

Querying the Semantic Web with Natural Language

Ernest Kirstein

March 21, 2015

Contents

1	Introduction	3
1.1	Purpose	3
1.2	Domain	3
2	Previous Work	4
2.1	Other NLI's	5
2.2	Common Issues	6
3	Converting Natural Language Questions into SPARQL Queries	7
3.1	Named Entity Recognition	8
3.2	Parsing	8
3.3	Resolution	11
3.4	Generation	11
4	Software Architecture	13
4.1	SPARQL Generation	15
5	Resolving User Input Names to RDF Entities	17
5.1	Name Standardization and Enumeration	17
5.2	Jaccard Index	18
5.2.1	Comparison with Levenshtein (Edit) Distance	19
5.2.2	Information Content Sensitive Jaccard Index	19
5.3	Ambiguity	21
6	Top-Down Parsing	23
6.1	Introduction to Top-Down Parsing without Syntax Diagrams	23
6.2	Compiling a Context Free Grammar for Top Down Parsing .	27
6.3	Effect of Grammar Transformations on Parse Trees	28
6.3.1	Substitution	29
6.3.2	Direct Left Recursion Elimination	30

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 [5]. Unfortunately for us humans, the semantic web has proven rather difficult for casual users to interact with.

1.1 Purpose

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. One of the main interfaces is called SPARQL. SPARQL is a query language that is used to ask questions about semantic data [3]. The main benefit to this type of interface is that it allows users to ask complex, exact questions. The main drawback, being a formal query language, is that SPARQL has a steep learning curve and is only accessible to expert users.

The system we propose in this work attempts to make the semantic web more accessible by allowing users to ask questions in natural language (i.e. plain English). The actual implementation of this tool handles only a small range of questions, but it does demonstrate promising possibilities.

1.2 Domain

The world wide web has its foundation in HTML/CSS/JS documents and HTTP requests. Similarly, the semantic web is rooted in RDF/OWL documents and SPARQL queries.

RDF documents are XML pages which describe object types (classes), and instances of those types. They can also document relations between objects hierarchically or compositionally. RDF documents are quite expressive, in and of themselves.

Then there are OWL ontologies which are RDF documents for describing relations between RDF entities [1]. OWL is more technically specified so as to aid in further automated reasoning over RDF data stores. RDF may make the semantic web expressive, but OWL allows it to be intelligent.

The website DBPedia [11, 19] 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 [16,29], but is not a key focus of this work.

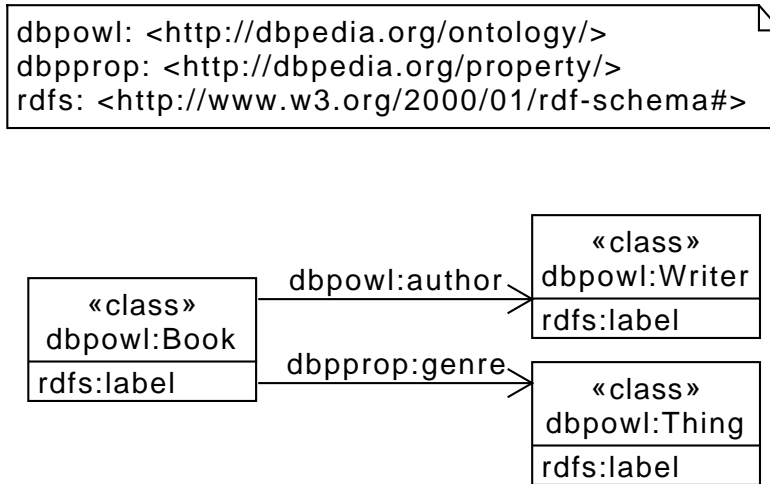


Figure 1: A few classes of the DBPedia ontology

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. We hoped our system would

be able to handle questions as complex as:

- What science fiction books has George Martin written?
- Which authors have written both science fiction and high fantasy novels?
- Which books has Dave Wolverton written that are the same genre as Eragon by Paolini?
- Who has written the most historical fiction books?

2 Previous Work

Work by Kaufmann and Bernstein [16] 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. [16]

Other recent works [10, 13–15, 17, 18] 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 [16]. This section will discuss a few of the more recent ones.

NLP-Reduce [16, 17] 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” [17]. Querix [18] 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 by casual users [16].

Another earlier system by Glaitsky [13] 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 [10] is a system which focused on the portability problem of most NLI systems. 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. [29] 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 [29]. This might even include issues of trusting information sources [5].

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. [16]

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). [16]

3 Converting Natural Language Questions into SPARQL Queries

Since SPARQL is already a well-worn tool for navigating the semantic web, it makes sense to make use of it. This system we have built receives natural language questions such as, “What author has written both high fantasy and science fiction?” Then the system produces a SPARQL query which asks the same question. The end goal is to allow general users to achieve the same level of sophistication as expert users using English rather than SPARQL.

The foundations of both compiler design and natural language processing have significant overlap [7, 26]. However, in practice, the two different disciplines often take very different approaches to problems [4, 12, 26]. Some think [16] that a middle ground between these estranged fields is where NLI will be grown successfully.

Our approach was to build a “natural language compiler” of sorts. That is, to approach the problem of converting natural language questions into SPARQL queries just as one might convert Java source code into byte code. The problems are several orders of magnitude apart in complexity, so our only expected goal was only to parse a limited range of natural language questions. But in that range of questions, we intend to show that multifaceted questions can be handled on a limited basis.

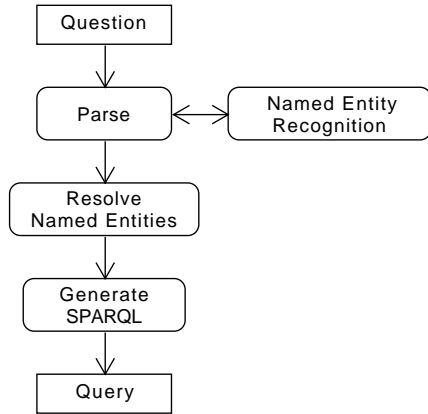


Figure 2: An overview of the NL to SPARQL Process

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 nouns, adjectives, or verbs whose names must be scraped from the RDF database. 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 Named Entity Recognition

Natural language names of ‘named objects’ are recognized by a component of the system called the ‘named entity recognizer’ [25, 27]. The NER is greatly improved by context from being integrated into the parsing process. Still, setting up the NER for a higher level of accuracy requires a bit of preprocessing.

Statistical character models are constructed from the output of queries for all objects of a certain type in the target RDF database. These models are used to recognize arbitrary tokens (or combinations of tokens) within the parser’s grammar. Generally, we have found that it was sufficient to simply pull the RDFS label for all the objects of a class to use as the data set for recognizing the names.

These models allow for probabilistic recognition of the input strings in $O(1)$ time (in relation to the number of objects in the RDF database). They recognize variations on the names (typos, variations, etc.) within some user-defined bound.

Furthermore, these models allow for $O(1)$ 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?).

3.2 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. The parser automatically handles left recursion and a few other hiccups which usually give top-down parsers trouble. However, 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.

The grammar generated to handle the natural language questions is composed of rules which handle each type of question. The ‘type’ of question is vague: it is largely a matter of design preference. The broader the scope of a ‘type’, the harder it will be to develop a grammar which recognizes that question. But the narrower the types, the more types will need to be implemented to cover the same range of questions.

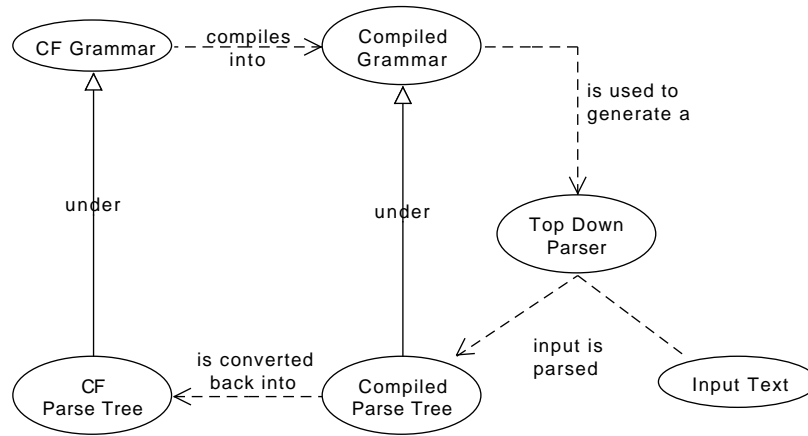


Figure 3: A high level overview of the parsing process.

```

Grammar:
0) S -> bibliography_request
1) bibliography_request -> 'WHAT' 'BOOKS' 'HAS' 'author' 'WROTE'
2) S -> author_search
3) author_search -> 'WHO' 'HAVE WRITEN' BOOK_DESCRIPTION
4) author_search -> 'WHICH' 'AUTHOR' 'HAVE WRITEN' BOOK_DESCRIPTION
5) BOOK_DESCRIPTION -> 'BOTH' BOOK_DESCRIPTION 'AND' BOOK_DESCRIPTION _BOOK_DESCRIPTION0
6) _BOOK_DESCRIPTION0 -> 'AND' BOOK_DESCRIPTION _BOOK_DESCRIPTION0
7) _0 -> ε
8) _0 -> 'BOOKS'
9) _BOOK_DESCRIPTION0 -> ε
10) BOOK_DESCRIPTION -> 'genre' _0 _BOOK_DESCRIPTION0
  
```

Figure 4: Grammar

Grammar rules are defined for each question, then compiled into a single unified grammar (see figure 4) with left recursion and other such nuisances automatically ‘compiled’ into the grammar. Again, that is to be discussed in a later section.

Given a natural language question, the parser then produces a number of parse trees and selects the “best” (most probable) interpretation. That interpretation gets converted back into a form that would be produced from the original grammar (with left recursion etc. restored).

```

Question:
Which writers have published both high fantasy and science fiction?

Best Interpretation:
S
  author_search
    'WHICH': ['WHICH']
    'AUTHOR': writers
    'HAVE_WRITEN'
      'HAVE': ['HAVE']
      'WROTE': published
    BOOK_DESCRIPTION
      'BOTH': ['BOTH']
    BOOK_DESCRIPTION
      GENRE_DESCRIPTION
        'genre': ['HIGH', 'FANTASY']: 4.475702
      'AND': ['AND']
    BOOK_DESCRIPTION
      GENRE_DESCRIPTION
        'genre': ['SCIENCE', 'FICTION']: 4.602974

```

Figure 5: Input and Parse Tree

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.

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.

3.4 Generation

Generating SPARQL queries from the resolved parse trees is the last step. Being the last step in a complex process, it is highly couple to the output of the previous steps which means that this module needs to be particularly adaptable. However, it is difficult to design such a module due, in part, to the complexity of the SPARQL query language. So, it took several iteration

to refine our approach.

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 variability 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.

```
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#>
SELECT ?author ?name
WHERE{
    ?author a dbpowl:Writer;
        rdfs:label ?name.

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

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

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

Figure 6: SPARQL Output from the first approach. It's ugly, but it worked.

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 tuples, applying filters, etc.). See section 4.1 for more on this.

In that iteration, our line-count for the query-generating code ballooned 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.

In the final iteration, we tried to increase the adaptability 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).

```
Grammar:
0) S -> subject_question
1) subject_question -> 'WHAT' NP VP
2) subject_question -> 'WHAT' VP
3) subject_question -> 'WHO' VP
4) NP -> Adj NP
5) NP -> N 'OR' N
6) NP -> N 'AND' N
7) NP -> 'BOTH' N 'AND' N
8) NP -> N
9) VP -> 'HAS' NP V
10) VP -> V NP
11) Adj -> 'genre'
12) N -> 'genre'
13) N -> 'author'
14) N -> 'book'
15) N -> 'AUTHOR'
16) N -> 'BOOKS'
17) V -> 'WROTE'
```

Figure 7: More difficult to convert, flexible grammar.

```
Grammar:
0) S -> subject_question
1) subject_question -> books_request
2) subject_question -> authors_request
3) books_request -> 'WHAT' general_books 'HAS' 'author' 'WROTE'
4) authors_request -> 'WHAT' general_authors 'HAS' 'WROTE' general_books
5) authors_request -> 'WHO' 'HAS' 'WROTE' general_books
6) authors_request -> 'WHAT' 'AUTHOR' 'WROTE' 'book'
7) authors_request -> 'WHO' 'WROTE' 'book'
8) general_books -> 'BOOKS'
9) general_books -> 'genre'
10) general_books -> 'genre' 'BOOKS'
11) general_books -> 'genre' 'AND' 'genre'
12) general_books -> 'genre' 'AND' 'genre' 'BOOKS'
13) general_authors -> 'AUTHOR'
14) general_authors -> 'genre' 'AUTHOR'
```

Figure 8: Easier to convert, rigid grammar

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 much harder to parse. Such grammars (see figure 7).

let in extraneous interpretations that (while gramatically 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 writen?” is rather easy to interpret because we can imediately classify ‘scify’ as a genre and ‘john wane’ as a name. But the parser is less sure (something which could be tweaked) and erres on the the side of caution so it has to check most words as possibly being each type of noun (where the grammar allows) and it furthermore needs to check many possible divisions between words and needs to consider misspellings. All of this is handled more efficiently if one simply specifies the grammar more explicitly in parser-friendly divisions (see figure 8).

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.

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 I 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. 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

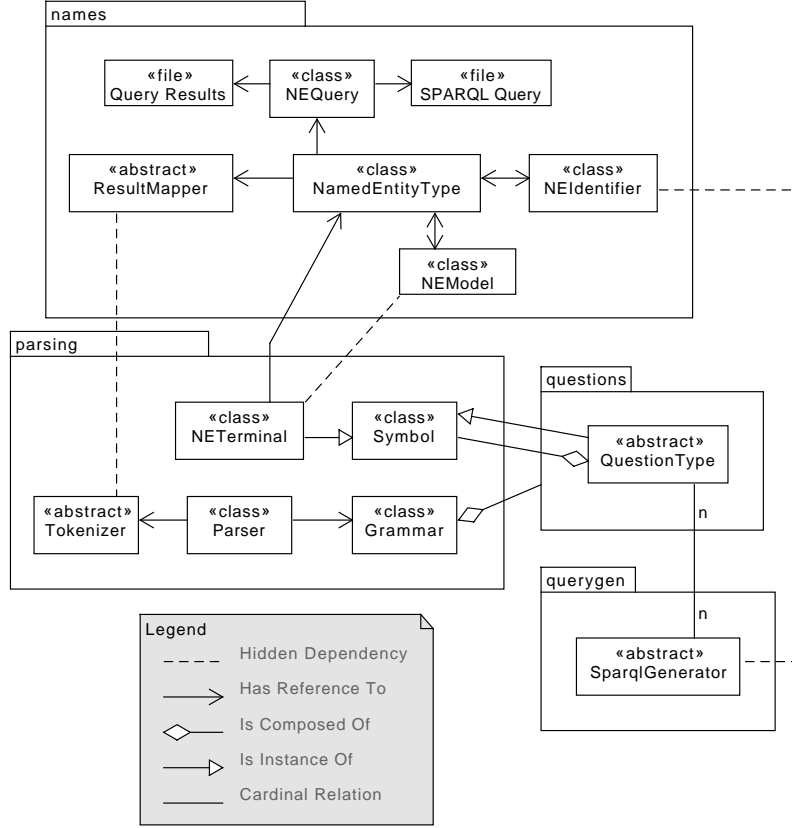


Figure 9: Architecture

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 [2,3]. Generating SPARQL queries can be more complicated than simply fitting URIs into a templated string (which was our first approach to the problem). Figure 10 is a UML diagram of the query generation framework we designed with several goals in mind.

First and foremost, 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

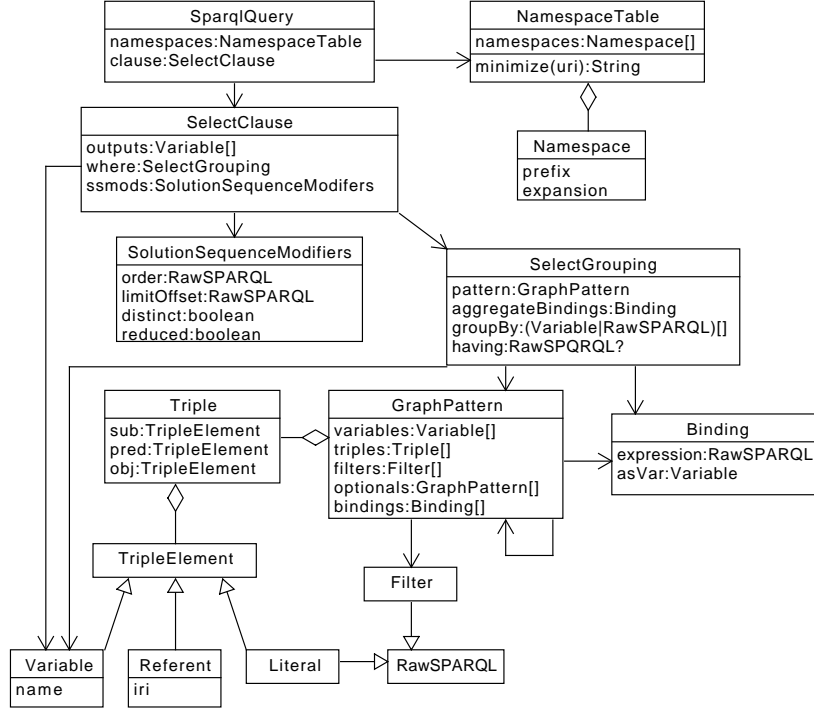


Figure 10: SPARQL Generation [2, 3]

manually inputting strings. Since SPARQL isn't trivial, 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 understanding and using SPARQL correctly.

It is also difficult to subdivide the task of building a query when that task only involves fitting strings into a 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 more daftly 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 except more so.

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.

My 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. 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 [9]. For my particular application, I 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 [9] and anecdotal evidence 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 I 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 trivialities to deal with. Names with abbreviations, punctuation, and names with multiple parts can all trip up comparison algorithms.

One standardization method I 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's best to tokenize the name and include the different partial variations as other names linked with the entity. As part of my own system, I 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 [6], 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 [9, 24]:

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

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' [24]) 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.

For a distance function, one can use:

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

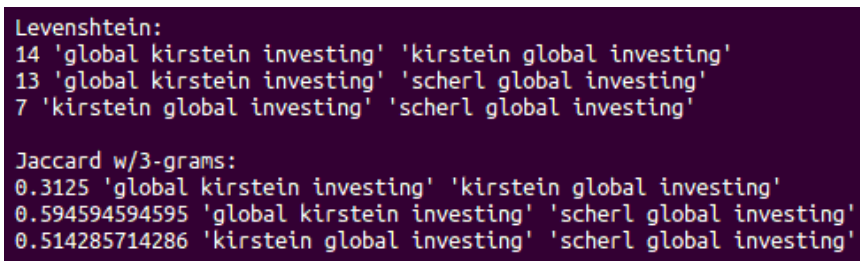
The Jaccard distance metric is a true metric space for ngrams [22]. But it doesn't quite produce a true metric space for string (since two different strings can have exactly the same n-grams). Still, it does satisfy the triangle inequality [6] and is always non-negative. As such, one could conceivably

construct an M-tree [8] 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 [23] 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 12.



```
Levenshtein:
14 'global kirstein investing' 'kirstein global investing'
13 'global kirstein investing' 'scherl global investing'
7 'kirstein global investing' 'scherl global investing'

Jaccard w/3-grams:
0.3125 'global kirstein investing' 'kirstein global investing'
0.594594594595 'global kirstein investing' 'scherl global investing'
0.514285714286 'kirstein global investing' 'scherl global investing'
```

Figure 11: 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" is such a common surname.

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 [6]:

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

Where \vec{x} and \vec{y} are large dimensional vectors 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 .

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

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 [28]. 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:

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

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: $\hat{J}(x, y)$:

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

```

Without Info:
0.647058823529 'tom smith' 'john smith'
0.769230769231 'tom smith' 'tom'
1.0 'john smith' 'tom'

Sample Set:
adam smith
bob smith
carl smith
dale jones
ernest kirstein

With Info:
0.729636835639 'tom smith' 'john smith'
0.723483108499 'tom smith' 'tom'
1.0 'john smith' 'tom'

```

Figure 12: 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 [28] 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. [20,21] Coding a RDP 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. 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 [30].

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 [20] 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 [30].

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:

$$S \rightarrow aS \quad (1)$$

$$S \rightarrow bS \quad (2)$$

$$S \rightarrow \epsilon \quad (3)$$

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 13)

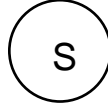


Figure 13: Parse Tree 0

The next parse tree we should consider is the parse tree that is generated when we follow the first production rule. (Figure 14) 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".

For the next parse tree, we will try to repeat are last action (following the first possible production rule). (Figure 15) This parse tree conflicts with the string we are trying to produce since any string produced by further

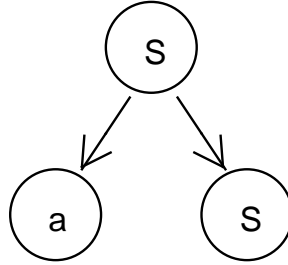


Figure 14: Parse Tree 1 - Valid

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.

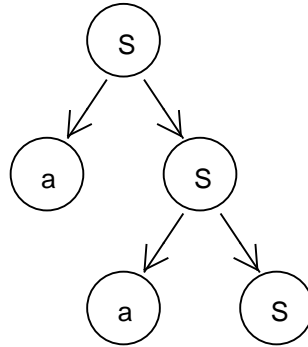


Figure 15: Parse Tree 2 - Invalid

In figure 16, 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 17), 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 18)

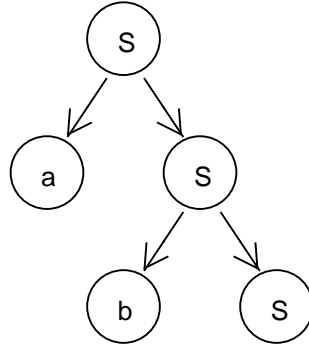


Figure 16: Parse Tree 3 - Valid

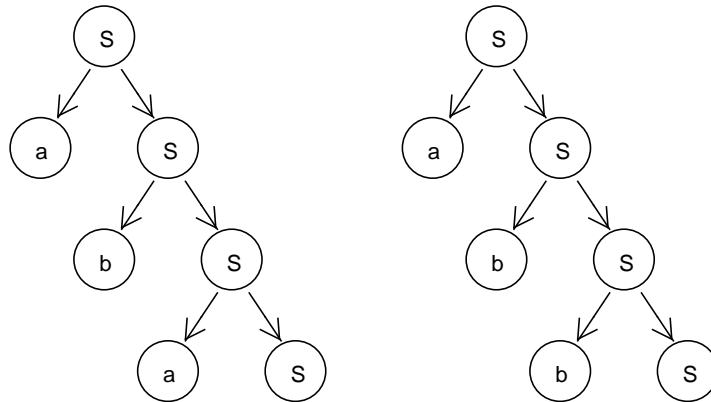


Figure 17: Parse Trees 4 and 5 - Both Invalid

Finally, let's diagram our traversal through the possible parse trees (Figure 19). 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. 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.

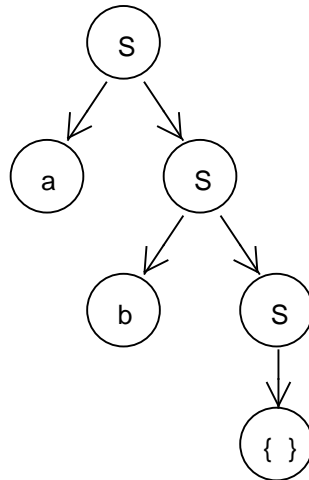


Figure 18: Parse Tree 6 - Complete

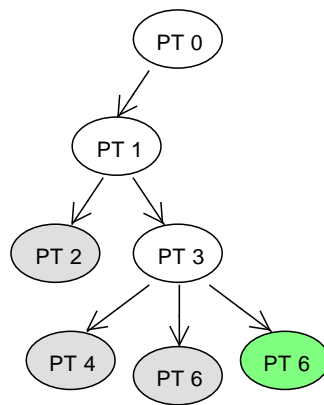


Figure 19: Parse Tree Search Progression

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) [30]. 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. My 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 produces a weakly equivalent 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. My parser keeps track of these inverse transformation rules as performs it's compilation procedure so that a compiled grammar's decision list can be easily converted to the initial grammar's equivalent decision list.

Take this simple grammar for example:

$$S \rightarrow AB \tag{1}$$

$$A \rightarrow a \tag{2}$$

$$A \rightarrow SA \tag{3}$$

$$B \rightarrow b \tag{4}$$

$$B \rightarrow SB \tag{5}$$

It compiles into the weakly equivalent grammar:

$$Z \rightarrow \epsilon \quad (1)$$

$$B \rightarrow b \quad (2)$$

$$S \rightarrow aBS' \quad (3)$$

$$S' \rightarrow \epsilon \quad (4)$$

$$A \rightarrow aZ \quad (5)$$

$$B \rightarrow aBS'B \quad (6)$$

$$S' \rightarrow aZBS' \quad (7)$$

$$Z \rightarrow bS'A \quad (8)$$

$$Z \rightarrow aBS'BS'A \quad (9)$$

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].

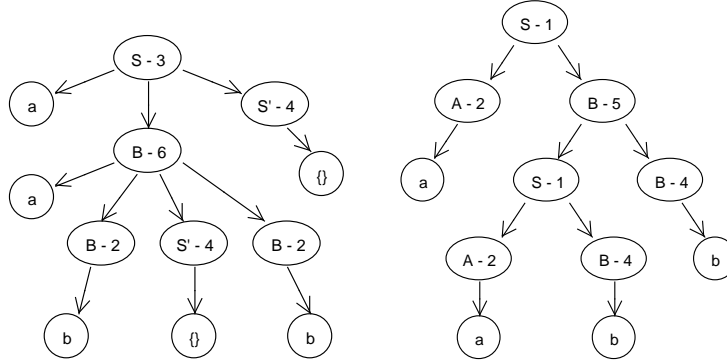


Figure 20: Compiled Grammar Parse Tree (left) Parent Grammar Parse Tree (right)

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, it's 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, $DLRE(G; R_\alpha, R_\beta) \rightarrow G'$ and substitution, $SUB(G; R_\alpha, R_\beta) \rightarrow G'$, which is necessary to remove indirect left recursion. [4,20] 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:

$$\begin{aligned} R_\alpha &= A \rightarrow B\alpha \\ R_\beta(i) &= B \rightarrow \beta_i \end{aligned}$$

Where $i \in [1, m]$. Let R_β represent the sets of the B rules where the notation $R_\beta(1)$ represents $B \rightarrow \beta_1$. 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 [4] $SUB(G; R_\alpha, R_\beta) \rightarrow G'$ which applies the substitution replaces the single-element R_α rule set with combined rules for each R_β :

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

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

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

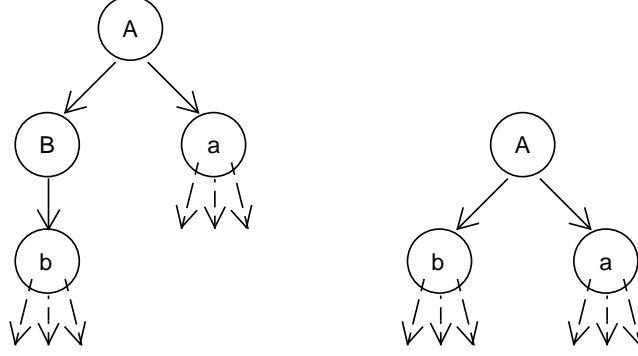


Figure 21: Original and Transformed Parse Trees

The transformation just removes the B node in the parse tree (Figure 22). 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 $SUB^{-1}(T'; A, B) \rightarrow T$ 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:

$$\begin{aligned} R_\alpha(i) &= A \rightarrow A\alpha_i \\ R_\beta(j) &= A \rightarrow \beta_j \end{aligned}$$

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

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

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

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

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

The transformation [4] $DLRE(G; R_\alpha, R_\beta) \rightarrow G'$ 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:

$$\begin{aligned} R'_\alpha(i) &= A' \rightarrow \alpha_i A' \\ R'_\beta(j) &= A \rightarrow \beta_j A' \end{aligned}$$

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:

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

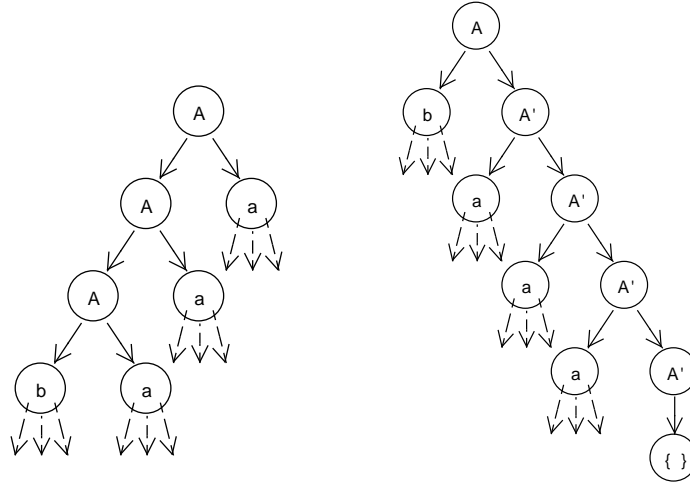


Figure 22: Original and Transformed Parse Trees

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 22) 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'_{x_p} node are $C_{A'_{x_p}}$, 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 $C_{A'_{x_p}}$ 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 $C_{A_{x_p}}$ 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.

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... [7] (see Production Rule)
- **Context-Free Grammar:** a context free grammar is one which only has production rules whose head is a single nonterminal symbol. [12, 21, 30]
- **Production, Production Rule, Rewrite Rule:** rules of the form $X \rightarrow Y$ where X and Y are strings in [a grammar] [7]; define the nonterminal symbols by sequences of terminals and nonterminal symbols [30]; 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] [12]
- **Weakly Equivalent (Grammar):** two grammars are [weakly] equivalent if they define the same language. [26]
- **Strongly/Structurally Equivalent (Grammar):** two grammars are strongly or structurally equivalent if they are weakly equivalent and can assign any sentence the same parse tree. [26]

References

- [1] Owl 2 web ontology language. <http://www.w3.org/TR/owl2-overview/>.
- [2] Sparql 1.1 query language. <http://www.w3.org/TR/sparql11-query/>.
- [3] Sparql query language for rdf. <http://www.w3.org/TR/rdf-sparql-query/>.

- [4] Alfhred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Pearson Education, Inc., 1986.
- [5] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.
- [6] Flavio Chierichetti, Ravi Kumar, Sandeep Pandey, and Sergei Vassilvitskii. Finding the jacard median. <http://theory.stanford.edu/~sergei/papers/soda10-jaccard.pdf>.
- [7] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.
- [8] 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>.
- [9] 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.
- [10] 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.
- [11] Ted Thibodeau et al. Dbpedia - about. <http://dbpedia.org/About>, 2015.
- [12] Alice E. Fischer and Frances S. Grodzinsky. *The Anatomy of Programming Languages*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.
- [13] B. Galitsky. Implementing commonsense reasoning via semantic skeletons for answering complex questions. *American Association for Artificial Intelligence*, 2005.
- [14] B. Galitsky. Natural language question answering system. *Adelaide, Australia: Advanced Knowledge International*, 2013.
- [15] 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.

- [16] 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.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] F. D. Lewis. Recursive descent parsing. <http://www.cs.engr.uky.edu/~lewis/essays/compilers/rec-des.html>, 2002.
- [21] Peter Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers, Inc., Sudbury, MA, 2001.
- [22] 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.
- [23] Gonzalo Navarro. A guided tour to approximate string matching. http://www.captura.uchile.cl/bitstream/handle/2250/132588/Navarro_Gonzalo_Guided_tour.pdf.
- [24] Jeff M. Phillips. Data mining - jaccard similarity and shingling. <http://www.cs.utah.edu/~jeffp/teaching/cs5955/L4-Jaccard+Shingle.pdf>, 2013.
- [25] Lev Ratinov and Dan Roth. Design challenges and misconceptions in named entity recognition. In *Proc. 13th Conference on Computational Natural Language Learning*, June 2009.
- [26] Stefano Crespi Reghizzi. *Formal Languages and Compilation*. Springer-Verlag London Limited, Italy, 2009.

- [27] Alan Ritter, Sam Clark, Mausam, and Oren Etzioni. Named entity recognition in tweets: An experimental study. In *Proc. Empirical Methods in Natural Language Processing*, 2011.
- [28] C. E. Shannon. A mathematical theory of communication. <http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf>, 1948.
- [29] 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.
- [30] William M. Waite and Lynn R. Carter. *An Introduction to Compiler Construction*. HarperCollins College Publishers, New York, NY, 1993.