A natural language interface for a music database

Master of Science Thesis in Computer Science: Algorithms, Languages and Logic

JORGE DIZ PICO

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Department of Computer Science and Engineering
Division of Computer Science
Göteborg, Sweden 2011
Master of Science Thesis 2011-06

A natural language interface for a music database
JORGE. DIZ PICO.

Examiner: BENGT. NORDSTRÖM.

Master of Science Thesis 2011-06
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Department of Computer Science and Engineering
Division of Computer Science
SE-412 96 Göteborg
Sweden
Telephone: + 46 (0)31-772 1000

MASTER OF SCIENCE THESIS 2011-06

A natural language interface for a music database

Master of Science Thesis in Computer Science: Algorithms, Languages and Logic

JORGE DIZ PICO

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Department of Computer Science and Engineering
Division of Computer Science

Göteborg, Sweden 2011

Abstract

We present an interface for a music database in which the user asks questions and the system replies with the exact information desired. Natural language parsing with the Grammatical Framework was used for input and output processing as to offer multilingual capabilities. The code architecture, from interface to database connection and results treatment, was built with Haskell. Arrows and an increased degree of function modularity were employed, looking for flexibility for future expansions.

Keywords: Grammatical Framework, Haskell, XML, Arrows, natural languages, interface, grammars, functional programming

Project accesible at: `http://diz.es/tese/`

Contact:

- Student: Jorge Diz Pico, author of the present Master's Thesis.
  jorge@diz.es
- Academic supervisor: Bengt Nordström, professor at the Chalmers University of Technology
  bengt@chalmers.se

Contents

Acknowledgements

This project has been carried out under supervision of Bengt Nordström. Without his wise guidance and encouragement, this wouldn't have been possible. His out-of-the-box thinking inspired and got me out of many problems. He was also very friendly and understanding, and made working with him a quite enjoyable experience.

I have to wholeheartedly thank first and foremost my family for always being there for me. Their unconditional support and faith in me and my work helped me keep pushing when stuck, and face every challenge, personal or academic, with smile and confidence. Three thousand kilometers sounds like a long way, but they managed to make me feel as if I had never left home.

I also want to dedicate the project to all the friends I made in my two years of stay in Göteborg. I was lucky to have met what I'm sure is the best people out of every corner of the Earth, gathered by chance in this city, and befriend them. Together we have shared many experiences, adventures, laughs and tears. Some of you have really earned a spot in my heart, and I hope our relationship will last for long, for I have nothing but good memories out of my times with you in this period of my life. You are too many to list, but you know who you are.

The final mention goes for those people who I selfishly left back home, but could not get rid of. Though I thought I would be safe in the distance, it didn't stop them from offering me support when I needed it, pestering me with our old jokes, incomodating me with their visits and asking continously for my return. I have to explicitly mention here Juan and Mario. I couldn't wish for better friends.

Also, Fer, because he doesn't fit in any of the previous two categories.

That was all. So long and thanks for all the fish, Sweden.

Göteborg, June 2011
Jorge Diz Pico

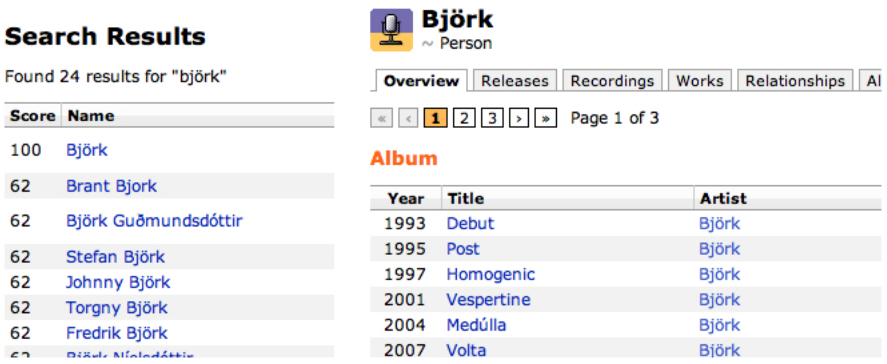
1 Introduction

1.1 The problem

In the last decades, we have seen a rise on the quantity of information offered to the public. Increasingly cheaper storage space has helped to spurt a trend of making huge amounts of collected data freely available in the internet. Many communities have been formed just to tag, describe and classify the world around us. But the amount of information has turned so huge, that a new problem arises: how to make this information easy to sift through for everyone? How do we help non tech-savvy users find the needle in the proverbial haystack?

In the specific case that we'll tackle, there is MusicBrainz, a community driven project backed up by the MetaBrainz Foundation, created in 2005. Their aim is to build a database listing metadata about music: all there is to know about artists, releases and tracks, among others. This database can be downloaded to access in any way the user pleases, but since this requires specific technical skills, it is far from being useful to the average person.

For the general public, MusicBrainz offers a web interface to search those records through name or title, and then in the individual page for each, find links to related content. For example, one could look for the band Metallica through its name, then once in its page, find connections to their members and released albums.



Search Results

Found 24 results for "björk"

Score	Name
100	Björk
62	Brant Bjork
62	Björk Guðmundsdóttir
62	Stefan Björk
62	Johnny Björk
62	Torgny Björk
62	Fredrik Björk

Björk
~ Person

Album

Year	Title	Artist
1993	Debut	Björk
1995	Post	Björk
1997	Homogenic	Björk
2001	Vespertine	Björk
2004	Medúlla	Björk
2007	Volta	Björk

Figure 1.1: On the left, search result for the artist name Björk. On the right, individual page for the first artist returned for this search (both screens cropped).

Although easy to understand, this listings become cumbersome quickly. It takes too many clicks to get a simple piece of data like the number of tracks in an album. It gets even worse when faced with more complicated questions, like finding out who is the youngest member of a given band, since we'd have to locate the band, navigate to the current members, pick out the birth dates, and then compare them. The time it takes is probably beyond the threshold of what anybody would call readily available information, and therefore many people would not consider that functionality as in the array of possibilities offered.

We want an intuitive method for common people to be able to query this database and get exactly the piece of information they want, without being forced to look for it themselves in several pages of results, having to worry about learning how the database is organized, or any other technical details.

1.2 The solution

People have an inner sense of how things should work. They draw parallels from what they already know to their expectations.

Right now, the gateway for information for the majority of people is the internet. When they want to know something, they go to the internet to look for it. This gives us our first clue: our interface should be similar to a search engine. It's quite a common sight that people know how to use.

Our second aspect is to find out what people expect from a search engine and give them that. The best intuition comes from observing first timers behave in front of a computer. Many of us in the computer field have learned to adapt around the quirks of technology, so we have a twisted sense of our expectations. But we can take advantage of the results that big search engines like Google have already gathered about novices.

According to Marissa Mayer [5], expert users write:

```
hike seattle area
```

While novice users write:

```
Where can I hike in the Seattle area?
```

That is, most of the new users go for fully expressed questions instead of using just the keyword terms like veterans would.

Now, in the particular case of Google, their search engine manages an overwhelming array of records. Therefore, they find in their favor to focus on isolated key terms, ignoring the syntactic relations between words marked by prepositions or articles. It's a scaling problem. But there is a lot of information being lost in that discarding: we could know that the user doesn't want to know where to buy hiking boots, or the different hiking associations in the area; just where can he go practice it.

Since we are working with a limited domain in which the answers are already categorized and labeled, we have much to gain from every extra detail our users give. By *understanding* the semantics, we can offer a concrete and exact answer. New search engines like Hakia [3] have made progress in taking advantage of the extra meaning offered by making a semantic analysis of the syntax when working in specific domains (health, aeronautics). Even projects like IBM Watson [2] that mostly focus on keywords, show that the future of the field passes through accepting input in natural language instead of forcing the user to make the culling.

Furthermore, from the way the questions are phrased, we can deduce that many people expect the computer to almost reply like a real person would. That means that sometimes they would prefer a one line explanation like *"Hiking is only allowed in the northwestern forest areas"* in lieu of links to a lot of parks and recreation laws to sift through.

In summary, we will build an interface similar to that of a search engine, that will accept natural language questions from the user and also come back to them with answers in a natural language with the exact information they are looking for.

2 Overview

2.1 The application

The system is comprised of several modules. We aimed for encapsulation of functionality so that if any part needed to be replaced (say, we swapped MusicBrainz's database for another one with different structure), as little as possible in non-related modules had to be corrected.

Three different layers can be delimited, corresponding to the three different steps the information goes through when traversing between the user and the database. In section 3, we will take a look at how these layers are divided into modules and files and how they interact. But for the moment, this abstraction will help us get a global idea of the system.

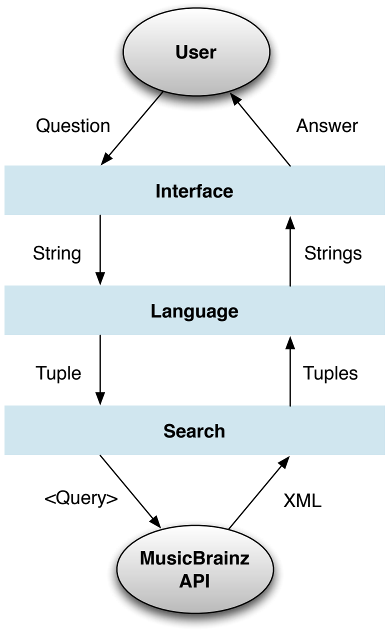


Figure 2.1: Overview of the layer structure

First, the User writes the question in the text box. The sentence could look like this:

```
Where was Beck born?
```

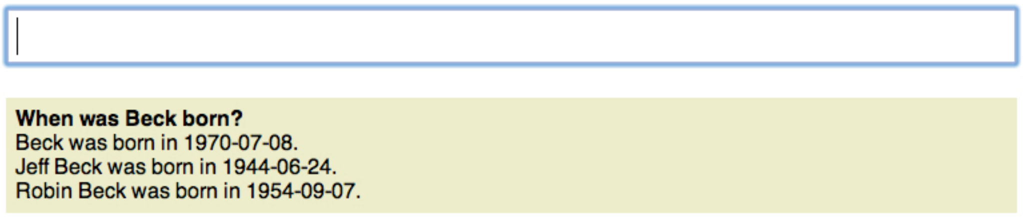
The Interface is but a conduit, so it doesn't modify the input at all. It is carried without modifications to to the next layer, called Language. In it, the question is analyzed syntactically. If it is detected as grammatically correct, the sentence is mined for the key elements, which depend on what type of sentence it was detected as. For the one showed before, it is detected as a birthdate question, so it is enough to extract the name *Beck*.

Now, this tuple of data is passed to the corresponding entry point on the Search layer. Search has lots of functions individualized depending on the kind of the information you are after. Here, we would forward this packet of data to that in charge of finding out when an artist was born. Since the function's only behaviour is to look for birthdates, it only needs the name of the artist to investigate upon.

The Search layer builds a REST call and calls the API at MusicBrainz, who replies with an XML containing plenty of related information (see section 2.2 for an explanation on REST). Still inside the Search layer, we strip the XML of all superflous data, extracting exactly what we want: the date of birth. Some ambiguity is expected, so a couple of possible solutions are returned.

The new tuples containing the name of the artist and her birthday are returned to the Language layer. Keeping in mind the language in which the question was originally posed, the different tokens are placed inside an answer template, generating a meaningful sentence.

Finally, this sentence is returned to the Interface and displayed for the user.



When was Beck born?
Beck was born in 1970-07-08.
Jeff Beck was born in 1944-06-24.
Robin Beck was born in 1954-09-07.

Figure 2.2: Example of question answered by the system with three possibilities.

2.2 The database

2.2.1 How is it organized

MusicBrainz's database currently comprises four elements: *artists*, *releases*, *tracks*, *labels* and the many relationships between them. In its most basic aspect, *Artists* make *Releases*, that contain *Tracks*, under a certain *Label*. This network of relationships is what makes MusicBrainz's more powerful than the other databases available, since it allows us to extract information that goes beyond the raw data. For example, *Artists* of the type **Group** have timed relations to other *Artists* of type **Person**, and therefore we can know what members belonged to what band and when.

2.2.2 How is it accessed

REST (*Representational State Transfer*) is a software architecture that describes communication between client-server systems. The basic principle behind it is that resources are unique and identified by an URI, and elements in the system exchange representations of this resources by requests. Also, servers are stateless, and all information necessary to act and respond to a request is contained in it.

The API exposed by MusicBrainz follows the REST standard, so each element has a unique ID that can be accessed by querying its corresponding URL. For example, to retrieve information about the band Placebo, with ID:

```
847e8284-8582-4b0e-9c26-b042a4f49e57
```

we would use this address:

```
http://musicbrainz.org/ws/1/artist/847e8284-8582-4b0e-9c26-b042a4f49e57/?type=xml
```

We would just get basic information, though. It is through the **inc** parameter that we can reach further. Two of the most remarkable values it can receive are *release-events*, for release events of the albums, and *artist-rels*, for members of a group and other elements related to an artist such as its website.

When the MusicBrainz identifier (commonly called **MBID**) is not known, the API allows us to perform searches that match a given token. We can, for example, look for releases with a certain title or performed by an artist with a name similar to the one we provide.

Futher information about this web service API can be found in the MusicBrainz documentation [1].

3 Architecture

In the previous section (2.1), we saw how the system is structured in three different layers.

First the Interface, that receives the input from the user and delivers it to the system, and likewise gets the answers from the system and delivers them to the user.

Second comes the Language, that on one side extracts the relevant tokens from the questions and identifies the semantics of it; and on the other wraps the data from the database in a meaningful natural language sentence.

Third and finally, the Search layer constructs the queries to the MusicBrainz API from the data extracted in Language, and hands back the tuples of relevant information it finds.

We will later explore in more detail the files that conform each of these layers. But there is one file that is not present in any particular explanation, since its refered by several of them. In the system, a token name, like *Metallica* or *Tubular Bells*, that is represented by a String, could be anything from a proper Artist name to an Album title. In functions that manage tuples with several elements of this type, it could become messy to infer what element represents what.

That's why *Types.hs* was created, defining the dummy types *Token* (global), *Artist*, *Track*, *Album*, *MBID* (MusicBrainz's unique identifier) and *Date*. They are all equivalent to String, but this masking helps make function's signatures understandable. Compare the possible types of the following arrow:

```
a_treleasebySingerSong :: IOSArrow (String, String) (String, String, String, String)
a_treleasebySingerSong :: IOSArrow (Track, Artist) (Track, Artist, Date, Album)
```

3.1 General notes

The code in this project consists of several Haskell files. The language was chosen primarily because of the good API that the Grammatical Framework offers for this language (see section 3.3 for details on this interaction or section 4 for the grammar building).

Nevertheless, functional programming was a perfect fit for the kind of work this project required, since it relies heavily upon reiteration and patterns. Most of the work done in the system is done by matching (e.g., the constructs of the trees to discern the question being asked) and applying repetitively the same action over elements of a list (e.g., linearizing the possible answer trees to generate sentences). Both of this cases apply as well to the XML processing that was required in parsing MusicBrainz's API response.

The choice for individual casing

The first thing done in this project was to brainstorm possible questions the user would like to have answered. With this list, the first thought is how many of the questions are quite similar, for example, *"when was the album Volta released?"* and *"when was the single Bachelorette released?"*. Since one of the good practices of software building is to reuse code as much as possible, our first instinct would be to approach both of these cases with the same function, marking the differences by use of parameters. And certainly it would work, since both require to access a **Release** object (one of type *Album*, another of type *Single*) and find its release date.

But then we come accross another sentence, *"when was the song Hyperballad relased?"*, and it seems innocent enough to group it with the other two. But in this case, the underlying database representation makes no sense of tracks on its own. They have to be accessed through their corresponding **Release** to get a release date. Therefore, the requests to the API and the posterior XML parsing are completely unrelated with the others.

We face a conundrum them: if we group all of them together, their behaviour would be more akin to two different functions joined by a *case-of* statement, defeating the purpose of grouping in the first place. But if we leave songs apart, it defies expectations to be grouping sentences by their looks, and then having a function for albums/singles and another for songs when they are phrased the same way.

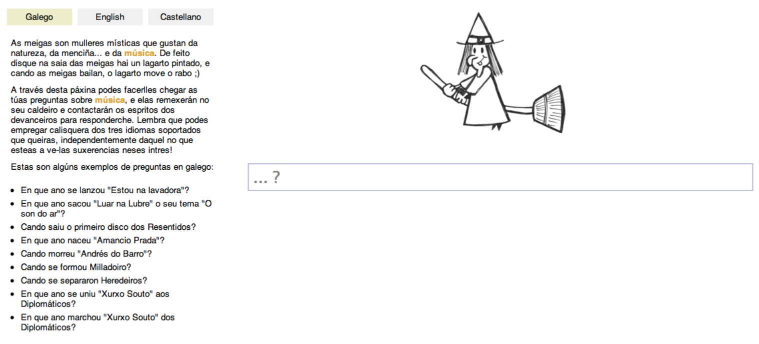
That is why it is much better to make individual functions for each case. In this example, one for release dates of albums, another for release dates of singles, another for release dates of songs.

Another, perhaps, more clear example comes when the user asks for a release of a certain artist. Normally, we would ask the database for releases whose artist matches the name given, pick those with higher matching score, and list their release dates. But if the user specifies that the artist name he's providing belongs to a band, things change. We are unable to filter the results in the previous case to bands. We have to first asks for bands that match the given name, then ask for the releases of those with higher score. A completely different approach for sentences that look almost like twins: *"when did the last single by Udo come out?"* and *"when did the last single by the band Udo come out?"*.

As we can see, the underlying implementation of the database (or rather, the "no expectations" approach we are taking with it) forces us to assume nothing, and make everything as flexible and not tied to patterns as possible. By treating each sentence as a unique case, we don't risk mangling the others when we have to make changes to its behaviour.

Furthermore, we can optimize the requests and answer generation for its particular idiosyncrasy. MusicBrainz, for server loading reasons, accepts only one REST request per second. We saw before how when we ask for releases of a band, we must perform around four (one for bands, then one for each of them selected). But if the user doesn't require it to be a band (or singer), we can speed up the process by asking in bulk of all releases matching the given name, and cutting the request time from 4 to 1 second (75% less delay).

3.2 The interface layer



As meigas son mulleres místicas que gustan da natureza, da menciña... e da **música**. De feito disque na saia das meigas hai un lagarto pintado, e cando as meigas bailan, o lagarto move o rabo ;)

A través desta páxina podes facerlles chegar as túas preguntas sobre **música**, e elas remexerán no seu caldeiro e contactarán os espritos dos devanceiros para responderche. Lembra que podes empregar calisquera dos tres idiomas soportados que queiras, independentemente daquel no que esteas a ve-las suxerencias neses intres!

Estas son algúns exemplos de preguntas en galego:

- En que ano se lanzou "Estou na lavadora"?
- En que ano sacou "Luar na Lubre" o seu tema "O son do ar"?
- Cando saiu o primeiro disco dos Resentidos?
- En que ano naceu "Amancio Prada"?
- Cando morreu "Andrés do Barro"?
- Cando se formou Milladoiro?
- Cando se separaron Heredeiros?
- En que ano se uniu "Xurxo Souto" aos Diplomáticos?
- En que ano marchou "Xurxo Souto" dos Diplomáticos?

Figure 3.1: Screenshot of the website

Interaction with the user starts at a webpage [1]. On the left, a small sidebar explains how the application works in the three supported languages. Examples are given, since imitating is the best way of starting to use a new system. A small logo and a textbox on the right keeps the interface non-threatening by being simple.

In the aforementioned textbox, the user inputs the question very much like he would do in a search engine. This string is delivered asynchronally through Ajax to a small PHP script. The use of Ajax (asynchronous Javascript) means that the user won't have his screen or webpage frozen while awaiting a response. Although he can't send other questions, he can still interact with other parts of the page, obtaining a more satisfying experience.

The PHP code opens a TCP socket to the program and makes the exchange, sending the question and receiving the answers. Then it performs a logging operation, adding the timestamp, IP address, question and answer received to a file. This will not only help with debugging, but also allow us to study the behaviour of our users (what kind of questions they ask, misconceptions about the use of the system, etc...).

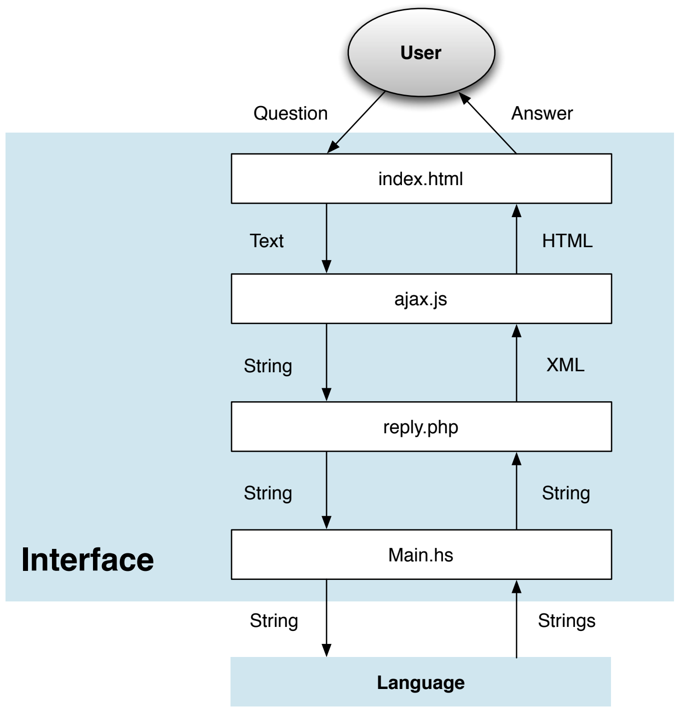


Figure 3.2: Files composing the interface layer

Finally, the PHP file also builds a XML document wrapping the answers and hands it back to the Ajax engine, that extracts the replies and displays them for the user in the website.

3.3 The language layer

The question is handed from the interface "as is", but the first thing done in the Language interface is to adapt it to known issues of the grammar parsing. For example,

[1] `http://diz.es/tese/`

the Grammatical Framework is still developing its token recognition API, so compound names in quotes like

```
"Freddy Mercury"
```

are stripped of the quotes and joined by underscores:

```
Freddy_Mercury
```

so the software identifies them correctly as only one token.

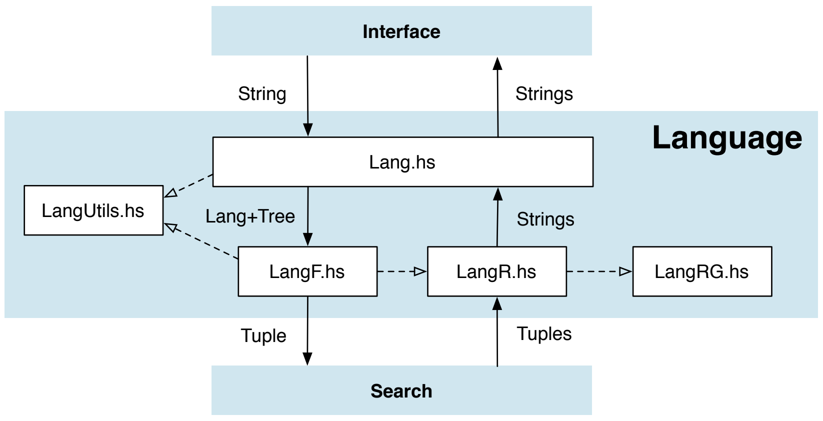


Figure 3.3: Files composing the language layer

After these preparations, the Grammatical Framework API is given the question, for which it returns a list of pairs. Each of them contain a possible abstract parse tree, and the language that tree corresponds to. This is because a given text could have different interpretations, even across different languages. See section 4 for more information on the grammar parsing.

Then the trees are separated by the type of questions they represent, and sent to the functions taking care of each. The purpose of those functions is twofold. First, they examine every possible configuration the tree can take and extract the token information needed to build a query. They forward this tuple to the Search layer. Secondly, they receive the tuple that layer returns, with the answer data, and hand it to the corresponding reply functions that will turn it into another abstract tree first, and then linearize it into a sentence.

As many trees could be interpreted, this layer produces a list of possible answers. Many of the interpretations could be grammatically correct but make no sense, so their answers would be error messages. If any reply in the final list is not an error, all others that are are discarded.

Before answers are handed back to the Interface layer, linearization errors are fixed (mostly, capitals and spacing around punctuation).

3.4 The search layer

The search layer can be understand as an array of arrows. Depending to which one we feed the tuple coming from the language layer, we obtain a different tuple on the other side.

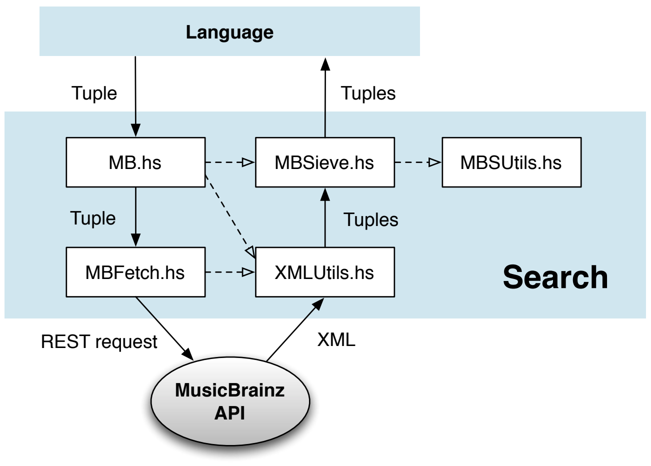


Figure 3.4: Files composing the search layer

Arrows are a Haskell construct proposed by John Hughes [4] that generalize monads. Arrows are defined as a structure that have a type for its input and a type for its output. They can be understood as conveyor belts that take a package on one end, transform it, and put it out again on the other side. Arrows are specially useful because of the easy way in which they can be composed and manipulated. In fact, the battery of arrows in the Search layer are a concatenation of smaller arrows, acting as building blocks.

Every main arrow in the Search layer starts with a fetching arrow that builds the REST call to MusicBrainz. It receives the tuple of data and returns an XML document. Then, this XML document is parsed by several small arrows that locate and filter the tags of the XML document, stripping out non-relevant information until having only tuples of data.

Having these two steps done by combining several smaller arrows (in a Unix-philosophy like approach) makes expansion of the system quite easy. We can reuse a lot of code, since many cases start by asking for the same XML document, but then focusing on different data; or ask for different documents but then perform similar operations on then when filtering the tags.

Finally, the result of the arrow is passed through a final sorting and preparation process before being sent up to the Language layer again.

This is an example of what one of the simpler arrows of this layer looks like:

```
a_treleasebyLastAlbum :: IOSArrow Artist (Artist, Album, Date)
a_treleasebyLastAlbum = (fetchAlbumByArtist >>>
                            ((getArtistName &&& getMyTitle) &&& getEventDate)
                        ) >>. s_treleasebyLastAlbum
```

In this case, the final *s_treleasebyLastAlbum* function is the responsible of taking all pairs of *(Artist, Album, Date)* and returning only the one with the latest date for each album. This purpose is why in this system, all the functions performing selections at the end of the arrows were called *sieve functions*.

4 The grammars

In section 3.3, we explained how the Language layer performs two important operations:

- For input, transforming a text sentence into an abstract tree of objects
- For output, transforming an abstract tree of objects into a text sentence

This falls on the field of computer science called natural language processing. There are several libraries dedicated to parsing and generating text based on lexical and syntax criteria. For these tasks, we opted for using the Grammatical Framework [2].

The Grammatical Framework (or GF for short) is a very powerful tool that takes grammars as input to generate language parsers. These parsers can then move their input back and forth between text and tree representations.

GF has a strong emphasis on multilingual capabilities. Grammars are first written in an abstract, language-independent form. Just categories and functions that transform categories are defined in this file. Then, for each language the grammar needs to understand, different concrete definitions are given, that linearize the categories into text.

This means, for example, that a text can be parsed from one language to an abstract tree, and then this tree linearized into text in another language (that is, a translation). But more importantly for us, it also means that sentences with the same meaning, expressed in different languages, all correspond to the same tree. Is this last point what allows us to parse a question, and regardless of the language it was posed in, decide on the query to perform based only in its semantic meaning. Futher information about how to build grammars can be found in the Grammatical Framework tutorial [6].

The project supports, at the moment of this report, three languages: English, Spanish and Galician. First we'll take a global look at the grammars, and then examine the specifics of each language.

4.1 General notes

Structure of the grammars

Since GF was chosen for its abilities to keep abstract concept and concrete wording separated, we tried to maximize this approach by keeping the abstract tree as vague as possible. This is the reason that only core concepts like *author*, *work* or *date* are present there, instead of *nouns* or *verbs*.

But obviously, when we delve deeper into specifics of a language, these concepts (usually nouns) don't show up isolated. They are part of phrases, with articles and prepositions and suffixes, connected to each other.

We thought that building a complex pile of categories would prove cumbersome when traversing the concrete tree to look for the relevant token. It would also spill some of the concrete inner workings into the abstract grammar definition (since all the category transformations need to be represented there). Therefore, it was decided that this phrasal building and declension was to be done by applying operations to the underlying nouns. Operations keep their workings to the language specific implementation and help maintain

[2]`http://www.grammaticalframework.org/`

the tree code clean. The approach was based on the methodology employed for converting words from a lemma into an inflection table (what is called a *paradigm* in Grammatical Framework terms).

This decision cannot be made in all projects of course. A more ambitious attempt at writing the grammar of a whole language would be foolish to try this method since it would make a bigger mess than the one it is trying to avoid. But since we are dealing with a very specific domain and therefore a very restricted set of question templates, we thought we could take advantage of it. Refer to the individual language sections to see this method at work and the simplifications it brought.

Organization-wise, each language has several files defining it. Let's take English for example. In **BasicsEng.gf** we define the basic categories: Work, Date and Author. These are the core concepts that will show up quite often. Then, in **TimeEng.gf** we define the questions that use those basics to ask questions related with time (the ones starting with *"when...?"*). For the previously mentioned declension, we define the functions that build noun phrases in the resources file, **ResEng.gf**. And finally, **GrammarEng.gf** defines the error messages and provides an entry point for the grammar.

The undefinition challenge

There was one issue we had to face in all the grammars. When the user provides a token name without any more indication (e.g. *Metallica*, *Björk*, *"Rolling Stones"*), it could be anything. A band, a boy, a girl... This presented problems whenever we needed agreement in gender or number with other elements of the sentence.

4.2 The English grammar

English has a very simple morphology. Verbs have the mostly the same form for all persons in a given tense. In present, for example, only the third of singular changes; in the past, is the first and third of singular. In questions it's even easier: in the past, they are all the same. Also, most of the words and adjectives have the same form in both genders. These facts made quite easy to draft the grammar, since we had to deal with very few conditions and variations.

So, going by our general approach, we wrote inflexing operations that allowed us to put articles and possessives in front of the nouns to form their clauses. We can see a use example in this template for questions about births of people:

```
TBorn t a = let time = t.s in
            let as = variants { a.s ! AThe ; a.s ! ANone } in
            { s = time ++ "was" ++ as ++ "born"; };
```

Accepting the name with and without article means both *when was the singer "Frank Sinatra" born?* and *when was singer "Frank Sinatra" born?* are valid.

But because of the general challenge presented by the undefinition of tokens, in the end it was impossible to use possessives. They do not only change depending on number, but also gender of the subject possessing. Best we could do was to guess, and it looked bad most of the times. So even the functionality is there, is only accepted in questions and never used in replies (determiners are used instead):

```
When did Björk release her first single?
Björk released the first single Human Behaviour in 1993-06-07.
```

4.3 The Spanish grammar

Spanish is a romance language, which means is much more prone to gendered nouns (in fact, every noun has a gender and usually they are different). This means also articles, adjectives and any other modifier of the noun has to be in agreement of gender and number with it. Even though verbs don't discriminate for gender, they have all of their forms different depending on the person (first, second, etc), with much more variety than English.

This meant our elements (Author, Work, not needed for Date) had to be defined with a gender and a number. For example, depending if we refer to Metallica as group or band, it could be masculine or feminine (*el grupo*, *la banda*).

We have a lucky break in the sense that this time we can use possesives, being that in Spanish hey don't depend on the subject possessing, rather, they only change depending on the number of the object being possessed. And since all elements we were working with (songs, albums, singles) were singular, we could use the form *su* for all of them:

```
¿Cuándo sacó Björk su canción Bachelorette?
¿Cuándo sacó Björk su último álbum?
¿Cuándo sacó Björk su último single?
```

The most important change, if we compare it to the previous English grammar, is the need for the flexing operations to take into account not only the article we want to prepend to the noun, but also the preposition! This is because Spanish has two contractions: *al* (*a* + *el*) and *del* (*de* + *el*).

So when the noun being flexed is masculine singular, the function makes special care for the case we want to use that combination of preposition and determiner, othersiwe resorting to a simple default concatenation of the terms:

```
  [...]
  table {
    <GMasc,NSg> => table {
        <PAa,DEl> => "al" ++ s ;
        <PDe,DEl> => "del" ++ s ;
        <p,d> => pf ! p ++ df ! <GMasc,NSg> ! d ++ s
      } ;
  [...]
```

We can see this use of the preposition and determiner flexing (and one of the cases for the contraction) in the answer for the "leaving a band" question:

```
TALeave a b d = let as = variants { a.s ! <PNo,DNo> ; a.s ! <PNo,DEl> }
                in let bs = variants { b.s ! <PAa,DNo> ; b.s ! <PAa,DEl> ;
                                       b.s ! <PNo,DNo> ; b.s ! <PNo,DEl> }
                in { s = as ++ "dejó" ++ bs ++ "en" ++ d.s };
```

Colloquial article

Flexibility for colloquialisms meant we had to accept things that don't sound *right*. For example, even though the name of the band is *Creedence Clearwater Revival*, it's quite common to refer to them as *la Creedence Clearwater Revival*. So we decided to accept an extra article in front of token names. This means, that strange as it is, *la Metallica* or even *el Björk* are also ok, since the system doesn't have a feeling of what is common use and what's not.

4.4 The Galician grammar

Galician presented the same problems as Spanish, only bigger, and then some. As most of the romance languages, everything had to be in agreement by gender and number, so words were given those same characteristics as we implemented in Spanish.

The problem is Galician is a very agglutinating language, so every combination of preposition and determiner presented a contraction. We had to define every combination manually. That in itself wouldn't be so bad as it is inevitable, but since possessives in Galician are always preceded by definite articles, we had to repeat a lot those contractions. Look at this excerpt and notice how every two lines the contraction needs to be stated again:

```
[...]
<XMasc,NPl> => table {
    <PAa,DOo> => variants{"aos"; "ós"} ++ s ;
    <PAa,DPos> => variants{"aos"; "ós"} ++ "seus" ++ s ;
    <PEn,DOo> => "nos" ++ s ;
    <PEn,DPos> => "nos seus" ++ s ;
    <PDe,DOo> => "dos" ++ s ;
    <PDe,DPos> => "dos seus" ++ s;
    <PNon,d> => df ! <XMasc,NPl> ! d ++ s ;
    <p,DNon> => pf ! p ++ s
  } ;
[...]
```

And this is only the plural masculine case, and not even all possible prepositions are used (only those presented in the questions of the grammar so far). It's easy to imagine how big and wasteful this approach gets as soon as we add more prepositions, or other kinds of words to interact, or if we try to use it with a more complex language.

This made us notice the fact that maybe our approach wasn't so good after all. We could have reused that code if we had defined Prepositions and Determiners as categories that joined nouns to form another categories (Prepositional Phrases and Determiner Phrases). And it would make the code easier to understand. Section 5.3 expands on the thoughts regarding this matter.

5 Conclusions

5.1 Left to implement

We treat questions asking "when" in the same way as "in what year", i.e., we don't filter out months and days from the latter. This was because of time constraints.

Also, right now the tuples returned for a question are just the bare basic needed to reply. With extra queries we could fetch from the database, for example, if a given token is a person or a band, and then discern the correct verb form to use in linearizing the answer sentence:

```
Björk lanzó su primer single Human Behaviour en 1993-06-07.
The Rolling Stones lanzó su primer single Come On / I Want to Be Loved en 1963-06-07.
```

The second sentence is wrong, since the Rolling Stones is a band, so it should use the verb in plural (*lanzaron*).

Finally, the array of possible questions asked is quite far from finished, as well as the languages the system could understand. The purpose of this thesis, nevertheless, was more about making a first tackle into the problem (thinking about the architecture and facing the first-step problems) than realistically try to be comprehensive (since it was unfeasible).

5.2 Problems with MusicBrainz

Gender of Artists

A problem parallel to the person/group issue, is that of gender. But while in the former case is something we haven't implemented yet, in the latter is something that we couldn't know from the information in the database. Right now we work around the issue:

```
Björk released the first single Human Behaviour in 1993-06-07.
```

But it would be more desirable to use natural expressions. Unfortunately the system cannot discern which one of these to use:

```
Björk released her first single Human Behaviour in 1993-06-07.
Björk released his first single Human Behaviour in 1993-06-07.
```

Note that Björk is a woman, so the second sentence is incorrect.

As of the writing of this thesis, a **Next Generation Schema** is being deployed in the MusicBrainz architecture that expands the database with new objects and new fields for the records. One of those additions is a *gender* field for artists, greatly missed when writing the romance language grammars. The lack of it made difficult to properly construct gender concordance in those languages (see section 4.3).

Filtering of Releases

This still doesn't address, though, that release filtering is only done in an inclusive and never exclusive form. That is, we can specify we look for releases that are *Albums* (as in, studio recordings), but then we will not find *Live* recordings. And if we indicate *Album+Live*, then we will only find live studio recordings (an oximoron in itself). Most of the time, what we want to do is to filter out *Singles*, but that option is not contemplated. Boolean support of types is something we believed should be included in the API.

The forced delay in requests

Finally, the one second delay forced upon us by the web service made many of our results too time consuming to determine with certainty. Examine this query:

```
When did Metallica release their last album?
Metallica released the last album Metallica (disc 4) in 2008-11-28.
```

The result is, even though correct, probably not what the user wanted (since it's just a reedition of their 1991 self-titled album). MusicBrainz has an entity called **ReleaseGroup** that joins together different releases of the same piece of work. Accessing the releases through it and then checking the earliest release date for each ReleaseGroup, we could work around this problem.

Unfortunately, for a band like Metallica, with 10 albums, and some of those with up to 7 different physical releases, it could mean at least around 50s of delay just to iterate through all of them and determine which one is the latest. This is unacceptable for the user, but there's no way to fix it in a project like this, working against the web service API instead of a SQL-accessable copy of the database.

5.3 Lessons learned

About user interaction

Revising the logs of the input our users tried to feed the program, we learn one important lesson: nobody reads the instructions. Even though it was clearly written on the webpage, people consistently:

- tried to ask questions about non-music stuff: food, football, politics...
- when they were on topic, ignored the given examples and asked non supported questions
- insisted on typing without spellchecking, then complained when it didn't work
- refused to use quotes to encompass compound names ("Freddy Mercury"), then complained again

The lesson to learn here is to never understimate how badly users will understand your product. And even if it's a hard truth, the author assumes all blame for not clearing things enough. Proposed solutions for the above problems are, correspondingly:

- Make heavy use of pictures and iconography for people to understand the topic at hand at a glance (texts are overlooked or skimmed)
- Try to implement more supported questions (repertory was a bit lacking)
- Try to offer some kind of spellchecking service
- Offer syntax help when parsing fails

About the architecture

The individual case-breaking worked fine for the moment, but started to show some strain when the number of cases grew. There were a lot of different functions and big tuples being passed around.

The solution that this author proposes is to make heavier use of the strong type support of Haskell, and redefine the datatypes to mimic closer the entities existing in the database. This would have an immediate impact of encapsulating data into the types, simplifying the passing of information and being able to discern exaclty the parameters further than just being "Strings".

But also, it would allow us to use overloading when defining the functions, branching them depending on the incoming type. This way we would solve the dicotomy we faced in section 3.1 when we had to choose between making completely unrelated functions or just one with parameters indicating the meaning represented by the incoming data.

About the grammar building

As mentioned in section 4.4 when we were building the grammars, the phrasal forming by operations instead of categories turned out to be a quick but dirty solution. It that worked good in the beginning, but as obstacles arised, it lacked the flexibility or clarity to solve them.

It would have been better to build the syntaxically correct tree and worry about the extra categories in the code, when extracting the tokens. Functions in the softare could mine the relevant information from the tree into datatypes, discarding the grammatical overhead. Trying to simplify the tree in the grammar definition is a spill of functionality from the actual tree processing code, brought on by trying too hard to keep the abstract grammar abstract. In the end, as we saw, this complicated the actual grammar building by forcing us to retype a lot of similar code and was more prone to errors and mistakes.

About the database access

Lastly, the bane of this project was, in many cases, the web service. Simple requests were made complicated by the limit of what information was available with each request, and the one-second delay made questions that required several requests painfully slow (when outright unacceptable, as we saw earlier).

Therefore, it's clear for the author that a project like this can only get serious results when working against a real copy of the database. The Search layer would be greatly simplified thanks to the power of SQL, saving lots of time and code. It would of course imply some extra initial work for the set up of a mirror database server and timely updates of the information from MusicBrainz's database dumps. But after trying the web service (more oriented to little picks of small data), the requirements of this application (heavy on intercrossed data communication) prove that the payoff would be worth it.

References

[1] Various Authors. *MusicBrainz Web Service API Version 1*. 2011. URL: http://musicbrainz.org/doc/XML_Web_Service/Version_1.

[2] David Ferrucci. *Building Watson: A brief overview of the DeepQA Project*. URL: http://www-03.ibm.com/innovation/us/watson/watson-for-a-smarter-planet/building-a-jeopardy-champion/how-watson-works.html.

[3] Hakia. *Semantic Search Technology*. 2010. URL: http://company.hakia.com/new/documents/White%20Paper_Semantic_Search_Technology.pdf.

[4] John Hughes. "Generalising Monads to Arrows". In: *Science of Computer Programming* 37 (2000), pp. 67–111.

[5] Marissa Mayer. *Scaling Google for every user*. 2007. URL: http://video.google.com/videoplay?docid=-7039469220993285507.

[6] Aarne Ranta. *Grammatical Framework Tutorial*. 2010. URL: http://www.grammaticalframework.org/doc/tutorial/gf-tutorial.html.

A Appendix: English grammar

One of the most interesting aspects of this project was the attempt of a new style of grammar building that made phrases out of nouns by using functions instead of a tree of categories. Even though its success can be disputed, it is still an interesting exercise. We attach here the grammar corresponding to the English language as reference.

As mentioned in section 4, grammar definitions for each language are divided in four files: one for basic components, one for the questions and answers made of this components, another one as entry point (with the error messages), and finally the resources file defining those flexing functions.

Source code

BasicsEng.gf

```
concrete BasicsEng of Basics = open ResEng, Prelude in {

  lincat

    Question, Answer,
    WorkUnitType , WorkUnitMod , WorkUnitName ,
    AuthorUnitName,
    Date, ListDate,
    Token = SS ;

    AuthorUnitType = { s : Str ; n : Number } ;

    Author = { s : Article => Str ; n : Number } ;

    Work = { s : Determiner => Number => Str } ;

  lin

    WUTSingle = ss "single" ;
    WUTAlbum = ss "album" ;
    WUTSong = ss "song" ;

    WUMFirst = ss "first" ;
    WUMLast = variants { ss "last" ; ss "latest" } ;
    WUNName t = t ;

    WNamed t = { s = table { _ => table { _ => t.s } } };
    WNamedUnit u n = { s = detflex (u.s ++ n.s) } ;
    WModUnit u m = { s = detflex (m.s ++ u.s) } ;
    WNamedModUnit u m n = { s = detflex (m.s ++ u.s ++ n.s) } ;

    AUTSinger = { s = "singer" ; n = NSg } ;
    AUTBand = { s = variants {"band"; "group" } ; n = NPl } ;
    AUNName t = t ;

    ANamed t = { s = table { _ => t.s } ; n = NUk };
    ANamedUnit u t = { s = artflex (u.s ++ t.s) ; n = u.n } ;

    DNamed t = t ;
    BaseDate d = d ;
    ConsDate h t = { s = h.s ++ "," ++ t.s } ;

    TStr t = t ;

}
```

TimeEng.gf

```
concrete TimeEng of Time = BasicsEng ** open Prelude, ResEng in {

  lincat
    TimeSought = SS;

  lin

    -- questions

    TRelease t w = let time = t.s in
                   let wa = w.s ! DArt ! NSg  in
                   { s = time ++ variants {
                                "did" ++ wa ++ "come out";
                                 "was" ++ wa ++ "released" } };

    TReleaseBy t w a = let time = t.s in
                       let attrib = variants { "by"; "of" } in
                       let wa = w.s ! DArt ! a.n in
                       let wp = variants { w.s ! DPos ! a.n ; w.s ! DArt ! a.n } in
                       let a = variants { a.s ! AThe ; a.s ! ANone } in
                       { s = time ++  variants {
                                      "did" ++ wa ++ attrib ++ a ++ "come out";
                                      "was" ++ wa ++ "released by" ++ a;
                                      "was" ++ wa ++ attrib ++ a ++ "released";
                                      "did" ++ a ++ "release" ++ wp ; } };

    TBorn t a = let time = t.s in
                let as = variants { a.s ! AThe ; a.s ! ANone } in
                { s = time ++ "was" ++ as ++ "born"; };

    TFormed t a = let time = t.s in
                    let as = variants { a.s ! AThe ; a.s ! ANone } in
                    { s = time ++ variants {"was";"were"} ++ as ++ variants{"formed";"
                        founded";"created"}; };

    TDie t a = let time = t.s in
               let as = variants { a.s ! AThe ; a.s ! ANone } in
               { s = time ++ "did" ++ as ++ "die"; };

    TDissolved t a = let time = t.s in
                       let as = variants { a.s ! AThe ; a.s ! ANone } in
                        { s = time ++ variants {"was";"were"} ++ as ++ "dissolved"; };

    TJoin t a z = let time = t.s in
                  let as = variants { a.s ! AThe ; a.s ! ANone } in
                   let zs = variants { z.s ! AThe ; z.s ! ANone } in
                   { s = time ++ "did" ++ as ++ "join" ++ zs; };

    TLeave t a z = let time = t.s in
                   let as = variants { a.s ! AThe ; a.s ! ANone } in
                   let zs = variants { z.s ! AThe ; z.s ! ANone } in
                     { s = time ++ "did" ++ as ++ "leave" ++ zs; };

    TSWhen = ss "when";
    TSYear = ss "in what year" ;


    -- answers


    TARelease a w d = { s = a.s ! AThe ++ "released" ++ w.s ! DArt ! a.n ++ "in" ++ d.s };
    TAReleaseX a w x d = { s = a.s ! AThe ++ "released" ++ w.s ! DArt ! a.n ++
                              "(from" ++ x.s ! DArt ! a.n ++ ")" ++ "in" ++ d.s };

    TABorn a d = { s = a.s ! AThe ++ "was born in" ++ d.s };
    TADie a d = { s = a.s ! AThe ++ "died in" ++ d.s };

    TAFormed a d = { s = a.s ! AThe ++ "was formed in" ++ d.s };
    TADissolved a d = { s = a.s ! AThe ++ "was dissolved in" ++ d.s };

    TAJoin a b d = { s = a.s ! AThe ++ "joined" ++ b.s ! AThe ++ "in" ++ d.s };
    TALeave a b d = { s = a.s ! AThe ++ "left" ++ b.s ! AThe ++ "in" ++ d.s };
}
```

GrammarEng.gf

```
concrete GrammarEng of Grammar = TimeEng ** open Prelude, ResEng in {

  lin

    AEAuthorMissing = ss "We think you forgot the artist";
    AENoResults = ss "We don't have the answer to that question";
    AENotImplemented = ss "We can't answer that yet";
    AENoParse = ss "We don't understand the question";
    AENoSense = ss "That makes no sense";

}
```

ResEng.gf

```
resource ResEng = {

  param Determiner = DArt | DPos | DNone ;
  param Article = AThe | ANone ;
  param Number = NSg | NPl | NUk;

  oper detflex : Str -> (Determiner => (Number => Str)) = \x ->
    table {
      DArt => table { _ => (artflex x) ! AThe } ;
      DPos => table {
        NSg => variants { "his" ; "her" ; "its" } ++ x ;
        NPl => "their" ++ x;
        NUk => variants { "the"; "its"; "their"; "his"; "her" } ++ x
        -- "the" is used as default because of the number/gender problem
        -- but all are accepted, the user knows best!
      } ;
      DNone => table { _ => (artflex x) ! ANone }
    } ;

  oper artflex : Str -> (Article => Str) = \x ->
    table {
      AThe => "the" ++ x;
      ANone => x
    } ;

}
```