From Language to Logic: Leveraging Lambda Calculus in Computational Semantics

Jon Rusert

jonathan-rusert@uiowa.edu

May 2018

1 Introduction

The human language is immensely invaluable in our every day lives. Humans use it to express emotions, ideas, instructions, and much more. Language is useful in the creative side of humanity, seen in places such as books and songs, as well as the logical side of humanity, expressing research findings or explaining mathematical proofs. While we can seen the power natural language offers, we often can take for granted how well the human mind works at comprehending the technical meanings behind the language. This is sometimes more apparent however, when we focus on explaining technical ideas (e.g. logical proofs) to others or even trying to extend our ideas to computers.

Recall some early logic (in the form of Syllogism) from Aristotle:

All men are mortal.

Socrates is a man.

Therefore, Socrates is mortal.

As humans, we can easily observe why the third statement follows from the first two. However, imagine we want to allow a computer to make the same logical observation. Simply giving a computer the straight text might work if it advanced enough. The computer might be able to combine the first two statement into

"Socrates is a man are mortal.", where *man* is the combining point, but this then leaves us with a broken idea of what we set out to do. Instead, it would be better to pass the computer logical representations of the above statements for it to work with.

Higher-order logic (HoL) is often leveraged when aiming to represent natural language as logical statements. Observe the original logic again in HoL form:

$$\forall x(man(x) \rightarrow mortal(x))$$

$$man(Socrates)$$

$$mortal(Socrates)$$

Again we view how the third statement logically follows from the first two. This time is more observable and streamlined. By inserting the second statement, *man(Socrates)*, into the first, we get $man(Socrates) \rightarrow mortal(Socrates)$ (since Socrates replaces the variable x in the man() instance it must also replace the x in the mortal instance). Not only is this logical form more straight-forward for us as humans but it is beneficial for computers as well since the logical representations follow cleanly from each other.

Since we have observed the usefulness of HoL expressions, it follows that converting natural languages to HoL form could be beneficial in multiple areas of work (e.g. artificial intelligence, proving mathematical axioms). However, we would like to make the conversion process somewhat more automatic, as converting every expression by hand is tedious and we might be converting statements that hold little relevance. We propose a combination of context free grammars (CFGs) and Lambda calculus which aim to be one possible solution to the problem. We next further explore CFGs, Lambda Calculus, and other related ideas, in order to provide a better base for the intuition behind our project.

2 Background/Intuition

In this section, we explore the ideas needed to further understand the intuition and methods behind our proposed system.

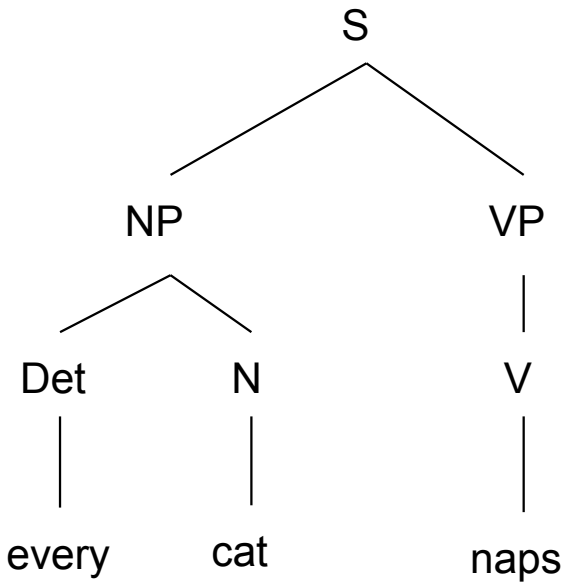

Figure 1: A CFG representation of "every cat naps"

2.1 Context Free Grammar

Context Free Grammars (CFGs) are useful to capturing general patterns of strings. For example, even though the sentences "Every cat naps." and "A dog runs." have different contextual meaning, they are both captured by the same CFG. Both sentences follow the form "Det Noun Verb". CFGs function by working with a set of rules to determine if a sentence is in the CFG. A common CFG representation of natural language is $S \rightarrow NPVP$, where S stands for "sentence" and NP/VP stand for Noun Phrase and Verb Phrase respectively. These two can be further divided down into a similar structure as S. An example of a CFG being applied to an earlier sentence can be observed in figure 1.

Focusing back on our main goal, we may take a simple approach at this point and convert all nouns and verbs into simple HoL expressions. In our previous case, we could convert *cat* to $cat(X)$ and *naps* to $naps(X)$. This makes CFGs a good starting point for our goal, however, they fall short since we still lack to ability to combine these HoL expressions into the larger HoL expression. In order to remedy this, we leverage Lambda Calculus.

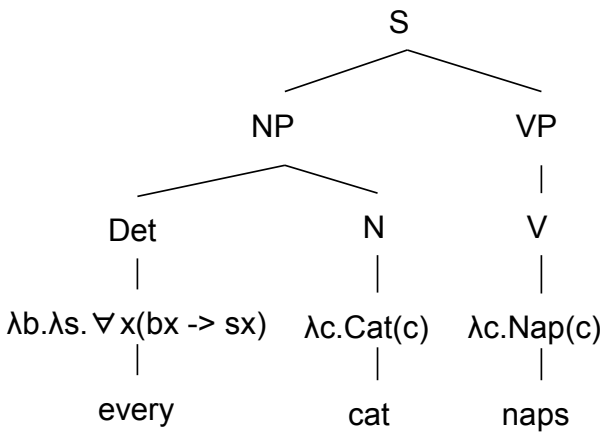

Figure 2: Lambda representation of "every cat naps"

2.2 Lambda Calculus

Lambda Calculus is a vast and complex field, which is why we limit our discussion in this section to only some of the simpler ideas we leverage as well as assume at least some Lambda Calculus knowledge. Lambda Calculus contains two main terms we will use, lambda abstractions and applications. Lambda abstractions ($\lambda x.M$) take in a single input x and submit it to the expression M. One example of a lambda abstraction can be seen by representing $f(x) = x + 7$ as $\lambda x.x + 7$. Similarly, we can represent our natural language with lambda abstractions. $cat(x)$ can be viewed as $\lambda x.cat(x)$ where λx binds x in $cat(x)$. We can do the same with $naps(x)$ as well. We can observe the newly represented CFG combined with lambda abstractions in figure 2.

Applications ($M@N$ or MN) are simply representing a function N being applied to a function M. When a lambda abstraction is applied (application) to another term, we are able to reduce the terms to a new term. This process is called beta reduction. For example, if we apply $\lambda x.cat(x)$ to $Belle$ to form $(\lambda x.cat(x))Belle$, we can beta reduce this expression to $cat(Belle)$. (Though this is a vague look at beta reduction, the actual beta reduction concept is too large to explain in this paper). Leveraging beta reductions we can now combine the lambda terms. The combination can be observed in 3

Combining CFGs and Lambda Calculus to find HoL forms is the main approach of our system, we now look at which tools and methods were utilized to implement our approach.

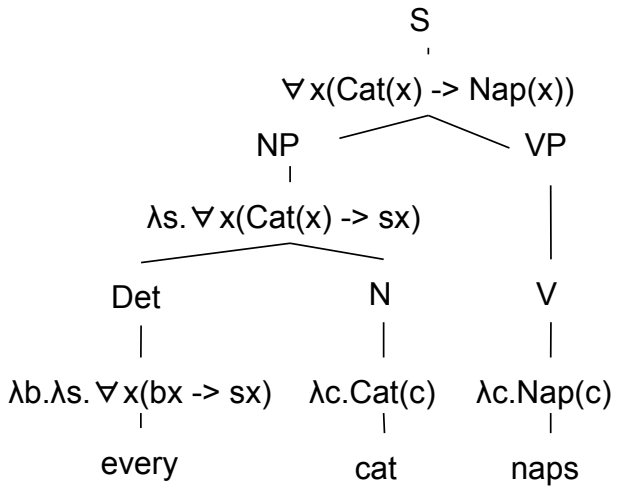

Figure 3: Fully reduced lambda representation of "every cat naps"

3 Methods and Tools

In order to express language in logical form, Prolog was leveraged. Furthermore, the idea to read in files and convert them into logical forms (explained more in section 4) brought us to exploit Python and the tools available to create a bridge between Prolog and Python.

3.1 Prolog

Prolog[1] is a commonly used logic programming language often used in areas such as computational linguistics and artificial intelligence. Prolog files are generally composed of facts and rules. A fact is simply a statement that is true for the file. *cat(belle)* is one example of a fact that simply states *belle* is a cat. Facts can be composed of multiple terms as well, e.g. *father(tim, sara)* is a fact that states *tim* is the father of *sara*. Note also how no capitalization exists within the facts, this is due to the fact that capitalized terms represent variables in Prolog. Variables are exploited in "rules" of Prolog. *parent(X,Y) :- father(X,Y)* is a rule that states, "X is a parent of Y if X is the father of Y".

Prolog also allows us to read in a file of facts and rules and make different queries on the file. If we read in the above facts and rules and make the query "parent(tim, sara).", Prolog will respond with "Yes." since Prolog searches the rules then substitutes *tim* and *sara* for the variables X and Y. This then satisfies the

[1] http://www.swi-prolog.org/

fact *father(tim, sara).* Furthermore, instead of looking for a specific query, we can exploit Prolog to return "yes" results for us. If we make the query "parent(X, sara)." Prolog will respond with "X = tim" since Prolog searches for a satisfaction of the queried problem.

Now that we've introduced Prolog, the question remains to how we represent lambda expressions and CFGs in Prolog. CFGs are straightforward to implement in Prolog with the use of Definite Clause Grammar (DCGs). DCGs allow us to represent rules in a readable fashion, for example, if we want to represent a sentence s as an application of a noun phrase (np) to a verb phrase (vp) we can represent this as:

$$s('@'(NpS, VpS)) --> np(NpS), vp(VpS).$$

As can be seen, we have also added in an application from lambda calculus as well using the @ symbol. $'@'(X, Y)$ is the same as $X@Y$ or XY from lambda calculus. Similarly, we represent λ as the rule $l(X, F)$. This is the lambda calculus representation $\lambda x.F$, where F represents another expression. As sentence, np and vp can be represented in Prolog as:

$$np('@'(DetS, NS)) --> det(DetS), n(NS).$$

$$vp(l(X,'@'(VSem, X))) --> vi(VSem).$$

$$vp(l(Y,'@'(NpS, l(X,'@'('@'(VtS, Y), X))))) --> vt(VtS), np(NpS).$$

Notice how vp has two different rules. This is due to transitive (vt) and intransitive (vi) verbs. Transitive verbs allow the verb phrase to be made up of itself and another noun phrase.

While Prolog is useful for representing facts and rules it doesn't have the same easy functionality for reading in unsorted files and converting them into facts and rules. This lead us to exploit Python's straightforward framework.

3.2 Python

Python[2] is a streamlined popular programming language. Python allows us to read in files and parse the sentences. We then converted sentences into lambda rules, explained more in section 4. Besides reading in new files, we wanted a way to automatically send in queries to Prolog from Python. This functionality was

[2]https://www.python.org/

found in the Python module pyswip[3]. Pyswip allows Python to work with Prolog by consulting Prolog files consisting of facts and rules, then making queries (as was shown above). With this added functionality, we were able to freely create and query Prolog files from the automated nature of Python. As the main tools have been presented, we now look further into how our system functions.

4 Functionality

The main program in our system is the Python file *learnFromAndQueryFile.py*. *learnFromFile.py* contains two different functionalities, *learnFromFile()* and *queryWhatWeKnow()*.

4.1 learnFromFile()

learnFromFile() takes in a text file (as well as Prolog database file) and converts each sentence into a Prolog rule. The current functionality converts nouns and verbs found in the file. Text is tokenized and tagged by use of NLTK[4]. Nouns are converted to Prolog lambda expression in a similar look to those described in section 3.1. Verbs are first identified as transitive or intransitive. We determine this in a simple fashion (since our program currently takes in simpler text files). If a verb if found at the end of a sentence or the word immediately follow that word is not a noun, then it is labeled as intransitive. Transitive verbs are then determined as those who are immediately followed by a noun. Since determinants represent a more difficult lambda expression, we leave these as hardcoded in the Prolog file. Once the sentences are converted into logical forms, we add the new logic to the passed in Prolog file. We have successfully added in new rules, in order to combined and query the rules we look to our next function.

4.2 queryWhatWeKnow()

queryWhatWeKnow() takes in a Prolog database (generally just generated with the previous function) and allows users to enter in sentences to get the logical form of the sentence. If the logical form does not exist in the database file, this is noted. The database is assumed to contain the described CFG, as well as a formula

[3]https://github.com/yuce/pyswip

[4]https://www.nltk.org/

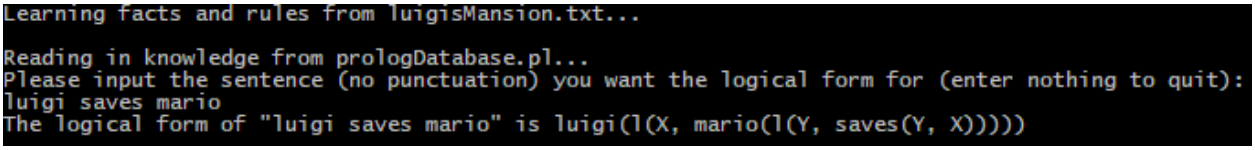

Figure 4: An example of our described system being trained and queried

to beta reduce the terms (these are both present in our example Prolog file [prologDatabase.pl]. An example of this use can be seen in figure 4. As we have described how our current iteration of the system function, we now examine the limitations and future work possibilities of this system.

5 Discussion

In this section we examine some shortcomings of our current system as well as discuss possible extensions of the work.

The first limitation of our system is the simplicity at which it operates. As it only currently captures nouns and verbs from sentences, it only retains basic knowledge of passed in files. Future work would look to extend the system to more parts-of-speech as well as process non declarative sentences. The next limitation involves what the system is "learning". In figure 4, we read in and learn from the file "luigisMansion.txt". In that file, the sentence "Luigi saves Mario." is presence, and we can query for the logical form of this sentence. On the other hand, he sentence "Mario saves Luigi" is not in the file, however, the user is still able to query for and receive the logical form of this. If we truly only wanted our system to learn what was present in the text, this wouldn't be allowed. Future work would possible remedy this by combining the two aforementioned functions and crosschecking queries on the input file.

6 Conclusion

We described and implemented a system which reads in a text file and converts the sentences into logical form. The learned knowledge can then be queried to receive the logical forms of sentences. Described code and files have been uploaded for use[5].

[5]https://github.com/JonRusert/lambdaCalcNLP