Question Answering over BioMedical Linked Data with Grammatical Framework

Anca Marginean a,∗

a Department of Computer Science, Technical University of Cluj Napoca, 401446 Cluj-Napoca, Romania

Abstract. The blending of linked data with ontologies leverages the access to data. GFMed introduces grammars for a controlled natural language targeted towards biomedical linked data and the corresponding controlled SPARQL language. The grammars are described in Grammatical Framework and introduce linguistic and SPARQL phrases mostly about drugs, diseases and relationships between them. The semantic and linguistic chunks correspond to Description Logic constructors. Problems and solutions for querying the datasets with Romanian, beside English, are described in the context of GF.

Keywords: querying linked data, description logics, controlled natural language, multilingual system, Grammatical Framework

1. Introduction

Linked Data means using the Web to connect related data. A large number of data from various domains such as government data, education, life sciences, art and others were made available in the context of the Linked Open Data initiative built around DBpedia. One of the greatest challenges of this new big set of data is querying it. In order to fill the gap between end users and formal languages like SPARQL, more approaches emerged: querying in full natural language [18], or in Controlled Natural Languages [3], [7], [5], or incremental query building [17].

The dimension of the terminology used for building the data is one impediment to be surpassed in order to query it. The lack of previous and detailed knowledge about the structure of the data makes the querying task tedious, even for users well adjusted to the semantic web technologies. Not to mention the common situation where more datasets are required in order to find the desired data.

In the context of the large adoption of ontologies and semantic reasoning on one hand, and the development of techniques for natural language processing on the other hand, the aim of having the natural language as a mean to query linked data becomes more feasible. Querying databases with meaning representation languages that rely on natural language is an old idea. CHILL system [16] used such a language for querying Geobase, a set of Prolog facts. The specialized parser for the language was learned through inductive logic programming. More recently, statistical machine learning was used. With an increasing popularity, Controlled natural languages aim at giving an intuitive representation to formal representations, by making a trade-off between precision of formal languages and ambiguity, but high expressiveness, of natural languages. A comparative analysis of controlled natural languages is done in [8].

Following the controlled natural language line, we propose a system based on application grammars manually built with Grammatical Framework (GF) [11]. The goal is to provide a natural, yet precise way of querying linked data. The starting point is the relation between the meaning of expressions in natural language and the
constructors of description logics. This relation is also analyzed in [7], while GF is also used for linked data in [3], [4]. The targeted linked data are biomedical data, described in Diseasome, Sider and Drugbank. These three sets were proposed in Task 2, Biomedical question answering over linked data, of the Question Answering over Linked Data (QALD-4) [13].

DrugBank is part of the project Bio2RDF [2] and gives chemical, pharmacological and pharmaceutical data about drugs with comprehensive drug target information. Diseasome provides information about human disease-gene network, while SIDER relates drugs to their adverse reactions. The Linked Data version of Diseasome publishes a network of 4300 disorders and disease genes, as well as possible drugs for diseases. SIDER includes 4192 side effects, 996 drugs and 99423 drug/side-effect pairs. The three datasets are connected: diseases from Diseasome are related to drugs described in DrugBank, DrugBank drugs are related to Sider drugs with the relation sameAs and there are side effects from Sider that are also diseases.

Grammatical Framework is a special purpose programming language with a high support for multilingual applications. Therefore a multilingual system was investigated starting from the grammars proposed for English, with Romanian as the second natural language.

The article is structured in 6 sections. Section 2 introduces the GF resource library that we propose for SPARQL. Section 3 describes the main functions from the abstract and concrete grammars which define the controlled language. Section 4 introduces our first attempt in building a multilingual language for querying biomedical data. Related work is analyzed in Section 5, while some conclusions are drawn in Section 6.

2. Controlled natural language with GF

The general workflow of a system for querying linked data with GF is detailed in figure 1.

GF grammars are divided into abstract and concrete grammars. An abstract grammar defines categories and functions. Each category stands for a set of trees. Functions produce trees of certain categories. The linearization types and functions are defined in concrete grammars. For each category, a linearization type is defined and for each function, a linearization function. Based on the abstract grammar and the concrete grammars for each language, GF is able to translate a phrase from one language to another by parsing it first into an abstract tree and then linearizing it by means of the concrete grammars.

GF has support for syntax, lexicon and inflections in 36 languages. It comes with a comprehensive library for English [11]. GFMed English concrete grammar relies on this library for the syntax, morphological paradigms used to introduce new elements in the lexicon, and coordination.

2.1. SPARQL resource library

The first step in using GF for querying linked data is to create support for the language SPARQL. From our knowledge, even though there are applications of GF for this task [3], GF libraries do not include resources for SPARQL.

Two main types were defined: Triplet and Statement. The type Triplet : Type = \{subj, obj : Str; prop : PropertyT\}; is a structure with three components; two string components for the subject and the object, and a structure for the property. The structure PropertyT : Type = \{s : Str; vt : ValueType\} for the property includes also two fields: a string for the name of the property and a field of the enumerated type ValueType. The field stores the type of the values for the property. The enumerated type has two values: String and Number. The type of the property is important in using the correct operator inside the FILTER expressions. For the moment, only equality and the regex operators are included.

By default, the regex operator is used for properties that have strings as values, while the equality operator is used for the properties with numerical values.
mkStatement : Triplet → Statement =

\( vp \rightarrow \{ s = vp.subj + +vp.prop.s + +vp.obj; extra = \"\"; aggreg = no\}\);

mkFilterStatement : PropertyT → Str → Str → Statement =

\( p, x, v_2 \rightarrow \text{case } p.vt \text{ of } \{

\text{String } \Rightarrow \{ s = "FILTER(regex(" + +v_2 + +" + +x + +" + +i)\"; extra = \"\"; aggreg = no\};

\text{Number } \Rightarrow \{ s = "FILTER(" + +v_2 + +" = " + +x + +")\"; extra = \"\"; aggreg = no\};

\}

\)

addStatement : Triplet → Statement → Statement =

\( vp, st \rightarrow \text{let } s_1 = \text{mkStatement } vp\)

\( \text{in } \{ s = s_1.s + +"." + +st.s; extra = st.extra; aggreg = st.aggreg\}\);

addStatement2 : Statement → Statement → Statement =

\( st_1, st_2 \rightarrow \{ s = st_1.s + +"." + +st_2.s; extra = st_1.extra + +st_2.extra;\)

\(\text{aggreg = case st_1.aggreg of } \{ \text{yes } => \text{yes};\)

\(\text{no } => \text{st_2.aggreg}\} \};

Fig. 2. Functions that create graph patterns

mkEmptyTriplet : PropertyT → Triplet =

\( p \rightarrow \{ subj = \"\"; prop = p; obj = \"\"\}\);

addSubj : Str → Triplet → Triplet =

\( ss, vp \rightarrow \{ subj = ss; prop = vp.prop;\)

\( obj = vp.obj\};

Fig. 3. Functions that create a Triplet

The type Statement : Type = \{ s : Str; extra : Str; aggreg : AggregationType\}; is a structure with three fields. It corresponds to a graph pattern formed by one or more triple patterns. The first component is the string expressing the graph pattern and it is going to be part of the WHERE clause. The third component, aggreg, has two possible values, yes and no, stating whether value aggregation is required. The field extra contains the expression for value aggregation in case it exists.

A set of functions were defined on these types. There are functions that create incomplete triple patterns, as the functions mkEmptyTriplet or addSubj (Figure 3). A subset of the functions that create graph patterns is mentioned in Figure 2. The function mkStatement concatenates the fields of a Triplet and creates a graph pattern with only one triple pattern. The function mkFilterStatement details the default behavior for FILTER expressions for the two defined property types. In order to support other datatypes, the enumerated type ValueType must be extended with other types, and a new field can be added to PropertyT allowing explicit specification in this function of the behavior for properties with a certain type of values. The last two functions build graph patterns with more than one triple pattern. The function addStatement adds a triple pattern to a graph pattern, while the function addStatement2 concatenates two graph patterns.

Finally, there is a set of functions that create the strings for SELECT and ASK queries (Figure 4). The function mkQuery applies on a string that includes the variables to appear in the query results and on a graph pattern. The final string is built according to whether aggregation is part of the query or not.

3. DL based questions for biomedical linked data

Description Logics (DLs) [1] are a family of knowledge representation languages that can be used to represent the knowledge of an application domain in a structured and formally well-
understood way. In the description logic $\mathcal{ALC}$, concepts are built using the set of constructors formed by negation, conjunction, disjunction, value restriction, and existential restriction. Extensions of $\mathcal{ALC}$ introduce inverse roles, number restrictions $(N, Q)$ and concrete domains $(O)$. Even though the three targeted datasets are not all providing for DLs descriptions or ontologies, approaching them from a DLs perspective indicates ways to efficiently split possible questions in semantic chunks that have straightforward translations to SPARQL and are highly composable.

GFMed, the proposed system, consists mainly of a GF grammar for the application domain given by SIDER, Diseasome and DrugBank datasets. GFMed also includes some minor preprocessing of questions and post-processing of translation results, mainly in order to deal with structures involving numeric values, e.g. values for water solubility, or free text, like different names of foods. Figure 5 shows the main grammars: an abstract grammar and two concrete grammars. For each concrete grammar, lexicons derived from SIDER, Diseasome and DrugBank were generated. Many syntactic structures in both English and SPARQL were driven by the datasets’ terminology.

For domain-specific applications, the GF abstract grammar must state the main semantic categories and trees of the language. For GFMed we introduced the following categories: Drug, Target, Disease, Gene, SideEffect and SIDER. Drug corresponding to the main resources in the targeted datasets. We also introduced the categories DrugBankProperty, DiseasomeProperty and SIDERProperty for describing the properties of the same datasets. For each mentioned resource category, classes of these semantic entities are described, resulting DrugClass, DiseaseClass, GeneClass, TargetClass and SideEffectClass. Trees for these categories are built either from a single named resource, or from a restriction on a property. For each $X$Class there are one or more CriterionFor$X$Class categories, where the class can be obtained from the Criteria for that class. For example, drugs that interact with food is a DrugClass tree, while interact with food is a CriterionForDrugClass. In other words, trees of type CriterionFor$X$Class are subtrees of $X$Class trees.

The core of the abstract grammar consists of functions to build trees. GFMed’s functions can be categorized in (i) functions that describe property restrictions, (ii) functions for transforming a criterion of a class into a class or for transformations between different types of classes and properties, (iii) functions expressing queries.

In the concrete SPARQL grammar built on top our SPARQL resource library, each property from the targeted datasets is linearized to a value of type $Triplet$, initially with null object and subject. These two are completed within linearization of different restriction functions: one of them must

\[
\text{mkQuery} : \text{Str} \rightarrow \text{Statement} \rightarrow \text{Str} = \\
\text{\$var, b \rightarrow case b.aggreg of}\{ \\
\text{yes \Rightarrow \"select\++var++} \\
\text{where\{\++b.s++\}}++b.extra; \\
\text{no \Rightarrow \"select distinct\++var++} \\
\text{where\{\++b.s++\}} \} \\
\text{mkAskQuery} : \text{Statement} \rightarrow \text{Str} = \\
\text{\$b \rightarrow \"ask\{\++b.s++\}"};
\]
### Table 1

<table>
<thead>
<tr>
<th>Category</th>
<th>English Examples and Short Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{X} )</td>
<td>( \mathcal{X} \in { \text{Drug, TargetConcept, Gene, Disease, SideEffect, SiderDrug} } )</td>
</tr>
<tr>
<td>DrugBank-Property</td>
<td>MeltingPoint, GeneralFunction, DosageForm, PredictedWaterSolubility, Manufacturers, Indication, Target, Interacting, FoodInteraction,...</td>
</tr>
<tr>
<td>Sider-Property</td>
<td>SideEffect</td>
</tr>
<tr>
<td>Disease-Property</td>
<td>AssociatedGene, PossibleDrug, ClassDegree, Degree, Class, Size, SubtypeOf, ChromosomalLocation</td>
</tr>
<tr>
<td>Property</td>
<td>any kind of property from the above three</td>
</tr>
<tr>
<td>( \mathcal{X}\text{Class} )</td>
<td>classes formed of a single named ( \mathcal{X} ) entity: \text{Leprudin}, or from drugs described by a criterion: \text{drugs that target Prothrombin DrugClass, TargetClass, SideEffectClass, SiderDrugClass, DiseaseClass, GeneClass, PropertyClass}</td>
</tr>
<tr>
<td>Criterion-For( \mathcal{X}\text{Class} )( Y )</td>
<td>criterion for getting a class of ( \mathcal{X} ), expressed by an ( Y ) syntactic structure</td>
</tr>
<tr>
<td>NP</td>
<td>\text{Leprudin as possible drug}</td>
</tr>
<tr>
<td>Adv</td>
<td>\text{with Leprudin as possible drug}</td>
</tr>
<tr>
<td>AP</td>
<td>\text{treated with Leprudin, indicated for Fever}</td>
</tr>
<tr>
<td>VP</td>
<td>\text{treat Tuberculosis}</td>
</tr>
<tr>
<td>RCl</td>
<td>\text{whose possible drug is Leprudin, whose possible drugs interact with Leprudin}</td>
</tr>
<tr>
<td>ClSlash</td>
<td>\text{Leprudin is used for}</td>
</tr>
<tr>
<td>Question</td>
<td>\text{which drugs interact with food}</td>
</tr>
<tr>
<td>Utterance</td>
<td>utterances from affirmative clauses \text{List the drugs that...} or from question clauses \text{What are the drugs that ...}</td>
</tr>
</tbody>
</table>

**3.1. Building Trees for Property Restrictions**

In DLs, there are two types of roles or properties: object properties and datatype properties. Object properties relate two concepts, while datatype properties relate one concept to a value of a certain datatype. For example, the object property \text{SideEffect} connects the resources Penicillin\( G \) and Fever. Differently, the \text{Mechanism of Action} property relates the drug \text{Lepirudin} to a string value.
Object properties  Object properties and datatype properties are treated differently in the GFMed grammar. When it comes to object properties, the DL existential restriction $\exists R.C$ on property $R$ describes the set of individuals having as value of the property (role) $R$ an individual from the concept $C$. For example, $\exists$ PossibleDrug.FeverInducingDrug is a restriction on property PossibleDrug whose interpretation, if it exists, is the set of all diseases that have at least one possible drug from the class FeverInducingDrug. FeverInducingDrug stands for all drugs that have fever as a side effect and it is a value restriction, with value Fever, on the property SideEffect. Some more examples are given in table 2. We consider that even though the targeted datasets are not described in DL, in order to have language constructors able to be composed based on their type, the functions to build trees in either English or SPARQL can be described similarly to DL constructors. This approach is also justified by the functional style of the chosen grammatical framework.

Each DL constructor can be expressed in natural language in more ways, either as noun phrase (NP), verbal phrase (VP), adjectival phrase (AP), verb-phrase-modifying adverb (Adv), relative clause (RCl) or clause with some missing part (ClSlash). These syntactic categories are defined by GF library. To each DL constructor identified at a conceptual level correspond more functions to build trees at concrete English level, one for each possible syntactic structure. Figure 6 shows functions that model restriction on the property PossibleDrug with values in DrugClass. In a similar manner, functions for restriction on the inverse property of PossibleDrug are defined. They allow statements about drugs used to treat a certain disease or a disease class. For all object properties, the abstract and concrete grammars include sets of functions to express existential and value restrictions on them. Since classeses formed from only one named drug are allowed, hasValue restrictions on object properties can be treated in the same way as existential restrictions.

Within this approach, treated by interferon beta-1a is identified as (WithPossibleDrugsCriterion (SingleDrug DB00060)), and its SPARQL linearization is the graph pattern ?dis ds:disease/possibleDrug db:drugs/DB00060 . ?dis a ds:diseases.

Datatype properties When it comes to restrictions on datatype properties, the English methods to express them are not anymore particular to each property, therefore it is possible to treat all with the same set of functions. Some examples are described in Fig. 7. The property becomes one of the functions’ parameters. The most important issue is that it is not possible to include all actual values in the grammar, because the set of values is not finite. This issue can not be completely solved in GF. The proposed solution is to include in the grammar generic trees with a dummy string. If the translation to SPARQL succeeds, the dummy value is replaced in the generated query during post-processing. Since the values for these restrictions tend to appear at the end of the question, e.g. Give me the side effects of drugs with a solubility of $3.24e-02$ mg/mL, in the preprocessing phase the string value is replaced with the dummy value and the question to be parsed becomes Give me the side effects of drugs with a solubility of XX. This is identified as [GiveSider-
Valuerestriction : DrugBankProperty → CriterionForDrugClass
Valuerestriction Adj → CriterionForTargetClass
ValuerestrictionRCl : DrugBankProperty → CriterionForDrugClassRCl
DiseaseValuerestriction : DiseaseomeProperty → CriterionForDiseaseClassNP
DiseaseValuerestrictionRCl : DiseaseomeProperty → CriterionForDiseaseClassRCl
LowestNumber : Property → CriterionForDrugClass
DiseaseWithLowestValue : DiseaseomeProperty → CriterionForDiseaseClassNP;
LowestNumberValue : Property → PropertyClass;

Fig. 7. Functions for restrictions on datatype properties

Property SideEffect [ToDrugClass [Valuerestriction Solubility]], where SideEffect indicates the object property whose value is asked for. The content of the innermost brackets represents the drugs indicated by the transformation to DrugClass of a value restriction on the datatype property Solubility. Another possible solution for covering numerical values for these restrictions could be based on the GF support for integers and floating point numbers.

Other described constructors include HighestNumber, LowestNumber, ZeroNumber, which are focused on the number of properties, or HighestValue, and LowestValue which are focused on values of properties. For example, the least common chromosome location is interpreted as [LowestValue ChromosomeLocation], where ChromosomeLocation is a DrugBank property.

Figure 8 details two examples. The function HighestNumber is used in the first example and its SPARQL linearization includes an aggregation function. The graph pattern for the WHERE clause is stored in the component s of the structure Statement (from the SPARQL resource library), and the description for the aggregation function is stored in the component extra of the same structure Statement. A second example includes a filter on the value of the property state from DrugBank.

3.2. Transformation Functions

For composability reasons, transformation functions are defined for getting from a criterion to a class, or for getting from one dataset to another.

The former are important for English linearization, while the latter play an important role in SPARQL linearization.

The first transformation functions take criteria and build on them the upper level linguistic structures needed in queries. For example, in order to get to the Noun Phrase drugs used for Rickets from the Adjectival Phrase used from Rickets, there is a transformation from CriterionForDrugClassAdj to DrugClass that adds the noun drugs to linearization of the AP. When building SPARQL queries, these transformation functions do not alter the linearization of the Criterion, because the corresponding SPARQL triplets are already completely built. All the English alternatives for expressing a conceptual DL constructor have the same SPARQL linearization. This is somehow expected, as SPARQL is a formal language tightly related to DLs.

The second type of transformations deals with queries requesting access to more datasets. In this case, English linearization does not alter the object of transformation, while the SPARQL lin-
treated by drugs with no side effect

?dis a ds : disease some/diseases.
FILTER NOT EXISTS {
?siderdrug sd : sider/sideEffect vp \}
?siderdrug a sd : sider/drugs.
?drug owl : sameAs ?siderdrug

Fig. 9. Abstract tree and concrete linearizations in English and SPARQL for an example with two transformation functions:

\[\text{WhichDisease2} : \text{DiseaseClass} \rightarrow \text{Question}] - - which are the diseases caused by D?
\[\text{WhichDisease} : \text{CriterionForDiseaseClass} \rightarrow \text{Question}] - - which diseases are caused by D?
\[\text{WhichTargetAdj} : \text{ValueRestrictionAdj} \rightarrow \text{Question}] - - which targets are involved in XX?
\[\text{WhatPropertyValue} : \text{PropertyClass} \rightarrow \text{Question}] - - which is the least common chromosome location?

Fig. 10. Functions for queries

earization introduces new variables and sameAs statements. For example, the function DBToSiderDrug converts the class of DrugBank drugs to the class of Sider drugs. Its SPARQL linearization introduces a new variable ?siderdrug that is related with a sameAs statement to the variable of the function’s parameter.

\[\text{DBToSiderDrug} : \text{DrugClass} \rightarrow \text{SiderDrugClass}]\]
\[\text{DBToSiderDrug} d = \{\text{var} = "]?siderdrug"; \]
\[\text{body} = \text{addStatement2} \]
\[\text{(mkSameAsStatement} "?siderdrug" d.var) \]
\[\text{d.body}]\}; - - concrete SPARQL
\[\text{DBToSiderDrug} d = d; - - concrete English

An example combining two transformation functions is given in Figure 9. The innermost tree \(t_1\) is a criterion for a class of Sider drugs [SiderZeroNumber SideEffect]. The tree \(t_2 = [\text{ToDrugClass Criterion} t_1]\) is determined by a transformation function between the sets Sider and Drugbank, while \(t_3 = [\text{ToDrugClass} t_2]\) corresponds to a transformation from a criterion to a class.

3.3. Functions for Queries

Several types of queries were identified: give, list, which, what, for/with which, and is/are there. They are applied on one class, one criterion, or on a list of classes or criteria for classes (Figure 10). The questions deal mostly with resource classes and criteria for these classes and less with properties. An exception to this rule is the question WhatPropertyValue. This question treats PropertyClass instead of a resource class, because it queries for information about a property class and not about a property of some resource. For example, the question which is the least common chromosome location is parsed to the abstract tree [WhatPropertyValue [LowestNumberValue [DBToProperty ChromosomeLocation]]]. Its SPARQL linearization requires aggregation and sort operations.

The advantage of taking the described approach is the flexibility in composition of trees/constructors, based on their types and transformation functions. For example, drugs that interact with the drugs used for diseases treated by tetracycline is parsed to the abstract tree \(t_3 = [\text{ToDrugClass withThatVP [DDrugClassCriterion VP} t_2]\), where \(t_2 = [\text{AdjToDrugClass PossibleDrugsForCriterionAdj} t_1]\) is the tree for the class of drugs that are used for diseases in \(t_1, t_1 = [\text{ToDiseaseClass WithPossibleDrugsCriterion [SingleDrug DB00759]]}\) stands for a DiseaseClass of diseases treated by tetracycline. DB00759 is the DrugBank ID for tetracycline. The abstract tree \(t_3\) is linearized in the SPARQL concrete grammar. By running the query, we get drugs which inter-
act with tetracycline, and also other drugs used to treat the same diseases as tetracycline.

The grammars can be used not only for translating natural language questions into SPARQL, but also for SPARQL queries to natural language questions, for the phrases that are not involved in the pre/post-processing of GFMed. For example, the next query is identified as being the SPARQL linearization of 17 different abstract trees.

SELECT DISTINCT ?possDrug
WHERE {
  ds:diseases/173
  ds:diseaseSome/possibleDrug ?possDrug
}

3.4. Pre- and Post-processing

GF comes with an HTTP server that supports REST services for its main functionality, as translation or parsing. GFMed includes (i) the abstract grammar and the concrete grammars for English and SPARQL described previously, and (ii) a Java standalone application that consumes GF translation service based on these grammars.

The standalone application includes a pre-processing module, a module for consuming the translation service, and a post-processing module. Algorithm 1 describes the main steps of the translation from a natural language to SPARQL.

Preprocessing includes a simple transformation of the question to lowercase, and a failure handling method. When the translation module gets a failure from the server, the failure handling method repeatedly trims the last word of the question and replaces the trimmed sequence with the dummy string XX. Thus, the previously mentioned phrase becomes drugs whose mechanism of action involves XX and YY. In case the translation works, XX is replaced with norepinephrine in the resulting query, and YY with serotonin.

After a successful translation, the post-processing module searches for the abstract tree with the smallest length. This is needed because is is possible to have more alternatives for translating the questions, mainly due to the transformation functions. Once the tree is found, its SPARQL linearization is extracted. In case it was a value restriction, solved by the failure handling method, some replacements are done.

3.5. Generated Lexicons

GF grammars must know, at compilation time, all the tokens that are part of the analyzed text. Therefore, GFMed includes lexicons for both SPARQL and English formed of all drugs, targets, diseases, genes, and side effects extracted from the three datasets (Figure 5).

These lexicons where generated from the data sources available on the sites of the three datasets,
Table 3

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Distinct Ids</th>
<th>Distinct Names</th>
<th>Considered properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrugBank</td>
<td>1470</td>
<td>22872</td>
<td>db : name,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>db : synonym,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>db : brandName</td>
</tr>
<tr>
<td>DrugBank</td>
<td>4553</td>
<td>3784</td>
<td>drugbank : name</td>
</tr>
<tr>
<td>Target</td>
<td>4213</td>
<td>3642</td>
<td>diseasome : name</td>
</tr>
<tr>
<td>Disease</td>
<td>3919</td>
<td>4328</td>
<td>rdfs : label,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>owl : sameAs</td>
</tr>
<tr>
<td>SIDER</td>
<td>1737</td>
<td>2398</td>
<td>sd : sideEffect-Name</td>
</tr>
</tbody>
</table>

Table 4

<table>
<thead>
<tr>
<th></th>
<th>Total/ Processed</th>
<th>Right/ Partially</th>
<th>Recall</th>
<th>Precision</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25/25</td>
<td>24/1</td>
<td>0.99</td>
<td>1</td>
<td>0.99</td>
</tr>
</tbody>
</table>

either by using SPARQL endpoints, or by parsing RDF files. Also, the same results were obtained from executing SPARQL queries on the QALD-provided endpoint. Special attention was given to side effects, drugs, and genes. For the same ID of a side effect more synonym names are known, expressed through the property sideEffectName. For one drug ID in DrugBank, there are more names and synonyms. Furthermore, as the name, the synonyms and the brand names of a drug can appear in a question, English linearization of each drug includes alternatives expressed by values of properties name, synonym, and brandName. For Genes, besides the property rdfs : label, it was considered the property owl : sameAs that relates some genes to DBpedia resources. Extended names for genes are extracted from these resources.

Table 3 shows the number of resources identified in this way, giving both the number of distinct IDs and distinct names for each category.

3.6. Results and Their Analysis

The system was evaluated against training and test questions of Task 2 in QALD4 and obtained the overall evaluation from the table 4.

GFMed correctly parsed all the questions, except one. It partially parsed question 21, Give me the drug categories of Desoxyn, for which it obtained 0.85714 recall and precision 1, meaning that all the answers retrieved by the proposed query were correct, but they were not complete. The reason for this is that Desoxyn is a brand name for drugs with DrugBank IDs DB00182, DB01576, DB01577. We wrongly assumed that one brand name can be associated either to only one drug, or to several drugs but with consistent descriptions. The drug DB00182 has one more category compared to the other two drugs: amphetamines. GFMed identified the drug as being DB01577 so it missed this category. Given the fact that more drugs with different names and different descriptions can have the same brand name, we think that the lexicon should treat the names differently compared to brand names. Instead of having one drug with alternative linearizations for both name and brand name, it would be better to linearize in English a drug ID only to its name and synonyms.

4. First steps towards a multilingual system

Grammatical Framework is a programming language for multilingual grammar applications. Therefore, starting from GFMed’s concrete grammars for English, the first steps were made towards a multilingual application. The Romanian language was considered, beside English.

The default GF mechanism for building multilingual grammar applications is through incomplete grammars that are language independent. The name of the properties from the schemas of the three sets were translated manually in Romanian, resulting in two lexicons, one for each language. The general syntactic patterns are described in the incomplete grammar, that is extended by two concrete grammars, one for English and one for Romanian. In Figure 11, two two-place adjectives are defined in the English and the Romanian lexicons. The criterion for drugs based on the inverse property of the possibleDrugs property is defined using the functions from the lexicons. The Figure 11 also contains an example of a property from DrugBank, MeltingPoint, as it is defined in Romanian lexicon.

The main problems in using GF library for Romanian were identified for expressing relational nouns and genitive relative phrases. Unlike English prepositions, Romanian prepositions have cases,
- **english lexicon**

`UsedFor_A2 = P.mkA "used" for_Prep;`

`Treated_A2 = P.mkA "treated" by_means_Prep`

`| P.mkA "treated" with_Prep;`

- **romanian lexicon**

`UsedFor_A2 = P.mkA ("utilizat") for_Prep;`

`Treated_A2 = P.mkA ("tratat") with_Prep`

`| P.mkA ("tratat") by_means_Prep;`

**Route of Elimination_CN** = `mkCN`

(P.mkN2 (P.mkN "cale" "căi") (P.mkPrep "de" Ac))

(mkNP (P.mkN "eliminare");)

- **incomplete grammar**

`PossibleDrugsForCriterionAdj disc` =

`mkAP UsedFor_A2 disc;`

`PossibleDrugsForCriterionVP disc` =

`mkVP Treated_V2 disc;`

---

Fig. 11. Functions from lexicons and incomplete grammar

and the nouns and adjectives have different forms for different cases. GF’s resource library has a good support for these cases, but we experienced a problem with expressing nouns in the genitive case due to the expected presence of the possessive article. The correct forms are *solubilitatea medicamentului*, *solubilitate a medicamentului*, *efect advers al medicamentului*, where the presence of the possessive article is variable. The forms of this article (a, al, ai, ale) are also described in GF library for Romanian, but, as far as we know, they are not used in dealing with the genitive case of nouns. As a result, all the nouns in genitive are wrongly built without the possessive article. The current solution was to consider the preposition *pentru (for)* instead, which requires a noun in accusative. For example, the phrase *side effect of the drug* has as correspondent the phrase *efect advers pentru medicamentul*, which means *side effect for the drug*.

The set of recognized questions in Romanian is smaller compared to that in English. The questions involving genitive relative phrases, as *diseases whose possible drug*, are not treated.

### 4.1. The Romanian names for human diseases

Apart from having phrases of common sense knowledge expressed in different languages, an important issue in building multilingual systems is the domain specific terminology. When there are structured versions of the terminology in different languages, the problem is easier to solve. For example, Medical Dictionary for Regulatory Activities (MedDRA) [9] is a multilingual terminology developed in order to provide a single standardised international medical terminology which can be used for regulatory communication and evaluation of data pertaining to medicinal products for human use. Unfortunately, Romanian is not included in the set of used languages.

However, based on existing classifications of disorders as OMIM, ICD and Orphanet, we investigated a solution to build a Romanian lexicon for the named diseases from Diseasome.

Online Mendelian Inheritance in Man (OMIM) captures the relation between genes and disorders. Each disorder has an OMIM code. This catalog is continuously updated. The initial bipartite graph from Diseasome was build on the OMIM version from 2005 [6], while the 2012 version of Diseasome was extended with drugs.

ICD-10 classification is the 10th revision of the International Classification of Diseases and Related Health Problems. There is a Romanian version of the Australian version ICD-10-AM, that is used in Romanian hospitals. Diagnosis Related Group (DRG)¹ provides for an application that includes this classification.

Orphanet is a portal for rare diseases, led by a consortium of around 40 countries. One of its freely accessible services is a classification of diseases elaborated using existing published expert classifications². The included alignments between disorders and external terminologies or resources, as OMIM, ICD10, MeSH, UMLS and MedDRA, are characterised in order to specify if the terms are perfectly equivalent (exact mappings) or not. There are versions for 7 languages, but again Romanian is not included.

Bio2RDF is a project that aims at providing Linked Data for the Life Sciences [2] and includes

¹[www.drg.ro](http://www.drg.ro)

many medical resources formalized with RDF. DBpedia also contains information for diseases, including OMIM, MeSH or ICD identifiers, and names in different languages. One way to use it for the task of building Romanian lexicon is to rely on a mapping between English and Romanian names without the use of ICD classification. If in case of other languages this could be an acceptable solution, in case of Romanian, the small number of resources in this language make it inappropriate. The alternative is to rely on DBpedia for identification of ICD codes from OMIM codes or names, followed by the use of the Romanian version for ICD-10AM. A first problem is the incompleteness of the OMIM and the ICD codes in DBpedia. Another problem is the inconsistency with the referred classifications. For example, for *Acute myeloid leukemia*, DBpedia gives the OMIM code 602439, which in OMIM is moved to 601626.

With this analysis, the current solution for building the Romanian lexicon for diseases follows the next steps:

1. the labels of the diseases are obtained from Diseasome
2. the ICD code is searched in the Linked Data version of OMIM from Bio2RDF. The search is done either on the disease’s label or on the OMIM code extracted from the label (in case it is included). The used endpoint is http://cu.omim.bio2rdf.org/sparql.
3. the ICD code is searched in Orphanet, again based on the name or the OMIM code. In case it is found, the type of the mapping is extracted, too.
4. the Romanian terms are extracted from the Romanian classification ICD-10-AM based on the identified ICD code. If the ICD code was obtained at step 2, the mapping is considered exact. Otherwise, the type of Orphanet mapping is analysed, and in case it is an exact mapping, only the Romanian term is kept. In case the mapping is not exact, the initial English label is concatenated to the Romanian term. Table 5 includes some examples of obtained terms. For the disease with OMIM code 180920, Orphanet mentions two mappings to ICD classified disorders, both having the status of *not decided*.

### Table 5

<table>
<thead>
<tr>
<th>Label in Diseasome</th>
<th>ICD10</th>
<th>Romanian term</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta-ureidopropionase deficiency</td>
<td>E79.8</td>
<td>(eng)+alte tulburari de metabolismul purinelor si pirimidiniei</td>
</tr>
<tr>
<td>abetalipoproteinemia</td>
<td>E87.6</td>
<td>(eng)+deficit in lipoproteine</td>
</tr>
<tr>
<td>aplasia of lacrimal and salivary glands, 180920</td>
<td>Q38.4</td>
<td>malformatii congenitale ale glandelor si canalelor salivare</td>
</tr>
<tr>
<td>hepatocellular carcinoma</td>
<td>C22.0 (E)</td>
<td>carcinosul celulei hepatice</td>
</tr>
</tbody>
</table>

**Results**

The steps 2 and 3 are alternatives for finding the ICD code. None of them is complete, and their combination is also incomplete. From 4213 different Diseasome resources, using only Orphanet, 1210 Romanian terms were found, including all types of mappings, not only exact mappings. By using also OMIM Bio2RDF, the number increases to 1815 terms. It must be emphasized that the process of building the lexicon can be improved through i) a more detailed filtering by name of the disease in Orphanet, since for example, in Diseasome, *type ii* phrase is used and in Orphanet it appears as *type 2*; ii) the extensive use of synonyms of diseases from Orphanet. Finally, medical experience is needed to solve some mapping and to validate the resulted lexicon.

A question parsed and linearized with GFMed grammars that include the generated lexicon for romanian terms of diseases is shown in Figure 12.

5. Related work

With the recent boost in available linked data and ontologies, the interest in extending the lexical context of ontologies increased as well. A recent result in extending ontologies with richer lexical layer is the ontology-lexicon model *lemon* [10]. This model proposes design patterns for the most common lexicalizations of labels from ontologies. The model was used in a manual approach of building an ontology-derived lexicon for DBpedia [12]. The building process consists of creating descriptions of verbalizations for classes and properties from ontologies. A significant part of DBpedia
ontology was covered, 98% of classes and 20% of properties. Common with this approach, GFMed grammars are built manually, starting from the schemas of datasets to be queried. Patterns are identified in our approach starting from DL constructors, mainly restrictions on properties. The patterns are directly described in GF, based on our own SPARQL resource library. The GF functions correspond to DL constructors, facilitating their composition in similar way to DL.

Manual development of the grammars strongly restricts the scalability of our approach. The semantic of the aimed linked data, biomedical data from Diseasome, Sider and DrugBank, is narrower compared to DBpedia. Nevertheless, the required precision in tackling medical data, can be obtained with a manual approach. A very recent result in automatic derivation of lexicons in lemon format is described in [14].

SQUALL [5] is another controlled natural language that allows SPARQL queries and updates and it relies on Montague grammars. Unlike SQUALL, GFMed is appropriate for multilingual development due to the fact that is a controlled natural language built with GF. GF was previously used in multilingual systems for querying linked data in [4], [3].

An incremental built of queries is described in [17]. Relevant concepts and properties are identified at each step and the user can choose one. The use of the ontology is the shared point with our approach, but in our case the user does not interact with the ontology and it uses natural language to express his needs.

From a completely different perspective, [15] and [19] propose learning and pattern matching for querying linked data in natural language. Ambiguity and named entity recognition are not easy to tackle within these approaches, unlike the case of controlled natural languages. But their important advantage is scalability and domain independence. Nevertheless, if we consider Romanian instead of English as querying language, the limited resources existing for processing this language, hamper the application of many approaches based on learning and pattern matching. In the same time, they sustain approaches based on GF and its resources for Romanian.
6. Conclusion

A controlled natural language for querying biomedical linked data was introduced. The language is built within Grammatical Framework by following a methodology based on DL constructors. The functions defined in GFMed’s grammar are highly composable, due to their relation with DL constructors and the types of resources from the targeted datasets. A GF resource for SPARQL was introduced and used. The resulting language is able to cover questions over more datasets, complex questions with different linguistic structures, and questions that involve lists and free text. The steps followed in building the language are not specific to the biomedical area, therefore the method can be considered general. The manual character of the language building process claims for future work in the direction of deriving automatically the GF functions from ontologies extended with lexical layer.

The proposed system addresses also multilinguality. Romanian was investigated near English as natural languages for querying. In order to obtain Romanian terms for diseases from Diseasome, a method based on more international classifications was analyzed, employing mainly the resources which follow the principles of linked data.

References