2019

Automatic repair and type binding of undeclared variables using neural networks

Venkatesh Theru Mohan
Iowa State University

Automatic repair and type binding of undeclared variables using neural networks

by

Venkatesh Theru Mohan

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Ali Jannesari, Major Professor
Chinmay Hegde
Jin Tian

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

DEDICATION

I would like to dedicate this thesis to my parents Mr.TA Mohan, Mrs.P.Mohanalakshmi, sister Mrs.TM Priya and brother-in-law, Mr.M Harish without whose prayers, support and encouragement I would not have been able to complete this work.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, I would like to thank my Major Professor Ali Jannesari for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members for their efforts and contributions to this work: Professor Chinmay Hegde and Professor Jin Tian.

ABSTRACT

Over the past few years, there had been significant achievements in the deployment of deep learning for analysing the programs due to the brilliance of encoding the programs by building vector representations. Deep learning had been used in program analysis for detection of security vulnerabilities using generative adversarial networks, prediction of hidden software defects using software defect datasets. Furthermore, they had also been used for detecting as well as fixing syntax errors that are made by novice programmers by learning a trained neural machine translation on bug-free programming source codes to suggest possible fixes by finding the location of the tokens that are not in place. However, all these approaches either require defect datasets or bug-free code samples that are executable for training the deep learning model. Our neural network model is neither trained with any defect datasets nor bug-free code samples, instead it is trained using structural semantic details of ASTs where each node represents a construct appearing in the source code. This model is implemented to fix one of the most common syntax errors, such as undeclared variable errors as well as infer their type information before program compilation. By this approach, the model has achieved in correctly locating and identifying 81% of the programs on prutor dataset of 1059 programs with undeclared variable errors and also inferring their data types correctly in 80% of the programs.

CHAPTER 1. INTRODUCTION

Deep learning has been influential in automating tasks in many domains such as robotics, speech generation, image/video processing, big data, natural language processing, data mining etc over the past few decades. Furthermore, recently there had been some significant contributions in the automation of program analysis tasks as well. Neural networks had been frequently used either for detection or generation tasks in programming language processing similar to natural language processing. It had been employed in detecting software defects as well as prediction of errors by using software metrics Jayanthi and Florence (2018).

Learning features in NLP is a relatively simpler task due to the fact that a vocabulary can be easily formed from the words of the regular language. Computers deployed with deep learning algorithms can understand only numbers, so features learned in NLP must be transformed into vectors of numbers in order to carry out training, testing and generation tasks. Therefore, learning in programming language processing takes place by building program vector representations using the features of the source code for program classification, using abstract syntax tree/network representations to capture structural syntactic and semantic dependencies of the non-terminal and terminal nodes in order to learn a vectored representation of source code for programming language processing tasks, representing the programs as graph networks and learning features of nodes by automating prediction tasks over nodes and edges in networks. These different kinds of feature processing techniques of the programming source codes has led to many significant breakthroughs in blending deep learning algorithms with many complex areas of programming language analysis such as syntax,semantic and run-time error detection approaches, suggestion of error fixes, automating

the task of fixing common syntax or semantic errors that are made usually by the beginners with very little background knowledge in the field of programming.

Adding to that, there have also been some state-of-art performances achieved by neural machine translation systems on language translation tasks in NLP that led to the deployment of these systems on error correction tasks in the programming source codes. These kinds of systems in Ahmed et al. (2018) builds and trains a single neural network model using a labelled set of paired examples that results in translation from the input directly. They are end-to-end in the sense that they have the ability to learn the source text directly by mapping them to the corresponding target text and uses an encoder-decoder(usually made up of RNN units) approach for applying the transformations where encoder consists of sequences of source text and decoder consists of sequences of target text. Generative Adversarial Networks (GAN) trains a neural network model to predict security vulnerabilities in Harer et al. (2018) without the necessity of being a paired set of source domain containing buggy codes and target domain consisiting of bug-free codes, instead being a bijection mapping. Generally, in this labelled set of paired or unpaired examples, the neural network model is trained on the set of positive examples where the mapping takes place between source sequences, made up of positive examples that are bug-free and compiling without any errors and target sequences consisting of negative examples that contains bugs within the code making the compilation to fail.

In sequence-to-sequence learning systems such as NMT, it is flexible to train and learn the model with a labelled set of paired examples, but consider the scenario where there are no paired examples, further to put it in a simpler context, consider a real-case scenario where some common syntax or semantic errors are being committed by freshers or novice programmers working in a software industry or student submissions of programming assignments in

coding competitions and there are no positive or bug-free reference examples, then learning becomes difficult for neural network approaches and neural machine translation models.

In our model, there are no positive examples to train and focus is only on the negative buggy examples specifically, the common semantic error caused due to undeclared variables which is often committed and unperceived by the novice programmers. The model is instead trained based on the structural and semantic elements, that is the non-terminal and terminal nodes of the programming source codes captured from the abstract syntax tree representation. The type of the undeclared variables is also determined by performing type conversions statically using the semantic elements of AST representation that provides about the type information of the variables thereby saving the time in inferring about the types of those undeclared variables in order to be compiled and executed successfully.

The comprehensive information of the ASTs, the motive of RNN, detailed view of the training approach and implementation, the generation approach and different scenarios where undeclared variables will be caused and possible cases of type inference of them will be discussed in the upcoming sections of the paper.

CHAPTER 2. RELATED WORK

Abstract syntax trees are the static intermediate model of a programming source code as discussed in G.Fischer et al. (2007) where the compiler's analytic front-end parses it, constructs the AST model eventually passing it to the compiler's synthetic back-end to produce an assembly code for a specific machine and also used for program analysis/transformation. A well-organized programming source code can be represented as abstract syntax tree for providing an insight analysis, simulation similar to an informal specification which can be seen from Wile (1997) where a code pattern is identified by a semantic chunk of code such that the instances are tagged positive if source code contains the corresponding pattern, else, it is tagged negative thereby providing a syntactic definition. AST transformations are applied in Carlos Araya () where they are augmented to prevent time consumption and save the code maintainer's time on discovery of the variables or location of definitions that are delocalized by developing an ontology for code-level knowledge based on annotating it with some semantic information depending on the questions recorded. Often low-level concrete syntax tree representations have a complex structure and it is difficult to characterize the semantics especially in the poorly understood domains, so in Welty (1997), describes a transformation process to get a good abstract syntax representation from low-level concrete specification where modern tools rely on its ability to analyze, simulate and synthesize programs easily in language processing. The importance of AST representation in understanding the structure of the source code and dependencies between the semantic elements of it can be seen from F. Pfenning (1988) where higher order abstract syntax acts as a central representation for programs, rules, syntactic objects in program manipulations and formal systems such as Common Lisp where there are matching, substitution and unification operations.

In the past recent decades, deep learning had achieved various scales in text domain such as natural language especially in neural machine translation tasks and used it successfully even for programming language processing tasks as well. Some of the recent works that had achieved empiricial success on neural machine translation include dialogue response generation, summarization and text generation tasks as explained in detail in Kosovan et al. (2017), also NMT tasks are very much useful in translation from one language to another like in Choudhary et al. (2018) where NMT encoder-decoder model had been implemented using word embedding along with byte-pair encoding method to develop an efficent translation mechanism from English-Tamil. Neural template generation had been implemented in Wiseman et al. (2018) using hidden semi-markov model decoder to generate more interpretable and controllable translation. The performance of such text generation tasks such as NMT is enhanced with attention mechanism in Bahdanau et al. (2014) to automatically search for a context of a source text that is relevant for prediction of target words and an ensemble model of global and local attention mechanism of Luong et al. (2015) improving the performance.

Natural language generation having a discrete output space had also been implemented by generative adversarial networks (introduced by Goodfellow et al. (2014)) on generating sentences from context-free grammars and probabilistic grammars as shown in Rajeswar et al. (2017). The quality of the text generation had been improvised by conditioning information on generated samples to produce realistic natural looking sentences compared to maximum likelihood training procedures. Text generation has also shown some encouraging results in Guo et al. (2018) in which the discriminator of the GAN leaks its own high-level features to the generator at each of the generator steps thus making the scalar guiding signal available continuously throughout the generative process. Also, reinforcement learning has

been used NLP tasks in Yuan et al. (2017) for generation of natural language questions from documents based on answers recorded in them using policy gradient estimation techniques.

In Liu et al. (2016), prediction of next tokens in the source code is implemented using LSTM neural networks where the model trains and learns associated subsequent nodes for code completion, given a partial AST containing left subset of nodes or semantic features with respect to a subtree. A coding criterion is used in Peng et al. (2015) to build vector representations of the source code in order to classify the programs based on the relationship and dependencies between the nodes of the AST using deep neural networks and is evaluated by k-means clustering. Semantic parsing and code generation is also implemented by mapping structured inputs to executable outputs in Rabinovich et al. (2017) by modelling the outputs obtained from decoder using abstract syntax networks and constructing the AST from the modular structure dynamically determined from the output tree. The efficiency of AST in extracting tokens and comparing the source codes based on them and the use of deep learning in classifying duplicate/clone codes can be helpful in software code maintenance as seen in Li et al. (2017b) where maintaining duplicate codes for reuse in order to improve productivity of programming becomes a burden when there are inconsistencies caused due to bug fixes and program modifications in the original code at multiple locations.

The significance of AST representation of source code can further be noted in Dam et al. (2017) where LSTM neural networks are leveraged to capture the long contextual relationships between semantic features to identify the related code elements in order to predict the software vulnerabilities which causes a security threat or makes the program buggy. Conventional software defect prediciton approaches that is used to train the ML classifiers using hard-coded features is combined along with a method Li et al. (2017a) to extract those features using AST semantic and syntactic elements by encoding them as vector representa-

tions and training them using CNN for prediction of the defects. Source code's semantics can also be extracted using control flow graph representations as in A. Viet Phan and Bui (2017) where these are constructed from assembly language after compiling the source code to automatically learn defect features dynamically using multi-view multi-layer directed graph CNN.

GAN approach is used in Harer et al. (2018) for repairing vulnerabilities in source codes without any paired examples or bijections by mapping from non-buggy source domain to buggy target domain and training the discriminator using the loss that is generated between real examples and NMT-generated outputs from generator of the desired output. Syntax errors poses a threat as it fails the compilation and some recent techniques such as Ahmed et al. (2018) where a RNN model is learnt on syntactically correct, executing student programming course submissions to model all valid sequences of token and use a prefix token sequence which is from the beginning of the program till the error location and is used to predict the following sequence that are able to automate the fixing of the errors present in corresponding locations of the code. Sequence-to-sequence NMT with attention mechanism is learned iteratively in repairing syntax errors in Gupta et al. (2017) using the tokenized vector representation of the program and is used to predict the erroneous program locations and the fixing statement without using any external compiler tools or any AST representation. A real-time feedback is given to the students enrolled in beginner level programming assignments in Bhatia and Singh (2016) of the compile-time syntax errors that are made by using RNN to predict the target lines from syntactically correct submissions given the source error lines from wrong submissions and a abstract version of top ranked suggestion of error fix is presented as a feedback.

CHAPTER 3. SYNTAX AND SEMANTIC ANALYSIS: ABSTRACT SYNTAX TREE

3.1 Phases of Compiler

The compiler is a program that performs static analysis or, in more general terms, analyses/reports error during compile-time rather than run-time, by translating a program from a high-level source language to low-level assembly language. Usually, the stages of the compiler are divided into analysis and synthesis parts where, in the analysis stage(also called the front-end of the compiler), the high-level source program is divided into a set of components and a grammatical or syntactical structure is constructed from it. There is also a symbol table, a data structure that stores all the information collected from the analysis stage. An intermediate structure is constructed from the grammatical structure of the source program. In the back-end stage of the compiler, that is, in synthesis stage, the intermediate structure created in the previous step and the symbol table together is used to construct the executable target machine-level program.

The analysis stage is further split into lexical, syntax and semantic analysis. During the lexical analysis, the lexical analyser groups the characters into meaningful sequences called lexemes defined by a set of production rules/patterns of the regular grammar, formed from stream of characters of the source program, where a stream of tokens is produced as the output of this phase along with its corresponding attribute-values that are associated with entries in the symbol table. In the syntax analysis or parsing phase, the parser builds an intermediate tree structure(also called the syntax tree) by matching the tokens generated

by lexical analyser against the production rules of context-free grammar due to its ability in checking for balanced tokens. In the next phase of the analysis stage, the semantic analyser uses the syntax tree, the symbol table and also musters type information that is obtained from attribute grammar formed from the context-free grammar due to the addition of type attributes either inherited or synthesized from the terminals to its non-terminals. Therefore, the output of the semantic analysis phase results in an annotated syntax tree. During the syntax analysis phase, syntax trees that are formed can either be concrete or abstract.

3.2 Concrete Syntax Trees

The underlying formalism that exists in concrete syntax trees follows the production rules of context-free grammar in order for the input to be recognized by the parser. The root of the parse trees usually begins with the start symbol and is used to form a derivation tree by deriving a certain string/statement in the programming language. The interior nodes represents the non-terminals which are named patterns and appear on the left-side of the production rules. The leaf nodes of the parse tree are the terminals of the context-free language that represents the identifiers,variable names and the literals of the statements in the programming language. Abstract syntax trees is efficient in representing the program constructs as it has a simplified parsed structure compared to one-to-one mapping of context-free grammar language to tree-form in concrete syntax trees.

3.3 Abstract Syntax Trees

The Abstract Syntax Tree is the tree representation of abstract syntactical structure of a programming language construct where the operators forms the root and the interior nodes and the operands of those corresponding operators forms the children nodes. They do not use interior nodes to represent a grammar production rule. It is the output obtained from

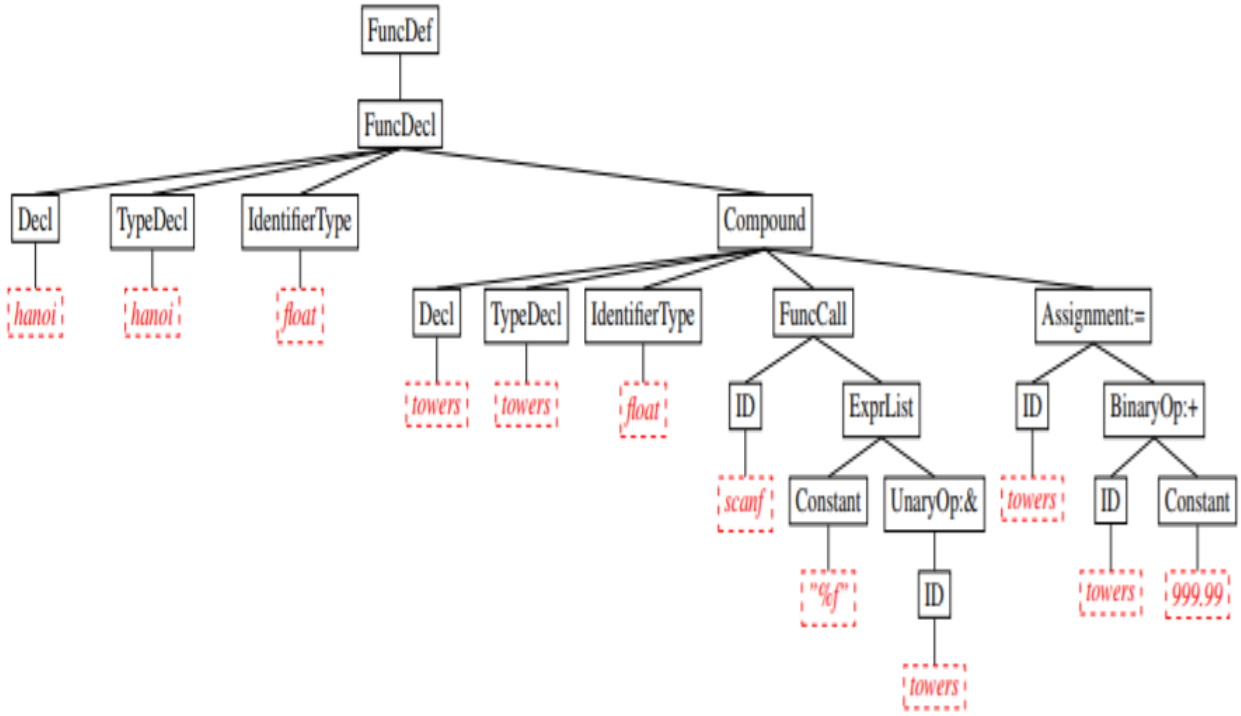


Figure 3.1: An example illustration of Abstract Syntax Trees (AST) of a C program with the non-terminals at the root as well as internal nodes represented by black-colored enclosed rectangle box and terminals at the leaf nodes of the rooted tree depicted by red-colored dashed box.

syntactic/semantic analysis phase of the compiler and follows an abstract grammar. The syntax of the abstract grammar is defined by a set of non-terminal symbols, set of terminal symbols and a set of syntactic domain symbols such as addition, multiplication, subtraction, types, identifiers etc. The syntactic domain symbols are nothing but the named symbols of production rules, non-terminals are on the left-hand side and the terminals on the right-hand side of these production rules. The underlying formalism that exists in abstract syntax trees are recursive data structures.

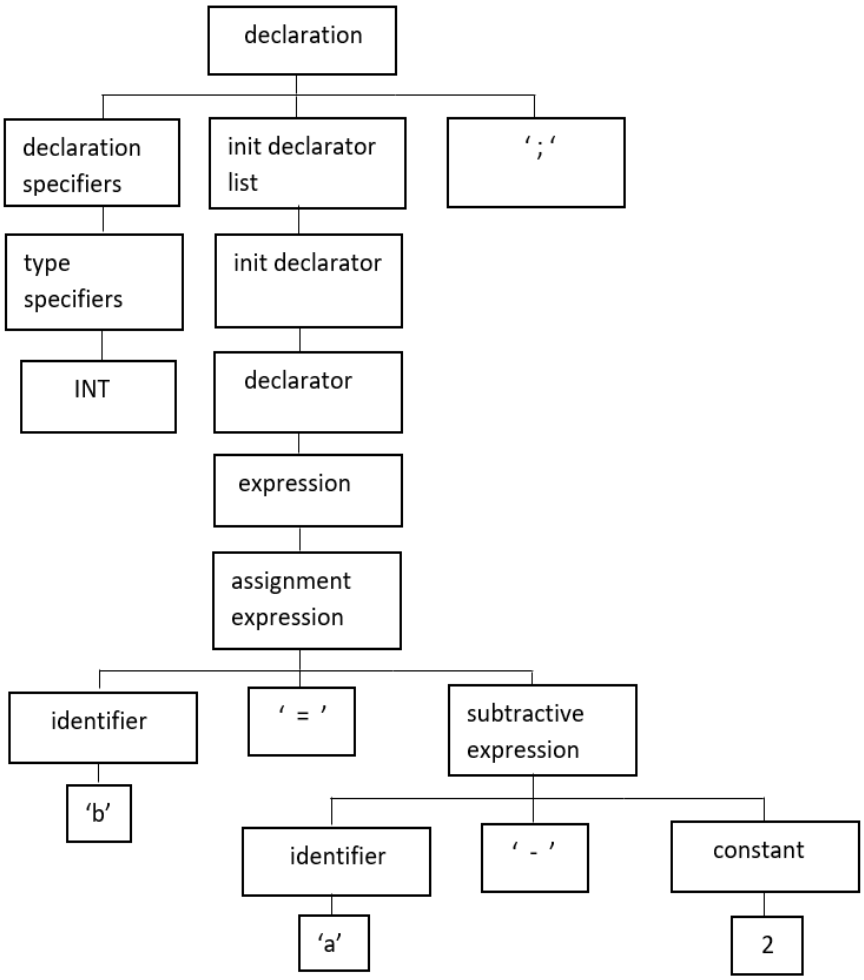


Figure 3.2: An example illustration of Concrete Syntax Trees (CST) of C program of the expression statement int b = a - 2.

CHAPTER 4. LONG SHORT-TERM MEMORY RECURRENT NEURAL NETWORKS

4.1 Overview

This section covers a brief overview about the recurrent neural networks and also its advantages over feedforward artifical neural networks(ANN) such as multilayer perceptrons(MLP) and convolutional neural networks(CNN) in sequence prediction or generation applications thereby dealing with the purpose of its implementation in our paper.

First and foremost, let us talk briefly about the MLP. It is a class of feedforward ANN consisting of atleast three layers of nodes, an input layer, a hidden layer and an output layer. There can be multiple hidden layers stacked between input and output layers where the nodes in the hidden layers as well as the output layer uses non-linear activation functions. The input nodes and the hidden nodes are associated with weights for training and MLP is enhanced with a supervised learning through backpropagation technique where learning occurs in the perceptron by adjusting weights based on the error in the output at each step thereby minimizing the error. MLP are used in sequence prediction problems but they do not have feedback loops from output layers to input and hidden layers. Nevertheless, although feedback is provided with backpropagation in which errors in the output nodes are minimized, there is neither any contextual information about the previous states of the output node nor the states of the previous nodes in sequence thereby making them not a good choice in sequence generation applications.

Now, let us discuss about CNN and its implications. CNN is also a class of feedforward ANN containing input, convolution, pooling, fully connected layers, activation units such as RELU or softmax. The convolution layer applies convolutional or striding convolutional operations using filters that slides over different locations of the input matrix layer and transforming to different feature maps which selects high level features from these local feature maps either by max, average or sum pooling and then the fully connected layers forms the end of the CNN capturing global features where it connects all the output to input nodes completely in such a way that it looks at the output of the previous layers that represent activation maps of high level features and determines the specific set of features that correlates to the output class by using softmax or RELU unit activation functions. Based on the working of CNN, it excels in applications such as feature extractions or image processing because of strong local correlation of its features, its hierarchial structure related data(in pixels) and the features that are invariant to rotations/translations. Although it is used in text/sequence prediction problems, it does not excel because of the above mentioned characteristics. This clearly explains the fact that RNN is the most suitable artificial neural network that can be used for sequence prediction and generation applications which will be explained in detail in the upcoming sections.

4.2 Recurrent Neural Networks

Recurrent Neural Networks(a.k.a vanilla RNN) form a class of feedforward as well as feedback artificial neural networks augmented by inclusion of edges connecting nodes to each other as well as self-connections in a temporal sequence of a directed graph thereby exhibiting dynamic temporal characteristics. Each unit of RNN shares same weights across several timesteps and maintains a hidden state which is basically a stored memory capturing long-range time dependency information eventually making them the best deep learning

model for text/sequence generation and machine translation applications. Each individual units of RNN depends on the output states of previous inputs and its own state. RNN mechanism works in such a way that parameters are shared across the units where each unit of the output is a function of previous units and same update rule is maintained for producing all the output units. RNN operates on an input sequence $x^{(1)}$,$x^{(2)}$... $x^{(T)}$ where each data point $x^{(t)}$ is a real-valued vector with the timesteps being in the range from 1 to T , the hidden states at each timestep from 1 to T are represented as $h^{(1)}$,$h^{(2)}$... $h^{(t-1)}$,$h^{(t)}$, the outputs are represented as $o^{(1)}$,$o^{(2)}$... $o^{(t-1)}$,$o^{(t)}$, the targets are represented by $y^{(1)}$,$y^{(2)}$... $y^{(t-1)}$,$y^{(t)}$

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \tag{4.1}$$

$$a^{(t)} = W * h^{(t-1)} + U * x^{(t)} \tag{4.2}$$

$$h^{(t)} = tanh(a^{(t)}) \tag{4.3}$$

$$o^{(t)} = c + V * h^{(t)} \tag{4.4}$$

$$\acute{y}^{(t)} = softmax(o^{(t)}) \tag{4.5}$$

where b and c are bias vectors along the weight matrices U,V and W corresponding to connections between input-to-hidden states,hidden-to-output and hidden-to-hidden state connections respectively. RNN can also be represented as unfolded or unrolled computational graph(resulting in shared parameters across RNN structure) after t steps with a function $g^{(t)}$

$$h^{(t)} = g^{(t)}(x^{(t)}, x^{(t-1)}, x^{(t-2)}, ..., x^{(2)}, x^{(1)}) \tag{4.6}$$

The function $g^{(t)}$ takes all the past sequences ($x^{(t)}$,$x^{(t-1)}$,$x^{(t-2)}$,...,$x^{(2)}$,$x^{(1)}$) and produces the current state whereas the unrolled recurrent structure factorizes $g^{(t)}$ into a repetition of function f. The unfolding helps in using a constant transition function with same parameters at each timesteps and also the input size of the model is the same due to the above mentioned reason.

4.3 Long Short Term Memory RNN

This section covers the basics of LSTM-RNN and the prediction model that is subsequently used for generation approach.

Long Short Term Memory(LSTM) are recurrent neural networks that have special memory units in the form of self-loops to produce paths so that information can be maintained for longer durations of time. LSTM is preferred over vanilla RNN due to the fact that the former tends to avoid vanishing or exploding gradient problem that occurs when trying to learn long term dependencies and store it in memory cells during backpropagation. This occurs when many deep layers with specific activation functions like sigmoid are used for training, it smoothens a region of input space into an output space between 0 and 1, then even a high change in input region effects almost negligible change in output region, thereby making the gradients/error signals of a long-term interaction becomes vanishingly very small. Further, the vanilla RNNs are affected by information morphology problem in which information contained at a prior state is lost due to non-linearites between the input and output space. LSTM avoids this problem by ensuring a constant unit activation function and uses gates to control the information flow between the memory cell and the outside layers without

any inference. LSTM uses three gates namely forget gate, input gate and output gate layers. The forget gate layer decides the information that is needed to be stored or erased from the LSTM cell state where the decision is made by the sigmoid layer outputting a number between 0 to 1.

$$f_{\mathrm{i}}{}^{(\mathrm{t})} = \sigma(b_{\mathrm{i}}{}^{\mathrm{f}} + \Sigma U_{\mathrm{i,j}}{}^{\mathrm{f}} x_{\mathrm{j}}{}^{(\mathrm{t})} + \Sigma W_{\mathrm{i,j}}{}^{\mathrm{f}} h_{\mathrm{j}}{}^{(\mathrm{t\text{-}1})}) \tag{4.7}$$

where $\mathrm{x}^{(\mathrm{t})}$ is the input vector at current timestep t and $\mathrm{h}^{(\mathrm{t})}$ is the current hidden layer vector at timestep t, and b^{f}, U^{f}, W^{f} are bias units, input weights and recurrent weights of forget gate units $\mathrm{f_i}^{(\mathrm{t})}$. The input gate layer is provided in order to control the flow of new information that is being stored in the cell state

$$g_{\mathrm{i}}{}^{(\mathrm{t})} = \sigma(b_{\mathrm{i}}{}^{\mathrm{g}} + \Sigma U_{\mathrm{i,j}}{}^{\mathrm{g}} x_{\mathrm{j}}{}^{(\mathrm{t})} + \Sigma W_{\mathrm{i,j}}{}^{\mathrm{g}} h_{\mathrm{j}}{}^{(\mathrm{t\text{-}1})}) \tag{4.8}$$

where $\mathrm{x}^{(\mathrm{t})}$ is the input vector at current timestep t and $\mathrm{h}^{(\mathrm{t})}$ is the current hidden layer vector at timestep t, and b^{g}, U^{g}, W^{g} are bias units, input weights and recurrent weights of input gate units $\mathrm{g_i}^{(\mathrm{t})}$.

$$s_{\mathrm{i}}{}^{(\mathrm{t})} = f_{\mathrm{i}}{}^{(\mathrm{t})} s_{\mathrm{i}}{}^{(\mathrm{t\text{-}1})} + g_{\mathrm{i}}{}^{(\mathrm{t})} \sigma(b_{\mathrm{i}} + \Sigma U_{\mathrm{i,j}} x_{\mathrm{j}}{}^{(\mathrm{t})} + \Sigma W_{\mathrm{i,j}} h_{\mathrm{j}}{}^{(\mathrm{t\text{-}1})}) \tag{4.9}$$

where the parameters W,U and b represents the recurrent weights, input weights and bias units present in a LSTM cell. The output gate layer in the memory cell decides the pieces of information that is going to be output by the LSTM cell state. This is done by passing the cell state through a tanh layer and eventually multiplying by the sigmoid of the output gate.

$$q_{\mathrm{i}}{}^{(\mathrm{t})} = \sigma(b_{\mathrm{i}}{}^{\mathrm{o}} + \Sigma U_{\mathrm{i,j}}{}^{\mathrm{o}} x_{\mathrm{j}}{}^{(\mathrm{t})} + \Sigma W_{\mathrm{i,j}}{}^{\mathrm{o}} h_{\mathrm{j}}{}^{(\mathrm{t\text{-}1})}) \tag{4.10}$$

where b^{o},U^{o},W^{o} are the parametric units of the output gate $\mathrm{q_i}^{(\mathrm{t})}$ that represents bias units, input weights and recurrent weights. The output hidden state $\mathrm{h_i}^{(\mathrm{t})}$ is obtained from output gate $\mathrm{q_i}^{(\mathrm{t})}$ as follows:

$$h_{\mathrm{i}}{}^{(\mathrm{t})} = tanh(s_{\mathrm{i}}{}^{(\mathrm{t})}) q_{\mathrm{i}}{}^{(\mathrm{t})} \tag{4.11}$$

CHAPTER 5. IMPLEMENTATION APPROACH AND MODEL

5.1 Motivating Examples

This section first covers some examples of undeclared variables and also the importance of semantic analysis to determine the type information of those undeclared variables. We introduce the Abstract Syntax Trees (AST) that is used as the input, the way in which we deploy the LSTM RNN for training the deep learning model, semantic analysis determining the types of undeclared variables, and the generation approach by performing the serialization of the AST in order to get back the intended clean and buggy-free source code.

The most frequent semantic error that goes unnoticeable by novice programmers is the undeclared variables. The cause of this error is due to the variables being undeclared or another common cause will be usually through spelling mistakes which makes it the first occurrence in the program. Figure 1 shows an example of an undeclared variable error thrown by the compiler.

The main challenge lies in the fact in determining whether the variable is an identifier, arrays, pointers or conclusion about type of the variable if it is an integer, float, character, double, long int and so on. The C99 standard, removes the implicit integer rule that states a variable declared without an explicit data type is assumed to be integer which was previously defined in C89 standard. Therefore, there is a need for determining the variables that are undeclared along with its type, else the compiler will throw an error.

```
1   #include <stdio.h>
2   int main()
3   {
4       int i,max,j,n,m,y;
5       scanf("%d %d",&n,&m);
6       for(i=1;i<=n;i++){
7           s=0;
8           for(j=1;j<=m;j++){
9           scanf("%d",&y);
10          s=s+j;
11          }
12          if(max<s)
13          max=s;
14      }
15      return 0;
16  }
```

Figure 5.1: An example illustrating the undeclared variable "s" that is frequently used in the program is caught by the compiler

```
1   #include <stdio.h>
2   int main()
3   {
4       int n,m;
5       int i,j;
6       int a[20];
7       int sum=0;
8       scanf("%d%d",&n,&m);
9       for(i=0;i<n;i++){
10          for(j=0;j<m;j++){
11              scanf("%d",&a[j]);
12              sum=sum+a[j];
13          }
14          printf("%d\n",sum);
15          i++;
16          J++;
17      }
18      return 0;
19  }
```

Figure 5.2: An example of compiler error caused by an undeclared variable "J" that is used once in the program

5.2 Model

In our prediction model, we use non-terminals and terminals obtained from the AST as the input. The main ideology behind our model is to specify a declaration for any identifier that is used throughout a C program. Here, we assume identifier in our context that excludes keywords and only includes alphanumeric variables. This in turn solves the complex problem of automatically fixing the undeclared variables.

5.2.0.1 Declaration Classification and Prediction:

`ID` is the non-terminal node of the AST that represents an identifier excluding keywords as mentioned above. `Decl` is the non-terminal node of the AST representation that is entitled to represent the declaration of the identifier where its terminal node is the corresponding identifier itself. Similarly, `TypeDecl` and `IdentifierType` are the semantic elements that is used to represent the type specifier information of the identifier where identifier and its type are its corresponding terminals respectively.

For the classification purpose, the non-terminal and terminal pair of `ID` and any alphanumeric identifier variables is augmented along with pairs of `Decl` and the respective identifier, `TypeDecl` and the identifier, `IdentifierType` and a generalized "type" referring to the corresponding types of those identifier variables so that they can be used for backsubstitution which will be explained later in the generation approach. After classification, the LSTM model is used to predict the `Decl`,`TypeDecl` and `IdentifierType` information for any alphanumeric identifier variable occurring with its corresponding non-terminal node `ID`. The classification model is sequential where the embedding layer, LSTM layer and softmax layers are stacked on top of each other sequentially which we will be understand the layers that are detailed below in this section.

5.2.0.2 Embedding Layer of non-terminal and terminal nodes

The sequences of input tokens are a combination of non-terminal and terminal nodes where in our model, we consider only 4 non-terminals `ID, Decl, TypeDecl, IdentifierType` and all the alphanumeric identifier variables as the terminals. These input tokens are formed by the concatenation of individual string encodings of non-terminal and terminal node vocabularies (that is discussed in evaluation section), subsequently embedding is computed on the integer encodings(converted from string) for performing the training of the model. The embeddings are computed as follows:

$$E_{\mathrm{i}} = A * concat(N_{\mathrm{i}} T_{\mathrm{i}}) \tag{5.1}$$

where A is $K \times V_{\mathrm{N,T}}$ matrix where K is the embedding vector size and $\mathrm{V_{N,T}}$ is the vocabulary size formed by the concatenated encodings of non-terminal and terminal nodes.

5.2.0.3 LSTM layer

The sequences of embedded tokens are passed on to the LSTM layer containing LSTM memory cells where each cell state stores information of the previous state and are controlled by the forget gate, input gate and output gate layers as mentioned above in equations (4.7), (4.8), (4.9), (4.10). Each LSTM cell state takes inputs from its previous LSTM cell hidden state $\mathrm{h_{i-1}}$,state information $\mathrm{s_i}$ as well as the input tokens and outputs the hidden state $\mathrm{h_i}$ of LSTM cell as in equation (4.11) where LSTM layers can be seen from Figure 5.3.

5.2.0.4 Dense Softmax Activation layer

The last LSTM memory cell state's output hidden state of the LSTM layer is passed to the softmax activation layer to predict the sequences of non-terminals `Decl`,`TypeDecl` and `IdentifierType` given the non-terminal `ID`. The predicted output sequences at timestep t

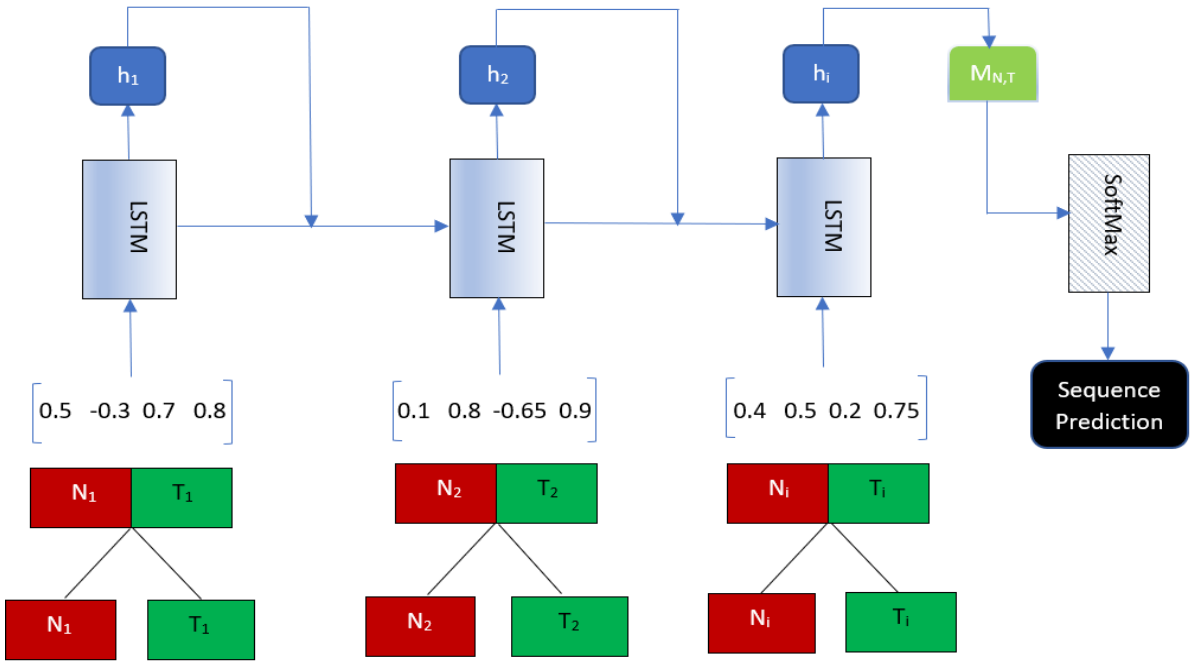


Figure 5.3: Illustration of the approach showing the concatenation of non-terminal and terminal node embeddings extracted from AST being used as the inputs to the LSTM model for the sequence classification and prediction.

(or fixed input sequence length) are represented by $\hat{y}(t)$ and is formulated as:

$$\hat{y}(t) = softmax(b_{\mathrm{N,T}} + M_{\mathrm{N,T}} h_{(\mathrm{t})}) \tag{5.2}$$

where $\mathrm{b_{N,T}}$ is the bias unit of softmax layer with size of $\mathrm{V_{N,T}}$ dimensional vector, $\mathrm{M_{N,T}}$ is a weight matrix of size $K \times V_{\mathrm{N,T}}$ and $\mathrm{h_{(t)}}$ is the hidden state of the LSTM cells at each timestep t.

CHAPTER 6. EVALUATION AND RESULTS

6.1 Dataset

The dataset that used in this approach is prutor which is a database that has student coding submissions for university programming assignments. It contains a set of 53478 C programs out of which there are 6978 erroneous programs which contains multiple and single line syntax as well as semantic errors. Out of 6978 programs, 1059 programs contains only undeclared variable errors which is the main focus of our evaluation.

6.2 Preprocessing and Training Details

We use pycparser, that acts as a front-end of the C compiler to parse source code of C language in python. AST are obtained as an output for the source code after the parsing stage and are stored in text files.

The source code is preprocessed in the form of tokens that represents the terminal nodes of its corresponding AST. Since the set of tokens are uncontinuous, discrete and in its textual form, it must be encoded into sequences of numerical vectors to be used for training the model. Additionally, there are 47 fixed set of non-terminals in C language that are encoded as in figure 6.1. The terminals can be keywords, strings, data types, integers or floating point numbers. The terminals of the above mentioned categories are encoded separately in a specific range of numbers. Now, the data for training is prepared by concatenating the encodings of non-terminals and terminals together. The individual encodings in the form of

integers are converted to strings initially and after concatenation, they are converted back to integers. For example, the non-terminal `IdentifierType` is encoded as 9 in the dictionary of non-terminals and converted to '9', if the terminals are data types like int, float, long etc. then they are encoded as 111111 referring to a generalized 'type' and converted to '111111'. Therefore, the concatenation of the non-terminals and terminals are mapped accordingly and stored in separate vocabulary. This vocabulary set is used in performing the training of the model. One-hot encoding approach is used to perform categorical multi-class classification to represent the elements of vocabulary as vectors with each of them of vocabulary size containing 1 at the corresponding index and rest of them are 0.

Training Details:

The training experiment is performed by using embedding dimension of size 512 and two LSTM layers are used each with number of hidden units as 512 and a dropout of 0.5. The input sequence length is 1, batch size is 3, vocabulary size is 583 and the total number of sequences is 2319. The dense layer is used for forming a fully connected layer in which each of the input layer nodes is connected with every hidden and output layer nodes. The activation function used in our model is softmax function because of its efficiency in dealing with multi-class classification problems compared to sigmoid and ReLU due to the fact that the outputs of the softmax is a categorical probability distribution summing to 1 and lying between 0 to 1. The total number of units of Dense layer is equal to the vocabulary size.

The vocabulary formed from concatenation is split into training and testing data where test data size is 0.2 and the split rule percentage used is 80/20. The loss function used is categorical cross-entropy function and performs a cross-entropy loss calculation due to the fact that the activation function used is softmax and it is efficient for multi-classification. The optimizer used is RMSprop with a learning rate of 0.01 as it is better in handling non

local maxima or minima points and has a constant initial global learning rate compared to optimizers such as Stochastic gradient descent optimizer.

6.3 Generation Approach

During generation, one-hot encoding is used to represent the new unseen sequences of the nonterminal `ID` and a terminal variable as a categorical distribution and the output class is classified as the sequences of non-terminals `Decl,TypeDecl,IdentifierType` along with the corresponding terminal variables.

6.3.1 AST Transformation

The program fix approach is carried out through an AST transformation performed by augmenting the predicted output sequences in each of the program's AST syntactical structure for the terminal variables associated with its corresponding non-terminal node `ID`. This augmentation is carried out on the source codes by performing a check on declaration of the variables in the vocabulary set of concatenated encodings of non-terminal and terminal nodes as created previously using the predicted output sequence of the non-terminal `Decl` and the associated terminal variable. If any of the predicted output sequences does not match with the declared variables present in a source code, then the output sequence `Decl` containing the particular terminal variable is augmented to the original AST structure of the code through serialization and deserialization that will be described below.

6.3.2 Serialization and Deserialization

Serialization is implemented using pycparser by transforming the data structures such as nodes of the python Node object by traversing the AST (the nodes being obtained by

parsing the source code also from pycparser previously) recursively into a dictionary object representation which is then subsequently serialized into a JSON object format that can be understood by the pycparser in deserializing it back to a dictionary object and thereby consequently back to the AST Node objects. An example JSON object representation of a AST node object is shown in figure 6.2 where the `_nodetype` key refers to the different types of non-terminal nodes such as `Decl, ArrayDecl, TypeDecl, IdentifierType`. The `TypeDecl` and `ArrayDecl` are the child nodes of `Decl` and `IdentifierType` is the child node of the intermediate non-terminal `TypeDecl` node. The key `name` refers to the terminal variables, `type` refers to the datatypes of the declared terminal nodes of the AST.

```
{
    "_nodetype": "Decl",
    "bitsize": null,
    "funcspec": [],
    "init": null,
    "name": "j",
    "quals": [],
    "storage": [],
    "type": {
        "_nodetype": "TypeDecl",
        "declname": "j",
        "quals": [],
        "type": {
            "_nodetype": "IdentifierType",
            "names": [
                "int"
            ]
        }
    }
}
```

Figure 6.1: Example demo of JSON object containing Decl, TypeDecl and IdentifierType nodes.

The serialization and deserialization is carried out when there is an AST transformation performed consistently without disintegrating the program ensuring its maximal quality.

6.3.2.1 Pre-compile Time Type Binding and Analysis Results

After determining and performing the augmentation of the undeclared terminal variables, the type of those undeclared variables is determined before compiling the program. The type binding is performed by determining the type of a `lvalue` from its `rvalue` in an assignment statement or finding the type of a undeclared variable from its neighboring variables in an expression statement of the source program. The type of the undeclared variables is determined and drawn from the following cases:

```
1   int main()
2   {
3   int k,n,x,a[100];
4   scanf("%d", &k);
5   scanf("%d", &n);
6   for(i=0;i<n;i++)
7         scanf("%d", &a[i]);
8
9   for(i=0;i<n;i++)
10  {
11        x=k-a[i];
12  }
13  return 0;
14  }
```

```
1   int main()
2   {
3   int i;
4   int k;
5   int n;
6   int x;
7   int a[100];
8   scanf("%d", &k);
9   scanf("%d", &n);
10  for (i = 0; i < n; i++)
11        scanf("%d", &a[i]);
12
13  for (i = 0; i < n; i++)
14  {
15  x = k - a[i];
16  }
17  return 0;
18  }
```

Figure 6.2: Case 1 demonstrating location of error in the for loop statement involving undeclared identifier "i" and the fix of it

Case 1: When the `rvalue` is a constant, where the non-terminal is `Constant` and is of integer type, then the `lvalue` whose non-terminal is `ID`, is assigned as an integer type which

can be seen from the figure 6.3 where in the assignment statement **i=0** on the left side of the figure, "i" is undeclared and is assigned to `integer` type.

```
1   int main()
2   {
3   int n[1000],a[500],nm,i
        ,j,ln,flag=0;
4   scanf("%d\n",&ln);
5   scanf("%d\n",&nm);
6   for(i=0;i<500;i++)
7   {
8       a[i]=0;
9   }
10  for(i=0;i<nm;i++)
11  {
12      scanf("%d ",&nm);
13      c=n[i];
14      a[c]=a[c]+1;
15  }
16  return 0;
17  }
```

```
1   int main()
2   {
3   int c;
4   int n[1000];
5   int a[500];
6   int nm;
7   int i;
8   int j;
9   int ln;
10  int flag = 0;
11  scanf("%d""\n", &ln);
12  scanf("%d""\n", &nm);
13  for (i = 0; i < 500; i++)
14  {
15      a[i] = 0;
16  }
17  for (i = 0; i < nm; i++)
18  {
19      scanf("%d" , &nm);
20      c = n[i];
21      a[c] = a[c] + 1;
22  }
23  return 0;
24  }
```

Figure 6.3: Case 2 indicating the error in assignment statement between variable and array identifier

Case 2: In this case, if the `rvalue` is an identifier that refers to an array element whose non-terminal is `ID` and its non-terminal parent is `ArrayRef`, then the `lvalue` terminal variable with non-terminal `ID` is assigned to the respective type of `rvalue` element. We can see in figure 6.4 where "b" is undeclared and is assigned to `integer` type of the `array` "n[1000]" from the statement **b=n[i]**.

```
1  #include<stdio.h>
2  int main()
3  {
4  int n,i,j,max;
5  int a[20];
6  for(i=0;i<n;i++)
7  {
8       for(j=i;j<n;j++)
9       {
10           if(a[i]<a[j])
11           {
12           count=count+1;
13           }
14      }
15 if(count>max){max=count;}
16 }
17 printf("%d",max);
18 return 0;
19 }
```

```
1  #include<stdio.h>
2  int main()
3  {
4  int count;
5  int n;
6  int i;
7  int j;
8  int max;
9  int a[20];
10 for (i = 0; i < n; i++)
11 {
12   for (j = i; j < n; j++)
13   {
14     if (a[i] < a[j])
15     {
16       count = count + 1;
17     }
18   }
19   if (count > max)
20   {
21     max = count;
22   }
23 }
24 printf("%d", max);
25 return 0;
26 }
```

Figure 6.4: Case 3 illustrating the repair of assignment statement of variable and array identifier

Case 3: If the non-terminal of `rvalue` and non-terminal of `lvalue` are the children nodes of the non-terminal `BinaryOp`, then the type of `rvalue` is assigned as the type of `lvalue`. In the figure 6.5, in the conditional expression statement **count > max**, `BinaryOp` is ">", the `lvalue` "count" is undeclared with its non-terminal being `ID` and the `rvalue` is max with its non-terminal also referring to `ID`. Therefore the variable count is assigned to `integer` type from the variable max.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4  int n, i, j, k;
5  scanf("%d", &n);
6  for(i=1; i<=n; i++)
7  {
8  for(j=1,z=i;j<=i;j++,k--)
9       {
10           if((k%2) == 0)
11             printf("*");
12           else
13             printf("x");
14      }
15 printf("\n");
16 }
17 return 0;
18 }
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5  int z;
6  int n;
7  int i;
8  int j;
9  int k;
10 scanf("%d", &n);
11 for (i = 1; i <= n; i++)
12 {
13 for (j = 1,z = i;j<=i;j++,k--)
14   {
15     if ((k % 2) == 0)
16       printf("*");
17     else
18       printf("x");
19
20   }
21
22   printf("\n");
23 }
24
25 return 0;
26 }
```

Figure 6.5: Case 4 illustrating the fix of variable "z" from the for loop statement

Case 4: This case is similar to case 2 but it deals with the assignment of a variable to another variable instead of array element. In this figure 6.6, in the left picture, the `lvalue` variable "z" inside the `For` statement is undeclared, and its non-terminal node is `ID`, it is assigned to the integer type of the variable "i" whose non-terminal is `ID`.

Case 5: This case deals with binary operation involved in an assignment expression state-

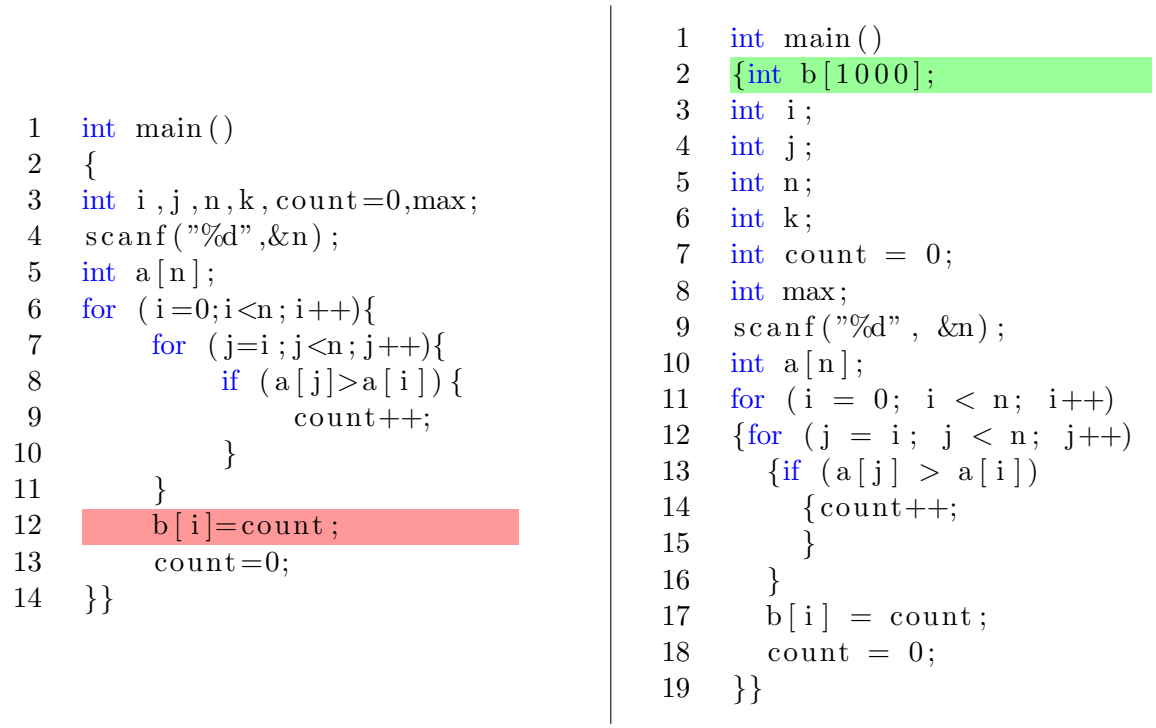
ment. According to the left picture of figure 6.7, the terminal variable "t" is undeclared and is assigned to the type of the terminal variable "summation" which is of type `double`. In the statement **summation = summation + t*delx**, there is non-terminal `Assignment` and its children nodes being the non-terminal `ID` with terminal "summation" variable and the node `BinaryOp` with its corresponding children nodes `ID`:summation, `BinaryOp:+`, `BinaryOp:*` with its children `ID`:t, `ID`:delx.

```
1   double sum(double a, double
        n, double delx)
2   {
3   double sum=0;
4   int j;
5   for (j=0;j<n;j++)
6   {double x=a+j*(delx);
7    double r=fabs(f(x)-g(x));
8    sum=sum+t*delx;
9   }
10  return sum;
11  }
```

```
1   double sum(double a, double n,
        double delx)
2   {
3   double t;
4   double sum = 0;
5   int j;
6   for (j = 0; j < n; j++)
7   {
8   double x = a + (j * delx);
9   double r = fabs(f(x) - g(x));
10  sum = sum + (t * delx);
11  }
12
13  return sum;
14  }
```

Figure 6.6: Case 5 demonstrating the error in binary operation and undeclared "t" getting fixed

Case 6: This case is an exact opposite of case 2 where the `lvalue` in the figure 6.8 is an array identifier "b" undeclared, whose non-terminal node is `ID` and its parent node is `ArrayRef` and `rvalue` is a terminal variable "count" whose non-terminal node `ID` is assigned of `integer` type from the type of `lvalue`.



```
1   int main()
2   {
3   int i,j,n,k,count=0,max;
4   scanf("%d",&n);
5   int a[n];
6   for (i=0;i<n;i++){
7        for (j=i;j<n;j++){
8             if (a[j]>a[i]){
9                  count++;
10            }
11       }
12       b[i]=count;
13       count=0;
14  }}
```

```
1   int main()
2   {int b[1000];
3   int i;
4   int j;
5   int n;
6   int k;
7   int count = 0;
8   int max;
9   scanf("%d", &n);
10  int a[n];
11  for (i = 0; i < n; i++)
12  {for (j = i; j < n; j++)
13    {if (a[j] > a[i])
14      {count++;
15      }
16    }
17    b[i] = count;
18    count = 0;
19  }}
```

Figure 6.7: Illustration of case 6 marked by red line indicating the error in assignment statement

Case 7: This case is slightly similar to case 5 but it does not involve any assignment operation. The type can be assigned to a variable not only from `lvalue` but also from its neighbouring variables involved in a binary operation. As we can see in left side of the figure 6.9, the terminal variable "diff" being undeclared is involved in a binary operation `BinaryOp:*` and `BinaryOp:+` with the terminal variables "key" and "a" respectively, so the variable "diff" is assigned to the `integer` type from variables "key" and "a".

Case 8: In this case, the type of a variable is bound from the type of a function call. In figure 6.10, the terminal variable "k" is undeclared in the `for` expression **k >= hanoi(j)-1**

```
1  int main()
2  {
3  const double E=0.000001;
4  double a,b,interval,sub=0;
5  int n,key=0;
6  scanf("%lf%lf%d", &a, &b,
       &n);
7  interval=(b - a)/n;
8  while(key<n&&a + diff*key)
9  {
10       sub+=1;
11       key++;
12 }
13 while(key<n)
14 {
15       sub=a+interval*key;
16       key++;
17 }
18 printf("%.4lf",sub*
      interval);
19 return 0;
20 }
```

```
1  int main()
2  {
3  int diff;
4  const double E = 0.000001;
5  double a;
6  double b;
7  double interval;
8  double sub = 0;
9  int n;
10 int key = 0;
11 scanf("%lf""%lf""%d", &a, &b, &
      n);
12 interval = (b - a) / n;
13 while ((key < n) && (((a + (diff
       * key)) + 1) < E))
14 {
15   sub += 1;
16   key++;
17 }
18 while (key < n)
19 {
20   sub=a+interval * key;
21   key++;
22 }
23 printf(%.4lf, sub * interval);
24 return 0;
25 }
```

Figure 6.8: Demo of case 7 involving the error in while loop statement

and it is being involved in a binary operation `BinaryOp:>=` with the function call **hanoi(j)** where the corresponding non-terminal node of the terminal "hanoi" is `ID` and parent node being `FuncCall`, is of type `integer`.

```
1  int main() {
2  int t,i,n,j;
3  int x;
4  scanf("%d",&t);
5  for(i=1;i<t;i++)
6  {
7  scanf("%d",&n);
8  for(j=0;k>=hanoi(j)-1;j++)
9  {
10       if(hanoi(j)-1==k)
11            printf("yes");
12       else
13            printf("no");
14 }
15 }
16 return 0;
17 }
```

```
1  int main()
2  {
3  int k;
4  int t;
5  int i;
6  int n;
7  int j;
8  int x;
9  scanf("%d", &t);
10 for (i = 1; i < t; i++)
11 {
12    scanf("%d", &n);
13    for (j = 0; k >= (hanoi(j) - 1);
           j++)
14    {
15    if ((hanoi(j) - 1) == k)
16       printf("yes");
17    else
18       printf("no");
19    }
20 }
21 return 0;
22 }
```

Figure 6.9: Case 8 indicating the undeclared "k" in for loop statement

Case 9: This case is similar to case 8, however instead of a binary operation, the type of the function call is a `rvalue` is bound to a `lvalue` variable in an assignment expression statement. This can be seen in figure 6.11, in which the variable "y" is undeclared in the assignment expression **y = tower(j)-1** and is assigned to the type of the function call **tower(j)** whose non-terminal node is `ID` and its parent node is `FuncCall`

```
1  int main()
2  {int i,n,j,t;
3   scanf("%d\n",&n);
4   for(i=1;i<=n;i++)
5   {
6   scanf("%d\n",&t);
7   for(j=1;j<=200;j++)
8   {y=tower(j)-1;
9  if(t<0||t<y) printf("no");
10 if(t==y) printf("%d",y);
11  }
12  }
13 return 0;
14 }
```

```
1  int main()
2  {
3  int y;
4  int i;
5  int n;
6  int j;
7  int t;
8  scanf("%d""\n", &n);
9  for (i = 1; i <= n; i++)
10 {
11    scanf("%d""\n", &t);
12    for (j = 1; j <= 200; j++)
13    {
14       y = tower(j) - 1;
15       if ((t < 0) || (t < y))
16          printf("no");
17
18       if (t == y)
19          printf("%d", y);
20    }
21 }
22 return 0;
23 }
```

Figure 6.10: Case 9 demonstrating the undeclared identifier "y" in the assignment statement with a function call expression

Table 6.1 shows the results of analysis obtained after performing the compilation of the programs manually where the first row consists of all the programs (1059) that are containing both undeclared variables and arrays in which our approach has located and identified the undeclared variables in 887(83.7%) programs out of total number of 1059 programs. However, our approach has correctly located and identified them in 857(80.9%) programs, but the repair is performed on 844(79.7%) programs by correctly locating as well as inferring and binding their types. The first column in the rest of the rows are not applicable as the results are analyzed only on programs in which undeclared variables are identified and located.

Similarly, the second row shows the results for programs with only undeclared variables and consisting of only single main functions (571) out of the identified programs (887). The third row displays the results of programs with undeclared variables and containing two or more functions including main function (195) out of 887 programs. The fourth row demonstrates the results shown by programs only with errors caused due to undeclared arrays and having one and only main function (93) out of the 887 programs. Finally, the last row illustrates the number of programs which contains undeclared variables and having two or more functions along with main function (22) out of those 887 programs.

Table 6.2 shows the summary of various cases along the rows and its brief description message along the columns through which the type binding is performed before the compile-time.

Table 6.1: Analysis results of both the undeclared variables and arrays

	Identified	**Not Identified**	**Correctly identified (True Positive)**	**Wrongly Identified (False Positive)**	**Correctly Identified + Correct Type Inferred (Fixed)**	**Wrongly identified + Wrong Type Inferred (Not Fixed)**	**Total**
Undeclared Variables and Arrays	887(83.7%)	172	857(80.9%)	202	844(79.7%)	215	1059
Undeclared variables - Main function	N/A	N/A	566(99.1%)	5	560(98%)	11	571
Undeclared variables - Two/more functions	N/A	N/A	179(91.7%)	16	172(88.2%)	23	195
Undeclared Arrays - Main functions	N/A	N/A	90(96.8%)	3	90(96.8%)	3	93
Undeclared Arrays - Two/more functions	N/A	N/A	22(78.5%)	6	22(78.5%)	6	28

Table 6.2: Summary of Type Binding Cases and their Description

Cases	**Brief Description**
Case 1	Assignment expression statement with a constant on the right-hand side of the expression
Case 2	Assignment expression statement with an array identifier on the right-hand side and an identifier on the left-hand side of the expression
Case 3	Conditional expression statement with a binary operation between the identifier variables
Case 4	Assignment expression statement with an identifier excluding the array identifier on the right-hand side of the expression
Case 5	Assignment expression statement with a binary operation between the identifier variables
Case 6	Assignment expression statement with an identifier excluding the arrays on the right-hand side and an array identifier on the left-hand side of the expression
Case 7	Binary operation between identifier variables in a loop expression statement
Case 8	Conditional expression statement with a binary operation between an identifier and a function call expression
Case 9	Assignment expression statement with a binary operation between an identifier and a function call expression

CHAPTER 7. DISCUSSION, CONCLUSION AND FUTURE WORK

7.1 Discussion

There are few limitations. Fixing the undeclared variables caused due to imperceptible spelling mistakes or a variable that is used only once throughout the program may cause the program to run in an infinite loop or lead to some possible run-time errors. As seen in figure 7.1, instead of incrementing the variable "j" inside the `for` loop, the programmer had used "J" instead which had caused the program to run into an infinite loop. Another major limitation is in the type binding approach in figure 7.1 shows an example where, the type has been wrongly bound to the variable "l" because "l" is used in the `for` loop expression in which "l" is assigned to constant "1" as well as it is used in an assignment expression inside the `for` loop body, in the statement **if((l*k) < -1)**, variable "k" is of the type `double` and "*" is a binary operation, so the variable "j" should be assigned of the type `double` instead it is inferred as an `integer` type due to the former case.

We had seen in our model that a vocabulary in the form of hash table is used for training purposes. The purpose of training neural networks on the hash table is due to the fact that it can be used for recognizing input patterns (keys) in the hash table and can be used to predict the sequences (values). Consider the case where an input pattern is not present in the hash table and we need to predict the sequence, a hash table would have return null in this case but neural networks will give the closest sequence prediction.

The benefits of our approach lies in the fact that our model could be used in real-time as a tool for any C programming environment online or offline editors in locating, reporting and repairing undeclared identifiers for any C programs. Additionally, our model can be used when there are lack of positive bug-free syntactically correct and executing source program reference examples for buggy source programs. Also, our type binding approach will be applicable even for declared variables.

7.2 Conclusion and Future Work

In this paper, we had seen different cases of one of the most common semantic error: undeclared variables. We had used a combination of AST and LSTM approach to extract a set of non-terminal and terminal nodes to carry out the classification and prediction tasks of the undeclared variables. We had also seen the generation of clean and buggy-free source programs by performing AST transformation and serialization as well as deserialization of AST to JSON and vice- versa. Furthermore, in this paper we had coined a new term known as Pre-Compile time Type binding where we had implemented the fix of the types of undeclared variables by binding them their corresponding types before providing it for the compiler to compile them. By our approach, we had correctly identified 81% of the programs that contains only undeclared identifier errors. Also, we had fixed those undeclared identifier errors by binding their corresponding types in 80% of the programs. In future, we would like to perform automatic repair on different types of syntactic, semantic errors and logical errors. Further, we plan to perform type binding for the limitation cases in figure 16 as well as also implement a repair approach for the logical errors that arises after the repair of syntactic and semantic errors caused by spelling mistakes/variable exchange.

```c
#include <stdio.h>
int main()
{
int J;
int n;
int i;
int j;
int flag = 0;
scanf("%d", &n);
int a[51];
for (i = 0; i < n; i++)
{
   scanf("%d", &a[i]);
}
for (i = 0; i < n; i++)
{
   for (j = 0; j < n; J++)
   {
      if (a[i] == a[j])
      {
         printf("YES");
         flag = 1;
         break;
      }
   }
   if (flag == 1)
   {
      break;
   }
}
if (flag == 0)
{
   printf("NO");
}
return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
int l;
double a;
double b;
double k;
double p;
int n;
scanf("%f" "%f" "%d", &a, &b, &n);
k = ((a - b) * 1.0) / n;
for (l = 1; l <= n; l++)
{
   if ((l * k) < (-1))
      p += k;

   if (((l * k) >= (-1)) && ((l * k)
        <= 1))
      p = p + (((l * k) * (l * k)) *
          k);

   if ((l * k) > 1)
      p = p + ((((l * k) * (l * k))
          * (l * k)) * k);
   }
printf("%.4f", p);
return 0;
}
```

Figure 7.1: Picture on the left side illustrates the repair that caused infinite loop due to the variable "J" incremented in the for loop expression and the picture on the right side depicts the repair that caused by binding type "int" instead of "double"

BIBLIOGRAPHY

A. Viet Phan, M. L. N. and Bui, L. T. (2017). Convolutional neural networks over control flow graphs for software defect prediction. In *IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI), Boston, MA, 2017*, pages 45–52.

Ahmed, U. Z., Kumar, P., Karkare, A., Kar, P., and Gulwani, S. (2018). Compilation error repair: for the student programs, from the student programs. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 78–87. IEEE.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bhatia, S. and Singh, R. (2016). Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129*.

Carlos Araya, Ivan Sanabria, F. Z. Programming language transformations with abstract syntax tree extensions.

Choudhary, H., Pathak, A. K., Saha, R. R., and Kumaraguru, P. (2018). Neural machine translation for english-tamil. In *Proceedings of the Third Conference on Machine Translation: Shared Task Papers*, pages 770–775.

Dam, H. K., Tran, T., Pham, T., Ng, S. W., Grundy, J., and Ghose, A. (2017). Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*.

F. Pfenning, C. E. (1988). Higher-order abstract syntax. In *PLDI '88 Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation*, pages 199–208.

G.Fischer, J.Lusiardi, and von Gudenberg, J. (2007). Abstract syntax trees - and their role in model driven software development. In *International Conference on Software Engineering Advances (ICSEA 2007)*, pages 38–38.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial networks. In *Advances in neural information processing systems*, pages 2672–2680.

Guo, J., Lu, S., Cai, H., Zhang, W., Yu, Y., and Wang, J. (2018). Long text generation via adversarial training with leaked information. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Gupta, R., Pal, S., Kanade, A., and Shevade, S. (2017). Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

Harer, J., Ozdemir, O., Lazovich, T., Reale, C., Russell, R., Kim, L., et al. (2018). Learning to repair software vulnerabilities with generative adversarial networks. In *Advances in Neural Information Processing Systems*, pages 7933–7943.

Jayanthi, R. and Florence, L. (2018). Software defect prediction techniques using metrics based on neural network classifier. *Cluster Computing*, pages 1–12.

Kosovan, S., Lehmann, J., and Fischer, A. (2017). Dialogue response generation using neural networks with attention and background knowledge. In *Proceedings of the Computer Science Conference for University of Bonn Students (CSCUBS)*, volume 2017.

Li, J., He, P., Zhu, J., and Lyu, M. R. (2017a). Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE.

Li, L., Feng, H., Zhuang, W., Meng, N., and Ryder, B. (2017b). Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260. IEEE.

Liu, C., Wang, X., Shin, R., Gonzalez, J. E., and Song, D. (2016). Neural code completion.

Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Peng, H., Mou, L., Li, G., Liu, Y., Zhang, L., and Jin, Z. (2015). Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer.

Rabinovich, M., Stern, M., and Klein, D. (2017). Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*.

Rajeswar, S., Subramanian, S., Dutil, F., Pal, C., and Courville, A. (2017). Adversarial generation of natural language. *arXiv preprint arXiv:1705.10929*.

Welty, C. A. (1997). Augmenting abstract syntax trees for program understanding. In *Proceedings 12th IEEE International Conference Automated Software Engineering, NV, USA*, pages 126–133.

Wile, D. S. (1997). Abstract syntax from concrete syntax. In *ICSE*, volume 97, pages 472–480. Citeseer.

Wiseman, S., Shieber, S. M., and Rush, A. M. (2018). Learning neural templates for text generation. *arXiv preprint arXiv:1808.10122*.

Yuan, X., Wang, T., Gulcehre, C., Sordoni, A., Bachman, P., Subramanian, S., Zhang, S., and Trischler, A. (2017). Machine comprehension by text-to-text neural question generation. *arXiv preprint arXiv:1705.02012*.