# Parsing Database Queries in Natural Languages: A Purely Empirical Method

**Li, Ziyao**
1500017776 / Yuanpei College
leeeezy@pku.edu.cn

## Abstract

This paper is the Project 2 of *Course EMNLP*. The main task is to realize a model to parse given database queries, which are in natural language forms, to an operable form. Literally, the given parser form do have a lot of differences than the first-order logic (FOL) forms. The model leveraged several hand-crafted rules according to the observations of the data and an Inductive Logic Programming framework proposed in CHILL [1]. Rules for different predicates are come up with, as well as several language phenomena are encoded. A 47.48% *Exact Accuracy*, as demanded by the project, over database geo800 was achieved.

## 1 Introduction - Datasets and Problem Definition

Instead of a semantic parsing project, the data provided in this project is more like a down-stream Question-Answering task. A functional form of data is provided instead of the $\lambda$-calculus formulas. Therefore, applying first-order logic (FOL)-based algorithms seems to be even more laborious, given the fact that all these algorithms are sometimes artificially established and delicately elaborated, and that due to the complication of the system, the implementations of algorithms are not thoroughly described in short conference papers.

Figure 1 is an example of the query data. The annotated labels of instances are actually old-fashioned *perl* codes. The $:-$ operator appearing in the data is inductions, or to say, clauses. These induction examples, combined with a similarly defined database query language system and hundreds of control rules and basic induction rules, are directly used to transform a natural language sentence into a value (or a list of values) in the given database.

What is the capital of the state with the largest population?

```
answer(C, (capital(S,C), largest(P, (state(S), population(S,P))))).
```

What are the major cities in Kansas?

```
answer(C, (major(C), city(C), loc(C,S), equal(S,stateid(kansas)))).
```

What state has the most rivers running through it?

```
answer(S, most(S, R, (state(S), river(R), traverse(R,S)))).
```

How many people live in Iowa?

```
answer(P, (population(S,P), equal(S, stateid(iowa)))).
```

Figure 1: An instance $(Input, Query)$, or $(parse, answer)$ pair of the data. A connection between the predicates in the language and the predicates in the logical form can be somehow inducted from a set of lexicon.

The original data and codes (the database and some of the complementary descriptions) are provided by Raymond J. Mooney from University of Texas, Austin [2]. To solve the problem defined above, an Induction Logical Programming (ILP)-based system was proposed in (J. Zelle & R. Moorey, 1996)[3], along with a more detailed and descriptive illustration proposed in (J. Zelle & R. Moorey, 1997)[4].

[1] Zelle, J. M. and R. J. Moorey, Learning to Parse Database Queries Using Inductive Logic Programming, In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI)*, Aug 1996

[2] ftp://ftp.cs.utexas.edu/pub/mooney/chillin/

[3] Zelle, J. M. and R. J. Moorey, Learning to Parse Database Queries Using Inductive Logic Programming, In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI)*, Aug 1996

[4] Zelle, J. M. and R. J. Moorey, An Inductive Logic Programming Method for Corpus-based Parser Construction, Technical Report, 1997

## 2 CHILL - Automatically Induction Rules Generator

CHILL (Constructive Heuristics Induction for Language Learning) is still a rule-based system, while its rule-generation mechanism is not by human forces but by an induction framework elaborated according to the structure of the problem. Figure 2 shows the total structure of CHILL. The starting point of CHILL, however, is still a lexicon created manually of a mapping from natural language words to logical predicates.

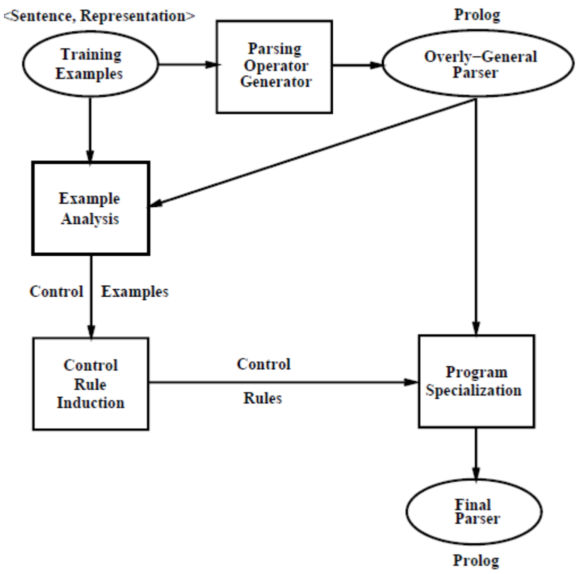


Figure 2: A brief structure demonstration of CHILL.

### 2.1 Parsing Framework

The parsing framework defined in CHILL is perhaps the most ingenious part of the paper. The parsing process is considered as a sequential one incorporating the state-transformation of the parser stack and the sentence queue. Several basic operations can be defined over this framework. As for this database query parsing problem, five basic actions, namely INTRODUCE, COREF_VARS, SHIFT, LIFT_CONJ and DROP_CONJ are defined.

INTRODUCE pushes a predicate onto the stack based on a word appearing in the input and information about its possible meanings in the lexicon. COREF_VARS binds two arguments of two different predicates on the stack. DROP_CONJ (or LIFT_CONJ) takes a predicate on the stack and puts it into one of the arguments of a meta-predicate on the stack. SHIFT simply pushes a word from the input buffer onto the stack.

Figure 3 is an example to illustrate the whole process. This structure is the inspiration of the rule-based model proposed in this paper.

```
      Parse State                                         Operation Type

1.  ps([answer(_,_):[]],
       [what,is,the,capital,of,texas,?])                  shift
2.  ps([answer(_,_):[what]],
       [is,the,capital,of,texas,?])                       shift
3.  ps([answer(_,_):[is,what]],
        [the,capital,of,texas,?])                         shift
4.  ps([answer(_,_):[the,is,what]],
       [capital,of,texas,?])                              introduce
5.  ps([capital(_,_):[],
        answer(_,_):[the,is,what]],
       [capital,of,texas,?])                              co-reference
6.  ps([capital(_,A):[],
        answer(A,_):[the,is,what]],
       [capital,of,texas,?])                              shift
7.  ps([capital(_,A):[capital],
        answer(A,_):[the,is,what]],
       [of,texas,?])                                      shift
8.  ps([capital(_,A):[of,capital],
        answer(A,_):[the,is,what]],
       [texas,?])                                         shift/introduce
9.  ps([equal(_,stateid(texas)):[texas],
        capital(_,A):[of,capital],
        answer(A,_):[the,is,what]],
       [?])                                               co-reference
10. ps([equal(B,stateid(texas)):[texas],
        capital(B,A):[of,capital],
        answer(A,_):[the,is,what]],
       [?])                                               conjoin
11. ps([equal(B,stateid(texas)):[texas],
        answer(A,capital(B,A)):[the,is,what]],
       [?])                                               shift
12. ps([equal(B,stateid(texas)):[?,texas],
        answer(A,capital(B,A)):[the,is,what]],
       [])                                                shift
13. ps(['EndOfInput',
        equal(B,stateid(texas)):[?,texas],
        answer(A,capital(B,A)):[the,is,what]],
       [])                                                conjoin
14. ps(['EndOfInput',
        answer(A,(capital(B,A),
                equal(B,stateid(texas)))):[the,is,what]],
       [])
```

Figure 3: An example of the parsing framework defined in CHILL.

### 2.2 Example Analysis & Induction Algorithms

In this part, CHILL take into account all possible emerging actions and states as the data space, considering the emerged ones in the dataset as Positive examples and those do not the Negative. The idea is to find minimized generalizations to reduce the structure of the positive samples while in the same time avoiding introducing negative samples. A least generalize generalization (LGG) is evaluated in all pairs of positive samples while a generalization algorithm is proposed to conduct the induction process.

After control rules are inducted, a Program Specialization combines the rules and the basic induction rules into a unified output parser. On the dataset the derived parser is the most compacted without introducing negative samples.

## 3 BroC Parser

However, implementing the task is really hard, especially not using *Perl* language (its powerful regular expression can save a lot of time). Therefore, by introducing several priors of the parsing framework we can have a easy-implementing algorithm - BroC Parser. The model is an *ad hoc* taking into account a lot of background knowledge.

Here are the basic priors for the parser.

- BroC Parser always `INTRODUCE` when the front of the sentence queue can be explained as a predicate in the lexicon.
- After `INTRODUCE`, BroC Parser always `SHIFT` all words used to `INTRODUCE` predicates. The shift process is simplified as a `pop` action.
- When `INTRODUCE` is not available, Broc Parser `SHIFT` one word from the sentence queue
- For each predicate, if it's a *meta* one, directly `LIFT_CONJ` it on the current mother *meta* predicate, and regard it as the mother *meta* predicate. If not, `LIFT_CONJ` it on the back of the current mother *meta* predicate. As `answer/2` is always the first predicate on the stack, reckon it as the first mother *meta* predicate so the process can always going on.
- Small adjustments are made according to the train set.

The lexicon defined for the predicate-recognition task is trivial and not very interesting, so as the detailed adjustments made in the very *ad hoc* project. The interested can take a look at the codes in `brocparser.py`.

## 4 Experiments

As the model is *ad hoc* and take overwhelming differences over different datasets, only `geo880` dataset is experimented. As this is a rule-based system tuned on the train set, the precision over the train set is still a little bit higher than that over the validation set. Table 1 shows the result.

An *Exact Accuracy* is examined as instructed by the project. However, taking such a measurement is unreasonable since the idea is never to produce identical logical forms by equivalent ones, ones that extract the same and therefore the correct responds to test queries. A great percentage of instances are provided with equivalent but different forms by the algorithm.

| | |
|---|---|
| Train | 0.5762 |
| Valid | 0.5444 |

Table 1: Exact Accuracy over `geo880` dataset.

## Appendix

Some of the codes are written with Jupyter Notebook, but are all laborious and redundant work. Codes are in `bash.py`, `brocparser.py` and `structures.py`. `bash.py` implements a basic file reading and writing job. `brocparser.py` is the main parser and `structures.py` provides basic tool functions and data structures used in the other two codes. Besides, `d1_train_myout.txt`, `d1_valid_myout.txt` and `d1_test_myout.txt` are results provided from the algorithm.

It is future work to clean up some redundant work in the RuleParser class and to implement better APIs. Limited by the strong priors, some of the code cannot correctly parse the idea, but can be fixed by altering the `COREF_VARS` principles, which should be easy to implement.