CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Multilingual Text Robots for Abstract Wikipedia

Using Grammatical Framework to generate multilingual articles on Swedish localities

Bachelor's thesis in Computer science and engineering

Omar Diriye
Filip Folkesson
Erik Nilsson
Felix Nilsson
William Nilsson
Dylan Osolian

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Bachelor's thesis 2022

Multilingual Text Robots for Abstract Wikipedia

Using Grammatical Framework to generate multilingual articles on Swedish localities

Omar Diriye
Filip Folkesson
Erik Nilsson
Felix Nilsson
William Nilsson
Dylan Osolian

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Gothenburg, Sweden 2022

Multilingual Text Robots for Abstract Wikipedia
Using Grammatical Framework to generate multilingual articles on Swedish localities
Omar Diriye Filip Folkesson Erik Nilsson Felix Nilsson William Nilsson Dylan Osolian

Supervisor: Aarne Ranta, Department of Computer Science and Engineering
Examiner: Krasimir Angelov, Department of Computer Science and Engineering

Bachelor's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Multilingual Text Robots for Abstract Wikipedia
Using Grammatical Framework to generate multilingual articles on Swedish localities
Omar Diriye, Filip Folkesson, Erik Nilsson, Felix Nilsson, William Nilsson,
Dylan Osolian

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The vast amount of Wikipedia articles and languages has resulted in a high cost of Wikipedia, i.e. the required time and dedication for making every article available in every language. This paper describes the development of a multilingual text robot that will use data from the database Wikidata to generate articles on Swedish localities in various languages and how such a text robot can be beneficial for reducing the cost of Wikipedia.

The text robot has been developed using the functional programming language Grammatical Framework, the query language SPARQL, and Python. The topic of Swedish localities was selected due to the large number of localities in Sweden, the sparseness of currently existing Wikipedia articles on the topic (excluding Swedish articles), and the fact that the same structure, with only slight variation, can be used to describe all of the localities.

The results were articles containing approximately five sentences describing the locality, a bullet list of events occurring in the locality, and corresponding media, such as a picture of the locality or a weather forecast for the upcoming week. Based on the results, one can deduce that the use of a text robot might be a good approach for reducing the cost of Wikipedia since it produces over a thousand articles in several different languages. Another notable fact is that all project group members are bachelor's students with no previous knowledge of Grammatical Framework or linguistics, which shows that it is possible to develop a text robot with limited previous knowledge.

Keywords: Text robot, Natural Language Generation, Grammatical Framework, Multilingual Natural Language Generation, Abstract Wikipedia, Wikidata

Sammandrag

Den stora mängden wikipedia-artiklar och språk har resulterat i en hög kostnad för Wikipedia, det vill säga den tid och det engagemang som krävs för att göra varje artikel tillgänglig på varje språk. Denna artikel beskriver utvecklingen av en flerspråkig textrobot som kommer att använda data från databasen Wikidata för att generera artiklar om svenska tätorter på olika språk och hur en sådan textrobot kan vara till nytta för att minska kostnaderna för Wikipedia.

Textroboten har utvecklats med det funktionella programmeringsspråket Grammatical Framework, query-språket SPARQL samt Python. Ämnet svenska tätorter valdes med hänsyn till det stora antalet tätorter i Sverige, det nuvarande ringa antalet wikipedia-artiklar om ämnet (bortsett från svenska artiklar), och det faktum att samma strukturkan användas för att beskriv alla orter med endast liten variation.

Resultaten var artiklar innehållande cirka fem meningar som beskriver tätorten, en punktlista över händelser som inträffat i tätorten och motsvarande media, såsom en bild på orten eller en väderprognos för den kommande veckan. Baserat på resultatet kan man dra slutsatsen att användningen av en textrobot kan vara ett bra tillvägagångssätt för att minska kostnaderna för Wikipedia eftersom den producerar över ett tusen artiklar på flera olika språk. Ett annat anmärkningsvärt faktum är att alla gruppmedlemmar är kandidatstudenter utan förkunskaper i Grammatical Framework eller lingvistik, vilket visar på att det är möjligt att utveckla en textrobot med begränsade förkunskaper.

Acknowledgements

We would like to extend a special thanks to Aarne Ranta, for tremendous help and guidance thoughout the project. Also we would like to acknowledge the advice from Inari Listenmaa.

Glossary

NLG - NLG (Natural Language Generation) is a software process that generates texts that are written in a natural language.

GF - GF (Grammatical Framework) is a functional programming language that is used for creating grammars for multilingual applications.

RGL - RGL (Resource Grammars Library) is a library for GF which contains the morphology and syntax for many languages.

Wikidata - Wikidata is a free database containing a lot of different data and is used for Wikipedia

Syntactical - Relating to syntax, which describes the rules of how words words can be put together to sentences.

Morphological - Relating to morphology, which is the study of words, how they are formed and how they behave with other words.

Orthographical - Relating to Orthography, which is a convention for how a language is written. This could be for example the use of punctuation and capitalization.

SPARQL - A query based language for managing data stored in then RDF format.

Text-robot - A program that can create texts, for example articles.

Contents

List of Figures

1
Introduction

One of the most popular resources for quickly finding information on the internet is Wikipedia, an "open" encyclopedia that can be edited by the users themselves, with the prospect that their combined knowledge leads to and expedites the creation of more detailed articles. In practice, however, only a limited number of languages have an active userbase, which is something that has a big effect on both the quality and quantity of articles. The quality is also strongly linked to the engagement of the users and their ability to communicate with each other [1]. The most popular language, English, has just above six million articles and one hundred thousand active users, while languages such as German and French has around two and a half million articles and twenty thousand active users [2].

A problem that arises is how to populate the less active languages with articles that are of comparable quality to those of the most active languages. A solution to the problem could be that the most detailed article on the topic is translated into a more popular language that can be used as a template for other articles. This is arguably not a good solution since it requires more initiative from users with more niche knowledge, depending on the article's topic. Users are expected to perform this time-consuming task without compensation and deliver articles that are concise and explanatory, with citations.

Another solution is using an algorithmic approach to generate articles, where the articles are produced by retrieving information from a common source and using software to adapt the text around the information so that the complete article becomes coherent. This technology is called Natural Language Generation or NLG. NLG is the process where software creates natural text that can be read by humans. Such systems are available in many different forms and can be developed in different ways all with different advantages and disadvantages. NLG is currently for example used to generate different kinds of reports, such as weather reports or daily reports of the stock market. Some of the popular NLG models are based on probabilities, where the next word in the sentence is decided by how likely it is to appear, following the previous words. These models and methods of NLG can produce texts that are so similar to human written texts that they can not be distinguished. However to produce informative texts these models are not good and the content of the texts produced are often nonsense. A completely different way to implement NLG is a rule-based approach where grammars specify how the sentences should be produced. This method can be used to generate good sentences containing correct information and could therefore be used to produce Wikipedia articles. When the text gener-

ation is grammar based a lot of similarities between the generation of articles in different languages can be seen. Therefore, it is possible to extend the concept of NLG with multilinguality where the same text is generated on multiple languages simultaneously. This is a perfect answer for the Wikipedia problem where a lot languages are not populated with articles.

Multilingual NLG for Wikipedia is a concept that has already been explored to some extent and has seen moderate success. One of the languages that stand out in the Wikipedia article coverage is Cebuano, a smaller Filipino language that has the second most articles, after only English, with around 6 000 000 articles even though it currently only has about 200 authors. This is the result of *lsjbot*, an article generating program that is estimated to have created between $80-99\%$ of the articles [3]. Lsjbot has been met with some criticism as the articles, which are mainly about geographical objects and organisms, only briefly list facts. According to critics, the articles lack human features, making them less interesting to read [4].

Today, Wikipedia is ranked as the fourth most visited website and is by far the most visited encyclopedia in the world [5]. In June 2021, approximately 13.6 billion users visited the website in total, which was more visitors than Twitter, Amazon and Instagram combined during the same period [5]. Managing the information available on Wikipedia, as well as expanding it with high-quality, computer-generated articles so that it benefits even more users, is therefore of academic relevance.

1.1 Background

The expansion of Wikipedia is something that can be done in a cost-efficient way with the help of Multilingual NLG. Denny Vrandecic explains that to create a Wikipedia which covers topics in all languages, the cost can be defined as the number of topics multiplied by the number of languages [6].

$$Cost = Topics \times Languages$$

Considering that there are huge amounts of topics and also a significant amount of languages to cover, the multiplication between the two factors lead to the cost being very high and it makes it unreasonable to think that all topics and languages will be covered. However, with multilingual NLG it can be possible to turn the multiplication into addition by making the topics and languages independent of each other.

$$Cost = Topics + Languages$$

If a stable and quick method to generate Wikipedia articles with the topics being independent from the languages exists, it makes the cost of Wikipedia much lower and a future where all knowledge could be shared might be possible. This is exactly what is being explored in the Abstract Wikipedia project. The goal of Abstract Wikipedia is to let people share knowledge in a language independent way

1.2 Purpose and Aim

The purpose of this project is to explore the use of the programming language GF (Grammatical Framework) for generating multilingual Wikipedia articles utilising data from Wikidata. Further, this project aims to investigate the feasibility of people with a similar education and basic programming skills but no prior knowledge of GF to produce decent articles using this method. Thus the goal is to develop a text robot, with the use of NLG and GF,that can generate articles about a specific topic in a variety of different languages. The selected topic of articles for this project is Swedish localities.

1.3 Benefits of multilingual text robots

Using NLG to generate articles has several advantages: the articles have a consistent quality across several languages and when the data source is updated with new information, the articles are also updated at the same time (something that is currently being looked into with the *abstract wikipedia* project [7]).

Most common NLG methods are based on probabilities and statistics. These systems can solve complex problems but unfortunately it is difficult to understand their inner workings. The developers decide the architecture of the system and which algorithms to use but do not have further control on the values of the parameters in the model. This makes difficult for the developers to understand the models in order to inspect and debug them. However, when using GF to generate natural language, this problem does not appear, since the developer decides exactly how the model works. Furthermore, GF works great with multilingual NLG since it is designed to use abstract syntax, which is language independent, and RGL, which has a common API for many languages.

1.4 Societal and ethical aspects

A societal and ethical aspect that could be of importance for this project is where the data used to create the articles is retrieved from. In this project the data will be retrieved from Wikidata, which is assumed to contain mostly high quality data, but because the database is open for change, anyone could contribute with false data. Therefore the data from Wikidata is not always reliable, which could lead to the generated articles spreading inaccurate information. If the robot is meant to produce thousands of articles, the accuracy of the data has to be compared to the weight of making the information known to many more people, which is a significant societal benefit.

This balancing question appears again when you consider how well-written the generated texts will be. Minor grammatical errors or a wrong choice of words could lead to misunderstandings, but a slight misunderstanding in a few cases might not be important if the articles can spread knowledge to the world. Furthermore, the

same reasoning could be applied when generating Wikipedia articles for languages that only have a very small amount of articles on the internet. The automatically generated articles could then quickly become the majority of the text in that language online. If the articles would then be used in some sort of machine learning project, it could create a skewed image of the language since the articles most likely will not represent the language perfectly. But since the articles generated in this project are meant for Wikipedia, authors can always edit and extend the articles and therefore minimise the risk of the articles having some kind of negative societal effect. Generated articles usually have some kind of indication that the text was not written by a human, which should make sure that the texts are not used in a way as if they were handwritten.

It is of interest to note that a significant portion of the articles created by Lsjbot have been removed. This is because the opinion on articles generated by text robots has shifted towards being more cautious of it [3]. Some of the concerns raised are that the articles are too technical and may even contain factual errors present in the database used by Lsjbot [8]. At one point the entire of the cebuano wikipedia was at risk of removal for this very reason, with community members calling for an increase in quality control to combat the bot generated articles [9].

2
Theory

This chapter covers the concepts and tools used during the project.

2.1 Tools

Several tools have used to carry out the steps defined in section 1.2. These tools and their implementations will be described in this section.

2.1.1 SPARQL

SPARQL is a "Query language" which is used to select elements from a database. It is used to manipulate and retrieve data which is stored in *RDF* format.

RDF, or Resource Description Framework is a machine readable format used to describe data as triples. It consists of entity nodes and directed edges which contain the relationship between those entities. Each triple is an RDF statement and a collection of these statements make up a graph database where it is commonly used. SPARQL contains all necessary query operations to work with sets of data and has been used to select data from Wikidata which can be downloaded to different formats and then be used by the text robot. Below is a simple example of how a query could look:

```
1  {
2  SELECT ?city ?cityLabelSv ?population ?area
3    WHERE
4    {{
5      ?city wdt:P31 wd:Q12813115.
6      ?city wdt:P1082 ?population.
7      ?city wdt:P2046 ?area.
8      ?city rdfs:label ?cityLabelSv .
9      filter(lang(?cityLabelSv) = 'sv') .
10   }}
11 }
```

Listing 2.1: An example of a simple SPARQL query which was an early version of the query gathering data on localities.

The program begins by setting the columns to list in the table, which is done on line 2. It sets the columns to be city (the Q-value of the locality, which is a unique

identifier on wikidata used later on), citylabelSv (the name in Swedish), population, and area. After this follows the a number of RDF statements: line 5 states that the city column will consist of *instances of* (denoted as wdt:P31) *localities in Sweden* (denoted as wd:Q12813115). Line 6 states that the population column will consist of populations of the localities that was just listed, and line 7 lists the corresponding areas in a similar fashion. Finally the Swedish labels are listed with the use of a filter command. The results are shown below:

city	cityLabelSv	population	area
wd:Q1017138	Burträsk	1636	192
wd:Q961978	Hogstad	230	45
wd:Q962019	Röke	184	79
wd:Q5027	Norsesund	274	65
wd:Q5035	Sjömarken	2829	225
wd:Q5036	Aspered	314	61
wd:Q5037	Aplared	432	65
wd:Q5038	Sandared	3394	285
wd:Q21165	Falköping	17858	916
wd:Q744491	Ljungby	16052	1238
wd:Q745164	Sidensjö	377	128
wd:Q746023	Hälleforsnäs	1615	216
wd:Q734356	Älvsered	441	110
wd:Q734671	Seija	551	144
wd:Q737196	Karungi	193	61
wd:Q738338	Nynäshamn	15108	723
wd:Q738946	Huaröd	266	83
wd:Q743720	Anneberg	1469	113
wd:Q21168	Varberg	36019	2461
wd:Q21169	Piteå	23181	2298
wd:Q21166	Skövde	39543	2312
wd:Q25462	Mölnlycke	18392	1159
wd:Q25789	Karlskrona	36904	2121

Figure 2.1: The results from the SPARQL query. Note that for the Q-value column, they have an attached wd: indicating that every locality is an RDF object itself. This was stored as a .CSV file for usage in the code.

2.1.2 Grammatical Framework

Grammatical Framework is a functional programming language used to support the grammars of different languages. It was created in 1998 for the purpose of Multilingual Document Authoring and has since been developed to a tool that is mostly used in rersearch in computational linguistics but has also been used to create multilingual applications. GF was specifically developed to write and create grammars and has an architecture that enables multilingual grammar to be created. This is done by letting an abstract syntax have multiple concrete syntaxes which make for correct and fast translations. By using GF, the generated texts should be inflected in a grammatically correct way. GF is currently the only platform that enables the development of multilingual applications in an easy and extensible way.

Along with GF comes RGL (Resource Grammar Library) which is a standard library that includes a collection of grammars for several languages. At the time of writing, the library library supports 38 languages. By offering ready to use functions, developers using GF can build their application specific grammar relatively quickly and thus save time. Also, this enables developers to build complex grammars

without having expert level knowledge in linguistics.

2.1.3 Python

The text robot has been developed with Python as the host language. The host language has been used to link the application together by calling SPARQL queries to collect data and then, with the help of GF, building abstract syntax trees to generate sentences. Other programming languages can be used as a host language, but Python was chosen because it is considered one of the simpler languages that can be easily adopted by people interested in GF and NLG.

2.2 Syntax Trees

A syntax tree is a representation of the structure of a text, where nodes are used to dictate a formal set of rules or *grammar*. Specifically, there are two different cases of syntax trees: *Abstract Syntax Trees* (AST) and *Concrete Syntax Trees* (CST), which are sometimes also known as *Parse Trees* (Parse tree is the term used in the context of GF), with the difference between them being that a CST will be one particular instantiation of an AST (there could be more than one).

Syntax trees have many usages, but are foremost used in compilers to set the structural rules for programming languages. Relevant to this project however is the type of syntax trees which make use of *Phrase Structure Grammar*, which is a special type grammar that formalises sentence structures into known syntactic units. The top unit will always be S (Sentence), and it will be the root node in the syntax tree and can for example be followed by two child nodes NP and VP (noun phrase, verb phrase). An example of the CST for the sentence “Alice likes pretty syntax trees” is included below, as well as its associated AST. A notable property of the abstract syntax tree is its language independence. Therefore, when designing an abstract syntax tree, one does not have to consider the word order or the morphology.

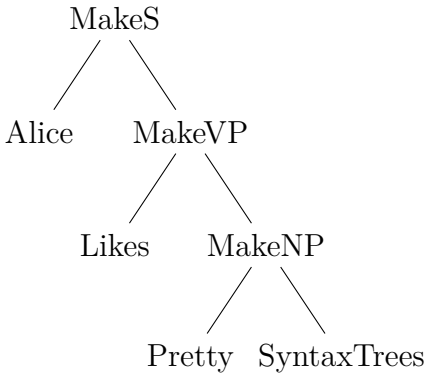

Figure 2.2: The AST for the alice example. Note that every node is a function which can be found in the abstract Alice code further below, with the leaf nodes being zero argument functions. As such the only relevance here is the type definitions, not the behaviour of the functions.

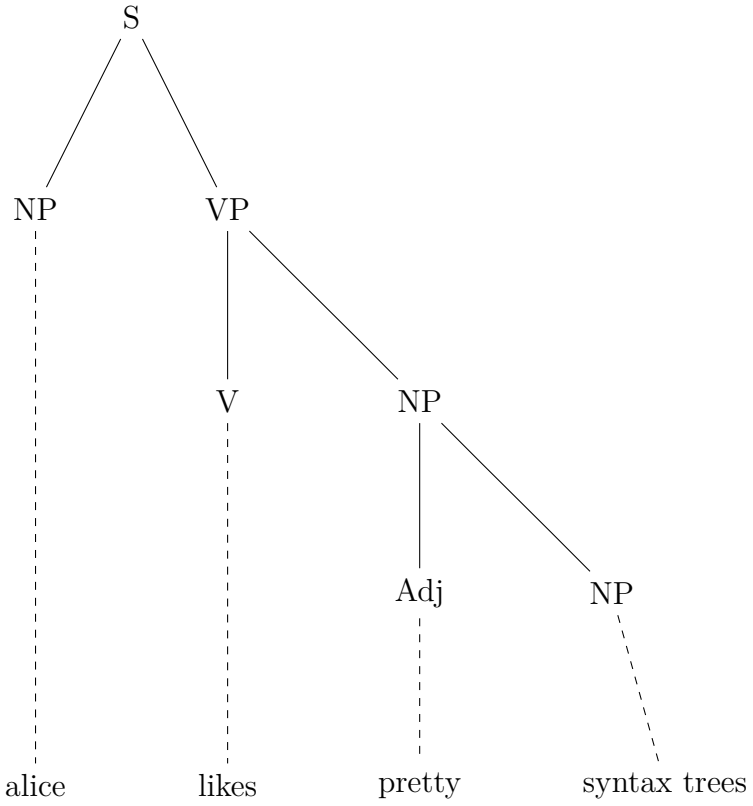

Figure 2.3: The CST for the alice example. The tree structure here is more involved and different in the way that nodes are now types instead of functions, and leaf nodes being strings.

2.3 Syntax Trees in Grammatical Framework

In GF, syntax trees and the process of analysing them are core concepts. In particular when writing a GF program, it will consist of an abstract syntax and one or more concrete syntaxes. Below follows an example of a concrete and an abstract syntax for the sentence "Alice likes pretty syntax trees.", used before to demonstrate the syntax tree (note that "–" indicates the start of a comment).

```
1  abstract alice = {
2          flags startcat = S ;
3
4          cat S; NP; VP; V; Adj;
5
6          fun
7                  MakeS              : NP  -> VP -> S;
8                  MakeVP             : V   -> NP -> VP;
9                  MakeNP             : Adj -> NP -> NP;
10
11                 Alice              : NP;
12                 Likes              : V;
13                 Pretty             : Adj;
14                 SyntaxTrees        : NP;
15
16 }
```

Listing 2.2: An abstract syntax which states that the start category for parsing will be S, then defines the categories (types) and then defines a series of functions using the categories as parameters. Since GF is a functional language, there are no variables, only functions with zero or more input parameters.

```
1  concrete aliceEng of alice = {
2          lincat
3                  S, NP, VP, V, Adj= {s : Str} ;
4
5          lin
6                  MakeS np vp      = {s = np.s ++ vp.s};
7                  MakeVP v np      = {s = v.s ++ np.s};
8                  MakeNP adj np    = {s = adj.s ++ np.s};
9
10                 Alice            = {s = "alice"};
11                 Likes            = {s = "likes"};
12                 Pretty           = {s = "pretty"};
13                 SyntaxTrees      = {s = "syntax trees"};
14
15 }
```

Listing 2.3: An accompanying concrete syntax which states which abstract syntax it implements, its linearisation category definitions which defines the categories contents, and the linerearisation definitions, which defines the behaviour of the functions.

With the code imported into GF, *parsing* the sentence "Alice likes pretty syntax trees." will generate the corresponding AST:

```
>parse "alice likes pretty syntax trees"
MakeS Alice (MakeVP Likes (MakeNP Pretty SyntaxTrees))
```

It should be noted that this is not the only valid sentence that can be parsed into an AST by this syntax. For example, when using the above syntax tree, an NP node can both have the child node of a string (i.e. a *terminal*) as well as two child nodes Adj and NP, so a valid sentence could also be "Alice likes syntax trees".

Conversely, *linearising* the AST will generate the parse tree, i.e. the string that was inputted at the start:

```
> linearise MakeS Alice (MakeVP Likes (MakeNP Pretty SyntaxTrees))

alice likes pretty syntax trees
```

Hence, parsing and linerarisation can be viewed as inverse functions of each other.

2.4 Multilinguality

Another core feature of GF is the ability to extend syntaxes with new languages. Continuing with the same example, we add a Swedish concrete syntax and we will see how this affects parsing and linerisation:

```
1  -- A concrete Swedish syntax
2  concrete aliceSwe of alice = {
3          lincat
4                  S, NP, VP, V, Adj= {s : Str} ;
5
6          lin
7                  MakeS np vp      = {s = np.s ++ vp.s};
8                  MakeVP v np      = {s = v.s ++ np.s};
9                  MakeNP adj np    = {s = adj.s ++ np.s};
10
11                 Alice            = {s = "alice"};
12                 Likes            = {s = "gillar"};
13                 Pretty           = {s = "fina"};
14                 SyntaxTrees      = {s = "syntaxträd"};
15
16 }
```

Listing 2.4: Another, Swedish, variant of the concrete syntax

Linearising the same AST as before now yields two results:

```
>linearise MakeS Alice (MakeVP Likes (MakeNP Pretty SyntaxTrees))

alice likes pretty syntax trees
alice gillar fina syntaxträd
```

Meanwhile, parsing either of the resulting sentences will generate the same AST as before.

2.5 Massaging syntax trees

Before linearising the syntax trees there is the possibility of altering or *massaging* them based on the language and other features. It can be seen as a first step of post-processing. It is at this step things can be changed like units (imperical instead of metrics) and calculations required for these unit changes can also be done. Massaging works by recursively traversing the tree and checking for nodes to be edited. This is done right before the linerisation. Our implementation of this can be seen in section 4.4.

3
Methods

In this chapter various methods pertaining to and used in the project are presented.

3.1 Development process

During the development process, we have developed the text robot using four steps each with increasing complexity, also known as a bottom-up development strategy. We started implementing the text robot for simple facts, for example, "The capital of Argentina is Buenos Aires" where all facts are described with the same sentence structure, i.e. "The *attribute* of *object* is *value*". This was initially done for only two languages but more languages were added in later stages of the development.

The next step of the process was to use RGL to adapt the grammar for different languages. This includes, for example, inflecting nouns for the correct gender form in various languages. In this step the base of the Python code was also implemented. This refers to the part of the code that is used to interact with GF and build syntax trees.

The third step of the development is about combining different facts into longer sentences and thus getting more fluency in the produced text. In this step, we also started using pronouns when referring to the object in order to avoid only listing facts.

In the fourth and last step, the text robot is developed to be able to select and summarise facts that appear to be interesting in the context. This is what is called content planning and can be used to different extents in text generation. When creating Wikipedia articles the human judgement is still partly needed to decide what is relevant to include in an article however there are still possibilities to include content planning to make the articles more enjoyable to read.

3.2 Using RGL to implement grammars

When designing the grammars used to represent the information fetched from Wikidata, we have used the RGL. The library contains language-specific morphological operators, the language-specific linearisation of various abstract categories and functions, and some language-specific operations. For example, using RGL lets us write a GF grammar, which describes how to linearise the different ASTs.

Listing 3.1: Example of using RGL to design grammars

```
1 concrete ExampleEngRGL of Example = open SyntaxEng,
2                                         ParadigmsEng in {
3
4 lin
5   ExamplePhrase =
6     mkCl (mkNP the_Det (mkN "programmer") (mkV "programs"))
7     --The programmer programs.
8 }
```

Thanks to the RGL, the linerasation of the different functions, i.e. mkNP or mkCl, is implemented so that we don not have to worry about the different word order or the different morphologicality between languages for the different phrases.

3.3 Virtues of pair programming

During the development of the text robot, we have opted to develop two and two. The reasoning behind this is that the number of bugs and errors will be reduced since they are more likely to be spotted during coding if two coders develop the code rather than one. Working in pairs also further enables discussing the problem with each other, which naturally increases the understanding of the problem at hand. Therefore, pair programming was chosen as our development practice.

3.4 Limitations

The software is judged based on the cohesion and grammatical correctness of the generated articles. In order for the quality of the texts to be assessed, the project has to be limited to languages that are understood on a higher level by at least one group member. The languages deemed to fit this category include but are not limited to: Swedish, English, Somali, Arabic, German, Spanish and Hungarian.

Different articles can follow very different patterns and because of this, the choice was made to initially limit the project only to articles with a shared topic/domain, with recurring properties present for every example. It could, for instance, be countries where each country has a capital and a population. A small-scale program could be able to display facts thus "The capital of *country* is *capital* and has a population of *number*. It can seem trivial to create a text robot executing this template, but some details such as the countries' capitals and names having to be in different languages makes the problem more complex. If the template would be further developed to say "There are X number of species in/on *country*" the problem is again more complex since it is correct for a country such as Iceland or Cyprus to, in Swedish, say "**på** Island/Cypern" whereas for a country such as Sweden one would say "**i** Sverige". A decision was made to initially limit variation to a reasonable degree in the text robot for the aforementioned reasons.

4

Results and Discussion

A text robot, generating articles about Swedish localities, was created during the project. The text robot can currently generate articles in three languages Swedish, English and German. The robot has access to data of about 1200 localities, including some bigger cities meaning that it can generate three articles for each one of the 1200 localities, for a total of 3600 articles. All of the articles are grammatically correct and also include an image, a map of where the locality is located, and a weather forecast for the upcoming week. A simple web application has also been developed to display the functionality of the text robot. In the web application, the user can select a locality, and articles in all the available languages will be generated for the selected locality, as well as relevant media such as images.

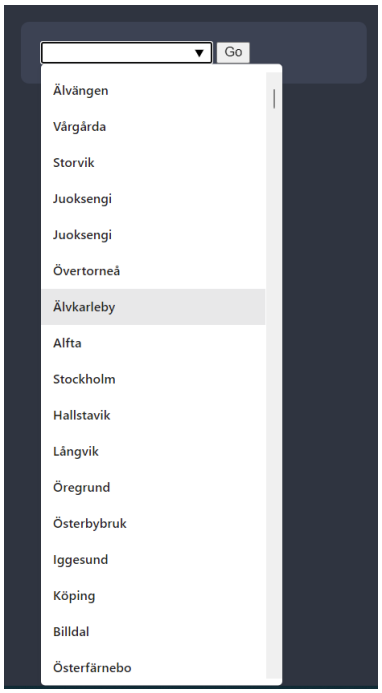

Figure 4.1: The list where the user can select which locality to generate articles about

Figure 4.2: Articles in Swedish, English and German about the Swedish locality Stånga

The texts give a short presentation about the locality and some related facts. They contain information about population, area, location and also mention famous people from the locality. The articles for a locality in the different languages all contain the same information, however there are small differences. For example the unit of the area is changed depending on which language the article is written in.

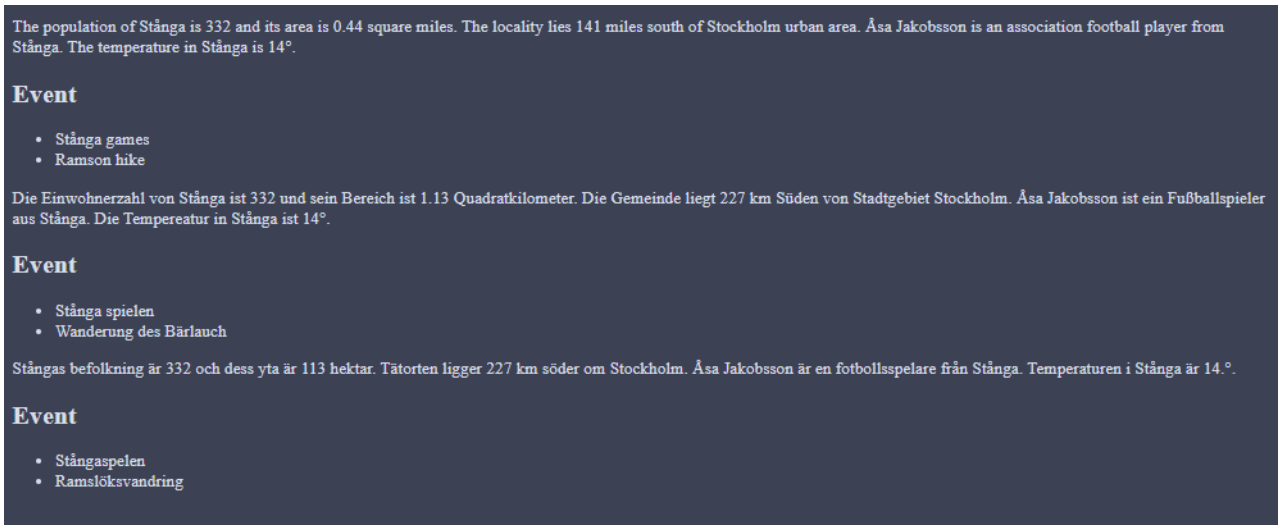

Figure 4.3: A closer look at the text in the articles

4.1 The algorithm behind the text robot

The program follows an algorithm, consisting of a series of steps, in order to generate the articles. The steps of the algorithm are defined as follows:

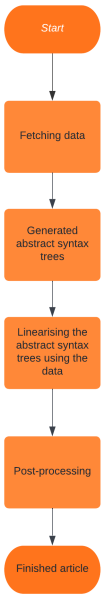

Figure 4.4: A visual representation of the algorithm behind the text robot

4.1.1 Data Fetching

The data needs to be fetched from the database Wikidata. This is done using the query language SPARQL, which allows for specifying what data is to be fetched. The data-fetching step is optional during runtime. Because the time it takes to fetch the data can vary due to outside factors (i.e. if many API calls slow down the Wikidata API), we have opted to fetch the data before running the program and storing it locally. However, live data such as the current weather at the given locality is always fetched at runtime.

To fetch the data, a Python wrapper of SPARQL is used which allows the execution of the queries remotely. It requires an endpoint URL to retrieve the data from and the query to execute. Furthermore, it is required to by the developer to specify the format of the returned data which is either JSON or TSV. After receiving the data, we save it as TSV files which are used in the later steps of the algorithm.

4.1.2 Creating the abstract syntax trees

To create the abstract syntax trees, we need to create GF constants (functions without argument) for some of the data, e.g. to be able to inflect the locality's name. After the functions have been created, the abstract syntax trees can be created with

help of RGL for the correct grammatical implementation of the sentences where the aforementioned constants can be used.

4.1.3 Linearisation of abstract syntax trees

After the data has been fetched, the program will now linearise the abstract syntax trees that have been built. The program will first select what data to show and the format in which it will be shown. Thereafter, the abstract syntax trees made for the different sentences and the different languages will be linearised with the data fetched in step one, and an article will be generated.

4.1.4 Post-processing the article

After the article has been generated, it needs to be processed. Here the article will be polished by, for example, making sure that there is a dot after each sentence, capitalising the first word of each sentence and fixing HTML header tags for titles.

4.2 Handling names

The handling of names is something that is important for the text to be correct. Looking at the chosen domain it is easy to see that a locality can have different names in different languages. For example the Swedish city Göteborg, is called Gothenburg in English. The difference might seem small but is important for the text to be correct. There could also exist cases where the name is not similar at all in other languages. The situation with names is something that is recurring and will be part of all text robots that are built. Another important reason to handle names separately and not just use them as strings when generating the articles is the possibility to inflect the names. Again this might not seem necessary in some cases however it is necessary if you want to use a name in the article in multiple ways. To look at an example the Swedish word for pilot is "pilot" in singular and would be inflected to "piloter" in plural. Had the name of this job title not been handled by including it in the grammar, it would not have been possible to inflect it.

To handle multilingual names in GF the names are included in the grammars as an abstract syntax and then their linearisations in concrete syntaxes. In the developed text robot this has been done with, for example, locality names and municipality names. The abstract syntax contains functions for each unique locality, named after their Q-value from Wikidata. The Q-value from Wikidata is a unique identifier and is used in the abstract syntax to avoid any issues with duplicated names. In the concrete syntax the linerisations are taken from Wikidata which has labels for Swedish localities in multiple languages.

These GF files containing the names of specific things can end up becoming very long and the code follows a similar pattern. In the case of, for example, locality names, the file is over 1200 rows long and contains a function for each locality. To write all these functions manually would be very time consuming, instead they can

be generated with the help of Python. In the project this has been done in a preprocessing python script,which is meant to be run before compiling the grammar, and it generates GF code for handling locality names, municipality names and also work titles in all of the currently implemented languages. Since the functions all have a similar pattern the generation of them is straight forward. One issue however that was encountered was that some names contained special characters which led to compile errors in the GF code. In the project this was solved by simply replacing the encountered special characters with something suitable, such as replacing a letter with a macron by simply removing the macron. We did however later learn that another way of doing it would have been to surround name with a pair of citations, because then GF would allow it. This solves the problem for all special characters and not just the ones that were encountered for the used data set. The whole process of handling names should however be seen as semi automatic since a manual check of the linearisations of the words is often needed in order to make sure that the translations in all the languages are correct.

Another point in the project where the handling of names became important was during the implementation of professions. The goal was to query famous persons who originated from a specific locality, as well as what their profession was and list them in the article. To do this, the plan was to query them once in Swedish (which is the language that should generate the most responses) and then use the *ESCO* dictionary to translate the professions in Swedish into any other language[10]. ESCO or European Skills, Competences, Qualifications and Occupations is an initiative by the European Comission to create a standardised dictionary for descriptions, titles and translation of professions and the like. For example this could help an employer to accurately advertise new opportunities and what qualifications are expected from an applicant.

There were however several difficulties that were hit upon with the ESCO approach. The professions and skills were listed in a CSV file such that some of them had unique codes which made translation easy, while some were considered variants of other professions which complicated the process. For example a famous person could be Astrid Lindgren, a children's fiction author from the locality Vimmerby. The Swedish query says one of her professions are "barnboksförfattare" and searching for it in the Swedish ESCO file (if it even existed) would name it as a variant of "författare" (author) which would have a unique code. Using this code in the English file would now yield the result of author, but finding the translation children's fiction author would be impossible.

On top of this there were several other issues. The file had an irregular structure which made the algorithm extracting the unique codes convoluted. Also the file sizes differed wildly which worsened the accuracy of the translations; The English file contained 30 000 lines, while the Swedish one contained around 8 000, and the German one around 18 000. As a result the solution that was chosen was to query profession names in a similar way to locality names, where the query directly states the language to use.

4.3 The article

To implement the text shown in figure 4.3 we had to consider how to design the sentence structure using the data retrieved. What follows is a description of how each sentence of the article is built.

> *The population of [locality] is [population] and its area is [area] square miles.*

This sentence is a conjunction of two separate facts. The first part is a population fact following the basic structure of *The ATTRIBUTE of OBJECT is VALUE*. The second part is of the same type but, since it is joined together with the first part, the *ATTRIBUTE* property is changed to *its* to improve the flow of the sentence. The area fact also makes use of massaging to present the area using square miles instead of hectare, since this is a sentence in English and the imperial units are used in the USA.

> *[Name] an is an athletic competitor from [locality]*

The sentence above is is one of the variations of a famous person fact. The first variation is produced when the person has one occupation and the other variation is generated if the person has more than one occupation. If the person has one occupation, the structure of the sentence is *PERSON is OCCUPATION from LOCALITY*. If the person has more than one occupation then the structure is *PERSON is OCCUPATION, OCCUPATION, ... and OCCUPATION from LOCALITY*. Comparing the two sentences, the difference is that the second sentence has a conjunction of occupations.

> *[Locality] lies in [municipality] municipality, which lies in the [south, middle or north] of Sweden.*

Here is an example of using a subordinate clause to make the sentence more natural. *The north* is not something that is listed on Wikidata, but rather something deduced from the coordinates and hard coded somewhat arbitrary values of what latitudes are considered the north, middle and southern part of Sweden.

> *The temperature in [locality] is 12.69°*

A sentence that differs somewhat from the others is the one describing the current temperature. This differs in that its data is gathered from a query not to Wikidata, but instead the weather API openweathermap.org. There were some issues that were ran into with this however, namely that since every query needs a longitude and a latitude, the amount of calls will at least be equal to the number of localities. This coupled with an initially unnecessary second query call per locality quickly made us exceed the allotted amount from the free API key. Some ways to work around this were to restructure the code to not call more than once per locality, as well as using

the web interface we created which will only create an article once we select it (that is, it will not automatically create every article once the website starts).

Events:

- Event1
- Event2
- ...

We end the article by listing a number of events in a bulleted list. As previously mentioned, Wikidata at points suffers from a lack of data, which in turn has an effect on the quality of the generated articles. Here this problem appeared once again in the form of events being generally available in Swedish, but not to an equal extent in the other languages.

4.4 Massaging syntax trees

As mentioned before a technique called massaging is used to process the syntax trees. The text robot uses this technique to convert the units of area and perform the calculations required for these conversions. For example, when generating an English article, square miles is used instead of hectare. The code block below shows a simplified implementation of this example. Massaging is also used to change finer details such as switching a comma sign for a dot when presenting decimals. The benefits of massaging trees is that the articles can use the same base facts, but still contain language specific properties, and therefore be grammatically correct.

```
1  def massage_tree(tree, lang):
2      fun, args = tree.unpack()
3
4      if fun == "unitHectare":
5          val = args[0].unpack()
6
7          if lang.name == "localitiesEng":
8              # Return unit in miles, with value converted
9          elif lang.name == "localitiesGer":
10             # Return unit in square km, with value converted
11         return pgf.Expr("unitHectare", args)
12
13     args = [massage_tree(t, lang) for t in args]
14     return pgf.Expr(fun,args)
```

Listing 4.1: Code example of using massaging to convert hectare to square miles and square km depending on the language

4.5 Content Planning

When generating articles a lot of decisions have to be made including the following: what facts should be included; in what order the facts should be written; and in what form to present the facts. This is dependant on both what data is available and its quantity. When listing historic events, for example, Gothenburg will clearly present more data than Juoksengi, which is why methods to handle both cases are required. This is solved by having multiple grammars ready for different cases.

> *Floda lies in Lerums municipality, which lies in the south of Sweden. The locality lies 28.08 km east of Gothenburg.*
>
> *Stockholm lies in 8 municipalities, in the south of Sweden.*

At this stage it is also possible to do calculations based on the data available and use this to create new facts. The second fact in the first example above shows how this is used to calculate a distance to the closest big city, from a list of our choice, using the coordinates alone. Also noteworthy is that for the Stockholm example, no such fact appears since there is no need to put Stockholm relative to another big city.

When creating articles the format in which to display information becomes interesting. Some information fits better in different formats, for example some kinds of information are best expressed in plain text while other kinds might need images or graphs. A format which is common in Wikipedia articles are bullet lists. Bullet lists are an easy way to list some kind of data in a clear way without cluttering it with text. An additional benefit with bullet lists is when list items are linked to other Wikipedia articles. This makes it very easy for users to navigate Wikipedia and read more about items in the list which might be interesting.

In the generated articles different events that occurred in Swedish localities have been listed using bullet lists. The implementation of the HTML lists is done mostly in GF with the help of the Markup module to create the HTML tags. Some additional post processing is also done in Python to remove conjunctions that have been applied in GF. Two categories, HTMLList and ListItem were created in GF which have been used for functions which can create lists and add list items to this list. The only thing that sets these functions apart from basic list functionality is that the list items are wrapped with <li> tags before they are added to the list. tags are the HTML tags for list items. For the bullet list to be completed it has to be wrapped with <ul> tags, which is done by function wrapWithUl. When doing this, the list which is of type listNP is converted to an NP by adding a conjunction between each element, the conjunctions are later removed in python.

4.6 Live data

A benefit of using NLG to generate articles is that the article will always be up-to-date with the latest data. The created articles in the project contain information about, for example, the population of different localities, which is data that can change. In fact a lot of data tends to change over time and therefore it is of relevance to display the latest and correct data. If we put the text robot in the context of the Abstract Wikipedia project where a Wikipedia article could be generated when opening the article, it is then possible to always use the latest available data when creating the article. This is because the relevant data can be queried from Wikidata when generating the article. To get a good result this assumes that the data in the database is correct and is updated when there is new data available. The data doesn't necessarily have to be some number that has changed but it can also be that some new data is added. If there is a person that has recently become famous and information about the person's birth location is added to Wikidata, this person would automatically be included in the article about the relevant locality. However if articles are manually written this addition would have to be done manually.

With this possibility in mind it also creates opportunities to use live data in articles. To differentiate from the data described in the paragraph above, live data is considered to be data that is updated live or very frequently. Although it might not be the type of data that would usually appear in Wikipedia articles, the generated articles about localities do contain live information to demonstrate the concept of using live data. The way it is applied to the text robot is by getting current weather information at the locality. Meaning that every time the article is generated, weather information is retrieved and displayed in the article. This feature demonstrates another upside of using NLG to generate articles instead of the conventional user written approach. This concept could be useful in some specific articles in the Abstract Wikipedia project however it might be more useful in cases outside of Wikipedia where live data has to be displayed.

4.7 Query data

Currently there are five queries which fetch a variety of different data. The first and main query is "query_localities" that gets data about the localities themselves, such as the locality's name in multiple languages, ID, population, area and coordinates along with a picture. The ID is a unique code for each item on Wikidata that starts with a Q. This is useful since there are multiple localities sharing the same name and the Q-code can thus be used to differentiate them in the code.

```
SELECT ?city ?cityLabelSv ?cityLabelEn ?cityLabelDe
?population (max(?area) as ?maxArea) ?coordinates
(group_concat(distinct ?municipalityLabelSv; separator=", ")
as ?muniLabels) (group_concat(distinct ?municipality;
separator=", ") as ?muniIds) ?image
  WHERE
  {
    ?city wdt:P31 wd:Q12813115.
    ?city wdt:P17 wd:Q34.
    ?city wdt:P1082 ?population.
    ?city p:P2046 [pq:P585 ?dateArea; ps:P2046 ?area].
    ?city wdt:P625  ?coordinates.
    ?city wdt:P131  ?municipality.
    ?city wdt:P18 ?image.
    ?municipality rdfs:label ?municipalityLabelSv.
    ?city rdfs:label ?cityLabelSv .
    ?city rdfs:label ?cityLabelEn .
    ?city rdfs:label ?cityLabelDe .
    filter(lang(?municipalityLabelSv) = 'sv') .
    filter(lang(?cityLabelSv) = 'sv') .
    filter(lang(?cityLabelEn) = 'en') .
    filter(lang(?cityLabelDe) = 'de') .
    FILTER NOT EXISTS {?city p:P2046 [pq:P585 ?dateArea_]
    FILTER (?dateArea_ > ?dateArea)}
  } group by ?city ?cityLabelSv ?population ?coordinates
  ?cityLabelEn ?cityLabelDe ?image
```

Listing 4.2: Example of a query, this is query_localities

The query_persons is, as the name suggests, used to fetch data about different persons from the localities along with what their occupation/occupations are and where they are born. Since more famous people are of more interest to be included in an article the persons-data is thus sorted by the number of sitelinks they have. Sitelinks area measure on the number of Wikimedia sites that link to a specific Wikidata item. This data can then be matched with different localities, using the birthplace information and thus the articles can be made to have text about certain famous people from the locality in question.

There are a further three queries used in the project: query_events, query_buildings and query_sport clubs. The event query, as the name suggests, gets data on different events that have been held in the city such as a festival or sports competitions. The buildings query is very similar, it returns different notable buildings in the localities along with their construction dates. Finally the sport clubs query retrieves data on different sports clubs. These three queries all fetch the Q-code of the specific items: events, buildings, sport clubs. Additionally the query fetches the names of them as well as the locality in which they are located.

4.8 Post processing

To clean the articles and remove minor issues, a post processing script has been written. The issues this script solves are some that we might have not been able to solve with GF or that were more effective to solve in Python script rather than spend time finding a solution using GF. The first issue is new lines, a new line identifier is added in the GF grammar which can then be targeted and replaced with "\n". Another minor issue solved is capital letters in the beginning of a sentence.

4.9 Adding new languages

Furthermore, something the project set out to examine during the course of the development of the text robot is the potential ease of adding an additional language using GF when the features are already fully implemented for the other languages. To add a new language one would firstly need to add the Spanish labels for localities and municipalities to the SPARQL-querys, then the program will create GF constants for the new language. Secondly, one would have to create the GF grammars for general facts and specific locality facts for the new language.

This is something that was investigated when adding the Spanish language towards the end of the project. However, Spanish turned out to not be easy to implement. This is partly due to the fact that the RGL functionality for the Spanish language is not as extensively implemented compared to Swedish, German and English (what is lacking specifically is mentioned in section 5.1). Therefore the GF grammars for general facts and specific locality facts were difficult to implement since the missing RGL functions would have to be created manually.

4.10 Using default values

Considering that the topic of the articles is Swedish localities, it is not surprising that the Swedish names of these localities are more common in Wikidata than in other languages. Currently, the text robot does not default to the Swedish value and simply omits the locality if its name does not exist in Wikidata for all of the languages used. Therefore it might have been better to use a Swedish default value for the names and, in doing so, write articles on more localities. However, other fields, i.e. occupation titles, would be strange to default to Swedish because the sentence would make no sense for non-Swedish speakers, for example, "Stellan Skarsgård is a *skådespelare* from Gothenburg". Furthermore, since English is considered a much more international and well-known language, it might be better to default to English instead of Swedish for occupations. The use of default values is, however, not yet implemented.

4.11 Difficulties

Various difficulties were encountered during the project some of them are as follows:

4.11.1 Fewer results with different languages

Various problems were encountered when querying data from Wikidata. Since we are working with several languages, we query the labels of the objects in all those languages. To achieve this, a filter has to be used in which the languages that the labels should be in are specified. The consequence of this is that fewer results are obtained. Objects that do not have labels in all of the specified languages will be ignored.

This was not such a significant problem when only working with Swedish and English since the localities' and other queried items' labels were available on Wikidata. However during the implementation of the German language and when simultaneously looking into adding the other languages such as Spanish and Arabic, noticeably fewer items had labels defined in these languages leading to the queries returning significantly less results since it does not return empty columns when something isn't available in that language.

This is obviously a severe hurdle to overcome since adding a language could significantly reduce the amount of data available and to help to alleviate this one could take several routes of action. For example the query could be split up to have a separate query getting the labels for the languages that have a sparser amount of data. After this it could be possible to look into translating, either manually or automatically, using another database to find equivalent labels in the languages or for languages with a different script looking into transliteration of the native names. The latter however, although working for things like names of places or people, would not work for labelling occupations e.g. one could not just transliterate "author" into the other languages but would rather have to translate it. With time though this issue would certainly diminish as Wikidata continues to grow and more data is added to it in all languages.

4.11.2 Query timeouts

Since the query run-time is limited to a mere 60 seconds, larger queries that fetch or have to filter through large amounts of data time out and thus not returning any of the wanted data. This happened many a time while developing our queries. Often this could be solved by optimising them such as limiting the query earlier or removing useful but non essential data, such as labels for everything in all languages, leaving more to the preprocessing stage. However sometimes, as when we were working on a query to fetch all events that occurred in a locality (such as concerts, accidents or historical events), we were not able to make it conform to the 60-second time limit because there was such a large amount of data. A way we found to solve this is to split the query into multiple smaller queries each fetching a

subset of the larger set “event”. This often works since the amount of data retrieved from the database is smaller and takes less than 60-seconds. Later, the results from the smaller queries is merged in preprocessing and works as if it were produced from one large query.

A concrete example of this issue and how it was solved is when the query fetching famous people was expanded to also retrieve the occupation labels for German and Spanish, the query timed out. To solve this a separate query, fetching the same data for people, was written but instead of Swedish and English labels now retrieving German and Spanish ones. Then these results could be merged in preprocessing where they were matched to the corresponding results from the other query using the person IDs.

We have seen another issue with Wikidata. We have a query that works on the Wikidata query site but does not work in python. We simply get a timeout error and it does not seem to be a way to further debug the issue. To deal with this issue, one can use the interactive wikidata query service to download the result files. These files containing the raw data could then be cleaned in preprocessing.

4.11.3 Linguistics

A significant part of the project came down to learning the subject of linguistics, which no member of the group had any real prior experience with. As a subject we found that it differed somewhat to our mostly technical background, but we also found related areas like data structures or formal languages to bridge the gap in experience. This meant getting familiar with terms and concepts commonly used and how they were integrated into GF, some of which were: syntax and syntax trees specifically, morphology as well as orthography. Other than researching linguistics on our own and experimenting with GF, getting over this initial road bump in knowledge was done by attending tutorials given by professor Aarne Ranta and Inari Listenmaa Ph.D who actively maintain the GF programming language. We were also given the chance to attend a meeting/discussion where all currently conducted bachelors theses as well as masters theses related to GF presented their work, hosted by Denny, which gave us some perspective on our project.

4.11.4 Available Material

When learning a new programming language, a common strategy is to look to websites like stackoverflow.com or similar for solutions when you run into errors. For popular programming languages this tends to work well, since most likely the problem you run into is common and there is ample information on how you could amend your code to solve the issue.

As it turns out however, GF is a programming language with a very particular niche and does not have a large base of developers who ask and answer question threads like the ones mentioned above. This complicated the development process,

and so we had to look elsewhere for bug fixes. Most often, we ended up asking Aarne whenever we ran into a bug which we could not solve, but we were also given the chance to ask questions in a dedicated GF discord channel.

4.12 Comparison with other existing generated articles

If an article generated by the text robot developed in this project figure 4.2 is compared to an example of an article created by another text robot, Lsjbot figure 4.5, many similarities and differences can be noted. One difference is in the content planning of the articles. Lsjbot seems to focus mostly on geographical and climate data which might be of interest for a certain audience. The robot in this project however includes data about famous people and events. This is to make the articles more enjoyable and interesting to read for a wider audience. When representing examples of notable famous people from the locality our article also provides information about their occupation. Furthermore the articles generated by our text robot has a picture taken in the locality included, as a nice decorative feature and as a way for the reader to get an idea of the locality while reading.

Both articles mention where the place is located geographically, both in text and map format, as well as including in what municipality it lies. The population is also mentioned, although in the article created by Lsjbot it is only mentioned as a statistic on the side and not in the actual body of the article. Some data about the weather is also included but unlike the article generated by our text robot, the Lsjbot article only provides historical information about the weather while ours provides live data as discussed above figure 4.6.

Another thing to note is that Lsjbot is only implemented for generating articles in Swedish, Cebuano and Waray, where it currently only is active in Cebuano [3]. Our text robot can generate articles in Swedish, English and German currently and implementing new languages, although being more difficult than initially perceived by us, is not that difficult and the project has the possibility to be greatly expanded to cover many languages without much additional labour as compared to when implement.

In conclusion, what separates our bot form Lsjbot is mainly that we have chosen a narrower topic which has allowed us to be more specific and able to plan ahead. Where Lsjbot fails is in that it creates articles for many broad topics, which results in a very short article in general with little interesting information. It would not be possible to generate millions of articles with our bot as lsjbot has done, without implementing many additional languages.

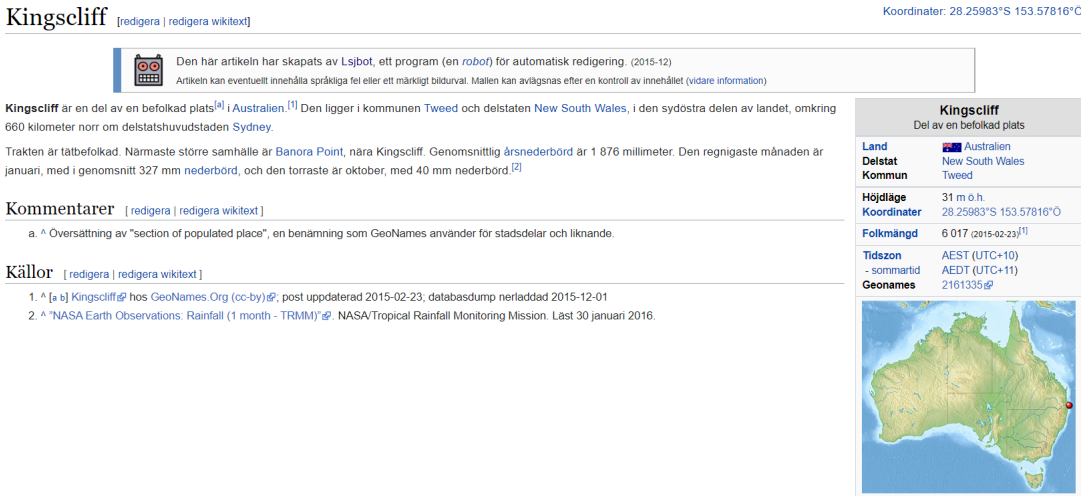

Figure 4.5: An example of an article about a town generated by Lsjbot

Figure 4.6: Articles in Swedish, English and German about the Swedish locality Stånga

5
Conclusion

Overall the project can be seen as a success and there are several conclusions and learning outcomes to discuss. These will be brought up in this chapter.

5.1 Localities as a domain

Before any coding started, there were discussions regarding which topic of articles the text robot should write. When selecting a topic for the project, it is important that the different articles share a similar structure since that enables the use of a "one fits all" approach when designing the layout of the article. Furthermore, the topics must have known properties that do not vary between the various articles. For example, all of the Swedish localities have a population, an area, is part of a municipality and so on. That makes it easier when designing the article. Also, the currently existing articles about the topic had to be short enough for the text robot to be able to write better articles than the preexisting ones realistically.

With these factors in mind several topics were considered, but in the end we settled for Swedish localities (Swedish: *orter*) as the topic. They have well known, well defined properties like population, area and other neighbouring localities. In general the existing articles are very sparse for most localities, especially for languages other than Swedish.

As time and work on the project progressed, some issues with the topic became apparent however. For example, the low amount of data available for less popular localities set the ceiling for what articles could be generated, even if more data was available for more popular localities. Facts like certain famous events could not be relied upon to exist for every locality.

With more time at our disposal, many new features could have been implemented. One such thing is adding more languages such as those that we were not able to implement like Arabic, Hungarian and Somali, but also potentially other languages. One of the benefits of GF is that adding new languages, after the structure is already defined for other languages, is relatively easy and the text robot could then be expanded to include many more. When adding a new language, only the concrete syntaxes would have to be added for the language while the abstract ones require little to no changes. Specifically there was an effort to implement Spanish, but due to a techical problem that did not appear for Swedish, English or German it, we

chose not to continue. The problem was that in the library SyntaxSpa.gfo the function CardCNCard was not included, which we used for other languages.

Currently the text robot only creates articles about Swedish localities. Expanding this to include localities in other countries would be of interest, especially if this project was to be used in the creation of Wikipedia articles meant to be published officially. It would not be such a hard task since the queries are already defined and only certain parts would have to be changed. For example, expanding the restriction on which areas the localities lies in.

5.2 Expanding the articles

At the moment the articles are not that long and in the future one would potentially want to add more information to them. This would require writing new queries to retrieve the new information. Additionally, one would have to think more about content planning and how to present the new information and structure the article accordingly. Furthermore, since a new query would have to be created, along with the new data being implemented in the grammar and code, it would thus not be as simple of a task as for example adding a new language. For the new queries to be useful the new information that is retrieved in order to be added to the articles has to be somewhat general and applicable for many localities. This since if the data that is retrieved is specific for a particular locality or small set of localities, the query could only be used for those localities. Thus writing such a query and implementing the sentences to represent the new information in GF would be a laborious task and would not have as big of a return. However, since Wikidata is continuously expanding, this issue could be diminished when more data is available.

5.3 What can be done to make it easier for programmers to get into GF

In order for developers and programmers to more easily learn to use GF a number of things could be done. For example, expanding the website with more documentation and concrete examples, and maybe tutorials and videos, would ease the learning curve of getting into GF. This would also be helpful since GF is still a smaller programming language and thus does not have as many bug fixes and resources online, and providing more documentation and resources would thus be a boon for people looking to get into it.

5.4 Improving the web application

At this stage, the web application is at its simplest form. It was developed mainly to easily browse the articles and to prove that generating articles on demand using GF is a possibilty for future projects. However, there are multiple features missing right now that could be implemented at a later stage or by other people. One example is a more Wikipedia-like look and feel. Another way this program could be implemented could be to make an API out of it and make it available to other sites such as Wikipedia to integrate it directly.

5.5 The value of the project

Our text robot can currently generate basic articles for hundreds of different Swedish localities in four languages, many of which do not currently have articles on Wikipedia. Thus this project could be used to expand the knowledge available to people in their languages, that was not previously existent. Additionally, the text robot can further be improved and expanded to include more languages and a bigger set of data, such as localities in many different countries, and thus the value provided to people with more information in several languages would increase.

5.6 Future of Abstract Wikipedia

The Abstract Wikipedia project is exciting and has good chances of succeeding in creating a Wikipedia which is available in many languages. As can be seen from this project the foundations for Abstract Wikipedia is already laid and the possibility to create multilingual text robots exist. However to really get the Abstract Wikipedia project to expand and grow there are some things that would help. As mentioned earlier the documentation of GF and RGL is something that could be improved, this would really help new people who are interested in contributing to Abstract Wikipedia to get started. Using RGL especially was something that limited the development of the robot since a lot of effort had to be put into learning the API and managing to build correct sentences. This is mostly down the lack of knowledge in linguistics prior to the project. Although for Abstract Wikipedia to really succeed there can't be an expectation that all the contributors have knowledge in linguistics. For that reason an introduction to linguistics and a more thorough tutorial on RGL is a good idea to make it easier for new contributors to the project.

Another reoccurring problem was the issue where there didn't exist Wikidata labels in different languages for a lot of information that was used in the text robot. The main goal of Abstract Wikipedia is to share knowledge in a language independent way. To do this the Wikidata data has to be be expanded so that the labels cover a lot more data in many more languages. Even in this project of limited size, big gaps of the labels in different languages were found. Having good coverage on the labels for many languages is something that should be considered important for the

expansion of Abstract Wikipedia. Since Wikidata relies on voluntary contributors it is not easy to just translate all the data, however it might be a good idea to encourage users to contribute. Exactly how this is done is not important but a simple interface to add labels in new languages might encourage to contribute and lead to a Wikidata with higher coverage.

5.7 Connecting back to the project's aim

Looking back at the project aim, the first part of it was to investigate the use of GF to generate Wikipedia articles using Wikidata as the data source. As our result shows, we were successful in using GF to generate articles with data retrieved from Wikidata. By making the articles longer and adding more facts, these could potentially be used in Wikipedia. This proves the value a text robot built with GF could have in generating large amount of articles all by using the same grammar. Our result also shows the difficulties that are there when implementing the text robot. Some of these difficulties, such as query timeouts, are caused by Wikidata.

The second part of the project aim was to evaluate how easy it would be for developers with no prior knowledge of GF and similar education background to us to produce relevant Wikipedia articles. The main hurdle here as discussed earlier sections is the limited amount of available information on GF. Since GF is not as widespread as some of the popular programming languages, it is difficult to find information on for example bug fixes. As the GF community grows with time this may become less of a problem, but real improvement would come from making the technology more accessible to wikipedia contributors without a background in programming. Forms and intuitive GUIs over programming interfaces may allow them to contribute their human knowledge necessary to generate articles, without having to go through the process of not only learning to program in general but also learning to use GF.

5.8 Reflections on Prerequisite Skills and Suggestions for future work

Given that the topic of localities is a relatively simple subject compared to other less conformative topics, one could assume that given more prerequisites, more difficult topics could have been selected. Even if the group as a whole had all the relevant experience in programming and computer science necessary to complete the project, one large obstacle that kept showing up was the inexperience in linguistics. It should be noted however that this was partly the point of the project: to test how well students proficient in the technical details could perform without a background in linguistics. Given more experience in linguistics more difficult language could be generated and therefore more advanced article topics.

6
Bibliography

[1] A. Kittur and R. E. Kraut, "Harnessing the wisdom of crowds in wikipedia," *Proceedings of the ACM 2008 conference on Computer supported cooperative work - CSCW '08*, 2008. DOI: 10.1145/1460563.1460572.

[2] C. t. W. projects, *List of wikipedias*, Apr. 2022. [Online]. Available: https://meta.wikimedia.org/wiki/List_of_Wikipedias.

[3] K. Wilson, *The world's second largest wikipedia is written almost entirely by one bot.* [Online]. Available: https://www.vice.com/en/article/4agamm/the-worlds-second-largest-wikipedia-is-written-almost-entirely-by-one-bot.

[4] E. E. Jervell, *For this author, 10,000 wikipedia articles is a good day's work*, Jul. 2014. [Online]. Available: https://www.wsj.com/articles/for-this-author-10-000-wikipedia-articles-is-a-good-days-work-1405305001.

[5] J. Clement, *Global top websites by monthly visits 2020*, Sep. 2021. [Online]. Available: https://www.statista.com/statistics/1201880/most-visited-websites-worldwide/.

[6] D. Vrandečić. Wikimedia Foundation, Jul. 2021. [Online]. Available: https://www.youtube.com/watch?v=if5TeJ8N2p8&t=648s&ab_channel=CentreforComputationalLaw%28CCLAW%29%2CSMU.

[7] *Abstract wikipedia*, Jan. 2022. [Online]. Available: https://meta.wikimedia.org/wiki/Abstract_Wikipedia.

[8] *Bybrunnen/arkiv 2016-11*, Sep. 2017. [Online]. Available: https://sv.wikipedia.org/wiki/Wikipedia:Bybrunnen/Arkiv_2016-11#Robotskapade_artiklar_en_nackdel?.

[9] *Proposals for closing projects/closure of cebuano wikipedia.* [Online]. Available: https://meta.wikimedia.org/wiki/Proposals_for_closing_projects/Closure_of_Cebuano_Wikipedia.

[10] *What is esco?* [Online]. Available: https://esco.ec.europa.eu/en/about-esco/what-esco.