Implementation of Natural Language Processing using Python

Jon-Paul Boyd

16th January 2018

Abstract In this report I describe the design of an application which transforms human natural language commands into the formal, "artificial" SQL language for querying of a movie database which negates the need for end-user technical database expertise. This application combines the simplicity and flexibility of Python [1], a procedural, object-orientated programming language, with the extensive and easily accessible language processing features of the open source Natural Language Tool Kit (NLTK) library [2]. By representing a sample of movie database queries in human form with NLTK grammar rules, and matching of rule nouns and verbs with a WordNet [3] corpus of synonyms, I will show how a larger number of meanings can be derived and transformed into SQL code generation from a compact set of grammar rules, facilitating rapid development and simplifying rule management whilst improving the overall robustness, usability and success of the application.

Keywords natural language processing, Python, Prolog

1. Introduction

By way of brief background, given the title of the IMAT5118 A.I. Programming assignment 2 module is *"Implementation of NLP using Prolog*", I agreed with David Elizondo and Muhanad Younis in November 2017 that development of a Natural Language Processing (NLP) application using Python in place of Prolog was acceptable. Changing programming language to Python was particularly relevant to me professionally, given today's prevalence of Python in machine learning and other AI fields. IEEE.org ranked the popularity of several dozen programming languages in 2017 [4], combining 12 metrics from 10 sources [5], and Python comes out on top at number 1. Stackoverflow [6], an extremely popular developer community, published an article titled "*The Incredible Growth of Python*" [7] which makes the case for Python "*being the fastest-growing major programming language*" with "*June 2017 was the first month that Python was the most visited tag on Stack Overflow within high-income nations. This included being the most visited tag within the US and the UK, and in the top 2 in almost all other high income nations (next to either Java or JavaScript)*". For the language processing component, I will use NLTK, an open source python toolkit which has a rich set of features for processing raw text, tagging words, classification of text, analysing sentence structure and building grammars. It is very easy to quickly prototype a solution, and is widely used in education, with 32 universities in the US and 25 countries using NLTK in their courses [8]. Given the popularity and accessibility of both Python and NLTK there was high incentive to learn both as a rewarding by-product of completing this assignment.

Committed in my choice of programming language and toolkit and yet a little daunted by the task ahead, I was interested in understanding the approach of others who undertook the design of a Natural Language Interface to Database (NLIDB). In the work of [9], "*Natural language has been the holy grail of query interface designers, but has generally been considered too hard to work with, except in limited specific circumstances*". This exception is important and encouraging, as it reminds that the SQL language and a well-formed SELECT statement for retrieval of data has a finite keyword corpus and limited grammatical structure, and may be achievable in an assignment which needs to have sensible boundaries and scope. The authors go on to remark "*The fundamental problem is that understanding natural language is hard. Even in human-to-human interaction there are miscommunications. Therefore, we cannot reasonably expect an NLIDB to be perfect*". I take this to underline that any solution needs to be robust and resilient to error when a human to formal translation cannot be made. I was very much interested by their technical design specific to the problem of SQL statement generation, with the implementation of a set of nodes, for example "*Select Node (NN)*", "*Operator Node (ON)*" and "*Function Node (FN)*", that formally represent component parts of the SQL grammar such as the SELECT keyword, operators like = and <=, and aggregate functions like AVG [9]. Given more time and a broader scope this is likely the approach I would have adopted: separating out each part of a SELECT clause into dedicated parsing nodes.

Another interesting approach was to formalise a SQL SELECT clause as a sketch or template containing the SELECT, WHERE and AND SQL keywords, with additional "*tokens starting with "$" indicate the slot to be filled. The name following the "$" indicates the type of the prediction. For example, the $AGG slot can be filled with either an empty token or one of the aggregation operators, such as SUM and MAX. The $COLUMN and the $VALUE slots need be filled with a column name and a sub-string of the question respectively. The $OP slot can take a value from {=, <, >}*" [10]. This is another example that demonstrates opportunity for NLP success where the formal grammatical structure, corpus and domain is well understood and limited in scope. Given the time-bounded nature of the assignment, I opted to employ a more restrained design choice with NLTK feature-based context-free grammars handling the majority of the SQL statement parsing.

In this report, section 2 details the system key features, components and design, whilst section 3 illustrates example human NLQ with SQL output response from the solution itself. Section 3 provides a comparison of this solution developed in Python with the sample Prolog MS-DOS solution and closes with final thoughts.

2. The System

Key Features

The goal is to solve the problem of connecting a natural language query in human form relating to movies and actors to a database that persists the corpus and provide the user with information relevant to the question they asked. To achieve this the designed system has the following key features:

- **Connection to a MySQL database** with sample movie and actor information
- **Grammar rule collection** that formally models typical corpus search question sentences
- **Integration with WordNet** [3], a well-known lexical database, to expand the set of successful human language queries by using its synonym sets to improve the grammar rule successful parse rate
- **Handling of mixed-case NLQ entry** enables input of "MOvIe" to match the grammar rule production lexicon "movie", and actress name input "*Mae Hoffman*" to match the database persisted uppercase "*MAE HOFFMAN*"
- **Handling of variety of complexity in NLQ**, from simple film title e.g. "*ARMY FLINTSTONES*", to "*display movies*" and "*show me a documentary movie about crocodile or monkey***"**
- **Robust handling of parse and database query failure**

In addition, the solution provides comprehensive logging to facilitate development and demonstration.

Key Custom Modules

Provided below is an outline of the four main solution custom-developed modules, with a more formal, detailed illustration in figure 1.

The main module

- Responsible for orchestrating the overall solution and instantiation of the database and grammar_config modules
- Contains the Human Question / SQL Answer loop
- Sets the logging configuration, including log level (info, critical), from the logging.conf yaml file

The database module

- Manages connection to the MySQL database
- Executes the SQL query as parsed by a grammar rule
- Outputs the SQL query result set

The grammar_config module

- Loads the grammar rules from the grammars.csv configuration file
- Loads the stopwords from the stopwords.csv configuration file
- Builds a set of synonyms for each right-hand side terminal lexicon in every grammar configuration file
- Outputs the synonym set to file synonym_generated_set.csv for test, documentation and demonstration purposes

The parse module

- If the NLQ conforms to the specification of a movie title returns a well-formed SQL query
- Transforms all NLQ keywords (excluding actor name) to lower-case to facilitate grammar rule matching
- If the NLQ contains actor names, name is converted to upper case, stored and replaced with #STARRING tag to facilitate rule matching, with name re-injected later
- If the NLQ contains "about" criteria a SQL WHERE predicate with value constraints is formed for later appending to a successful grammar rule parse
- Uses word tokens from the user supplied NLQ to match against a synonym-grammar rule word pair within the earlier compiled synonym set, to construct a second NLQ with lexicons known to be specified in the grammar rules
- Loops through each grammar rule, with the NLTK parser attempting to fully resolve the NLQ input sentence (original and synonym matched) according to the grammar rule specification
- Parsing success returns well-formed SQL query or fail code for error handling

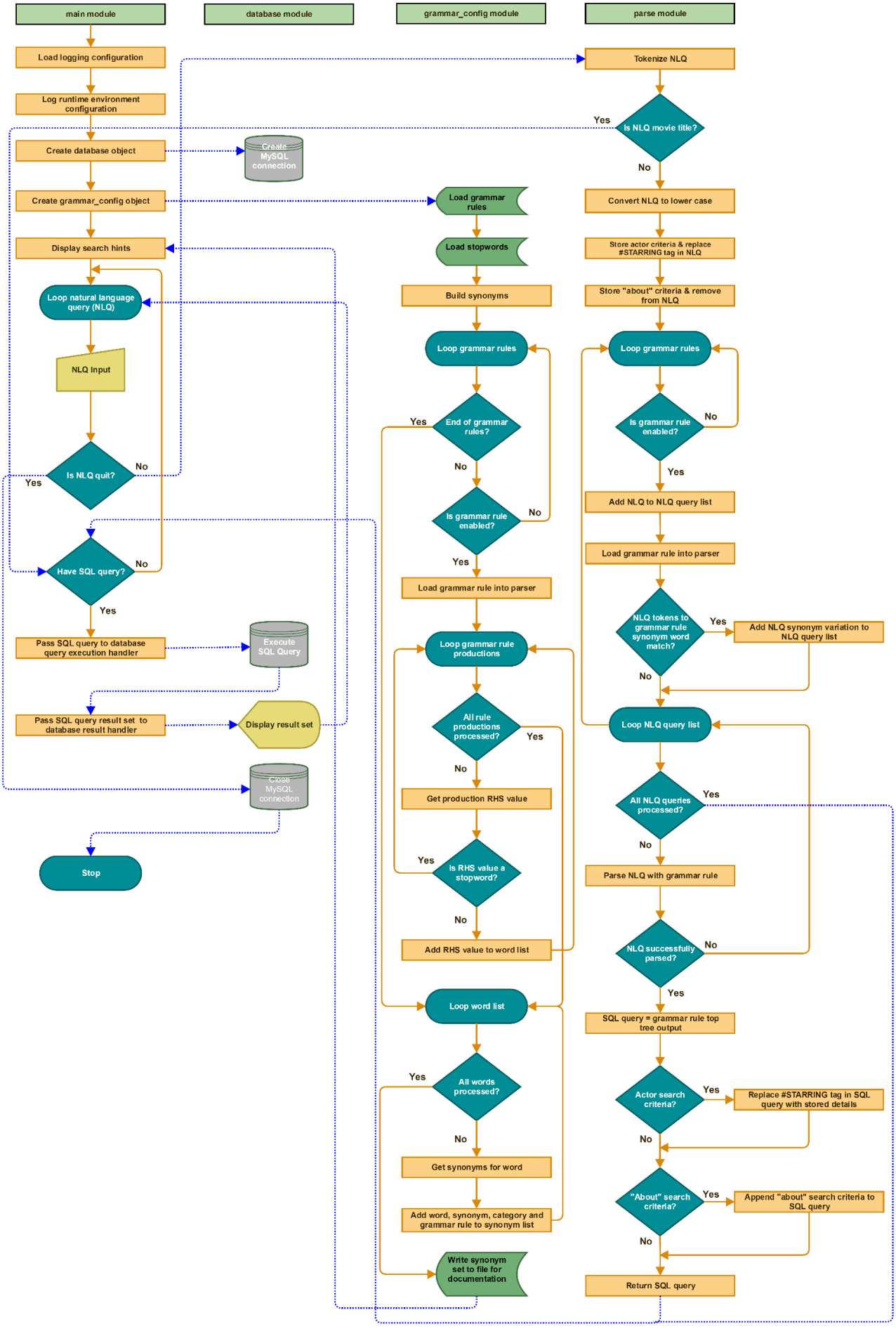


Figure 1: Application logic flow across the four main custom modules during human NLQ to formal SQL response

Solution Architecture and Specification

Figure 2 below provides a complete overview of all components with versions used in the solution.

Figure 2: Solution Architecture and Specification

The MySQL Community server is a free open source database available here [11]. It comes complete with the sample sakila database which includes the film_list and actor_info views, shown below in Figure 3. The solution consumes these to provide the movie domain corpus. Note the capitalisation of "first_name" and "last_name" columns in the actor_info view, also seen in the "title" and "rating" columns of view film_list (but not column "category"). The solution manages these data characteristics.

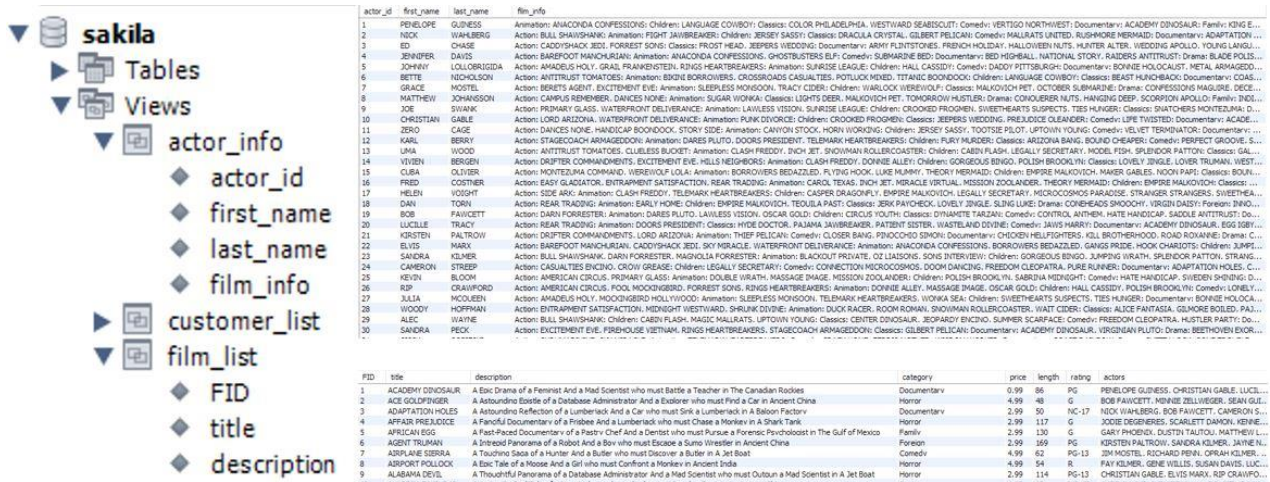


Figure 3: MySQL Sakila database view structure and sample data content

Grammar Rule Design & Build

The solution uses feature-based context-free grammar (FCFG) to formally and explicitly describe typical natural English language sentences used to request movie information. CFG is described as "*In formal language theory, a context-free grammar (CFG) is a certain type of formal grammar: a set of production rules that describe all possible strings in a given formal language. A context-free grammar provides a simple and mathematically precise mechanism for describing the methods by which phrases in some natural language are built from smaller blocks, capturing the "block structure" of sentences in a natural way. Its simplicity makes the formalism amenable to rigorous mathematical study. Important features of natural language syntax such as agreement and reference are not part of the context-free grammar, but the basic recursive structure of sentences, the way in which clauses nest inside other clauses, and the way in which lists of*

adjectives and adverbs are swallowed by nouns and verbs, is described exactly" [12]. The "feature" aspect comes by way of being able to assign properties to a syntactic category, like the noun "film" is singular whilst "films" is plural and these properties can be used to constrain the unification of the left and right-hand side of the rules in order to determine where a SQL SELECT statement should select one or all rows from the film_list database table.

Let's look at formalising an English sentence into an FCFG by considering a simple example: "*show me drama films*". Figure 4 shows this sentence repeatedly broken down into smaller units such as "*show films*" and "*show drama films*" that still retain grammatical sense Each unit is syntactically classified with part of speech (POS) tags (VB verb, PRP pronoun, NNS noun common plural etc) [13] [14] [15].

VB show	**PRP** me	**JJ** drama	**NNS** films
VP show		**JJ** drama	**NNS** films
VP show		**NP** films	

Figure 4: Sentence constituent units with part-of-speech tagging

The overall sentence structure with each of its constituent parts can be transformed into a standard phrase structure tree by flipping the above table over and adding a parent constituent S (sentence) to the constituent VP (verb phrase) and NP (noun phrase) units, as shown in Figure 5. By assigning SQL grammar to appropriate natural language units we can see how to both represent the meaning of natural language sentence and parse it into formal SQL.

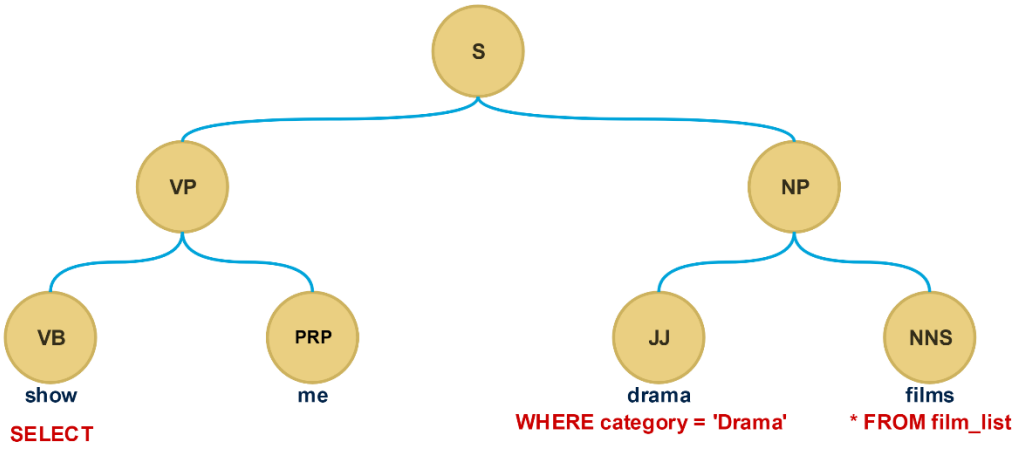


Figure 5: Standard phrase structure tree for sentence "show me drama films"

With the above preparation done, the NLTK fcfg grammar can now be declaratively formalised, building a grammar rule able to parse the sentence "*show me drama films*" and generate the equivalent SQL statement, guided by the phrase structure tree to notate the formal model. Each line is a production, with a left-hand side (LHS) and right-hand side (RHS), separated by the symbol ->. The LHS NNS production is satisfied by the RHS terminal lexicon "*films*", the LHS JJ production by the RHS terminal lexicon "*drama*", the LHS PRP production by the RHS terminal lexicon "*me*" and the LHS VB production by the RHS terminal lexicon "*show*".

Taking the first sentence token "*show*", we have a match in the LHS VB production, and the lexical value "*show*" is passed up the tree via the SEM (semantic) feature and instantiates the variable ?vb in RHS VB of production LHS VP. In the same way, the LHS PRP production is satisfied with "*me*", the empty value (effectively filtering out "me" as we are not concerned with it) flowing up the tree to RHS PRP in production LHS VP, which becomes fully specified and assumes the value of "*SELECT*", in turn flowing upwards to RHS VP production to build LHS S for parsing of the formal SQL statement.

```
% start S
S[SEM=(?vp + ?np)] -> VP[SEM=?vp] NP[SEM=?np]
VP[SEM=(?vb)] -> VB[SEM=?vb] PRP[SEM=?prp]
NP[SEM=(?nns + ?jj)] -> JJ[SEM=?jj] NNS[SEM=?nns]
VB[SEM='SELECT'] -> 'show' | 'display' | 'list' | 'give' | 'tell'
PRP[SEM=''] -> 'me'
JJ[SEM="WHERE category = 'Drama'"] -> 'drama'
NNS[SEM='* FROM film_list'] -> 'films'
```

The earlier grammar notation example is simple for purposes of illustration. However, to enable a compact grammar collection, single grammar definitions can be specified so that individual LHS productions can be satisfied by one of several RHS sentence structures and

therefore support several human language structures. The example showed the VP production requiring the RHS productions VP ("*show*") and RHS PRP ("*me*") to be satisfied. But what if the NLQ is "*show drama films*" without the pronoun "*me*"? This is resolved by specifying the verb phrase as satisfied by both the verb and pronoun, or just the verb itself, as per below, using the pipe | symbol as a disjunction to specify VP -> VB PRP **OR** VP -> VB.

```
VP[SEM=(?vb)] -> VB[SEM=?vb] PRP[SEM=?prp] | VB[SEM=?vb]
```

The solution also adds the feature property NUM to some grammar rules to specify nouns marked as singular (NUM=sg) or plural (NUM=pl), allowing correct grammatical specification of lexical words "*film*" and "*films*", and resolving SQL selection of 1 row or many. The example below shows the LHS NP production constrained with NUM=sg and will be satisfied only if the RHS production NNS propagates up the feature property NUM with the value sg.

```
NP[SEM=(?nns + 'ORDER BY RAND() LIMIT 1')] -> NNS[SEM=?nns, NUM=sg]
..
..
NNS[SEM='title, description, category, rating, actors FROM film_list', NUM=sg] -> 'film'
NNS[SEM='title, description, category, rating, actors FROM film_list', NUM=pl] -> 'films'
```

The table in Appendix A shows how I've built a domain language for movie and actor search with a collection of grammar rules. Examples of supported NLQ with the specific grammar rule parsing the sentence and formal SQL generated are given.

Parsing

We have seen the formal notation of a feature-based context free grammar (FCFG) in the previous section. The custom python module parse.py handles parsing of the NLQ sentence by looping through each grammar specification (over a dozen in the solution) until the outcome is a successful parse with generated SQL ready for database execution, otherwise a fail code that is appropriately managed.

Prior to any sentence parsing 7 preparation tasks are performed:

1. Take each grammar rule RHS lexicon and lookup the WordNet synonym set, store as a grammar rule word-synonym pair
2. Tokenize the NLQ sentence
3. Check for a stand-alone movie title sentence which is converted to upper-case prior to SQL query
4. Convert NLQ to lowercase to assist satisfying the grammar production rule RHS lexicons which are specified lowercase
5. Check for actor name, convert to upper-case, replace with #STARRING tag prior to parsing
6. Remove "*about*" token and remainder of sentence string, which indicates search on film description column
7. Build a second NLQ sentence using the grammar rule-synonyms pairs, replacing the word tokens provided by the user (synonym) with lexicons (words) known to the grammar rule via a grammar rule word-synonym lookup

Following a successful parsing, the following post-parse tasks are performed:

1. Any actor name is injected into the parsed SQL statement by replacing the #STARRING tag
2. Any "*about*" keywords appended to SQL statement as WHERE clause predicate values, handling "AND" and "OR" operators

Figure 1 earlier in the report shows the full application logic flow for sentence parsing. FCFGs in NLTK are parsed with an Earley Chart parser [16]. To see how this parser works, let's again use the NLQ "*show me drama films*" as an example. The parser puts all the sentence word tokens into a dynamic table data structure as shown below:

	Col 1	Col 2	Col 3	Col 4
Row 0	'show'			
Row 1		'me'		
Row 2			'drama'	
Row 3				'films'

The VB verb phrase from the grammar notation is satisfied by the word in [0:1] and becomes [0:1] VB[SEM='SELECT'] -> 'show' *. As the VB is satisfied the VP phrase can now be partially satisfied: VP[SEM=(?vb)] -> VB[SEM=?vb] * PRP[SEM=?prp] {?vb: 'SELECT'}, assigning the value 'SELECT' that has been propagated up from VB to variable ?vb. The leftmost edge of the tree in Figure 5 has now been fully traversed, so the parser now moves to column [1:2] with word "*me*". This PRP column is satisfied with the RHS lexical "*me*" and propagates the empty SEM value up to RHS PRP variable ?prp of LHS VP, which now becomes fully satisfied. As VP is fully satisfied, S becomes partially satisfied with the value of VP, and column [0:2] is updated with the resolved grammar so far. Here's the dynamic table looks so far, with the parser visibly resolving half of the sentence, making its way from left to right, bridging column 1 to 2.

	Col 1	Col 2	Col 3	Col 4
Row 0	VB[SEM='SELECT'] -> 'show' *	S[SEM=(?vp+?np)] -> VP[SEM=?vp] * NP[SEM=?np] {?vp: (SELECT)}		
Row 1		PRP[SEM=''] -> 'me' *		
Row 2			'drama'	
Row 3				'films'

Word "*drama*" in column [2:3] is parsed with LHS production JJ by RHS terminal lexicon "*drama*", which means that the NP production can be partially satisfied. Values for columns [0:2] and [2:3] can be updated:

	Col 1	Col 2	Col 3	Col 4
Row 0	VB[SEM='SELECT'] -> 'show' *	S[SEM=(?vp+?np)] -> VP[SEM=?vp] * NP[SEM=?np] {?vp: (SELECT)}		
Row 1		PRP[SEM=''] -> 'me' *		
Row 2			NP[SEM=(?nns+?jj)] -> JJ[SEM=?jj] * NNS[SEM=?nns] {?jj: "WHERE category = 'Drama'"}	
Row 3				'films'

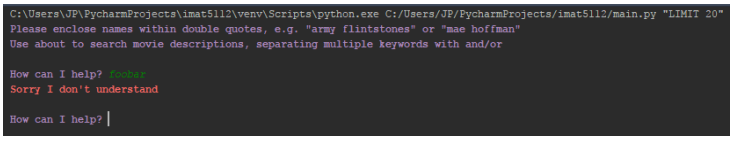
Word "*films*" in column [3:4] is parsed with LHS production NNS by RHS terminal lexicon "*films*", which means that the NP production can be fully satisfied. As both the VP and NP are fully satisfied, the sentence S can be fully satisfied and hence parsed successfully though the grammar, with column [0:1] and [0:4] bridged. The final state of the table is as follows:

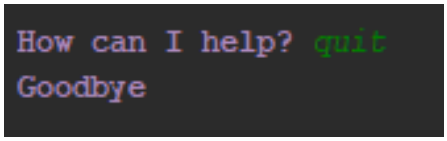
	Col 1	Col 2	Col 3	Col 4
Row 0	VB[SEM='SELECT'] -> 'show' *	S[SEM=(?vp+?np)] -> VP[SEM=?vp] * NP[SEM=?np] {?vp: (SELECT)}		S[SEM=(SELECT, * FROM film_list, WHERE category = 'Drama'))] -> VP[SEM=(SELECT)] NP[SEM=(* FROM film_list, WHERE category = 'Drama'))] *
Row 1		PRP[SEM=''] -> 'me' *		
Row 2			NP[SEM=(?nns+?jj)] -> JJ[SEM=?jj] * NNS[SEM=?nns] {?jj: "WHERE category = 'Drama'"}	NP[SEM=(* FROM film_list, WHERE category = 'Drama')] -> JJ[SEM="WHERE category = 'Drama'"] NNS[SEM='* FROM film_list'] *
Row 3				NNS[SEM='* FROM film_list'] -> 'films' *

I have demonstrated the Earley parser is a hybrid parser, breadth first top-down parser but incorporating a bottom up value propagation element, as parsing begins with the first token "show" in row [0:1] but passes that value up the parse tree to production VP so that it can be partially resolved. It then moves horizontally right and vertically down to row [1:2] to parse token "me" but again propagates the value up the parse tree to fully resolve the VP production.

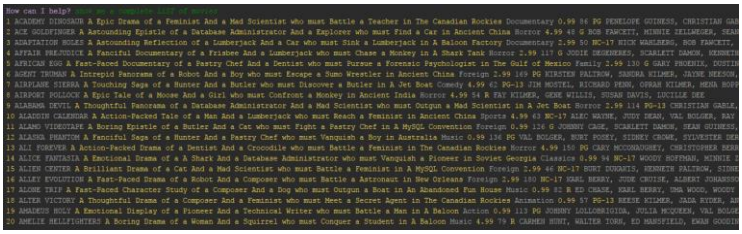
Solution Working Examples

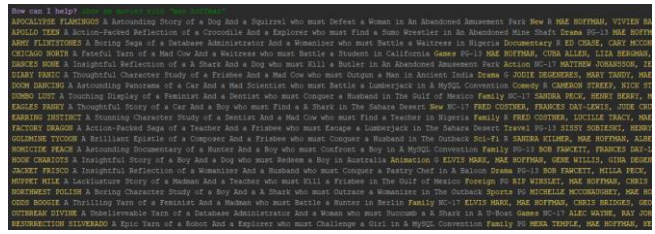
The solution runs in the Python shell, with keyboard used for input. Upon launch search tips are suggested, with elegant handling both of NLQ sentences that no grammar can parse and any ill-formed SQL statements that raise a database exception, providing a robust solution – see screenshot bottom left. For test and demonstration purposes the SQL result set limit is restricted to 20 rows, with the limit number maintainable in the application runtime configuration parameters. Ending the application is with the "quit" command.



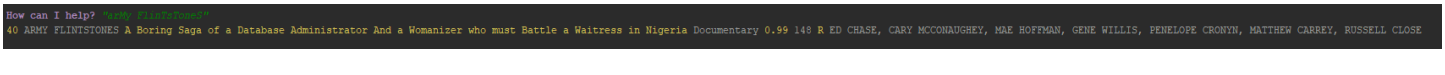


The example below left shows the response for "*show me a complete LiST of movies*", with "LiST" converted to lower-case to conform to lower case characteristics of grammar rule RHS terminal productions. A simple alternating column colour scheme is used to aid readability. Example below right shows response for *show me movies with "mae hoffman*", with parser conversion of name to upper case.

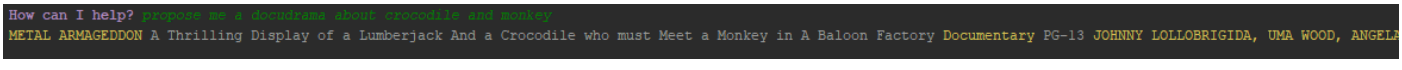




The example below shows a simple movie title request enclosed within double quotes. Any movie titles will be converted to upper-case to match the data characteristics of the persisted movie titles in the database.



The example below shows illustrates a more complex example, "*propose me a docudrama about crocodile and monkey*". The category "docudrama" is not any grammar RHS lexical terminal production, but because the synonym set contains a docudrama-documentary pair, and documentary is a RHS lexical terminal production, we can successfully parse the NLQ and return information to the user. The table in Appendix A shows a comprehensive list of typical NLQ sentences that can be successfully parsed.



Comparison with Prolog MS-DOS application, and Final Thoughts

Prolog is a logic programming language based on first order logic, or predicate logic, where the "what" is defined and not any procedural "how's". Python on the other hand provides greater flexibility in program flow with coding styles ranging from simple top-down imperative to use of functions and classes for abstraction and encapsulation. Like SWI-Prolog, Python code is interpreted at runtime. Both languages have their own peculiarities that catch newcomers out. Prolog stipulates clauses for a predicate are grouped together, whereas Python is fussy about indentation to the left of statements which is used to define code blocks.

Similarities can be drawn between Prolog and the NLTK parsing of context free grammars with the Earley Parser (described earlier). The Prolog MS-DOS template has its knowledge base with sets of facts and conditions to be satisfied. My Python with NLTK solution also has a set of facts with conditions in the form of grammar notation. Both have the same objective of unifying a goal on the left side of a rule (NLTK production) with the satisfaction of the right-hand side clauses (productions). Enhancing the knowledge base in Prolog requires programming expertise, whereas context free grammars are independent of Python. In theory the grammars could be developed by non-programmers provided the notation specification is understood. With NLTK grammars I do not need to be concerned with cuts and fail statements. Similar to Prolog, in NLTK grammars recursion and backtracking can occur, when for example, the LHS production noun requires feature singular when plural "films" is passed, so the parser will backtrack and look for another LHS production noun with feature plural. Another similarity is the availability of a first-order logic parser within NLTK, so showing movies by an actor could be defined in NLTK as the predicate *show* with two arguments, for example show(movies,'Mae Hoffman').

One advantage Prolog has is the straight-forward ability to modify the knowledge base with assert. I did consider some form of automatic rule generation, based on synonym matching and learning from previous user NLQ search. However, I don't believe this would be so elegant, possibly having to dynamically create new grammars on the fly via the file system. I'm sure with further research a method could be found given the popularity of Python for machine learning.

For this assignment I wanted to incorporate an additional feature of substance that delivered real value to the solution rather than for example replacing one input method for another (e.g. speech recognition in place of keyboard). I elected to incorporate synonym sets from WordNet [3] into my solution as an added feature, to provide a simple means of expanding the domain language whilst keeping the rule definitions compact. In Python this was simply done by adding the NLTK package to my project and then importing it into my custom code where needed. Working with the synonym sets was as easy as 1 line of code to loop the set, passing a word (e.g. "film") as an argument to return associated synonyms. As far as I understand, integration of WordNet into Prolog requires file download and understanding of the file structure as no interface is provided [17].

I appreciate the Prolog MS-DOS program is a template, but perhaps also due to the SWI-Prolog development environment it is not stable, and it is easy to crash the program. In contrast, with the use of try-catch exception handling my solution is very robust.

To directly answer the question "*which language would you use if a real world system like this was assigned to you?*" Without hesitation Python would be my choice over Prolog. In November 2017 Joseph Sirosh, Corporate Vice President, Microsoft Cloud AI, visited my employer Nestle S.A. for a seminar. A colleague asked him the question "*So do you think that Prolog has a place in AI going forward?*". His response was "*No, Prolog is dead*".

My preference in Python is based on a number of considerations. First, within a practical industry perspective the pillars that make AI useful and available are Cloud and Big Data, where Python does very well in integration terms whilst of course very powerful for developing the algorithms themselves that power A.I. Some practical business considerations need to be made. From a practical business. Second, there are simply not enough developers with Prolog programming expertise that can adequately support any design and build or sustain service line for a large enterprise, and this will not change. On the other hand, as commented earlier, Python has a huge and very active developer community, and a broad range of literature materials, not be underestimated as powerful supporting educational tools. Third, with speed to market of solutions is critical in helping business maintain competitive edge. With 125850 open source packages available [18] at time of writing, with some examples being PyTorch [19] for tensors and dynamic neural networks, to Pendulum [20] for working with dates and time, it makes sense to take advantage of open source, readily available solutions. Fourth and finally, it's important for the developer to be supported with a strong development environment in order to have an enjoyable, supported experience with an efficient workflow. Personally I use PyCharm [21], an excellent modern and feature-rich IDE with code completion, automatic code refactoring, a built-in Python console, coding tips and a visual debugger, that is updated monthly.

Closing with some thoughts on future improvements to my solution, I would look at abstracting out each main component of the SQL language corpus like the SELECT, WHERE and function (AVG, MIN) clauses into their own grammar rules to simplify any future expansion of the NLQ language domain which will no doubt raise the complexity of SQL statements to be generated. I would also look to make the solution more database agnostic and remove any hardcoding of tables and column definitions in the grammar rules themselves. One further enhancement would be handling of multiple NLQ words to one rule RHS lexical value, for example matching input of "*motion picture*" with rule word "*film*". NLTK does not except multi-word lexical items, they must be joined with an underline e.g. "motion_picture". There is opportunity to improve the user experience with search suggestions following an NLQ failure, and should not be so difficult to achieve, as I already have at hand the NLQ input sentence tokenized and so use this in an intelligent way, perhaps with some probability algorithm or weighted grammars, to make sensible suggestions.

References

[1] "Python.org," [Online]. Available: https://www.python.org/.

[2] "Natural Language Toolkit," [Online]. Available: http://www.nltk.org/.

[3] "WordNet - a lexical database for English," [Online]. Available: https://wordnet.princeton.edu/.

[4] "Interactive: The Top Programming Languages 2017," [Online]. Available: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017.

[5] "IEEE Top Programming Languages: Design, Methods, and Data Sources," [Online]. Available: https://spectrum.ieee.org/ns/IEEE_TPL_2017/methods.html.

[6] "stackoverflow," [Online]. Available: https://stackoverflow.com/.

[7] D. Robinson, "The Incredible Growth of Python," 06 09 2017. [Online]. Available: https://stackoverflow.blog/2017/09/06/incredible-growth-python/.

[8] "Wiki NLTK," [Online]. Available: https://en.wikipedia.org/wiki/Natural_Language_Toolkit.

[9] F. Li and H. Jagadish, "Understanding Natural Language Queries over Relational Databases," *ACM SIGMOD Record,* vol. 45, no. 1, pp. 6-13, 2016.

[10] X. Xu, C. Liu and D. Song, "SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning," 2017.

[11] "MySQL Server Community Downloads," [Online]. Available: https://dev.mysql.com/downloads/.

[12] "Wiki Context-free grammar," [Online]. Available: https://en.wikipedia.org/wiki/Context-free_grammar.

[13] "Wiki Syntactic Category," [Online]. Available: https://en.wikipedia.org/wiki/Syntactic_category#Lexical_categories_vs._phrasal_categories.

[14] "Stackoverflow part of speech tags NLTK," [Online]. Available: https://stackoverflow.com/questions/15388831/what-are-all-possible-pos-tags-of-nltk.

[15] "Wiki Part of speech tagging," [Online]. Available: https://en.wikipedia.org/wiki/Part-of-speech_tagging.

[16] "Wiki Earley Parser," [Online]. Available: https://en.wikipedia.org/wiki/Earley_parser.

[17] "WordNet with Prolog," Princeton University, [Online]. Available: https://wordnet.princeton.edu/wordnet/man/prologdb.5WN.html.

[18] "Python Package Index," [Online]. Available: https://pypi.python.org/packages.

[19] "PyTorch," [Online]. Available: http://pytorch.org/.

[20] "Pendulum," [Online]. Available: https://pendulum.eustace.io/.

[21] "PyCharm," [Online]. Available: https://www.jetbrains.com/pycharm/.

Appendix A - Example NLQ sentences, associated handling grammar rules and generated SQL

User NLQ	Synonym Matching	Rule Ref / Method	Comments	SQL Generated
"ARMY FLINTSTONES"		movie_title()	Simple movie title request respecting case	`SELECT * FROM film_list where title = 'ARMY FLINTSTONES'`
"army flintstones"		movie_title()	Title to uppercase	`SELECT * FROM film_list where title = 'ARMY FLINTSTONES'`
Complete list of film	film -> films	001	Mixed case, synonym match	`SELECT title, description, category, rating, actors FROM film_list`
consummate listing of celluloid	consummate -> complete listing -> list celluloid -> films	001	Multi synonym match	`SELECT title, description, category, rating, actors FROM film_list`
MOVIE llsT		002	Singular NLQ. Handles mixed case. 1 random movie	`SELECT title, description, category, rating, actors FROM film_list ORDER BY RAND() LIMIT 1`
movies list		002	Plural NLQ - all movies	`SELECT title, description, category, rating, actors FROM film_list`
pic listing	pic -> films listing -> list	002	plural - all movies	`SELECT title, description, category, rating, actors FROM film_list`
show me a complete list of movies		003	NQL fully matches rule RHS	`SELECT * FROM film_list`
give me a full list of films		003	2nd example NLQ fully matches rule RHS	`SELECT * FROM film_list`
display me a broad listing of film	broad -> full listing -> list film -> films display -> show	003	Multi-synonym match	`SELECT * FROM film_list`
display a broad listing of film	broad -> full listing -> list film -> films display -> show	003	Handles different syntactical structure from previous example – No PRP pronoun “me”. Rule handles multiple verb phrase structures	`SELECT * FROM film_list`
give me the full list of actors		003	Handles different NNS (noun common plural), with “actors” in place of “movies”, and selects correct database view	`SELECT * FROM actor_info`
present me the entire list of actors	present -> show entire -> full	003	Handles actor selection with synonym matching	`SELECT * FROM actor_info`
show films rated pg13		004	Handles pg13 lowercase/without hyphen & resolves to PG-13 as per database. Also handling of no PRP pronoun “me”	`SELECT title, description, category, rating, actors FROM film_list WHERE rating = 'PG'`
give me films rated r		004	Handles PRP 'me'	`SELECT title, description, category, rating, actors FROM film_list WHERE rating = 'R'`
display film rated g	film -> flicks display -> show	004	Synonym matching	`SELECT title, description, category, rating, actors FROM film_list WHERE rating = 'G'`
what films are rated PG		005	Distinguishes NLQ plural request	`SELECT title, description, category, rating, actors FROM film_list WHERE rating = 'PG'`
which film is rated pg		005	Distinguishes NLQ singular request	`SELECT title, description, category, rating, actors FROM film_list WHERE rating = 'PG' ORDER BY RAND() LIMIT 1`
show me horror films		006	Distinguishes NLQ plural request	`SELECT title, description, category, rating, actors FROM film_list WHERE category = 'Horror'`
list of drama movies		006	NLQ fully matches rule RHS	`SELECT title, description, category, rating, actors FROM film_list WHERE category = 'Drama'`
give me the international movies		006	NLQ plural request, PRP pronoun “me” & DT determiner “the” handled	`SELECT title, description, category, rating, actors FROM film_list WHERE category = 'Foreign'`
give me a modern film	film -> movie modern -> new	006	Singular. Proposes 1 film, synonym matching	`SELECT title, description, category, rating, actors FROM film_list WHERE category = 'New' ORDER BY RAND() LIMIT 1`
give me an actor		006	NLQ singular request. Different NNS (actor)	`SELECT first_name, last_name, film_info FROM actor_info ORDER BY RAND() LIMIT 1`
Give me the catelog of films		007	Pronoun and determiner use, plural, mixed-case	`SELECT title, description, category, rating, actors FROM film_list`
show a listing of film	listing -> list film -> movies show -> display	007	Multi-synonym matching	`SELECT title, description, category, rating, actors FROM film_list`
show me the database of actors		007	Different NNS (actor)	`SELECT first_name, last_name, film_info FROM actor_info`
films rated Pg		008	Handles mixed-case “Pg”	`SELECT title, description, category, rating, actors FROM film_list WHERE rating = "PG"`
cinema classified r	cinema -> films	008	Synonym matching	`SELECT title, description, category, rating, actors FROM film_list WHERE rating = "R"`

show me movies with "MAE HOFFMAN"		009	Handles actor naming using #STARRING tag feature	`SELECT title, description, category, rating, actors FROM film_list WHERE actors LIKE '%MAE HOFFMAN%'`
list movies starring "mae hoffman"		009	Handles no PRP pronoun unlike previous example. Handles lower case actor name	`SELECT title, description, category, rating, actors FROM film_list WHERE actors LIKE '%MAE HOFFMAN%'`
see all "mae hoffman" films	see -> find	009	Handles actor naming using #STARRING tag feature, synonym matching. Note handling difference in location of actor name	`SELECT title, description, category, rating, actors FROM film_list WHERE actors LIKE '%MAE HOFFMAN%'`
find drama movies with "MAE HOFFMAN"		011 & parse.starring()	Handles combined category/actor NLQ criteria	`SELECT title, description, category, rating, actors FROM film_list WHERE category = 'Drama' AND actors LIKE '%MAE HOFFMAN%'`
get docudrama flick starring "MAE HOFFMAN"	get -> find docudrama -> documentaries flick -> films	011 & parse.starring()	Synonym matching in addition to previous example	`ELECT title, description, category, rating, actors FROM film_list WHERE category = 'Documentary' AND actors LIKE '%MAE HOFFMAN%'`
movies about japan		012.fcfg & parse.about()	Simple NLQ with description keyword	`SELECT title, description, category, rating, actors FROM film_list WHERE description LIKE '%japan%'`
drama movie about Japan		012.fcfg & parse.about()	Adds category to previous example	`SELECT title, description, category, rating, actors FROM film_list WHERE category = 'Drama' AND description LIKE '%japan%'`
propose me a docudrama about crocodile and monkey	propose -> suggest docudrama -> documentaries	012.fcfg & parse.about()	Synonym matching with category and multiple description keyword with AND conjunction. Note absence of noun (e.g. movie)	`SELECT title, description, category, rating, actors FROM film_list WHERE category = 'Documentary' AND description LIKE '%crocodile%' AND description LIKE '%monkey%'`
show me a documentary movie about crocodile or monkey		012.fcfg & parse.about()	Multiple description keyword with OR conjunction. Note inclusion of noun movie compared to previous example	`SELECT title, description, category, rating, actors FROM film_list WHERE category = 'Documentary' AND description LIKE '%crocodile%' OR description LIKE '%monkey%' LIMIT 20`