Translating LaTeX to Coq: A Recurrent Neural Network Approach to Formalizing Natural Language Proofs

A thesis presented to
the Honors Tutorial College
Ohio University

In partial fulfillment
of the requirements for graduation
from the Honors Tutorial College
with the degree of
Bachelor of Science in Computer Science

Benjamin A. Carman

May 2021

This thesis titled

Translating LaTeX to Coq: A Recurrent Neural Network Approach to Formalizing Natural Language Proofs

by

BENJAMIN A. CARMAN

has been approved by

the Honors Tutorial College and

the School of Electrical Engineering and Computer Science

Dr. David Juedes

Chair, Professor, Computer Science

Thesis Advisor

Dr. David Chelberg

Director of Studies, Computer Science, Honors Tutorial College

---

Dr. Donal Skinner

Dean, Honors Tutorial College

Abstract

CARMAN, BENJAMIN A., B.S., May 2021, Computer Science

Translating LaTeX to Coq: A Recurrent Neural Network Approach to Formalizing Natural Language Proofs (48 pp.)

Director of Thesis: Dr. David Juedes

There is a strong desire to be able to more easily formalize existing mathematical statements and develop machine-checked proofs to verify their validity. Doing this by hand can be a painstaking process with a steep learning curve. In this paper, we propose a model that could automatically parse natural language proofs written in LaTeX into the language of the interactive theorem prover, Coq, using a recurrent neural network. We aim to show the ability for such a model to work well within a very limited domain of proofs about even and odd expressions and exhibit generalization at test time. We demonstrate the model's ability to generalize well given small variations in natural language and even demonstrate early promising results for the model to generalize to expressions of intermediate lengths unseen at training time.

Acknowledgments

I would like to offer my deepest thanks to my advisors on this project, Dr. David Juedes, with whom I've had the lucky opportunity to take two tutorials during my first year at Ohio University. His practical, project-focused teaching and wisdom from years of research experience laid the groundwork for not only fundamental skills in algorithms but also in LaTeX and Coq that I needed to complete this project. He has been a mentor and role model to me from the start, always pushing my abilities to be the most critically thinking scientist I can be. He has always been available and willing to offer his wisdom to help me tackle roadblocks—empowering me with tools and techniques to debug similar issues on my own in the future. This project of translating LaTeX to Coq has always been a dream of his and I am very grateful to have tackled such an important, interesting problem together.

Next I would like to thank Dr. Razvan Bunescu who has taught me nearly everything I know about machine learning and deep learning. I have been lucky to take two fascinating courses and two exciting tutorials with him during my Junior year of college. He is an exceptional teacher who finds the perfect blend of theory and fundamentals with practical application—enabling me to understand and implement machine learning models end-to-end. His dedication to research and AI applications is infectious and continues to inspire me to find new ways we can apply math and science to advance each discipline and provide a benefit to society.

Finally, I would like to thank the Honors Tutorial College for making this thesis possible by allowing me to have space in my curriculum to discover and explore my interests and passions in math and computer science. I have had a challenging and engaging curriculum filled with opportunities for experiential learning thanks to HTC.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Background

Many mathematical proofs are commonly written using natural language and the typesetting system LaTeX. However, there is a strong desire to formally express mathematical statements and develop machine-checked proofs to ensure their validity. One of the most common tools for doing this is the interactive theorem prover: Coq. However, Coq poses a steep learning curve before being able to convert natural language mathematical statements into a Coq proof successfully, thus leaving its use somewhat inaccessible to mathematicians who do not possess the time or resources to learn such a system. Still, Coq could pose a great benefit to the broad community of mathematicians in ensuring that past and future proofs are truly correct and thorough, as natural language proofs are highly susceptible to human error. With Coq, there is no room to simply skip a logical step because it is "obvious" to the mathematician. With this system, proofs must be formal, rigorous, and above all—valid.

Luckily, LaTeX is a document preparation system that marries natural language with explicit tokens to express mathematical symbols. Thus, proofs written in LaTeX are already rich with information that a computer can easily parse and learn to understand. Here, we propose the possibility that a machine could learn to automatically translate these rich LaTeX proofs into Coq commands that can be compiled and checked for correctness. This could allow mathematicians to reap the benefits of Coq's formal proof expression and machine verification without having to invest the time and resources into developing the Coq representation completely from scratch. In order to do this, we will employ a recurrent neural network on pairs of proofs written in both LaTeX and Coq from a small domain to develop a model that can correctly translate similar proofs unseen during training. We demonstrate the ability for the model to learn higher-level patterns in the

input data to generalize in various ways on proofs unseen at training time. Demonstrating small generalizations like this opens the doors for countless future developments in this application area—to design systems that can correctly translate a wider variety of far more advanced theorems and proofs.

In this report we outline the approach we use and the datasets we have developed to demonstrate interesting generalization performance. We also evaluate an implementation of the architecture in PyTorch along with discussion on what the model currently learns to do very well and where the current implementation struggles.

1.2 Problem Definition

The model proposed in this project seeks to use a sequence-to-sequence approach to take an input sequence of natural language LaTeX tokens from a limited domain of proofs and produce a well-formatted output sequence of tokens in Coq that correctly expresses the same proof as the input. For example, we would like the model to take the following LaTeX proof (after tokenization) as input:

```
\begin{document}
\newtheorem{theorem}{Theorem}[section]
\begin{theorem}
Given any natural number J, $3006*J + 659$ has to be odd.
\end{theorem}
\begin{proof}
It is known that 3006 must be even and 659 must be odd. Hence
3006*J has to be even. Definitely, $3006*J + 659$ is odd, due
to the fact that summing an odd number to an even number
results in an odd number.
\end{proof}
\end{document}
```

The model should then produce an equivalent Coq proof like the following:

```
Require Import Arith.
Lemma L3006_659: forall J:nat, Nat.odd(3006 * J + 659)=true.
Proof.
intros.
```

```
assert (HY3006:Nat.even(3006)=true).
{
        simpl.
        reflexivity.
}
assert (HY3006J: Nat.even(3006*J)=true).
{
        rewrite Nat.even_mul.
        rewrite HY3006.
        simpl.
        reflexivity.
}
assert (HY659:Nat.odd(659)=true).
{
        unfold Nat.odd.
        simpl.
        reflexivity.
}
rewrite Nat.odd_add.
rewrite HY659.
rewrite <- Nat.negb_even.
rewrite HY3006J.
simpl.
reflexivity.
Qed.

Theorem T3006_659:forall J:nat, Nat.Odd(3006 * J + 659).
Proof.
intros.
apply Nat.odd_spec.
apply L3006_659.
Qed.
```

We will consider the model to have produced a correct translation of the LaTeX proof when the following criteria are met: 1. The network produced a correct, matching translation for the LaTeX proof statement. 2. The produced Coq proof compiles and runs by the Coq interpreter flawlessly until “Qed” is reached, without any “Admitted” statements. Thus, a successful model would be able to automatically translate LaTeX proofs into Coq proofs making the formalization and verification of mathematics much

easier and more accessible. By demonstrating this success on a limited domain of proofs and showing the ability for the model to generalize at test time, this paves the way for future models that can formalize a much wider variety of novel, natural language proofs.

Chapter 2: Literature Review

Here we provide a summary of related work that helped inform both the problem and the neural network architectures used to solve it in this project. Work on this specific task of translating LaTeX to Coq has not yet been done. As this is a novel task, we will focus on work that has been done on very similar tasks and explain what has made them successful. First, we explore developments in the literature on the general task of mathematical parsing and formal proof. Next, we consider recent developments of neural network approaches since their first successful application to NLP problems. Finally, we draw on papers that use neural networks to solve semantic parsing problems and relate them back to how they might best be used to convert LaTeX to Coq.

2.1 Formal Proof and Mathematical Parsing

A formal proof is a proof written in a precise, artificial language and has a specific grammar such that it is interpretable by a machine. This language is usually designed so that a mechanized process can check the proof for correctness. Coq is one such example of an interactive theorem prover as it mechanizes exactly this process of checking formal proofs for correctness. Harrison [Har08] argues that the formalization of math proofs like those in Coq is extremely important and valuable for two primary reasons. First is due to its ability to supplement or replace aspects of peer review for mathematical papers by checking proofs automatically. This would in turn make math much more precise and reliable. The second reason is for the ability of formalization to extend rigorous proofs to the verification of programs. He explains how the latter is of growing value as computer programs are ubiquitous in daily life and in many safety-critical systems. Being able to apply formal proof to verify the correctness of a program ensures safety by eliminating our reliance on extensive software testing. Harrison clearly shows the importance of formal proofs while recognizing it has attracted little interest from the mathematical

community at large because creating them is “difficult and painstaking” [Har08]. Still, he remains optimistic as progress continues to be made in making this process more automatic, convenient, and accessible so more mathematicians can “put the correctness of their proofs beyond reasonable doubt” [Har08].

Clearly, there is a great benefit to formalizing mathematics in such a way that it is verifiable by a machine, but we also have the problem that many mathematical results don’t even exist in an explicit machine-interpretable format. Many proofs are written in natural language using LaTeX, a format that leaves out a lot of information that can be reconstructed by the human audience. Stuber, et al. [SVDB03] considered this problem as they wrote software to try and parse LaTeX into MathML, a standard that allows machines to work with the semantics of formulas. They quickly realized the numerous challenges of this as certain LaTeX expressions could take on multiple meanings. For example, if the ‘*’ operation is omitted, the expression $w(a + b)$ could be the multiplication of w by $a + b$ or it could be the application of a function named w. This requires them to consider the proof’s context to best type these variables. Similarly, expressions like $\sin(ax)\cos(bx)$ would usually be assumed to mean $\sin(ax) * \cos(bx)$ but is ambiguous in its original notation. Because of this, they design a fairly complex grammar to parse the LaTeX and must make a number of manually engineered decisions on how to handle such ambiguities.

This process Stuber, et al. [SVDB03] go through is highly similar to our goal of semantically parsing LaTeX documents into the formal language, Coq. However, in order for their model to be successful, they had to manually engineer numerous rules to parse types of mathematical statements and their ambiguous variations. This makes it impossible for their architecture to generalize to more complex proofs or varieties of style without additional human engineering. Our goal with this project is to use neural networks to automatically recognize these patterns of ambiguous expressions and their literal mappings in Coq without engineering rules for them by hand. If this were to be

successful in a limited domain, it transforms the bottleneck for the architecture's intelligence from one of engineering rules by hand to one of collecting sufficient data. With enough high-quality paired examples of LaTeX and Coq proofs, theoretically our model should be able to learn these rules automatically, even on more advanced mathematical expressions and proof techniques.

The main limitation here is that while a model could learn to generalize by combining proof techniques from various pairs of training data, any Coq tactic that must be generated at test time has to be seen during training alongside rich, highly correlated natural language in LaTeX. Thus, rich data creation is one of the main priorities and bottlenecks for a system like this. But with such data, analysis can be done to see just how much an NLP-based system can learn to generalize proof translation.

2.2 Recurrent Neural Networks and Transformer

Our task of parsing natural language LaTeX proofs and translating them into formal Coq proofs is one of natural language processing (NLP), that is best represented using a sequence-to-sequence architecture where a neural network encodes a source sentence (the LaTeX proof) and uses that learned representation to decode a target sentence (the Coq proof). One of the main architectures used to do this is a recurrent neural network (RNN). RNNs first achieved great success on translation tasks by Bahdanau et al. [BCB16] through the use of an attention mechanism. The use of this mechanism alleviates the burden of encoding an entire source sequence into a fixed length vector by allowing the model to learn which tokens in a source sentence are most relevant for predicting the target word. This mechanism allowed RNNs to perform as well as conventional phrase-based translation mechanisms. Luong et al. [LPM15] further expanded on this mechanism by showing how an attention mechanism could be used locally, opening the door for the translation of long sentences with a reasonable amount of training time. Here,

they also proposed alternative methods for the computation of attention scores, from using dot products between encoder and decoder states, products between states with an additional matrix of parameters in between them, and a matrix of parameters multiplied by a concatenated vector of encoder and decoder states. Combining several of these new approaches, they demonstrate the capability for RNNs to achieve truly state-of-the-art test performance on translation tasks. This shows the high promise for RNNs to learn the relationships between LaTeX and Coq proofs even with stylistic and structural differences in natural language proofs as attention can help the model to focus on the most important parts.

However, attention won't be enough for our model to be successful on this task. There is also the problem of out-of-vocabulary (OOV) tokens like variable names or values that must be properly transferred to the Coq proof. Gu et al. [GLLL16] solves this through the introduction of a copying mechanism that allows the decoder to determine whether to generate a vocabulary token or copy a token from the input sequence based on a learned distribution. A similar mechanism with better performance was proposed by Gulcehre et al. [GAN+16]. However, the copying mechanism was further simplified by Chen et al. [CB19] using a method where the decoder is trained to generate special tokens <REF> when a token is expected to be copied from the input. They show how this <REF> token can then be translated into a token from the source sentence via a separate logistic regression mechanism.

While an RNN with attention and copying mechanisms is likely to be capable of learning the desired relationships between natural language and Coq proofs well enough to have good test performance, there is another architecture that has seen even better test performance and training efficiency for translation tasks. This architecture, known as the transformer was proposed by Vaswani et al. [VSP+17] and is based solely on attention—completely eliminating the need for recurrence. This means that this

architecture is both highly parallelizable as well as accurate at test time because it better captures global dependencies in the data. Developments have even been made to allow transformers to learn dependencies forward and backward in time leading to even better test performance [DCLT19]. Transformers have seen such great success on translation and generation tasks in the recent literature, that it is likely another great candidate worth implementing and comparing for this task of converting LaTeX to Coq.

2.3 Semantic Parsing with Neural Networks

More specifically, our task of converting LaTeX to Coq isn't one of typical machine translation like those used to test RNNs and transformers in all the literature discussed so far. Rather, this task is called semantic parsing as we are parsing natural language into a machine-readable representation. These types of tasks have also been considered previously in the literature and have led to variations on the vanilla RNN's architecture. Dong et al. [DL16] proposed a structure that is more faithful to the compositional nature of logical forms by using a sequence-to-tree model. In this model, the decoder is changed to be allowed to generate nonterminal tokens. The hidden states associated with these tokens are then used to instantiate another decoder sequence. This creates a hierarchical tree structure and helps the model to produce a more well-formed output considering the strict format requirements of machine representations.

In another paper [DL18], the same authors further improve their model for the task of semantic parsing by breaking the decoder into two stages. In the first, the decoder produces a rough sketch of the meaning representation without details like arguments or variable names filled in. In the second, they condition the decoder on both the encoder and the sketch itself to fill in the details. This structure allows the model to disentangle high-level and low-level information as opposed to trying to decode the sequence in one pass.

Through each of these modifications, the authors create RNN models that have even better test performance on the task of semantic parsing. As our project involves creating machine representations with a certain hierarchy and strictness of formatting to them, some of these proposed techniques could be helpful to generate a Coq proof that is both correct and able to be compiled, however we take a simpler approach for now.

Chapter 3: LSTM-Based Model

In order to create this model, we first implement a Long Short-Term Memory (LSTM) recurrent neural network in PyTorch. Each LaTeX proof will be fed into the network as a series of tokens in the encoder using word embeddings that will be trained simultaneously. The decoder will be initialized using the last cell state of the encoder multiplied with a matrix of learned parameters. An attention mechanism like those explored in [LPM15] will then be used for each position in the decoder as tokens are produced.

A simple copying mechanism based on the one described in [CB19] will also be used between the encoder and the decoder. In the encoder, when tokens involving numbers and variables are reached, they will be given word embeddings corresponding to generic tokens, <num> and <var>. During the decoder stage, after the very last hidden layer the model will choose whether to generate a standard token from the vocabulary or utilize the copying mechanism to copy a corresponding number or variable into that position. A similar mechanism will be used solely within the decoder stage. Special tokens like <genP> and <genH> will be used to represent the generation of a proof name or hypothesis name in Coq allowing these names to be generated in a predetermined way. Later, the model will be able to copy and refer back to these tokens. When a proof or hypothesis name is copied from the decoder, the corresponding word embedding for a generic token <refP> or <refH> is fed into the next decoder state. Finally, after computing an output sequence during each epoch of training, the model will perform backpropagation using a standard cross entropy loss.

An overview of the model's architecture is provided in Figure 3.1 with the model's architecture detailed throughout the remainder of the chapter.

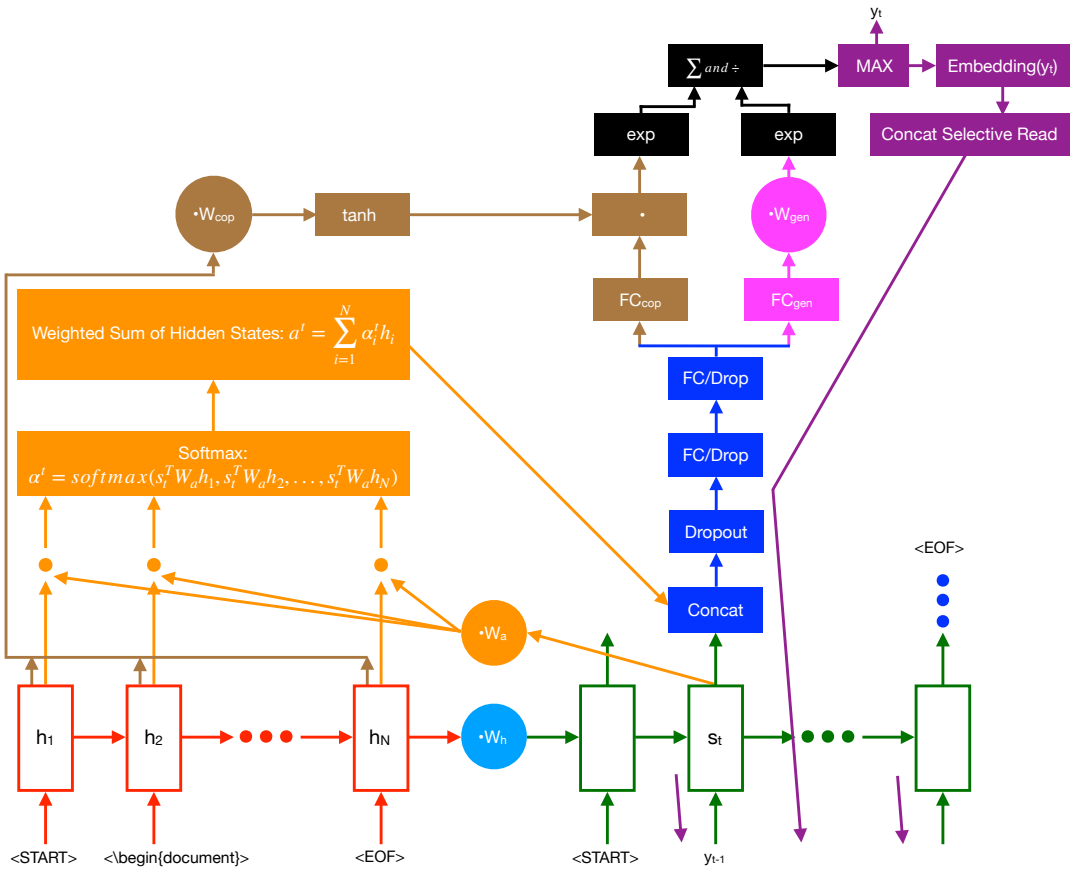

Figure 3.1: Diagram of LSTM-based architecture with attention and copying mechanism

3.1 LSTM Architecture

The foundation of the model's architecture is an encoder-decoder architecture of LSTM cells based on the LSTM design in [SSB14]. A Long Short-Term Memory architecture is used due to its enhanced ability to model the context of a sequence at greater distances. Traditional RNNs suffer significantly from vanishing and exploding gradients and lack the ability to model dependencies more than 5-10 time steps away. LSTMs address these issues through the use of special units known as memory blocks in each hidden layer. Input, forget, cell, and output gates help to control activations and pass along information from early in the sequence. Each of these formulas for the LSTM cells

in our model are as follows:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg})$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = o_t \odot \tanh(c_t)$$

where h_t is the hidden state at time step t, c_t is the cell state at time step t, and x_t is the input (in our case the corresponding word embedding of the token at time step t concatenated with a selective read vector). Then, each equation for i_t, f_t, g_t, and o_t correspond to the input, forget, cell, and output gates at time step t. σ is the sigmoid function and $\odot$ is the element wise product. Each W represents a matrix of learned parameters and each b is a learned bias.

Notice that in Figure 3.1, the encoder is represented by the red component of rectangles. The last hidden state is then multiplied by a matrix of parameters, W_h, and fed into the first hidden state of the decoder which is represented by the green component in the figure.

3.2 Attention Mechanism

The attention mechanism is used at each state of the decoder to allow the model to learn to pay close attention to the most relevant states in the encoder when choosing a token to output. This attention mechanism is modeled by the orange component of Figure 3.1 and implements the following two formulas:

$$\alpha^t = softmax(s_t^T W_a h_1, s_t^T W_a h_2, ..., s_t^T W_a h_N)$$

$$a^t = \sum_{i=1}^{N} \alpha_i^t h_i$$

First, the hidden state of the decoder is multiplied by a matrix of parameters, W_a. It is then multiplied by each hidden state of the encoder and a softmax is performed. This yields a probability distribution over the states of the encoder which tells the model what states to pay closest attention to. Using this distribution, we perform a sum of the hidden states weighted by that distribution. This yields an attention vector which is concatenated with the output of the LSTM cell at that time step in the decoder. After this, the model goes through a dropout layer and then two fully connected layers, each with dropout performed after them.

3.3 Copying Mechanism

After the final fully connected/dropout layer on top of the hidden decoder state, this is where the model would typically perform a softmax to determine which token should be generated from the Coq vocabulary. In our case, the model must choose not just from a vocabulary of Coq tokens to generate, but it also must choose from a copying vocabulary consisting of tokens in the input sequence. This way, the model will have the ability to copy natural number and variable tokens from the input.

In order to support copying, we feed each natural number in the input sequence to the encoder as the embedding for a generic token <nat> and each variable as the embedding for a generic token <var>. The model then computes a score for each token that could be generated from the Coq vocabulary, $\mathcal{V}_c$, and each token that can be copied from the input sequence, $\mathcal{X}$, separately. These scores are denoted $\psi_g(\cdot)$ and $\psi_c(\cdot)$ respectively. The score for a given token, $y_t \in \mathcal{V}_c \cup \mathcal{X}$ is computed in the following way:

$$\psi_g(y_t) = \mathbf{v}_{y_t}^\top \mathbf{W}_{gen} \mathbf{l}_t^{gen}$$

$$\psi_c(y_t = x_j) = \tanh\left(\mathbf{h}_j^\top \mathbf{W}_{cop}\right) \mathbf{l}_t^{cop}$$

where $\mathbf{l}_t^{gen}$ is the output of the last layer of the decoder, FC_{gen} for generation mode, on top of the state $\mathbf{s}_t$ at time t. Similarly, $\mathbf{l}_t^{cop}$ is the output of the last layer of the decoder for copying mode, FC_{cop}. $\mathbf{v}_{y_t}^\top$ is the one hot vector corresponding to the token y_t and $\mathbf{h}_j^\top$ corresponds to the hidden state of the last layer of the encoder for $x_j \in \mathcal{X}$. The computation of the scores for generation mode is shown in Figure 3.1 by the brown component. The generation mechanism is shown by the pink component.

A softmax is then performed over the union of each set of scores as shown by the black colored component in Figure 3.1. The computation of this softmax results in a probability distribution over $\mathcal{V}_c \cup \mathcal{X}$ and gives a probability for each token in this set, computed as follows:

$$p(y_t) = \begin{cases} \frac{1}{Z_t} e^{\psi_g(y_t)} & , \quad y_t \in \mathcal{V}_c \\ \frac{1}{Z_t} \sum_{x_j \in X : x_j = y_t} e^{\psi_c(y_t = x_j)} & , \quad y_t \in \mathcal{X} \end{cases}$$

$$Z_t = \sum_{y_t \in \mathcal{V}_c} e^{\psi_g(y_t)} + \sum_{x_j \in X} e^{\psi_c(y_t = x_j)}$$

Note in the above computation that we anonymize the position of each copyable token from the input sequence $\mathcal{X}$ by summing together the probabilities for each repeated token in the sequence.

As shown in the purple component of Figure 3.1 we can then output the token with the maximum probability from this distribution as y_t. We then compute the embedding for this token. If the token was a natural number or variable we use the embedding for the corresponding generic token, <nat> or <var>. We then concatenate a selective read vector as described in Section 3.4 and feed it into the next hidden state, s_{t+1}. This process continues until an <EOF> token is generated by the decoder.

3.4 Selective Read

We additionally employ the use of a selective read vector when providing input to any given state, s_t, in the decoder, which ultimately helps the copy mechanism to get a sense of position within the text. This input vector is represented as $[\mathbf{e}(y_{t-1}); \zeta(y_{t-1})]^\top$ where $\mathbf{e}(y_{t-1})$ is the word embedding for y_{t-1} and $\zeta(y_{t-1})$ is the selective read vector. If y_{t-1} is copied from the input, $\zeta(y_{t-1})$ is the sum of hidden states corresponding to that copied token, weighted by the copy probabilities for each hidden state. This vector is $\mathbf{0}$ if y_{t-1} is not copied from the source sentence. Let M denote the set of hidden states $\{\mathbf{h}_j \mid x_j \in X\}$ from the encoder and the computation of $\zeta(y_{t-1})$ is as follows:

$$\zeta(y_{t-1}) = \sum_{j=1}^{|M|} \rho_{tj} \mathbf{h}_j$$

$$\rho_{tj} = \begin{cases} \frac{1}{Z} p(y_{t-1} = x_j \in \mathcal{X} \mid \mathbf{s}_{t-1}, M), & x_j = y_{t-1} \\ 0 & \text{otherwise} \end{cases}$$

$$p(y_{t-1} = x_j \in \mathcal{X} \mid \mathbf{s}_{t-1}, M) = \frac{1}{Z_{t-1}} e^{\psi_c(y_{t-1} = x_j)}$$

$$Z_{t-1} = \sum_{x_k \in X : x_k = y_{t-1}} p(y_{t-1} = x_k \mid \mathbf{s}_{t-1}, M)$$

Chapter 4: Datasets

4.1 Standard Theorems and Proofs Over Even/Odd Expressions

Our primary focus is in testing this model on proofs about the parity of expressions of the form $a * x + b * x + ... + c$. A complete example of the LaTeX input and Coq output for this type of proof is given in Section 1.2. In an effort to help the model to generalize the patterns seen in these types of proofs, we try to generate as diverse of LaTeX inputs as possible. We achieve this by implementing a Backus-Naur form (BNF) grammar using Rust. For the sake of brevity, we show the types of diversity used in the theorem statements with a grammar here and provide a more complete grammar for the theorems with their corresponding proofs in Appendix A. Shown next is the BNF grammar used for the LaTeX statements of the theorem:

⟨*theoremStatement*⟩ ::= ⟨*forEvery*⟩ ⟨*varList*⟩ ⟨*expression*⟩ ⟨*declaration*⟩

⟨*forEvery*⟩ ::= For every natural number
| For any natural number
| Given any natural number
| Given a natural number

⟨*varList*⟩ ::= ⟨*VAR1*⟩
| ⟨*VAR1*⟩ and ⟨*VAR2*⟩
| ⟨*VAR1*⟩, ⟨*VAR2*⟩ and ⟨*VAR3*⟩
| ...

⟨*expression*⟩ ::= $ ⟨*NAT1*⟩ * ⟨*VAR1*⟩ + ⟨*NAT2*⟩ $
| $ ⟨*NAT1*⟩ * ⟨*VAR1*⟩ + ⟨*NAT2*⟩ * ⟨*VAR2*⟩ + ⟨*NAT3*⟩ $
| $ ⟨*NAT1*⟩ * ⟨*VAR1*⟩ + ⟨*NAT2*⟩ * ⟨*VAR2*⟩ + ⟨*NAT3*⟩ * ⟨*VAR3*⟩ + ⟨*NAT4*⟩ $
| ...

⟨*declaration*⟩ ::= is odd
| must be odd
| is an odd number

| has to be odd
| has to be an odd number
| must be an odd number
| is even
| must be even
| is an even number
| has to be even
| has to be an even number
| must be an even number

Provided LaTeX theorem statements generated using the above grammar, a correct, corresponding Coq proof is produced. These Coq proofs follow the same format throughout the dataset, but are made to uniquely match the expression used in the LaTeX proof.

4.2 Diverse Theorems Over Even/Odd Expressions

One of the goals of this project is to get the model to generalize to expressions of unseen lengths during training. As will be shown in Chapter 5, the model struggled to do this using the dataset explained in Section 4.1. We believe one of the reasons for this was due to a lack of diversity in the training data. In particular, we believe the model may overfit to the length seen in training, especially when the length of the input sequence so closely corresponds to the length of the expression. To alleviate this, we created a more diverse dataset of theorem statements, particularly in terms of the lengths of natural language expressions used and the ordering of various phrases and mathematical expressions. This was created to help the model stop learning to fit to the exact lengths and positions consistently seen in training. The adjusted grammar for this dataset is provided here:

⟨*theoremStatement*⟩ ::= ⟨*forEveryNat*⟩ ⟨*varList*⟩ ⟨*expression*⟩ ⟨*adverb?*⟩ ⟨*declaration*⟩
| ⟨*adverb?*⟩ ⟨*expression*⟩ ⟨*declaration*⟩ ⟨*forEveryNat*⟩ ⟨*varList*⟩

| ⟨*adverb?*⟩ ⟨*expression*⟩ ⟨*declaration*⟩ ⟨*forEvery*⟩ ⟨*varList*⟩ ⟨*givenNat*⟩

⟨*forEveryNat*⟩ ::= for every natural number
| for any natural number
| given any natural number
| given a natural number
| for any
| for every
| given
| for any natural
| given that the following variables are natural
| given that the following are natural
| given that the following are natural
| provided that the following are natural numbers
| given the following natural numbers
| given any
| given the following list of natural number variables

⟨*forEvery*⟩ ::= for any
| for every
| given
| given any

⟨*givenNat*⟩ ::= given that they are natural
| provided that they are natural
| that is natural
| that is a natural number

⟨*adverb?*⟩ ::= ⟨*adverb*⟩
|

⟨*adverb*⟩ ::= clearly
| obviously
| of course
| certainly
| definitely

⟨*declaration*⟩ ::= is odd
| must be odd
| is an odd number
| has to be odd
| has to be an odd number
| must be an odd number
| evaluates to an odd result
| must evaluate to be an odd result
| must evaluate to be odd
| when simplified, has to be an odd number
| when simplified, must be odd
| is an odd number when simplified
| ... (repeat for even)

⟨*varList*⟩ ::= ⟨*VAR1*⟩
| ⟨*VAR1*⟩ and ⟨*VAR2*⟩
| ⟨*VAR1*⟩, ⟨*VAR2*⟩ and ⟨*VAR3*⟩
| ...

⟨*1Term*⟩ ::= NAT1 ⟨*mult*⟩ VAR1
| VAR1 ⟨*mult*⟩ NAT1

⟨*2Terms*⟩ ::= ⟨*1Term*⟩ + NAT2 ⟨*mult*⟩ VAR2
| ⟨*1Term*⟩ + VAR2 ⟨*mult*⟩ NAT2

⟨*...Term*⟩ ::= ⟨*...Term*⟩ + ... ⟨*mult*⟩ ...

⟨*expression*⟩ ::= $ ⟨*1Term*⟩ + NAT2 $
| $ ⟨*2Term*⟩ + NAT3 $
| $ ⟨*...Term*⟩ + NAT... $
| ...

⟨*mult*⟩ ::= *
| cdot

Chapter 5: Experimental Evaluation

5.1 Methodology

Our experimental methodology consists of training and testing the model described in Chapter 3 using the datasets described in Chapter 4. Our model is implemented in PyTorch, with careful attention paid to the vectorization of the attention, copying, and selective read mechanisms to allow the model to train as quickly as possible. Each dataset is generated randomly from its corresponding Backus-Naur form grammar with 1000 examples in the training set, 150 examples in the validation set, and 150 examples in the test set. We train until we consistently see no improvement in the sequence-level accuracy on the validation data. This sequence-level accuracy is determined based on the number of generated Coq proofs that exactly match each token in the generated Coq proof. We also compute token-level accuracy which is the number of positions in the generated Coq proof with the correct token. This is useful to get a sense of the model's performance even when sequences are not generated perfectly. To monitor each experiment's hyperparameter setting, loss, and accuracy, we use the web-based tool, Weights & Biases [Bie20].

In addition, we implement both teacher forcing and beam search in the model. The use of teacher forcing means that, during training, the token embedding used as input at each state in the decoder is the correct token that should have been produced at time $t-1$ regardless of which token was produced by the model. Beam search is used at validation and test time. This means throughout each step of the decoder, the model maintains k beams of sequences with the highest probabilities. Sometimes, this results in a sequence with a higher probability overall than the model would have produced if each token were chosen greedily at every step.

5.2 Results

Here we describe the results from our best performing model on several datasets. In each case, we performed a fairly extensive hyperparameter search manually to discover what sizes of models would perform best on each task.

5.2.1 Standard Theorems and Proofs Over Even/Odd Expressions

First, we test the model on the full dataset of theorems and proofs about even/odd expressions as explained in Section 4.1. Here, the model sees expressions of lengths 2 through 10 terms. Each of these lengths are seen in all of training, validation, and testing. The main distinction between examples seen at training and test time is in its choices and orderings of natural language. Here, we are able to achieve a 96.7% sequence-level accuracy at test time. This demonstrates the model's ability to properly recognize the patterns necessary to correctly translate variations of natural language proofs to Coq—when the number of terms in the expression is seen at training (2-10). This result allows us to move on to more sophisticated forms of generalization, particularly in terms of expression length.

5.2.2 Standard Theorems Over Even/Odd Expressions Alone with Varying Length

Next, the model was tested on the same dataset described in 4.1 but only using the theorem statements. The primary objective of using this dataset was to see the model generalize to intermediate lengths. Thus, the dataset was split into training, validation, and testing with the following properties:

Table 5.1: Standard Theorems Dataset Breakdown by Expression Length

Dataset	**# Examples**	**Expression Lengths (# Terms)**
Training	1000	3, 5, 7, 9
Validation	150	3, 4, 5, 7, 8, 9
Test	150	2, 3, 4, 5, 6, 7, 8, 9, 10

The best results on this dataset were achieved using the following hyperparameters, with the learning rate manually reduced over time:

Table 5.2: Best Experimental Results on Standard Theorems with Model Hyperparameters

Parameter	Value
Batch Size	100
Beam Size	5
Dropout Rate	0.1
Embedding Size	8
Hidden Size	32
LSTM Size	8
LSTM Depth	1
Initial Learning Rate	0.01
Scheduler	Manual
Total Parameters	4504

Dataset	**Accuracy w/o Beam**	**Accuracy w/ Beam**
Training	98%	-
Validation	77%	80%
Test	50%	53%

The proportion of correct translations by length for this experiment's result is provided in the following table:

Table 5.3: Standard Theorems Sequence-Level Accuracy by Length

Length	2	3	4	5	6	7	8	9	10
Training (w/o beam)	-	98%	-	98%	-	98%	-	97%	-
Validation (w/ beam)	-	96%	0%	93%	-	100%	75%	96%	-
Test (w/ beam)	0%	100%	0%	100%	0%	100%	60%	100%	0%

It is evident from these results that the model simply will not generalize well to most expression lengths unseen at training time as lengths 2, 4, 6, and 10 all get a 0% sequence-level accuracy. It is possible that this is partially due to the model overfitting to sequence lengths seen during training time. This might be especially compounded by the fact that each LaTeX input sequence has highly limited diversity, perhaps causing the model to fit to other unseen spurious patterns.

5.2.3 Diverse Theorems Over Even/Odd Expressions Alone with Varying Length

Since the experimental results in 5.2.2 did not exhibit good generalization performance on the majority of expression lengths unseen at training time, the grammar used to generate LaTeX theorems was made more diverse as described in 4.2. Using this new dataset, we retrained the model in an effort to exhibit better generalization performance on unseen lengths. Thus, this dataset was split into training, validation, and testing with the following properties:

Table 5.4: Diverse Theorems Dataset Breakdown by Expression Length

Dataset	**# Examples**	**Expression Lengths (# Terms)**
Training	1000	3, 5, 7, 9
Validation	150	3, 4, 5, 7, 8, 9
Test	150	2, 3, 4, 5, 6, 7, 8, 9, 10

The best results on this dataset were achieved using the following hyperparameters, with the learning rate manually reduced over time:

Table 5.5: Best Experimental Results on Diverse Theorems with Model Hyperparameters

Parameter	Value
Batch Size	100
Beam Size	5
Dropout Rate	0.1
Embedding Size	16
Hidden Size	32
LSTM Size	16
LSTM Depth	1
Initial Learning Rate	0.01
Scheduler	Manual
Epochs Used	997
Total Parameters	10,040

Dataset	**Accuracy w/o Beam**	**Accuracy w/ Beam**
Training	96%	-
Validation	81.33%	81.33%
Test	59.33%	62.67%

The proportion of correct translations by length for this experiment's result is provided in the following table:

Table 5.6: Diverse Theorems Sequence-Level Accuracy by Length

Length	2	3	4	5	6	7	8	9	10
Training (w/o beam)	-	99%	-	99%	-	97%	-	91%	-
Validation (w/o beam)	-	100%	93%	100%	-	100%	26%	81%	-
Test (w/ beam)	0%	100%	88%	100%	68%	100%	54%	92%	0%

As is evident in the above results, by increasing the diversity of the training data in terms of sequence length (without respect to expression length) as well as in the variability of natural language and expression ordering, we significantly improve generalization performance. Now, the model performs much better at test time on unseen intermediate lengths 4, 6, and 8, but still does not generalize well to smaller or longer lengths.

While this exhibits much greater promise, we would like to increase the performance on intermediate lengths further. We hypothesize that training with a slightly deeper model could help, so we use a model with an LSTM depth of 2 meaning another LSTM cell is attached on top of each original LSTM cell in the encoder and decoder. Here we do not evaluate using beam search due to its limited performance boost in previous runs, but we include token level accuracy as an overall comparison metric. The best results on this dataset were achieved using the following hyperparameters, with the learning rate manually reduced over time:

Table 5.7: Best Experimental Results on Diverse Theorems using Deeper Model

Parameter	Value
Batch Size	100
Dropout Rate	0.2
Embedding Size	16
Hidden Size	32
LSTM Size	16
LSTM Depth	2
Initial Learning Rate	0.01
Scheduler	Manual
Epochs Used	1338
Total Parameters	14,392

Dataset	**Token Accuracy**	**Sequence Accuracy**
Training	97%	88%
Validation	86%	82%
Test	66%	49%

The proportion of correct translations by length for this experiment's result is provided in the following table:

Table 5.8: Diverse Theorems Sequence-Level Accuracy by Length using Deeper Model

Length	2	3	4	5	6	7	8	9	10
Training	-	96%	-	94%	-	88%	-	76%	-
Validation	-	96%	86%	97%	-	95%	42%	88%	-
Test	0%	100%	56%	88%	11%	100%	39%	92%	0%

Here we also provide the token-level accuracy broken down by length for comparison:

Table 5.9: Diverse Theorems Token-Level Accuracy by Length using Deeper Model

Length	2	3	4	5	6	7	8	9	10
Training	-	99%	-	99%	-	97%	-	94%	-
Validation	-	96%	90%	97%	-	97%	60%	97%	-
Test	41%	100%	83%	92%	39%	100%	67%	99%	23%

As is evident by these results, it does not appear that increasing the LSTM's depth has a positive impact on generalization performance. This suggests that in order to help the model perform even better on unseen lengths, work may still need to be done to further diversify the LaTeX input sequences or to adjust the implementation of the model itself in some way.

Chapter 6: Summary and Future Directions

Clearly, there is a need to have formalized mathematical proofs such that they can be verified with certainty for their correctness. However, it has been shown how little this has been done due to the task's relative difficulty. We have even seen how few mathematical formulas and proofs have been written in an explicit, machine-readable format. Luckily, the task of converting natural language LaTeX proofs to Coq proofs is known as semantic parsing and neural networks have shown to be very promising on these types of tasks. The semantic parsing problem can be modeled as a sequence-to-sequence task using either recurrent neural networks or transformers. Advances in both architectures have seen great success in recent years on both machine translation tasks as well as semantic parsing tasks. However, in spite of this success and the need for an easier way to formalize mathematics, neural networks have had yet to be applied to the task of converting LaTeX proofs to Coq proofs automatically. It is a brand new, vast, and completely unexplored application area with a great deal of potential, especially when considering recent tangential results on other problems.

In this thesis, we have proposed a model that combines knowledge of semantic parsing with architectures used for neural machine translation to create a model that can successfully convert natural language proofs written in LaTeX from a limited domain into correct Coq proofs. We have adapted and specialized the model using an attention mechanism, copying mechanism, and selective read mechanism to optimize it for this task. We have also proposed new, relatively simple datasets of LaTeX and Coq proof pairs from a limited domain—proofs about the parity of simple mathematical expressions. We generate these datasets using natural language and structure that varies as much as possible by sampling them from a Backus-Naur form grammar written in Rust.

When our model is experimentally applied to our datasets, we immediately achieve strong results in favor of the model's ability to generalize to proofs that use varied natural

language and structure. It is observed that it is more challenging for the model to generalize to expressions of lengths unseen at training time. However, when extra diversity is added to the input data, the model immediately benefits from a strong boost in performance on intermediate lengths. We believe this exhibits a great deal of promise for the ability of the LSTM-based model to learn higher-level translation patterns to not only achieve near 100% accuracy on intermediate lengths, but also to solve tasks that require more advanced forms of generalization.

Going forward, one of the biggest bottlenecks to the modeling and experimentation on this task is in the limited availability of high quality, paired proofs written in both LaTeX and Coq with a strong 1-to-1 correlation. Thus, as a primary future direction, work should be done to further diversify the current datasets as well as design new ones that could exhibit more advanced generalization. In particular, it would be interesting to see if the model could learn two separate proof techniques at training time and be able to combine them to translate proofs requiring both techniques at test time.

Beyond this manual creation, one should look more deeply in the literature to find further datasets with even more diversity. For example, the MATH dataset from [HBK+21] includes 12,500 problems from high school math competitions complete with step by step solutions written in LaTeX which could be a very useful dataset if corresponding, correct Coq proofs could be created. Datasets could even be taken from university courses focused on mathematical proof as well as verification systems like Coq. This could even lead to a use-case where a model like this could automatically grade student proofs by translating them to Coq and checking for correctness.

Finally, and most interestingly, a future direction could include incorporating feedback from the Coq interpreter at each step of the decoder. The Coq interpreter will not only tell the model when it makes an error, but it provides a great wealth of information about the state of the proof and what is needed for the next step in the proof. Incorporating

this feedback during training and testing can only stand to dramatically increase accuracy and generalization at test time.

Ultimately, this project explores a completely novel application of neural network architectures so the current goal is to demonstrate a promising ability for these models to scale by first exhibiting generalization performance within highly limited domains. Based on the results found so far using proofs about the parity of even and odd expressions, we feel hopeful about advancement of an LSTM or transformer-based architecture on tasks like this, but significant work on modeling and data creation is still needed. Nonetheless, this work will be highly rewarding as this is opening the door to a brand-new application area unexplored in the literature—one that stands to have a grand impact on the way we formalize and verify our mathematics in the future.

References

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv:1409.0473 [cs, stat]*, May 2016. arXiv: 1409.0473. URL: http://arxiv.org/abs/1409.0473

[Bie20] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com. URL: https://www.wandb.com/

[CB19] Charles Chen and Razvan Bunescu. Context Dependent Semantic Parsing over Temporally Structured Data. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3576–3585, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/N19-1360, doi:10.18653/v1/N19-1360

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/N19-1423, doi:10.18653/v1/N19-1423

[DL16] Li Dong and Mirella Lapata. Language to Logical Form with Neural Attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43, Berlin, Germany, August 2016. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/P16-1004, doi:10.18653/v1/P16-1004

[DL18] Li Dong and Mirella Lapata. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 731–742, Melbourne, Australia, July 2018. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/P18-1068, doi:10.18653/v1/P18-1068

[GAN+16] Caglar Gulcehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. Pointing the Unknown Words. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 140–149, Berlin, Germany, August 2016. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/P16-1014, doi:10.18653/v1/P16-1014

[GLLL16] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640, Berlin, Germany, August 2016. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/P16-1154, doi:10.18653/v1/P16-1154

[Har08] John Harrison. Formal proof – theory and practice. *Notices of the American Mathematical Society*, 55:1395–1406, 2008.

[HBK+21] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset, 2021. arXiv:2103.03874.

[LPM15] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal, September 2015. Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/D15-1166, doi:10.18653/v1/D15-1166

[SSB14] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition, 2014. arXiv:1402.1128.

[SVDB03] Jürgen Stuber and Mark Van Den Brand. Extracting Mathematical Semantics from LaTeX Documents. In *Workshop on Principles and Practice of Semantic Web Reasoning - PPSWR'2003*, page 15, Mumbai, India, December 2003. Colloque avec actes et comité de lecture. internationale. URL: https://hal.inria.fr/inria-00099469

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *arXiv:1706.03762 [cs]*, December 2017. arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762

Appendix A: Theorem/Proof Grammar for Even/Odd Expressions of Lengths 2/3

Shown here is the complete grammar used to generate LaTeX theorems and proofs about even and odd expressions. For the sake of brevity, we only show the grammar to generate expressions with a length of 2 terms or 3 terms. However, for generating longer expressions, the grammar is extended following the same pattern shown in this one.

⟨*latex*⟩ ::= ⟨*even*⟩
| ⟨*odd*⟩

⟨*odd*⟩ ::= ⟨*odd2*⟩
| ⟨*odd3*⟩

⟨*even*⟩ ::= ⟨*even2*⟩
| ⟨*even3*⟩

⟨*odd2*⟩ ::= `\begin{document}` ⟨*oddTheorem2*⟩ ⟨*oddProof2*⟩
`\end{document}`

⟨*odd3*⟩ ::= `\begin{document}` ⟨*oddTheorem3*⟩ ⟨*oddProof3*⟩
`\end{document}`

⟨*even2*⟩ ::= `\begin{document}` ⟨*evenTheorem2*⟩ ⟨*evenProof2*⟩
`\end{document}`

⟨*even3*⟩ ::= `\begin{document}` ⟨*evenTheorem3*⟩ ⟨*evenProof3*⟩
`\end{document}`

⟨*oddTheorem2*⟩ ::= ⟨*theoremHeader*⟩ ⟨*oddTheoremStatement2*⟩
⟨*theoremFooter*⟩

⟨*oddTheorem3*⟩ ::= ⟨*theoremHeader*⟩ ⟨*oddTheoremStatement3*⟩
⟨*theoremFooter*⟩

⟨*evenTheorem2*⟩ ::= ⟨*theoremHeader*⟩ ⟨*evenTheoremStatement2*⟩
⟨*theoremFooter*⟩

⟨*evenTheorem3*⟩ ::= ⟨*theoremHeader*⟩ ⟨*evenTheoremStatement3*⟩
⟨*theoremFooter*⟩

⟨*theoremHeader*⟩ ::=
`\newtheorem{theorem}{Theorem}[section] \begin{theorem}`

⟨*theoremFooter*⟩ ::= `\end{theorem}`

⟨*oddTheoremStatement2*⟩ ::= ⟨*forEvery*⟩ ⟨*VARLIST1*⟩ , ⟨*expression2*⟩ ⟨*isOdd*⟩ .

⟨*oddTheoremStatement3*⟩ ::= ⟨*forEvery*⟩ ⟨*VARLIST2*⟩ , ⟨*expression3*⟩ ⟨*isOdd*⟩ .

⟨*evenTheoremStatement2*⟩ ::= ⟨*forEvery*⟩ ⟨*VARLIST1*⟩ , ⟨*expression2*⟩ ⟨*isEven*⟩ .

⟨*evenTheoremStatement3*⟩ ::= ⟨*forEvery*⟩ ⟨*VARLIST2*⟩ , ⟨*expression3*⟩ ⟨*isEven*⟩ .

⟨*forEvery*⟩ ::= For every natural number
| For any natural number
| Given any natural number
| Given a natural number

⟨*VARLIST1*⟩ ::= ⟨*VAR1*⟩

⟨*VARLIST2*⟩ ::= ⟨*VAR1*⟩ and ⟨*VAR2*⟩

⟨*expression2*⟩ ::= $⟨*NAT1*⟩ *⟨*VAR1*⟩ + ⟨*NAT2*⟩ $

⟨*expression3*⟩ ::= $⟨*NAT1*⟩ *⟨*VAR1*⟩ + ⟨*NAT2*⟩ *⟨*VAR2*⟩ + ⟨*NAT3*⟩ $

⟨*isOdd*⟩ ::= is odd
| must be odd
| is an odd number

| has to be odd
| has to be an odd number
| must be an odd number ⟨*isEven*⟩ ::= is even
| must be even
| is an even number
| has to be even
| has to be an even number
| must be an even number

⟨*oddProof2*⟩ ::= ⟨*proofHeader*⟩ ⟨*oddFacts2*⟩ ⟨*Because*⟩ ⟨*oddReasoning2*⟩ , ⟨*lowerOddRestatement2*⟩ .⟨*proofFooter*⟩
| ⟨*proofHeader*⟩ ⟨*oddFacts2*⟩ ⟨*upperOddRestatement2*⟩ , ⟨*because*⟩ ⟨*oddReasoning2*⟩ .⟨*proofFooter*⟩

⟨*oddProof3*⟩ ::= ⟨*proofHeader*⟩ ⟨*oddFacts3*⟩ ⟨*Because*⟩ ⟨*oddReasoning3*⟩ , ⟨*lowerOddRestatement3*⟩ .⟨*proofFooter*⟩
| ⟨*proofHeader*⟩ ⟨*oddFacts3*⟩ ⟨*upperOddRestatement3*⟩ , ⟨*because*⟩ ⟨*oddReasoning3*⟩ .⟨*proofFooter*⟩

⟨*evenProof2*⟩ ::= ⟨*proofHeader*⟩ ⟨*evenFacts2*⟩ ⟨*Because*⟩ ⟨*evenReasoning2*⟩ , ⟨*lowerEvenRestatement2*⟩ .⟨*proofFooter*⟩
| ⟨*proofHeader*⟩ ⟨*evenFacts2*⟩ ⟨*upperEvenRestatement2*⟩ , ⟨*because*⟩ ⟨*evenReasoning2*⟩ .⟨*proofFooter*⟩

⟨*evenProof3*⟩ ::= ⟨*proofHeader*⟩ ⟨*evenFacts3*⟩ ⟨*Because*⟩ ⟨*evenReasoning3*⟩ , ⟨*lowerEvenRestatement3*⟩ .⟨*proofFooter*⟩
| ⟨*proofHeader*⟩ ⟨*evenFacts3*⟩ ⟨*upperEvenRestatement3*⟩ , ⟨*because*⟩ ⟨*evenReasoning3*⟩ .⟨*proofFooter*⟩

⟨*proofHeader*⟩ ::= `\begin{proof}` ⟨*proofFooter*⟩ ::= `\end{proof}`

⟨*oddFacts2*⟩ ::= ⟨*upperDeclarativePhrase*⟩ ⟨*NAT1*⟩ ⟨*isEven*⟩ and ⟨*lowerDeclarativePhrase?*⟩ ⟨*NAT2*⟩ ⟨*isOdd*⟩ . ⟨*Thus*⟩ ⟨*NAT1*⟩ *⟨*VAR1*⟩ ⟨*isEven*⟩ .
| ⟨*upperDeclarativePhrase*⟩ ⟨*NAT1*⟩ ⟨*isEven*⟩ ⟨*thus*⟩ ⟨*NAT1*⟩ *⟨*VAR1*⟩ ⟨*isEven*⟩ . ⟨*upperDeclarativePhrase*⟩ ⟨*NAT2*⟩ ⟨*isOdd*⟩ .

⟨*oddFacts3*⟩ ::= ⟨*upperDeclarativePhrase*⟩ ⟨*NAT1*⟩ ⟨*isEven*⟩ ⟨*thus*⟩ ⟨*NAT1*⟩ *⟨*VAR1*⟩ ⟨*isEven*⟩ and ⟨*lowerDeclarativePhrase?*⟩ ⟨*NAT2*⟩ ⟨*isEven*⟩ ⟨*thus*⟩ ⟨*NAT2*⟩ *⟨*VAR2*⟩ ⟨*isEven*⟩ . ⟨*upperDeclarativePhrase*⟩ ⟨*NAT3*⟩ ⟨*isOdd*⟩ .

⟨*evenFacts2*⟩ ::= ⟨*upperDeclarativePhrase*⟩ ⟨*NAT1*⟩ ⟨*isEven*⟩ and ⟨*lowerDeclarativePhrase?*⟩ ⟨*NAT2*⟩ ⟨*isEven*⟩ ⟨*also?*⟩ . ⟨*Thus*⟩ ⟨*NAT1*⟩ *⟨*VAR1*⟩ ⟨*isEven*⟩ .
| ⟨*upperDeclarativePhrase*⟩ ⟨*NAT1*⟩ ⟨*isEven*⟩ ⟨*thus*⟩ ⟨*NAT1*⟩ *⟨*VAR1*⟩ ⟨*isEven*⟩ . ⟨*upperDeclarativePhrase*⟩ ⟨*NAT2*⟩ ⟨*isEven*⟩ ⟨*also?*⟩ .

⟨*evenFacts3*⟩ ::= ⟨*upperDeclarativePhrase*⟩ ⟨*NAT1*⟩ ⟨*isEven*⟩ ⟨*thus*⟩ ⟨*NAT1*⟩ *⟨*VAR1*⟩ ⟨*isEven*⟩ and ⟨*lowerDeclarativePhrase?*⟩ ⟨*NAT2*⟩ ⟨*isEven*⟩ ⟨*thus*⟩ ⟨*NAT2*⟩ *⟨*VAR2*⟩ ⟨*isEven*⟩ . ⟨*upperDeclarativePhrase*⟩ ⟨*NAT3*⟩ ⟨*isEven*⟩ ⟨*also?*⟩ .

⟨*upperDeclarativePhrase*⟩ ::= ⟨*upperStatement*⟩

⟨*lowerDeclarativePhrase?*⟩ ::= ⟨*lowerDeclarativePhrase*⟩
|

⟨*lowerDeclarativePhrase*⟩ ::= ⟨*lowerStatement*⟩

⟨*upperStatement*⟩ ::= ⟨*upperAdverb*⟩ ⟨*lowerDeclaration*⟩
| ⟨*upperDeclaration*⟩

⟨*lowerStatement*⟩ ::= ⟨*lowerAdverb*⟩ ⟨*lowerDeclaration*⟩
| ⟨*lowerDeclaration*⟩

⟨*upperDeclaration*⟩ ::= It is known⟨*that?*⟩
| We know⟨*that?*⟩
| We have that
| We can see⟨*that?*⟩
| It has been proven⟨*that?*⟩
| By definition

	| You can show⟨*that?*⟩
	| It can be shown that
	| It can be proven that

⟨*lowerDeclaration*⟩	::=	it is known⟨*that?*⟩
	|	we know⟨*that?*⟩
	|	we have that
	|	we can see⟨*that?*⟩
	|	it has been proven⟨*that?*⟩
	|	by definition
	|	you can show⟨*that?*⟩
	|	it can be shown that
	|	it can be proven that

⟨*oddReasoning2*⟩	::=	⟨*adding*⟩ an even number ⟨*to an*⟩ odd number ⟨*isAlways*⟩ an odd number
	|	⟨*adding*⟩ an odd number ⟨*to an*⟩ even number ⟨*isAlways*⟩ an odd number

⟨*oddReasoning3*⟩	::=	⟨*adding*⟩ two even numbers ⟨*to an*⟩ odd number ⟨*isAlways*⟩ an odd number
	|	⟨*adding*⟩ an odd number to two even numbers ⟨*isAlways*⟩ an odd number
	|	since the sum of two even numbers ⟨*isAlways*⟩ even and ⟨*adding*⟩ an even number ⟨*to an*⟩ odd number ⟨*isAlways*⟩ an odd number
	|	since the sum of two even numbers ⟨*isAlways*⟩ even and ⟨*adding*⟩ an odd number ⟨*to an*⟩ even number ⟨*isAlways*⟩ an odd number

⟨*evenReasoning2*⟩	::=	⟨*adding*⟩ an even number ⟨*to an*⟩ even number ⟨*isAlways*⟩ an even number
	|	⟨*adding*⟩ two even numbers together ⟨*isAlways*⟩ an even number

⟨*evenReasoning3*⟩	::=	⟨*adding*⟩ three even numbers together ⟨*isAlways*⟩ an even number

⟨*upperOddRestatement2*⟩ ::= ⟨*expression2*⟩ ⟨*lowerAdverb*⟩ ⟨*isOdd*⟩
| ⟨*upperAdverb*⟩ , ⟨*expression2*⟩ ⟨*isOdd*⟩

⟨*upperOddRestatement3*⟩ ::= ⟨*expression3*⟩ ⟨*lowerAdverb*⟩ ⟨*isOdd*⟩
| ⟨*upperAdverb*⟩ , ⟨*expression3*⟩ ⟨*isOdd*⟩

⟨*lowerOddRestatement*⟩ ::= ⟨*expression*⟩ ⟨*lowerAdverb*⟩ ⟨*isOdd*⟩
| ⟨*lowerAdverb*⟩ , ⟨*expression*⟩ ⟨*isOdd*⟩

⟨*lowerOddRestatement2*⟩ ::= ⟨*expression2*⟩ ⟨*lowerAdverb*⟩ ⟨*isOdd*⟩
| ⟨*lowerAdverb*⟩ , ⟨*expression2*⟩ ⟨*isOdd*⟩

⟨*upperEvenRestatement2*⟩ ::= ⟨*expression2*⟩ ⟨*lowerAdverb*⟩ ⟨*isEven*⟩
| ⟨*upperAdverb*⟩ , ⟨*expression2*⟩ ⟨*isEven*⟩

⟨*upperEvenRestatement3*⟩ ::= ⟨*expression3*⟩ ⟨*lowerAdverb*⟩ ⟨*isEven*⟩
| ⟨*upperAdverb*⟩ , ⟨*expression3*⟩ ⟨*isEven*⟩

⟨*lowerEvenRestatement2*⟩ ::= ⟨*expression2*⟩ ⟨*lowerAdverb*⟩ ⟨*isEven*⟩
| ⟨*lowerAdverb*⟩ , ⟨*expression2*⟩ ⟨*isEven*⟩

⟨*lowerEvenRestatement3*⟩ ::= ⟨*expression3*⟩ ⟨*lowerAdverb*⟩ ⟨*isEven*⟩
| ⟨*lowerAdverb*⟩ , ⟨*expression3*⟩ ⟨*isEven*⟩

⟨*upperAdverb?*⟩ ::= ⟨*upperAdverb*⟩
|

⟨*upperAdverb*⟩ ::= Clearly
| Obviously
| Of course
| Certainly
| Definitely

⟨*lowerAdverb?*⟩ ::= ⟨*lowerAdverb*⟩
|

⟨*lowerAdverb*⟩ ::= clearly
| obviously
| of course
| certainly
| definitely

⟨*that?*⟩ ::= that
|

⟨*also?*⟩ ::= ⟨*also*⟩
|

⟨*also*⟩ ::= also
| too
| as well

⟨*thus*⟩ ::= thus
| therefore
| so
| hence

⟨*Thus*⟩ ::= Thus
| Therefore
| So
| Hence

⟨*Because*⟩ ::= Because
| Since
| Due to the fact that

⟨*because*⟩ ::= because
| since
| due to the fact that

⟨*isAlways*⟩ ::= is always
| must always be
| has to be
| is

	| results in
	| always results in
	| always is
⟨*adding*⟩	::= adding
	| summing
⟨*to an*⟩	::= to an
	| with an