CONTENTS

ACRONYMS

GF	Grammatical Framework
RGL	Resouce Grammar Library
Java EE	Java Enterprise Edition
PGF	Portable Grammar Format

INTRODUCTION

1.1 AN ALTERNATIVE USER INTERFACE

In this thesis we will investigate how we can create a user interface which allows us to execute queries in a query language by expressing instructions in natural language. In other terms, we want to 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. A natural language is a language that humans use to communicate with each other.

1.2 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 the server for information in a natural language.

In this project, we have chosen 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 an intranet which consists of a database with information about employees, customers and projects. A typical instruction in this environment could be

```
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. The simply rely purely on keywords [1]. The following instruction is more suited for expert users

```
people know java
```

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

---

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

1.3 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 into query language. A very flexible tool we can use to accomplish this is a multilingual grammar.

A grammar is a set of structural rules which decide how words can 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.

Grammars has been used since the 1950's in the field of computer science [3, p. 4], and they have played a main role in the development of compilers where they are used to translate source code into machine readable instructions.

1.4 RELATED WORK

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

1.4.1 *Facebook graph search*

Section 1.2 describes that we want to build an application similar to Facebook graph search, which is a semantic search engine. Like most search engines, Facebook graph search takes its input as a string by using a text field. Whats special with this search engine is that it is built for extracting the semantics behind a search in order to construct machine readable instructions. More specifically, by combining natural language words and entities into a natural language phrase, Facebook graph search can translate the phrase into instructions that a machine to understand.

Machine readable instructions is something that can be executed by a CPU.

The natural language that can be understood by Facebook Graph Search is represented by a weighted context free grammar (WCFG) [2]. The grammar consists of a set of production rules which is used to extract semantic a parse tree from a natural language sentence. This parse tree 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.

1.4.1.1   *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 that 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 most probability.

1.4.1.2   *Lexical analysis*

Synonyms are supported by the grammar. Synonyms could be words or sentences. For example the sentence *people who like surfing* has the synonyms *people who surf* and *surfers*. They represent the same semantics.

Since computers normally only accept perfectly correct input when dealing with machine instructions, Facebook has 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*.

1.4.2   *Grammatical Framework*

As described previously, Facebook graph search adopts a grammar to achieve translation from a natural language to a machine language. The development of a grammar is far from straight forward and like most programmers, this project has not focused on reinventing the wheel and instead using the technology that is available. We will therefore in this section describe Grammatical Framework (GF), which is an open source functional programming language for creating grammars that can interpret natural languages [3, p. 1]. GF features a strong type system, separate abstract and concrete syntax rules and reusable libraries to facilitate design of grammars. 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[4, pp. 69-70].

Abstract syntax is a tree representation which captures the *meaning* of a sentence, and leaves out anything irrelevant. The concrete syntax represents a natural language. It describes how an abstract syntax tree is represented as a string in the natural language.

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

Because GF separates abstract and concrete syntax, one can easily add a new concrete syntax (a natural language) to an existing abstract syntax. This advantage also makes it easy to translate between many languages.

2

A SIMPLE GRAMMAR

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

2.1 ABSTRACT SYNTAX

To model the meaning of sentences, 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. We then define how our data types can take on values. This is achieved by using functions. The functions in an abstract syntax are not implemented, we can therefore only see the function declarations. The purpose for this is to allow the concrete syntaxes to choose how to represent the semantics. Two concrete syntaxes can therefore implement the same semantics differently.

In Figure 1, 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.

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.

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

```
1 abstract Instrucs = {
2   flags startcat = Instruction;
3   cat
4     Instruction ; -- An Instruction
5     Subject ;  -- The subject of an instruction
6     Relation ; -- A verb phrase
7     Object ;   -- an object
8
9   fun
10     MkInstruction : Subject -> Relation -> Instruction ;
11     People : Subject ;
12     Know : Object -> Relation ;
13     Java : Object ;
14 }
```

Figure 1: Abstract syntax

6

We can now use this abstract syntax to create an abstract syntax tree as seen in Figure 2

```
MkInstruction People (Know Java)
```

Figure 2: Abstract syntax tree

2.2 CONCRETE SYNTAX

We are now going to implement 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 3 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 strings. 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*.

```
1  concrete InstrucsEng of Instrucs = {
2    lincat
3      Instruction = Str ;
4      Subject = Str ;
5      Relation = Str ;
6      Object = Str ;
7    lin
8      MkInstruction subject relation = subject ++ "who" ++ relation ;
9      People = "people" ;
10     Know object = "know" ++ object ;
11     Java = "Java" ;
12 }
```

Figure 3: English concrete syntax

2.2.2 *Solr concrete syntax*

The final step in this example is to linearize abstract syntax into Solr concrete syntax. As Figure 4 shows, the categories are stringssec:more-instructions 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 which type a document is. 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[1] 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.

```
1 concrete InstrucsSolr of Instrucs = {
2    lincat
3      Instruction = Str ;
4      Subject = Str ;
5      Relation = Str ;
6      Object = Str ;
7
8    lin
9      MkInstruction subject relation = "q=" ++ subject ++ "AND" ++ relation ;
10     People = "object_type : Person" ;
11     Know object = "expertise : " ++ object ;
12     Java = "Java" ;
13 }
```

Figure 4: 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 not requires the user to have GF installed.

---

1 http://en.wikipedia.org/wiki/Query_string

```
$ gf InstrucsEng.gf InstrucsSwe.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 5: 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 6: Parse a string

Abstract syntax trees can be linearized into concrete syntaxes, here we linearize one abstact 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 7: 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*[2] to pass the result of the parsing as an argument to the linearizing function.

---

2 http://en.wikipedia.org/wiki/Pipeline_(Unix)

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 8: Translate between concrete syntaxes

2.4 GF RESOURCE GRAMMAR LIBRARY

The previous example is fairly easy to understand, but it is also very small. As a grammar grows to support translation of more sentences, the complexity grows as well. GF has the power to make distinctions between singular and plural forms, genders, tenses and anteriors. However, in order to correctly develop a grammar to translate these sentences one needs to have knowledge of linguistics. It is also very time consuming to implement basic morphologies over and over again. Instead, one can use GF Resource Grammar Library (RGL)[5]. The RGL contains at the time of writing grammars for 29 natural languages. These grammars includes categories and functions which can be used to represent all kinds of different words and sentences. A developer needs therefore only knowledge of her *domain* and do not need to worry about linguistic problems. By domain, we mean specific words which may have special grammatical rules, e.g. fish in English which is the same in singular as in plural form.

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 concete syntax which can be used by both English and Swedish.

Figure 9 shows the concrete syntax for English by using the RGL. Instead of just concatenating strings, we now use functions to create specific type of words and sentences. The categories are now set to be built in types that exists in the RGL and the functions are now using the RGL in order to create values of the correct types.

The most simple function in this case is `People`, which shall return a noun (N). 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` creates a noun similarly, except that it only gives `mkN` one argument. By doing this, GF applies an algorithm in order to find the plural form automati-

cally. This does however only work on regular nouns. We then create a noun phrase (NP) from the noun received. `Know` returns a relative sentence (RS). A relative sentence can for example be *who know Java*. A relative sentence is constructed by first creating a *verb phrase* (VP) from a verb and an object. This verb phrase is then used together with a constant operation `which_RP` to create a *relative clause*. We can then finally 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.

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 (CN). As CN's do not have any determiners, we have to construct a noun phrase together with the determiner `aPl_Det` in order to to only allow translation of plural forms.

```
1  concrete InstrucsEng of Instrucs = open SyntaxEng, ParadigmsEng in {
2    lincat
3      Instruction = NP ;
4      Subject = N ;
5      Relation = RS ;
6      Object = NP ;
7
8    lin
9      MkInstruction subject relation = mkNP aPl_Det (mkCN subject relation) ;
10     People = mkN "person" "people" ;
11     Know object = mkRS' (mkVP (mkV2 (mkV "know") object)) ;
12     Java = mkNP (mkPN "Java") ;
13
14   oper
15     mkRS' : VP -> RS = \vp -> mkRS (mkRCl which_RP vp) ;
16 }
```

Figure 9: English concrete syntax using the RGL

2.5 GENERALIZING THE CONCRETE SYNTAX

The English concrete syntax which uses the RGL is more complicated than the first version which only concatenates strings. In order to motivate why one shall use the RGL, 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 10. We no longer have any strings defined, as we want to keep the syntax generalised. Constant operations are used in place of strings, and they are imported from the lexicon interface LexInstrucs.

```
1  incomplete concrete InstrucsI of Instrucs = open Syntax, LexInstrucs in {
2    lincat
3      Instruction = NP ;
4      Subject = N ;
5      Relation = RS ;
6      Object = NP ;
7
8    lin
9      MkInstruction subject relation = mkNP aPl_Det (mkCN subject relation) ;
10     People = people_N ;
11     Know object = mkRS' (mkVP (know_V2 object)) ;
12     Java = java_NP ;
13
14   oper
15     mkRS' : VP -> RS = \vp -> mkRS (mkRCl which_RP vp) ;
16 }
```

Figure 10: Incomplete concrete syntax

LexInstrucs is an *interface*, which means that it only provides declarations. Figure 11 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.

```
1  interface LexInstrucs = open Syntax in {
2    oper
3      person_N : N ;
4      know_V2 : V2 ;
5      java_NP : NP ;
6 }
```

Figure 11: Lexicon interface

Figure 12 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.

```
1 instance LexInstrucsEng of LexInstrucs = open SyntaxEng, ParadigmsEng in {
2     oper
3       person_N = mkN "person" "people" ;
4       know_V2 = mkV2 (mkV "know") ;
5       java_NP = mkNP (mkPN "Java");
6 }
```

Figure 12: Lexicon instantiation of English

Figure 13 shows another instance of `LexInstrucs`, the lexicon for Swedish.

```
1 instance LexInstrucsSwe of LexInstrucs = open SyntaxSwe, ParadigmsSwe in {
2     oper
3       person_N = mkN "person" "personer" ;
4       know_V2 = mkV2 (mkV "kunna" "kan" "kunna" "kunde" "kunnat" "kunna") ;
5       java_NP = mkNP (mkPN "Java");
6 }
```

Figure 13: Lexicon instantiation of Swedish

Why can't we define `know_V2` as just `mkV2 (mkV "kan")`?

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`.

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

Figure 14: English instantiation of the incomplete concrete syntax

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

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

Figure 15: 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.

```
1 Instrucs> p -lang=InstrucsEng "people who know Java" | l -lang=InstrucsSwe
2 personer som kan Java
3
4 personer som kan Java
5
6 personer som kan Java
```

Figure 16: Swedish instantiation of the incomplete concrete syntax

Why we get three results is unknown at the moment. Whats really interesting is that we can translate Instrucs formulated in English and Swedish into Solr-syntax.

3

APPLICATION DEVELOPMENT

3.1 REQUIREMENTS SPECIFICATION

3.1.1 *Generation of mock data*

As described in Section 1.2, 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 the 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 17: All sentences supported by the application

Two more cases has been added to instructions regarding *people*. In addition, two new type of instructions has been added, translations about *customers* and *projects*. Note that Figure 17 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 [6, p. 41]. This project, however, does not.

GF has the power to suggest valid words of an incomplete sentence by using incremental parsing [7]. 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 it also inflexible since it does not support the use of words outside the grammar, e.g. `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 and indexing them in Apache Solr. This makes it very easy to search on matching sentences, we also gain powerful techniques such as approximate string matching.

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 retrieve the most relevant instructions.

As the suggestion engine uses a search platform, it is possible to type anything 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 adopts a specific development platform, Java Enterprise Edition

(Java EE). 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 make 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 make use of an open source tool called Apache Maven to handle dependencies. One simply list 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 gui[1].

To summarize, this application is a web application built in Java EE and uses Apache Maven for library management.

3.1.7   Running the application

Web applications built in Java are usually has 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 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.

---

1 http://en.wikipedia.org/wiki/Graphical_user_interface

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, it forces us to update the grammar every time we want it to support a new programming language. 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

```
1 -- Abstract syntax
2 MkObject : String -> Object ;
3 -- RGL implementation
4 MkObject str = mkNP (mkPN str.s) ; -- PN = Proper Name
5 -- Solr implementation
6 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. GF cannot create a proper name by using an arbitrary string. Why?

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.

```
1 -- Abstract syntax
2 MkObject : Symb -> Object ;
3 -- RGL implementation
4 MkObject symb = mkNP (SymbPN symb) ; -- PN = Proper Name
5 -- Solr implementation
6 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.

3.2.3   *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

The first step towards a solution to this problem is to modify the 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.

```
1  -- Instructions.gf
2  cat
3    Internal ;
4    External ;
5    Resource ;
6  fun
7    People   : Internal ;
8    Customer : External ;
9    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 values 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.

```
1 cat
2   InternalRelation ;
3   ExternalRelation ;
4   ResourceRelation ;
5 fun
6   Know     : Object -> InternalRelation ;
7   UseExt   : Object -> ExternalRelation ;
8   UseRes   : Object -> ResourceRelation ;
9   WorkIn   : Object -> InternalRelation ;
10  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. The function MkInstruction is replaced by three new functions: InstrucInternal, InstrucExternal and InstrucResource. However, as we do not need to make any distinction between different type of instructions at this level, all three functions returns a value of the type Instruction.

```
1 cat
2   Instruction ;
3 fun
4   InstrucInternal : Internal -> InternalRelation -> Instruction ;
5   InstrucExternal : External -> ExternalRelation -> Instruction ;
6   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 has 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.

```
1 lincat
2   Instruction = NP ;
3   Internal, External, Resource = N ;
4   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.

```
1 lin
2   People = person_N ;
3   Customer = customer_N ;
4   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.

```
1 lin
2   Know object = mkRS' (mkVP know_V2 object) ;
3   UseExt object = mkRS' (mkVP use_V2 object) ;
4   UseRes object = mkRS' (mkVP use_V2 object) ;
5   WorkIn object = mkRS' (mkVP (mkV2 work_V in_Prep) object) ;
6   WorkWith object = mkRS' (mkVP (mkV2 work_V with_Prep) object) ;
7
8 oper
9   -- Make a relative sentence
10  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 mkQ has been defined in order to reuse code.

```
1 lin
2   InstrucInternal internal relation = mkI internal relation ;
3   InstrucExternal external relation = mkI external relation ;
4   InstrucResource resource' relation = mkI resource' relation ;
5
6 oper
7   mkI : N -> RS -> NP = \noun,rs -> mkNP aPl_Det
8                              (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.

```
1 lincat
2   Instruction = Str ;
3   Internal, External, Resource = Str ;
4   InternalRelation, ExternalRelation, ResourceRelation = Str ;
5   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.

```
1 lin
2   People = "Person" ;
3   Customer = "Organization" ;
4   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.

```
1 lin
2   Know obj = "KNOWS" ++ ":" ++ obj ;
3   UseExt obj = "USES" ++ ":" ++ obj ;
4   UseRes obj = "USES" ++ ":" ++ obj ;
5   WorkWith obj = "WORKS_WITH" ++ ":" ++ obj ;
6   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.

```
1 lin
2   InstrucInternal internal relation = select internal relation ;
3   InstrucExternal external relation = select external relation;
4   InstrucResource resource' relation = select resource' relation;
5
6 oper
7   select : Str -> Str -> Str = \subj,relation ->
8                   "select?q=*:*&wt=json&fq=" ++ "object_type :"
9                   ++ 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.

```
1 lin
2   And_O : Object -> Object -> Object ;
3   Or_O : Object -> Object -> Object ;
```

Figure 31: Abstract syntax for boolean operators and objects

The RGL implementation is shown in Figure 32.

```
1 lin
2   And_O o1 o2 = mkNP and_Conj o1 o2 ;
3   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.

```
1 lin
2   And_O o1 o2 = "(" ++ o1 ++ "AND" ++ o2 ++ ")" ;
3   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 relationship 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.

```
1  lin
2    InternalAnd : InternalRelation -> InternalRelation -> InternalRelation ;
3    InternalOr : InternalRelation -> InternalRelation -> InternalRelation ;
4
5    ExternalAnd : ExternalRelation -> ExternalRelation -> ExternalRelation ;
6    ExternalOr : ExternalRelation -> ExternalRelation -> ExternalRelation ;
7
8    ResourceAnd : ResourceRelation -> ResourceRelation -> ResourceRelation ;
9    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.

```
1  lin
2    InternalAnd rs1 rs2 = mkRS and_Conj rs1 rs2 ;
3    InternalOr rs1 rs2 = mkRS or_Conj rs1 rs2 ;
4
5    ExternalAnd rs1 rs2 = mkRS and_Conj rs1 rs2 ;
6    ExternalAnd rs1 rs2 = mkRS or_Conj rs1 rs2 ;
7
8    ResourceAnd rs1 rs2 = mkRS and_Conj rs1 rs2 ;
9    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.

```
1  lin
2    InternalAnd s1 s2 = "(" ++ s1 ++ "AND" ++ s2 ++ ")";
3    InternalOr s1 s2 = "(" ++ s1 ++ " OR " ++ s2 ++ ")";
4
5    ExternalAnd s1 s2 = "(" ++ s1 ++ "AND" ++ s2 ++ ")";
6    ExternalOr s1 s2 = "(" ++ s1 ++ " OR " ++ s2 ++ ")";
7
8    ResourceAnd s1 s2 = "(" ++ s1 ++ "AND" ++ s2 ++ ")";
9    ResourceOr s1 s2 = "(" ++ s1 ++ " OR " ++ s2 ++ ")";
```

Figure 36: Solr concrete syntax for boolean operators and relations

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 creates 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 of 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 58 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, and the latter has maximum 5 edges between root and leaf.
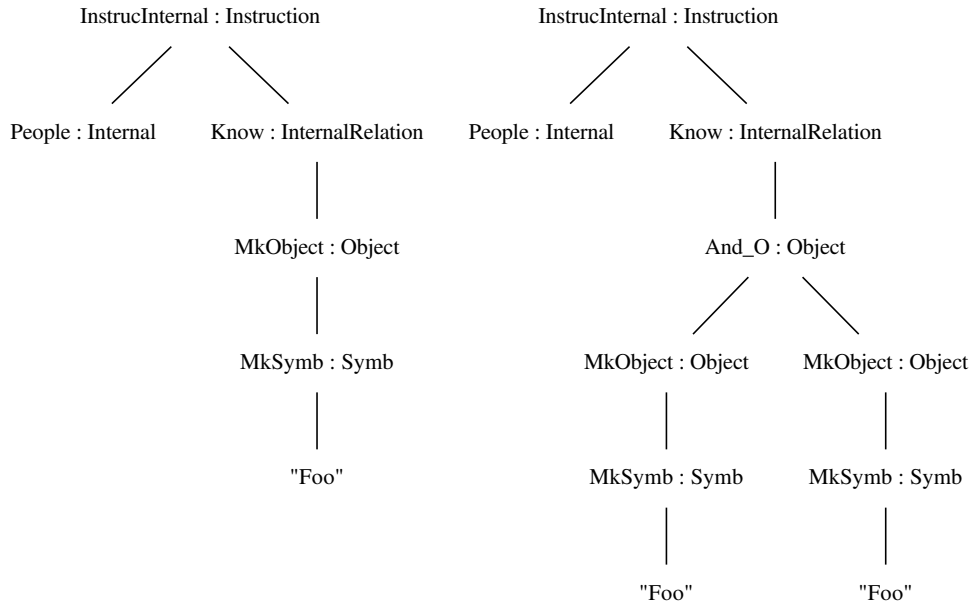
InstrucInternal : Instruction

People : Internal   Know : InternalRelation

MkObject : Object

MkSymb : Symb

"Foo"

InstrucInternal : Instruction

People : Internal   Know : InternalRelation

And_O : Object

MkObject : Object   MkObject : Object

MkSymb : Symb   MkSymb : Symb

"Foo"   "Foo"

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. Re-

trieve 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 `Work_With` relation. Locations are used with the `Work_In` 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 generate 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.

```
1  suggestions(sentence) {
2      // sentence = "persons who knew java or python and work in London"
3      // names[] = {Java, Python, London}
4      names[] = extractNames(sentence);
5
6      // "persons who knew java or python and work in London" ===>
7      // "persons who knew Skill0 or Skill1 and work in Location0"
8      sentence = replaceNamesWithTypes(sentence, names);
9
10     // suggestions =
11     //   { "people who know Skill0 or Skill1 and who work in Location0", ... }
12     suggestions[] = findSentences(sentence);
13
14     foreach suggestion in suggestions {
15         // "people who know Skill0 or Skill1 and who work in Location0" ===>
16         // "people who know Java or Python and who work in London"
17         suggestion = restoreNames(names, suggestion);
18     }
19     return suggestions;
20 }
```

Figure 42: Pseudocode for the suggestion algorithm

3.3.2 *Supporting invalid sentences*

This section aims to add better accuracy of the suggestion engine by linearizing different variations of a sentence. While the linearization function in GF returns a string representation of a tree, there also exists a parameter `-list` which allows GF to return all semantically equivalent ways of expressing a sentence.

```
Languages: InstrucsEng
Instrucs> p "people who know Java" | l -list
people who know Java, people's who know Java
```

Figure 43: Linearization to InstrucsEng using `-list`

Here we actually get four results, three duplicates! Fix, but how?

Figure 45 shows that `people who know Java` actually can be expressed in two ways. We add both linearizations as a list to the same object in the linearizations index, and we list the one we think is the most suitable in the beginning of the list. This makes it possible to search for everything the grammar can parse, but get a more suitable suggestion. For instance, if type *people's who know Java* then the application will match the lesser suitable sentence in the index, but return *people who know Java* as a suggestion.

It is possible to add more linearizations than those listed in Figure 45. 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`. The reason why the plural version is chosen as the preferable sentence is because `aPl_Det` is listed before `aSg_Det`.

```
1 oper
2   mkI : N -> RS -> NP = \noun_N,rs_RS -> mkNP (aPl_Det | aSg_Det)
3                           (mkCN noun_N rs_RS) ;
```

Figure 44: Treating singular form equivalent to plural form

The second 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.

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

Figure 45: 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 clearly is incorrect. 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 incorrect instructions.

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.

```
1 lincat
2   Instruction = Str ;
3   Internal, External, Resource = Str ;
4   InternalRelation, ExternalRelation, ResourceRelation = Str ;
5   Skill, Organization, Location, Module = Str ;
```

Figure 46: 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*.

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

Figure 47: Concrete syntax for English by concatenating strings with implementation of subjects

As with subjects, variance is also used with relations.

```
1 lin
2   Know_R obj = ("know" | "knows") ++ obj;
3   UseExt_R obj = ("use" | "uses") ++ obj ;
4   UseRes_R obj = ("use" | "uses") ++ obj ;
5   WorkWith_R obj = ("work with" | "works with") ++ obj ;
6   WorkIn_R obj = ("work in" | "works in") ++ obj ;
```

Figure 48: 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*.

```
1 lin
2 InstrucInternal internal relation = internal ++
3                              ("who" | "which" | "that") ++ relation ;
4 InstrucExternal external relation = external ++
5                              ("who" | "which" | "that") ++ relation ;
6 InstrucResource resource' relation = resource' ++
7                              ("which" | "that") ++ relation ;
```

Figure 49: Concrete syntax for English by concatenating strings with implementation of instructions

A name is just a string.

```
1 lin
2 MkSkill s = s.s ;
3 MkOrganization s = s.s ;
4 MkLocation s = s.s ;
5 MkModule s = s.s ;
```

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

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

```
1  lin
2  And_S s1 s2 = s1 ++ "and" ++ s2 ;
3  Or_S s1 s2 = s1 ++ "or" ++ s2 ;
4
5  And_O s1 s2 = s1 ++ "and" ++ s2 ;
6  Or_O s1 s2 = s1 ++ "or" ++ s2 ;
7
8  And_L s1 s2 = s1 ++ "and" ++ s2 ;
9  Or_L s1 s2 = s1 ++ "or" ++ s2 ;
10
11 And_M s1 s2 = s1 ++ "and" ++ s2 ;
12 Or_M s1 s2 = s1 ++ "or" ++ s2 ;
```

Figure 51: 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.

```
1 lin
2   InternalAnd s1 s2 = s1 ++ "and" ++ s2 ;
3   InternalOr s1 s2 = s1 ++ "or" ++ s2 ;
4
5   ExternalAnd s1 s2 = s1 ++ "and" ++ s2 ;
6   ExternalOr s1 s2 = s1 ++ "or" ++ s2 ;
7
8   ResourceAnd s1 s2 = s1 ++ "and" ++ s2 ;
9   ResourceOr s1 s2 = s1 ++ "or" ++ s2 ;
```

Figure 52: 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 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 53 shows an example of data that could exists in the Solr index.

```
1  { // Customer
2    "name" : "Amnesty"
3    "object_type" : "Customer",
4    "USES" : ["Solr", "Tomcat", "GF"]
5  },
6  { // Person
7    "name" : "Jane Doe"
8    "object_type" : "Person",
9    "KNOWS" : ["Java", "Python", "C"],
10   "WORKS_IN" : ["London", "Gothenburg"],
11   "WORKS_WITH" :  ["Amnesty", "Unicef"]
12 },
13 { // Project
14   "object_type" : "Project",
15   "USES" : ["Android", "GF"]
16 }
```

Figure 53: 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 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 first five figures shows the most simplest translations. They include all subjects and all types of names. Each one contains a subject, one verb and one name of a certain type.

Figure 54 shows that the word Java is of the type Skill.

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

```
1  [
2   {
3    'ast': 'InstrucInternal People (Know_R (MkSkill (MkSymb "Java")))',
4    'linearizations': [
5     {
6      'query': 'people who know Java',
7      'language': 'InstrucsEngConcat'
8     },
9     {
10      'query': 'people who know Java',
11      'language': 'InstrucsEngRGL'
12     },
13     {
14      'query': 'select?q=*:*&wt=json&fq= object_type : Person AND KNOWS : ( Java )',
15      'language': 'InstrucsSolr'
16     },
17     {
18      'query': 'personer som kan Java',
19      'language': 'InstrucsSweRGL'
20     }
21    ]
22   }
23  ]
```

Figure 54: Translation including people and a name of type Skill

Figure 55 shows that the word London is of the type Location.

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

```
1  [
2    {
3      'ast': 'InstrucInternal People (WorkIn_R (MkLocation (MkSymb "London")))',
4      'linearizations': [
5        {
6          'query': 'people who work in London',
7          'language': 'InstrucsEngConcat'
8        },
9        {
10         'query': 'people who work in London',
11         'language': 'InstrucsEngRGL'
12       },
13       {
14         'query': 'select?q=*:*&wt=json&fq= object_type : Person AND WORKS_IN : ( London )',
15         'language': 'InstrucsSolr'
16       },
17       {
18         'query': 'personer som arbetar i London',
19         'language': 'InstrucsSweRGL'
20       }
21     ]
22   }
23 ]
```

Figure 55: Translation including people and a name of type Location

Figure 56 shows that the word Amnesty is of the type Organization.

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

```
 1  [
 2    {
 3      'ast': 'InstrucInternal People (WorkWith_R (MkOrganization (MkSymb "Amnesty")))',
 4      'linearizations': [
 5        {
 6          'query': 'people who work with Amnesty',
 7          'language': 'InstrucsEngConcat'
 8        },
 9        {
10          'query': 'people who work with Amnesty',
11          'language': 'InstrucsEngRGL'
12        },
13        {
14          'query': 'select?q=*:*&wt=json&fq= object_type : Person AND WORKS_WITH : ( Amnesty )',
15          'language': 'InstrucsSolr'
16        },
17        {
18          'query': 'personer som arbetar med Amnesty',
19          'language': 'InstrucsSwe'
20        }
21      ]
22    }
23  ]
```

Figure 56: Translation including `people` and a name of type `Organization`

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

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

```
1  [
2    {
3      'ast': 'InstrucExternal Customer (UseExt_R (MkModule (MkSymb "Solr")))',
4      'linearizations': [
5        {
6          'query': 'customers who use Solr',
7          'language': 'InstrucsEngConcat'
8        },
9        {
10         'query': 'customers who use Solr',
11         'language': 'InstrucsEngRGL'
12       },
13       {
14         'query': 'select?q=*:*&wt=json&fq= object_type : Organization AND USES : ( Solr )',
15         'language': 'InstrucsSolr'
16       },
17       {
18         'query': 'kunder som använder Solr',
19         'language': 'InstrucsSwe'
20       }
21     ]
22   }
23 ]
```

Figure 57: Translation including `customer` and a name of type `Module`

Similarly, also **??** shows that the word `Solr` is of the type `Module`.

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

```
1  [
2    {
3      'ast': 'InstrucResource Project (UseRes_R (MkModule (MkSymb "Solr")))',
4      'linearizations': [
5        {
6          'query': 'projects who use Solr',
7          'language': 'InstrucsEngConcat'
8        },
9        {
10         'query': 'projects who use Solr',
11         'language': 'InstrucsEngRGL'
12       },
13       {
14         'query': 'select?q=*:*&wt=json&fq= object_type : Project AND USES : ( Solr )',
15         'language': 'InstrucsSolr'
16       },
17       {
18         'query': 'projekt som använder Solr',
19         'language': 'InstrucsSwe'
20       }
21     ]
22   }
23 ]
```

Figure 58: Translation including `project` and a name of type `Module`

Figure 59 shows how the applications handles the boolean operator *and*. The application handles the case for *or* similarly.

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

```
1  [
2    {
3      'ast': 'InstrucInternal People (Know_R (And_S (MkSkill (MkSymb "Java"))
4                                                     (MkSkill (MkSymb "Python"))))',
5      'linearizations': [
6        {
7          'query': 'people who know Java and Python',
8          'language': 'InstrucsEngConcat'
9        },
10       {
11         'query': 'people who know Java and Python',
12         'language': 'InstrucsEngRGL'
13       },
14       {
15         'query': 'select?q=*:*&wt=json&fq= object_type : Person AND
16                                          KNOWS : ( ( Java ) AND ( Python ) )',
17         'language': 'InstrucsSolr'
18       },
19       {
20         'query': 'personer som kan Java och Python',
21         'language': 'InstrucsSwe'
22       }
23     ]
24   }
25 ]
```

Figure 59: Translation including `people` and two names of the type `Skill`

Figure 60 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
```

```
 1  [
 2    {
 3      'ast': 'InstrucInternal People (Know_R (Or_S (And_S (MkSkill (MkSymb "Java"))
 4                          (MkSkill (MkSymb "Python"))) (MkSkill (MkSymb "Haskell"))))',
 5      'linearizations': [
 6        {
 7          'query': 'people who know Java and Python or Haskell',
 8          'language': 'InstrucsEngConcat'
 9        },
10        {
11          'query': 'people who know Java and Python or Haskell',
12          'language': 'InstrucsEngRGL'
13        },
14        {
15          'query': 'select?q=*:*&wt=json&fq= object_type : Person AND
16                        KNOWS : ( ( ( Java ) AND ( Python ) ) OR ( Haskell ) )',
17          'language': 'InstrucsSolr'
18        },
19        {
20          'query': 'personer som kan Java och Python eller Haskell',
21          'language': 'InstrucsSwe'
22        }
23      ]
24    },
25    {
26      'ast': 'InstrucInternal People (Know_R (And_S (MkSkill (MkSymb "Java"))
27                          (Or_S (MkSkill (MkSymb "Python")) (MkSkill (MkSymb "Haskell")))))',
28      'linearizations': [
29        {
30          'query': 'people who know Java and Python or Haskell',
31          'language': 'InstrucsEngConcat'
32        },
33        {
34          'query': 'people who know Java and Python or Haskell',
35          'language': 'InstrucsEngRGL'
36        },
37        {
38          'query': 'select?q=*:*&wt=json&fq= object_type : Person AND
39                        KNOWS : ( ( Java ) AND ( ( Python ) OR ( Haskell ) ) )',
40          'language': 'InstrucsSolr'
41        },
42        {
43          'query': 'personer som kan Java och Python eller Haskell',
44          'language': 'InstrucsSwe'
45        }
46      ]
47    }
48  ]
```
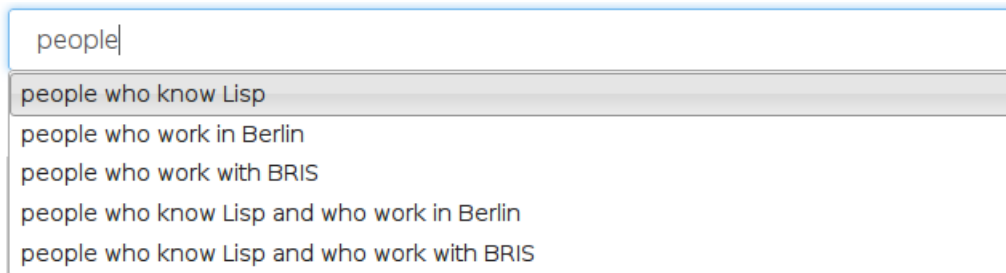
Figure 60: Translation of an ambiguous instruction involving `people` and three names of the type `Skill`

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.
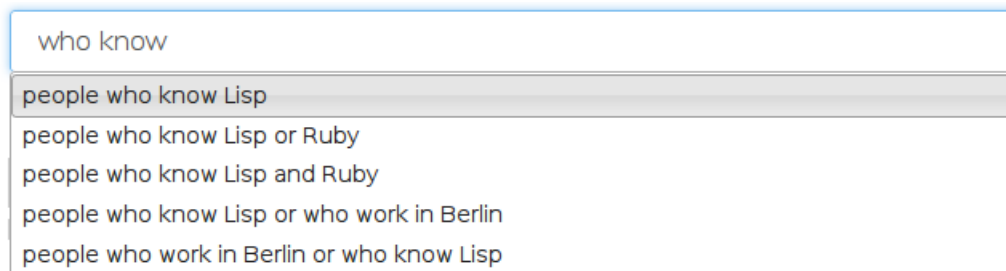
Figure 61 shows suggestions based on the first word in a valid sentence.

Figure 61: Suggestions based on the word `people` by using EnglishRGL or English-Concat

Figure 62 shows that it is possible to use a combination of words to retrieve suggestions. It also demonstrates that the words do not have to be the first words in an instruction.

Figure 62: Suggestions based on the words `who know` by using EnglishRGL or English-Concat

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

Figure 63: Suggestions based on a name of type Skill by using EnglishRGL or English-Concat

Figure 64: Suggestions based on a name of type Location by using EnglishRGL or EnglishConcat

Figure 65: Suggestions based on a name of type Organization by using EnglishRGL or EnglishConcat

Figure 66: Suggestions based on a name of type Module by using EnglishRGL or EnglishConcat

Figure 67 shows suggestions based on two names of the same type.

Figure 67: Suggestions based on two names of type Skill by using EnglishRGL or EnglishConcat

Figure 68 shows that names do not have to be of the same type to suggest relevant instructions.

Figure 68: Suggestions based on two names of different types by using EnglishRGL or EnglishConcat

Figure 69 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.

Figure 69: Synonyms based on a synonym by using EnglishRGL or EnglishConcat

Also Figure 70 shows how we can get suggestions based on a synonym. However, only EnglishConcat gives suggestions based on the word *that*.

Figure 70: Synonyms based on a synonym by using EnglishConcat

Figure 71 shows how the application shows relevant suggestion based on a misspelled word.

Figure 71: Suggestions based on a misspelled word by using EnglishRGL or English-Concat

Figure 72 and Figure 73 show how the same word can obtain different suggestions by using the concrete languages EnglishRGL and EnglishConcat.

Figure 72: Suggestions based on the string *projects* by using EnglishRGL

Figure 73: Suggestions based on the string *projects* by using EnglishConcat

Lastly, Figure 74 demonstrates that the application also can translate give suggestions in Swedish. The application can translate valid Swedish sentences as long as the user has chosen Swedish as input language in the application. All instructions that can be translated from English can also be translated from Swedish.

Figure 74: Suggestions in Swedish by using SwedishRGL

CONCLUSION

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 in programming. The two syntaxes supports equally many abstract syntax trees, but as we have added more variations of linearizations to EngConcat, one can represent a few abstract syntax trees in more ways in EngConcat than in EngRGL. One example is *people **that** know Java*. We have not found out how this can be express by using the RGL.

In addition, concatenation of strings make it possible to create all kind of different (and strange) linearizations with the variance function. For instance, we can add all tenses and make it possible to map sentences expressed in bad grammar into a correct suggestion. Therefore, users with little knowledge of writing in English have a higher probability of finding the correct sentence by using this approach.

In Chapter 3, we wanted the application to support the sentence *people who know Java and work in London*, however, we did not manage to create this sentence by using the RGL. Instead we had to content ourselves with *people who know Java and **who** work in London*. This problem could easily be solved by using concatenation of strings.

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 accout that this rule is only valid

47

if the subject refers to a group of humans, it applies the rule to any group of things.

5.3 KNOWN ISSUES

Multiple equal abstract syntax trees

The resulting application always presents distinct abstract syntax trees if there exists more than one in the results of a parsing. This is because the application processes the result and deletes any duplicate abstract syntax trees with its linearizations. GF actually produces 6 equal abstract syntax trees if translating from EngRGL or SweRGL. The PGF-implementation for Java produces 2 equal abstract syntax trees if translating from EngRGL or SweRGL. EngConcat however does not produce any duplicated abstract syntax trees.

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` can sometimes 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.

5.4 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 now 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 typ 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 improvment of the suggestion engine is to automatically choose the first suggestion available if the user tries to translate an invalid instruction. By doing so, one will always translate a correct sentence.

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 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 resulting application will show all resulting abstract syntax trees with their corresponding linearizations. A better program would ask the user to clarify the instruction or make choose one of the astract 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.

Rewrite grammar with lexicon

Not finished! Instead of storing names in solr, store them in a lexicon and find a way of parsing using a wide coverage grammar.

A

GF SHELL, RUNTIME SYSTEMS AND LIBRARIES

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

A.1 GF SHELL

The GF shell 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`. Cabal is available if the haskell platform is installed on a system. The haskell platform can be installed in 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 [8, 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 an simple and convenient syntax. Interpreting the GF programming language directly is however a heavy operation[8][p. 13], especially with larger grammars. This is where the Portable Grammar Format (PGF)[8][p. 14] comes in. PGF is a custom made machine language which is dynamically 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 generate a few libraries which is 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 and at the time of writing, no pre-generated libraries exists. We therefore need to generate the libraries by ourselves. We will start by generating and installing the C-libraries. We will then go through how we can generate 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. This step assumes that you have Eclipse installed with the CDT-plugin. 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.

Unfortunately, the build-configuration for the jpgf-project is not complete at time of writing. We therefore need to make additional adjustments in order to build the project.

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 cannot 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 can be downloaded from http://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 also can 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. 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 make 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. Excecute the following:

```
$ cd <project_directory>/mock-data
$ 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] Marissa Mayer. Seattle Conference on Scalability: Scaling Google for Every User. https://www.youtube.com/watch?v=Syc3axgRsBw, 2007. [Online; accessed 1-July-2014].

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

[3] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

[4] 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.

[5] Aarne Ranta. The GF resource grammar library. *Linguistic Issues in Language Technology*, 2009.

[6] Krasimir Angelov, Björn Bringert, and Aarne Ranta. Speech-enabled hybrid multilingual translation for mobile devices. *EACL 2014*, page 41, 2014.

[7] Krasimir Angelov. Incremental parsing with parallel multiple context-free grammars. In *European Chapter of the Association for Computational Linguistics*, 2009.

[8] Krasimir Angelov. *The Mechanics of the Grammatical Framework*. Chalmers University of Technology, Göteborg, 2011. ISBN-13: 978-91-7385-605-8.