PRESENTATION CONTENT

MARTIN AGFJORD

CONTENTS

1 FIRST PAGE AND OUTLINE

Welcome to my masters project presentation! My name is Martin Agfjord and my project is about translating natural language sentences into machine readable instructions.

1.1 Outline

I'll first introduce the problem that we want to solve. Then I'll go through a solution to the problem, which is the largest part of the presentation. Then I'll show some results from the solution. Lastly, I'll go through what I can conclude from this work. I'll also talk a bit about some future work.

2 INTRODUCTION & PROBLEM DESCRIPTION

If a user wants to give a computer an instruction through a graphical user interface, it will probably use graphical elements like text fields, checkboxes and drop-down menus. In this project, we introduce an alternative interface, an interface which allows the user to write an instruction in a natural language.

This instruction is then translated into a machine readable language. We have chosen to use Solr query language as target language. The reason is because I have worked in close collaboration with a software development company called Findwise, and they frequently use Solr.

We need to have some kind of environment in order to define some instructions, therefore, we assume that we build this system for a software development company and the instructions are used to obtain information from their intranet. In fact, this project has gotten much inspiration from Findwise's intranet. This project only focus on a few instructions, because of limited time.

2.1 Interface definition

We assume that it is possible to obtain information of people's skills in the intranet, and the instruction `'people who know Java'` shall obtain a list of all persons that know the programming language Java. It has been said that novice users usually simply write what they want when searching on the web,

However, expert users usually just type keywords like `people know java`. We want to design an application that is suffient for both novice and expert users.

3 SOLUTION

Query languages and machine languages in general require precise syntax, which means a query cannot be interpreted if one character is in the wrong place in a query.

When translating, we want to map a natural language to Solr query language, how can this be achieved when also preserving precise syntax?

We use a grammar to solve this problem.

3.1 Translation with a grammar

So what is a grammar? In loose terms it is structured production rules for strings.

The production rules describe how to form strings, in our case combinations of *words* that are valid in a formal language.

If we know which rules that are used to create a a specific sentence in one language, we can use the same rules to create a sentence in another language that represents the same information.

It is not trivial to develop a grammar, so we make use of Grammatical Framework when building grammars.

3.2 Introducing Grammatical Framework (GF)

GF is a development platform for natural languages.

Grammars that can be used with GF are written in the GF programming language, which is a functional programming language.

This language is specifically designed for creating natural language grammars.

One of GFs powers is that it separates abstract and concrete syntax in the same sense as programming languages.

The abstract syntax captures the logic of a sentence. Or in other words, it captures the meaning of a sentence. This is achieved by using a hieratical tree structure.

The concrete syntax expresses how an abstract syntax tree looks like a string.

Compilers also uses abstract and concrete syntax.

The programmer writes source code in concrete syntax

The compiler translates the concrete syntax into an abstract syntax tree

The compiler can then manipulate the abstract syntax tree.

3.3 A simple example

Here we can see an example of an abstract syntax tree which I've developed for this presentation. This abstract syntax tree captures the meaning of the sentence `people who know Java`. This is my way of representing it, it can be represented in many other different ways.

We also have three concrete syntaxes that represents this abstract syntax as strings. The first two are natural languages, while the third is a representation in Solr query language in our environment.

In order to represent this abstract syntax in a concrete syntax, one combines the function `People` with the function `Know`, however the abstract syntax leaves the implementation to the concrete syntax, so two concrete syntaxes can therefore implement the same function in different ways. This can be seen by looking at the concrete syntaxes. While we don't know how these concrete syntaxes are implemented, a reasonable assumption is that the function `People` is implemented as `"personer"` in Swedish and as `"people"` in English.

3.4 GF implementation: Abstract syntax

I will now show how the example I just described can be implemented in GF. When developing a grammar in GF, one always starts with the abstract syntax to define the logic.

We start by defining our categories here in green. A category in GF is the same as a data type.

We then proceed to define how one can create an instruction. An instruction is created by executing the function `MkInstruction` with two arguments, one of the type `Subject` and one of the type `Relation`. The function can then output a value of the type Instruction.

A value of the type subject can be obtained by executing the function `People` with no arguments. A value of the type `Relation` can be obtained by executing the function `Know` with an argument of the type `Object`. Lastly, a value of the type `Object` can be obtained by executing the function `Java` with no arguments.

3.5 GF implementation: English concrete syntax

We will now implement a concrete syntax for English. We define all categories to be strings. The function implementations is done by concatenating strings. In MkInstruction we concatenate the subject argument with the string `who`, and we then concatenate the resulting string with the relation argument.

The function `People` is implemented as the string "people". Know is implemented by concatenating the string "know" with the object argument. Java is implemented as the string "Java".

3.6 GF implementation: Solr concrete syntax

We will also show how one can implement the concrete syntax for Solr query language. Similarly as with the concrete syntax for English we define all categories to be strings.

The functions are also concatenation of strings, but the strings are mostly different. To create an instruction we again use MkInstruction and concatenate subject with `q=` to form a Query parameter. A Solr query contains of several boolean statements, and we assume that the arguments subject and relation are valid boolean statements, we den add an `AND` between them.

In the function People we say that the field `'object_type'` shall have the value `'Person'`.

The function Know is implemented similarly, but the value is dynamic as we concatenate the object argument.

Java is just the string `Java`

3.7 GF implementation: Translation

So now when we have our abstract syntax and our concrete syntaxes, what can we do with them?

We can parse strings from concrete syntax to abstract syntax by using the parser that GF created.

We can go from abstract syntax to concrete syntax. We say that we linearize an abstract syntax tree to concrete syntax. By using both the parser and the linearizer, one can translate a string in one concrete syntax into another string in another concrete syntax.

It is also possible to generate all possible abstract syntax trees with the generator. However, as our grammar is very small, only one abstract syntax tree exists.

3.8 GF implementation: Resource Grammar Library

I am now going to talk about the GF Resourcec Grammar Library which is a library that one can use when developing a concrete syntax for a natural language.

The library contains the morphology and basic syntax of currently 29 languages.

I.e, it contains types for nouns, verbs, adjectives, noun phrases, verb phrases, relative sentences, phrases and so on.

by using these functions, a programmer does not need to know the linguistics of each natural language and does therefore not need to focus on tedious low-level details like in a natural language for example word order or agreement.

For example, consider the sentence `'Yesterday I ate an apple'`. If we just translated each word into swedish we would get the sentende `'Igår jag åt ett äpple'` but this looks odd right?

The correct translation is `'Igår åt jag ett äpple'`.

By using GF resource grammar library, one only needs to know the domain specific words in order to translate grammatically correct sentences. In this case one need so know the words Yesterday, I, ate and apple in Swedish.

The programmer can then use these words with the functions that exists in the resource grammar library.

3.9 Resource Grammar Library: English concrete syntax

I will now show how English for our grammar can be implemented by using the Resource Grammar Library.

We'll start by defining our categories. We use types from the Resource Grammar Library, an instruction is a noun phrase, a subject is a noun, a relation is a relative sentence and an object is a noun phrase.

The functions now use the functions provided by the resource grammar library. I'll start explaining the simplest functions.

Java is a proper name which is converted into a noun phrase.

The function `People` is implemented as a noun with the word `person` in singular form and the word `people` in plural form.

The function Know first makes a verb out of the string `know` and then convertes the verb into a verb second value. We then combine the verb with the object by creating a verb phrase. We then combine the verb phrase and the relative pronoun `which_RP` by creating a Relative Clause. Finally, we can convert the relative clause into a relative sentence.

An instruction is created by combining the subject which is a noun with the relation which is a relative sentence and creating a common noun. In order to only allow plural forms of the subject noun we use a combine a plural determiner together with our common noun and creates a noun phrase.

3.10 Resource Grammar Library: Swedish concrete syntax

I've also included resource grammar implementation for Swedish, where I've just exchanged the strings into Swedish words. So once you have the concrete syntax for one language, it is fairly easy to develop concrete syntaxes for other languages.

3.11 Extending the grammar

I'll now show I've extended my application to be sufficient to my requirements.

3.12 Extending the grammar: All programming languages

The grammar we previously developed could only translate one sentence regarding the programming language `Java`. We would like the grammar to support *any* programming language.

We solve this by using arbitrary names instead of hard coded functions.

Remember how we defined the function Java to be of the type Object and linearized into the string `"Java"`.

We replace this function with the function `MkObject` which make use of the built in type `Symb`. Symb is a special data type which has a field named s which contains a string. GF will now understand that the function `MkObject` can take any on any value.

3.13 Extending the grammar: More instructions

Even though we just introduced arbitrary names, the grammar is still only supports one kind instruction.

We want to extend the grammar to support additional valid instructions regarding customers, people and projects.

These are the instructions that the final application shall support.

However, we do not want to support invalid sentences like `'projects who work in London'`.

3.14 Extending the grammar: More instructions

We resolve this problem by adding more categories.

And we assign people to the Internal category, customer to the external category and project to the resource category.

The relation functions are also changed, Know returns a value of the type InternalRelation and we Use function for externals returns a value of the type ExternalRelation and Use function for resources returns a value of the type ResourceRelation. I have omitted the implementation of the other relation functions due to lack of space in this slide.

Instructions can be created by combining a two values of the correct types. For example a value of the type internal with a value of the type InternalRelation, which in this case can be obtained by executing the functions `People` and `Know`.

3.15 Extending the grammar: Boolean operators

It is very common in query languages to use boolean operators in between statements, we would like our grammar to support translation of such instructions from a natural language to query language.

For example, the instruction `'people who know Java and Python'` shall retrieve all persons in the database that know the programming languages Java and Python.

Another example is the instruction `people who know Java or work in London`.

We add support for these kind of instructions by adding recursive functions, one for `And` and one for `Or`. The implementation here shows how its done in the concatenation version of the concrete syntax for English, where we simply add the boolean operator in between the two objects.

3.16 Suggestion Engine

We have developed a grammar which can translate a few sentences into Solr query language. However, we have built a *narrow application grammar*, which means that it requires precise input. But how can a user translate anything if she have no idea of what instructions the application supports? And how can we support the use of only keywords when translating sentences?

We need a mapping from invalid or partial instructions into supported instructions.

We use a suggestion engine to accomplish this.

As GF can generate all possible instructions, we can store the instructions in a Solr index which we can use to map an invalid or partial instruction into an existing one.

However, as GF has now idea of what kind of names the application shall support we run into problems.

If we generate abstract syntax trees, we'll see that GF use the name `'Foo'` for all arbitrary names.

And if we linearize them into English, we'll get instructions like `'people who know Foo'`. If we store these sentences in the solr index, we will not obtain any result if we just type a name, like `Java`, `Python` or any other name that exists in the database.

3.17 Suggestion Engine 2

We resolve this by introducing types of names.

We introduce the types. Skill, Organization, Module and Location. Each of them will be used together with the appropriate relation function.

If we now generate abstract syntax trees again, we will see that each name is still named `'Foo'`, but it is of a different type, here in purple color.

We postprocess these trees and set the each name to be the corresponding type.

If we now linearize these trees into concrete syntax, we'll receive something better. The sentence `people who know Skill0` now dictates that Skillo can be replaced by a name of the type Skill. We store all linearizations in the Solr index. Note that we still don't have any actual valid names in the index, so a search for a name like `'Java'` not give any result at all.

3.18 Suggestion Engine: Pseudocode of algorithm

I'll now explain how we can use the Solr index we just created to give suggestions.

If we type the sentence `"anyone that ever knew java"` we first extract all names from the sentence. This is achieved by having a separate index with all supported names with corresponding types.

We then replace each name in the sentence with its type followed by a number.

We then query the solr index with the linearizations and we'll retrieve a list of suggested instructions.

Finally, we change back each type into the correct name.

4 RESULTS

I will now show the actual results from the application and as I like applications much more than I like charts

I'm going to give the reults live by using the web application.

5 CONCLUSION

From the results we can conclude that the application is sufficient for both novice and experts users as it supports suggestions based on both partial sentences and keywords.

While I find the resource grammar library very useful for translating between two natural languages, I did not think it was that useful when translating from a natural language to a query language. Especially not if you want to map a lot of grammatically incorrect sentences into valid queries.

It is also not possible to express the sentence `'people that know Java'`, we could however express `people who know Java`.

Also, this sentence could not be expressed by using the resource grammar library.

We could however express this very similar sentence.

We also had a problem with the constant `which_RP` which does not linearize correctly in some contexts.

Even though `'projects'` is a noun in plural form, `who` shall only be used if the noun a group of humans, like people.

5.1 Future Work

While the suggestions work fairly good, theres always room for improvements.

It would be good to suggest instructions even though the user hasn't typed anything.

If a user starts to type a valid sentence without any name, the application will fetch names to fill in the missing spots in the suggestions. It would be good to add some heuristic so we could fetch the most relevant names.

The application cannot translate invalid instructions at the moment, it just suggests valid sentences based on the input. It would be good take the first suggestion if the sentence is invalid.

There exists various speech recognition software today, and some of them are even free. It would be nice to use such software to transform speech into text. We can then use the resulting string with the application in order to add speech to instruction feature to the application.

When a user types an ambiguous instruction, the application just returnes the resulting abstract syntax trees. It would be good to explain to the user that the instruction was ambiguous and ask the user to clarify the instruction.

I don't know how realistic my example have been in the application with the intranet of a software development company, but I do think it can be used in many other environments that can be conrolled by natural language input. One does not have to be limited to translation from natural language to query language, it is very easy to use another machine language. I do have some ideas of my own that I'm going to experiment with after this.