Formal Grammar as a Basis for Interpreting English Statements

Zach Dwiel
Washington University in St. Louis
zdwiel@wustl.edu
December 2007

Abstract Information is becoming more abundant and easier to access every day through sources such as DBpedia.[1] Simple interfaces to this vast store of knowledge are becoming vital to its value. There is a large amount of research translating from language to language, and attempting to understand which part of speech words belong to. I believe that by limiting the scope to the kinds of language used to communicate with a computer, the problem becomes much more manageable and the language can be viewed in its simplest form as a formal grammar. This provides a vastly simplified view of the language and makes it easy to quickly design software which can interpret a wide variety of statements and queries.

1 Introduction

I will examine the potential for using a formal grammar-based parser to interpret basic commands given in plain English to computers as a new way of accessing information. The system will be designed to be scalable, with the intent for the parser to understand commands covering a large domain, conceivably the majority of the commands that a average user might think to ask. However, to keep the analysis simple, the system will be tested with commands dealing only with querying and creating emails with at least the functionality that a typical email client such as Outlook or Thunderbird might allow.

A lot of thought and research went into deciding what would be the best foundation for this system. The behind the scenes technologies that were used greatly influenced the design and ultimate success of many parts of the system due to their analogous design patterns.

I approached this problem with a focus on the following features in mind:

- **Simple** - a way to map English commands into queries or computer understandable instructions. Simplicity in the formation of these instructions is important in allowing others to help adapt and improve upon the system.
- **Repeatable** - it should not appear fuzzy to the user in that commands should either work as expected or ask for more information. Performing actions that were not asked for is far worse than not performing actions that were.
- **Flexible** - users must be able to formulate any request appropriate for the given domain, in the language that makes the most sense to them. A strict language that the user must learn is not acceptable.
- **Abstract** - the system should store and access data in a format that can be used to describe all kinds of data. New ontologies [2] should be easy to add. The system should operate on a level above a database, where the data types have already been abstracted.

2 Database

An RDF [3] database was used to store all of the data. This level of abstraction has become very popular for describing information on the web. There is also a very popular query language not unlike SQL called SPARQL [4] and which can be used to access data. It does have some very powerful differences. One of the advantages to an RDF over an SQL database is that there is no need for different tables for each data type. All data is stored in a single large database which is accessible in the same way. This provides a nice abstraction layer over the data. Like SQL, it allows arbitrary algebraic queries and uses standard data types. This is also the database format and query language which is used by DBpedia and at semantic web labs at MIT [5] among others. It is popular in new semantic web applications so there is a constant stream of new data being released in this format.

3 Parser

The parser is a formal grammar parser at its core with extra cases to handle types other than simple nonterminal or terminal symbols. There were many reasons that this seemed like a potentially fruitful path. To start with, there was very little published research about it. It seemed to satisfy many of the requirements that I had set out at the start. A formal grammar will be repeatable thanks to

its strictness. It is also highly modular. Once the grammar for a particular ontology has been engineered, that grammar can be used in all other cases where that ontology is used. For example, once phrases about dates and times can be understood (this week, last week, 1 year ago, three years ago, sometime last year, earlier this morning, etc) they can be easily reused by later grammar programmers, making it very easy to implement their own language without worrying about common sub-domains. In this way, I plan on deconstructing the basic building blocks of language not into its parts of speech or standard grammatical rules but functional blocks of information.

A formal grammar alone, quickly fails at many of the requirements above, namely flexibility. Formal grammars are by definition very strict whereas English is not. However, I hope to show that this strictness can be overcome with a few slight additions to the basic formal grammar algorithm and practical restrictions to the size of the language to be understood.

One simple addition would allow symbols to be matched against an arbitrary function written in a basic scripting language. This allows a programmer to match emails with the regular expression `/^[\w_\.-]+@[\w_\.-]+$/` for example. This also allows more complex functions such as queries into the database to determine if a given symbol is a name in our contact list or not. This makes it easier to make informed decisions about what type of data each phrase is most likely to be. It also increases the flexibility of the grammar. It now has a way to understand symbols that follow a known pattern without having to know what all of the possible symbols might be.

Another problem with this approach is that it can produce multiple matches for a single query each with different meanings, some of them sensible, some of them not. In some cases, the ambiguity is in the command given; even a human would not be able to unambiguously determine the intent of the user. However, some cases need to employ other measures in order to work past a limitation of this approach. In the work that I did, this was rarely a problem. It is possible, although not necessarily likely that by adding new domains to the grammar this might become more of a concern.

3.1 Context

I also investigated how the context could play a role and how people use it. The context being defined as the information about the most recently used commands. I took advantage of the

common syntactic style where extra information is added onto an past commands: `emails from bob ↵ yesterday ↵ about noon ↵` In this case, flattening all of these into one statement actually produces a single sentence which retains the relationships between all of the ideas that the speaker implied: `emails from bob yesterday about noon` is a sentence that should be understandable already. In this way, some uses of the context were very simple to handle.

A slightly more flexible way to handle the context was to allow the most recent queries to fill in a non-terminal symbol in the sentence if one was missing. It uses the same grammar and parser, it simply allows one of the symbols throughout the parse to be pulled from the context. This actually did not work very well, causing more misunderstandings than simply concatenating previous statements. There is however hope for this approach if disambiguation algorithms are used. For this test, there were none.

As mentioned earlier, one of the major problems with such a strict grammar is that it might be difficult to handle the infinitely complex expressiveness found in English. To help deal with this, it has been made very easy to supply many possible phrases that all match to same rule or translation. It is also easy to add modifier words and phrases that change slightly the meaning of the surrounded data. One example is the addition of the word about to the beginning of a date range. This can be mapped to an arbitrary (or statistically generated) error bound added to the range. This resulting data will be of the same %date% type and so will fit in any other phrase that matches against a %date%. As for this example there are 8 different ways I've come up with to express this fuzziness around a date. When each rule can be defined with so many possible matches, the total number of understood commands increase exponentially.

3.2 Example

Here is an example rule which is actually used in this demo system:

```
Matches:
       %date%s %emails%
       %date%'s %emails%
       %emails% %date%
       %emails% from %date%
       %date% %emails%
       %emails% on %date%
Matchtypes:
       emails
Content:
```

```
return query_build(
       vars.emails,
       prefix("dc:", "http://purl.org/dc/elements/1.1/"),
       where("?email", "dc:date", "?date"),
       filter("regex(?date, '" .. vars.date .. "')")
       )
```

This example shows a rule which can add a date requirement to a set of emails. Phrases inside of `%s` are nonterminal symbols and all other characters comprise the terminal symbols. This rule itself is of the type 'emails' and can be substituted into any other rule which requires this datatype. In this case, it only matches one type, but there could be a list of types (this list would be equivalent to the left hand side of a BNF rule). The content is code which uses the values from the matched values date and emails as stored in the variables `vars.emails` and `vars.date`. In this case a query is built starting with the emails query and appending date requirements to it. This example is a common type of rule which deals with building a SPARQL query.

As a test of this system, I created a set of rules that I thought were fairly comprehensive. I then let others use the system to get a sense of how well I was able to define the language. There are some features that I was not able to implement due to time limitations such as finding emails that the user was cc'd on.

4 Results

When I let other people use the system to see what they would try, 68% of the 393 queries entered were successful. Of the unsuccessful queries, many were only unsuccessful because the user searched for people or words in the email that did not exist. They did not match anything because there was nothing there to match, for example there are no Brians in my contact list, so the query `emails from brian` were not successful. This situation could be dealt with a more informative match which could respond with 'there are no emails from Brian' rather than simply not understanding the user. However, I did not implement this.

Some other cases such as appending please to the beginning of a query caused problems a few times, but was easily fixes. The word was added to a list which matched any type and simply returned the matched data unchanged. Another query referred to email as e-mail, again a fix which involved adding e-mail to a list which previously only contained email, mail, e, etc. This could most likely be fixed with a simple spell checker.

Other common cases included referring to date formats that I had not yet considered, names of months being one of them as well as standard mm/dd/yyyy formats. These are also easily added to the system. The most common case which was not understood and not trivially corrected was when the user used the query as a general search looking for keywords in the email. Typing a person's name produces a most likely undesired result, a printout of their email address. A list of emails from them might also be what the user was looking for.

In most cases, the user was able to quickly figure out how to use the system and thought of successfully interpreted queries. This could have also been due to the list of examples provided to guide them in understanding what was expected. The test was still slightly difficult to explain and because I was unable to import data from each tester's email account, instead they were given access to my own personal email archive which left people unsure of what to even search for.

4.1 Efficiency

I did run into some issues with the size of the search as new rules were added, many of which included no terminal symbols. However, this did not pose any serious problems. With 83 rules in the final system (about 30 of which deal only with translating numbers from their English equivalents) the largest search performed was about 3000 node visits. Each of these visits consist of a regular expression match and about 25% of the time, a query into the RDF database. A search this big took many seconds to execute, however with some simple caching, about 2/3 of the node visits and database queries were eliminated. As the size of the grammar increases, this will become a bigger problem. However there are many other conceivable heuristics which could decrease the search time drastically that were not explored. Simple heuristics commonly used in other search algorithms should apply to this one also.

5 Further Study

There is a lot of room for further study in this area. Other algorithms could be used as backup in the case that this one fails. A basic keyword search feature as the default for short queries would likely be fruitful given the experimental user data that I have collected.

One area where there is a lot of room left is in resolving ambiguous queries. In most cases there was only one possible match, however, there were some cases where the ambiguity should have been resolved. Calculating statistics about which rules are used most often and from which rules the search has thus far traveled could yield valuable information about which rule is most likely to apply.

The same algorithms which will improve the ambiguous queries will also help with optimization. The search is currently a complete search, which should not be necessary if there is a heuristic for finding the most likely rules to apply for each query. This should also cut down on any increase in processing time caused by a larger grammar.

It should also be interesting to query a database such as the DBpedia [1] or some other RDF store. There are a large number of these types of databases containing a lot of interesting information that would be more accessible if an English query interface were made available.

6 References

[1] Soren Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard, Cyganiak, and Zachary Ives. *DBpedia: A Nucleus for a Web of Open Data*. in *The Semantic Web* 2007.

[2] Nicola Guarino. Formal Ontology and Information Systems. in *Proceedings of FOIS'98, Trento, Italy, 6-8 June* 1998

[3] Dan Brickley, W3C, R.V. Guha, IBM. *RDF Vocabulary Description Language 1.0: RDF Schema.* http://www.w3.org/TR/rdf-schema/ 2004

[4] Eric Prud'hommeaux. *SPARQL Query Language for RDF*. http://www.w3.org/TR/rdf-sparql-query/ 2007

[5] MIT. SIMILE Project. http://simile.mit.edu/