CONTENTS

ACRONYMS

GF Grammatical Framework

RGL Resource Grammar Library

Java EE Java Enterprise Edition

PGF Portable Grammar Format

1 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. 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 [2, 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 GRAMMATICAL FRAMEWORK

Grammatical Framework (GF) is a functional programming language for creating grammars for natural languages.[2, 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 easily see that GF is very much based on the theory of programming languages. GF adopts the use of abstract and concrete syntax in the same sense as in compiler theory.

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.

1.4.1 *A simple example*

The following section 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. We will explain more about Solr query language later in the thesis.

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.

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 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`.

```
1  abstract Instructions = {
2    flags startcat = Instruction;
3    cat
4      Instruction ; -- A Instruction
5      Subject ;  -- The subject of a 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

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

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.

English concrete syntax

Figure 3 shows the implementation of the concrete syntax for English. Categories are linearized by the keyword `lincat`. 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*.

```
concrete InstructionsEng of Instructions = {
  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 3: English concrete syntax

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 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 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[2] 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.

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

```
concrete InstructionsSolr of Instructions = {
    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 4: Solr concrete syntax

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 and Java. These libraries can be used to build a translation application which not requires the user to have GF installed.

```
$ gf InstructionsEng.gf InstructionsSwe.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 Instructions.gf...   write file Instructions.gfo
- compiling InstructionsEng.gf...   write file InstructionsEng.gfo
- compiling InstructionsSolr.gf...   write file InstructionsSolr.gfo
linking ... OK
 Languages: InstructionsEng InstructionsSolr
Instructions>
```

Figure 5: GF shell prompt

A string can be parsed into an abstract syntax tree.

```
Instructions> parse -lang=InstructionsEng "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.

```
Instructions> 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 `InstructionsEng` into `InstructionsSolr`. We use a *pipeline*[3] to pass the result of the parsing as an argument to the linearizing

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

function. Note how we use `p` instead of `parse` and `l` instead of `linearize`. They are just shorthands of their longer representations.

Note: We shortened the name of the concrete syntaxes here so they could fit in the box.

```
Instructions> p -lang=Eng "people who know Java" | l -lang=Solr
q= object_type : Person AND expertise : Java
```

Figure 8: Translate between concrete syntaxes

1.5 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). 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.

1.5.1 *Example usage of 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 GF cannot create a noun with only plural form.

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 automatically. This does however only work on regular nouns. `Know` returns a verb

phrase (VP). We use a VP to combine a verb with a noun in order to create a `Relation`, as seen in lines 16-19.

How to combine a `Subject` and a `Relation` into a `Instruction` can be seen in lines 9-14. We first create a relative clause (RCl) by combining a built in constant `which_RP` with the verb phrase (`relation`). There is however no built in operation available to combine a RCl with a noun, so we must create a relative sentence (RS). The RS is then combined with the subject by creating a common noun (CN). In order to only allow translations of plural forms we create a noun phrase with `aPl_Det` as determiner.

```
1  concrete InstructionsEng of Instructions = open SyntaxEng, ParadigmsEng in {
2    lincat
3      Instruction = NP ;
4      Subject = N ;
5      Relation = VP ;
6      Object = NP ;
7
8    lin
9      MkInstruction subject relation = mkNP aPl_Det
10                                          (mkCN subject
11                                              (mkRS
12                                                  (mkRCl which_RP relation)
13                                              )
14                                          ) ;
15     People = mkN "person" "people" ;
16     Know object = mkVP
17                       (mkV2
18                           (mkV "know")
19                       ) object ;
20     Java = mkNP (mkN "Java") ;
21 }
```

Figure 9: English concrete syntax using the RGL

1.5.2 *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 `LexInstructions`.

```
incomplete concrete InstructionsI of Instructions = open Syntax, LexInstructions in {
    lincat
      Instruction = NP ;
      Subject = N ;
      Relation = VP ;
      Object = NP ;

    lin
      MkInstruction subject relation = mkNP aPl_Det
                                          (mkCN subject
                                              (mkRS
                                                  (mkRCl which_RP relation)
                                              )
                                          ) ;
      People = person_N ;
      Know object = mkVP know_V2 object ;
      Java = java_NP ;
}
```

Figure 10: Incomplete concrete syntax

`LexInstructions` 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.

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

Figure 11: Lexicon interface

Figure 12 shows how the operations defined in `LexInstructions` are implemented in `LexInstructionsEng`, we represent the words in the same way as in the old version of the concrete syntax for English.

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

Figure 12: Lexicon instantiation of English

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

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

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 `InstructionsI` is instantiated as `InstructionsEng`. Note how we override `Syntax` with `SyntaxEng` and `LexInstructions` with `LexInstructionsEng`.

```
concrete InstructionsEng of Instructions = InstructionsI with
                                            (Syntax = SyntaxEng),
                                            (LexInstructions = LexInstructionsEng)
                                            ** open ParadigmsEng in {}
```

Figure 14: English instantiation of the incomplete concrete syntax

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

```
concrete InstructionsSwe of Instructions = InstructionsI with
                                            (Syntax = SyntaxSwe),
                                            (LexInstructions = LexInstructionsSwe)
                                            ** open ParadigmsSwe in {}
```

Figure 15: Swedish instantiation of the incomplete concrete syntax

If we load the GF-shell with `InstructionsEng.gf` and `InstructionsSwe.gf` we can make the following translation from English to Swedish.

```
Instructions> p -lang=InstructionsEng "people who know Java" | l -lang=InstructionsSwe
personer som kan Java

personer som kan Java

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 instructions formulated in English and Swedish into Solr-syntax.

2 APPLICATION DEVELOPMENT

2.1 REQUIREMENTS SPECIFICATION

2.1.1 *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. Although Findwise has an index with information about employees, projects and customers, their information 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.

2.1.2 *Grammar development*

The grammar in section 1 can only translate the question `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`. In addition, it also needs to support other questions involving not only people, but customers and projects.

2.1.3 *Suggestions*

If a user have no idea of which questions the application can translate, how can she use the application? GF can only parse syntactically correct input, therefore, if the question has one character in the wrong place it will not be able to translate anything.

A suggestion engine can be used to complete the input of the user. For instance, if a user types `'All persons that know Java'` the application can display a list of related valid sentences where the user can choose which questions that she wants to replace the original sentence with.

To make the application more flexible, a better suggestion engine shall be able to complete sentences based purely on keywords, for example `'people java'` shall suggest sentences that can be formulated with these two words.

2.1.4 *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.

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 a tool called 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.

Some text about tomcat

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

2.2 GENERATION OF MOCK DATA

2.3 GRAMMAR DEVELOPMENT

2.4 SUGGESTION ENGINE

A

GF RUNTIME SYSTEMS AND LIBRARIES

A.1 GF RUNTIME SYSTEMS

While the GF-shell is a powerful tool, it is not very convenient to interact with when programming an application. It requires the hosting computer to have installed and also requires the application to have access to executing other applications, a major drawback. Luckily, the creators of GF has thought about this and built embeddable runtime systems for a few programming languages [1, 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. The main purpose is because it is used by many developers and companies in the industry.

The GF-shell uses the Haskell runtime system.

A.1.1 *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[1][p. 13], especially with larger grammars. This is where the Portable Grammar Format (PGF)[1][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.

A.1.2 *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 to generate the wrapper library.

A.1.2.1 *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
```

The C-runtime for PGF is now installed.

A.1.3 *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`. 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 Ubuntu 14.04 amd64 with OpenJDK 7, hence the correct directory is

```
/usr/lib/jvm/java-7-openjdk-amd64/include
```

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 some dependencies are missing, try to locate the files and move them to `/usr/local/lib` (or `/usr/lib` in some distros).

The last step is to move `libjpgf.so` into the correct directory.

```
mv libjpgf.so /usr/local/lib (or /usr/lib)
```

You have now installed the wrapper library.

A.1.4 *Using the Java-runtime*

Don't forget how to set java lib path when using tomcat!

BIBLIOGRAPHY

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

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