Joseph Swartz

JKSwartz2019@manchester.edu

Natural language processing: an analysis and application

# Abstract

This document first covers the history of natural language processing along with explanation of the speech recognition and grammar areas; then second covers the development of a system that applies speech recognition and parsing to convert human speech into evaluable calculus expressions.

Table of Contents

[1. Introduction 1](#_Toc8358747)

[1.1. Definitions 1](#_Toc8358748)

[2. Background 1](#_Toc8358749)

[2.1. Machine Translation 1](#_Toc8358750)

[2.2. Natural Language Processing 2](#_Toc8358751)

[2.2.1. From Machine Translation to NLP 2](#_Toc8358752)

[2.1.2. A Shift of Focus 2](#_Toc8358753)

[2.1.3. The Beginning of a New Era 2](#_Toc8358754)

[2.3. Achievements 3](#_Toc8358755)

[3. Natural Language Processing Applications 3](#_Toc8358756)

[3.1. Speech Recognition 3](#_Toc8358757)

[3.1.1. Converting Speech to Data 4](#_Toc8358758)

[3.2. Grammar 4](#_Toc8358759)

[3.2.1. Part-of-Speech Tagging 4](#_Toc8358760)

[3.2.2. Parsing 4](#_Toc8358761)

[4. Application: “Calculus Assistant” 6](#_Toc8358762)

[4.1. Development Environment 6](#_Toc8358763)

[4.1.1. python Packages 6](#_Toc8358764)

[4.1.2. Java IDE 6](#_Toc8358765)

[4.2. Software Diagrams 7](#_Toc8358766)

[4.2.1. Layered Architecture 7](#_Toc8358767)

[4.2.2. Sequence Diagram 7](#_Toc8358768)

[4.3. User Interface 8](#_Toc8358769)

[4.4. Speech Recognizer 8](#_Toc8358770)

[4.5. Parser 8](#_Toc8358771)

[4.5.1. How it Works 8](#_Toc8358772)

[4.6. Evaluator 8](#_Toc8358773)

[4.7. Difficulties 9](#_Toc8358774)

[4.7.1. Development Environment Difficulties 9](#_Toc8358775)

[4.7.2. NER Difficulties 9](#_Toc8358776)

[4.7.3. Parsing Difficulties 9](#_Toc8358777)

[4.7.4. Evaluator Difficulties 10](#_Toc8358778)

[4.8. System Constraints 10](#_Toc8358779)

[5. Future Development 11](#_Toc8358780)

[5.1. Different Evaluator 11](#_Toc8358781)

[5.2. Complete Generalization of Calculus Expressions 11](#_Toc8358782)

[5.3. Deployment 11](#_Toc8358783)

[References 12](#_Toc8358784)

# 1. Introduction

This document is a culmination of research pertaining to natural language processing (NLP). While anybody should be able to understand the content, the targeted audience is composed of students and faculty in the fields (or related fields) of mathematics and software engineering. Topics covered include background history, how natural language processing works, and an application project.

## 1.1. Definitions

|  |  |
| --- | --- |
| Machine Translation | Going, by algorithm, from source text (e.g. Russian) to useful target text (e.g. English) without recourse to human translation or editing |
| Parse | Result of parsing a string or text |
| Parsing | Converting a string or text into logical syntactic components |
| Semantics | The linguistic study of meaning |
| Subparse | A parse that makes up a part of a larger parse |
| Syntax | Grammatical arrangement of words in a sentence and their relationship with each other |

# 2. Background

## 2.1. Machine Translation

One of the first patents for machine translation was in 1933 from a man named Peter Troyanskii (Pestov, 2018) and it was for a machine used for “selection and printing of words when translating from one language to another”. This machine was ditched by the USSR and the idea of machine translation picked back up in the 1950s when IBM’s 701 computer translated 60 Russian sentences into English—the first time in History. Alongside this feat, the Cold War spurred more research with machine translation; the goal here was to translate any Russian to English.

## 2.2. Natural Language Processing

### 2.2.1. From Machine Translation to NLP

Linguistic research during the 1950s and the next decade or so focused on syntax—this was due to syntactic processing being necessary and many people believed that syntax parsing was the way to accomplish this task; focusing on syntax was also believed to allow the use of ambiguous words so that output could be correctly determined via the context of the target text.

Those opposed to focusing on syntax were in that position because they believed that semantics were more important and thus they adopted semantic parsing, or the process of mapping a natural language sentence into a formal representation of its meaning. This process uses semantic categories: Agent (words that refer to animate beings), Object (inanimate objects), Action (action words or verbs), and Location (the notion of place; where an object is or moves to, or where an action takes place) (Williamson, 2014). The process also uses semantic case frames, which are collections of facts that specify “characteristic features, attributes, and functions of a denotatum (actually existing object), and its characteristic interactions with the things necessarily or typically associated with it” (Allan, 2001)—in short, case frames allow one to determine meaning by examining words that generally surround a trigger word.

### 2.1.2. A Shift of Focus

All research almost came to a dead halt when in 1966 the Automatic Language Processing Advisory Committee deemed that the progress of NLP had slowed down enough to no longer be worth the funding; however some fringe research groups continued their work. Since machine translation basically ended, researchers began work on artificial intelligence. This shifted focus to world knowledge and NLP’s role in the construction and manipulation of meaning representations.

### 2.1.3. The Beginning of a New Era

As computers got more and more powerful, artificial intelligence became more feasible to work with as did work on semantics and computational linguistics. An aspect of the work on semantics is the Lexicalist approach to grammar. Lexicalists study lexical chunks, or pairs or groups of words commonly found together (Islam & Timmis, n.d.), and these are useful because they tend to follow grammar rules.

This surge of computer power, coincided with the advent of the World Wide Web in the early 1990s, sparked immense progress within NLP as the sudden flood of information prompted the need to be able to analyze the data and at the same time provided a plethora of data to use for research. A result of this was statistical language processing; this involved data analysis as well as the application of statistical methods to NLP, for example probabilistic parsing and Hidden Markov Models. Probabilistic parsing is defining a probability distribution using weighted parsing, which is attaching a weight to a parse and subparse—higher weights correlate to more confidence in a given parse being correct (Nederhof & Satta, 2008). A Hidden Markov Model is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (i.e. hidden) states. Hidden means that the state changes within the model are invisible; but the output is still visible, and dependent on states, so weights are assigned to outputs.

## 2.3. Achievements

* 1957: Noam Chomsky published “Syntactic Structures” and created Phrase-Structure grammar
* 1958: John McCarthy released LISP (Locator/Identifier Separation Protocol)
* 1961: BASEBALL question answering system could take punch card input and answer questions about baseball games
* 1964: ELIZA, one of the first chatbots ever created, successfully impersonated a psychiatrist
* 1970: SHRDLU could take input like “put the blue cube on top of the red cube” and performed the action in the real world
* 1982: Jabberwacky, a chatbot designed to humorously mimic human speech
* 2011: IBM’s Watson won a game of Jeopardy against some of the best players of all time

# 3. Natural Language Processing Applications

This section will cover how speech-to-text conversion works as well as how textual NLP works, complete with underlying theory and examples.

## 3.1. Speech Recognition

According to Grabianowski, current speech recognition programs fall into two categories: small-vocabulary/many-users and large-vocabulary/limited-users (Grabianowski). He states that the small-vocabulary/many-user systems are ideal for automated phone answering where users need to be able to speak with varying accent and speech patterns; due to the nature of needing to work with a wide variety of inputs, the usage is limited to predetermined commands and inputs and thus not many use cases. On the contrary, large-vocabulary/limited-user systems are ideal for business environments where the system learns from and is tailored to the use of a small number of users; these systems are able to support massive vocabularies but adding untrained users results in a loss of accuracy.

### 3.1.1. Converting Speech to Data

When one speaks, vibrations are sent through the air. Speech recognition software utilizes a microphone to capture the sound, and then an analog-to-digital converter (ADC) to convert it into digital data. The ADC samples (or digitalizes) sound by taking precise measurements of the soundwave at frequent intervals, filters out unwanted noise, and normalizes the sound so that it is easier analyzed (Grabianowski).

After the sound is collected and cleaned up, the signal gets divided into chunks roughly 10-20 milliseconds in length (Amos, n.d.). These chunks are then matched against known phenomes of the language in use—a phenome being the smallest unit of language. From here the two dominant means of choosing the most likely said speech are Hidden Markov Models (HMM) and neural networks, with HMMs being the most common. The application of an HMM involves mapping vectors to phenomes; these vectors are comprised of real numbers that are known as cepstral coefficients, which are found by taking the inverse Fourier transform of the logarithm of the power spectrum of the respective phenome. The dimension of these vectors is generally kept small (about 10), but accurate systems can use dimensions of 32 or more (Amos, n.d.). The HMM then uses these vectors to, statistically, determine what was said.

## 3.2. Grammar

This section gathers information from *Developing a Chunk-based Grammar Checker for Translated English Sentences*, an article in which a Grammar Checker system was developed to reduce grammar errors in translating from Myanmar to English.

### 3.2.1. Part-of-Speech Tagging

Part-Of-Speech (POS) tagging is a necessary step in being able to determine the grammatical structure of a sentence. POS tags are parts of speech, including but not limited to: verbs, adverbs, nouns, pronouns, prepositions, and adjectives. These are then organized into chunks, or textual units of adjacent POS tags (Lin, Soe, & Thein). To achieve correct chunk structure, the sentence first needs parsed.

### 3.2.2. Parsing

There are two methods by which one can parse sentences and they are Top-down and Bottom-up parsing (Lin, Soe, & Thein). The more used method is Bottom-up parsing, and this is the process in which a parser takes an input sentence, combines the words into chunks, and then combines the chunks until a sentence structure is obtained. Some chunk types are as follows (Lin, Soe, & Thein):

|  |  |  |
| --- | --- | --- |
| **Chunk Types** | **Description** | **Example** |
| NC | Noun Chunk | a young boy, the girls |
| VC | Verb Chunk | is playing, goes, went |
| AC | Adjective Chunk | more beautiful, younger, old |
| RC | Adverb Chunk | usually, quickly |
| PTC | Particle Chunk | up, down |
| PPC | Prepositional Chunk | at, on, in, under |
| COC | Conjunction Chunk | and, or, but |
| QC | Question Chunk | Where, Who, When |
| INFC | Infinitive Chunk | to |
| TC | Time Chunk | tomorrow, yesterday |

An example of Bottom-up parsing would be as follows:

1. POS tag “The students are playing football in the playground.”
   1. The[DT] students[NNS] are[VBP] playing[VBG] football[NN] in[IN] the[DT] playground[NN].
2. Combine POS tags into chunks and then combine the chunks until a sentence structure is obtained.
   1. [DT\_NNS] becomes an NC
      1. NC\_ [VBP\_VBG][NN][IN][DT][NN]
   2. [VBP\_VBG] becomes a VC
      1. NC\_VC\_ [NN][IN][DT][NN]
   3. [NN] becomes an NC
      1. NC\_VC\_NC\_ [IN][DT][NN]
   4. [IN] becomes a PPC
      1. NC\_VC\_NC\_PPC\_ [DT\_NN]
   5. [DT\_NN] becomes an NC and we have a completed sentence structure
      1. NC\_VC\_NC\_PPC\_NC

# 4. Application: “Calculus Assistant”

This section will cover my senior project, an application of NLP. The goal of the project was to have a textual or speech input—within the context of a calculus expression—and understand what was input in order to evaluate. This project utilizes the application of NLP in the fields of speech recognition and parsing.

## 4.1. Development Environment

In order for all packages to run, as well as to further experience with Ubuntu, all development was done on an Ubuntu operating system. Other than the graphical user interface, all other portions of the software use python, namely python version 3.7.3.

### 4.1.1. python Packages

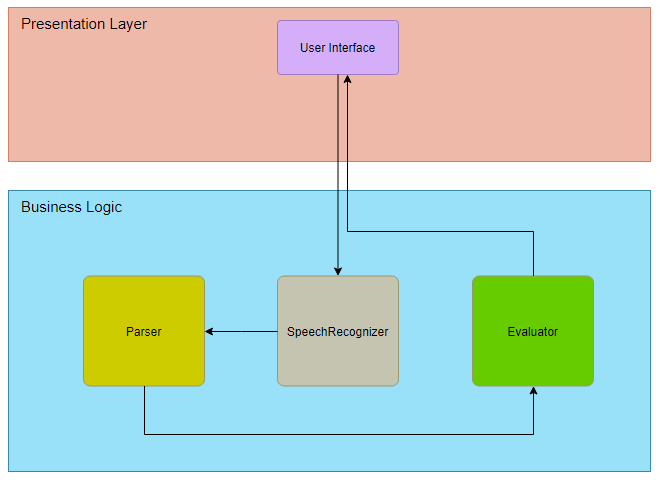
* SpeechRecognition
  + To gather speech from a microphone, SpeechRecognition was implemented. Details for software requirements as well as how to install can be found at <https://pypi.org/project/SpeechRecognition/>
* spaCy
  + In order to define my own part-of-speech (POS) tagging and to define actions that happen upon matched phrases, spaCy was implemented. Details for installation as well as user guides can be found at <https://spacy.io/usage>
* SymPy
  + To evaluate calculus expressions, SymPy was implemented. Since development was done with python version 3.7.3, pip3 is needed. Install pip3 and SymPy using these commands:
    - sudo apt-get install python3-pip
    - sudo pip3 install sympy

### 4.1.2. Java IDE

Java was chosen for the sole reason of familiarity with JavaFX for developing graphical user interfaces (GUI). The IDE chosen for Java development was IntelliJ, which is free for students. The Linux download can be found at <https://www.jetbrains.com/idea/download/#section=linux> and student application for free licenses can be found at <https://www.jetbrains.com/shop/eform/students>.

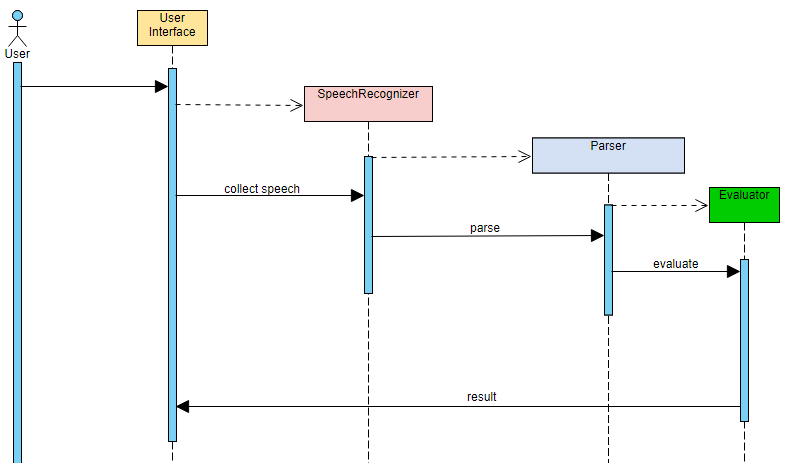
## 4.2. Software Diagrams

### 4.2.1. Layered Architecture



Since the system is currently just a prototype, the architecture is easily modeled using a layered architecture. The Presentation Layer contains the User Interface and interacts with the Business Logic where data is transferred to and fro.

### 4.2.2. Sequence Diagram



The sequence diagram shows the general flow of the desktop prototype. The User of the system interacts with the GUI and notifies the system to begin collecting speech. The system uses the SpeechRecognizer to collect the speech and then relay the said speech as text. Next the Parser takes the text and converts it into an expression usable by the Evaluator. The Evaluator then evaluates the expression and relays the result back to the GUI to show the User.

## 4.3. User Interface

The original idea for this software was to have it able to be distributed and ran as any other application would be, so providing a GUI seemed appropriate. I spent some time researching developing GUI with python, but yielded no result and so opted to stick with JavaFX since that is what I am familiar with. The GUI provides a means for notifying the software when to capture speech, a means for textual input, and a means for displaying evaluated expressions. This does so by implementing command processes to run python scripts.

## 4.4. Speech Recognizer

The speech recognizer uses the onboard microphone to collect speech, which it then sends to the google-cloud-speech API to determine what was said. The speech recognizer then writes the spoken text to a text file for use with the parser.

## 4.5. Parser

While the foundational knowledge of parsing text for POS tagging was gained using NLTK and the free NLTK eBook, all parsing and tagging was done using spaCy.

## 4.5.1. How it Works

The parser reads the command text from the text file that was created by the speech recognizer. It then utilizes spaCy’s nlp() command to tokenize the text. While the text is tokenized, any defined phrase tags are then associated with the tokens—I must note that in order to bypass errors thrown from assigning multiple tags, I had to disable spaCy’s named-entity recognition (ner). From the tags, one can define actions that are to be carried out upon matching; let’s call these on-action events. The on-action events handle how to interpret what commands were given, and build a string that is a valid SymPy expression. This string is then used as input when calling the evaluator.

## 4.6. Evaluator

The evaluator utilizes SymPy’s expression parser to convert the passed in string into a valid expression and then evaluates it. It then returns the result for display by the GUI.

## 4.7. Difficulties

As with all software development, numerous difficulties were encountered. This section will outline some of the major difficulties along with their mitigation strategies.

### 4.7.1. Development Environment Difficulties

The first step for this project was implementing NLP packages, with which python was chosen due to its ease of use and numerous packages for such tasks. The main problem arose when a speech recognizer was implemented, for some python packages simply do not have Windows support. To counter this, development took place in an Ubuntu virtual environment, since python has complete support of Unix operating systems. From here it was a matter of installing packages and libraries to the correct directories so that they would all run together.

### 4.7.2. NER Difficulties

spaCy implements an auto-ner and issues arose when trying to assign new tags to tokens for they were already automatically tagged. To counter this, spaCy allows for the disabling of the ner when importing the libraries.

### 4.7.3. Parsing Difficulties

Two difficulties arose when parsing textual commands: the speech recognition software returns variables with coefficients as one token, and one must strategically build the end command with everything in the correct order. To counter the variable difficulties, I made use of python’s regular expression capabilities—which introduced another issue in which using regular expressions parses the entire text and it is difficult to keep track of where things should go. Thus some constraints on the system were put in place; these will be outlined later.

To counter the difficulty in keeping things in order, I had the phrase-match on-action events build into the main command text in such ways as:

* To match SymPy’s syntax, a function needs parentheses surrounding the function inputs, so when the match of a function occurs, an open parenthesis is also added, only to have the close parenthesis added at the end of the parsing.
* Similarly, calculus functions such as “diff” or “integrate” also need open and close parentheses. The only difference is that before the close parenthesis these commands also need variables that are operated on or other parameters for evaluation. These are also appended at the end of the parsing.
* To handle constants multiplying functions, upon matching functions the parser checks to see if the preceding token is a number—if yes, then the number along with “\*” is added to the overall command text.

### 4.7.4. Evaluator Difficulties

Upon testing the implemented functionality, some annoyances were found using SymPy. Some integral expressions returned are written in unfamiliar ways and so at first glance one may think that the answer is wrong, but when compared to more familiar expressions they are correct. However, some definite integral values return complex numbers where they should be returning real numbers. An example of this is “Integrate 2 tangent of 4.6x from 0 to 1.”

Similarly, some derivative expressions are written in unfamiliar ways but once compared to familiar expressions they are similar. An example of this is “What is the derivative of 2 tangent of x squared?”

## 4.8. System Constraints

Due to parsing difficulties and time spent on development, some system constraints are put in place until the difficulties can be completely mitigated. These constraints are as follows:

* The system can handle normal calculator functionality, derivative, and integration commands. These are limited to:
  + Simple expressions, e.g. “2\*ln(2\*x)” or “2\*4”.
  + Single use of quantities, meaning one cannot evaluate “(2+x)\*(x-5)” or “(4+9)/(5-3)”.
  + At most squaring variables and numbers, e.g. “x2” or “52”
  + Basic trigonometric functions (sine, cosine, tangent) and logarithmic functions.
    - No current support of exponential or any other functions.
  + Single derivatives and integrals.
* The system has to be ran within a Linux desktop environment, with the python scripts having knowledge of the directory location of each other, and the GUI also has knowledge of the python scripts and any text files that they write to/read from.
* The system needs internet access for the speech recognition to work due to the use of the google-cloud-speech API.

# 5. Future Development

Upon a successful prototype, I began thinking long term and how the system can benefit others. To achieve future goals, some major changes need made.

## 5.1. Different Evaluator

Due to the aforementioned annoyances, research into other calculus evaluators are desired. It is known that Wolfram Alpha has the capability of taking math expressions and evaluating them, and since it was used to verify answers it is a potential candidate. This does not alleviate the need for the parser, for the user would still need to be able to have spoken expressions such as quantities and specific placement of exponents implemented correctly.

## 5.2. Complete Generalization of Calculus Expressions

As apparent by the system constraints, the parser needs a significant amount of extra work in order to accommodate any and all variations of functions, variables, numbers, and even numbers that are represented by symbols such as pi. Some improvements here could be made by redesigning the regular expression functionality in order to more accurately detect variables and their coefficients, while keeping them in the correct order in which they were entered. spaCy supposedly offers support of regular expressions in its phrase matching, but during development of this project it did not work correctly—if in the future it works correctly, it could solve the problem of the regular expression searching the entire text and instead use them to match patterns in separate tokens.

## 5.3. Deployment

While desktops are very powerful, it would make more sense for this application to exist as a mobile application. This conclusion was drawn based on my own experience in tutoring and noticing that students tend to always search things on their phones. As the system currently stands, a means for deploying the python packages would need to be researched so that a mobile application can run the needed scripts. Potential options here would be to redesign the system as a web service and have the application make calls to the service rather than contain all necessary software.

Mobile application development is not currently a skill I own, and so development of the mobile app itself needs researched. Along with this would be coordinating with fellow students or creating my own user experience design so that the app is appealing and not difficult for the average user to utilize.

# References

Allan, K. (2001). *Natural Language Semantics.* Melbourne, Australia: Oxford: Blackwell Publishers.

Amos, D. (n.d.). *The Ultimate Guide To Speech Recognition With Python*. Retrieved March 29, 2109, from Real Python: https://realpython.com/python-speech-recognition/

Grabianowski, E. (n.d.). *How Speech Recognition Works.* Retrieved from HowStuffWorks: https://electronics.howstuffworks.com/gadgets/high-tech-gadgets/speech-recognition.htm/printable

Islam, C., & Timmis, I. (n.d.). *Lexical Approach 1 - What does the lexical approach look like?* Retrieved from TeachingEnglish: https://www.teachingenglish.org.uk/article/lexical-approach-1-what-does-lexical-approach-look

Jackins, T. (2012, March 1). *The History of Natural Language Processing*. Retrieved from Engineering and Technology History Wiki: https://ethw.org/The\_History\_of\_Natural\_Language\_Processing

Jones, K. S. (2001, October). Natural language processing: a historical review. Cambridge, England, United Kingdom.

Jr., B. F., Wolf, A. K., Chomsky, C., & Laughery, K. (1973). *Baseball: An Automatic Question-Answerer.* Lexington, Massachusetts: Lincoln Laboratory, Massachusetts Institute of Technology.

Lin, N. Y., Soe, K. M., & Thein, N. L. (n.d.). *Developing a Chunk-based Grammar Checker for Translated English Sentences.* Yangon, Myanmar: Association for Computational Linguistics.

*Machine Learning Research Group*. (n.d.). Retrieved from University of Texas at Austin: http://www.cs.utexas.edu/~ml/publications/area/77/learning\_for\_semantic\_parsing

McCarthy, J. (1960). *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.* Cambridge, Massachusetts: Massachusetts Institute of Technology.

Nederhof, M.-J., & Satta, G. (2008). *Probabilistic Parsing.* North Haugh, Saint Andrews, United Kingdom: School of Computer Science, University of Saint Andrews.

Nordquist, R. (2018, July 09). *Phrase Structure Grammar*. Retrieved from ThoughtCo: https://www.thoughtco.com/phrase-structure-grammar-1691509

Pestov, I. (2018, March 12). *A history of machine translation from the Cold War to deep learning*. Retrieved from freeCodeCamp: https://medium.freecodecamp.org/a-history-of-machine-translation-from-the-cold-war-to-deep-learning-f1d335ce8b5

Tayal, M. (2014). Syntax Parsing: Implementation using Grammar-Rules for English Language. *ResearchGate*, 376.

Williamson, G. (2014, January 29). *Semantic Categories*. Retrieved from SLTinfo: https://www.sltinfo.com/semantic-categories/