CONTENTS

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[3]. 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 [4, 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 an open source functional programming language for creating grammars for natural languages.[4, 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[1, 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.

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`.

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

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

Figure 1: Abstract syntax

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

Figure 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.

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

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

Figure 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.

2 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*[3] to pass the result of the parsing as an argument to the linearizing function.

3 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

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. 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.

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

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

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

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 `LexInstrucs`.

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

  lin
    MkInstruction subject relation = mkNP aPl_Det (mkCN subject relation) ;
    People = people_N ;
    Know object = mkRS' (mkVP (mkV2 know_V object)) ;

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

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.

```
interface LexInstrucs = 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 `LexInstrucs` are implemented in `LexInstrucsEng`, we represent the words in the same way as in the old version of the concrete syntax for English.

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

Figure 12: Lexicon instantiation of English

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

```
instance LexInstrucsSwe of LexInstrucs = 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 `InstrucsI` is instantiated as `InstrucsEng`. Note how we override `Syntax` with `SyntaxEng` and `LexInstrucs` with `LexInstrucsEng`.

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

Figure 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.

```
concrete InstrucsSwe of Instrucs = InstrucsI with
                                             (Syntax = SyntaxSwe),
                                             (LexInstrucs = LexInstrucsSwe)
                                             ** 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.

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

2.1.2 *Grammar development*

The grammar in Section 1.4.1 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 has 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.

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.

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.

2.1.4.1 *Running the application*

Web applications built in Java are usually has the WAR file format. It a special JAR-file which includes classes, dependencies and webpages. 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 each request.

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

2.2 GENERATION OF MOCK DATA

This section describes how the data used by the application is generated. In order to generate data, one must know what to generate. In this context, this means that

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

2.3 GRAMMAR DEVELOPMENT

This section continues the work on the grammar introduced in Section 1.4.1.

2.3.1 *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.

2.3.2 *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

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

Figure 17: 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.

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

Figure 18: Working approach on names

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

2.3.3 *More questions*

We have only covered translation of one sentence in the application so far. Figure 19 shows all sentences that the end application supports.

```
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 19: 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 19 only shows instances of instructions. As described in the previous section, the grammar uses names that can take on any string for specific parts of the sentences. This means that the words *Java*, *London*, *Unicef* and *Solr* can be exchanged into anything.

2.3.4 *Extending the grammar*

It is not trivial to extend the grammar to support the instructions described in the previous section.. One has to take into account that it shall not be possible to translate invalid instructions like *projects who work in London*.

2.3.4.1 *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.

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

Figure 20: 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. Figure 21 also shows the new supported relations.

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

Figure 21: 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`.

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

Figure 22: New functions for instructions

2.3.4.2 *Concrete syntax for 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.

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

Figure 23: New category implementations

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

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

Figure 24: Implementation of subjects

Four new relation functions are added. Line 5-6 in Figure 30 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`.

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

Figure 25: Implementation of relations

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.

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

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

Figure 26: Implementation of combining functions

2.3.4.3 *Concrete syntax for Solr*

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

The new categories are all defined as strings.

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

Figure 27: Implementation of combining functions

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

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

Figure 28: Implementation of combining functions

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

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

Figure 29: Implementation of combining functions

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

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

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

Figure 30: Implementation of combining functions

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 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.

We begin by extending the abstract syntax. In addition to the previous example with the boolean operator *and*, we will also add support for the boolean operator *or*. As seen in Figure 31, two new functions are defined to handle these cases, one for each operator.

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

Figure 31: Implementation of combining functions

The RGL implementation is shown in Figure 35.

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

Figure 32: Implementation of combining functions

The Solr implementation is shown in **??**.

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

Figure 33: Implementation of combining functions

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.

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

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

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

Figure 34: Implementation of combining functions

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

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

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

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

Figure 35: Implementation of combining functions

Don't forget to change actual source code (files) to the same as the report. I've just changed the report to support AndOr with relations

2.5 SUGGESTION ENGINE

3 DISCUSSION

3.1 CONTRIBUTIONS

My work has improved PGF, at least a bit

Issue with the suggestion algorithm, cannot take more than one word..

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. Luckily, the creators of GF has thought about this and built embeddable runtime systems for a few programming languages [2, 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.

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[2][p. 13], especially with larger grammars. This is where the Portable Grammar Format (PGF)[2][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 we can 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*

Have not started on this yet... Don't forget how to set java lib path when using tomcat!

ACRONYMS

GF Grammatical Framework

RGL Resource Grammar Library

Java EE Java Enterprise Edition

PGF Portable Grammar Format

BIBLIOGRAPHY

[1] 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.

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

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

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