Conversational Mathematics

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in the
Mathematics and Computer Science at The College of
Wooster

by
Dylan Orris
The College of Wooster
2019

Advised by:

Dr. Fox (Mathematics and Computer
Science)

THE COLLEGE OF
WOOSTER

© 2019 by Dylan Orris

Abstract

Natural language processing systems have revolutionized the way we interact with technology over the last several years, from simple calendar management with digital assitants, to safer driving by allowing dictated communications behind the wheel. Though everyday activities are a bit easier, these systems have not lowered the barrier of entry to mathematics. In this study, we seek to create a translation system to link mathematical proofs constructed by users in everyday English with an automated proving system called Vampire. While the system is limited, it works well within certain domains.

Acknowledgments

I would like to acknowledge my advisor Dr. Nathan Fox for ensuring I made progress every week and his great patience with me. Without his knowledge, I never would have been able to learn what I did during this study. I would also like to thank the Computer Science and Mathematics departments for the lessons and skills they have taught me.

I would also like to thank the friends who helped to make a stressful process one I will look back upon fondly: Joe, Gianni, Kasey, Phillip, Thanh, and Scott.

Contents

Abstract v

Acknowledgments vii

Contents ix

List of Figures xi

List of Tables xiii

List of Listings xv

CHAPTER PAGE

1 Introduction 1
- 1.1 Problem Statement . . . 2
  - 1.1.1 Project Aims . . . 2
    - 1.1.1.1 Natural Language Processor . . . 3
    - 1.1.1.2 Automated Theorem Prover . . . 3

2 Background on Natural Language Processing 5
- 2.1 Early Years . . . 5
  - 2.1.1 SHRDLU . . . 5
  - 2.1.2 ELIZA . . . 8
- 2.2 Advancments of the Twentieth Century . . . 9
  - 2.2.1 Context Free Grammars . . . 9
  - 2.2.2 Advancements of the 1970s and 1980s . . . 11
- 2.3 Syntax and Semantics . . . 12
  - 2.3.1 Pragmatics . . . 16

3 Background on Automated Theorem Proving 19
- 3.1 Historical Development . . . 19
- 3.2 Decidability, Incompleteness, and First Order Logic . . . 21
  - 3.2.1 Decidability . . . 21
  - 3.2.2 Incompleteness and First Order Logic . . . 22
- 3.3 Automated Proving Theory . . . 23
  - 3.3.1 Herbrand Universe . . . 24
  - 3.3.2 Resolution . . . 25
    - 3.3.2.1 The Resolution Rule . . . 25
    - 3.3.2.2 The Resolution Technique . . . 26

3.3.3 Superposition . . . 27
3.3.3.1 Knuth-Bendix Completion Algorithm . . . 27
3.4 Vampire . . . 30

4 Software Implementation and Use 31
4.1 Requirements for Use . . . 31
4.2 treeRead . . . 32
4.2.1 Noun Phrase Breakdown Functions . . . 33
4.3 How to Use the Program . . . 33
4.3.1 Proper Language Formatting . . . 34
4.3.2 Transitional TPTP File and Vampire . . . 36
4.3.3 Axioms . . . 37

5 Conclusion 39
5.1 Failings . . . 39
5.2 Future Work . . . 41

References 43

List of Figures

Figure Page

2.1 Sample SHRDLU starting world 6
2.2 Sample SHRDLU dialogue 6
2.3 Flowchart for Figure 2.2 7
2.4 SHRDLU world after Figure 2.2 7
2.5 Sample ELIZA dialogue 8
2.6 Flowchart for dialogue in 2.5 9
2.7 Three production rules, one of which produces a variable and two of which produce a terminal 10
2.8 A production rule which produces a terminal 10
2.9 Sample Context Free Grammar 11
2.10 Sample Semantic Analysis of a Sentence 13
2.11 An incorrect parsing of "The man walked down the road wearing a hat" 15
2.12 The correct parsing of "The man walked down the road wearing a hat" 15

List of Tables

Table Page

4.1 Examples of proper and improper formatting. 36

List of Listings

Listing Page

CHAPTER 1

Introduction

Over the past decade, natural language processors have become more and more commonplace. In late 2011, Apple introduced their digital assistant, Siri, to iOS. Siri was the successor to voice control, an iOS feature that allowed the user to interact with their phone using their voice, though in both an extremely limited fashion and rather unreliably. Siri was powered by a much better voice recognition model. Not only could it process many commands into actions, but it could even ask for clarification based on context. Since then, Google introduced the Google Assistant to its Android and ChromeOS platforms, Amazon released smart home products powered by Alexa, and Siri has been ported to MacOS and WatchOS. All of these digital assistants rely on understanding the commands of users, and the only way this is possible is through natural language processing.

From Q2 2017 to Q2 2018, smart speakers, the quintessential aspect of the smart home to many, had shipments grow 187% [1]. This growth is expected to continue in the coming years, with smart speakers and other smart home technologies becoming an enormous industry. As this industry expands, there will be greater and greater access to these systems among the general population. Smart devices offer more than simply convenience, they could become an amazing source of educational opportunities which are not otherwise available for many. For example, what if it were possible for a student to ask their smart screen why a mathematical claim was true, and that system could respond by displaying the proof?

While smart devices do not appear in this project, we have created a front end interface for students to more easily investigate mathematical questions, without the barrier of technical language.

1.1 Problem Statement

Automated theorem proving systems are a powerful tool which can allow a student to check their work or a researcher to prove a claim which could be extremely time consuming if done by hand. Unfortunately, these systems are not trivial to use, with the use of any such system requiring the user to learn the proper method for input to that system. As a result, these systems are quite intimidating to learn, and once one learns one such system, it is easy to be stuck using what is already known, even if better options may be available. A wonderful solution to this problem is leveraging computational power to convert a human language request, e.g. "Prove that the length of the third side of a triangle is determined by the two other sides and the angle between them" into a language the proving system understands.

1.1.1 Project Aims

This project has two separate but interconnected goals; the production of a natural language processor and a translator to allow the prover to test input. The processor:

- Serves as a front end for any automated prover using TPTP format.
- Analyzes basic questions on algebra, set theory, and geometry.
- Is usable without any understanding of symbolic notation or programming.

The translator:

- Does all translation work and prover functionality without input from user.
- Accurately translates standard English to symbolic notation.

As the two pieces are separate, the processor is not limited by the prover, and may instead be joined with a more robust system by specifying what said system is. This is necessary to convert the request properly, so that the processor knows which symbolic language to translate to.

1.1.1.1 Natural Language Processor

The processor takes a user's input in English, and determines the goal of the user based on the content of that input. The user is not required to know anything, other than that the system is intended for mathematical queries and will be able to do nothing else. Once the input is analyzed, the processor converts it to the proper symbolic language for the automated prover.

1.1.1.2 Automated Theorem Prover

The prover takes a mathematical claim, in a symbolic notation, and applies axioms, theorems, and other knowledge as necessary to prove or disprove the claim. If the prover is able to use theorems in its proofs, it is barred from using them in such a way as to give the appearance of circular logic. For example, when a user asks for a proof of the Pythagorean Theorem, the prover may not use the Pythagorean Theorem as its justification for the truth of the theorem. Should the user ask for a proof of an axiom, it is permissible to simply point to the axiom. The prover is not expected to explain an axiom to a user.

CHAPTER 2

Background on Natural Language Processing

2.1 Early Years

Natural language processing has been an area of interest in computer science for the past 60 years. Initially, the subject was limited to machine translation, with the Georgetown experiment in 1954 being an early foray into the field. In this experiment over sixty Russian sentences were translated to English entirely automatically. The results were promising enough that the researchers anticipated automatic translation being solved within the next five years [5]. Progress was, obviously, much slower than this.

Funding was cut when the program had run twice as long as promised, and progress in the field slowed. Two successful systems were created in the 1960s, SHRDLU and ELIZA.

2.1.1 SHRDLU

SHRDLU was a querying system which created a small "Blocks World" which was populated by cones, spheres, cubes, and other geometric shapes of various sizes and colors [16]. The user was able to instruct SHRDLU on how to move these objects around simply by specifying the shape in addition to its color or size. SHRDLU also had a simple memory system, allowing for reference to objects recently interacted with. This memory also allowed for SHRDLU to be queried on what she had previously done. SHRDLU's world contained basic physics, allowing for the program to describe what was possible in the world and what was not. The final major feature was to remember the name a user gave an object or collection of objects. From this, it was possible to more easily instruct SHRDLU.

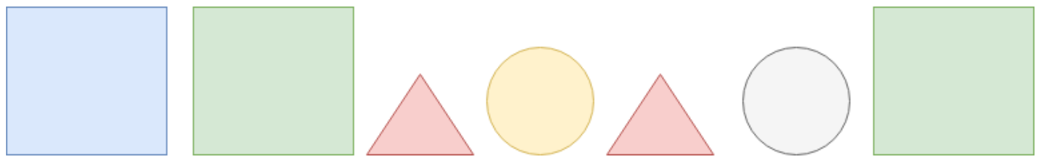
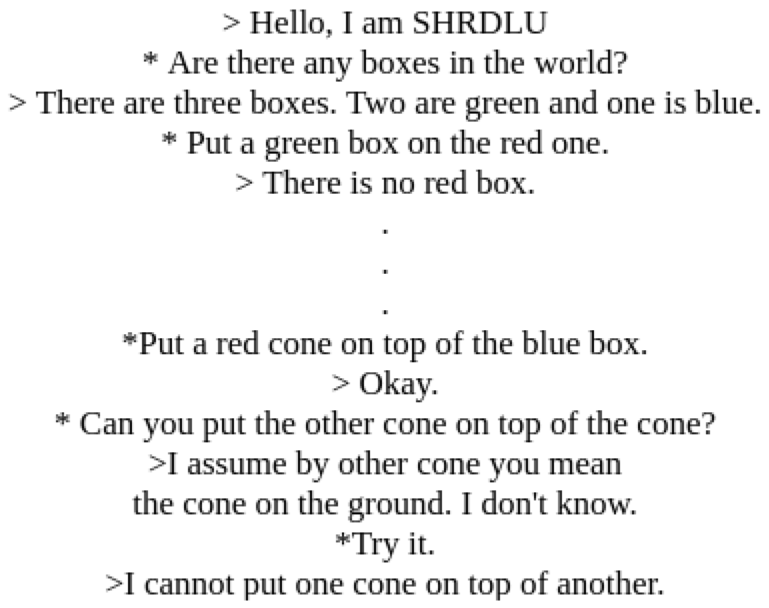


Figure 2.1: Sample SHRDLU starting world



> Hello, I am SHRDLU
* Are there any boxes in the world?
> There are three boxes. Two are green and one is blue.
* Put a green box on the red one.
> There is no red box.
.
.
.
*Put a red cone on top of the blue box.
> Okay.
* Can you put the other cone on top of the cone?
>I assume by other cone you mean
the cone on the ground. I don't know.
*Try it.
>I cannot put one cone on top of another.

Figure 2.2: Sample SHRDLU dialogue

SHRDLU would remember changes in the environment until reset after use. From this simple technique, it became much easier to consider SHRDLU to inhabit a real world, as locations were consistent and basic physics were the same as reality. While SHRDLU's domain was small, effective code and good design helped making

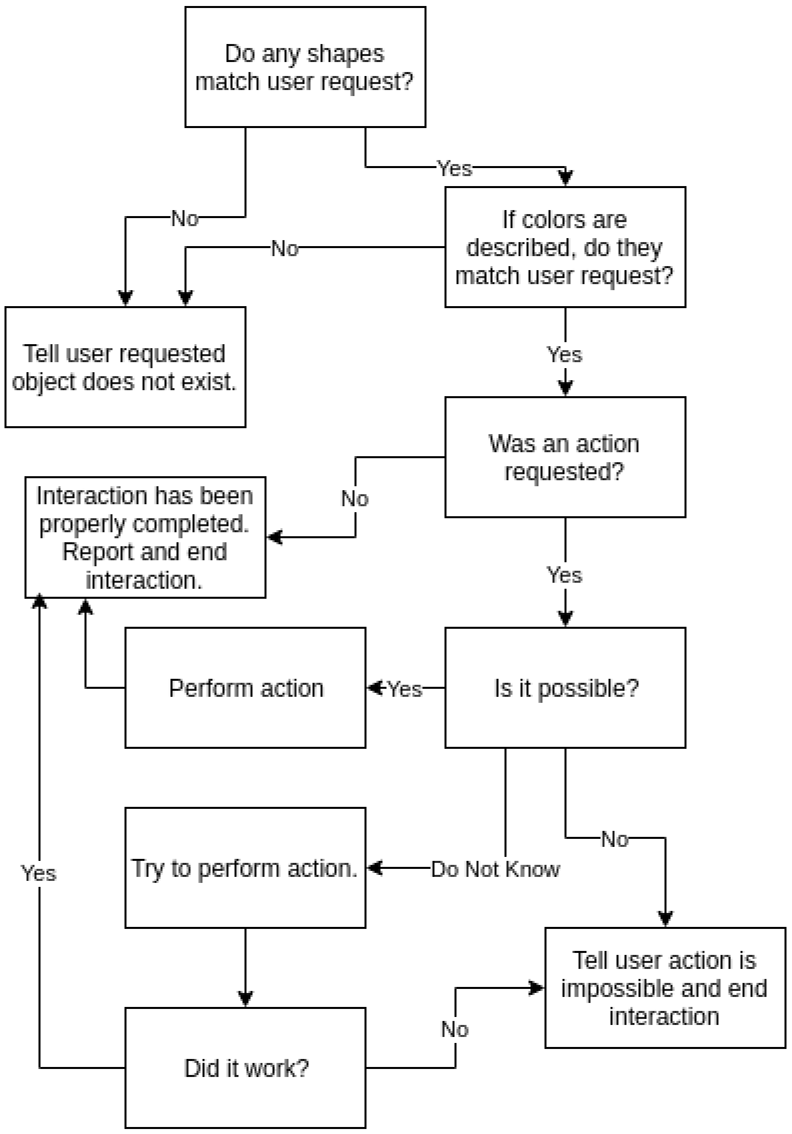

Do any shapes match user request?
Yes
No
If colors are described, do they match user request?
No
Tell user requested object does not exist.
Yes
Was an action requested?
No
Interaction has been properly completed. Report and end interaction.
Yes
Perform action
Yes
Is it possible?
Try to perform action.
Do Not Know
No
Yes
Tell user action is impossible and end interaction
No
Did it work?

Figure 2.3: Flowchart for Figure 2.2

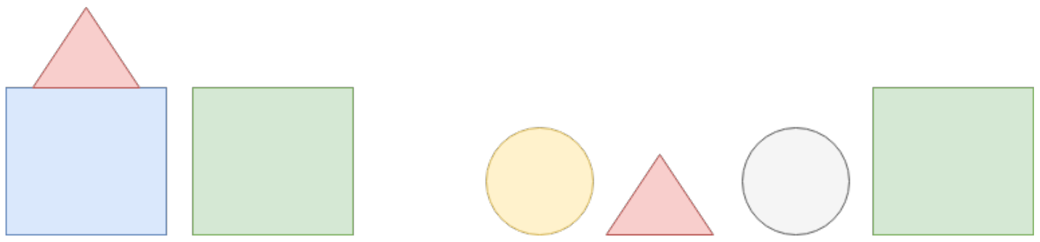


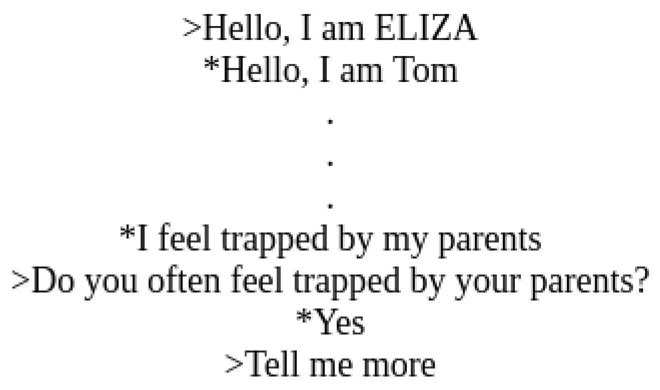
Figure 2.4: SHRDLU world after Figure 2.2

for a convincing approximation of both a world and an intelligent entity which allowed interaction with that world.

2.1.2 ELIZA

ELIZA was an early language-based program which took user input and responded much like a Rogerian psychologist [15]. She was created by Joseph Weizenbaum around 1965 at MIT, using pattern matching and substitution to give an illusion of understanding. While ELIZA has only a very small vocabulary, she convinced many users that she was truly intelligent through the use of these techniques, despite being completely unable to go into detail on almost all subjects.

ELIZA uses NLP in a very different way from SHRDLU – rather than attempting to understand what the user is inputting, she instead uses the context surrounding phrases, such as “I feel” or “I am” to insert the phrase properly into previously constructed sentence types.



```
>Hello, I am ELIZA
*Hello, I am Tom
.
.
.
*I feel trapped by my parents
>Do you often feel trapped by your parents?
*Yes
>Tell me more
```

Figure 2.5: Sample ELIZA dialogue

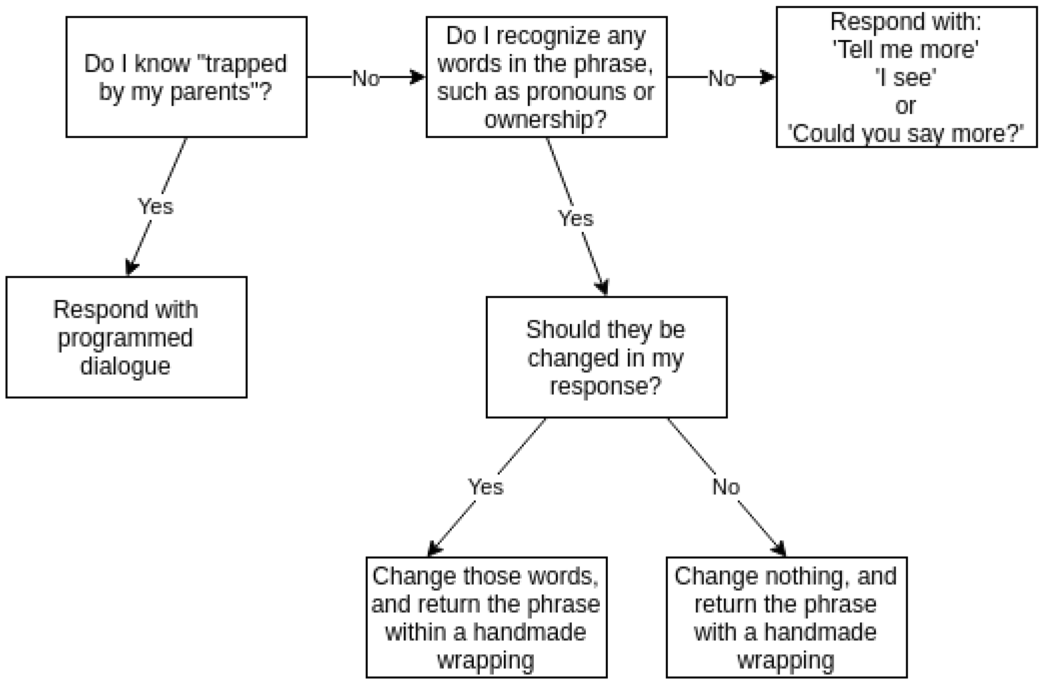

Do I know "trapped by my parents"?
No
Do I recognize any words in the phrase, such as pronouns or ownership?
No
Respond with: 'Tell me more' 'I see' or 'Could you say more?'
Yes
Respond with programmed dialogue
Yes
Should they be changed in my response?
Yes
No
Change those words, and return the phrase within a handmade wrapping
Change nothing, and return the phrase with a handmade wrapping

Figure 2.6: Flowchart for dialogue in 2.5

2.2 Advancments of the Twentieth Century

2.2.1 Context Free Grammars

To understand the functioning of NLP systems, context free grammars (CFGs) should first be described. CFGs, sometimes referred to as phrase structure rules, are a set of rules, known as *production rules* which describe all strings which can possibly be produced within a language. Here, string simply means a sequence of words, such as, "The man walked down the road" and grammar refers to the possible structure of strings and to the words which may be included in these strings. There are two basic elements of a CFG – variables and terminals. A variable is a part of the grammar which will be modified by continuing to follow the production rules which are described by the grammar. We denote variables by enclosing them in angle brackets ('<' and '>'). A terminal is a word, such as "the," which is in its most basic form and undergoes no further changes.

This is most easily seen through an example. Consider the following two figures:

$$\langle S \rangle \rightarrow \langle B \rangle$$

$$\langle B \rangle \rightarrow \text{Dog} \mid \text{Pig}$$

Figure 2.7: Three production rules, one of which produces a variable and two of which produce a terminal

$$\langle S \rangle \rightarrow \text{Cat}$$

Figure 2.8: A production rule which produces a terminal

In each of the cases, we have the variable $\langle S \rangle$ on the left of the arrow. To the right of the arrow is what will replace $\langle S \rangle$, which is another variable $\langle B \rangle$ in Figure 2.7, and a terminal ("Cat") in Figure 2.8. Notice that in Figure 2.7, the second variable has two terminals on the right of the arrow with '|' between them. This indicates that $< B >$ may be replaced by "Dog" or "Pig". Variables may follow rules which are:

- one-to-one, where the variable is replaced by a single other variable or terminal.
- one-to-many, where the variable has several possible variables or terminals which may replace it, but only one of which is selected.
- one-to-none, where the variable is replaced by nothing (blank space).

The first variable used for a CFG is $\langle S \rangle$, which stands for "start". In language processing, $\langle S \rangle$ will typically produce a noun-phrase ($\langle NP \rangle$) and verb-phrase ($\langle VP \rangle$), which then produce many possible sentences by defining the location of parts of speech. Each variable serves as a part-of-speech tag, such as noun, verb, or adjective. For this reason, CFGs are an effective way of capturing the inherent structure of language. A valid sentence always contains a noun and a verb, for example, and the CFG will be unable to produce a sentence without a noun or a verb, assuming it is constructed properly. Thus an incorrect sentence such as "Jumped." will not be produced by our grammar. This allows for a somewhat simple checking of proper English sentences – if we know the parts of speech of each word, we can either work our way up from the sentence to see if $\langle S \rangle$ could have produced it, or work our way down and see if the sentence structure appears.

Let us consider a CFG structured as follows:

- $\langle S \rangle \rightarrow \langle NP \rangle \langle VP \rangle$
- $\langle NP \rangle \rightarrow \langle DET \rangle \langle N \rangle$
- $\langle VP \rangle \rightarrow \langle ADVB \rangle \langle VB \rangle \,|\, \langle VB \rangle$
- $\langle N \rangle \rightarrow$ Dog
- $\langle DET \rangle \rightarrow$ The | A
- $\langle VB \rangle \rightarrow$ Runs | Jumps
- $\langle ADVB \rangle \rightarrow$ Excitedly

Figure 2.9: Sample Context Free Grammar

Here, we have a simple grammar which breaks a sentence into noun and verb phrases, which then break into a determiner and a noun, and a possible adverb and verb, respectively. Once these variables are reached, they lead to terminals, which here are the English words which are represented. In this grammar, the sentence "The dog excitedly runs" is valid, while "The dog happily jumps" is not. The second sentence is not incorrect due to any issues with the English grammar, but rather due to the given grammar not accounting for the terminal 'happily'.

2.2.2 Advancements of the 1970s and 1980s

Come the 1970s, William Woods introduced augmented transition networks (ATNs) as a method to represent language input, rather than phrase structure rules [17]. ATNs use finite state machines in order to parse sentences. Woods claimed that, by adding a recursive mechanism to finite state models, it was possible to parse much more efficiently. The system builds a set of finite state automata, which have transition states between them. Should a sentence reach a final state, the sentence is valid. These systems have advantages, including the delaying of ambiguity (see page 13). Rather than simply guessing a path as some systems will, the ambiguity may be delayed until more of the sentence has been parsed, allowing for greater information to be used in resolving said ambiguity. Additionally, they effectively capture the structure of languages, allowing for ease of processing [17].

The 1980s, the introduction of machine learning algorithms, led to immense changes. No longer were parsers built based on complex rules formed by the programmer; instead, algorithms like decision trees began to make the classification rules. Eventually, this change led to the use of modern statistical models, which assign probabilities to words for part-of-sentence identification, rather than rigid if-then rule sets [6].

2.3 Syntax and Semantics

When examining natural language, meaning is found through an analysis of both syntax and semantics. Syntax consists of the rules which determine the structure of sentences in a language. For example, English requires a verb and subject for a sentence to be grammatically correct. Semantics refers to the meaning of words within the language, such as the word ‘dog’ referring to a four-legged, furry animal which descends from wolves and was domesticated by humans. Early implementations of natural language processors typically focused on one of these aspects of language to the detriment of the other [8].

Certain camps believed that, by simply knowing the definition of each word in a sentence, their relationship to one another would be determinable. Others saw syntax as the defining feature to study, so they broke sentences down based purely on anticipated structures of sentences, fitting the sentences into some hypothetical structure which could be determined by a CFG [8].

Let us first examine the issues with a purely semantic analysis of a sentence. Consider "The dog ran to the man who owned him." Each word can easily be defined, but the meaning of the sentence becomes extremely unclear. In this situation, it is clear that the final three words "who owned him" refer to a relationship between the man and dog. However, we may not examine the structure of the sentence, as it was regarded as unnecessary. The "him" in this context has an unclear referent without examining previously discussed entities.

- The → Indexical
- Dog → Canine
- Ran → Moved Quickly
- To → In The Direction Of
- The → Indexical
- Man → Adult Male Human
- Who → Identity Request or Explanation
- Owned → Posessed
- Him → Male Pronoun

Figure 2.10: Sample Semantic Analysis of a Sentence

Even disregarding this issue, a hypothetical system using this methodology would be completely powerless to even attempt to understand a sentence containing a word it did not already have the definition of. The system, assuming parsing completed, would return a description of each word other than the unknown. Parsing refers to the analysis of a string by conversion to variables in the language in order to test its conformity. A string parsed successfully is in the language given, while one in which the parsing fails is not. Without understanding the meaning of this region, the meaning of the sentence becomes unintelligible. This system can also never attempt to determine potential meaning, as relationships between words are completely ignored.

With some issues of purely semantic analysis discussed, let us now move on to the issues facing a purely syntactic analysis. First, consider the possible shapes of every sentence in English. When we understand the meaning of words in a sentence, it is trivial to see the difference between two sentences of the same length and structure, say "Timothy ran to his crying father" and "Sparky bit at the panicking neighbor". By seeing how each component comes together, we can easily match sentences to their derivation from a CFG. However, by purely examining syntax, we cannot check the part-of-speech of the component words. What we are left doing is taking an n-word sentence, and applying to it the structure of every possible n-word archetype.

The introduction of syntactic analysis brings ambiguity into our parser as well. Ambiguity is the scenario when a single string can be parsed in two different ways. Consider a sentence such as "The man gave a dollar to his friend who was wearing his hat." In this sentence, it is not clear who the "his" refers to – we could parse it such that the friend is wearing the man's hat while receiving a dollar, or the man is giving a dollar to a friend, who is in a hat he owns. When the meaning of a word is known to the parser, ambiguity can often be resolved. As natural language is full of ambiguity, this is not always true, but understanding the meaning can resolve situations such as "The man walked down the road wearing a hat."

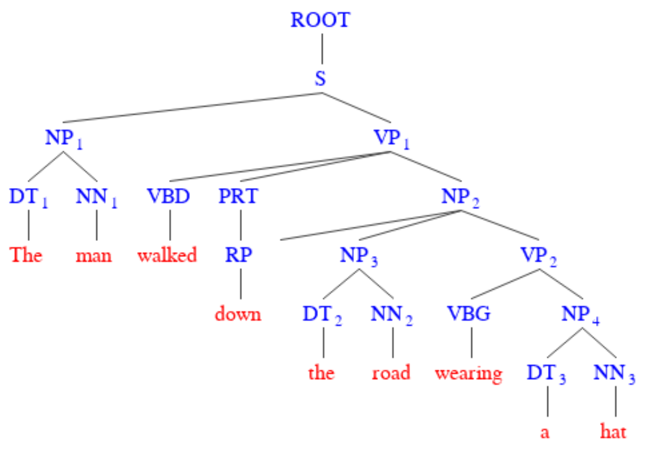


Figure 2.11: An incorrect parsing of "The man walked down the road wearing a hat"

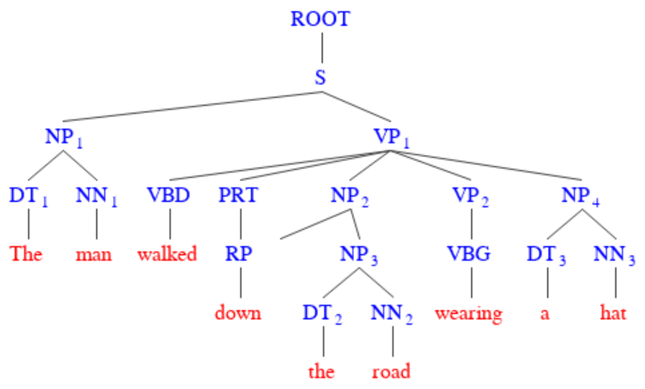


Figure 2.12: The correct parsing of "The man walked down the road wearing a hat"

When we don't understand what a 'road' and 'hat' are, we find ourselves unsure which noun is wearing the hat. The sentence could be parsed either way, and either would be valid. The best way to resolve these issues is to consider syntax and semantics simultaneously.

A semantic consideration need not be of the definition of the word, but can instead consist of part-of-speech tagging, or a way of indicating what types of actions it can take. In Figure 2.11 and Figure 2.12, simply knowing that roads, hats, and men are nouns will not help to resolve the ambiguity. However, if we know that men have a property such as *can wear apparel* and the hat is part of a class *apparel*, we know that the man wearing a hat is valid. Inspecting the road, we will see it has no *can wear apparel* property, and thus, the sentence in which the road wears the hat is invalid. Recognition of word position and relations between words allows semantic analysis to provide a far better description of the content of each word.

2.3.1 Pragmatics

Primarily, natural language processing relies on syntax and semantics to determine the meaning of a given string. In the scenarios which human beings use language, pragmatics is also a concern. Where syntax comes from the structure of a sentence, and semantics from the words used, pragmatics stems from the context in which it is used. For example, if one were going on a date to a horror movie, and their partner said “I’m glad you picked such a relaxing movie for our date,” this would clearly be a sarcastic comment. Horror movies are meant to not be relaxing, so the meaning of the sentence completely changes with that knowledge.

Some pieces of general knowledge like this are possible for a NLP system to pick up. If we consider a digital assistant which has access to the location data of users, when queried “Who is the president?” the assistant can assume that the user is asking for the president of the nation they reside in. Previously, we saw SHRDLU, a system which created its own world to work in. As SHRDLU knew everything about the world, it was possible to refer to objects such as “the blue one,” assuming there was only a single blue object.

One piece of pragmatic information which NLP systems are currently oblivious to is tone of voice. As humans, we are able to interpret changes in voice, whether as an indicator of sarcasm or a marking of a change in mood which could change the meaning of the sentence. This is a very difficult problem to solve with computing. Consider using a text-based communication service with a friend. You believe that they are acting a little odd, so ask them if they are okay. They respond “I’m fine.” Were you face to face with the person, you could use body language or vocal cues to determine if the statement was an accurate representation of their emotional state, or if it was a social nicety so as not to worry you. Without this, we have a sentence of only two words which could take, at minimum, two different meanings.

Of course, we do not expect to have friendly conversations with our computers, so this may seem a moot point. However, a good NLP system should make conversation easy, and with easy conversation comes slipping into habits which one uses in real life. Maintaining clarity is important with a NLP system, to ensure understanding on the ends of both the user and the system.

CHAPTER 3

Background on Automated Theorem Proving

3.1 Historical Development

While the main additions to mathematics from computational development previously came from automation of basic operational calculations, new potentials arose in the 1950s. With artificial intelligence beginning to be developed, the first attempts were made to automate reasoning, and mathematical understanding along with it.

At the turn of the 20th century, David Hilbert posed 23 problems which he hoped would be solved within the next 100 years. Of particular interest to us is his second problem, which asks for a proof of the consistency of arithmetic – that it is not possible to prove both a statement and its negation within arithmetic. This was resolved in 1931 with the publishing of Kurt Gödel's two incompleteness theorems, which we discuss in Section 3.2.2. In short, Gödel proved that it is impossible to prove or disprove the consistency of arithmetic, unless arithmetic is inconsistent [12].

In 1908, French mathematician Henri Poincaré stated that, were formalism of mathematical proof to gain further traction "one could imagine a machine where one would put in axioms at one end while getting theorems at the other end." This was not an idea Poincaré was happy with however, with many mathematicians in this period finding the idea of removing the intuition of the mathematician from proving to be disturbing.

This mechanical idea became most clear in 1937, with Turing's description of the Turing machine, which he used to determine if mathematics is decidable (see 3.2.1). He determined mathematics to be undecidable, but nonetheless started mathematics down the path of automated theorem proving. Following years of development primarily on automating arithmetic, automated theorem proving began to appear as a question in computing with early attempts to create artificial intelligence [9].

Allen Newell and Herbert Simon met at the Rand Corporation in 1952. Simon held a PhD in political science, focusing on thought processes, and was intrigued by the idea of using computers to simulate human problem solving. Newell, on the other hand, wanted to use computers to play chess games. In 1955, the two decided to work together, but they decided that building a chess playing AI was far too difficult. So they picked what they saw as a much easier task – building an automated theorem prover.

Newell and Simon went on to create the Logic Theory Machine, based on the propositional calculus described in Russell and Whitehad's *Principia*. Propositional calculus is logic which entirely relates to truth and falsity of statements and argument flow, meaning it is significantly simpler than first order logic, which we will describe in the next section. The machine proved the majority of the theorems discussed in *Principia*, even finding a simpler proof than the authors in one case, but this was refused publishing by the *Journal of Symbolic Logic*, as the Logic Theory Machine was considered a co-author. Though an impressive first attempt, Newell and Simon's machine was capable of only very short proofs, as it made no attempts to choose the proper path and instead took every option available to it.

This early work into automated theorem proving divided academia into two camps – the logicists and proceduralists. Logicists argued for a system using purely logic based inferences, and were also known as “neats” as they did not want the actual details of problems dirtying the logic. Rather than work from an understanding of the input material, a logicist’s prover would look only at the underlying predicate logic. If the logic was deemed valid, the claim would be considered to be verified. The proceduralists, or “scruffies,” instead believed that knowledge should be considered procedural, rather than axiomatic. The example given by one such proceduralist was that of checking to see if it was safe to cross the road. This, like a proof, will return a truth value. However, it does not seem correct to use the logicist’s method in this case. A better understanding of the task is to check each direction, and if no car is seen, consider the road safe to cross. As the proceduralist states, the logicist would be left attempting to prove that no car was coming using logic, rather than this procedure [9]!

3.2 Decidability, Incompleteness, and First Order Logic

3.2.1 Decidability

In order to understand automated theorem proving (ATP) and some of the difficulties faced by it, we will first discuss decidability. Decidability, as the name implies, describes the ability of a claim to be proven. Note that this is not describing the truth value that would be returned after being proven – it describes if the question can even be proven. Decidability features heavily in computational theory. Problems can be loosely divided into two categories, decidable and undecidable. While further distinctions exist, we will not focus on them here. It is important to note that a problem may take billions of years to solve, but it remains decidable – at the end of the time period, a truth value will be returned for the claim. Undecidable claims can be given unlimited computing power and unlimited time, but will still be unable to return a truth value.

The typical example of an undecidable problem is what is known as the halting problem. This problem seeks to create a program which, when passed another program as input, will determine if the input program will ever cease execution (halt). While this program may seem possible to create, it is not.

Theorem 1. *There is no program which will be able to determine if any arbitrary program will halt.*

Proof. Let us consider a program h which determines if a program passed to it will halt. Let us consider a second program, p which takes h as an input and passes itself to h. This program is then able to see the output of h, as h will return its value for reading by p. From this, p reads this input, and performs the opposite of what was predicted by h. If it was predicted to halt, it will run forever, and if it was predicted to run forever, it will halt. As p uses its knowledge of what h will return to determine its behaviour, there is no way for h to properly predict the behaviour, and thus no program h is possible [7]. □

3.2.2 Incompleteness and First Order Logic

This worry of decidability relates to mathematics through Gödel's incompleteness theorems:

Theorem 2. *If a formal system is consistent, it cannot be complete.*

Theorem 3. *The consistency of axioms cannot be proven within their own system [**?**].*

While both of these theorems are important, we will be focusing on Theorem 2. This matters to our work as problems which cannot be proven will never halt. Further, we now see that simply providing a series of axioms is not enough to ensure that the resulting system is consistent.

Knowing this about incompleteness and decidability, we now examine the language of automated theorem provers – first order logic. First order logic is a system used in linguistics, mathematics, computer science, and philosophy in order to logically describe claims and statements. For example, rather than state "The person reading this sentence understands English" we instead state "There exists X such that X is reading sentence Y and X understands English". The "first" indicates that the statements made utilize variables, such as X in the previous sentence, but functions are not used as arguments. First order logic dictates the axiomatization of mathematics. This project uses first order logic axioms for each of the question domains, such as set theory or algebraic arithmetic, in order to define the systems concisely.

As first order logic produces systems which cannot be proven fully based on their axioms, we give a program a set of axioms which may or may not be able to prove our claim, and this program may or may not ever halt. With this in mind, we can see why a system could not simply be allowed to work undisturbed for years to verify a claim. While a theorem prover will be able to become more efficient over time, there will always be problems which cannot be proven just from the axioms for a system.

3.3 Automated Proving Theory

Before going into the theory and methodology of modern theorem provers, we will first describe what these provers are not. Proof assistants are a type of theorem prover which analyze input from a user, whether that be a potential next step in a proof which the computer will investigate or the next step the user wishes to take which the computer determines the validity of. These systems, though useful, require a much greater amount of interaction than was intended in this project. In our case, the only interaction the user should be allowed is providing the program with axioms and a conjecture to prove. This allowed for a user to be unsure how a proof needs to progress while still being able to see a result.

3.3.1 Herbrand Universe

The basic theory of automated proving begins with Herbrand and the Herbrand universe of S. Rather than seeking to prove that a claim is unsatisfiable in every universe, Herbrand devised a universe in which the negation of a conjecture being unsatisfiable meant that it would be so in all universes. By doing this, rather than having infinitely many universes to test a claim and thus never being able to absolutely prove a claim, the universes requiring testing dropped to just one. In this context, a universe refers to an arbitrary set of objects and functions [4].

Definition 3.3.1. The Herbrand Universe of S: Let H_0 be the set of constants which appear in S. If no constant appears in S, then H_0 is to consist of a single constant, $H_0 = a$. For $i = 0, 1, 2, \ldots$, let H_{i+1} be the union of H_i and the set of all terms of the form $f^n(t_1, \ldots t_n)$ for all n-place functions f^n occuring in S, where t_j, $j = 1, 2, \ldots, n$, are members of the set H_i. Then, each H_i is called the *i-level constant set* of S, and H_{inf} is called the *Herbrand universe of S* [**?**].

Essentially, a Herbrand Universe of S is the set of all ground terms that can be formed using any of the functions and constants which appear in a set of axioms. If no constants appear in the set, then an arbitrary a is used.

To place all of our claims within the Herbrand universe, we must convert them from clauses to **ground instances**. To do this, the variables within our clauses must be replaced with members of the universe we seek to use. We use the term "clause" here rather than claim, as we now refer to a conjunction of one or more disjunctions. While this system brings us to a single universe for testing, it unfortunately experiences rapid growth as we attempt to convert all of the axioms and claims which we know to ground instances.

3.3.2 Resolution

To solve this issue, first-order resolution is used. First described by Davis and Putnam in 1960, it was refined to its current form in 1965 due to Robinson's syntactical unification algorithm. This technique is relatively straightforward.

First, the claim to be proven is negated. Then, we must have the axioms and this negation be conjunctively connected, which simply means that there are shared literals and clauses between members of the set. At this point, the negation is conerted to conjunctive normal form, meaning it is a conjunction of one or more clauses. At this point, setting up for the algorithm has completed, and application is ready to begin.

3.3.2.1 The Resolution Rule

The resolution rule is a single inference rule which produces a single clause from two clauses which contain at least some complementary literals. The term "complementary" is used here to indicate that the two literals are the negation of one another, for example, c and c. The clause which results contains all literals from each of the component clauses which did not contain a complement in the other clause. This resulting clause is known as the resolvent.

The resolution rule is applied until no new clauses may be created through its application. This results in one of two scenarios – either the production of the empty clause, or one or more clauses which are not complementary. In the event that the empty clause is produced, the negation provided is unsatisfiable, and thus, the claim that is being tested must follow from the axioms. If the empty clause is not the only result, the negation has not been shown to be unsatisfiable, so the original conjecture is invalid.

3.3.2.2 The Resolution Technique

When used in conjunction with a search algorithm, the resolution rule produces an algorithm which decides the satisfiability of a propositional formula. The technique uses proof by contradiction and that any sentence in propositional logic may be transformed into a conjunctive normal sentence without loss of meaning. What this means is that by only changing the form of the sentence, no information that was in the original sentence is lost or altered, and no further relationships are made with the sentence and other sentences.

1. All axioms of the domain, as well as the negation of the conjecture, are conjunctively connected.

2. The resolvent is converted to conjunctive normal form with conjuncts as elements in a set of clauses.

3. The resolution rule is applied to all pairs of clauses with complementary literals. Repeated literals are removed after each use of the rule to reduce the size of the set of clauses from the previous rule.

   (a) If the resolvent contains complementary literals, it is not added to the set of clauses.

   (b) If it does not contain complementary literals and is not already in the set, it is added and becomes eligible for further application of the resolution rule.

4. When it is no longer possible to apply the resolution rule:

(a) The empty clause is derived, and thus the negation of our conjecture is a contradiction. This indicates that the original conjecture was valid.

(b) The empty clause cannot be derived, and thus the negation of our conjecture has not been shown to be false. This indicates that the original conjecture is incorrect.

3.3.3 Superposition

By combining first-order resolution with Herbrand universes, we reach what the majority of modern automated theorem proving systems use – superposition. Superposition is a system which, for any decidable problem in first-order logic, is refutationally complete, meaning that as long as the refutation is able to be found, the strategies employed by the prover are fair, and resources (such as time and memory) are infinite, the refutation will always be found.

While primarily based on logical resolution, elements of ordering based equality handling are important to its implementation, specifically, an unfailing Knuth-Bendix Completion Algorithm.

3.3.3.1 Knuth-Bendix Completion Algorithm

The Knuth-Bendix Completion Algorithm is what is known as a semi-decision algorithm, which transforms a set of equations into a confluent term rewriting system. Semi-decision indicates that the algorithm may or may not halt, depending on the input. Confluent refers to a property of systems where, if some element x yields y and z, then at some later point in derivation y and z will yield the same value w. As we previously stated that superposition is based on an unfailing algorithm, we ignore discussion of non-halting results here.

By rewriting equations in terms of a confluent term system, a translated result is essentially solved for the given domain. A description of the algorithm, and an example of how it functions, follows. Descriptions of new terms will be within the example, as explanations of the terms are very unclear without seeing them in action.

1. Let $T = (L, \Gamma)$ be a theorem in which Γ is a finite set of axioms and L is a claim.
2. A set of initial rules, R, are constructed using members of Γ according to reduction order.
3. More reductions are performed in order to eliminate potential exceptions of confluence.
4. Should confluence fail at any point, the reduction matrix $max(r_1,r_2) = min(r_1,r_2)$ is added to R.
5. Any rules in R which have reducible left sides are removed.
6. The process repeats until all overlapping left sides have been checked.

Example 3.3.1. Consider a manipulation of strings consisting of the letters x and y such that $x^3 = y^3 = (xy)^3 = 1$, where any string equalling 1 may be eliminated. The first three reductions are very clear, and are determined entirely by the presence of an atom having an equivalence to a numeric value.

1. $x^3 \rightarrow 1$
2. $y^3 \rightarrow 1$
3. $(xy)^3 \rightarrow 1$

We will now look at reductions 1 and 3. Expanding expansion 3 produces $xyxyxy$, which shares a variable in its prefix in common with the sufix of the expanded reduction 1, xxx, which is x. Thus, we consider x^3yxyxy, which through application of the first production rule produces

4. $yxyxy \to x^2$

We see the same holds true with reductions 2 and 3, with the y simply being a suffix in this case for reduction 3.

5. $xyxyx \to y^2$

Reduction 3 is obsoleted by reductions 4 and 5, and thus is removed from our list of reduction rules. An "obsolete" rule is one which may be manipulated to produce other rules in the system through overlapping. We desire all reduction rules to be in the least general form which can be produced by the application of overlapping rules.

Now, we can consider the strings x^3yxyx and $xyxyx^3$ with an overlapping of reductions 1 and 5, resulting in

6. $yxyx \to x^2y^2$

7. $y^2x^2 \to xyxy$

Together, these two rules obsolete reductions 4 and 5.

We are left with the following reduction rules:

1. $x^3 \to 1$

2. $y^3 \to 1$

3. $yxyx \to x^2y^2$

4. $y^2x^2 \to xyxy$

Through the use of this algorithm, a prover is able to produce its own theorems which are as computationally simple as possible, allowing for less usage of memory along with more clear paths to follow. As prover's are multithreaded, eliminating confluence helps to minimize the number of threads performing different operations to produce the very same results.

3.4 Vampire

Vampire is a first order theorem prover which started development in the late 1990s at the University of Manchester. The system takes input in *Thousands of Problems for Thousands of Provers* (TPTP) format, with the user providing a file which contains axioms based on the domain of the problem, as well as a single conjecture at the end of the file.

First, Vampire takes the conjecture given by the user and negates it. For example, if it had been given a statement of the form p and q implies r, it would convert it to $\neg p$ and $\neg q$ and $\neg r$ before performing further analysis. Following this, Vampire then uses the provided axioms to attempt to find a contradiction. In the event that a contradiction is found, the original claim will be true. If in the time Vampire is given to run no contradiction can be found, it is not necessarily the case that the original claim is false. Vampire may also find an example satisfying the negated claim and the axioms, and will output that the axioms and the negated conjecture are satisifiable, in which case the original claim was false.

Over the past two decades, Vampire has won many awards at the CADE ATP System Competition's top division, winning first in 1999 and then every year from 2001–2010 [**?**].

CHAPTER 4

Software Implementation and Use

4.1 Requirements for Use

To use the software created during this project, the user must have an installation of Python 3, as well as the following packages:

- requests
- NLTK

The user will also need to run the Stanford CoreNLP server, which can be obtained from `https://github.com/stanfordnlp/CoreNLP`, as well as the Vampire theorem prover, available at `https://github.com/vprover/vampire`. Vampire is to built on the release version, using the included documentation to do so. The CoreNLP server need merely to be decompressed by the user, and then within the directory the command

```
java -mx4g -cp "*"
edu.stanford.nlp.pipeline.StanfordCoreNLPServer
-port 9000 -timeout 15000
```

which will set a timeout of 15000 milliseconds for processing of any statement sent to it.

4.2 treeRead

In order to use the data output by the CoreNLP server, a simple API was created which allows for all of the necessary information for conversion from English to TPTP format. The bulk of the module is within the `children()` function.

The `children()` function implements what is essentially a Lisp parser for the Stanford tree. After being provided with a tree to parse and a starting index, the children of the given tree are returned. As the number of children can be any value of one or greater, the function determines where children end by keeping track of parentheses, adding a child to a list of children whenever the count of parentheses returns to zero. After all children are accounted for, the function returns both this list and the final position within the tree. Currently, this final position is not used due to alterations in other functions since its creation.

The functions `word()` , `find_all()` , and `check_not()` provide utility for the functions which follow. As the name implies, `word()` returns the word associated with a given part of speech tag. In order to allow for different words with the same tag to be found, there is an optional parameter `shift`, which changes the location at which the search for the part of speech tag begins. A second value is also returned by `word()` , which gives the final index within the tree which was visited.

The function `find_all()` performs a search of the given tree for a provided string. Upon finding an instance of the string, the index it begins at is added to a list, and the search is run once more from the position immediately following the discovered string. Finally, `check_not()` determines if, prior to the final variable which has been found, the word "not" was present. If so, a tilde (̃) is added to the front of the function, indicating that that claim is not fulfilled.

4.2.1 Noun Phrase Breakdown Functions

The final three functions are `NP_PP()`, `NP_VP()`, and `NN()`, creatively named based on how the original noun phrase (NP) they come from is constructed. `NP_PP()` is called when a noun phrase and prepositional phrase (PP) are the children of a noun phrase. It first locates an operator term which will be present as a child of the noun phrase, such as “equal” or “element”. Following this, the child noun phrase’s sibling, the prepositional phrase, is searched for two nouns (NN), each of which are the variables which the function acts upon. For example, the prepositional phrase could contain the sets X and Y, which the operator from before could inform us indicates that X is a *subset* of Y.

`NP_VP()` is executed when the noun phrase produces a noun phrase and verb phrase (VP). The noun phrase is searched for a variable, while the verb phrase is searched for two nouns, one of which is a function and the other of which is a variable.

Finally, `NN()` executes only when the noun phrase has a single child which is a noun. In this instance, said noun will be a variable. To determine the function and other variable, the sibling of the noun phrase is determined which will be a verb phrase containing the remaining variable(s) and function. In this case, it is possible that rather than a noun, a number (cardinal number, CD) will be a variable. This is checked for by determining if the tag “CD” is present in the verb phrase; if it is, then said tag is used to find a variable, while if it is not, the standard “NN” tag is used.

4.3 How to Use the Program

To run the program, simply place the Vampire executable in the same directory as the program. Then, run the main Python file and provide a statement to test when prompted. The sentence will then be processed and converted to TPTP format behind the scenes, returning the output from Vampire when running the input claim with the proper domain of axioms. It is important to remember here that a refutation returned by Vampire indicates a true conjecture, while any other output means no contradiction was found between the negated conjecture and provided axioms.

4.3.1 Proper Language Formatting

To allow for accurate processing of input, certain limitations exist on the natural language portion. The first and most basic change, is a requirement to separate each claim with a comma. This means that, rather than stating "X is less than Y and Y is less than Z", input will be of the form "X is less than Y, and Y is less than Z". This is necessary not due to the implementation of the translator between trees and TPTP format, but rather due to occassional confusion during sentence parsing. Without clear separation by the comma, it is often the case that the "and" assigns a relationship between what is meant to be a new independent clause and the previous clause. By simply including the comma, this issue is almost always avoided.

A second alteration to the language required for using the program is an elimination of words like "it" which allow for indirect, non-explicit reference. This is more than just an elimination of words in this case, it is a change to some usual ways of speech. Consider the statment "There was riot in the market, which caused it to shut down." This includes the aforementioned "it" but also uses "which" to avoid repetition of "the riot." To put a statement like this into our program, though obviously this statement is rather light in mathematical reference, one would input "There was a riot in the market, the riot caused the market to shutdown." This would be a rather awkward manner of speech, but it allows for much more clarity than indirect reference does.

This change will be defended with one further example, as it is unfair to say it is a weakness of the system. Consider a statement like “When Joe shook Tom’s hand, he was completely unaware that he would die within the year.” In this context, which is grammatically correct English, it is quite difficult to determine to who the “he” applies to in each situation. Were there to simply be a change to reusing nouns in this situation, we would easily see that Joe was unaware that Tom would be dying.

Another limitation on word choice is the naming of variables. While it is acceptable to name a variable after any noun or most capitalized characters, the names “A” and “I” are reserved. When these characters are used, they are identified as different parts of speech than other variable names, so cause unreliable functionality.

Language is also limited, currently, by positioning of operator statements, those which translate as meaningful functions. These operators *must* not come after the two objects they are acting on. A statement such as “B and C are not equal” would thus be incorrect, as the language should have been “B does not equal C”.

The remaining changes relate to vocabulary choice for mathematical terms and directionality of statements. Consider the example of some element X which is in a set L. While typically it does not matter if we say “X is in L,” “X is an element of L,” “L has an element X,” and so on, it matters to the program due to the way axioms are defined and claims are constructed during translation. The translator always assumes that the set comes after the element, so the third statement from before, “L has an element X”, would actually translate as an element L in the set X. The other change to this language is similar to the removal of “it” and other indirect vocabulary – the membership operator is “element” rather than “in”, so the term “element” must be used to describe the relationship.

Usage is also limited due to Vampire being a first-order theorem prover. Due to this, it is not possible to call a function on a variable, meaning statements such as "The sum of X and Y equals Z" are unable to be run. This is, of course, a major limitation. At this time, there does not appear to be a resolution to this problem, greatly reducing the functionality of the system for some domains, namely any dealing with arithmetic. This does not affect functionality with questions of membership or comparisons as long as they may be phrased in such a way as to avoid said conflict.

In Table 4.1, inappropriate input sentences are shown alongside the same query input properly.

Incorrect Formatting	Correct Formatting
"X is less than Y which is less than Z so X is less than Z"	"X is less than Y, and Y is less than Z, thus X is less than Z'
"If a set A has an element B and B is not in C, then C and A are not equal"	"B is an element of D, and B is not an element of C, so C does not equal D"
"The following values are not equal: a, b, c"	"B does not equal C, and C does not equal D, and B does not equal D"

Table 4.1: Examples of proper and improper formatting.

4.3.2 Transitional TPTP File and Vampire

Following the parsing of the input claim, the program produces a TPTP format file containing selected axioms along with the generated conjecture. Axioms and conjectures within TPTP take the following form:

```
fof(subsetDef, axiom,((
(subset(X, Y))
& (element(A, X))
=> element(A, Y)))).
```

where *axiom* is used for an axiom, and *conjecture* for a conjecture.

Once this file has been created, Vampire is run on it. The program then outputs the results of the test, displaying either the successful run which details whether or not the conjecture was counter-satisfiable, or returning an error. Under typical runs, prior to the termination of the program the transitional file is deleted, but it is possible to execute the program in such a way as to leave the file for viewing. This is of course useful for determining why some input fails to execute or returns a questionable result, but it can also be convenient for seeing how these files are structured for execution.

Should a user desire to use the program for their work but find themselves needing axioms which have not been provided, they can easily add TPTP format files to the `axioms` directory, which contains all the axioms used by the program.

4.3.3 Axioms

Included with the software are three complete axiom sets along with a mostly implemented fourth set. The axiom sets are:

- Set theory
- Algebra
- Absolute geometry
- Geometry (Incomplete)

Absolute geometry refers to Euclidean geometry without the parallel postulate, which states that, should two straight lines be intersected by another, the side on which the sum of the interior angles created by these intersections is less than 180° will be the side on which the two lines will intersect [**?**]. Geometry in these context refers to the geometry defined by Hilbert's Axioms of Geometry [**?**].

Through examination of the files, a user can get an idea of how a new set of axioms would be created, both through examination of the format of each claim and the overall structure of the file.

To use additional axioms, a user need only create the file and put it within the `axioms` directory. All work of creating a master axiom file is performed by the program. This master file contains all axioms from within the directory, combined into a single file. For this reason, a user may not place their conjecture within their axiom file should they want to use it with the program.

Only TPTP formatted files are appended, so any other files, such as notes or work-in-progress axioms, may exist within the directory as long as they are saved with a different file extension.

The software checks for user determined words within the constructed list of claims to determine which axiom file to use as its base. While initial plans were for all axioms to be formed into a single file at runtime, it was found that some errors arose as a result of this. The first issue, which was solved, was a difference in parsing between the statements "X equals Y" and "X is a subset of Y," both of which contain their information in a format which the `NP_VP()` function, found in section 4.2.1,would read. In the subset claim, all pertinent information, X, Y, and "subset" are nouns, while in the other case "equals" is a verb. Thus, the information retrieved must be changed, in one case grabbing only nouns while the other retrieves both nouns and a verb.

The second issue was finding that, when running with too long of an axiom file (over 100 lines) all statements would simply be returned as true. This may have been due to some conflict between phrasing of certain axioms, but rather than change what was working, the axioms have instead been kept separate when evaluating conjectures. As end users do not see a difference during use, the change was considered to be a non-issue.

CHAPTER 5

Conclusion

This study serves as a proof-of-concept for the idea of a conversational theorem proving system more than it stands as a successful translator in its own right. The study has shown that even with basic techniques to parse language and a limited domain of questions, the program can still be useful for testing simple claims. More sophisticated language parsing systems, along with a more in depth understanding of automated theorem proving programs could make a robust, easy to use system for both students and mathematicians.

5.1 Failings

Progress on software was slowed primarily due to initially overly specific software design and difficulties with TPTP formatting. Initially, progress on the software appeared to be advancing very quickly as the problem domain was only that of set theory. The questions the software can handle are of the form "Is some X in Y?" expanded out based on relations between X, Y, and other sets. Unfortunately, having only a single tester during the design of the software led to a rigid structure for queries. Due to personal preference, a standard query would never be of the form "Does Y have some element X?" This led to easy design; if the term "in" or "element" were spotted, then the two terms within a certain distance could be taken as the variables which were being acted upon. This design, though, is not able to be generalized to new domains. Parts of speech were initally ignored, with the design based upon the general structure of these set theory queries. Once algebraic tests were to be performed, functionality immediately broke down. Initial changes were to continue adding more rules to a flawed system, rather than throwing everything out to create a proper foundation. This slowed down the progress which could have been made over several weeks to improve the parser.

Issues with TPTP and its relationship with Vampire are an area which seems to have problems on all sides, in many ways due to the target audience. The expected users for TPTP and Vampire are not undergraduate students, and do not appear to be graduate students either. They are designed with an expectation of knowledge that someone trained in using automated theorem proving systems would have, as well as those with experience with using first order logic. Vampire is simple to use, given that proper TPTP files are given to it, but it is not particularly clear what a good file is. Vampire intends to have a reference manual, but one has yet to be created. So, users are instead pointed to a 2013 academic paper on the software. The paper is under 15 pages in length, so it does not have the space to go into depth on what files should look like beyond more than very rudimentary examples. The TPTP webpage contains many dead links and necessarily vague instructions to ensure compatibility with a wide range of theorem provers. Personal lack of knowledge on the subject is a failing on the part of the author, but it seems only fair to have addressed the many unintentional hurdles put in place by a lack of clear documentation for both TPTP and Vampire.

5.2 Future Work

The most immediately clear expansion for this idea would be the incorporation of language analysis techniques which use machine intelligence. The issues faced here regarding more particular word choice, the addition of "and" between all clauses, and the ban on postfix function words could be eliminated. As a goal of this study was ease of use, being more accepting of language would be a great improvement.

A second avenue would be an expansion of subject domains. This could be accomplished through the addition of more axioms for further subjects, but simply because it requires no new technology does not mean the task is easy. To define anything in first order logic requires much more time than it appears to, especially as some claims must be rephrased to become several claims.

Cleaning up output to a human-readable form would be an amazing addition to the software, and theorem proving systems in general. While the provers helpfully put out what operations are performed in order to determine the truth value of a conjecture, this output is far from simple to read. Processing steps appear with what seem to be arbitrary numbers alongside them, refering to rules which a user may not have even included as reasoning for the action performed. If this output could be analyzed to provide greater clarity to what the prover has done, it would be possible to learn techniques from the program rather than simply be given a truth value.

The final addition which could make this software reach its full potential would be connection to digital assistants. With an always-on server ready to process input and run said input through Vampire, students could quickly check their work or answer questions they find interesting. Ideally, some sort of system would be put in place to prevent these results from being used to cheat on academic assignments, while still granting enough information for a student to follow the progression of the proof.

References

1. Global smart speaker shipments grew 187% year on year in q2 2018, with china the fastest-growing market, Aug 2018. URL `https://www.canalys.com/newsroom/global-smart-speaker-shipments-grew-187-year-on-year\-in-q2-2018-with-china-the-fastest-growing-market`.

2. Steven Bird and Edward Loper. Nltk: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, page 31. Association for Computational Linguistics, 2004.

3. Gobinda G Chowdhury. Natural language processing. *Annual review of information science and technology*, 37(1):51–89, 2003.

4. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

5. W John Hutchins. The georgetown-ibm experiment demonstrated in january 1954. 3265:102–114, 09 2004.

6. Karen Sparck Jones. Natural language processing: a historical review. In *Current issues in computational linguistics: in honour of Don Walker*, pages 3–16. Springer, 1994.

7. Craig S Kaplan. The halting problem. URL `http://www.cgl.uwaterloo.ca/csk/halt/`.

8. Steven L Lytinen. Dynamically combining syntax and semantics in natural language processing. In *AAAI*, volume 86, pages 574–587, 1986.

9. Donald Mackenzie. The automation of proof: A historical and sociological exploration. *IEEE Annals of the History of Computing*, 17(3):7–29, 1995.

10. Nitin Madnani and Bonnie J Dorr. Combining open-source with research to re-engineer a hands-on introductory nlp course. In *Proceedings of the Third Workshop on Issues in Teaching Computational Linguistics*, pages 71–79. Association for Computational Linguistics, 2008.

11. James Pustejovsky and Branimir Boguraev. Lexical knowledge representation and natural language processing. *Artificial Intelligence*, 63(1-2):193–223, 1993.

12. Panu Raatikainen. Gödel's incompleteness theorems, Jan 2015. URL `https://plato.stanford.edu/entries/goedel-incompleteness/`.

13. Philip Resnik. Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language. *Journal of artificial intelligence research*, 11:95–130, 1999.

14. Alan F Smeaton. Progress in the application of natural language processing to information retrieval tasks. *The computer journal*, 35(3):268–278, 1992.

15. Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966.

16. Terry Winograd. Procedures as a representation for data in a computer program for understanding natural language. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1971.

17. William A Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606, 1970.