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

Abstract

Include a short summary of your thesis, including any pertinent results. This section is *not* optional for the Mathematics and Computer Science or Physics Department ISs, and the reader should be able to learn the meat of your thesis by reading this (short) section.

This work is dedicated to the future generations of Wooster students.

Acknowledgments

I would like to acknowledge Prof. Lowell Boone in the Physics Department for his suggestions and code.

Contents

List of Figures

List of Tables

Table Page

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 was 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 the creation of a basic automated prover. The processor:

- Serves as a front end for any automated prover.
- Is usable without any understanding of symbolic notation or programming.
- Is modular enough to become the front-end to any automated proving system.
- Accurately translates standard English to symbolic notation.

The prover had far more modest aims:

- Understand basic questions in a limited scope, such as geometry or arithmetic.
- Take symbolic input, and return the truth value of said statement.
- Display the steps of the proof to the end user.

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 will 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. [4] 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. [15] 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.

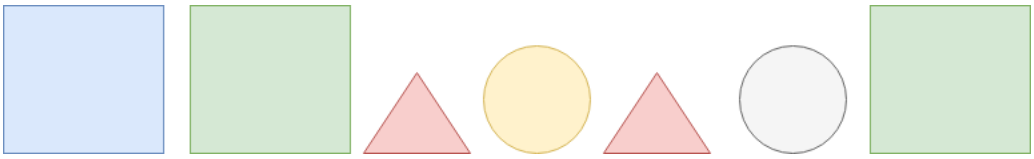

Figure 2.1: Sample SHRDLU starting world

> 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

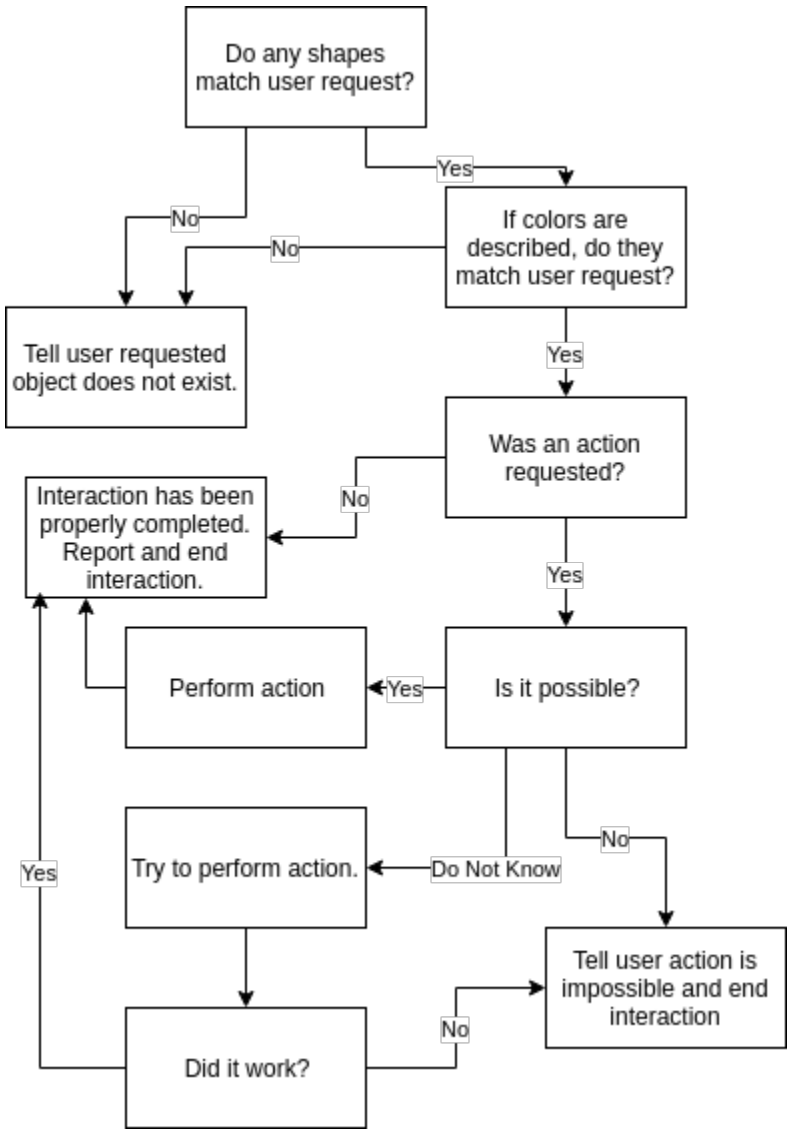

Figure 2.3: Flowchart for Figure 2.2

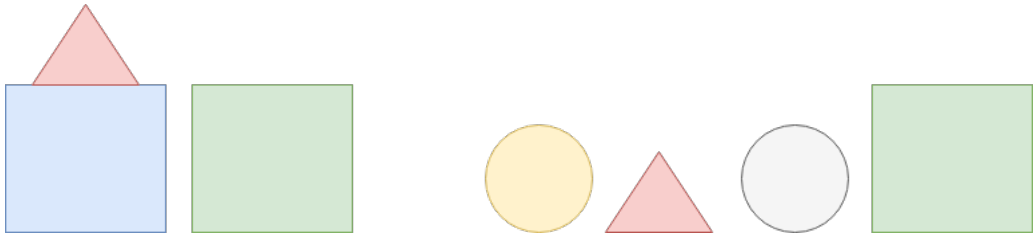

Figure 2.4: SHRDLU world after Figure 2.2

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

2.1.2 ELIZA

ELIZA is an early language based program which took user input and responded much like a Rogerian psychologist [14]. 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

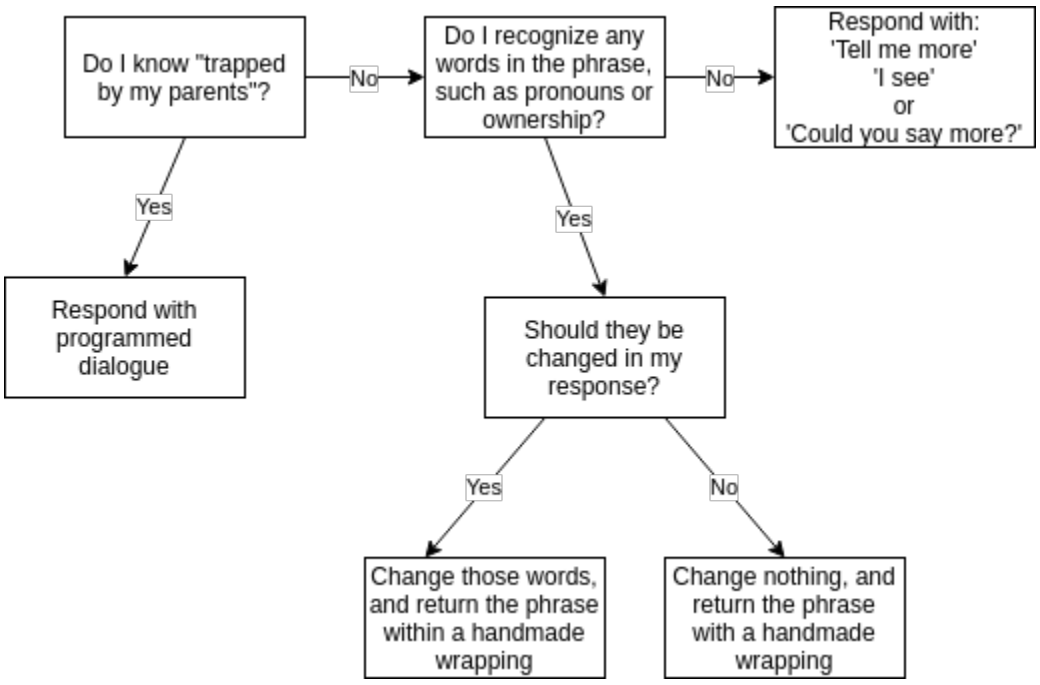

Figure 2.6: Flowchart for dialogue in 2.5

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 grammar. 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:

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

Figure 2.7: Two production rules, one of which produces a variable and one of which produces 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 variables or terminals which may replace it.
- 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:

- $\langle S \rangle \rightarrow \langle NP \rangle \langle VP \rangle$
- $\langle NP \rangle \rightarrow \langle DET \rangle \langle N \rangle$
- $\langle VP \rangle \rightarrow (\langle ADVB \rangle) \langle VB \rangle$
- $\langle N \rangle \rightarrow$ 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 [16]. 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. 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 [16].

The 1980s, with 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 [5].

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 [7].

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 [7].

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 see 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".

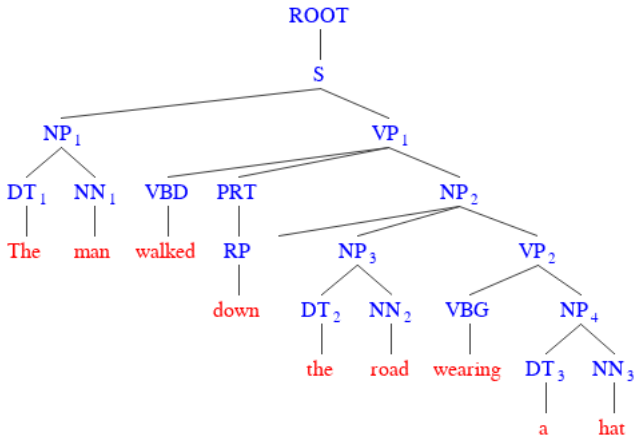

Figure 2.11: An incorrect parsing of "The man walked down the road wearing a hat"

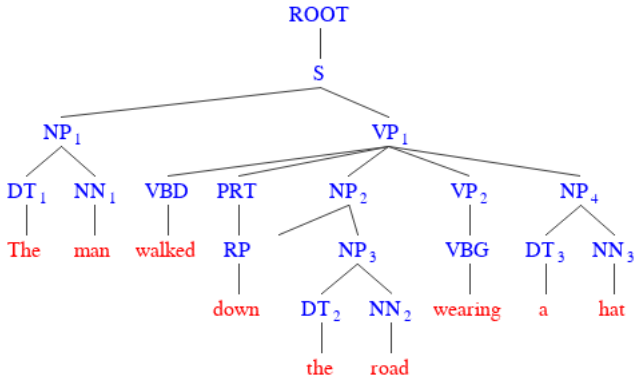

Figure 2.12: The correct parsing of "The man walked down the road wearing a hat"

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 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. 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 [8].

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 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 the 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 [8]!

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. [6] □

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. [11]*

While both of these theorems are important, we will be focusing on Theorem 3. 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 analyzes 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. The program produced as part of this study was to be interaction free, following the user providing axioms and a conjecture to prove. This allowed for a user to be unsure how a proof needed 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.

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* [**?**].

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 straight-forward,

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 complimentary literals. The term "complimentary" is used here to indicate that the two literals are the negation of one another, for example, c and $\overline{c}$. The clause which results contains all literals from each of the component clauses which did not contain a compliment 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 complimentary. 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.

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 complimentary literals. Repeated literals are removed after each use of the rule to reduce the size of the set.

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

   (b) If it does not contain complimentary 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 essentially a refutationally complete calculus for equational first-order logic. As long as the strategies given are fair and resources unlimited, a system which uses computational superposition is guaranteed to derive any unsatisfiable claim.

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. 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.

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 the monoid $\langle x, y | x^3 = y^3 = (xy)^3 = 1 \rangle$. 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. $x^3 \to 1$
2. $y^3 \to 1$
3. $(xy)^3 \to 1$

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. Obsoleted simply means that the rule may be simplified. We desire all reduction rules to be in their simplest 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. $y^2x^2 \to xyxy$

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 $\sim p$ and $\sim q$ and $\sim r$ 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 that the negated claim is logically valid with 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 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.

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

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

5. 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.

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

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

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

9. 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.

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

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

12. 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.

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

14. 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.

15. 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.

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