Querying Semantic Data with Natural Language

Ernest Kirstein

May 2, 2015

Contents

1 Introduction **4**
1.1 The Semantic Web 4
1.2 DBPedia 5

2 Previous Work **7**
2.1 Other NLIs 7
2.2 Common Issues 8

3 Converting Natural Language Questions into SPARQL Queries **8**
3.1 Parsing 10
3.2 Named Entity Recognition 14
3.3 Resolution 15
3.4 Generation 17

4 Software Architecture **19**
4.1 SPARQL Generation 20

5 Resolving User Input Names to RDF Entities **23**
5.1 Name Standardization and Enumeration 23
5.2 Jaccard Index 24
5.2.1 Comparison with Levenshtein (Edit) Distance 25
5.2.2 Information Content Sensitive Jaccard Index 26
5.3 Ambiguity 28

6 Top-Down Parsing **28**
6.1 Introduction to Top-Down Parsing without Syntax Diagrams 29
6.2 Compiling a Context Free Grammar for Top Down Parsing . 33
6.3 Effect of Grammar Transformations on Parse Trees 34
6.3.1 Substitution 35
6.3.2 Direct Left Recursion Elimination 36

7 Future Work **38**

8 Conclusion **38**

Abstract

Searching through semantic data using SPARQL requires expert knowledge and thus remains out of reach for casual users. In this work, we explore the development of natural language interfaces for semantic data. This work looks at a number of approaches towards this goal as well as providing a new approach. Our system works like a natural language compiler and handles a limited set of natural language questions which, though narrow in scope, have complex semantics that other systems would be unable to handle.

1 Introduction

The semantic web, as envisioned by Tim Berners-Lee et al., is an extension of the internet which provides more structure to the vast chaos of data on the net. The goal has always been for the semantic web to become a means for artificially intelligent agents to share information and reasoning [9]. Unfortunately for us humans, the semantic web has proven rather difficult for casual users to interact with.

To unlock the power behind the semantic web, there needs to be a way for people to interface with it and ask questions about stored (or reasoned) information. SPARQL is a common query language for searching through data on the semantic web [4]. The main benefit using a formal query language is that it allows users to ask complex, exact questions. The main drawback to formal query languages is that they have a steep learning curve and are really only viable for expert users.

So the usefulness of semantic data is largely inaccessible to casual internet users, but it does not have to be. Natural languages (English, Spanish, etc.) are more than capable of expressing the same sorts of questions that SPARQL can ask. But parsing and interpreting natural language is a little bit beyond our current technology. Tools like Siri [7] have already begun to bridge that gap but there is a lot of room for improvement.

This thesis will discuss the various approaches that have been made to make semantic data accessible through natural language interfaces, the direction of ongoing research in the field, and our own contributions to the solution.

1.1 The Semantic Web

The definition of the semantic web is somewhat nebulous. Any semantic data or interface for that data might qualify as part of the semantic web. Semantic data is information which is structured in ways that allow artificial intelligence agents to understand the meaning behind the data - data which captures relationships and connections and not just numbers.

The most common type of semantic data is in the form of 'triples'. These are small pieces of information which relate a subject and object via a predicate. Triples store information like, "John is an author." In such a sentence, "John" would be the subject, "is a" the predicate, and "author" the object. Except, in the semantic web, each of those concepts would be represented by a URI (uniform resource identifier). So the triple would actually be represented by something like, "http://example.com/john rdf:type dbpdiaowl:writer".

More complex information can be represented by multiple triples. As one might imagine, this makes expressing information in semantic data a little tedious. So, more commonly, semantic data is stored in specialized documents which translate into triples.

RDF [5] is a semantic data framework which specifies some foundational URIs for triples as well as an XML format to more easily express those triples. This framework is used to describe object types (classes), properties of those types, properties of those properties, and further foundation information for a well-rounded grounding of more comprehensive semantic data. RDF documents are quite expressive in and of themselves. Some basic reasoning engines can be built just rooted in RDF statements which propagate information about classes and their properties. For example, "Dogs bark. Spot is a dog. Can spot bark?"

OWL [2] is an extension to the RDF framework which provides another layer of reasoning to the rigid class specifications of plain RDF. For example, "Y is the pet of X implies that X is the owner of Y." RDF may make the semantic web expressive, but OWL allows it to be intelligent.

Semantic data is hosted on the internet and this makes up a part of the semantic web. Interfaces to that data may are also a part of the web and many of these interfaces use SPARQL. As mentioned before, SPARQL is a structured query language for asking rigorously defined questions over a semantic data set. SPARQL queries set up constrains for triples or sets of triples to fit and an interface which executes SPARQL queries returns the values from triples which match those constrains. Some engines also incorporate reasoning using ontological and semantic knowledge - others simply look at existing triples and return results from imediately available triples.

1.2 DBPedia

The website DBPedia [16, 24] is a RDF data store containing information which has been curated from Wikipedia. It is a massive collection of structured data containing over four million ontology-classified objects in over one hundred different languages. It is one of the largest open-source RDF databases on the web.

The SPARQL queries in this work will target a portion of DBPedia's data pertaining to books and authors. There is no clear division between this set of data and the rest of the data store but that is just one more real-world complexity to hurdle. Any system interacting with RDF data should consider the data's naturally unbounded scope, even if the system itself is more constrained.

DBPedia has its own ontology and ways of relating objects to one another. This means that, although the general principals involved would be the same, it may be more difficult to create an interface for another data set than simply replacing the URIs from DBPedia with another database's URIs. This 'portability problem' is an issue that is an area of active research [21, 34], but is not a key focus of this work.

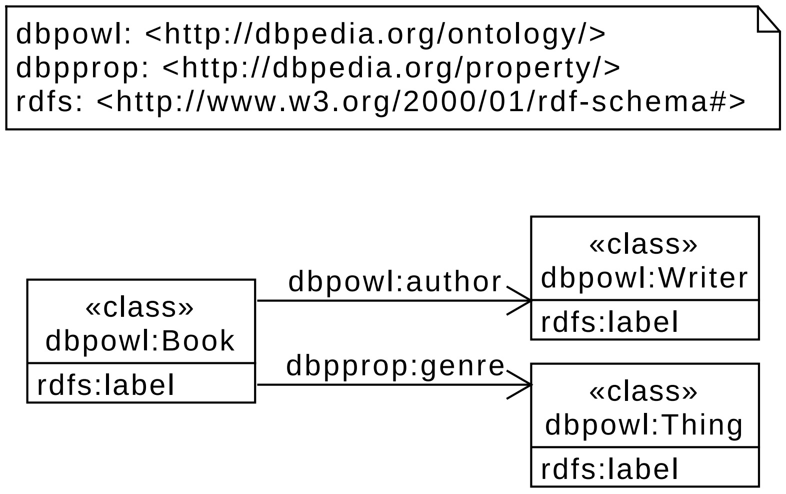

dbpowl: <http://dbpedia.org/ontology/>
dbpprop: <http://dbpedia.org/property/>
rdfs: <http://www.w3.org/2000/01/rdf-schema#>
«class»
dbpowl:Book
rdfs:label
dbpowl:author
«class»
dbpowl:Writer
rdfs:label
dbpprop:genre
«class»
dbpowl:Thing
rdfs:label

Figure 1: A few classes of the DBPedia ontology

Figure 1 shows a small portion of the DBPedia ontology that will be used in this work. The RDFS labels of each class are what plug into the natural language parser to recognize the names of books, authors, and genres. The classes are related by DBPedia's own property predicates.

Three classes of objects are enough to ask challenging questions but keep the scope within range of a research project. Our system can handle questions as complex as:

- What books has Heinlein written?
- Who wrote Fahrenheit 450? *(Handling typos; that's supposed to be 451)*

- Which authors have written both science fiction and high fantasy novels?
- Which books has Dave Wolverton written that are the same genre as Game of Thrones?

2 Previous Work

Work by Kaufmann and Bernstein [21] indicates that users have a "clear preference" for even limited natural language interfaces when compared to keyword or query language interfaces. Casual users of the semantic web have needs that require a more precise interface than the common keyword-based search engines of the early web. But they also need a less rigorous tool than a formal query language. [21]

Other recent works [15, 18–20, 22, 23] have shown that simple natural language questions can be translated based on known sentence structure or leveraging named-entity recognition, NER.

2.1 Other NLIs

Many natural language interfaces, NLIs, have been developed since they gained traction in the 70s [21]. This section will discuss a few of the more recent ones.

NLP-Reduce [21, 22] is a highly portable system which supports full natural language questions, fragment questions, and keyword questions. It leverages relationships between semantic classes while deliberately avoiding "complex linguistic and semantic technologies" [22]. Querix [23] is a more restricted NLI by some of the same developers as NLP-Reduce. It only handles specific question beginnings and specific question forms. This interface is more limited but was shown to be preferred buy casual users [21].

Another earlier system by Glaitsky [19] focused more on the logical complexity of some questions. His proposed approach used semantic headers to reason out complex questions that involved making "commonsense" deductions.

FREyA [15] is a system which focused on the portability problem of most NLIs. This system applied Stanford's part-of-speech parser and custom heuristic algorithms to classify groups of words into potential ontology concepts. Comparing an OWL ontology with the potential ontology concepts of the heuristic output, they were able to generate SPARQL queries.

The system even applied some machine learning techniques in the heuristic algorithm to achieve marginal improvements to question recognition and SPARQL generation.

2.2 Common Issues

A recent publication by Sharef et al. [34] outlines obstacles in developing full natural language interfaces for the semantic web. That paper notes a particular difficulty with parsing what we call "multifaceted" questions - questions with multiple variables, constraints, or operations. This is the precise gap which this work attempts to bridge.

NLIs for the semantic web will also eventually need to deal with multiple sources and unmitigated ontologies that need to be unified to have truly global search capabilities [34]. This might even include issues of trusting information sources [9].

Linguistic variability and ambiguity make building such a system a highly complicated and time consuming task. Also, domain-restricted NLIs are difficult to adapt and port: there is a trade off between retrieval performance and portability. Still, the meta-data of the semantic web provides assistance which can potentially overcome some of these problems. [21]

Expressive interfaces, even in natural language, can be overwhelming. As such, even NLIs require some user support and training (though much less than a formal query language would). [21]

3 Converting Natural Language Questions into SPARQL Queries

Since SPARQL is already an established tool for navigating the semantic web, it makes solid foundation to build higher level natural language queries. This system we have built receives natural language questions such as, "What books has Dave Wolverton written that are the same genre as Game of Thrones?" That question is parsed, related back to a custom indexing of DBPedia URIs, then a SPARQL query gets compiled and executed.

In short, our approach was to build a natural language compiler of sorts. Something robust enough to handle real-world natural language usage but rigid enough to translate into SPARQL queries. So our research touches a little more on compiler design than one would typically see in natural language processing research. The two areas of research have significant overlap [12, 31] and other work has suggested that that a middle ground between these two research angles might be where NLIs find the most success [21].

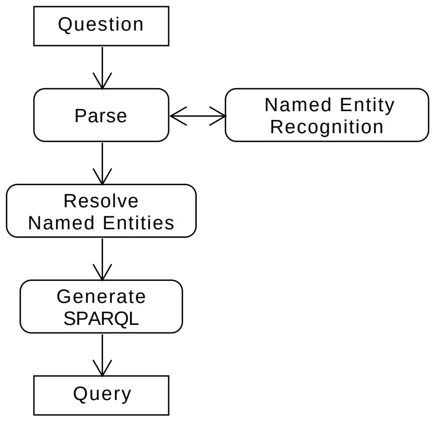


Figure 2: An overview of the NL to SPARQL Process

Figure 2 shows a brief outline of the full process. Questions come in the form of natural language questions and are fed into a parser. The parser is informed by the named entity recognizer which provides recognition of named entities (primarily nouns and adjectives, though any class of words might be a named entity type) whose names must be scraped from the RDF database. The parser also leverages the python natural language toolkit [6] for access to synonyms and hyponyms. The parse tree (the output of the parser) then goes through a name/URI resolution process which matches the natural language names to their corresponding RDF URIs. Finally, the resolved parse tree goes into a SPARQL-generating process which outputs a query.

3.1 Parsing

This system uses a custom parser which has been implemented especially for this project. One of the contributions of this work is the unique implementation of this parser. Parsing is typically done around a *grammar* which specifies how words fit together and what is a valid statement within the context of a language. There are other models for the description of a language [12] but typically specifying a grammar like in figure 4 is one critical aspect to developing a language-processing system.

Certain parsers require their grammars to have certain properties. Those properties might be something of a hindrance to the development process, so we have sought a way to work round the problem. Our parser automatically changes a grammar to remove problems like left recursion and a few other hiccups which usually give top-down parsers trouble. But then the parser outputs parse trees under the original structure of the input grammar. This is accomplished by reversing the transformation required to remove left recursion (etc.) using the algorithms described in section 6.3. This achieves a big design goal - it is a huge step towards decoupling the design of the grammar from the rest of the system.

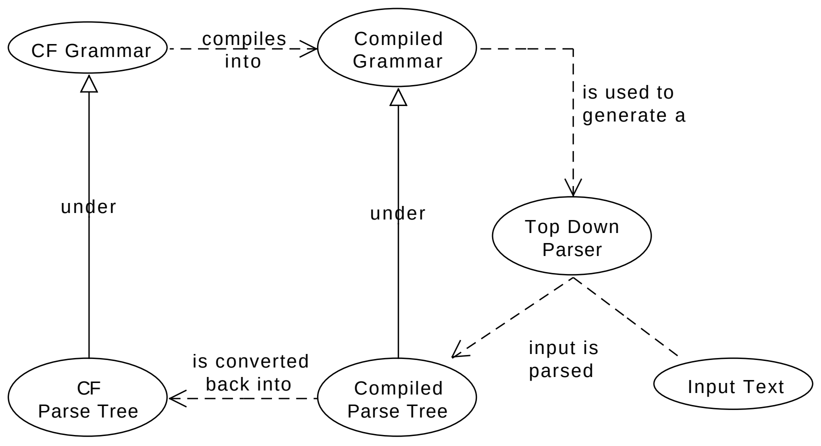


Figure 3: A high level overview of the parsing process.

The grammar is manually coded to handle a small set of natural language questions but is given more coverage by recognition of hyponyms and misspellings of named entities. The specification of the grammar is a design task which could be partially automated by using the ontology to enforce grammatical relationships between named entities. That was a bit beyond the scope of this project but it is an idea to keep in mind for further research.

Figure 4 describes the grammar which we used in our system to recognize a limited set of natural language sentences. The grammar is composed of *production rules* which describe how a string can be formed by replacing symbols within a string. Strings always start off as a starting symbol, in this case it is "*S*". From there, production rules are applied until a valid string is formed. If we follow rules 1, 3, and 9 we get the strings "*subject_question*",

$$S \rightarrow subject\_question \tag{1}$$

$$subject\_question \rightarrow books\_request \tag{2}$$

$$subject\_question \rightarrow authors\_request \tag{3}$$

$$books\_request \rightarrow \text{WHAT } general\_books \text{ HAS author WROTE} \tag{4}$$

$$books\_request \rightarrow \text{WHAT BOOKS HAS author WROTE THAT } book\_info \tag{5}$$

$$authors\_request \rightarrow \text{WHAT } general\_authors \text{ HAS WROTE } general\_books \tag{6}$$

$$authors\_request \rightarrow \text{WHO HAS WROTE book} \tag{7}$$

$$authors\_request \rightarrow \text{WHAT AUTHOR WROTE book} \tag{8}$$

$$authors\_request \rightarrow \text{WHO WROTE book} \tag{9}$$

$$general\_books \rightarrow \text{BOOKS} \tag{10}$$

$$general\_books \rightarrow \text{genre} \tag{11}$$

$$general\_books \rightarrow \text{genre BOOKS} \tag{12}$$

$$general\_books \rightarrow \text{genre AND genre} \tag{13}$$

$$general\_books \rightarrow \text{genre AND genre BOOKS} \tag{14}$$

$$general\_authors \rightarrow \text{AUTHOR} \tag{15}$$

$$general\_authors \rightarrow \text{genre AUTHOR} \tag{16}$$

$$book\_info \rightarrow \text{ARE THE SAME GENRE AS book} \tag{17}$$

Figure 4: Grammar

"*authors_request*", then "WHO WROTE book". This is a sentence constructed from abstract *terminal nodes* which fit to a standardized word or words. In this grammar we used upper case words to represent words or sets of words that are not associated with RDF entities. Lower case words represent RDF entities. So our constructed sentence might match the real question, "Who wrote The Giver?"

The terminal nodes for some words also leverage the python NLTK [6] to gain access to synonyms, hyponyms, and different tenses/pluralities of the word; for instance "WROTE" actually matches a lot of words; "written", "penned", "authored", "scrawled". Other terminal nodes simply include a fixed set of words that were manually entered. Since these terminal nodes are not directly paired to individual words it may appear that the grammar has some gramatical errors, but in fact the grammar matches a number of words which are gramatically correct and incorrect (you can't count on users to use perfect English). "Who has wrote X," and "Who done written X" are both matched in this grammar.

For another example, if we followed the rules 1, 2, and 4 then our strings would progress as follows: "*S*", "*subject_question*", "*books_request*", then "WHAT *general_books* HAS author WROTE". Rule 1 told us to replace *S* with *subject_question*. Rule 2 told us to replace *subject_questions* with *books_request*. Nodes that get replaced like this are called 'non-terminal'. We can apply another rule for the remaining non-terminal: *general_books*. If we chose the rule 10 then we end up with the string, "WHAT BOOKS HAS author WROTE". This question might match the real natural language question, "What novels has Heinline written?" Notice that, in the grammar, non-terminal nodes are in italics.

Grammar rules are specified for each type of question and each descriptive sentence fragment which corresponds to some SPARQL generation abstraction (see figure 4). In our example grammar, you'll see "general books" as a non-terminal grammar node which corresponds to phrases like "science fiction novels", "history and fantasy books", etc. Choosing how to design the grammar is not an easy task and required several iterations before some design principals became apparent. For more on this, see section 3.4.

A parse tree is a hierarchical representation of the application of a grammar's rules. For example, the parse tree in 5 shows the how the application of rules 1, 2, 5, and 17 are applied to build the string, "WHAT BOOKS HAS author WROTE THAT ARE THE SAME GENRE AS book". The parse takes natural language sentences like, "Which books has Dave Wolverton written that are the same genre as Game of Thrones?" and determines the appropriate parse tree which corresponds to it.

Given a natural language question, the parser actually produces a number of parse trees. Each parse tree can be considered an interpretation of the question, so the parse then selects the best (most probable) interpretation. The probability of the interpretation depends on how close the named entities found in the interpretations match the model defined for that entity (see section 3.2). The probability of the interpretation is estimated by taking the product of the estimated probabilities of all the named entities (again, see section 3.2 for more details). The interpretation goes through the query generation process to see if there are any logic conflicts that the grammar cannot see. If an error is thrown generating the SPARQL, the next best interpretation is used.

The example question's parse tree in figure 5 shows the parse trees nodes

```
S
└── subject_questions
    └── books_request
        ├── WHAT
        ├── BOOKS
        ├── HAS
        ├── author:  Dave Wolverton, 0.408655
        ├── WROTE
        ├── THAT
        └── book_info
            ├── ARE
            ├── THE
            ├── SAME
            ├── GENRE
            ├── AS
            └── book:  Game of Thrones, 0.928662
```

Figure 5: Parse Tree - the Output of the parser for the question, "Which books has Dave Wolverton written that are the same genre as Game of Thrones?"

in a tabbed hierarchy. On the left are the names of the nodes and to the right of each name you'll see some representation from the sentence that corresponds with the node. Named entity nodes also have a number which is the *information content* of that node (an inverse log of the probability) which is higher the further away a string is from the expected average in that named entity class. Both of the information content values for the named entities in figure 5 are quite good (low values) and indicate a strong match. This value is explained in more detail in section 3.2.

3.2 Named Entity Recognition

Natural language names of *named objects* are recognized by a component of the system called the *named entity recognizer* [30, 32]. The NER would be rather poor at predicting the type associated with a given name if that were the only information it had to go on. That is why each named entity type is associated with a terminal node in the grammar - it gains a great deal of accuracy by narrowing down the possible type for a given string using the context the grammar provides. Setting up the NER requires a bit of preprocessing: lists of natural language names are downloaded from the semantic database using manually entered SPARQL queries then compiled into classifier models.

Typically, NERs use some means of machine learning to classify objects within a sentence based on parts of speech, word or phrase feature extraction, and other rather advanced topics [30]. For our work, the definition of classes for our named entities was determined by the DBPedia ontology. Also, having access to the data, we also have a rather large sample set of names for each classification.

So for each class we built bloom filters [10] and n-gram classification models which could give a simple probability measure of, "Is X string Y classification?". Our n-gram classification models capture features like common prefixes and suffixes while bloom filters trained on a range of single-edit variations of a name help to catch non-standard names. Other more accurate models exist, but these served our purposes and had certain properties that were important within the context of the parser.

If the bloom filter does not classify a string as a named entity class then the string is checked against the n-gram classification model for that class. The n-gram classification models were rather simple to implement and worked well with the Jaccard index comparison we use in section 3.3. We simply checked for the absence or presence of n-grams as features and use a naive Bayesian classifier over that feature set [1] so the same features informed both the NER and the resolution process.

Bayesian classification involves calculating the probability of feature results (in this case the absence or presence of an n-gram in a string) then using the product of all those probabilities as an estimation of the total probability of that string occurring in a given set assuming that all features are independent. This is a pretty bold assumption and doesn't generally hold true for an n-gram feature set, but it gives a rough estimation which is all that was necessary in this case.

These models need to allow for probabilistic recognition of the input strings in below $O(n)$ time (in relation to the number of objects in the RDF database) to be done efficiently within the context of the parser. Additionally, they need to recognize variations on the names (typos, variations, etc.) within some user-defined as a standard feature that casual users expect from a search engine.

Furthermore, these models need to provide faster than $O(n)$ approximation of their probability within the set of all the names of instances of that class. This is important for determining which interpretation of a question is the correct one: we can generally assume that the most probable interpretation is the correct one. Exceptions of that rule are where there exist logical ambiguities rather than semantic ones ("How big is New York?" is a common example of an ambiguous question - which New York and how is "big" defined?).

This probability is expressed as *information content* [33] which is the value that you'll see in figure 5. Information content is a function which is useful for expressing very small probabilities, there was no other significant reason for expressing probabilities in this form. Suffice it to say that larger values for information content mean smaller probabilities and ranges from zero to positive infinity. Where $I(x)$ is the information content in an event x and $P(x)$ is the probability:

$$I(x) = -\log(P(x))$$

3.3 Resolution

After parsing, specific instances of named entities need to be *resolved* to corresponding RDF entities. Natural language names (like "John Smith") are mapped to RDF URIs (like "http://sbc.net/smith394") in a semi-automated process. This boils down to searching a database of names with a good fuzzy string matching algorithm and falling back on the user to select the appropriate name when there is no obvious best match.

The resolution process differs from NER name matching in that the NER only determines whether or not a name belongs to a certain type while the resolver actually figures out which URI goes with that name. For instance, the NER determines if "John Smith" is an author, but the resolution process will match that name to a URI like 'http://example.org/john_smith'. The NER can determine weather a name belongs to a set quicker than the resolution process can determine exactly which URI matches a name and is therefore able to be used in the parser. Using the resolver in the parser directly would be too slow.

A lot of work went into selecting the best fuzzy matching algorithm for natural language names. Though Levenshtein's *edit distance* metric is a good comparison function for spelling mistakes, we've found that a Jaccard index based function is more suited to the task.

Figure 6 shows a case where semi-automatic reasoning is applied. For the first named entity, "science fiction" is automatically paired with the named entity 'dbp:Science_fiction' because its RDF label, 'Science Fiction', is considered an exact match using our string comparison function. Notice that it is a relatively poor named-entity match within the statistical model used for parsing since it has an information content of 4.6. It may seem odd

```
S
└── subject_questions
    └── books_request
        ├── WHAT
        ├── general_books
        │   ├── genre:  science fiction, 4.602974
        │   └── BOOKS
        ├── HAS
        ├── author:  E. A. Poe, 2.184804
        └── WROTE
```

Figure 6: The parse tree for the question, "What science fiction books has E. A. Poe written?" There are two named entities in this parse tree, the genre and the author.

that this is a poor match even though it is an exact match using our string comparrison function: this happens because the set of possible grammars is small so it is not an ideal candidate for our name classification method which does best with large sets of names. The name "E. A. Poe" is another mater. It is very close to two author names using the Jaccard index comparison function; both "Edgar Alen Poe" and "Michael Poe" are equally close matches by this metric so the user is prompted to choose between them.

3.4 Generation

The last step is to actually produce SPARQL queries from the resolved parse trees. Being the last step in a complex process, it is tightly coupled to the output of the previous steps which means that this part needs to be particularly adaptable to design changes. Minor changes to the grammar, adding or removing named entity types, and any otherwise insignificant change trickles down to cause problems in this step. However, it is difficult to design this generation process due, in part, to the complexity of the SPARQL query language. So, it took several iterations to refine our approach which are described in more detail in section 4.1.

You can see in figure 7 an example of the final output from the SPARQL generation process. In this instance, three generation rules were applied to the parse tree during the generation phase; one which setup the query for being a DBPedia select query (configuring namespaces mostly); one which configured the query to be a request for books (selecting outputs and specifying key variables); and the last to link the subject variable to the general

```
S
└─ subject_questions
   └─ books_request
      ├─ WHAT
      ├─ general_books
      │  ├─ genre:  science fiction, 4.602974
      │  └─ BOOKS
      ├─ HAS
      ├─ author:  George Martin, 0.488245
      └─ WROTE
```

```
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX dbp: <http://dbpedia.org/resource/>
PREFIX dbpowl: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?book ?title ?genre
WHERE {
    ?book rdfs:label ?title;
        dbpprop:genre ?genre;
        dbpowl:author dbp:George_R._R._Martin;
        dbpprop:genre dbp:Science_fiction.
    FILTER langMatches( lang(?title), "EN" ).
}
LIMIT 100
```

Figure 7: An example of SPARQL being generated from a parse tree.

books adjectives (specifying the books grammar).

4 Software Architecture

This architecture is aimed at handling complex questions in a narrow domain. It does more than named entity recognition - it actually considers the full syntax of the language and processes natural language questions much like a compiler might process source code. It uses top-down parsing to generate a parse tree for the input question then compiles SPARQL queries from that parse tree. This section will explain the design of the system at the highest level.

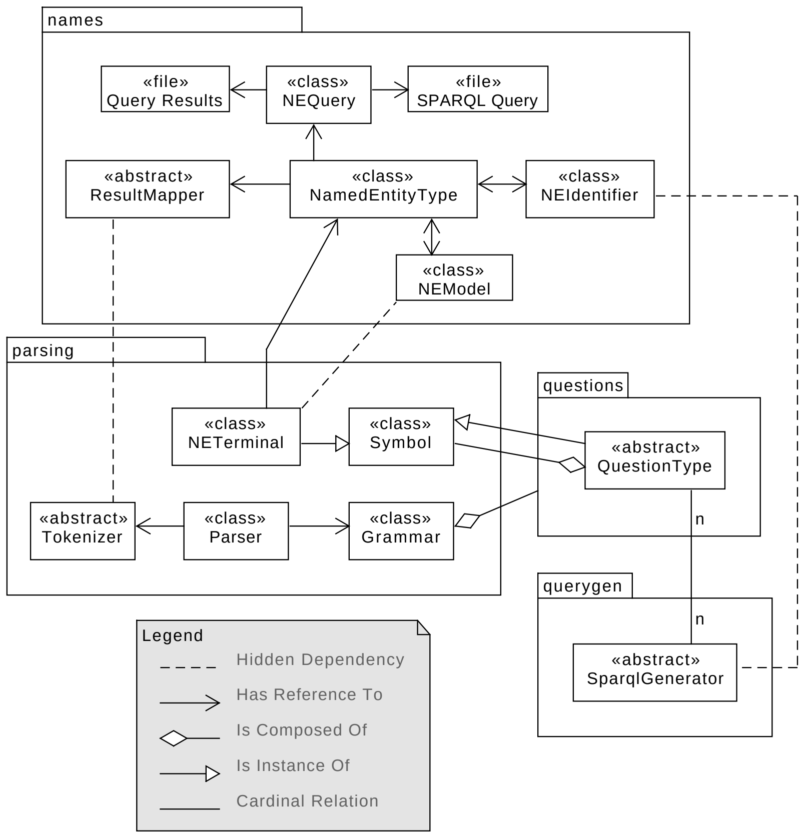


Figure 8: Architecture

One of the goals of this project was to develop a highly extensible architecture. There wasn't enough time to cover the breadth of questions one would hope for in a mature product - such being the nature of research.

But we felt it was important to create a practical working system that could handle a wide range of questions if more time was dedicated towards that end.

To make the system extensible, it was important to decouple the various components as much as possible. Still - language processing is a naturally interdependent process with many overlapping concerns. For instance, parsing seems to be an independent problem from named entity recognition (NER), but to achieve better parsing results it was necessary to integrate NER into the parser so that semantic context could inform the NER.

Another design concern was the accessibility of the SPARQL endpoint (an online interface where a remote server handles SPARQL queries over an RDF database). SPARQL endpoints may connect to a large number of RDF databases (each of which is a potential point of failure) and our experience has shown that they tend to go down more often than other online resources. It would be ideal if the endpoint was always accessible, would accept an unlimited number of queries, and would always respond quickly. But in practice, none of those ideals are true. As such, the results of certain queries (necessary for developing NER models and URI resolution) are cached into local files. The trade-off is that data in the cache can stagnate. That problem is mitigated by simply flushing the cached results periodically.

4.1 SPARQL Generation

SPARQL is a complex query language with a long, well-documented technical specification [3,4]. Generating SPARQL queries can be more complicated than simply fitting URIs into a templated string which was our first approach to the problem. In fact, it took us several iterations to realize some of the underlying design goals when designing the SPARQL generation framework.

The first (inelegant) solution was to create a separate query-generating classes that would handle each type of question on a case-by-case basis. These classes did little more than fill in templated strings with URIs and did nothing to capture the malleability of natural language. Trying to extend the types of questions using that architecture, we found that this approach wasn't feasible for a larger application. Output from this approach was ugly but workable for simple questions (see the appendix for an example of this old output).

In the next iteration, we improved the adaptability of the process by decoupling the query specification from the string generation. That is, we built a module to encapsulate the structure and output of SPARQL queries with functions for common query-building tasks (adding triples, applying filters, etc.). Figure 9 shows the design that came from actually reading through the dense technical documentation and understanding the underlying grammar of the SPARQL language.

In that iteration, it took many more lines of code to specify the queries since we were explicitly constructing queries rather than relying on manually entered strings. But at the same time, it became less error-prone, more easily readable, and produced cleaner output.

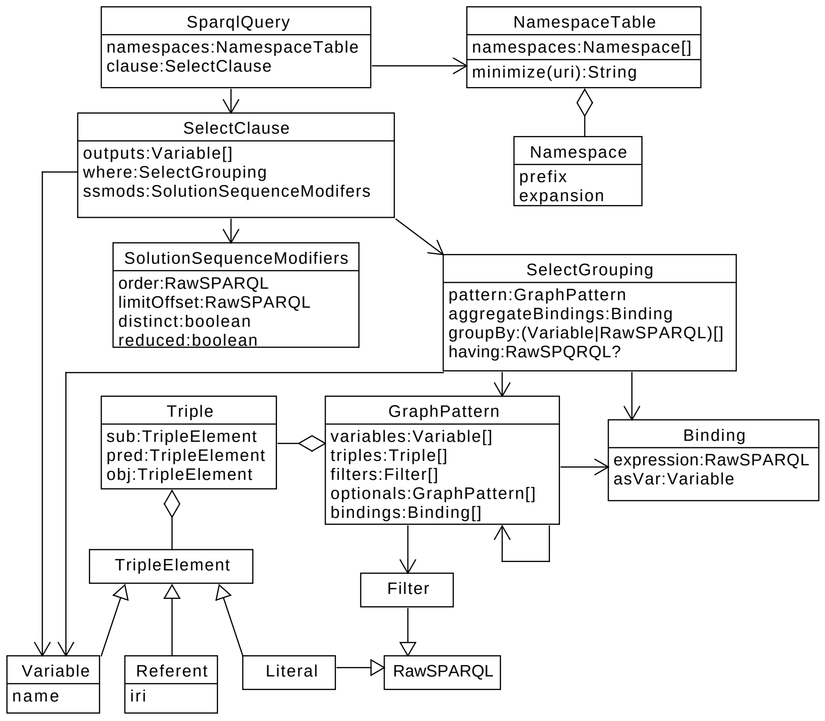


Figure 9: SPARQL Generation [3, 4]

Query generation is a source of bugs that can be prevented by a well designed framework. Every manually entered string is a potential source of bugs, the architecture should reduce the need for manually inputting strings. Since SPARQL is complex in and of itself, these manually entered strings are also a potential source of logic errors. Though some logic errors are impossible to prevent using anything short of a strong-AI, the proposed framework rigidly defines logical SPARQL code structures which will aid in producing SPARQL queries with as few bugs as possible.

Another reason to develop a query-generating framework is that it helps to develop a more modular system. Putting strings together is messy. Using a template system to fill in URIs means that there is at least one step in the process where every part is coupled to every other part: the point where the values are injected into the template. That makes it hard to design a modular system to interpret the parse trees which come out of the natural language parser. And it *will* be necessary to subdivide that process in a larger, real-world system. With the building encapsulated in a separate module, functions can be coded which more deftly manipulate the query rather than just plugging in values. It will aid the development processes in much the same way that automated-refactoring tools help coders produce code manually.

In the final iteration, we tried to improve the flexibility of the query-generating code by expressing the procedure in terms of re-usable functions. Nodes from the parse tree are iterated over in a depth-first algorithm which effects changes to the query at only key nodes such as particular noun phrase nodes.

During that final iteration, it became clear that some grammars were much easier to parse than others. Grammars which used typical grammar-school abstractions for symbols (nouns, verbs, noun phrases, etc.) were easier to specify but were very slow to parse (see figure 22 in the appendix). Such grammars let in extraneous interpretations that (while grammatically correct) bungle the named-entity-recognizer due to the number of possibilities which need to be ruled out. For example, "What scify books has john wane written?" is rather easy to interpret because we can immediately classify "scify" as a genre and 'john wane' as an author's name. It is easy to underestimate how much of that is from context; "scify" could be the name of an author or "john wane" the name of a biography.

The NER's models are lax on purpose to account for these variabilities and even names which might not be in the model's sample set: it will generally accept strings as names more often than it will reject them so it has to check most words belonging to each type of noun (where the grammar allows). Then it needs to check many possible divisions between words and even needs to consider misspellings. All of this is handled more efficiently if one simply specifies the grammar to restrict the possible types within the grammar rather than relying on the NER completely. The grammar in figure 4 is an example where this design principal was applied.

5 Resolving User Input Names to RDF Entities

Matching natural language names to RDF entities is essential to evaluating natural language questions over RDF databases. This poses several problems: names can be misspelled (e.g. "Swartseneger" for the label "Schwarzenegger"), may be reordered (e.g. "John Smith" for the label "Smith, John"), or the name may be abbreviated (e.g. "R.L. Stine" for the label "Robert Lawrence Stine"). That's not to mention the small problem posed by people who have changed their name entirely, sometimes multiple times (e.g. "The artist formerly known as 'The artist formerly known as Prince'"). And despite all of these convolutions, a natural language system will need to recognize and match these arbitrary instances with the often-sparse naming information present in RDF data.

Our approach to this problem was to find a string distance function which was robust to these changes and use that to simply find the best match name in $O(n)$ time (where n is the number of names). Since the names are narrowed down by context (using the parser), this was an acceptable solution: we don't need to iterate over every single label in the RDF set, just those which belong to the particular type of object the user is asking about.

There are a surplus of string comparison functions to choose from [14]. For our particular application, we have used a variation on the *Jaccard index*. It is robust to misspellings and reorderings, with the added benefit of being quite efficient. The Jaccard index is used in data mining for efficiently comparing long documents, but it is comparable to other more complex methods of name comparison [14] and our experience suggests it will work well here.

Another problem is resolving multiple close names (or even exactly the same name) to a single entity. I took a simplistic approach (just asking the user) but we will also discuss other more sophisticated possibilities for further research.

5.1 Name Standardization and Enumeration

Before jumping into the specifics of the name comparison algorithm, there are a few issues to deal with. Names with abbreviations, punctuation, and names with multiple parts can all trip up comparison algorithms.

One standardization method we used was to remove punctuation and change all letters to upper case. There may be a few edge cases where "John O'neal" isn't the same as "John Oneal", but the mistake is acceptable the majority of the time. And such names are so close that they would trigger the system to prompt the user for confirmation anyways.

Names with multiple parts, like “John Jacob Jingleheimer Schmidt”, need to be matched by partial variation like “Schmidt”, “Mr. Schmidt”, and “John Schmidt”. Using a sufficiently robust string comparison function, these variations will often still match the full, multi-part name better than other multi-part names. But that’s a dubious assumption to rely upon - it is best to tokenize the name and include the different partial variations as other names linked with the entity. As part of our own system, we simply included the first and last names as variations on the name, excluding the middle name(s).

5.2 Jaccard Index

The Jaccard index is a measure of how similar two sets are to each other. This is useful in a whole host of applications [11], as you might imagine. In this application, using the Jaccard index on fragments of strings yields a very robust string comparison function.

Let A and B be sets or multisets, then the Jaccard index $J(A, B)$ is defined [14, 29]:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Or if both sets are empty, $J(A, B) = 1$. For comparing strings, the sets might be of characters (e.g. 'HELLO' $\mapsto \{E, H, L, O\}$), of tokens (e.g. 'JOHN H SMITH' $\mapsto$ {'JOHN', 'H', 'SMITH'}), or in this case, *n-grams.*

These n-grams (also known as 'k-grams' or 'shingles' [29]) are all unbroken substrings of length n of a given string. So the 3-grams of the string 'anabasis' are {'ana', 'nab', 'aba', 'bas', 'asi', 'sis'}.

In this application, n-grams were constructed to include imaginary pre and post string characters (represented here as '^' and '$' respectively). So, for instance, the 3-grams of 'cat' are then {'^^c', '^ca', 'cat', 'at$', 't$$'}. This gives significance to the beginning and end of a string when using the Jaccard index with the n-grams. We use enough prefix and suffix character so that n-grams start with or end with a single character in the string; this gives a bit of additional weight and focus to prefixes and suffixes in the comparison function.

For a distance function, one can use:

$$d(s_1, s_2) = 1 - J(ngrams(s_1), ngrams(s_2))$$

The Jaccard distance metric, d, is a metric space for ngrams [27]. But it doesn't quite produce a true metric space for strings (since two different strings can have exactly the same n-grams). Still, it does satisfy the triangle inequality [11] and is always non-negative. So one could conceivably construct an M-tree [13] (a datastructure which requires these properties) to improve look-up speed. That hasn't been necessary in this research, but then again, this domain might be smaller than one might experience in practice.

5.2.1 Comparison with Levenshtein (Edit) Distance

Levenshtein distance (also known as 'edit distance') is a more common fuzzy string comparison algorithm than the Jaccard index. Its ubiquity might be due to a simple happenstance: the algorithm for calculating edit distance is a favorite example in algorithm design courses for demonstrating dynamic programming. But its popularity is by no means a guarantee that it is the best choice, as I'll demonstrate.

Levenshtein distance is defined as [28] the minimum number of *edits* (additions, deletions, or swaps) that must occur before one string matches another. But these are only single-character edits, and so the algorithm doesn't handle large displacements of parts of the name with any sort of grace. This is best demonstrated by example; see figure 11.

Levenshtein Matching Results

Distance	String A	String B
14	'global kirstein investing'	'kirstein global investing'
13	'global kirstein investing'	'scherl global investing'
7	'kirstein global investing'	'scherl global investing'

Jaccard (3-gram) Matching Results

Distance	String A	String B
0.3125	'global kirstein investing'	'kirstein global investing'
0.5946	'global kirstein investing'	'scherl global investing'
0.5143	'kirstein global investing'	'scherl global investing'

Figure 10: Bad Levenshtein Name Matching

Using Levenshtein distance to match closest strings, word inversions would be considered bulk deletions and insertions. The Jaccard index handles this better because inverting whole words still preserves the ngrams within those words even if it breaks the joining ngrams between the words.

Admittedly, the Jaccard index is less forgiving of small typos. A single character edit breaks n n-grams. So, for short single word names, Levenshtein distance is probably the preferred metric.

5.2.2 Information Content Sensitive Jaccard Index

The Jaccard index by itself is a fairly good way to compare names. But it would be better if the algorithm also noticed things like how unusual certain name patterns were. “Tom Smith” might be closer to “John Smith” than “Tom” based purely on their Jaccard index, but any reasonable person would pick “Tom Smith” and “Tom” to be the closer names because “Smith” isn't the discriminating factor.

In more technical terms, the part of the string “Smith” should receive less weight in the comparison function because it conveys less information.

Consider the more general form of the Jaccard index [11]:

$$J(\vec{x}, \vec{y}) = \frac{\sum_i \min(x_i, y_i)}{\sum_i \max(x_i, y_i)}$$

Where $\vec{x}$ and $\vec{y}$ are large dimensional vectors of real numbers rather than sets. To help relate it back to the original form, imagine that $\vec{x}$ and $\vec{y}$ are lists of counts of all the possible things that could be in the two sets (or multisets) A and B.

$$\begin{aligned} x_i &= \mathit{count}(e_i; A) \\ e_i &\in \{A \cup B\} \end{aligned}$$

Where y_i is defined similarly for B. Now if we want to consider the amount of information each little portion of the string contains, we can calculate it using the equation laid down by Shannon [33]. Namely, that the self information of an event is the log of the inverse of the probability that event occurring. In this case, the event is the occurrence of a certain n-gram in the string.

$$\begin{aligned} I(e_i) &= \log\left(\frac{1}{P(e_i)}\right) \\ &= -\log(P(e_i)) \end{aligned}$$

To determine this probability, we can look at the frequency of the n-gram in a large sample set of names. We need to index those names anyways, as part of the named entity recognition process. With a large set, we can get pretty close to a true approximation of the probability of an n-gram occurring in a general population of those names:

$$P(e_i) \approx \frac{count(e_i; N) + 1}{|N| + 2}$$

Where N is the multiset containing the union of all the n-grams of all the names. Then, we can weigh the vectors x and y such that we have an information sensitive comparison function: $\hat{J}(x, y)$:

$$\begin{aligned}
\hat{J}(\vec{x}, \vec{y}) &= \frac{\sum_i I(e_i) \min(x_i, y_i)}{\sum_i I(e_i) \max(x_i, y_i)} \\
&= \frac{\sum_i \min(x_i I(e_i), y_i I(e_i))}{\sum_i \max(x_i I(e_i), y_i I(e_i))} \\
&= J\left(\vec{x} \cdot \vec{I}, \vec{y} \cdot \vec{I}\right) \\
\vec{I} &= \langle I(e_1), I(e_2), ...I(e_n)\rangle
\end{aligned}$$

Matching Results without Information Content

Distance	String A	String B
0.6471	‘tom smith’	‘john smith’
0.7692	‘tom smith’	‘tom’
1.0	‘john smith’	‘tom’

Sample Set
‘adam smith’
‘bob smith’
‘carl smith’
‘dale jones’
‘ernest kirstein’

Matching Results with Information Content

Distance	String A	String B
0.7296	‘tom smith’	‘john smith’
0.7235	‘tom smith’	‘tom’
1.0	‘john smith’	‘tom’

Figure 11: Information Sensitive Name Matching

The above example shows this comparison in action. In the top comparison, without considering information content, the closest pair of names is “tom smith” and “john smith”. But after indexing the sample names, we can calculate comparisons that do consider the information content. Then we can see that the closer names are “tom smith” and “tom”.

5.3 Ambiguity

What should the system do when the user inputs a name which is close to the names of several entities by the chosen name comparison function? The easiest solution would be to simply ask the user which of the possible entities they meant. But this isn't always a great solution; what if you're asking about a cornucopia of names? Sure, it might be out of scope for this particular system to resolve questions such as "Which of the actors in 'my_data_file.txt' have been in movies together?" But that's certainly within the realm of possibilities for some future work.

One approach to this problem would be to create a separate system for distinguishing the "right" name from a small(er) selection of possible candidates. The aforementioned Jaccard index (or similar distance metric) might be used to narrow down the problem space to something manageable, then a much more sophisticated (yet slower) system could choose the right name from the smaller set.

Since there are a number of string comparison algorithms, one could (if time permitted) implement several of them and use a combined metric to evaluate the names for fitness. For example, a feed-forward neural net could be trained to recognize the 'right' name based on a number of factors:

- String comparison functions (Jaccard, Levenshtein, etc.)
- The self-information [33] of the input and potential match names
- The closeness of other potential match names
- The prevalence of the potential match entity in the RDF data
- The frequency of queries involving the potential match entity
- Etc.

6 Top-Down Parsing

Historically, top-down parsers were been coded manually or else generated from a context free grammar specification. [25, 26] Coding a RDP (recursive decent parser) manually is tedious, error prone, and difficult to maintain. Programmatically generating a top-down parser from a grammar still isn't ideal - converting a context free grammar into a top-down parsable form may not preserve the *strong equivalence* of the grammar. Two grammars are weakly equivalent if they define the same language (set of possible strings).

Two grammars are strongly equivalent if they are weakly equivalent and there is a one to one correspondence between their parse trees which match equivalent strings. [31] And as a result, parse trees generated by that RDP will not be in the same form as they might appear in the initial (non-RD-parsable) grammar, which is often a more natural representation of the desired language [35].

In this work, we hope to describe a useful adaptation to top-down parsing which addresses these problems. Our system compiles a context free grammar specification into a top-down parsable grammar. A parser is generated from the top-down parsable grammar, and used to parse a input streams. The resulting parse trees are then transformed back into equivalent parse trees under the original grammar. (Figure 3)

6.1 Introduction to Top-Down Parsing without Syntax Diagrams

Introductory material by Dr. Lewis [25] describes a recursive-descent parser (a type of top-down parser) as a piece of software which takes a sentence and turns it into a parse tree by performing a *depth-first* search. But a search of what? One might try to call it a search of the parse tree, but that's not exactly right. The "recursive descent" name comes from the way that a RDP traverses through a syntax diagram of a context free grammar [35].

Recursive decent is just one form of top down parsing. The top-down parsing in this thesis does not use syntax diagrams (in hindsight, this was a questionable design choice, but such is life). This section will describe how top-down parsing works without using syntax diagrams.

Consider a context-free grammar with the following production rules:

$$S \rightarrow aS \tag{1}$$
$$S \rightarrow bS \tag{2}$$
$$S \rightarrow \epsilon \tag{3}$$

And the following string which we will attempt to parse: "ab"

We know, right off the bat, that the start symbol S will be the root of any parse tree created under this grammar, by virtue of it being the start symbol. (Figure 12)

The next parse tree we should consider is the parse tree that is generated when we follow the first production rule. (Figure 13) Notice that, in this instance, the parse tree does not conflict with the string we are trying to parse - i.e. regardless of the production rules we follow the string that is produced from any further production rules we follow will start with "a".

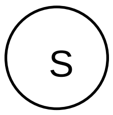


Figure 12: Parse Tree 0

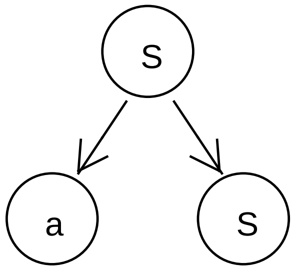


Figure 13: Parse Tree 1 - Valid

For the next parse tree, we will try to repeat are last action (following the first possible production rule). (Figure 14) This parse tree conflicts with the string we are trying to produce since any string produced by further production rule applications will produce a string starting with “aa”. In this case, we go back to the previous parse tree and try a different production rule.

In figure 15, we’ve followed the second production rule and our new parse tree fits with the input string, so we can continue our descent.

The next two parse trees (Figure 16), created by applying the first and second production rules to Parse Tree 3, are both invalid because they extend past the length of our input string.

In the last step, by applying the third production rule to Parse Tree 3, we have a parse tree which terminates and produces the desired input string, “ab”. (Figure 17)

Finally, let’s diagram our traversal through the possible parse trees (Figure 18). Shown this way, one can notice a pattern in our attempt to build the tree. The top-down parser performs a depth first search of the graph of possible parse trees, looking for a parse tree which fits the input string.

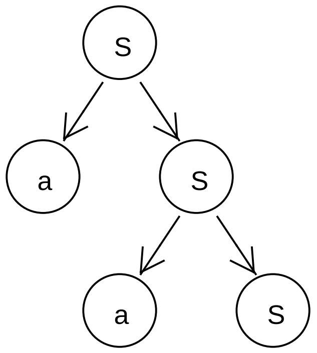


Figure 14: Parse Tree 2 - Invalid

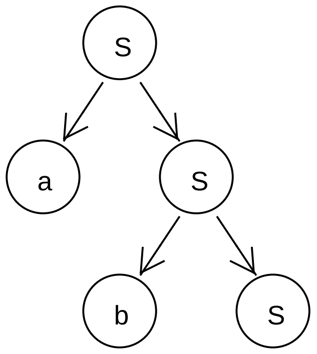


Figure 15: Parse Tree 3 - Valid

The child nodes from each PT (Parse Tree) node in the graph are PT nodes generated by applying each of the production rules to the first (left, deepest) nonterminal symbol in that PT node. The depth of each node in the PT tree corresponds with the number of production rules that have been applied.

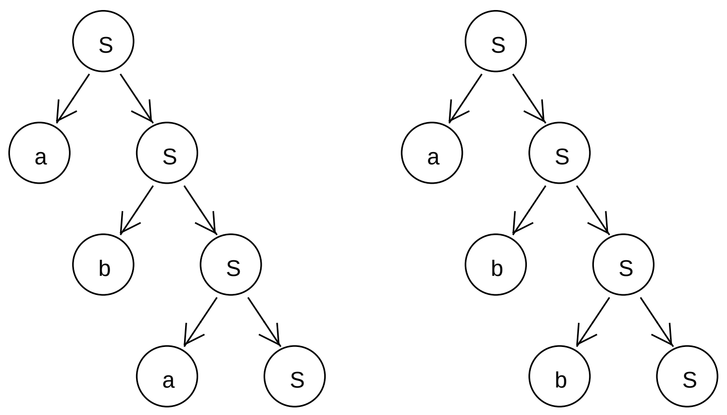


Figure 16: Parse Trees 4 and 5 - Both Invalid

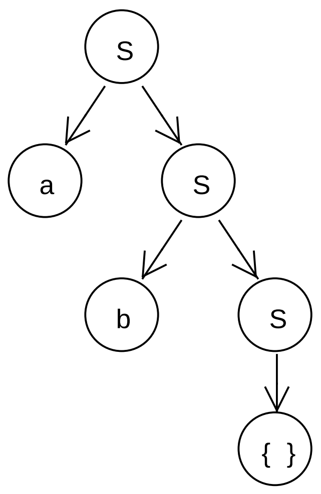


Figure 17: Parse Tree 6 - Complete

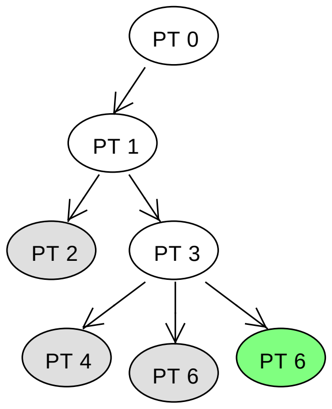


Figure 18: Parse Tree Search Progression

6.2 Compiling a Context Free Grammar for Top Down Parsing

It is important to notice that not all context free grammars can be directly parsed by a top-down parser. Some context free grammars require a bit of manipulation to remove left recursion (direct or otherwise) [35]. This process of converting a context free grammar into a *weakly equivalent* top-down parsable grammar shall be referred to as *compiling* the grammar.

A grammar is formally defined from an ordered collection of production rules. Our parser uses context-free grammar rules, which are comprised of a *head* (the single-symbol left hand side of the production rule), and a 'tail' (one or more symbols comprising the right hand side of the production rule).

These grammars may be compiled using the four procedures: factoring, substitution, removing left recursion, and removing useless rules. Let *decision list* define an ordered list of production rule choices which produces a parse tree. As each of these four procedures changes an input grammar into a weakly equivalent output grammar, there exists a mapping for any decision list in a compiled grammar back into a same-terminal-producing decision list in the pre-compiled (parent) grammar. Our parser keeps track of these inverse transformation rules as performs its compilation procedure so that a compiled grammar's decision list (the result of parsing a string) can be easily converted to the initial grammar's equivalent decision list.

Take this simple grammar for example:

$$S \to AB \tag{1}$$
$$A \to a \tag{2}$$
$$A \to SA \tag{3}$$
$$B \to b \tag{4}$$
$$B \to SB \tag{5}$$

It compiles into the weakly equivalent grammar:

$$Z \to \epsilon \tag{1}$$
$$B \to b \tag{2}$$
$$S \to aBS' \tag{3}$$
$$S' \to \epsilon \tag{4}$$
$$A \to aZ \tag{5}$$
$$B \to aBS'B \tag{6}$$
$$S' \to aZBS' \tag{7}$$
$$Z \to bS'A \tag{8}$$
$$Z \to aBS'BS'A \tag{9}$$

So when the terminal stream "aabb" is parsed in the compiled grammar to the decision list $[3, 6, 2, 4, 2, 4]$ it can be transformed into the parent-grammar-equivalent decision list: $[1, 2, 5, 1, 2, 4, 4]$.

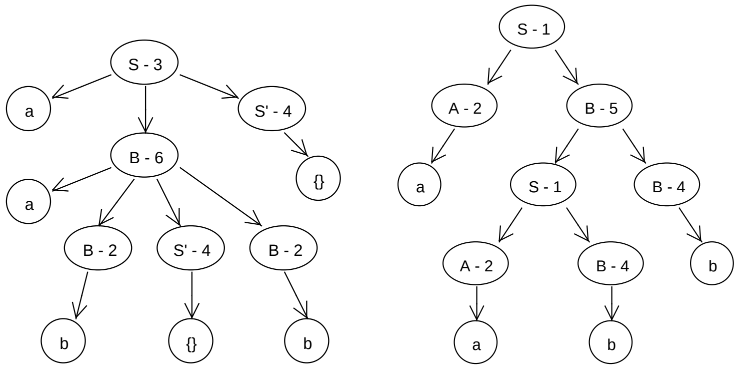


Figure 19: Compiled Grammar Parse Tree (left) Parent Grammar Parse Tree (right)

6.3 Effect of Grammar Transformations on Parse Trees

Left recursion is a problem for top-down parsers because it may cause them to go into an infinite loop. Using the model described in section 6.1: when a PT node is reach where the first nonterminal symbol has a production rule with left recursion, its child node will have the same nonterminal symbol so it will produce a child node with the same nonterminal symbol ad infinitum -

and none of those children will consume any terminals from the input stream so the parser will not proceed.

So, removing left recursion from context free grammars is a necessary evil for top-down parsing. It just takes two transformations to turn any context free grammar with left recursion into a *weakly equivalent* grammar with only right recursion. These two transformations are direct left recursion elimination, $DLRE(G; R_\alpha, R_\beta) \to G'$ and substitution, $SUB(G; R_\alpha, R_\beta) \to G'$, which is necessary to remove indirect left recursion. [8,25] These sections will describe how these transformations work on the grammar and on their parse trees.

6.3.1 Substitution

A grammar, G, where substitution can be applied has rules for some non-terminals A and B of the form:

$$R_\alpha = A \to B\alpha$$
$$R_\beta(i) = B \to \beta_i$$

Where $i \in [1, m]$. Let R_β represent the sets of the B rules where the notation $R_\beta(1)$ represents $B \to \beta_1$. Such a grammar represents a language which contains substrings of the form:

$$S = \beta_x \alpha$$

With $x \in [1, m]$. To produce such a substring, the rules need to be followed in the opposite order:

$$Prods(S; G) = \{R_\alpha, R_\beta(x)\}$$

Where $Prods$ is the function which outputs the production rules from G which generate the substring S.

The transformation [8] $SUB(G; R_\alpha, R_\beta) \to G'$ which applies the substitution replaces the single-element R_α rule set with combined rules for each R_β:

$$R_\alpha(i)' = A \to \beta_i \alpha$$

Then, to produce the same substring S, just a single new R_α rule is followed.

$$Prods(S; G') = \{R_\alpha(x)\}$$

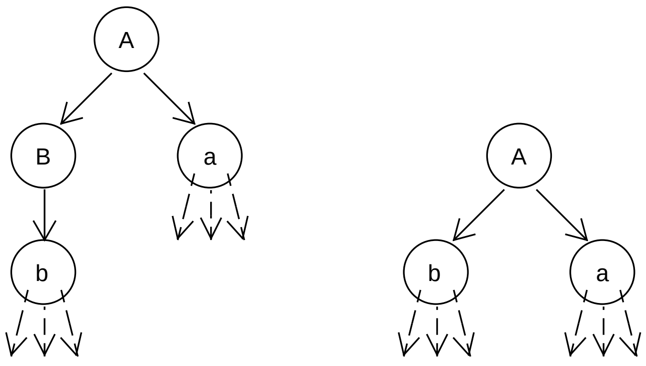


Figure 20: Original and Transformed Parse Trees

The transformation just removes the B node in the parse tree (Figure 21). So the inverse transformation simply puts the B node back in.

More formally, for each A node in the transform-affected parse tree, the inverse transformation $SUB^{-1}(T'; A, B) \to T$ modifies trees rooted at A nodes where the children of the A node are just some β_x and α. The β_x node will be replaced with a B node which then just connects back to the β_x node.

6.3.2 Direct Left Recursion Elimination

A grammar with direct left recursion, G, has rules for some nonterminal A of the form:

$$R_\alpha(i) = A \to A\alpha_i$$
$$R_\beta(j) = A \to \beta_j$$

Where $i \in [1, n]$ and $j \in [1, m]$. Let R_α and R_β represent the sets of those rules respectively where the notation $R_\alpha(1)$ represents $A \to A\alpha_1$. Such a grammar represents a language which contains substrings of the form:

$$S = \beta_y \alpha_{x_1} \alpha_{x_2} ... \alpha_{x_p} ... \alpha_{x_{k-1}} \alpha_{x_k}$$

where $y \in [1, m]$ and each $x_p \in [1, n]$. However, to produce such a substring, the α_* rules need to be followed in reverse order:

$$Prods(S; G) = \{R_\alpha(x_k), R_\alpha(x_{k-1}), ...R_\alpha(x_p), ...R_\alpha(x_2), R_\alpha(x_1), R_\beta(y)\}$$

Where $Prods$ is the function which outputs the production rules from G which generate the substring S.

The transformation [8] $DLRE(G; R_\alpha, R_\beta) \rightarrow G'$ which achieves Direct Left Recursion Elimination transforms each of those R_α and R_β rules to corresponding R'_α and R'_β rules respectively:

$$R'_\alpha(i) = A' \rightarrow \alpha_i A'$$
$$R'_\beta(j) = A \rightarrow \beta_j A'$$

and adds an additional rule:

$$R_\epsilon = A' \rightarrow \epsilon$$

Consequently, the order of the followed production rules in G' to produce the same substring uses rules in forward order:

$$Prods(S; G') = \{R'_\beta(y), R'_\alpha(x_1), R'_\alpha(x_2), ...R'_\alpha(x_{k-1}), R'_\alpha(x_k), R_\epsilon\}$$

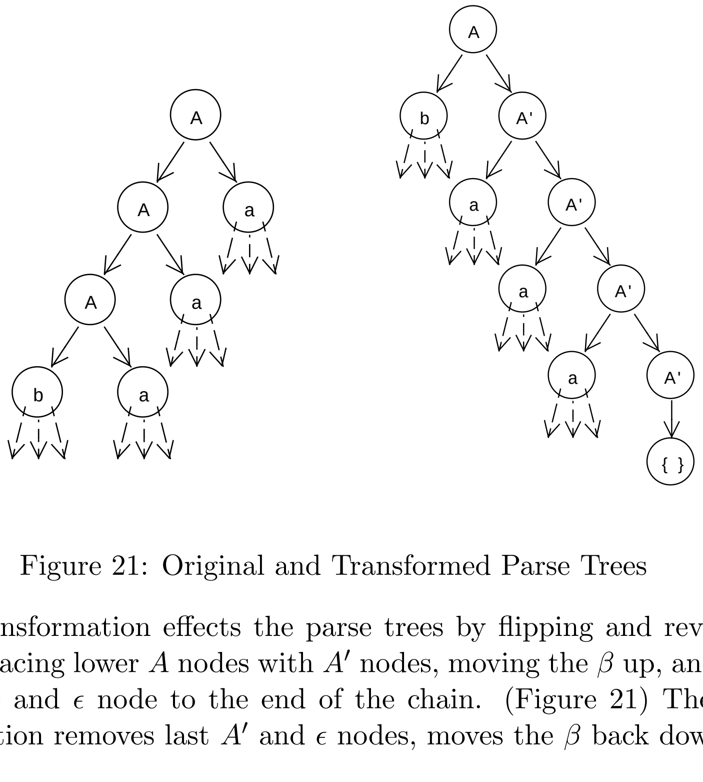


Figure 21: Original and Transformed Parse Trees

The transformation effects the parse trees by flipping and reversing A chains, replacing lower A nodes with A' nodes, moving the β up, and adding an A' node and ϵ node to the end of the chain. (Figure 21) The inverse transformation removes last A' and ϵ nodes, moves the β back down to the end of the chain, and changes the A' nodes back into A nodes, then flips and reverses the A chain. Note that the α and β nodes are arbitrary substrings, so in reality they might be multiple nodes which might have any number of children.

More formally, for each A node in the transform-affected parse tree, the inverse transformation $DLRE^{-1}(T'; A, A') \rightarrow T$ modifies chains of A/A' nodes where the children of the A node are C_A, of all but the last A'_{xp} node are $C_{A'_{xp}}$, and the last A'_ϵ node has only the child ϵ. From the $DLRE$ transformation rules we know that the C_A will be of the form $\beta_y A'_{x_1}$. We can also conclude from the $DLRE$ transformation that each $C_{A'_{xp}}$ will be of the form $\alpha_p A'_{x_{p+1}}$ when $p < k$ and A'_ϵ when $p = k$.

The inverse transformation first removes the A'_ϵ node. Then it changes each remaining A'_{x_p} node into an A_{x_p} node with the same children. Next the A_{x_p} are restructured such that each $C_{A_{xp}}$ is equal to $A_{x_{p-1}}\alpha_p$ where $1 < p$ and $\beta_y\alpha_k$ where $p = 1$. The top node of the chain, A, is replaced with A_{x_k}. And this process is repeated for each chain.

7 Future Work

As this project draws to a close, there have been a number of key areas where more work could have been done and where this work might lead. Recent works have directed their research towards the portability problem as a key area for research. The portability problem being: the ability to quickly generate or adapt a NLI to a particular semantic dataset or interface. One could say that, although this is obviously an important area to push forward, we should also focus on creating interfaces with a deeper understanding of their subject matter. NLIs which focus on niche subjects in great detail may have more potential to attract casual users to the semantic web than broader but less expressive NLIs which may only provide marginal benefits compared with the current internet.

8 Conclusion

The issue with natural language interfaces is the insurmountable scope of the problem. Natural language is too rich to ever be completely documented. Perhaps even more troubling, natural language is a moving target; always evolving along with society and the times. The lines between languages can be blurred. The lines between dialects even more so. And that leads to the obvious question: how can we ever hope to get there?

We do have a fairly solid framework for understanding languages as static constructs [12]. We're slowly accumulating knowledge in the semantic web like Tim Berners-Lee imagined, though not yet to the extent that he'd hoped. Perhaps it is naive of me to think so, but this seems like the kind of problem where progress will be exponential. When we finally do start making serious headway into creating natural language interfaces for the semantic web, we'll be able to communicate and accumulate knowledge more effectively which will increase the rate at which we'll be able to improve natural language interfaces ad infinitum.

Even just in this work we've demonstrated how one could build an in-depth natural language interface for a narrow domain - perhaps that narrow domain will be one of NLI development in later work.

Terms

- **Grammar**: a phrase-structure grammar is defined by a finite vocabulary (alphabet), a finite set of initial strings, and a finite set of rules... [12] (see Production Rule)
- **Context-Free Grammar**: a context free grammar is one which only has production rules whose head is a single nonterminal symbol. [17, 26, 35]
- **Production, Production Rule, Rewrite Rule**: rules of the form $X \to Y$ where X and Y are strings in a grammar (ordered lists of symbols of the grammar) [12]; define the nonterminal symbols by sequences of terminals and nonterminal symbols [35]; rules which specify how nonterminal symbols may be expanded into new sequences of symbols (terminal or otherwise).
- **Head (Production Rule)**: the left hand side of a production rule
- **Tail (Production Rule)**: the right hand side of a production rule
- **Parse Tree**: an ordered, rooted tree whose nodes are symbols in a context-free grammar where the children of each branch node correspond to the tail of some production rule in said grammar; a tree-representation of the grammatical structure of an input stream [17]
- **Weakly Equivalent (Grammar)**: two grammars are weakly equivalent if they define the same language. [31]
- **Strongly/Structurally Equivalent (Grammar)**: two grammars are strongly equivalent if they are weakly equivalent and there is a one to one correspondence between their parse trees which match equivalent strings. [31] [31]

Appendix

This appendix contains miscellaneous figures and examples that came out of this work.

Below is an example of SPARQL generated by one of the early iterations of our generation process which used simple string template filling. You'll notice some constants that are not abbreviated where they could be and less standard whitespace.

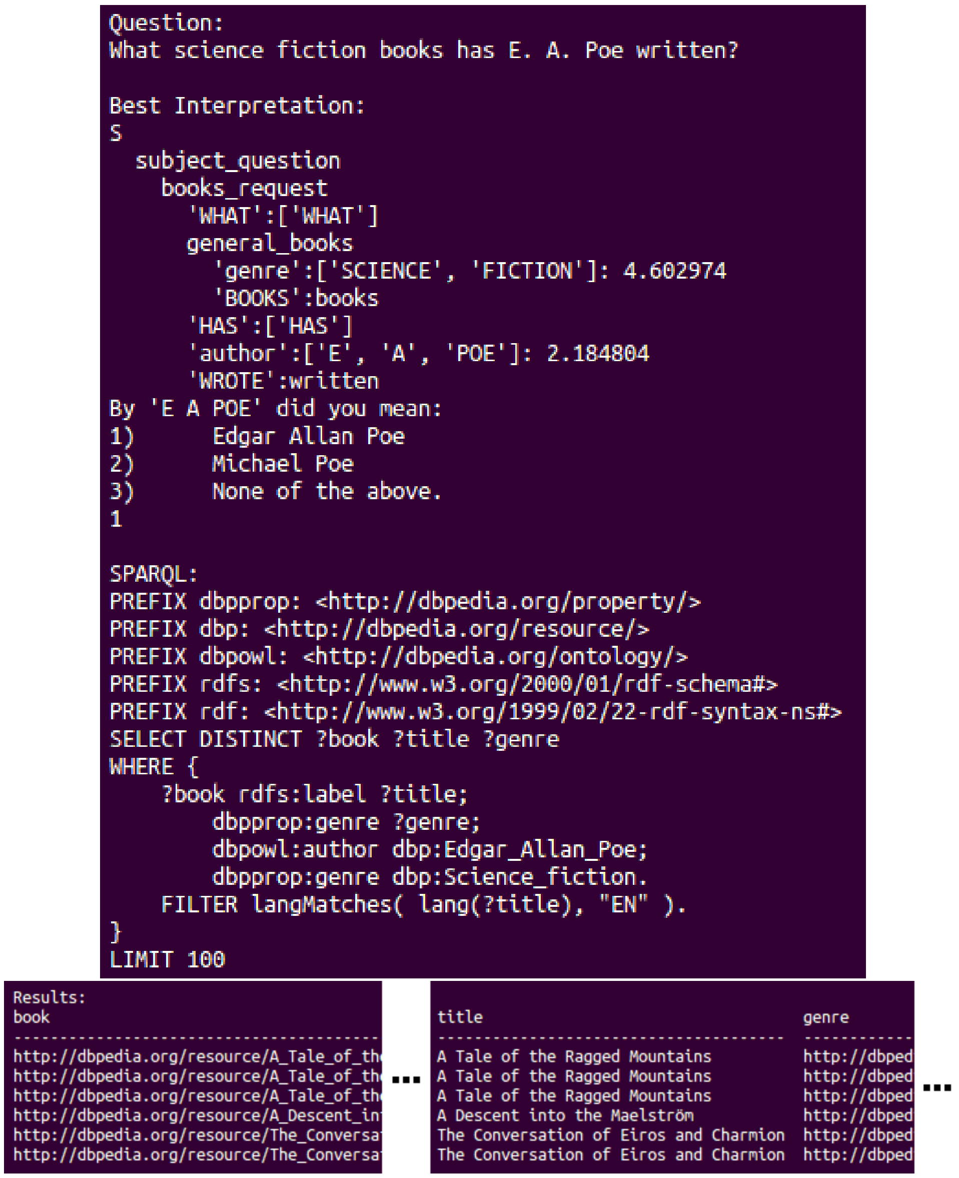
```
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX dbp:<http://dbpedia.org/resource/>
PREFIX dbpowl:<http://dbpedia.org/ontology/>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
SELECT ?author ?name
WHERE{
    ?author a dbpowl:Writer;
        rdfs:lable ?name.

    ?book0 a bpowl:Book;
        dbpowl:author ?author;
        dbpprop:genre <http://dbpedia.org/resource/High_fantasy>.


    ?book1 a bpowl:Book;
        dbpowl:author ?author;
        dbpprop:genre <http://dbpedia.org/resource/Science_fiction>.

    FILTER langMatches( lang(?name), "EN" ).
}
GROUP BY ?author
```

This is an earlier version of the grammar which was slow to parse for various reasons mentioned in section 4.1:

$$S \rightarrow subject\_question \tag{1}$$
$$subject\_question \rightarrow \text{WHAT } NP\ VP \tag{2}$$
$$subject\_question \rightarrow \text{WHAT } VP \tag{3}$$
$$subject\_question \rightarrow \text{WHO } VP \tag{4}$$
$$NP \rightarrow Adj\ NP \tag{5}$$
$$NP \rightarrow N \text{ OR } N \tag{6}$$
$$NP \rightarrow N \text{ AND } N \tag{7}$$
$$NP \rightarrow \text{BOTH } N \text{ AND } N \tag{8}$$
$$NP \rightarrow N \tag{9}$$
$$VP \rightarrow \text{HAS } NP\ V \tag{10}$$
$$VP \rightarrow V\ NP \tag{11}$$
$$Adj \rightarrow \text{genre} \tag{12}$$
$$N \rightarrow \text{genre} \tag{13}$$
$$N \rightarrow \text{author} \tag{14}$$
$$N \rightarrow \text{book} \tag{15}$$
$$N \rightarrow \text{AUTHOR} \tag{16}$$
$$N \rightarrow \text{BOOKS} \tag{17}$$
$$V \rightarrow \text{WROTE} \tag{18}$$

Figure 22: An earlier version of the parser's grammar.

And finally, figure 23 is an example of the system's operation with full output to show each step of the process:



```
Question:
What science fiction books has E. A. Poe written?

Best Interpretation:
S
  subject_question
    books_request
      'WHAT':['WHAT']
      general_books
        'genre':['SCIENCE', 'FICTION']: 4.602974
        'BOOKS':books
      'HAS':['HAS']
      'author':['E', 'A', 'POE']: 2.184804
      'WROTE':written
By 'E A POE' did you mean:
1)      Edgar Allan Poe
2)      Michael Poe
3)      None of the above.
1

SPARQL:
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX dbp: <http://dbpedia.org/resource/>
PREFIX dbpowl: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?book ?title ?genre
WHERE {
    ?book rdfs:label ?title;
        dbpprop:genre ?genre;
        dbpowl:author dbp:Edgar_Allan_Poe;
        dbpprop:genre dbp:Science_fiction.
    FILTER langMatches( lang(?title), "EN" ).
}
LIMIT 100
```

```
Results:
book
---------------------------------------
http://dbpedia.org/resource/A_Tale_of_th
http://dbpedia.org/resource/A_Tale_of_th   ...
http://dbpedia.org/resource/A_Tale_of_th
http://dbpedia.org/resource/A_Descent_in
http://dbpedia.org/resource/The_Conversa
http://dbpedia.org/resource/The_Conversa
```

```
title                                  genre
-------------------------------------  -----------
A Tale of the Ragged Mountains         http://dbped
A Tale of the Ragged Mountains         http://dbped
A Tale of the Ragged Mountains         http://dbped   ...
A Descent into the Maelström           http://dbped
The Conversation of Eiros and Charmion http://dbped
The Conversation of Eiros and Charmion http://dbped
```

Figure 23: An example of the program's output.

References

[1] Naive bayes text classification. http://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classification-1.html.

[2] Owl 2 web ontology language. http://www.w3.org/TR/owl2-overview/.

[3] Sparql 1.1 query language. http://www.w3.org/TR/sparql11-query/.

[4] Sparql query language for rdf. http://www.w3.org/TR/rdf-sparql-query/.

[5] Rdf schema 1.1. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/, 2014.

[6] Natural language toolkit. http://www.nltk.org/, 2015.

[7] Siri. https://www.apple.com/ios/siri/, 2015.

[8] Alphred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Pearson Education, Inc., 1986.

[9] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[10] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[11] Flavio Chierichetti, Ravi Kumar, Sandeep Pandey, and Sergei Vassilvitskii. Finding the jacard median. http://theory.stanford.edu/~sergei/papers/soda10-jaccard.pdf.

[12] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[13] P. Ciaccia, M. Patella, F. Rabitti, and P. Zezula. Indexing metric spaces with m-tree. http://www-db.deis.unibo.it/research/papers/SEBD97.pdf.

[14] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. https://www.cs.cmu.edu/~pradeepr/papers/ijcai03.pdf, 2003.

[15] A. Damljanovic, M. Agatonovic, and H. Cunningham. Freya: An interactive way of querying linked data using natural language. *FREyA: An Interactive Way of Querying Linked Data Using Natural Language*, 7117:125–138, 2011.

[16] Ted Thibodeau et al. Dbpedia - about. `http://dbpedia.org/About`, 2015.

[17] Alice E. Fischer and Frances S. Grodzinsky. *The Anatomy of Programming Languages*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.

[18] B. Galitsky. Natural language question answering system. *Adelaide, Australia: Advanced Knowledge International*, 2003.

[19] B. Galitsky. Implementing commonsense reasoning via semantic skeletons for answering complex questions. *American Association for Artificial Intelligence*, 2005.

[20] M. Gao, J. Liu, N. Zhong, F. Chen, and C. Liu. Semantic mapping from natural language questions to owl queries. *Computational Intelligence*, 27(2):280–314, 2011.

[21] E. Kaufmann and A. Bernstein. Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):377–393, 2010.

[22] Esther Kaufmann, Abraham Bernstein, and Lorenz Fischer. Nlp-reduce: A naive but domain-independent natural language interface for querying ontologies. In *Proc. 4th European Semantic Web Conference*, Innsbruck, Austria, June 2007.

[23] Esther Kaufmann, Abraham Bernstein, and Renato Zumstein. Querix: A natural language interface to query ontologies based on clarification dialogs. In *In: 5th ISWC*, pages 980–981. Springer, 2006.

[24] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 6(2):167–195, 2015.

[25] F. D. Lewis. Recursive descent parsing. `http://www.cs.engr.uky.edu/~lewis/essays/compilers/rec-des.html`, 2002.

[26] Peter Linz. *An Introduction to Formal Languages and Automata.* Jones and Bartlett Publishers, Inc., Sudbury, MA, 2001.

[27] Zdravko Markov and Daniel T. Larose. *Data Mining the Web - Uncovering Patterns in Web Content, Structure, and Usage.* John Wiley & Sons, Inc., Hoboken, New Jersey, 2007.

[28] Gonzalo Navarro. A guided tour to approximate string matching. `http://www.captura.uchile.cl/bitstream/handle/2250/132588/Navarro_Gonzalo_Guided_tour.pdf`.

[29] Jeff M. Phillips. Data mining - jaccard similarity and shingling. `http://www.cs.utah.edu/~jeffp/teaching/cs5955/L4-Jaccard+Shingle.pdf`, 2013.

[30] Lev Ratinov and Dan Roth. Design challenges and misconceptions in named entity recognition. In *Proc. 13th Conference on Computational Natural Language Learning*, June 2009.

[31] Stefano Crespi Reghizzi. *Formal Languages and Compilation.* Springer-Verlag London Limited, Italy, 2009.

[32] Alan Ritter, Sam Clark, Mausam, and Oren Etzioni. Named entity recognition in tweets: An experimental study. In *Proc. Empirical Methods in Natural Language Processing*, 20011.

[33] C. E. Shannon. A mathematical theory of communication. `http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf`, 1948.

[34] N. Sharef, S. Noah, and M. Murad. Issues and challenges in semantic question answering through natural language interface. *Journal of Next Generation Information Technology (JNIT)*, 4(7):50–60, 2013.

[35] William M. Waite and Lynn R. Carter. *An Introduction to Compiler Construction.* HarperCollins College Publishers, New York, NY, 1993.