, not all verbs can be used in progressive form. An example of such verb is *know*. An example of *know* in progressive form is *people who are knowing Java* which is not grammatically correct. Although it might look ugly, it is not an issue for our application as the suggestion engine only suggest the most suitable suggestion. The extension will contribute to make the suggestion engine richer as it allows users to use invalid instructions.

If we now linearize the abstract syntax tree which represents the sentence *people who know Java* we will obtain several linearizations (see Figure 49) which by the program are treated to be semantically equivalent. Note how the first linearization is the one we treat as the correct one. It is not a coincidence that it is listed first. The first argument when using the variance function is always linearized first, so we make sure that all the correct linearizations is in the beginning of the list. We can then use this to map the incorrect representations to the correct ones in the suggestion engine.

```
Languages: InstrucsEngRGL
Instrucs> l -lang=InstrucsEngRGL -list
>             InstrucInternal People (Know_R (MkSkill (MkSymb "Java")))
people who know Java, , people who are knowing Java, , people that know Java,
people that are knowing Java, persons who know Java,
persons who are knowing Java, persons that know Java,
persons that are knowing Java, a person which knows Java,
a person which is knowing Java, a person that knows Java,
a person that is knowing Java
```

Figure 49: Obtaining multiple linearizations by using `-list`

## 3.4 alternative implementation without the rgl

We have implemented the concrete syntax for English by using the RGL. This section shows how the same functionality can be achieved by concatenating strings. The purpose of this section is to be able to compare the complexity of a concrete syntax developed by using RGL and by concatenating strings.

As we concatenate strings in this implementation, all categories are defined as strings.

```
lincat
 Instruction = Str ;
 Internal, External, Resource = Str ;
 InternalRelation, ExternalRelation, ResourceRelation = Str ;
 Skill, Organization, Location, Module = Str ;
```

Figure 50: Concrete syntax for English by concatenating strings with implementation of categories

In order to achieve the same functionality as the concrete syntax using the RGL, we use variance of strings to support both singular and plural forms. We also use variance to make *persons* a synonym for *people*.

```
lin
 People = "people" | "persons" | "person" ;
 Customer = "customers" | "customer" ;
 Project = "projects" | "project" ;
```

Figure 51: Concrete syntax for English by concatenating strings with implementation of subjects

As with subjects, variance is also used with relations.

```
lin
 Know_R obj = ("know" | "knows") ++ obj;
 UseExt_R obj = ("use" | "uses") ++ obj ;
 UseRes_R obj = ("use" | "uses") ++ obj ;
 WorkWith_R obj = ("work with" | "works with") ++ obj ;
 WorkIn_R obj = ("work in" | "works in") ++ obj ;
```

Figure 52: Concrete syntax for English by concatenating strings with implementation of relations

An instruction is created by concatenating a subject with a relation. In between them we use variance to support connecting words *who, which* and *that*.

```
lin
 InstrucInternal internal relation = internal ++
                                  ("who" | "which" | "that") ++ relation ;
 InstrucExternal external relation = external ++
                                  ("who" | "which" | "that") ++ relation ;
 InstrucResource resource' relation = resource' ++
                                  ("which" | "that") ++ relation ;
```

Figure 53: Concrete syntax for English by concatenating strings with implementation of instructions

A name is just a string.

```
lin
 MkSkill s = s.s ;
 MkOrganization s = s.s ;
 MkLocation s = s.s ;
 MkModule s = s.s ;
```

Figure 54: Concrete syntax for English by concatenating strings with implementation of name

Two names are combined by adding the boolean operator in between them.

```
lin
 And_S s1 s2 = s1 ++ "and" ++ s2 ;
 Or_S s1 s2 = s1 ++ "or" ++ s2 ;

 And_O s1 s2 = s1 ++ "and" ++ s2 ;
 Or_O s1 s2 = s1 ++ "or" ++ s2 ;

 And_L s1 s2 = s1 ++ "and" ++ s2 ;
 Or_L s1 s2 = s1 ++ "or" ++ s2 ;

 And_M s1 s2 = s1 ++ "and" ++ s2 ;
 Or_M s1 s2 = s1 ++ "or" ++ s2 ;
```

Figure 55: Concrete syntax for English by concatenating strings with implementation of boolean operators for names

As with boolean operators for names, two relations are combined by adding the boolean operator in between them.

```
lin
 InternalAnd s1 s2 = s1 ++ "and" ++ s2 ;
 InternalOr s1 s2 = s1 ++ "or" ++ s2 ;

 ExternalAnd s1 s2 = s1 ++ "and" ++ s2 ;
 ExternalOr s1 s2 = s1 ++ "or" ++ s2 ;

 ResourceAnd s1 s2 = s1 ++ "and" ++ s2 ;
 ResourceOr s1 s2 = s1 ++ "or" ++ s2 ;
```

Figure 56: Concrete syntax for English by concatenating strings with implementation of boolean operators for relations

## 3.5 GENERATION OF MOCK DATA

This section describes how the data used by the application is generated. In order to generate data, we must know what to generate. From Section 3.1 we know the Solr queries our index must support. We have three different types of objects: *Customer*, *Person* and *Project*. Each of these types of objects has their own data. A customer shall have one value, a list of technologies it uses. A person shall have three values, a list of programming languages it knows, a list of cities it works in and a list of organizations it has been or are working with. A project shall have one value, the same as a customer, a list of technologies it uses. In addition to these requirements by the grammar, customers and persons also have names. Figure 57 shows an example of data that could exists in the Solr index.

```
{ // Customer
  "name" : "Amnesty"
  "object_type" : "Customer",
  "USES" : ["Solr", "Tomcat", "GF"]
},
{ // Person
  "name" : "Jane Doe"
  "object_type" : "Person",
  "KNOWS" : ["Java", "Python", "C"],
  "WORKS_IN" : ["London", "Gothenburg"],
  "WORKS_WITH" :  ["Amnesty", "Unicef"]
},
{ // Project
  "object_type" : "Project",
  "USES" : ["Android", "GF"]
}
```

Figure 57: Example data in JSON

Mock data is generated by combining data from text-files. We have two text-files concerning personal names (`first_names.txt` and `last_names.txt`). One text-file with organization names (`charity_organizations.txt`). One text file about programming languages (`programming_languages.txt`). Lastly, one text file containing cities (`cities.txt`).

An object is generated by taking one value from each text file that the object needs. For instance, to create an object of the type `Person`, we fetch values from `first_names.txt`, `last_names.txt`, `programming_languages.txt`, `cities.txt` and `charity_organizations.txt`. The values are randomly chosen from the text files. The object is then exported to Solr to be stored in the index.

# 4 RESULTS

This chapter presents results from the end application.

## 4.1 TRANSLATIONS

We begin by demonstrating how a few sentences are parsed into an abstract syntax which is linearized into all possible concrete syntaxes. The input is shown in the first box in each figure (with input language as a comment) and the result in second (larger) box. The result of a translation is JSON-data, given by the application.

The solr linearization of each abstract syntax tree is a clickable hyper-link in the web application. This link leads to an actual execution of a HTTP-GET request to the Solr engine. All such links leads to a Solr result which correctly filters data depending on the query. This data is however omitted from the results as the actual translations (and suggestions) are the main results of this project.

The first five figures shows the simplest translations. They include all subjects and all types of names. Each one contains one subject, one verb and one name of a certain type.

Figure 58 shows that the word `Java` is of the type `Skill`.

```
-- EnglishRGL
people who know Java
```

```
[
 {
  ’ast’: ’InstrucInternal People (Know_R (MkSkill (MkSymb "Java")))’,
  ’linearizations’: [
   {
    ’query’: ’people who know Java’,
    ’language’: ’InstrucsEngConcat’
   },
   {
    ’query’: ’people who know Java’,
    ’language’: ’InstrucsEngRGL’
   },
   {
    ’query’: ’select?q=*:*&wt=json&fq= object_type : Person AND KNOWS : ( Java )’,
    ’language’: ’InstrucsSolr’
   },
   {
    ’query’: ’personer som kan Java’,
    ’language’: ’InstrucsSweRGL’
   }
  ]
 }
]
```

Figure 58: Translation including `people` and a name of type `Skill`

Figure 59 shows that the word `London` is of the type `Location`.

```
-- English RGL
people who work in London
```

```
[
 {
  ’ast’: ’InstrucInternal People (WorkIn_R (MkLocation (MkSymb "London")))’,
  ’linearizations’: [
   {
    ’query’: ’people who work in London’,
    ’language’: ’InstrucsEngConcat’
   },
   {
    ’query’: ’people who work in London’,
    ’language’: ’InstrucsEngRGL’
   },
   {
    ’query’: ’select?q=*:*&wt=json&fq= object_type : Person AND WORKS_IN : ( London )’,
    ’language’: ’InstrucsSolr’
   },
   {
    ’query’: ’personer som arbetar i London’,
    ’language’: ’InstrucsSweRGL’
   }
  ]
 }
]
```

Figure 59: Translation including `people` and a name of type `Location`

Figure 60 shows that the word `Amnesty` is of the type `Organization`.

```
-- EnglishRGL
people who work with Amnesty
```

```
[
 {
  ’ast’: ’InstrucInternal People (WorkWith_R (MkOrganization (MkSymb "Amnesty")))’,
  ’linearizations’: [
   {
    ’query’: ’people who work with Amnesty’,
    ’language’: ’InstrucsEngConcat’
   },
   {
    ’query’: ’people who work with Amnesty’,
    ’language’: ’InstrucsEngRGL’
   },
   {
    ’query’: ’select?q=*:*&wt=json&fq= object_type : Person AND WORKS_WITH : ( Amnesty )’,
    ’language’: ’InstrucsSolr’
   },
   {
    ’query’: ’personer som arbetar med Amnesty’,
    ’language’: ’InstrucsSwe’
   }
  ]
 }
]
```

Figure 60: Translation including `people` and a name of type `Organization`

Figure 61 shows that the word `Solr` is of the type `Module`.

```
-- EnglishRGL
customers who use Solr
```

```
[
 {
  ’ast’: ’InstrucExternal Customer (UseExt_R (MkModule (MkSymb "Solr")))’,
  ’linearizations’: [
   {
    ’query’: ’customers who use Solr’,
    ’language’: ’InstrucsEngConcat’
   },
   {
    ’query’: ’customers who use Solr’,
    ’language’: ’InstrucsEngRGL’
   },
   {
    ’query’: ’select?q=*:*&wt=json&fq= object_type : Organization AND USES : ( Solr )’,
    ’language’: ’InstrucsSolr’
   },
   {
    ’query’: ’kunder som använder Solr’,
    ’language’: ’InstrucsSwe’
   }
  ]
 }
]
```

Figure 61: Translation including `customer` and a name of type `Module`

Similarly, also Figure 62 shows that the word `Solr` is of the type `Module`.

```
-- EnglishRGL
projects who use Solr
```

```
[
 {
  'ast': 'InstrucResource Project (UseRes_R (MkModule (MkSymb "Solr")))',
  'linearizations': [
   {
    'query': 'projects who use Solr',
    'language': 'InstrucsEngConcat'
   },
   {
    'query': 'projects who use Solr',
    'language': 'InstrucsEngRGL'
   },
   {
    'query': 'select?q=*:*&wt=json&fq= object_type : Project AND USES : ( Solr )',
    'language': 'InstrucsSolr'
   },
   {
    'query': 'projekt som använder Solr',
    'language': 'InstrucsSwe'
   }
  ]
 }
]
```

Figure 62: Translation including `project` and a name of type `Module`

Figure 63 shows how the applications handles the boolean operator *and*. The application handles the case for *or* similarly.

```
-- EnglishRGL
people who know Java and Python
```

```
[
 {
  ’ast’: ’InstrucInternal People (Know_R (And_S (MkSkill (MkSymb "Java"))
                                                      (MkSkill (MkSymb "Python"))))’,
  ’linearizations’: [
   {
    ’query’: ’people who know Java and Python’,
    ’language’: ’InstrucsEngConcat’
   },
   {
    ’query’: ’people who know Java and Python’,
    ’language’: ’InstrucsEngRGL’
   },
   {
    ’query’: ’select?q=*:*&wt=json&fq= object_type : Person AND
                                                KNOWS : ( ( Java ) AND ( Python ) )’,
    ’language’: ’InstrucsSolr’
   },
   {
    ’query’: ’personer som kan Java och Python’,
    ’language’: ’InstrucsSwe’
   }
  ]
 }
]
```

Figure 63: Translation including `people` and two names of the type `Skill`

Figure 64 (on the next page) shows how the application handles an ambiguous instruction. Two abstract syntax trees are seen in the result, as there are two ways of interpreting the instruction. The different interpretations can be modelled as follows: *people who know (Java and Python) or Haskell* or *people who know Java and (Python or Haskell)*.

```
-- EnglishRGL
people who know Java and Python or Haskell
```

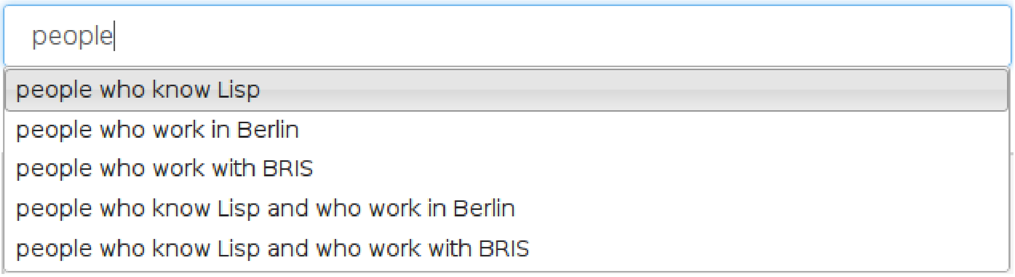
```
[
 {
  ’ast’: ’InstrucInternal People (Know_R (Or_S (And_S (MkSkill (MkSymb "Java"))
                       (MkSkill (MkSymb "Python"))) (MkSkill (MkSymb "Haskell"))))’,
  ’linearizations’: [
   {
    ’query’: ’people who know Java and Python or Haskell’,
    ’language’: ’InstrucsEngConcat’
   },
   {
    ’query’: ’people who know Java and Python or Haskell’,
    ’language’: ’InstrucsEngRGL’
   },
   {
    ’query’: ’select?q=*:*&wt=json&fq= object_type : Person AND
                      KNOWS : ( ( ( Java ) AND ( Python ) ) OR ( Haskell ) )’,
    ’language’: ’InstrucsSolr’
   },
   {
    ’query’: ’personer som kan Java och Python eller Haskell’,
    ’language’: ’InstrucsSwe’
   }
  ]
 },
 {
  ’ast’: ’InstrucInternal People (Know_R (And_S (MkSkill (MkSymb "Java"))
                      (Or_S (MkSkill (MkSymb "Python")) (MkSkill (MkSymb "Haskell")))))’,
  ’linearizations’: [
   {
    ’query’: ’people who know Java and Python or Haskell’,
    ’language’: ’InstrucsEngConcat’
   },
   {
    ’query’: ’people who know Java and Python or Haskell’,
    ’language’: ’InstrucsEngRGL’
   },
   {
    ’query’: ’select?q=*:*&wt=json&fq= object_type : Person AND
                      KNOWS : ( ( Java ) AND ( ( Python ) OR ( Haskell ) ) )’,
    ’language’: ’InstrucsSolr’
   },
   {
    ’query’: ’personer som kan Java och Python eller Haskell’,
    ’language’: ’InstrucsSwe’
   }
  ]
 }
]
```

Figure 64: Translation of an ambiguous instruction involving `people` and three names of the type `Skill`

## 4.2 SUGGESTIONS

This section shows how the application reacts to input before the user has chosen to translate the instruction. Each figure shows an image of how the application suggest valid sentences from a partial instruction.

Figure 65 shows suggestions based on the first word in a valid sentence.

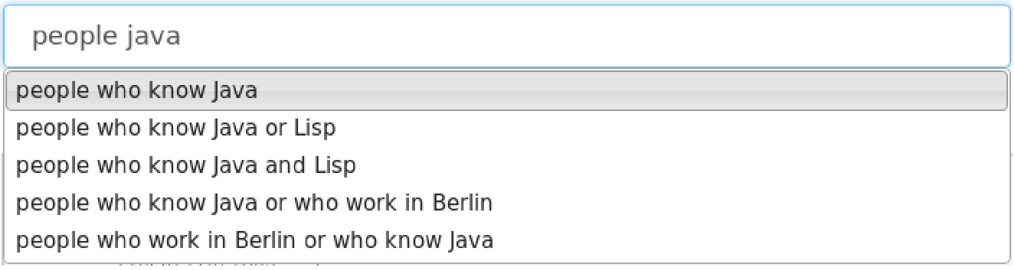


people

people who know Lisp
people who work in Berlin
people who work with BRIS
people who know Lisp and who work in Berlin
people who know Lisp and who work with BRIS

Figure 65: Suggestions based on the word `people` by using EnglishRGL or EnglishConcat

Figure 66 shows that it is possible to use a combination of words to retrieve suggestions. It also shows that we can use the first and the last word to get suggestions.

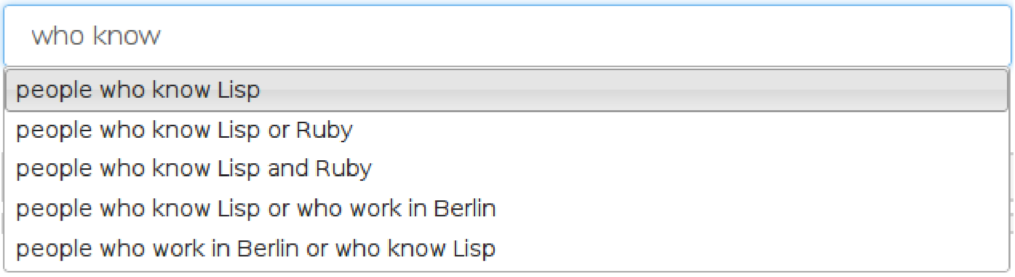


people java

people who know Java
people who know Java or Lisp
people who know Java and Lisp
people who know Java or who work in Berlin
people who work in Berlin or who know Java

Figure 66: Suggestions based on the words `people java` by using EnglishRGL or EnglishConcat

Figure 67 demonstrates that the words do not have to be the first or last words in the sentence.

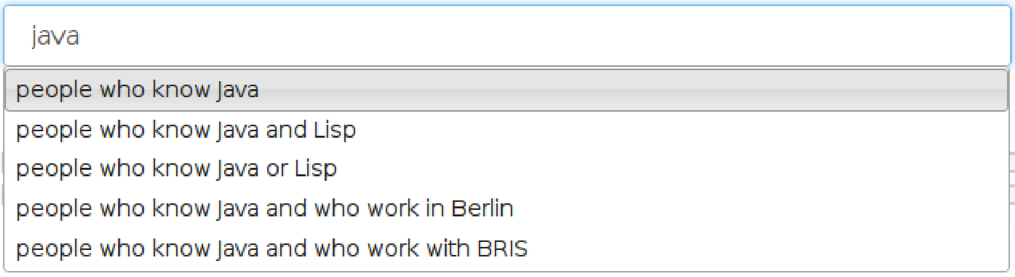


who know

people who know Lisp
people who know Lisp or Ruby
people who know Lisp and Ruby
people who know Lisp or who work in Berlin
people who work in Berlin or who know Lisp

Figure 67: Suggestions based on the words `who know` by using EnglishRGL or EnglishConcat

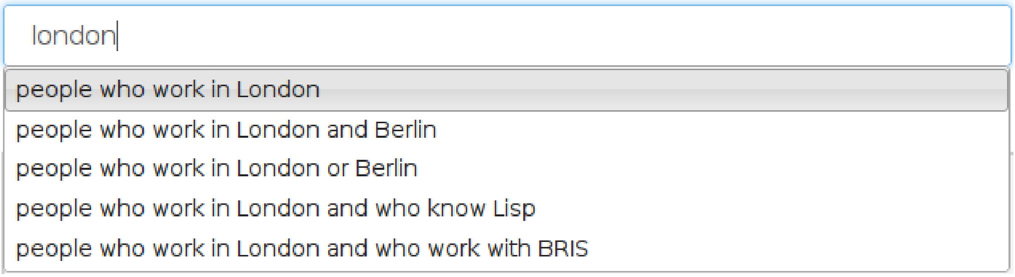
The next four figures shows how it is possible to retreive suggestions based on only names.



java

people who know Java
people who know Java and Lisp
people who know Java or Lisp
people who know Java and who work in Berlin
people who know Java and who work with BRIS

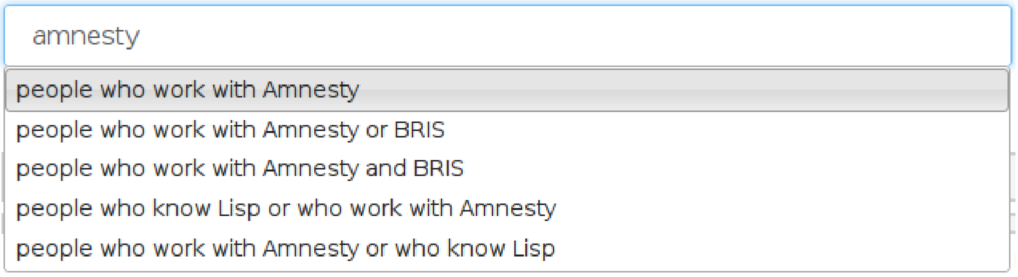
Figure 68: Suggestions based on a name of type Skill by using EnglishRGL or EnglishConcat



london

people who work in London
people who work in London and Berlin
people who work in London or Berlin
people who work in London and who know Lisp
people who work in London and who work with BRIS

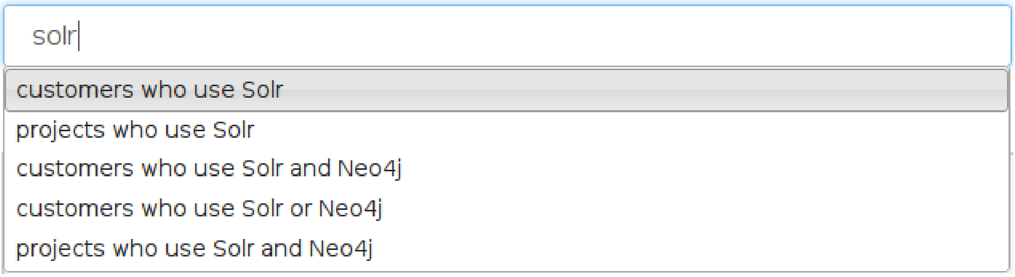
Figure 69: Suggestions based on a name of type Location by using EnglishRGL or EnglishConcat



amnesty

people who work with Amnesty
people who work with Amnesty or BRIS
people who work with Amnesty and BRIS
people who know Lisp or who work with Amnesty
people who work with Amnesty or who know Lisp

Figure 70: Suggestions based on a name of type Organization by using EnglishRGL or EnglishConcat

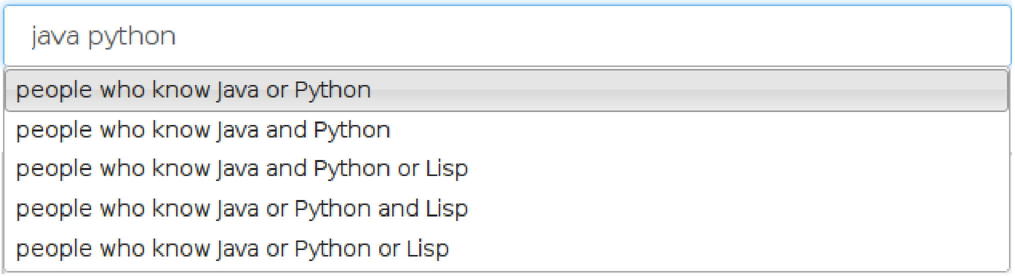


solr

customers who use Solr
projects who use Solr
customers who use Solr and Neo4j
customers who use Solr or Neo4j
projects who use Solr and Neo4j

Figure 71: Suggestions based on a name of type Module by using EnglishRGL or EnglishConcat

Figure 72 shows suggestions based on two names of the same type.

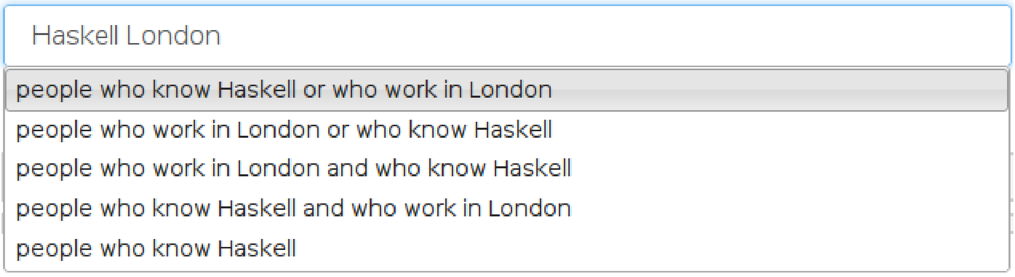


java python

people who know Java or Python
people who know Java and Python
people who know Java and Python or Lisp
people who know Java or Python and Lisp
people who know Java or Python or Lisp

Figure 72: Suggestions based on two names of type Skill by using EnglishRGL or EnglishConcat

Figure 73 shows that names do not have to be of the same type to suggest relevant instructions.

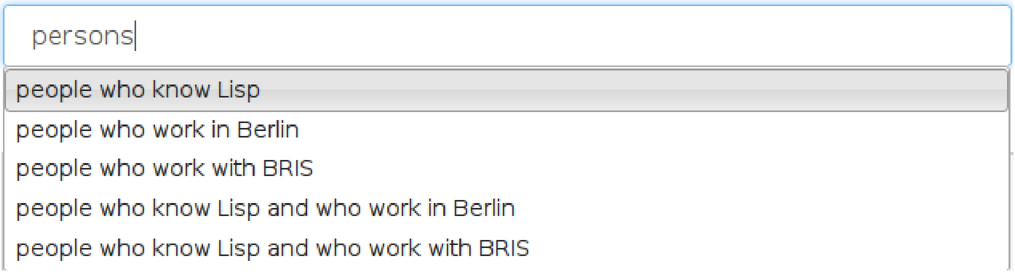


Haskell London

people who know Haskell or who work in London
people who work in London or who know Haskell
people who work in London and who know Haskell
people who know Haskell and who work in London
people who know Haskell

Figure 73: Suggestions based on two names of different types by using EnglishRGL or EnglishConcat

Figure 74 shows how we can use the word `persons` in order to get suggestions. The word `persons` does not exist in any suggestion, but as it is a synonym to `people`, the application suggest relevant instructions.

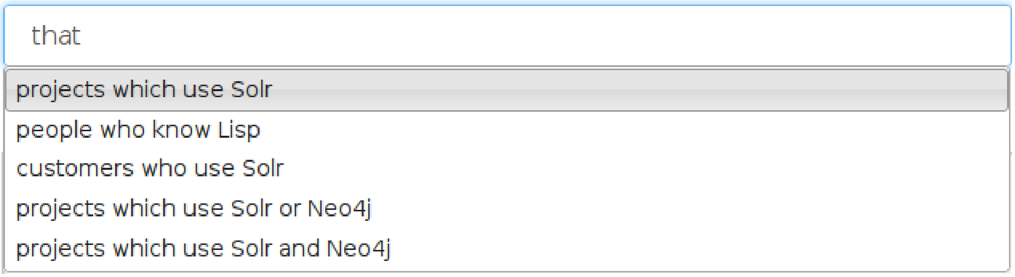


persons

people who know Lisp
people who work in Berlin
people who work with BRIS
people who know Lisp and who work in Berlin
people who know Lisp and who work with BRIS

Figure 74: Synonyms based on a synonym by using EnglishRGL or EnglishConcat

Also Figure 75 shows how we can get suggestions based on another synonym. Both EnglishRGL and EnglishConcat gives suggestions based on the word *that*.

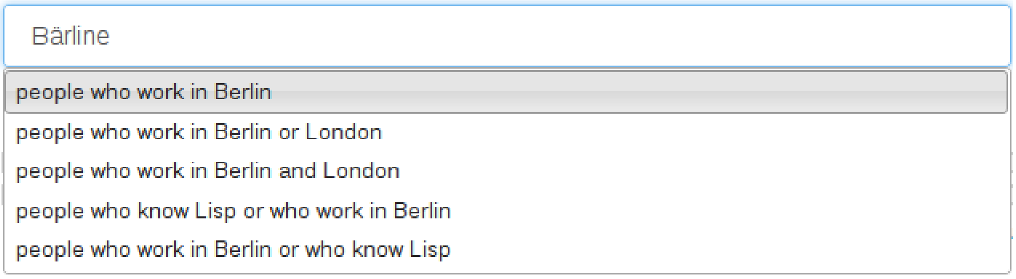


that

projects which use Solr
people who know Lisp
customers who use Solr
projects which use Solr or Neo4j
projects which use Solr and Neo4j

Figure 75: Synonyms based on a synonym by using EnglishConcat

Figure 76 shows how the application shows relevant suggestion based on a misspelled word. This is achieved by using a approximate string matching algorithm in Solr.

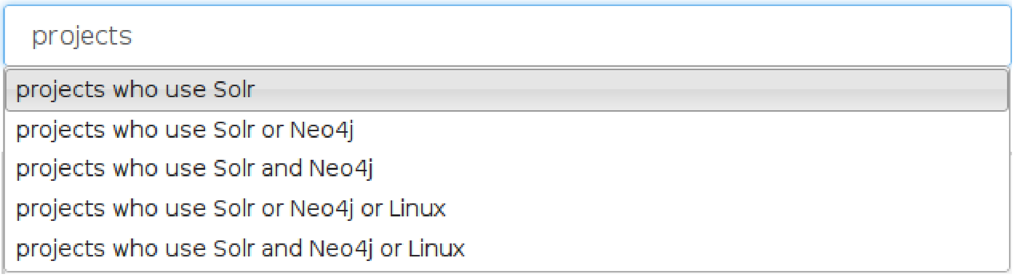


Bärline

people who work in Berlin
people who work in Berlin or London
people who work in Berlin and London
people who know Lisp or who work in Berlin
people who work in Berlin or who know Lisp

Figure 76: Suggestions based on a misspelled word by using EnglishRGL or EnglishConcat

Figure 77 and Figure 78 show how the same word can obtain different suggestions by using the concrete languages EnglishRGL and EnglishConcat.



projects

projects who use Solr

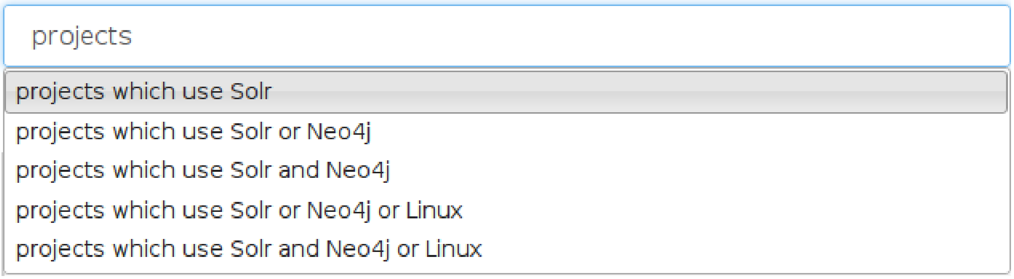
projects who use Solr or Neo4j

projects who use Solr and Neo4j

projects who use Solr or Neo4j or Linux

projects who use Solr and Neo4j or Linux

Figure 77: Suggestions based on the string *projects* by using EnglishRGL



projects

projects which use Solr

projects which use Solr or Neo4j

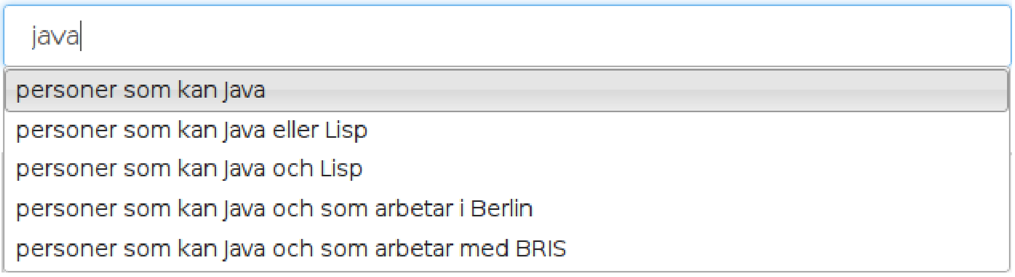
projects which use Solr and Neo4j

projects which use Solr or Neo4j or Linux

projects which use Solr and Neo4j or Linux

Figure 78: Suggestions based on the string *projects* by using EnglishConcat

Lastly, Figure 79 demonstrates that the application can also give suggestions in Swedish. The application can translate valid Swedish sentences as long as the user has chosen SwedishRGL as input language in the application. All instructions that can be translated from EnglishRGL can also be translated from SwedishRGL.



java

personer som kan Java

personer som kan Java eller Lisp

personer som kan Java och Lisp

personer som kan Java och som arbetar i Berlin

personer som kan Java och som arbetar med BRIS

Figure 79: Suggestions in Swedish by using SwedishRGL

# 5 CONCLUSIONS

## 5.1 A BRIEF DISCUSSION ABOUT THE RESULTS

The results are overall very positive. We have managed to build a system which can translate a small subset of instructions formulated in English or Swedish into Solr query language which represent the same semantics. In addition, as the results show, we can obtain relevant suggestions of instructions based on a partial sentence or by just using keywords - which was exacly what we wanted.

## 5.2 COMPARISON OF THE RGL AND SIMPLE CONCATENATION

We developed a concrete syntax for English by using the RGL (EngRGL) and one for English by concatenating strings (EngConcat). EngRGL is without doubt the most complex and least intuitive if the reader has no prior knowledge of the RGL. Conversely, EngConcat is very instuitive and straight forward for any reader with basic knowledge of programming. The two syntaxes supports equally many abstract syntax trees, but as we have added more variations of linearizations to EngConcat, it is possible to represent a few abstract syntax trees in more ways in EngConcat than in EngRGL. One example is *people who know Java and work in London* which we could not express by using the RGL. We had to content ourselves with the sentence *people who know Java and* ***who*** *work in London*.

### 5.2.1 *Support for ungrammatical sentences*

Concatenation of strings make it possible to create all kind of different (and strange) linearizations with the variance function. For instance, we can mix singular words with plural words and make it possible to map sentences expressed in bad grammar into a correct suggestion. One example is *people* ***which knows*** *Java* where the correct representation is *people who know Java*, but it is very clear that we should treat them semantically equivalent.

By doing so, users with little knowledge of writing in English have a higher probability of finding the correct sentence by using this approach. This technique is not supported by the RGL.

### 5.2.2 *The RGL produces an invalid sentence*

Another problem we had with the RGL is the constant `which_RP` which linearizes into *which* if the subject is in singular form and *who* if the subject is in plural form. The RGL does not take into account that this rule is only valid if the subject refers to a group of humans, it applies the rule to any group of things. This is probably a bug which will be fixed soon.

### 5.2.3 *RGL and concatenation - Conclusions*

We found out that concatenation is much more suitable when developing a project which translates from a natural language into machine readable instructions. However, this conclusion assumes that the developer of the grammar has knowledge of the natural language in order to correctly map ungrammatical sentences into their grammatical representations. If the developer is not an expert of the natural language, it will be very hard to use concatenation since she might not now which sentence that is the correct one. The RGL can be used in such cases, but then only grammatical sentences can be produced and there might be cases when it is not possible the articulate a sentence in a speficic way.

## 5.3 suggestion engine

We generate all possible sentences and store the result in a Solr index. This provides us with a strong suggestion engine which can suggest relevant suggestions based on a partial sentence or only keywords. We can also generate all possible Solr queries and also store them in the index. If we now link each suggestion to the corresponding Solr query, we get a very fast and efficient translating service (faster than translating with GF). However, if we do so, the end application does not use the grammar at all. The grammar is only used for initialization when we generate the instructions in natural languages and Solr syntax. It is however unknown if this is an optimal solution. If so, is it worth to invest time to learn GF with its programming language and libraries to just generate sentences? As with learning any programming language, this varies depending on the learning person. How complex it is to generate sentences in a stand alone application without GF and a grammar is also unknown.

We conclude that we are unsure if the method of using a Solr index is the right approach on suggestions together with a grammar.

## 5.4 KNOWN ISSUES

### *Incorrect English grammar*

As described in Section 5.2, the concrete syntax based on the RGL does not properly linearize the plural form of the constant `which_RP` when using a subject that does not refer to a group of humans.

### *Name suggestions*

The suggestion engine splits the input on words. It checks if a word can be represented by a name. However, the application will not find names based on multiple words. If a name is *Apache Solr* then at the current implementation we would replace *Apache* with *Apache Tomcat* and *Solr* with *Apache Solr* if both names existed in the index. The reason is because the application simply takes the first name it finds. A better solution would try to find longer names first in order to get more precise results.

### *PGF runtime*

As described in Appendix A, the Java runtime make us of the PGF-format when dealing with grammars. The Java class `org.grammaticalframework.pgf.PGF` sometimes can not be initialized properly. It is unknown why this occur, but it might depend on how Tomcat handles its native resources, as the PGF-class is a connector to the native libraries. The problem can be temporarly solved by redeploying the application.

### *Limited amount of suggestions*

Suggestions are limited to depth 6 of an abstract syntax tree. This means maximum 6 edges from root to leaf in an abstract syntax tree. It is therefore not possible to get suggestions containing many names. The depth can however be increased in the program.

## 5.5 FUTURE WORK

### *Improvments of suggestions*

The suggestions obtained from the appliction are relevant to the input words, but they can definitely be better. The suggestion engine do not give any suggestions when the textbox is empty, in other words, it does not give suggestions based on no words. For a user that has no clue of what to type, a suggestion of base sentences could help the user to get to know the system.

If one starts to type a partial instruction that does not contain any name, e.g. *people who know*, then the application will suggest the sentence *people who know **Lisp***. The reason is because application automatically tries to fill the missing name, and `Lisp` is the first name of the type `Skill` it finds. It would be good to use some heuristic that can learn what the most relevant name of the specific type would be.

Another nice improvement of the suggestion engine is to automatically choose the first suggestion available if the user tries to translate an invalid instruction. Or at least ask the user if the suggested sentence is what they meant.

*Instructions in speech*

Speech recognition software translate from speech to text. If one has access to such software, it is relatively easy to enable speech to instructions translation by using the application developed in this project. One simply uses the speech recognition software to translate speech to text, and then take the text and use it as input to the program. The program will show the most relevant suggestions based on the input.

*Proper handling of ambiguous instructions*

If one translates an ambiguous instruction, the application will show all resulting abstract syntax trees with their corresponding linearizations. A better program would ask the user to clarify the instruction or choose one of the abstract syntax trees as the one to use. Both alternatives are non trivial to implement, since the former need a better grammar in order to distinguish between ambiguous instructions and the latter need some heuristic to know which instruction to choose.

# A

# GF SHELL AND RUNTIME SYSTEMS

This appendix describes how the dependencies of the application can be installed.

## A.1 GF SHELL

The GF shell [12][p. 31] is the interpreter which can be used together with GF-grammars. The shell is used by the application when generating abstract syntax trees and is therefore needed in order to use the application.

An easy and convenient method of installing GF is by using the Haskell program `cabal` [25]. Cabal is available if the haskell platform [26] is installed on a system. The haskell platform can be installed on Debian by executing

```
sudo apt-get install haskell-platform
```

Run `'cabal update'` in a shell to download the latest package list from `hackage.haskell.org`. GF can now be installed by executing:

```
$ cabal install gf
```

The GF-binary can now be found in `~/.cabal/bin/gf`. Append this directory to the path variable:

```
export PATH=$PATH:~/.cabal/bin/gf
```

Note that `export` does not set $PATH permanently, so add the command to `~/.bashrc` or similar.

## A.2 GF RUNTIME SYSTEMS

While the GF-shell is a powerful tool, it is not very convenient to interact with when programming an application. Luckily, the creators of GF has thought about this and built embeddable runtime systems for a few programming languages [23, p. 3]. These runtime systems makes it possible to interact with a grammar directly through language specific data types. We have chosen to work with the Java-runtime system in this project.

### 1.2.1 *Portable Grammar Format*

The GF-shell interacts with grammars by interpreting the GF programming language. This allows us to write our grammars in a simple and convenient syntax. Interpreting the GF programming language directly is however a heavy operation [23][p. 13], especially with larger grammars. This is where the Portable Grammar Format (PGF) [23][p. 14] comes in. PGF is a custom made machine language which is created by compiling a grammar with GF into a PGF-file. The runtime systems works exclusively with PGF-files.

### 1.2.2 *GF libraries*

In order to use the Java-runtime, we first need to build a few libraries which are used by the runtime system. The Java-runtime system depends on the C-runtime system and a special wrapper between the C- and the Java-runtime. The libraries are platform dependent. There are some some pre-generated libraries in the GF-project, but we have chosen to build the libraries from source in this tutorial. The main reason is because we want to make the project suitable for many architectures. We will start by building and installing the C-libraries. We will then go through how we can build the wrapper library.

### 1.2.2 *Building and installing the C-runtime*

Start by fetching the needed dependencies

```
$ sudo apt-get install gcc autoconf libtool
```

Download the latest source code of GF from GitHub.

```
$ git clone https://github.com/GrammaticalFramework/GF.git
```

It is also possible to download the project as an archive by visiting the repository url.

You will receive a directory `GF/`. Change the current working directory to the C-runtime folder.

```
$ cd GF/src/runtime/c/
```

Generate a configuration file

```
$ autoreconf -i
```

Check that all dependencies are met

```
$ ./configure
```

If there exists a dependency that is not fulfilled, try to install an appropriate package using your package-manager.

Build the program

```
$ make
```

Install the libraries you just built

```
$ sudo make install
```

Make sure the installed libraries are installed into `/usr/local/lib`. It is crucial that they exists in that directory in order for the program to work.

### *1.2.3 Building and installing the C to Java wrapper library*

Start by installing the needed dependency

```
$ sudo apt-get install g++
```

The wrapper is built by using a script which is executed in Eclipse [27]. This step assumes that you have Eclipse installed with the CDT-plugin [28]. If you don't have Eclipse, you can download it with your package manager, just do not forget to install the CDT-plugin.

Start Eclipse and choose `File > Import..` in the menu. Choose `Import Existing Projects into Workspace` and click on the `Next` button. Select `Browse...` and navigate to the location where you downloaded GF from GitHub and press enter. Uncheck everything except `jpgf` and click on `Finish`. You have now imported the project which can build the Java-runtime system.

Eclipse need to have pointers to some directories of the Java virtual machine. It is unfortunately not possible to use environment variables in eclipse, so we need therefore to set the values manually.

Right-click on the project and choose `Properties`. Expand the `C/C++ Build` menu, click on `Settings`. Click on `Includes` which is located below `GCC C Compiler`. You will see one directory listed in the textbox. You need to check that this directory exists. If not, change it to the correct one. For instance, this tutorial was written using Debian 7 amd64 with Oracle Java 8, hence the correct directory is

```
/usr/lib/jvm/java-8-oracle/include
```

In addition, one more directory is also needed by the project to build properly.

```
/usr/lib/jvm/java-8-oracle/include/linux
```

The project also needs another flag in order to build properly. In the `Properties`-window, click on `Miscellaneous` below `GCC C Compiler`. Add `-fPIC` to the text field next to `Other flags`. Click on `Ok` to save the settings.

You can now build the project by choosing `Project > Build Project` in the menu. If everything went well you shall have generated a file `libjpgf.so` in `Release (posix)/`. You can check that the dependencies of `libjpgf.so` is fulfilled (i.e. it finds the C-runtime) by executing the following in a terminal

```
$ ldd libjpgf.so
```

If you do not see `'not found'` anywhere in the results, all dependencies are met. However if the C-runtime libraries are missing then `LD_LIBRARY_PATH` is probably not set. This is achieved by executing the following in the terminal:

```
$ export LD_LIBRARY_PATH=/usr/local/lib
```

This is only a one time setting and the variable will not exist for the next terminal session. This is however not a problem, since the libraries will be used by Apache Tomcat which will set the variable at startup.

Finish the tutorial by moving the wrapper library to the correct location.

```
$ mv libjpgf.so /usr/local/lib
```

# B INSTALLING THE APPLICATION

While Appendix A focused on installing GF-related dependencies, this appendix explains how the application can be installed into an application server. Note that the application will not run unless all GF-dependencies are installed. The application (and the source code) can be downloaded from thesis.agfjord.se where also a working demo of the application can be found.

## B.1 INSTALLING AND CONFIGURATING APACHE TOMCAT

This application can be executed by using any application server that supports `WAR`-files. The `WAR`-file in this project is built using Maven, which can also upload the file to an instance of the application server Tomcat. This method is very convenient since it automates a lot work. The following section describes how to install and configure Tomcat and Maven to work with this project.

Download and install Tomcat 8 and Maven (here by using aptitude package-manager).

```
$ sudo apt-get install tomcat8 tomcat8-admin maven
```

Tomcat requires an uploader to have the correct permissions.
Edit `/etc/tomcat8/tomcat-users.xml` and add the following:

```
/etc/tomcat8/tomcat-users.xml
------------------------------------------------
<tomcat-users>
  <role rolename="manager-gui"/>
  <role rolename="manager-script"/>
  <user username="admin" password="secr3t" roles="manager-gui,manager-script"/>
</tomcat-users>
```

As the application will use the generated wrapper library `libjpgf.so`, we need to make a proper reference to this library and its dependencies (the C-libraries). This is achieved by creating a new file `setenv.sh` in the directory `/usr/share/tomcat8/bin/`, the location of this directory can differ on different Linux-distributions. The directory shall contain the file `catalina.sh`, so a search on the file should show the correct directory.

Create the file `setenv.sh` and add the following

```
/usr/share/tomcat8/bin/setenv.sh
------------------------------------------------
#!/bin/sh
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
export JAVA_OPTS='-Dsolr.solr.home=<project_workspace>/solr-instrucs'
```

Note that `<project_workspace>` must be replaced by the actual location of the workspace, and make sure it is writeable by Tomcat. Restart Tomcat for the changes to take effect.

```
$ sudo service tomcat8 restart
```

The next thing we would like to do is to allow Maven to upload applications to Tomcat. As Tomcat now has an admin user with a password, we can use this to setup a server definition in Maven.
Add the following to `/etc/maven/settings.xml`

```
/etc/maven/settings.xml
------------------------------------------------
<servers>
 <server>
    <id>localTomcatServer</id>
    <username>admin</username>
    <password>secr3t</password>
  </server>
</servers>
```

The field `id` is used by the application to define that it shall be uploaded to the server we just specified.

## B.2 uploading the solr-service

The application makes use of a Solr-service which is bundled as a maven project inside `<project_workspace>/solr_mvn`. The Solr-service can be uploaded to Tomcat by executing the following:

```
$ cd <project_directory>/solr_mvn/
$ mvn tomcat7:deploy
```

## B.3 generating mock-data

We generate mock-data for the suggestion engine by executing a program. The program uses a the jar file `org.grammaticalframework.pgf.jar` as dependency, the jar must therefore be added to the local maven repository. Execute the following:

```
$ cd <project_directory>/
$ mvn install:install-file -Dfile=org.grammaticalframework.pgf.jar
                           -DgroupId=org.grammaticalframework
                           -DartifactId=pgf -Dversion=1.0 -Dpackaging=jar
```

Mock data can now be generated by executing the following:

```
$ cd <project_directory>/mock-data/
$ mvn compile
$ export MAVEN_OPTS=’-Djava.library.path=/usr/local/lib’
$ mvn exec:java -Dexec.mainClass="org.agfjord.graph.Main"
```

There also exists a script `populize_solr` inside `mock-data/` that is more convenient to use.

## B.4 UPLOADING THE WEBSITE

The project can be uploaded to tomcat by executing the following:

```
$ cd <project_directory>/nlparser/
$ mvn tomcat7:deploy
```

The application shall now be accessible through the URL `http://localhost:8080/nlparser`.

# BIBLIOGRAPHY

[1] Donald E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, December 1974. (Cited on page ii.)

[2] Danica D. Damljanovic. Natural Language Interfaces to Conceptual Models. 2011. URL http://gate.ac.uk/sale/dd/thesis/thesis.pdf. [Online; accessed 9-August-2014]. (Cited on page 1.)

[3] Esther Kaufmann and Abraham Bernstein. How useful are natural language interfaces to the semantic web for casual end-users? *6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007. Proceedings*, pages 281–294, 2007. URL http://www.cs.xu.edu/csci390/08s/NLInterfacesUsefulToEndUsers.pdf. (Cited on pages 1 and 3.)

[4] Unicorn: A System for Searching the Social Graph. *International Conference on Very Large Data Bases (VLDB)*. URL http://www.vldb.org/pvldb/vol6/p1150-curtiss.pdf. [Online; accessed 24-July-2014]. (Cited on pages 1 and 3.)

[5] Marissa Mayer. Seattle Conference on Scalability: Scaling Google for Every User, 2007. URL https://www.youtube.com/watch?v=Syc3axgRsBw. [Online; accessed 1-July-2014]. (Cited on page 2.)

[6] Donald E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968. ISSN 0025-5661. doi: 10.1007/BF01692511. (Cited on page 2.)

[7] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990. ISBN 020102988X. (Cited on page 2.)

[8] Donald E. Knuth. Backus Normal Form vs. Backus Naur Form. *Commun. ACM*, 7(12):735–736, December 1964. ISSN 0001-0782. doi: 10.1145/355588.365140. URL http://doi.acm.org/10.1145/355588.365140. (Cited on page 2.)

[9] Aarne Ranta. Grammatical framework: A type-theoretical grammar formalism. journal of functional programming. *Journal of Functional Programming*, 14(2):145–189, 2004. URL http://www.cse.chalmers.se/~aarne/articles/gf-jfp.pdf. (Cited on page 2.)

[10] Per Martin-Löf and Giovanni Sambdin. *Intuitionistic type theory*. 1984. URL http://www.cs.cmu.edu/afs/cs/Web/People/crary/819-f09/Martin-Lof80.pdf. [Online; accessed 10-August-2014]. (Cited on page 2.)

[11] Xiao Li. Under the Hood: The natural language interface of Graph Search, 2013. URL https://www.facebook.com/notes/facebook-engineering/under-the-hood-the-natural-language-interface-of-graph-search/10151432733048920. [Online; accessed 23-July-2014]. (Cited on page 3.)

[12] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth). (Cited on pages 4 and 54.)

[13] Aarne Ranta. The GF resource grammar library. *Linguistic Issues in Language Technology*, 2009. URL http://elanguage.net/journals/index.php/lilt/article/viewFile/214/158.Rosetta. (Cited on pages 4 and 9.)

[14] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6. (Cited on page 4.)

[15] Ramona Enache. *Frontiers of Multilingual Grammar Development*. Chalmers University of Technology, Göteborg, 2013. URL http://publications.lib.chalmers.se/publication/183497-frontiers-of-multilingual-grammar-development. ISBN-13: 978-91-628-8787-2. (Cited on page 4.)

[16] Rafał Kuć. *Apache Solr 3.1 cookbook*. Packt Publishing, Birmingham, UK, 2011. (Cited on page 5.)

[17] Björn Bringert, Thomas Hallgren, and Aarne Ranta. GF Resource Grammar Library: Synopsis. http://www.grammaticalframework.org/lib/doc/synopsis.html. [Online; accessed 10-August-2014]. (Cited on page 9.)

[18] Krasimir Angelov, Björn Bringert, and Aarne Ranta. Speech-enabled hybrid multilingual translation for mobile devices. *EACL 2014*, page 41, 2014. URL http://www.aclweb.org/anthology/E/E14/E14-2011.pdf. (Cited on page 15.)

[19] Krasimir Angelov. Incremental parsing with parallel multiple context-free grammars. In *European Chapter of the Association for Computational Linguistics*, 2009. URL www.aclweb.org/anthology/E09-1009. (Cited on page 15.)

[20] Andrzej Bialecki, Robert Muir, and Grant Ingersoll. Apache Lucene 4. http://opensearchlab.otago.ac.nz/paper_10.pdf, 2012. [Online; accessed 20-June-2014]. (Cited on page 15.)

[21] Ricardo Jimenez-Peris and Marta Patiño-Martinez. Java Enterprise Edition. In Ling Liu and M.Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1578–1579. Springer US, 2009. ISBN 978-0-387-35544-3. doi: 10.1007/978-0-387-39940-9_1542. (Cited on page 16.)

[22] Jason Van Zyl. *Maven - the definitive guide: everything you need to know from ideation to deployment.* O'Reilly, 2008. ISBN 978-0-596-51733-5. (Cited on page 16.)

[23] Krasimir Angelov. *The Mechanics of the Grammatical Framework*. Chalmers University of Technology, Göteborg, 2011. URL `http://publications.lib.chalmers.se/publication/149979-the-mechanics-of-the-grammatical-framework`. ISBN-13: 978-91-7385-605-8. (Cited on pages 17, 54, and 55.)

[24] Vivek Chopra, Sing Li, and Jeff Genender. *Professional Apache Tomcat 6*. Wrox, 2011. ISBN 978-0-471-75361-2. (Cited on page 18.)

[25] Isaac Jones. The Haskell Cabal, a common architecture for building applications and libraries. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.127.9361`. [Online; accessed 9-August-2014]. (Cited on page 54.)

[26] The Haskell Platform. URL: `https://www.haskell.org/platform/`. [Online; accessed 10-August-2014]. (Cited on page 54.)

[27] Steve Holzner. *Eclipse Cookbook*. O'Reilly Media, Inc., 1 edition, 2004. ISBN 978-0-596-00710-2. (Cited on page 56.)

[28] Eclipse Foundation. Eclipse CDT. URL `http://www.eclipse.org/cdt/`. [Online; accessed 10-August-2014]. (Cited on page 56.)