



















# Contents

<b>List of Figures</b>	<b>xi</b>
<b>List of Tables</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Accessing databases using natural language . . . . .	1
1.1.1 Usefulness . . . . .	2
1.2 Approach . . . . .	2
1.2.1 Goals . . . . .	2
1.2.2 Strategy . . . . .	3
<b>2 Theory</b>	<b>5</b>
2.1 Previous work . . . . .	5
2.2 Grammatical Framework . . . . .	6
2.2.1 Resource Grammar Library . . . . .	8
2.2.2 Functors . . . . .	9
2.3 Databases . . . . .	10
<b>3 Methods</b>	<b>13</b>
3.1 Grammar files . . . . .	13
3.2 Grammar contents . . . . .	14
3.2.1 Abstract . . . . .	14
3.2.2 SQL . . . . .	18
3.2.3 Natural languages . . . . .	18
3.2.3.1 Resource grammar library . . . . .	19
3.2.3.2 Variants . . . . .	20
3.2.3.3 Functor . . . . .	20
3.3 Application . . . . .	22
3.3.1 Hosting . . . . .	23
3.3.2 CGI and XHtml . . . . .	23
3.3.3 Reading and writing GF files . . . . .	24
3.3.4 Database access . . . . .	24
3.3.5 Application walkthrough . . . . .	25
<b>4 Results</b>	<b>29</b>
4.1 Assessment of goals . . . . .	29

<b>5</b>	<b>Discussion</b>	<b>33</b>
5.1	SQL . . . . .	34
5.2	Variants . . . . .	35
5.3	Security . . . . .	37
5.4	Limitations . . . . .	37
5.5	Further work . . . . .	38
<b>6</b>	<b>Summary</b>	<b>41</b>
<b>7</b>	<b>Bibliographic notes</b>	<b>43</b>
<b>A</b>	<b>User guide to installation</b>	<b>I</b>
A.1	Grammatical Framework . . . . .	I
A.2	Getting the necessary code . . . . .	I
A.3	Configuring a local server . . . . .	II
<b>B</b>	<b>User guide to the application</b>	<b>III</b>
B.1	Accessing the server . . . . .	III
B.2	Reaching a Database . . . . .	IV
B.3	Adding a language for a database . . . . .	IV
B.4	Performing translations . . . . .	V
<b>C</b>	<b>Code from Databases.gf</b>	<b>VII</b>

# List of Figures

2.1	The interface of pgAdmin when sending a query. . . . .	11
2.2	The result in pgAdmin of a join query. . . . .	12
3.1	The domain and language dimensions of .gf files used. . . . .	13
3.2	A map of the created grammar. . . . .	16
3.3	A tree depicting the abstract representation of the SQL query <b>SELECT</b> <b>name FROM countries WHERE area &gt; 5000000 ;</b> . . . . .	17
3.4	The components of the application. . . . .	22
3.5	The database selection page of the application . . . . .	25
3.6	The language adding page of the application . . . . .	26
3.7	The translation page of the application . . . . .	27
4.1	A map of the supported subset of SQL. . . . .	31
B.1	The database selection page of the application . . . . .	III
B.2	The language adding page of the application . . . . .	V
B.3	The translation page of the application . . . . .	VI



# List of Tables

- 5.1 A demonstration of some alternative ways of expressing a statement available in the English concrete syntax, for a database which has the table "countries" with the columns "capital", "currency" and "area". . . . 36



# 1

## Introduction

Databases are powerful tools which allow users to store information in a structured environment. A common way of interfacing with database implementations is through using the Structured Query Language (SQL). [7, 6] However SQL is a programming language using specific symbols and keywords, there are many people who do not possess a lot of knowledge about how programming languages work. Therefore, jumping into SQL can have a steep learning curve for many, and an alternative requiring less technical skill which feels more natural to people would be useful.

### 1.1 Accessing databases using natural language

An obvious way to make this technology more accessible and less cumbersome to use would be to give users the ability to express themselves using natural language. Natural language here refers to the languages people speak and use in real life, such as English, French, German, Finnish or Japanese as opposed to e.g. programming languages (such as C, Java or SQL) or formal languages used in mathematics. For the case of databases specifically, this means that a user would formulate a question or command in for instance English. This sentence would then be processed with the aim of fetching the requested information from a database or altering the contents of a database accordingly.

An approach to accomplishing that goal while still making use of the existing infrastructure of a database implemented using a distribution of SQL could be to use layers: First, a system which translates from a natural language to SQL. Second, an existing SQL distribution which performs the database accessing. The part which needs to be constructed is in that case that first layer which can translate from a natural language to SQL, along with a way of connecting the two layers. Additionally, if that first layer is also capable of translating in the opposite direction (from SQL to a natural language), that too could be useful for those who are unfamiliar with SQL specifically, or programming languages more generally. This is because translations in that direction could serve to explain SQL code to the user in a real life language they already master.

### 1.1.1 Usefulness

A system capable of performing translations between natural languages and SQL has a variety of usecases:

For people who are just starting out with learning the SQL language the system could help them by offering translation from a language they use much more commonly, and it could explain SQL queries to them using that same natural language. This could potentially aid in forming an understanding of how the SQL language works in terms of syntax, punctuation etc.

Those who use SQL but only rarely may find it easier to type queries in their natural language of choice over having to re-acquaint themselves with the SQL syntax each time they wish to access a database. Perhaps some who know SQL well still prefer expressing themselves in a natural language.

People who never had any interest in learning a programming language such as SQL may be more willing to start using databases in some capacity if they can do so entirely in their own natural language.

In a setting where it is infeasible that someone would learn SQL but allowing them some basic access to a database would still be beneficial. This could for example be a store offering customers access to information about their products, allowing the staff to spend less time answering simple questions by the customers. The customers could perhaps be introduced to how the system works by being shown a few examples of natural language inputs and the data shown as a result.

The ability to use natural language also opens up for the possibility of connecting speech-to-text and/or text-to-speech technology in order to enable voice input and simulated spoken audio output. This could further increase accessibility to using databases and could even speed up access if done well.

## 1.2 Approach

This project aims to construct a prototype of a system which can translate between natural languages on the one side and SQL queries on the other, to construct a natural-SQL translator.

### 1.2.1 Goals

In order to make this system as useful as possible a number of goals have been pursued in this project:

#### **Generality**

The system should not be shaped based on one or a small number of databases, or for one or a small number of natural languages. It should not be designed for



specific implementations, but instead take on a more general form. It should be database-agnostic and natural language-agnostic to as great an extent as is possible.

**Adaptability**

Where the system cannot be designed to work for any given database or with any given language without manual input, such inputs should be made as easy as possible to supply. Where the system is not able to be general, it should be easy to adapt.

**Accessibility**

Making a user friendly interface which accesses the system, not requiring the user to enter any commands or use any programming language. Someone unfamiliar with programming should actually get value out of it in accessing databases.

**Naturalness**

Users should be able to express themselves fairly naturally, and not have to think more than necessary about what inputs the system allows for on the natural language side.

**Coverage of SQL**

To demonstrate that a variety of features of SQL can be incorporated into the system, and successfully formulated in natural languages.

**1.2.2 Strategy**

The translator was constructed using the Grammatical Framework programming language (GF) (as is discussed further in section 2.2). This language was appropriate for the task at hand since GF is made for multilingual translations. It allows for the formulation of an abstract syntax of meanings and descriptions of how every such meaning of that grammar is expressed in each language (such as SQL, English or Swedish).

GF uses different modules for the different languages it is used with, and allows for inheritance between modules. This functionality is useful for the stated goals of shaping the translator to be general with respect to both languages and databases, since each new language-database combination can be treated separately in its own file.

In pursuit of the goal of naturalness, GF has the useful ability that it allows for multiple inputs to mean the same thing in the same language. This can be used to for instance say that both "display the population of the city" and "show me the population of the city" should be parsed to mean the same thing.

The construction of an application can make the translator easy to use and accessible, only needing to use natural languages for the whole duration of a user's interaction with the system. Some aspects of adapting the system to new databases and languages can be made more user-friendly in such an application.



# 2

## Theory

In this chapter some information is supplied which is useful in understanding the project.

### 2.1 Previous work

In the past, systems have been constructed which use a subset of natural language to extract information from a database. The creation of these systems involved several different approaches and used several different representations of the meaning of a given query.

The LUNAR system, created by Woods et al. in 1972, uses an ATN grammar to parse inputs to syntactic tree structures [33]. That system was entirely made with an English dictionary and English grammar rules. While this paper expresses an interest in a "general purpose natural language interface between men and machines" it only implements a system for the domain of lunar geography.

Another system was created by Warren and Pereira in 1982, which they implemented in the program Chat-80. They did not use SQL however, instead they used modified prolog code to represent the logical meanings of queries and to perform the processing and execution to find the answers to them [32]. While they started their work from an earlier project carried out in Spanish, they themselves only implement their system in English. While their system in itself is not general with respect to the domain, they do claim that that their system should easily be able to be adapted to other applications.

The attempts back in the 1980's tend not to support more than one language and are often only implemented for one domain.

In the 2010's products marketed to consumers as digital assistants, such as Apple's Siri, Google's Google assistant and Amazon's Alexa, came on the market. These assistants use mainly natural language voice input and attempt to accomplish a wide variety of tasks for the user. These tasks include among others: Setting a timer for a specific duration, making a phone call to a specific contact, controlling playback of movies or music, dictating a text message or asking for information such as weather or upcoming appointments. These assistants offer all of these features in one place which means that there are a lot of combinations of possible inputs. They do not

attempt a structured approach to a small subset of language with a well specified intent. These products have a much wider aim and also have a focus on voice input. They are therefore quite different from what is attempted in this project which is a structured approach to translation targeting only databases. Their presence nevertheless shows that natural language-computer interaction remains a relevant topic.

In a paper by Damova et al. translation is performed from natural languages to SPARQL, a semantic query language for databases. [20] In that project, Grammatical Framework is used as well.

## 2.2 Grammatical Framework

Grammatical Framework is the name of a functional programming language which is designed for "multilingual grammar applications". [9] A lot of information about the system can be found in the book about it by Aarne Ranta. [31] It allows for the formulation of grammars which can be used to translate between different languages (natural as well as non-natural).

Creating such a grammar takes on the form of first describing the structure of the new grammar in an "abstract syntax" file, this is essentially a formal description of all the meanings that can be formed within this grammar. [28] This abstract structure can then be given ways to express its meanings by creating one or more "concrete syntax" files. Every concrete syntax describes how each formable meaning in the corresponding abstract is expressed in one specific language. A concrete syntax only implements one abstract syntax. For a number of examples of the GF system at work, see the GF tutorial. [28]

As a simple example, a grammar for hamburgers could contain the following abstract syntax:

```
abstract Burger = {  
  
  flags startcat = Meal ;  
  
  cat Meal ; Ingredient ;  
  
  fun  
    Burger : Ingredient -> Meal ;  
    Beef, Chicken, Fish : Ingredient ;  
}
```

It could then have a concrete syntax for the English language which explains how to express every abstract meaning in English:

```

concrete BurgerEng of Burger = {

  lincat Meal, Ingredient = {s : Str} ;

  lin
    Burger ing = {s = ing.s ++ "burger"} ;
    Beef = {s = "beef"} ;
    Chicken = {s = "chicken"} ;
    Fish = {s = "fish"} ;
}

```

More similar concrete syntaxes could then be added, such as for Swedish:

```

concrete BurgerSwe of Burger = {

  lincat Meal, Ingredient = {s : Str} ;

  lin
    Burger ing = {s = ing.s ++ "burgare"} ;
    Beef = {s = "biff"} ;
    Chicken = {s = "kyckling"} ;
    Fish = {s = "fisk"} ;
}

```

Syntaxes such as these are individually contained in `.gf` files as separate modules, one for the abstract syntax and one for each concrete syntax.

In the abstract, categories are defined (in the example these are `Meal` and `Ingredient`), and a default start category is specified. These categories are in each concrete grammar given types which specify the representations of these categories in the relevant languages (in the example as `{s : Str}` which is a record containing a string).

Functions are declared with types in the abstract syntax, e.g.

```
Burger : Ingredient -> Meal ;
```

These functions are then implemented in each concrete syntax to return the appropriate values, for example

```
Burger ing = {s = ing.s ++ "burger"} ;
```

which says that the argument `ing` (of the type `Ingredient`) should first be linearized and from the resulting record the variable `s` should be used (`ing.s`) with the string `"burger"` to form the new record to return.

In GF it is also possible to create a grammar which extends another grammar through inheritance between modules. This makes all of the contents of the inherited module available in the inheriting module as well. This feature can be used to reduce redundant programming by including shared code in a more generic module which then is inherited by multiple more specific modules which supply only the additional functionality they need. Inheritance is performed in each module where it is desired, abstract as well as concrete. [28]

Using a grammar, three main operations are available: Parsing, linearization and translation. Parsing is to analyze the content of an input string according to one of the concrete syntaxes (languages) that have been implemented, parsing returns an abstract representation of the input, e.g:

```
parse -lang=BurgerEng "fish burger" -> Burger Fish
```

Linearization is to do the opposite of parsing, namely to take an abstract representation of a meaning and concretize it into a string in accordance to one of the concrete syntaxes (languages), e.g:

```
linearize -lang=BurgerSwe Burger Beef -> "biff burgare"
```

Translation involves parsing in one language and then feeding the resulting abstract representation into a linearization to another language. This process takes a string as an input and returns another string as output. [28] For example:

```
parse -lang=BurgerEng "chicken burger"  
| linearize -lang=BurgerSwe -> "kyckling burgare"
```

These translations can be performed in any direction, meaning any concrete syntax of the relevant abstract can serve as input language as well as output language.

These three operations can be performed from a command line environment using various commands specific to GF, but another way of accessing the translation potential is through the Portable Grammar Format (PGF). [30] This format is a way to take GF grammars and put them into a `.pgf` file which is easier to distribute and embed in applications. Such a file can be generated from a list of `.gf` files. This `.pgf` file can then be called directly from e.g. a command line environment to perform translations with an input string, a parsing language and a linearization language. Running such a command returns an output string in the indicated language. A `.pgf` file can also be accessed through functions included in the PGF package for haskell. Those functions can be used in applications, making it possible to perform translations without writing any manual commands anywhere.

### 2.2.1 Resource Grammar Library

When working with natural languages in GF, there is a library called the Resource Grammar Library (RGL) which contains functions and constants which are helpful in forming sentences in many natural languages. See the RGL synopsis for a guide to using this library. [11]

Within RGL, there are GF categories reflecting linguistic categories for words and sentences such as N for common nouns, V for most verbs, Prep for prepositions, NP for noun phrases and Utt for any utterance. There exist RGL dictionaries such as `DictGer.gf` or `MorphoDictEng.gf` which contain long lists of constants representing nouns, verbs and adjectives mainly, these constants contain information on how to correctly inflect the words.

Sentences can be formed by using constructors for various RGL types in combination with RGL constants. A few examples in English and Swedish:

```
mkNP the_Det house_N - Forms the English noun phrase "the house"
mkNP (mkNP the_Det number_N) (mkAdv possess_Prep (mkNP aPl_Det
house_N)) - Forms the English noun phrase "the number of houses"
mkNP the_Det hus_N - Forms the Swedish noun phrase "huset"
mkNP (mkNP the_Det antal_N) (mkAdv noPrep (mkNP aPl_Det hus_N))
- Forms the Swedish noun phrase "antalet hus"
```

In the second case `aPl_Det` turned houses plural and `possess_Prep` represents "of".

To use RGL, language specific modules need to be imported. In English for example, multiple files are needed depending on the functionality used. For this project the interesting ones include `SyntaxEng.gf`, `ParadigmsEng.gf`, `SymbolicEng.gf` and `MorphoDictEng.gf`, with similar lists of files for other languages.

The different parts of RGL have somewhat varied support for different natural languages. In total it covers the basic syntax of 38 languages, while `MorphoDict` has only been implemented for a handful of languages at the moment of writing. [11]

## 2.2.2 Functors

In GF, functors are a feature which allows for the creation of an incomplete syntax and the use of parameters to complete it in multiple different instantiations. [29]

This functionality can be used to take a number of similar concrete syntaxes and use their common parts to make one incomplete general functor. Then multiple instantiations can be made of that functor by using instances, only the definitions for those parts that differ are put in these instance files. This is useful in reducing the amount of redundancy in the code and in simplifying the addition of new concrete syntaxes.

With a functor, a number of files are used with different roles.

The central file is the functor file itself starting with the GF keyword `incomplete`. It is essentially a parameterized version of a concrete syntax. It is incomplete in that it uses certain constants which are not defined in the file itself, which instead need to be defined in separate files, the contents of these files can be seen as the parameters.

A functor is said to "open" one or multiple interfaces. These interface files start with the GF keyword `interface` and declare the undefined constants from the functor, for example:

```
about_Prep : Prep ;
```

The constants of such an interface are defined in instances, one for each interface and concrete syntax. Instances start with the GF keyword `instance` and define

the values of the declared constants from the interface in the desired language, for example:

```
about_Prep = P.mkPrep "about" ;
```

Each concrete syntax is then implemented by completing the incomplete functor with instances of the corresponding interfaces for the language in question, this is known as a "functor instantiation".

### 2.3 Databases

Databases are a way of storing and retrieving data in an organized, easily managed way. They often run on a computer system and are accessed to read information stored in them, write new information to them or edit existing data.

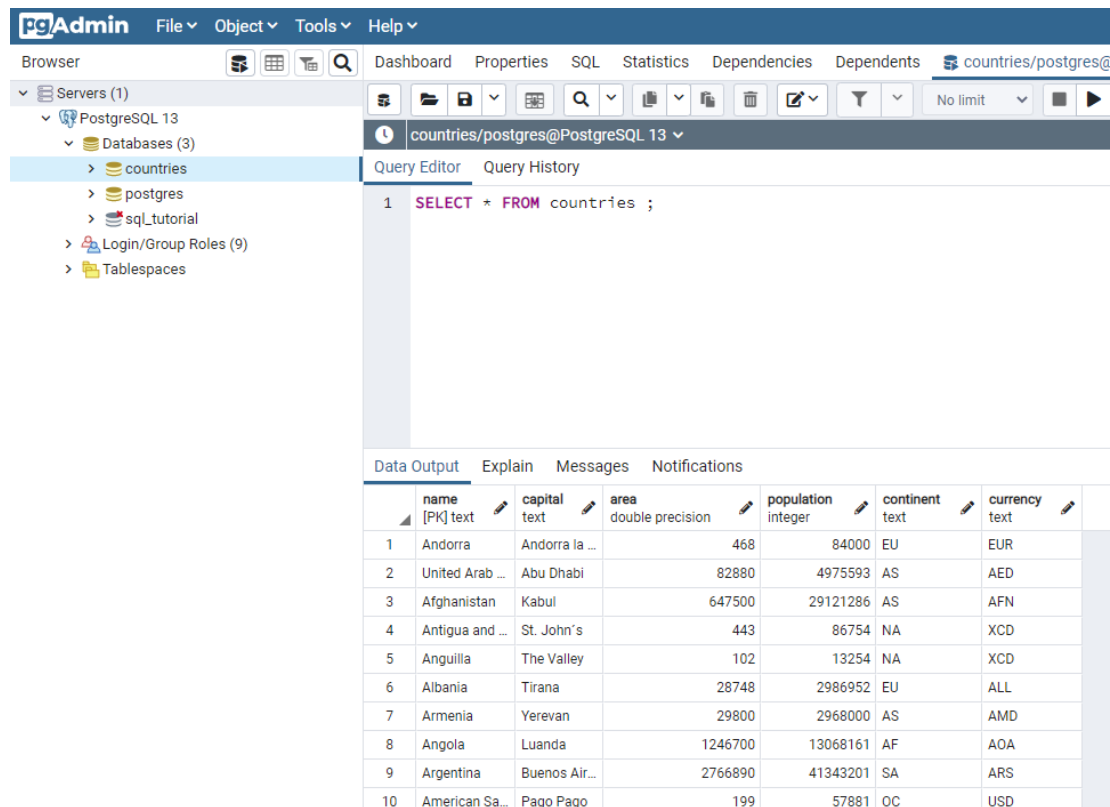
Most commonly, databases are controlled by database management systems (DBMS), and the contents are declared in a series of statements. [26] Relational DBMS are by far the most common type with one site ranking their popularity at 72.8% among all DBMS. [7] These databases contain a series of tables which have columns and rows/entries. The most common language used for writing to and reading from a database is Structured Query Language (SQL). All of the eight most common relational database management systems use SQL according to one source. [6] In SQL a user can specify formally what it is they want to add, remove, edit or view from the contents of the database. They do this with the use of reserved keywords in SQL such as **SELECT** for viewing or **DELETE** for removing data.

The database management systems used in order to manage a database come in various distributions developed and maintained by different companies, for example PostgreSQL, SQLite and Oracle's MySQL. The exact syntax of SQL varies slightly depending on the distribution used. In this project PostgreSQL was chosen as the only distribution to support. [22] PostgreSQL will in this report be interchangeably referred to simply as SQL since it's the only implementation used.

A useful tool for administering and accessing PostgreSQL databases is pgAdmin, which presents a visual interface to the user. [27] For an example of a query being sent to a database within pgAdmin, see figure 2.1.

In figure 2.1, we see at the bottom right of the screen a number of the rows in the "countries" table. In this table the columns present are **name**, **capital**, **area**, **population**, **continent** and **currency**. Each column has a datatype and the name column is marked as having the primary key ([PK]) and contains unique values for every entry in the table. Every entry is shown as a row and can have a value for each column in the table matching that column's datatype.





**Figure 2.1:** The interface of pgAdmin when sending a query.

Many different queries can be formed in SQL. The query in the input field of figure 2.1 instructs PostgreSQL to show all the rows and columns of the countries table. If only some columns are to be shown they can be chosen:

```
SELECT name, capital FROM countries ;
```

A condition can be placed on which rows are to be shown:

```
SELECT * FROM countries WHERE continent = 'EU' ;
```

Entries can be deleted from a table with the DELETE keyword:

```
DELETE FROM countries WHERE ...
```

Entries can have one or more field changed:

```
UPDATE countries SET capital = 'Gothenburg' WHERE name = 'Sweden' ;
```

Entries can be added to a table:

```
INSERT INTO countries VALUES ('Gondor', 'Minas Tirith') ;
```

There are also more complex operations such as subqueries where an entire SELECT statement is used in a WHERE clause or is inserted into a table:

```
SELECT * FROM countries WHERE capital IN (SELECT name FROM countries) ;
```

Another piece of more advanced SQL are joins which connect multiple tables, by matching the specified columns, and perform a SELECT statement on the larger resulting table, see figure 2.2:

```
SELECT * FROM countries FULL JOIN currencies
ON countries.currency = currencies.code ;
```

## 2. Theory

```
1 SELECT * FROM countries FULL JOIN currencies ON countries.currency = currencies.code ;
```

Data Output		Explain	Messages	Notifications					
	name text	capital text	area double precisic	population integer	continent text	currency text	code text	name text	
1	Andorra	Andorra la ...	468	84000	EU	EUR	EUR	Euro	
2	United Arab E...	Abu Dhabi	82880	4975593	AS	AED	AED	Dirham	
3	Afghanistan	Kabul	647500	29121286	AS	AFN	AFN	Afghani	
4	Antigua and B...	St. John's	443	86754	NA	XCD	XCD	Dollar	
5	Anguilla	The Valley	102	13254	NA	XCD	XCD	Dollar	
6	Albania	Tirana	28748	2986952	EU	ALL	ALL	Lek	
7	Armenia	Yerevan	29800	2968000	AS	AMD	AMD	Dram	
8	Angola	Luanda	1246700	13068161	AF	AOA	AOA	Kwanza	
9	Argentina	Buenos Air...	2766890	41343201	SA	ARS	ARS	Peso	
10	American Sam...	Pago Pago	199	57881	OC	USD	USD	Dollar	
11	Austria	Vienna	83858	8205000	EU	EUR	EUR	Euro	
12	Australia	Canberra	7686850	21515754	OC	AUD	AUD	Dollar	
13	Aruba	Oranjestad	193	71566	NA	AWG	AWG	Guilder	
14	Aland Islands	Mariehamn	1580	26711	EU	EUR	EUR	Euro	
15	Azerbaijan	Baku	86600	8303512	AS	AZN	AZN	Manat	
16	Bosnia and He...	Sarajevo	51129	4590000	EU	BAM	BAM	Marka	
17	Barbados	Bridgetown	431	285653	NA	BBD	BBD	Dollar	
18	Bangladesh	Dhaka	144000	156118464	AS	BDT	BDT	Taka	
19	Belgium	Brussels	30510	10403000	EU	EUR	EUR	Euro	

Figure 2.2: The result in pgAdmin of a join query.

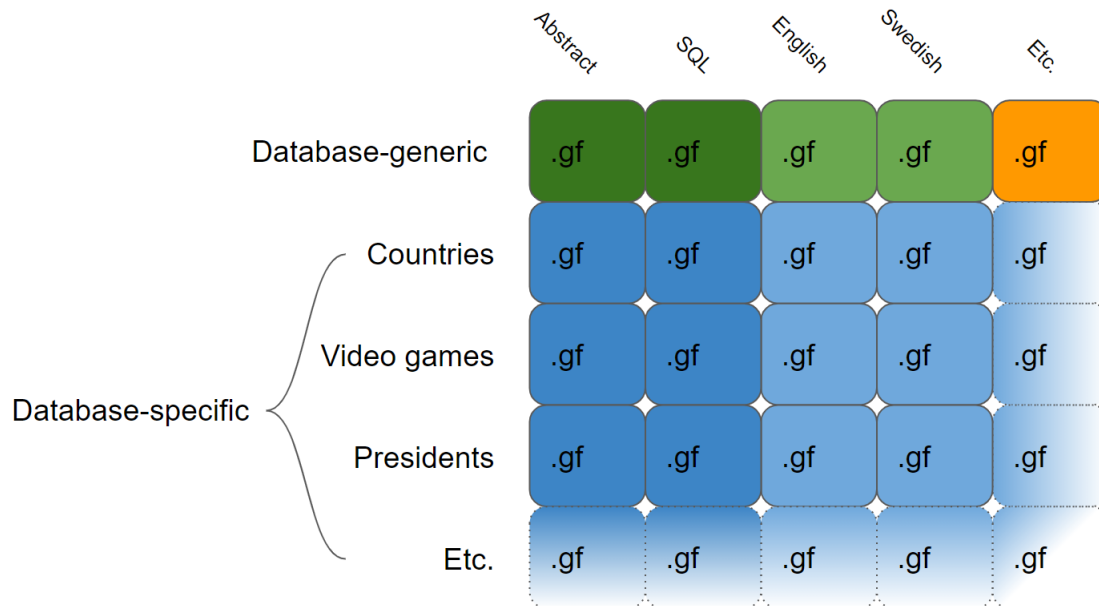
# 3

## Methods

In this chapter, the work carried out as part of the project will be presented. This work is centered around creating a system for translation between natural languages and SQL, which chiefly takes the form of two activities: Structuring and implementing the GF grammar at the core of the system, as well as designing and programming a user interface to the system.

### 3.1 Grammar files

The whole system relies on GF and the `.gf` files used are illustrated in figure 3.1. These gf files are divided into abstract and concrete, together they describe a grammar for queries into databases and how to express the meanings of this grammar in both SQL and in natural languages.



**Figure 3.1:** The domain and language dimensions of `.gf` files used.

The central `.gf` file is in the top left corner of figure 3.1 and is called "Databases.gf", it describes the abstract structure of the created grammar. The contents of that file are visible in appendix C. The Database-generic SQL file, called "DatabasesSQL.gf",

is a concrete implementation of "Databases.gf". It describes how to say the meanings of the grammar in the SQL language. The Database-generic English file, called "DatabasesEng.gf", is another concrete implementation of "Databases.gf". Similarly, this file specifies how one can say the formable meanings of "Databases.gf" in English. For Swedish or other natural languages, the equivalent files have the same purpose as in English.

All the files in the top row of figure 3.1 are database-generic, this means that they look the same for any database being used with the system. Any information that differs based on which database is being used can instead be found in the blue database-specific files. Each of the blue files inherits from the database-generic file at the top row in the same column, meaning these files provide additional functionality on top of the database-generic part.

All of the database-generic `.gf` files are located in one directory, while the database-specific `.gf` files are separated into one directory per database. These directories have very specific names given based on this structure. All of the files and all of the functions within the application expect this to be the case.

On how each file is created: The green files in figure 3.1 have been created manually as part of the project. The orange file represents that if additional natural languages are added then the corresponding database-generic `.gf` files have to be manually added as well. The blue files are not meant to be written manually, but are instead generated by the application as the user provides the necessary information.

The dark columns in figure 3.1 represent language-agnostic files which don't have anything to do with any specific natural language, while the light columns do. For light blue files, which are each specific to one database and to one natural language, information is required about what words in that language the user will use for elements in that database. That information is provided by the user through the application. Approximately we can say that we have one row per database, and one column per language, which reflects two dimensions along which generality is desired.

## 3.2 Grammar contents

The contents of the grammar will be described in this section. First the abstract `.gf` files, then concrete files for SQL and finally the concrete files for natural languages. For each of these, database-generic as well as database-specific files will be discussed. The section on GF (Section 2.2) may be useful to following these descriptions.

### 3.2.1 Abstract

The abstract grammar is divided into a database-generic part and the database-specific modules (one per database).

The database-generic abstract `.gf` file is called "Databases.gf", whose code is visible in appendix C. This file contains the central description of what meanings are able to be formulated in the Databases grammar. It does this by declaring categories of abstract meaning and functions whose types are those categories. The grammar has the start category of `Statement` since that is the category we parse from and linearize to.

Some examples of constructors from the code are:

```
StQuery : Query -> Statement ;
QSelect : ColumnPart -> FromLimPart -> PredicatePart
        -> OrderPart -> Query ;
QUnion  : Query -> Query -> Query ;
JInner, JLeft, JRight, JFull : JoinType ;
```

The meanings of these constructors are as follows:

`StQuery` takes something of the type `Query` and forms a `Statement`, thus if `q1` is a `Query`, then `StQuery q1` is a `Statement`. Similarly, `QSelect` takes a `ColumnPart`, a `FromLimPart`, a `PredicatePart` and an `OrderPart` and forms a `Query`. `QUnion` forms a `Query` by recursively taking two `Query`-s. `JInner` takes no arguments and forms a `JoinType`, the same is true for `JLeft`, `JRight` and `JFull`.

For a map of the Databases grammar, see figure 3.2. The way to read this figure is explained here: A `Statement` can be formed among other ways by `StQuery` of a `Query`, or by `StInsert` of a `Table` and one of seven different options. The first of these options is formed by `IColValOne` of `InsertColWith` which in turn takes a `Column` and a `Value`. A `Value` can either be formed by `ValInt` taking an integer, or by `ValStr` taking a string enclosed in single quotes. And so on, it serves as a visual overview of the Databases grammar.

An example, take the statement representing the SQL query:

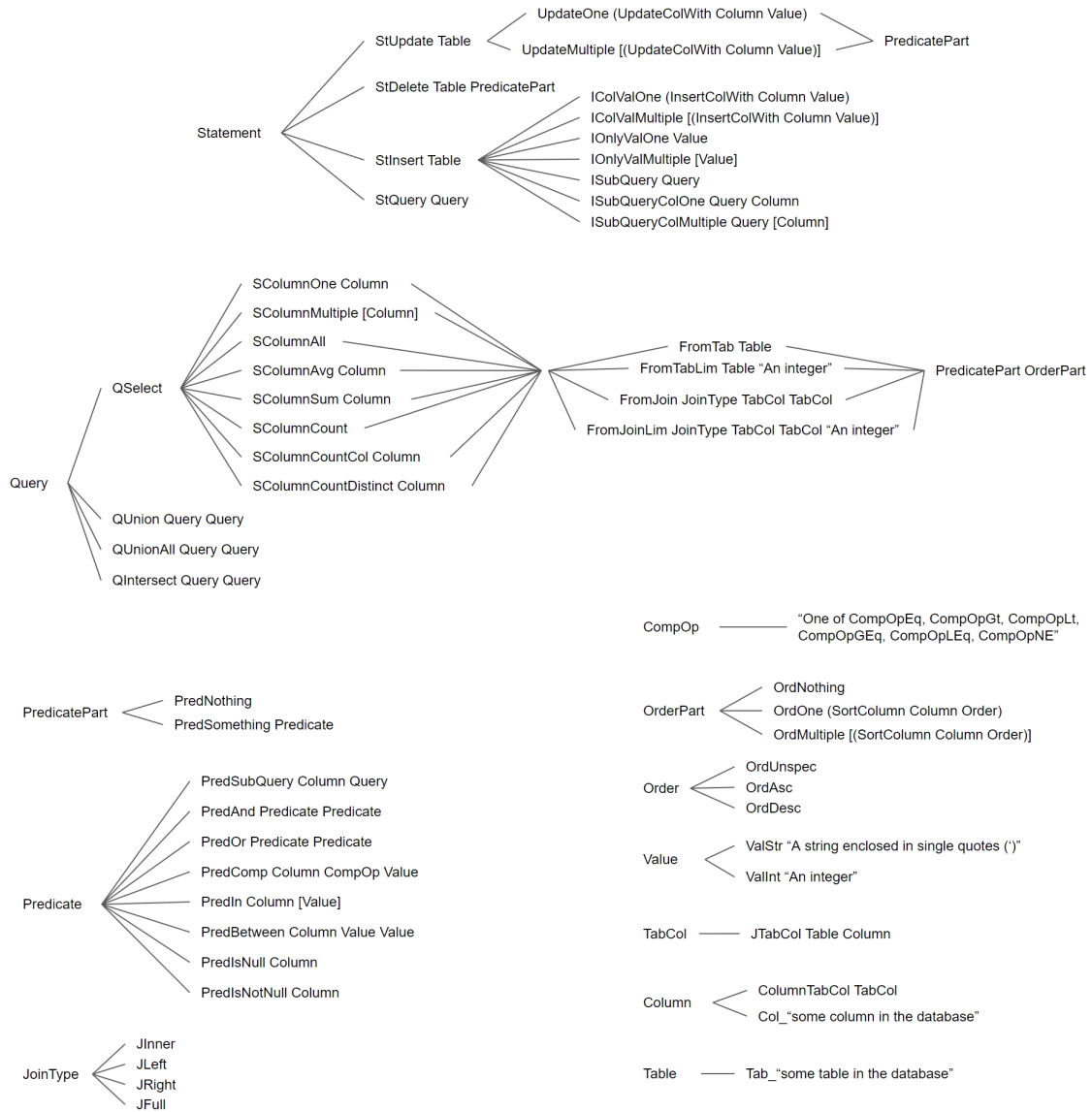
```
SELECT name FROM countries WHERE area > 5000000 ;
```

This would be formed by starting with the constructor `StQuery` and giving it a `Query`. This `Query` is formed by the constructor `QSelect` which in turn takes a `ColumnPart`, a `FromLimPart`, a `PredicatePart` and an `OrderPart`, each of which are formed by relevant constructors in the abstract grammar. The abstract representation of the whole statement can be written as:

```
StQuery (QSelect (SColumnOne Col_name) (FromTab Tab_countries)
  (PredSomething (PredComp Col_area CompOpGt (ValInt 5000000)))
  OrdNothing)
```

But it can also be represented as its own tree starting at the root of the start category of the grammar which is `Statements`, where each node is a function or a constant used in forming the abstract meaning, such a tree for this statement is shown in figure 3.3.

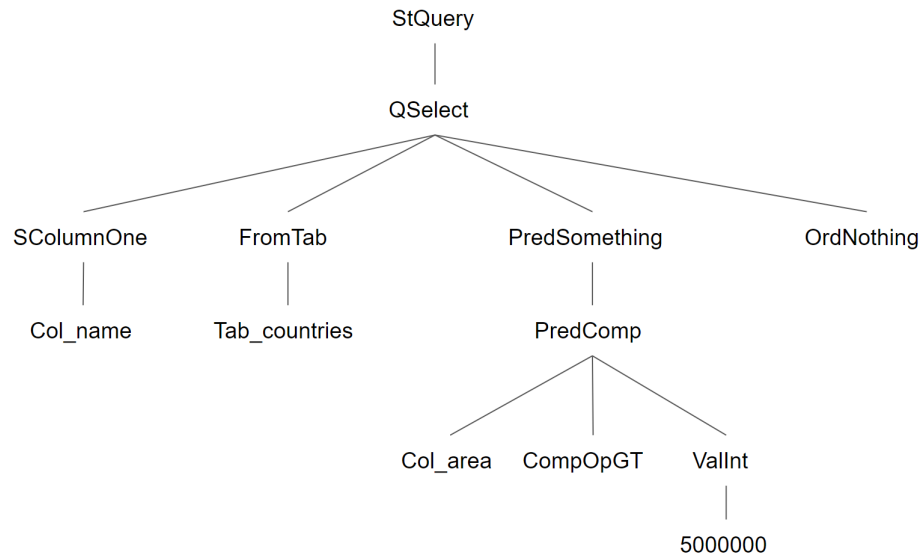
### 3. Methods



**Figure 3.2:** A map of the created grammar.

All legal Statements can be represented in this fashion by starting at the start category of the grammar and inserting the appropriate constructors to form subtrees repeatedly. While the "Databases.gf" file defines the categories Column and Table, these categories are mainly formed from declarations in the database-specific abstract files which inherit from "Databases.gf". All the parts needed in the abstract which are the same no matter the database are present in "Databases.gf" while those parts that differ, namely what columns and tables any one specific database contains, can be found in database-specific abstract files. Each database-specific abstract gf file is given its name based on the name of the respective database according to the following formula:

DB ++ name\_of\_the\_database ++ .gf



**Figure 3.3:** A tree depicting the abstract representation of the SQL query `SELECT name FROM countries WHERE area > 5000000 ;`.

Since the columns and tables can be declared without any more information required, these files basically just list the tables and columns which are in a database.

As an example, the database-specific abstract `.gf` file for the database named "countries" would be called "DBcountries.gf", and look somewhat like what is shown here:

```

abstract DBcountries = Databases ** {

  flags startcat = Statement ;

  fun

    Col_area, Col_capital, Col_code, Col_continent, Col_currency,
    Col_name, Col_population : Column ;

    Tab_countries, Tab_currencies : Table ;

}

```

Since these files inherit "Databases.gf" (as seen from the notation `Databases **`) they contain all the necessary parts to form the meanings of queries into the relevant database.

#### 3.2.2 SQL

In the concrete grammar for SQL, most parts are fairly straightforward because of the fact that the whole grammar borrows its structure from the SQL functionality it implements. These files related to SQL are found in column number 2 in figure 3.1.

While the meanings which are formable in the grammar are defined in the abstract files, the way to express each of these meanings in SQL is given in files which follow the same division of database-generic and database-specific. For SQL the database-generic file is called "DatabasesSQL.gf" and for an example database called "countries" the database-specific file is called "DBcountriesSQL.gf".

The database-specific file once again inherits from the database-generic one and thus contains all the information needed to parse from and linearize to SQL queries related to the database in question. The database-specific SQL file implements the database-specific abstract file, and the database-generic SQL file implements the database generic abstract file.

Thus a file like "DBcountriesSQL.gf" contains descriptions for each column and table in the database of how to say their names in SQL. For instance:

```
Col_capital = "capital" ;
```

That code represents that the way to express the name of the column `capital` from the "countries" database in SQL is precisely as the string "capital".

In "DatabasesSQL.gf" the way to write everything from the database-generic grammar in SQL is described. For example:

```
QUnion q1 q2 = q1 ++ "UNION" ++ q2 ;
```

That line describes that the way to linearize the union of two queries is to concatenate the result from linearizing the first query, the string "UNION", and the result from linearizing the second query. Thus the linearization of a statement which can be seen as a tree as in figure 3.3 entails linearizing smaller components or sub-trees. This happens repeatedly until a leaf is reached in the tree which can be linearized on its own, for instance the following code which returns a string directly:

```
SColumnAll = "SELECT *" ;
```

The concrete grammar files for SQL in this way describe how to put together an SQL query by breaking the problem down into smaller pieces.

#### 3.2.3 Natural languages

Similarly to SQL, each natural language also has concrete files describing how meanings are expressed in those languages. For English this is the database-generic "DatabasesEng.gf", and for a database called "countries" the database-specific "DBcountriesEng.gf". For Swedish the equivalent files similarly are "DatabasesSwe.gf" and "DBcountriesSwe.gf". Much like for SQL these files describe how to linearize for every function declaration in the abstract grammar. For natural languages the resource grammar library was used, see section 2.2.1 for information about this library.



### 3.2.3.1 Resource grammar library

Each category in the abstract linearizes to one or a collection of variables of RGL types. Commonly used such types in the system's grammar include e.g. NP for noun phrase, Adv for verb-phrase-modifying adverb, Prep for preposition and Det for determiner phrase. Section 2.2.1 may be useful to understanding RGL.

In the database-specific natural language `.gf` files, columns linearize to CN (common noun (without determiner)) and tables linearize to N (common noun). In both of these cases this is accomplished with the help of `mkN` from the Paradigms module for the given natural language. This constructor takes as arguments some forms of the desired noun (and in some languages the noun's gender) and forms a variable of type N representing that noun. As an example, consider the database "america" which contains the tables "presidents", "vice\_presidents" and "states", the first two of which have columns called "party" for the political party of the corresponding politician. This column is formed in the following way:

```
Col_party = mkCN (P.mkN "party" "parties") ;
```

In the database-generic "DatabasesEng.gf" and "DatabasesSwe.gf" the linearization categories have been chosen among the available ones in RGL in ways which make sense considering the contents of each category and how it is used elsewhere in the grammar. For instance `Query`-s are used in combination with other `Query`-s to form unions and intersections, but are also used to form a statement or as a subquery in a where clause or in an insert statement. `Query`-s contain information about what information is desired to be used out of a table or tables. This all taken together led to the decision to chose to linearize a `Query` to an NP, since it can be used in all of those ways using various grammatical constructions as needed. Similar reasoning was applied to the other categories in the grammar.

Each meaning in the grammar can be seen as a tree, for one such example see figure 3.3. What "DatabasesEng.gf" does is that it describes how to linearize each tree to English by using the results from linearizing its sub-trees. Thus recursively the result for the whole tree can be found. This same description is also sufficient to enable parsing from an English string to the abstract tree. With RGL the linearizations are accomplished with the help of constructors and constants from modules such as "SyntaxEng.gf", "ParadigmsEng.gf", "StructuralEng.gf" and "MorphoDictEng.gf". The equivalent description also applies for Swedish.

Here is an example of using RGL in English:

A `PredicatePart` linearizes to an Adv and can represent e.g.

*where the population is above 5000000*

This is formed by the constructor

```
PredSomething p = mkAdv (P.mkSubj "where") p ;
```

which takes a Predicate which linearizes to an S. The predicate in this case represents

*the population is above 5000000*

which is formed by the constructor

```
PredComp c compOp v = mkS (mkCl (mkNP the_Det c) (mkAdv compOp v)) ;
```

which uses a `Column`, a `CompOp` and a `Value` which linearize to a `CN`, a `Prep` and an `NP` respectively. These parts are then used to construct an `S` with the help of the constructors `mkS`, `mkCl`, `mkNP` and `mkAdv` as well as the constant `the_Det` representing the determiner "the".

#### 3.2.3.2 Variants

In order to allow the user more variety in their entry of queries in natural languages, the GF feature of variants has been employed. With this feature, for some constructors, multiple ways of linearizing for the same input are described. In these cases, the first one listed is used for linearizing to, but for parsing any of them are accepted.

```
StQuery q = variants {
  mkUtt q ;
  mkUtt (mkImp (P.mkV2 D.display_V) q) ;
  mkUtt (mkImp (P.mkV2 D.show_2_V) q) ;
  mkUtt (mkImp (P.mkV2 (P.partV D.show_2_V "me")) q) ;
  mkUtt (mkQC1 (what_IP) q)
} ;
```

For example the constructor `StQuery` shown above allows the user to enter any of the following for the database "america":

*the states where ...*  
*display the states where ...*  
*show the states where ...*  
*show me the states where ...*  
*what are the states where ...*

All of these sentences are parsed to mean the same thing in the abstract grammar, and thus are translated to the same SQL queries:

```
SELECT * FROM states WHERE ...
```

#### 3.2.3.3 Functor

An alternative way of structuring the database-generic natural language part of the system was with the use of the GF feature of functors (as described in section 2.2.2). This approach was partially implemented but not fully incorporated with the system. Variants were not included in the functor implementation.

The functor implementation replaces the database-generic natural language concrete syntax files with the following set of files:

The incomplete functor `DatabasesFunctor.gf`, the interface of what is missing from the functor `DatabasesInterface.gf`, for each language an instance of the interface e.g. `DatabasesInterfaceEng.gf`, and finally for each language an instantiation of the functor e.g. `DatabasesEng.gf` which uses the corresponding instance.

The file `DatabasesFunctor.gf` is similar to the non-functor database-generic concrete natural language `.gf` files except for that it has undefined constants in place of certain language-specific words from RGL modules. For example where in the non-functor `DatabasesEng.gf` file one variant for `SColumnAll` is:

```
{np = mkNP all_Predet (mkNP D.info_N) ; prep = P.mkPrep "about"} ;
```

The equivalent code in `DatabasesFunctor.gf` instead looks like this:

```
{np = mkNP all_Predet (mkNP info_N) ; prep = about_Prep} ;
```

In this case `info_N` and `about_Prep` are constants which are not defined in the file `DatabasesFunctor.gf`, they are part of what makes the functor "incomplete", they are left to be defined by instances of its interface.

The undefined constants used in the functor are declared in the `.gf` file for the interface of that functor: `DatabasesInterface.gf`. This file therefore simply contains the names and types of those constants, an excerpt from this file looks like this:

```
interface DatabasesInterface = open Syntax in {
oper
  show_V2 : V2 ;
  info_N : N ;
  about_Prep : Prep ;
  where_Subj : Subj ;
  ...
```

For the interface `DatabasesInterface.gf`, each natural language implements an instance. For English, that instance is `DatabasesInterfaceEng.gf`. This file contains definitions for the constants of `DatabasesInterface.gf` which are formed using RGL functionality, see here an excerpt from the English instance:

```
instance DatabasesInterfaceEng of DatabasesInterface = open
  SyntaxEng,
  (P = ParadigmsEng),
  (I = IrregEng) in {
oper
  show_V2 = P.mkV2 (P.mkV "display") | P.mkV2 I.show_V ;
  info_N = P.mkN "info" ;
  about_Prep = P.mkPrep "about" ;
  where_Subj = P.mkSubj "where" ;
  ...
```

The concrete syntaxes which are accessed by GF for each language such as the English `DatabasesEng.gf` are then instantiations of the functor `DatabasesFunctor.gf` using RGL modules and the previously discussed instance for the same language:

```
concrete DatabasesEng of Databases = DatabasesFunctor with
  (Syntax = SyntaxEng),
  (Symbolic = SymbolicEng),
  (DatabasesInterface = DatabasesInterfaceEng)
;
```

If a language does not exactly conform to the structure of the functor, exceptions can also be given, basically overwriting the functionality of the corresponding con-

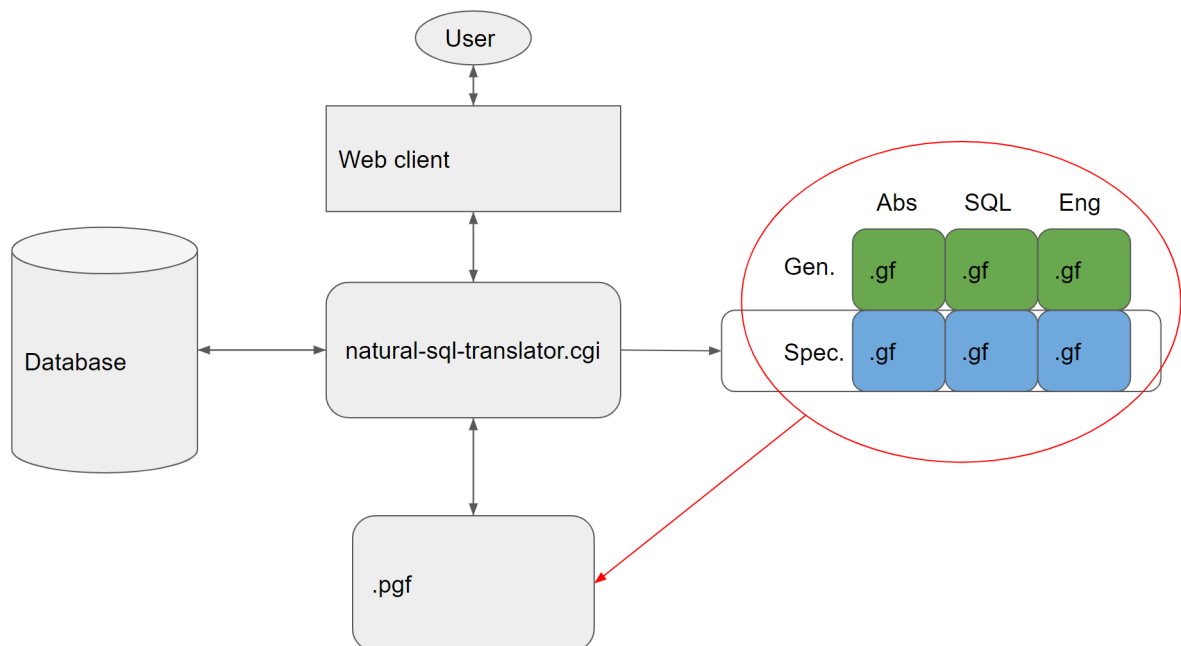
structors defined in the functor. For example in Swedish `LimNone` ended up differing from the functor implementation, therefore an exception was used as seen here in the swedish instantiation `DatabasesSwe.gf`:

```
concrete DatabasesSwe of Databases = DatabasesFunctor - [LimNone]
with
  (Syntax = SyntaxSwe),
  (Symbolic = SymbolicSwe),
  (DatabasesInterface = DatabasesInterfaceSwe)
** {
-- exceptions to functor
lin
  LimNone t = mkNP all_Predet (mkNP aPl_Det t) ;
}
```

## 3.3 Application

What has been described in previous sections details a GF grammar which enables translation between SQL and natural languages. Additionally this project has produced an application which at its core uses that GF grammar. The application allows users access to the translation potential of the "Databases" grammar. This section will describe various parts of this application.

The application consists of, and interacts with, a number of components. The way these components fit together is illustrated in figure 3.4.



**Figure 3.4:** The components of the application.

The central component of the application is a CGI executable which is hosted on a web server and accessed by the user via a web browser.

This `.cgi` file is compiled from a haskell source file which is where the application was written. This haskell source file makes use of a number of different packages and functions in order to facilitate the desired functionality. Among this functionality there is: Presenting the user with the correct information (and the available options), accessing the database of choice, writing `.gf` files, generating `.pgf` files, accessing `.pgf` files in order to carry out the actual translation, and printing out results returned from the database.

A guide to how the user can install the application is available in Appendix A. A guide to using the application is available in Appendix B.

### 3.3.1 Hosting

The central `.cgi` file is made available to the user by being hosted on a web server, this can for example be done locally using IIS in windows. [25] In order to ensure proper functionality, some permissions may need to be changed for the relevant folders, so the server can write and modify `.gf`, `.gfo` and `.pgf` files as expected. The folder where the `.cgi` file is located can be added as a virtual directory to the server to make it reachable by the client.

The user accesses the server by running a client in their web browser. They navigate to the URL where the server is hosted and specifically to the virtual directory and the `.cgi` executable for the application, e.g.

`http://localhost/cgi-bin/ext-app/natural-sql-translator.cgi`

That will present the user interface and allow the user to begin using the application (illustrated by the topmost part of figure 3.4).

### 3.3.2 CGI and XHTML

The `.cgi` file accessed by the user reacts when the buttons on the different pages are clicked by performing the appropriate actions and then serving the user with the next page they should be seeing. This functionality was implemented by using the `Network.CGI` haskell package. [1]

The different pages of the application and the elements on those pages were implemented using various functionality from the `Text.XHtml` haskell package, notably forms for getting the relevant data to the server. [19] Before a page is shown to the user, the server parametrically produces the page to be shown based on previously entered data. This can be in terms of what fields should be present on the page, what options should be available to the user or what choices should be pre-selected. When one of the relevant buttons is pushed, the data in form fields are collected and sent to the server. The layout of the pages of the UI can be seen in figures 3.5, 3.6 and 3.7.

### 3.3.3 Reading and writing GF files

One function used as part of the application was to scan a directory in order to see what `.gf` files are available. This is performed by using the function `listDirectory` from the haskell package `System.Directory`. [17]

The application comes equipped with a number of functions to write database-specific `.gf` files. This operation is represented in figure 3.4 by the grey arrow from the center pointing toward the right. These functions take parameters concerning the contents of the new `.gf` file and format it appropriately. The file is written or overwritten to the directory of the given database using the `writeFile` function of the `Data.Text.IO.UTF8` haskell package. [5]

In order to generate a `.pgf` file for a given database, the application first looks up what database-specific `.gf` files are available in the corresponding directory. Next, it runs a command to instruct GF to generate a `.pgf` file from the found file names by using the `runProcess` function of the `System.Process` haskell package. [18] This newly generated or overwritten `.pgf` file is placed in the directory for the database in question. This function is symbolized by the red circle and arrow in figure 3.4.

The `.pgf` files are accessed when a translation is performed on a given input string with a given input- and output language (illustrated by the bottom part of figure 3.4). This is done through the use of functions in the PGF haskell package, mainly the functions `readPGF`, `parse` and `linearize`. [16]

Before and after sending strings through the translation process, they are sent through manually constructed lexer functions. These functions change capitalization and spacing around punctuation so that GF can parse them properly. They also ensure that values (`ValStr`) in the grammar are properly translated.

### 3.3.4 Database access

The application has functions to access the database the user has chosen (illustrated by the leftmost part of figure 3.4). This access is used for two purposes: First, to compile two lists: A list of all the tables in a given database, as well as a list containing the columns of all those tables. Second, to send an SQL query to the given database and fetch the results from that query, and optionally to commit the changes made to the database.

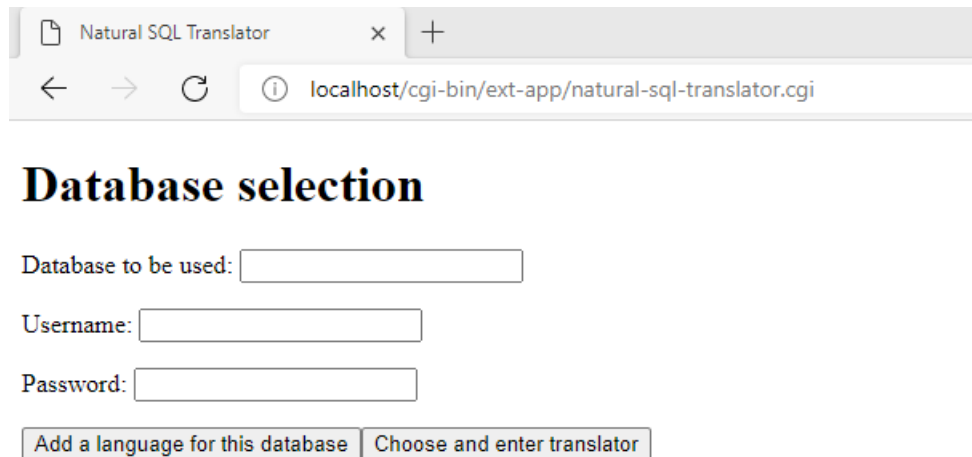
Whether a given user has the rights to access some database, table or column is regulated by PostgreSQL using a username and password provided by the user.

If the result from the database is just the number of rows affected (as in the case of sending an INSERT statement for example) that gets displayed, if it is a table it gets formatted and displayed on the page as a table in html, if it is an error the error message gets displayed instead. Functions such as `connectPostgreSQL`, `catchSql` and `sFetchAllRows` from the `Database.HDBC` and `Database.HDBC.PostgreSQL`

haskell packages are used to accomplish this functionality. [3, 4]

### 3.3.5 Application walkthrough

This section of the report presents the functionality of the application from the perspective of someone using it, explaining certain elements of pages and sequences of actions. Also included in this report is a user guide in appendix B.



**Database selection**

Database to be used:

Username:

Password:

**Figure 3.5:** The database selection page of the application

Upon booting up the application, the user is greeted with the database selection page, with input fields for which database they wish to use as well as for a username and a password, as seen in figure 3.5. The user can at that point type the name of their locally hosted database and a username and password for a PostgreSQL user. What data is accessible to what user is handled by PostgreSQL using this information.

They can then click one of two buttons:

- "Add a language for this database" contacts the database and fetches lists of the tables and columns therein, the user is then presented with the language adding page (as seen in figure 3.6).
- "Choose and enter translator" generates the `.pgf` file for the chosen database and presents the translation page (as seen in figure 3.7).

The language adding page (as seen in figure 3.6) has a drop down list containing the database-generically supported languages. This list allows the user to choose which language to add support for with respect to the chosen database specifically. This page also contains fields in which the user can enter the desired names of columns and tables in singular and plural forms in the selected language. The user can choose to go back to the database selection page discarding the entered data or submit it and proceed to the translation page.

Natural SQL Translator

localhost/cgi-bin/ext-app/natural-sql-translator.cgi?page\_id=database&db\_choice=countries&

## Language adding

Language to add for the countries database: Eng ▼

Please type names in that language (in singular and plural) for the tables and columns of the countries database:

Tables:

	Singular	Plural
countries		countries
currencies		currencies

Columns:

	Singular	Plural
area	area	
capital	capital	
code	code	
continent	continent	
currency	currency	
name	name	
population	population	

Clear all Choose another database

Submit and enter translator

**Figure 3.6:** The language adding page of the application

If they proceed, the entered information is used to create a database specific `.gf` file for the selected natural language. At this point, if they didn't exist already, an abstract and a concrete SQL `.gf` file are also created for this database. After that, a `.pgf` file is generated for the database and the translation page (figure 3.7) is shown.

The translation page (as seen in figure 3.7) contains an input and an output field, drop down lists for which language to translate both from and to, and a "translate" button. The drop down lists only contain the languages which are available for this database specifically. The user chooses languages and enters their query in the input field and then they click "translate". This button takes the input string and feeds it through the `.pgf` file with the chosen languages, the output of which is printed to the output field.

If the output language is set to SQL, the user also has the option of sending the output SQL statement to the database manager using the "Send to DB" button. If they so choose the result from the database manager is rendered at the bottom of the screen. Whether or not the changes to the database are committed is controlled by a checkbox.



The screenshot shows a web browser window with the title "Natural SQL Translator". The address bar displays the URL: `localhost/cgi-bin/ext-app/natural-sql-translator.cgi?page_id=database&db_choice=countries`. The main heading is "Translation". Below it, the text "Using the database: countries" is followed by two buttons: "Choose another database" and "Add a language for this database". The "From:" field is set to "Eng" and the "To:" field is set to "SQL". There is a large text input area for the user's query. Below this is a "Translate" button. The "Output:" section contains a large text area for the translated SQL. At the bottom, there is a section for database interaction: "Send SQL code to countries database:" followed by a "Send to DB" button and a checkbox labeled "Also commit changes:". Below this is a box titled "Output from the database" which currently displays "Nothing".

**Figure 3.7:** The translation page of the application

The user additionally has two buttons for leaving the translation page:

- "Choose another database" sends them to the database selection page.
- "Add a language for this database" sends them to the language adding page.



# 4

## Results

The project produced a system, taking on a general form, which is capable of translating between a natural language and SQL in either direction.

At the core of the system lies the "Databases" GF grammar with the central component "Databases.gf". The code in this file is available in appendix C. For a map of the structure of the grammar see figure 3.2.

The application implemented as the interface of the system allows users to perform translations with the grammar, access their database and adapt the grammar to new databases.

### 4.1 Assessment of goals

For each of the goals listed in section 1.2.1, below is a presentation of how well they were accomplished.

#### **Generality**

The design of the system along with its core, consisting of the code for the application as well as the gf files "Databases.gf" and "DatabasesSQL.gf", takes on a form which is both language- and database-agnostic. Everything in the GF grammars was able to be placed in the database-generic files except for the declarations of the columns and tables which are inherently specific to each database.

The system is mainly general with respect to language, but GF files do need to be manually written for the addition of new languages.

#### **Adaptability**

All of the GF code which is language and/or database-specific is entirely separated into distinct files following the structure in figure 3.1. Adding more of these files does not affect the core of the system, but simply makes more data available to the application.

New databases can be used out-of-the-box with the application, without editing any files. For each generically supported language which is to be added for a specific database, the user needs to enter some information on the language adding page of the application, see figure 3.6. On this page they need to give the names of

the database's columns and tables, in singular and plural form, in the new language.

A new concrete GF syntax has to be implemented for each new natural language used with the system. If the alternative functor implementation is used, it is instead only a new instance of the functor's interface which needs to be written. In the current version, the functor file has 176 lines of code and has 19 parameters in the interface file.

If support for more of SQL is desired, the central files of the system do need to be edited.

### **Accessibility**

An application was created which, once installed, does not require the entry of any commands or the use of any programming language. It does not require the use of any language other than the natural languages the user wishes to employ (and English, for the application's UI).

The application makes the translation potential of the GF files accessible through a graphical user interface. This means that the user can translate between SQL and the implemented natural languages. This interface can access databases and display results for the user to see.

### **Naturalness**

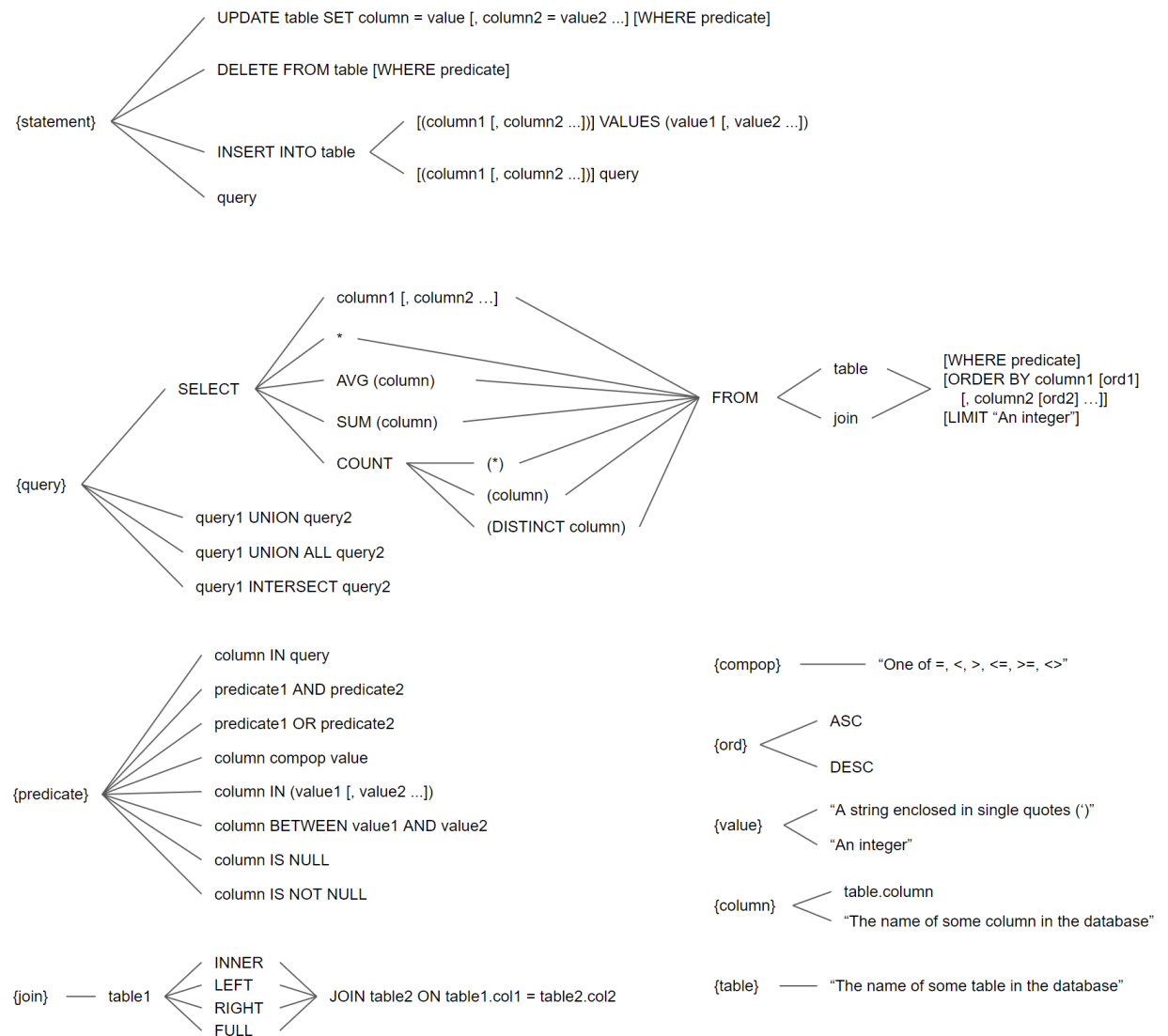
When using a natural language as input, the user has some flexibility available to them in the form of there being alternative ways implemented of expressing each statement. The number of alternatives varies by language used and by the specific statement in question. In order to add more alternatives, more options can be listed in the variants featured in the database-generic `.gf` files for natural languages, as seen in section 3.2.3.2.

A lexer was implemented which allows the user to express queries in upper-, lower- or mixed-case letters, as well as use punctuation without concern for spacing.

### **Coverage of SQL**

For a graphical overview of the subset of SQL that is supported by the system, see figure 4.1.

As a guide to understanding figure 4.1: In this figure all SQL statements that are supported by the grammar can be represented by a route starting from the `{statement}` node and moving to the right through the graph by following one of the edges out of that node. This is done repeatedly until reaching a node without any edges to its right. Where lower-case words are present, these are meant to be replaced by a similar route through the sub graph for that word, e.g. `{predicate}` or `{query}`. The corresponding SQL statement can be assembled from this route by writing the text of each node on the route in sequence. The notation of square brackets (`[...]`) means an optional part and double quotes ("`...`") simply means follow the enclosed instruction instead.



**Figure 4.1:** A map of the supported subset of SQL.

For a complete list of the SQL commands which PostgreSQL allows, see the PostgreSQL documentation. [21]



# 5

## Discussion

Toward the stated goals of generality and adaptability, while the whole system isn't a universal translator, it is formulated to be general in a modular sense. This means that adding support for translation with an additional language or a new database does not involve a ground-up redesign, but rather just the addition of another module.

The design allows the majority of the system to remain the same when adding a new natural language. Only one file needs to be created per natural language (or two with the functor), English and Swedish were implemented as part of the project.

More languages can be added manually, the easiest way to do this for the non-functor grammar is by basing the new file on an existing concrete syntax for a natural language such as "DatabasesEng.gf" and then editing the parts that should differ. It is only expected that languages in the resource grammar library are added, in which case it should be a relatively straightforward process. If a language is not supported in RGL however the process would be much more difficult. In order to make the new file work properly and to allow for certain constructions in the new language, some files containing RGL resources for the new language need to be accessed. This includes the new language's RGL files for syntax (a file such as `SyntaxSwe.gf` in RGL), lexical paradigms (such as `ParadigmsSwe.gf`), use of symbolic notation (`SymbolicSwe.gf`) and some dictionary (such as `MorphoDictSwe.gf` or `DictSwe.gf`).

An alternative way of implementing database-generic natural language concrete syntaxes is with the GF feature known as functors. More is written about functors in sections 2.2.2 and 3.2.3.3. The benefit of this is reducing redundant code and copy-paste programming, to instead add instances to the interface of the functor. Having functors makes the grammar more general with respect to languages. This approach was partially implemented but not fully incorporated with the system. The functor implementation in its current state does not contain all of the variants which the main system has. More ideas for what can be done using functors can be found in further work, section 5.5.

Relating to database generality, the complicated and central `.gf` files are general with respect to which database is used. The grammar was divided so that as much as possible (everything which could be formulated to be the same no matter the database) is in the database-generic files to enhance generalizability and reduce un-

necessary work. This left only the necessary column and table linearizations in the database-specific .gf files which therefore became lightweight and easy to manage.

The parts of the system which do differ by database can be created through the user interface application, without doing any programming. See figure 3.6 for the interface used for that. As it was inevitable to require user input in order to create these files if a system was to be created which uses accurate names for all languages, and because their creation within the application is considered a part of the whole system, generality along this dimension is considered to have been achieved to a certain degree. However, since some natural languages require gender to form nouns while others do not, the language adding page (which currently does not support gender) will need to be updated to truly be general.

The system is easy to use once set up properly and doesn't require any programming skills. The application implemented as the interface of the system eliminates the need for using command line tools, manual compilation or even copy-pasting (e.g. from the translator to some database manager interface). This helps make databases more accessible to a wider audience such as people who are new to SQL along with making usage cleaner, and in some cases faster.

A lexer was implemented which allows users the freedom to choose how they capitalize letters and use spacing around punctuation. So they can say for instance "Show the capital, area and name of ..." instead of having to type in lower-case only and space out the comma as in: "show the capital , area and name of ...".

### 5.1 SQL

Between the different distributions of SQL (PostgreSQL, MySQL, SQLite etc.), there is some variation in functionality. Different distributions comply with different parts of the SQL standard.

For this project one of these distributions was used, since the distributions do not vary so wildly as to make it interesting to show multiple of them implemented. As even the PostgreSQL implementation of SQL is a language with a lot of available functionality, it was decided as the project progressed that it would be infeasible to implement support for that entire language in the grammar. [21]

The coverage of SQL achieved is illustrated in figure 4.1. The subset of SQL that was implemented contained different query types, functions, optional parts and more advanced features such as joins or subqueries. This strategy is meant to demonstrate that various different types of features of the SQL language can be integrated into the translation system without having to radically alter the shape of it. This fact shows that it is plausible to keep expanding the coverage to eventually support all of the legal commands of PostgreSQL.

By comparing the map of the achieved SQL coverage (in figure 4.1) and the map



of the abstract grammar (in figure 3.2), similarities can be observed due to the fact that the shape of the grammar was inspired by the structure of SQL. This is because of the fact that each query can only be expressed in one way in SQL, while it can potentially have multiple equivalents in each natural language. Another reason is that SQL is the more formal, constrained side to linearize to, while natural languages are often more open in possible formulations.

The fact that the grammar follows the structure of SQL queries can be shown in a few more specific examples: In the grammar a select query (QSelect) takes a Predicate as an argument (or an indication of no predicate) since the **SELECT** statement in SQL can have an optional **FROM** part containing a predicate. A predicate in SQL can consist of two predicates joined by the **AND** operation, correspondingly a Predicate in the grammar can be formed via the constructor **PredAnd** which takes two Predicates as arguments.

## 5.2 Variants

How naturally users can express themselves is a fairly subjective issue, but in this section, a picture is painted of the breadth of variety in natural language which is successfully understood by the system's lexer and parser.

Alternative ways of expressing the same query were implemented using variants in GF. These variants give the user various options in the way they wish to express a query in natural languages. See table 5.1 for some illustrative examples of these options implemented for English.

The variants were implemented for different linearizations, and many options were sometimes included to ensure the system allowed plenty of variation in formulations. In the English and Swedish syntaxes which were written, different variants can be combined in any way the types allow.

This also means that the system allows the user to express themselves in rather nonsensical ways. If we are looking in the database "america" for the people who have been both president and vice president we can say "the presidents whose names are the names of vice presidents" and the system will understand and give the correct result:

```
SELECT * FROM presidents
WHERE name IN ( SELECT name FROM vice_presidents ) ;
```

It would however get the same meaning from a strange sentence like "every president where names are the names for all vice presidents" due to the open implementation of variants. This is not considered a big problem, however, since that just means that the user provides a nonsense input and the system returns a sensible output.

Feature	Some options for inputs	Comment
Using different words	display all countries   show all countries   show me all countries   all countries	So the user doesn't have to remember one specific word
Optional parts	display all info about all countries   display all countries	All info is assumed if columns not specified
Short and long	countries   show all countries   show me all info about all countries	Rapid access or expressing it in the way you thought it
Singular and plural	show the countries where ...   show the country where ...	If the user knows or assumes a number
Slightly different formulations	the capitals of the 5 first countries   the capitals of 5 countries	If the user isn't concerned with which 5 for instance
	... where the currency is 'GBP'   ... whose currency is 'GBP'	More grammatical freedom for the user
Wh-questions	what is the mean area of a country   display the mean area of a country	To allow for questions as well as commands

**Table 5.1:** A demonstration of some alternative ways of expressing a statement available in the English concrete syntax, for a database which has the table "countries" with the columns "capital", "currency" and "area".

When translating from SQL to natural languages on the other hand default formulations have been picked for each constructor in English. This means that when translating to English, the start of each subquery looks the same in terms of grammatical form as the start of each whole query. For instance: "the name of the country where the currency is one of the code of the currency". This could potentially have been avoided by having a query linearize to two values, one for if the query is used as a subquery, and another if not. To do that would however have necessitated writing out a lot of combinations or restructuring the grammars with different types for each case in order to still ensure that the natural language input side is open to any way of expressing oneself. This exercise was not considered a priority for the scope of this project considering the little gain that would come from it.

The main goal with using variants was naturalness through allowing for a less constrained vocabulary on the natural language-side. That goal was reached fairly well considering that most of the ways to phrase a query in English and Swedish that I could come up with, given that I included the information the database needs, were supported. There was however no structured evaluation of the naturalness. For a discussion on better evaluating naturalness, see section 5.5. Some queries humans naturally come up with could not reasonably be understood by a computer without some AI solution, given the implications and assumptions inherent in many formulations.

## 5.3 Security

There are a few different security considerations for the application.

Since the user hosts their own server for the application, they can host it in an unsafe way. For instance a concern could be allowing outsiders access to the server they are running the application on.

The application cannot give the user access to content they are restricted from seeing in the database implementation. The reason being that the application simply makes requests for information from PostgreSQL using a username and password provided by the user. PostgreSQL is responsible for who can see what.

If the application returns an output from a translation which the user does not wish to execute, that is not a problem either. This is because the outputs from translations are never sent to the database immediately, the user always has to click on "send to DB" before that happens.

A user can enter unintended information in fields, leading to unexpected behavior in the application, because inputs aren't sanitized and are used in concatenation operations with other strings. This could be a problem on the database choice page (as seen in figure 3.5) where SQL code can be injected into the commands used to access the database. On the language adding page (as seen in figure 3.6) GF code could potentially be injected into the database specific files by entering half of a GF command in some of the boxes for column and table names. On the translation page (as seen in figure 3.7) the input field may be used to make the application run a different command than what is intended. The output field is not considered a security risk however, since its contents are simply sent to the database, so misusing it is just the same as misusing an input field in any database management software. In the current system it is assumed the user doesn't attempt to inject code. Avoiding unintended behavior because of injections is left to future work.

## 5.4 Limitations

One limitation of the system is that it assumes that languages which are added are supported by the Resource Grammar Library (at the time of writing that list includes 38 languages). If a language is not supported by RGL, such as is the case for Bengali or Indonesian, it is significantly more difficult to add.

Variants are written manually for each language at every point where they are desired in the grammar. This can be quite many variants if the user is to be given as natural an experience as possible, all of these can take some time to write.

When it comes to short and simple SQL queries, most of the common ways of expressing them in a natural language are supported by the grammar. For longer,

more complicated queries however, the way to express them in natural languages becomes less clear. For these queries, there is a lot of information which needs to be present in the statement. If many variants are implemented there is also the issue of the meaning becoming ambiguous as two SQL queries then both might be able to be written in the same way. This could be seen as reflecting the fact that natural language often is ambiguous and is only understood by other humans because of assumed and implied knowledge as well as common sense in the given context.

Some issues related to limitations, such as the fact that only PostgreSQL was supported or the lack of a structured evaluation of the system's naturalness, are discussed in section 5.5.

### 5.5 Further work

Further improvement of the system from its current state is possible in a number of directions. In this section, a number of future additions and changes to the system are suggested.

For example, there is the potential of expanding the grammar to cover more of the SQL language, this would involve editing the database-generic `.gf` files to reflect the new options. This could feasibly be done until all of SQL was supported.

Expanding to support database distributions other than PostgreSQL, such as MySQL or SQLite, could feasibly be done without changing the current form of the grammar. Since the different database distributions vary slightly in their feature support and keyword usage, this could be accomplished by having one concrete syntax for each distribution's particular implementation of SQL, rather than just having one concrete SQL syntax as is the case now. The application should in this case also be updated so it can access these other database distributions.

To more fully incorporate the functor with the whole system, all the variants of the non-functor `.gf` files would have to be implemented in the functor versions. Since the functor only deals with the database-generic part of the grammar, the application can still work mainly as-is after implementing a functor. With more work though, working with functors could perhaps be implemented in the application in order to allow the user to add languages database-generically with a GUI similarly to how they can database-specifically in the current application. This would mean that the system with ease could be used both with new databases and with new languages.

Different natural languages have different rules for how to form nouns. In some languages every noun has a gender while other languages do not use gender. This is not reflected in the application's language adding page, and thus supporting the way each language uses gender is something to be considered future work.

Also on the language adding page, the user could be guided in translating col-

umn and table names to the new language with the help of an external translation service. One idea for aiding the user in this process was to use GF's Wordnet, which contains records for equivalent words in different languages. A UI could let the user choose which translation is appropriate or allow them to manually enter something else instead. This was not considered central and due to the limited time of the project, it became future work instead.

Another possibility is to have alternative ways of expressing statements in natural languages which explain how the results of SQL queries are calculated. This could perhaps mean that an SQL query such as

```
SELECT name, home_state FROM presidents WHERE age < 50
ORDER BY age LIMIT 5 ;
```

would be translated to the English explanation

```
Take the list of all presidents.
Keep only those with an age under 50.
Take their names and home states.
Sort the entries by age.
Return only the first 5 entries.
```

The purpose such a feature would have is mainly pedagogical, to allow users to have an SQL query explained to them in the order that the different parts are used in the process of producing the result.

One avenue of further work is avoiding unintended behavior of the application resulting from code injections in various input fields, as discussed in section 5.3.

The application in its current state is hosted locally by each user, but it could potentially be hosted on a server connected to the internet so that users simply access it on the world wide web. This would require a redesign of where `.gf` and `.pgf` files are stored and how the database is accessed by the application.

This project did not involve a structured evaluation of how natural the system ended up being. This would be a useful addition to see how well the stated goal of naturalness was reached. One suggestion for how to carry out such an evaluation is the following rough process:

- A number of users are given no instructions and are presented with the application.
- They enter inputs into the translation page unguided at first, and those inputs are collected and analyzed, and the percentage of inputs which the system understands is calculated.
- Then each user is given some prepared basic instructions on how the system understands them and what information they need to provide in order to be understood, after which another round of inputs are entered, collected and analyzed.
- More and more detailed explanations of how to communicate with the system could be provided with the same analysis cycle each time to see how much guidance each user needs to successfully use the system.

The results from this process could also be broken down by categorizing the users by familiarity with databases or SQL, by whether or not they are knowledgeable in programming, by whether or not they commonly use computers etc. The system could then be adapted based on the collected results with the pursuit of trying to make the system understand more of the inputs users entered, or at least something similar to what they entered.

Such a process could additionally include users giving feedback on what they found difficult to understand about the system, feedback which then could be used to improve the system to fix the problems. Additionally based on the feedback, a guide to using the system could be improved to better explain what the user can do to be understood by the system. These feedback gathering sessions could be useful regularly to evaluate each new version of the system.

# 6

## Summary

A grammar was created, using the Grammatical Framework programming language (GF), which can translate between SQL and natural languages. This grammar has generality as its central design aim, this mainly takes the form of the central parts being database-agnostic and language-agnostic. It supports a varied, demonstrative subset of SQL, but is expandable to cover more in the future. This grammar is implemented in English and Swedish, but it is also expandable to more natural languages. The grammar was divided between database-generic and database-specific which helps give it a general shape with respect to databases.

This grammar was implemented at the core of an application. A central aim with this application was to make the system more accessible. It allows the user to translate using the GF grammar, but also has the capability to send queries to a database and display the results. The database-specific part of the grammar can be created from within the application in a GUI setting without requiring programming skills.

The focus remained on creating a general system throughout the project's course and instances of designing the system for a specific situation occurred only temporarily and were successfully removed at later stages.





# 7

## Bibliographic notes

A summary of sources and their relevance to the project, sorted by topic.

### Grammatical Framework

- [9] - The GF homepage, contains much information on the GF system and a useful tutorial
- [31] - The book on the GF system, with many useful tips
- [13] - How to download and install GF
- [28] - A tutorial to the GF language, with good examples
- [30] - The GF tutorial part for PGF, including the haskell API
- [29] - The GF tutorial part for Functors
- [12] - The GF core code
- [2] - GF contrib code, various little files used
- [11] - A useful synopsis of the Resource Grammar Library
- [14] - The code for RGL (The Resource grammar library)

### Databases

- [26] - Some information about databases
- [22] - The PostgreSQL homepage
- [27] - The pgAdmin homepage, used for administering and accessing databases
- [21] - Documentation of the PostgreSQL query language
- [7] - A ranking of different database models
- [6] - A ranking of different database management systems

### Haskell packages

- [10] - The GF Haskell package, needed to generate .pgf files
- [15] - The network Haskell package, needed for the GF package
- [1] - The Network.CGI Haskell package, used to formulate the user application functionality
- [19] - The Text.XHtml Haskell library, used to form the user application pages
- [17] - The System.Directory Haskell package, used to scan directories for available .gf files
- [5] - The Data.Text.IO.Utf8 Haskell package, used to write .gf files
- [18] - The System.Process Haskell package, used to run commands to generate .pgf files
- [16] - The API documentation of the PGF Haskell package, used to access .pgf files to parse and linearize

- [3] - The Database.HDBC Haskell package, used to exchange information with the PostgreSQL server
- [4] - The Database.HDBC.PostgreSQL Haskell package, used to connect to the PostgreSQL server

### Other

- [25] - Microsoft's IIS (Internet Information Services), one option for hosting a local web server for the application
- [8] - How to install the Haskell Platform, used to make the application
- [24] - Blog post about usage of GF grammars from external applications
- [23] - Code used in a tutorial on usage of GF grammars from external applications
- [33] - The Lunar system, a natural language question answering system concerning lunar geography
- [32] - Chat-80, an easily adaptable system for answering queries in natural language
- [20] - Previous work with GF to make a system translating from natural language to SPARQL

# Bibliography

- [1] cgi: A library for writing cgi programs. URL: <https://hackage.haskell.org/package/cgi>.
- [2] Community contributions to the grammatical framework. URL: <https://github.com/GrammaticalFramework/gf-contrib>.
- [3] Database.hdbc haskell package - documentation. URL: <https://hackage.haskell.org/package/HDBC-2.4.0.3/docs/Database-HDBC.html>.
- [4] Database.hdbc.postgresql haskell package - documentation. URL: <https://hackage.haskell.org/package/HDBC-postgresql-2.5.0.0/docs/Database-HDBC-PostgreSQL.html>.
- [5] Data.text.io.utf8 haskell package - documentation. URL: <https://hackage.haskell.org/package/with-utf8-1.0.2.2/docs/Data-Text-IO-Utf8.html>.
- [6] Db-engines ranking - popularity ranking of relational dbms. URL: <https://db-engines.com/en/ranking/relational+dbms>.
- [7] Db-engines ranking per database model category. URL: [https://db-engines.com/en/ranking\\_categories](https://db-engines.com/en/ranking_categories).
- [8] Download haskell platform. URL: <https://www.haskell.org/platform/>.
- [9] Gf - grammatical framework. URL: <https://www.grammaticalframework.org/>.
- [10] gf: Grammatical framework. URL: <https://hackage.haskell.org/package/gf>.
- [11] Gf resource grammar library: Synopsis. URL: <http://www.grammaticalframework.org/lib/doc/synopsis/index.html>.
- [12] Grammatical framework core: compiler, shell & runtimes. URL: <https://github.com/GrammaticalFramework/gf-core>.
- [13] Grammatical framework download and installation. URL: <https://www.grammaticalframework.org/download/index-3.10.html>.
- [14] Grammatical framework's resource grammar library (rgl). URL: <https://github.com/GrammaticalFramework/gf-rgl>.
- [15] network: Low-level networking interface. URL: <https://hackage.haskell.org/package/network>.
- [16] Portable grammar format - haskell api. URL: <https://hackage.haskell.org/package/gf-3.10/docs/PGF.html>.
- [17] System.directory haskell package - documentation. URL: <https://hackage.haskell.org/package/directory-1.3.6.2/docs/System-Directory.html>.
- [18] System.process haskell package - documentation. URL: <https://hackage.haskell.org/package/process-1.6.11.0/docs/System-Process.html>.

- [19] xhtml: An xhtml combinator library. URL: <https://hackage.haskell.org/package/xhtmll>.
- [20] Mariana Damova, Dana Dannells, Ramona Enache, Maria Mateva, and Aarne Ranta. Natural language interaction with semantic web knowledge bases and linked open data. *Towards the Multilingual Semantic Web*, pages 211–226, 2013.
- [21] The PostgreSQL Global Development Group. Postgresql documentation - the sql language. URL: <https://www.postgresql.org/docs/current/sql.html>.
- [22] The PostgreSQL Global Development Group. Postgresql: The world’s most advanced open source relational database. URL: <https://www.postgresql.org/>.
- [23] Inari Listenmaa. inariksit/gf-embedded-grammars-tutorial. URL: <https://github.com/inariksit/gf-embedded-grammars-tutorial>.
- [24] Inari Listenmaa. Using gf grammars from an external program, December 2019. URL: <https://inariksit.github.io/gf/2019/12/12/embedding-grammars.html>.
- [25] Microsoft. The official microsoft iis site. URL: <https://www.iis.net/>.
- [26] Oracle. What is a database? URL: <https://www.oracle.com/database/what-is-database/>.
- [27] The pgAdmin Development Team. pgadmin - postgresql tools. URL: <https://www.pgadmin.org/>.
- [28] Aarne Ranta. Grammatical framework tutorial, December 2010. URL: <http://www.grammaticalframework.org/doc/tutorial/gf-tutorial.html>.
- [29] Aarne Ranta. Grammatical framework tutorial - functors: functions on the module level, December 2010. URL: <http://www.grammaticalframework.org/doc/tutorial/gf-tutorial.html#toc92>.
- [30] Aarne Ranta. Grammatical framework tutorial - the portable grammar format, December 2010. URL: <http://www.grammaticalframework.org/doc/tutorial/gf-tutorial.html#toc142>.
- [31] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011.
- [32] David H. D. Warren and Fernando C. N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *Comput. Linguist.*, 8(3–4):110–122, July 1982.
- [33] William Woods, Ronald Kaplan, and Bonnie Webber. The lunar science natural language information system: Final report. 01 1972.

# A

## User guide to installation

This guide is directed at a user of the application which was created during the course of the project.

In order to install the application, follow these steps:

### A.1 Grammatical Framework

Go to the Grammatical framework page for downloading and installing GF and follow the instructions in order to get the GF system installed. [13]

### A.2 Getting the necessary code

Find the repository of the project which produced this application on github, known as Natural-SQL-Translator: <https://github.com/Dan-dav/Natural-SQL-Translator>. Please note that the code in this repository is subject to change. Clone that repository to some desired location on your machine.

You also need to have the Haskell Platform installed in order to make things work, see the relevant download page for instructions on how to get it onto your machine. [8]

Make sure all of the needed Haskell packages are installed for all of the involved code to run properly:

- The Haskell package for gf [10]
- The network package [15]
- Network.CGI [1]
- Text.XHtml [19]
- The Database.HDBC package [3]
- The Database.HDBC.PostgreSQL package [4]

In order to install the gf package in particular the following guide by Inari Listenmaa may be helpful to adapt to your situation:

<https://inariksit.github.io/gf/2019/12/12/embedding-grammars.html> [24], with the relevant code available at:

<https://github.com/inariksit/gf-embedded-grammars-tutorial> [23]

In order to give the `.gf` files access to the Resource Grammar Library (RGL) files they need, clone the RGL repository to a folder in the same directory as the one you cloned this project's repository to. [14] Also clone `gf-core` and `gf-contrib` to the same location, so all four cloned repositories are in the same directory. [12, 2]

### A.3 Configuring a local server

Set up a server hosted on your machine. On Windows 10 for instance there is the option of using the preinstalled IIS (Internet Information Services) manager to do this. [25] Create a virtual directory linking to the folder `ext-app` in the cloned repository of this project.

The URL to access in a web browser should now be at `localhost/.../ext-app` depending on where you put the virtual directory within the file structure of the web server.

You may have to enable permissions for the server to be able to access the `ext-app` directory. You may also have to enable some option for the execution of `.cgi` files on your server as otherwise the server will not know what to do with files of this type.

# B

## User guide to the application

This guide explains how to run the application once installed by following the guide in Appendix A. The sections of this appendix contain useful tips for first time users.

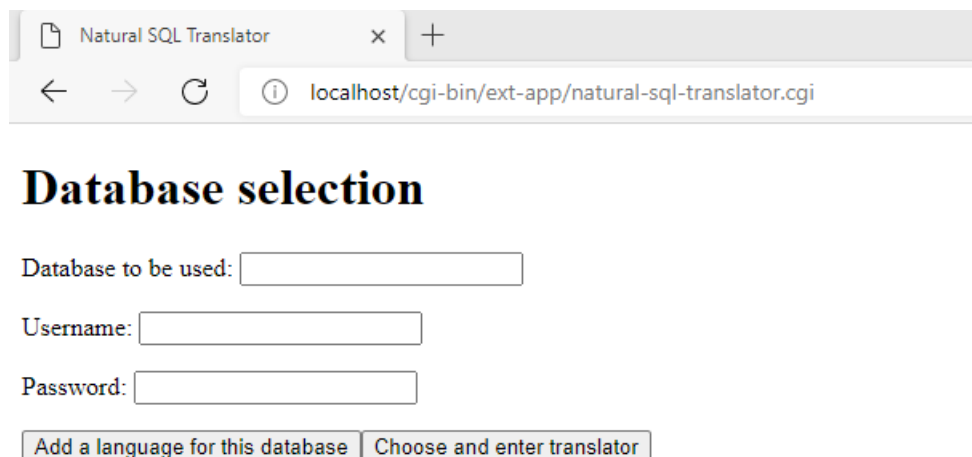
### B.1 Accessing the server

In order to access the application a web browser is used, but first the application needs to be hosted locally by a web server. Assuming the instructions in section A.3 were followed, the server should just need to be started up using whatever hosting software it was configured it with, for instance if IIS (Internet Information Services) was used to configure the server, it can be started using the IIS manager. [25]

When the server is running, navigate to the URL of the application within the web server by using your web browser. This URL would commonly be

`localhost/.../ext-app/natural-sql-translator.cgi`

with the exact path depending on the server's directory structure. Navigating to that URL should result in being presented with a page somewhat like the one in figure B.1.



Natural SQL Translator

localhost/cgi-bin/ext-app/natural-sql-translator.cgi

### Database selection

Database to be used:

Username:

Password:

**Figure B.1:** The database selection page of the application

## B.2 Reaching a Database

The database needs to be hosted locally in order to be able to use certain features of the application. These features include adding support for a new database and sending SQL queries to the database manager with the click of a button. The database can be set up with the pgAdmin software. [27]

Once the application has been reached, fill in the presented fields on the database selection page (as seen in figure B.1). This includes typing the name of the database, along with a username and password to access it. The username and password are the same as those used by PostgreSQL for different users. Next, click on "Add a language for this database". It should be apparent that the application has made contact with the correct database based on the tables and columns listed on the language adding page, as seen in figure B.2.

## B.3 Adding a language for a database

If working with a new database (one that hasn't been used with the application before) then enter the relevant information on the first page (figure B.1) and click "Add a language for this database". This makes the language adding page (figure B.2) appear, so a first language can be added for that database. Having gone through the language adding process at least once is a prerequisite to performing translations with a given database.

If working with a database that has been used before, but the addition of support for another language is desired, also click "Add a language for this database".

The page which appears should look similar to the one shown in figure B.2.

On the language adding page (figure B.2), choose one of the available languages which you wish to add support for with this database. This page contains fields for filling out the names used to refer to each of the tables and columns found in the database, in both singular and plural forms. These fields instruct the application in how the different database entities will be referred to during translation, but only when using the selected database in the selected language.

There is a "Clear all" button for resetting the text fields and starting over. The button "Choose another database" discards the entered data and goes to the database selection page (figure B.1) again in case another database is wanted instead. When the entered data is satisfactory click on "Submit and enter translator", this makes the translation page appear (figure B.3).



Natural SQL Translator

localhost/cgi-bin/ext-app/natural-sql-translator.cgi?page\_id=database&db\_choice=countries&username=

## Language adding

Language to add for the countries database: Eng ▼

Please type names in that language (in singular and plural) for the tables and columns of the countries database:

Tables:

	Singular	Plural
countries	country	countries
currencies	currency	currencies

Columns:

	Singular	Plural
area	area	areas
capital	capital	capitals
code	code	codes
continent	continent	continents
currency	currency	currencies
name	name	names
population	population	populations

Clear all Choose another database

Submit and enter translator

**Figure B.2:** The language adding page of the application

## B.4 Performing translations

To enter the translation page for a given database simply access the application through a web browser, enter the relevant information into the database selection screen (figure B.1), and click on "Choose and enter translator". The translation page can be seen in figure B.3. If a language was just added for a database via the language adding page, the translation page will also have appeared.

On the translation page (figure B.3), options are available to choose between the natural languages currently supported for the chosen database as well as SQL, for both input and output. Additionally, more natural languages can be added for the current database by clicking on "Add a language for this database", this makes the language adding page (figure B.2) appear. To go to the database selection page (figure B.1) and switch to another database, click on "Choose another database".

The screenshot shows a web browser window with the title "Natural SQL Translator". The address bar shows the URL: `localhost/cgi-bin/ext-app/natural-sql-translator.cgi?page_id=database&db_choice=countries`. The page has a heading "Translation". Below the heading, there is a section "Using the database: countries" with two buttons: "Choose another database" and "Add a language for this database". Below this, there are two dropdown menus: "From: Eng" and "To: SQL". Below the dropdowns is an "Input:" label followed by a large text input field. Below the input field is a "Translate" button. Below the button is an "Output:" label followed by a large text output field. Below the output field, there is a section "Send SQL code to countries database:" with a "Send to DB" button and a checkbox labeled "Also commit changes:". Below this section is a box labeled "Output from the database" which contains the text "Nothing".

**Figure B.3:** The translation page of the application

The main purpose of the application is on this page: Translation. In order to translate from a natural language to SQL or the other way around, first choose the languages to translate from and to in the corresponding drop down lists. Next, enter a query in the input field and click on "Translate". After clicking that button an output should appear in the output field, in the chosen output language, unless some error was encountered while performing the translation, in which case an error message should appear.

SQL strings in the output field can additionally be sent directly to a database manager to actually execute the code on a locally hosted database. If the button "Send to DB" is clicked, the application will send the contents of the output field to- and present the result from your database manager at the bottom of the screen, or an error if the database manager couldn't understand the query.

# C

## Code from Databases.gf

In this appendix, the code from the database-generic abstract syntax file "Databases.gf" is presented. Included are some comments which explain what certain constructors do in the grammar.

```
abstract Databases = {

  flags startcat = Statement ;

  cat
    Statement ;
    Query ;
    ColumnPart ;
    FromLimPart ;
    JoinType ;
    TabCol ;
    Column ;
    [Column] {2} ;
    PredicatePart ;
    Predicate ;
    Value ;
    [Value] {2} ;
    CompOp ;
    LikeOp ;
    OrderPart ;
    SortBy ;
    [SortBy] {2} ;
    Order ;
    InsertPart ;
    InsertCol ;
    [InsertCol] {2} ;
    UpdatePart ;
    UpdateCol ;
    [UpdateCol] {2} ;
    Table ;

  fun
    -- Query statements, UNION, INTERSECT ----
```

```
-- A complete statement can be formed by a select (or other) query
StQuery : Query -> Statement ;

-- The union or intersection of two queries also functions as a query
QUnion : Query -> Query -> Query ;
QUnionAll : Query -> Query -> Query ;
QIntersect : Query -> Query -> Query ;

-- SELECT -----

-- The parts of a select query, some of which are optional
-- A query like this can also be used as a subquery in parts
QSelect : ColumnPart -> FromLimPart -> PredicatePart
  -> OrderPart -> Query ;

-- Selecting all columns, or one or more specific columns
SColumnAll : ColumnPart ;
SColumnOne : Column -> ColumnPart ;
SColumnMultiple : [Column] -> ColumnPart ;
-- Representing the count operation in SQL,
-- counting numbers of entries
SColumnCount : ColumnPart ;
SColumnCountCol : Column -> ColumnPart ;
SColumnCountDistinct : Column -> ColumnPart ;
-- Representing the avg and sum operations in SQL,
-- giving the average or sum of values in a column
SColumnAvg : Column -> ColumnPart ;
SColumnSum : Column -> ColumnPart ;

----- FROM, LIMIT

-- represents both what table or join to take data from,
-- and an optional limit
FromTab : Table -> FromLimPart ;
FromTabLim : Table -> Int -> FromLimPart ;
FromJoin : JoinType -> TabCol -> TabCol -> FromLimPart ;
FromJoinLim : JoinType -> TabCol -> TabCol -> Int -> FromLimPart ;

----- JOIN

JInner, JLeft, JRight, JFull : JoinType ;

-- Allows a table-column pair to be used as a column, for joins
JTabCol : Table -> Column -> TabCol ;
```

```
ColumnTabCol : TabCol -> Column ;

----- WHERE

-- An optional predicate
PredNothing : PredicatePart ;
PredSomething : Predicate -> PredicatePart ;

-- Some different predicates which can be formed in SQL
PredAnd : Predicate -> Predicate -> Predicate ;
PredOr : Predicate -> Predicate -> Predicate ;
PredComp : Column -> CompOp -> Value -> Predicate ;
PredIn : Column -> [Value] -> Predicate ;
PredBetween : Column -> Value -> Value -> Predicate ;
PredLike : Column -> LikeOp -> String -> Predicate ;
PredIsNull : Column -> Predicate ;
PredIsNotNull : Column -> Predicate ;
PredSubQuery : Column -> Query -> Predicate ;

-- A value can either be an integer or a string
ValInt : Int -> Value ;
ValStr : String -> Value ;

-- Representing the operations: =, >, <, >=, <=, <>
CompOpEq, CompOpGt, CompOpLt, CompOpGEq, CompOpLEq
  , CompOpNE : CompOp ;

-- Representing something beginning with, ending with
-- or containing another string
LikeBegins, LikeEnds, LikeContains : LikeOp ;

----- ORDER BY

-- Option to order the results by one or more columns
OrdNothing : OrderPart ;
OrdOne : SortBy -> OrderPart ;
OrdMultiple : [SortBy] -> OrderPart ;

-- A column and how to order by it
SortColumn : Column -> Order -> SortBy ;

-- Ordering by ascending, descending or the default
OrdUnspec, OrdAsc, OrdDesc : Order ;

-- DELETE -----
```

```
StDelete : Table -> PredicatePart -> Statement ;

-- INSERT -----

StInsert : Table -> InsertPart -> Statement ;

-- Inserting specified value(s) at specified column(s)
IColValOne : InsertCol -> InsertPart ;
IColValMultiple : [InsertCol] -> InsertPart ;
-- Inserting specified value(s) at the first column(s)
IOnlyValOne : Value -> InsertPart ;
IOnlyValMultiple : [Value] -> InsertPart ;
-- Inserting value(s) from a subquery,
-- optionally at specified column(s)
ISubQuery : Query -> InsertPart ;
ISubQueryColOne : Query -> Column -> InsertPart ;
ISubQueryColMultiple : Query -> [Column] -> InsertPart ;

-- One specific value to be added to one specific column
InsertColWith : Column -> Value -> InsertCol ;

-- UPDATE -----

StUpdate : Table -> UpdatePart -> PredicatePart -> Statement ;

-- One or more columns with their new value(s)
UpdateOne : UpdateCol -> UpdatePart ;
UpdateMultiple : [UpdateCol] -> UpdatePart ;
-- add support for x = x + 1 etc.

-- One specific value to be set at one specific column
UpdateColWith : Column -> Value -> UpdateCol ;
}
```