B.Sc. in Computer Science and Engineering Thesis

Knowledge Graph Generation from Program Source Code for Semantic Representation

Submitted by

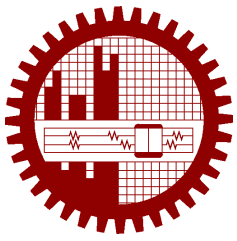
Sourav Saha Dip
201505003

MD Al Imran
201505100

Waqar Hassan Khan
201505107

Supervised by

Dr. Muhammad Masroor Ali



Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

Dhaka, Bangladesh

February 2021

CANDIDATES' DECLARATION

This is to certify that the work presented in this thesis, titled, "Knowledge Graph Generation from Program Source Code for Semantic Representation", is the outcome of the investigation and research carried out by us under the supervision of Dr. Muhammad Masroor Ali.

It is also declared that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Sourav Saha Dip
201505003

MD Al Imran
201505100

Waqar Hassan Khan
201505107

CERTIFICATION

This thesis titled, **"Knowledge Graph Generation from Program Source Code for Semantic Representation"**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering in February 2021.

Group Members:

Sourav Saha Dip

MD Al Imran

Waqar Hassan Khan

Supervisor:

Dr. Muhammad Masroor Ali
Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

ACKNOWLEDGEMENT

We are indebted to our supervisor Dr. Muhammad Masroor Ali sir for introducing us to the research world, for providing us with directions whenever we got stuck, for teaching us how to go through any literature. We are thankful to him for introducing us to the world of knowledge graph. Also for helping us improving our writing and presentation skills.

We are also thankful to our parents for their continuous support they have given to us throughout undergrad life.

Dhaka
February 2021

Sourav Saha Dip

MD Al Imran

Waqar Hassan Khan

Contents

List of Figures

ABSTRACT

This research work concentrates on generating knowledge graph and RDF-triples from C source code. A knowledge graph is a set of related entities that are interconnected using relations. These entities can be documents, events, real-life objects, or even abstract concepts. The RDF (Resource Description Framework) is a framework for representing information on the Web. It represents entities and relationships between them. This research uses semantic analysis on source code to generate a knowledge graph that has a graph-like representation, i.e: entities are represented as vertices and the relationship between them as an undirected edge between them. The RDF-triple that is generated is in RDF/XML format. At first, the codes are analyzed lexically and fed into a semantic analyzer that follows certain grammar. The semantic analyzer not only finds out any error that is present in the code but also uses a data-structure defined by us to create the knowledge graph. It also creates RDF-triples alongside knowledge graph using a template written using Protege. The semantic analyzer covers a subset of C

Chapter 1

Introduction

In this chapter we have represented an overview of our work, what we have already done and what we will be presenting in the upcoming chapters.

1.1 Knowledge Graph Generation of Source Code

Knowledge Graph plays a vital role in the domain of search engine. Knowledge graph is the graph representation of entities and the relations between them. It enables us to fetch all the data related to any entity thus improving search engine results.

In this research our main target is to generate knowledge graph from C source code. We consider the building blocks of C source code such as variables, functions, loops etc. as entities and have tried to create relation between them and their attributes.

We also tried to create RDF-triples from the source code. An RDF Triple is a statement which relates one object to another and can be visualized in ontology building tools like Protege [1].

We got the motivation of the work from [2] where they have extracted RDF-triples for a subset of Java source code.

1.2 Methodology in Brief

Our task was to generate knowledge graph and RDF-triples from C source codes. We followed the compilation process of C code where it goes through syntax analysis, lexical analysis, and semantic analysis. During semantic analysis we used a data structure to hold all the information and finally we have written them in a text file, in a knowledge graph representation defined by us.

1.3 Organization of the Thesis

In the upcoming chapters we have presented our work briefly. In Chapter 2 we have discussed about the terms used throughout the literature. In Chapter 3 we have discussed about how we would generate knowledge graph from the source code. The steps to analyze the source code. In Chapter 4 we have presented the outputs given by our proposed framework. And, finally in Chapter 5 we have discussed how can we improve the current approach and future works.

The next chapter focuses on the theory related to our work.

Chapter 2

Methodology

In order to generate knowledge graph or RDF-triples, we need information about the building blocks of the source code. To do that we have used the steps of compilation of C code. We have used parsers, lexical analyzers. This chapter focuses on the theory related to parsers and what we are going to generate finally.

2.1 Parser

Parsing is the process of determining how a string of terminals can be generated by a grammar [3]. A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens, interactive commands, or program instructions and breaks them up into parts that can be used by other components in programming. A parser usually checks all data provided to ensure it is sufficient to build a data structure in the form of a parse tree or an abstract syntax tree. A parsers main purpose is to determine if input data may be derived from the start symbol of the grammar and in what ways this input data can be derived. There are two methods of parsing:

1. **Top-Down Parsing:** Involves searching a parse tree to find the left-most derivations of an input stream by using a top-down expansion. Parsing begins with the start symbol which is transformed into the input symbol until all symbols are translated and a parse tree for an input string is constructed. Examples include LL parsers and recursive-descent parsers. Top-down parsing is also called predictive parsing or recursive parsing.

2. **Bottom-Up Parsing:** Involves rewriting the input back to the start symbol. It acts in reverse by tracing out the rightmost derivation of a string until the parse tree is constructed up to the start symbol This type of parsing is also known as shift-reduce parsing. One example is an LR parser.

The parser used in this thesis uses a bottom-up parsing approach.

There are mainly three stages involved in parsing: Lexical Analysis, Syntax Analysis and Semantic Analysis.

2.1.1 Lexical Analysis

A lexical analyzer reads characters from the input and groups them into "token objects" [3]. It is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. A lexical analyzer is used to produce tokens from a stream of input string characters, which are broken into small components to form meaningful expressions. A token is the smallest unit in a programming language that possesses some meaning. For example, C language has tokens like- identifiers, operators, keywords, punctuators etc. Two sample statements in C is given below:
$int\ x;$
$x = 5 + 3;$

In these statements, int, x, 5, 3, $+$, $=$, ; are different tokens. The scanner first divides these statements into the tokens, then determines $5 + 3$ as an expression. The expression is then evaluated and the value 8 is stored in x. The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a **lexeme**. When the lexical analyzer reads the source-code, it scans the code letter by letter; and when it encounters a white-space, operator symbol, or special symbols, it decides that a word is completed. If the analyzer finds a token invalid, it generates an error.

2.1.2 Syntax Analysis

Syntax Analysis is a second phase of the compiler design process in which the parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream [3]. It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not. A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree. Syntax Analysis in the compiler design process comes after the Lexical analysis phase. A lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata. Syntax Anal-

ysis is also known as the Parse Tree or Syntax Tree. The Parse Tree is developed with the help of pre-defined grammar of the language. The syntax analyzer also checks whether a given program fulfills the rules implied by a context-free grammar. If it satisfies, the parser then creates the parse tree of that source program. Otherwise, it will display error messages.

2.1.3 Semantic Analysis

Semantic Analysis is the third phase of compiler. The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation [3]. Semantic analysis makes sure that declarations and statements of a program are semantically correct. It is a collection of procedures which is called by parser, when required by grammar. Both syntax tree of the previous phase and symbol table are used to check the consistency of the given code. Type checking is an important part of semantic analysis where the compiler makes sure that each operator has matching operands. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

2.2 Ontology

An ontology defines a set of representational primitives in a particular knowledge area. [4]. It refers to the systemic representation of concepts, data and entities of a scenario and the relationships between them. It is a knowledge representation technique which can be used to represent a specific domain or across a wide range of domain. The term was introduced in computer science by artificial intelligence researchers who constructed computer models with some kind of automated reasoning [5].

2.2.1 Types of Ontology

Ontology can be catergorize into three types.

1. **Domain Ontology**: A domain specific ontology is the representation of a specific realm of world. For example: Bird ontology, Reptile ontology etc.

2. **Upper Ontology**: Upper ontology represents relation across a wide range of related domain ontologies. For example: Reptile ontology, Bird ontology, Insect ontology are the domain level ontologies of the upper Animal ontology.

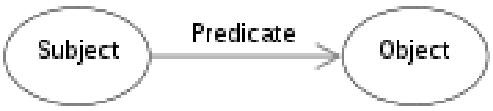


Figure 2.1: RDF-triples Example

3. **Hybrid Ontology**: Hybrid ontology consists of both domain ontology and upper ontology.

2.2.2 Components of Ontology

Most ontology consist of classes, individuals, relations and attributes. Common components of ontology include:

1. **Individual**: Instance or object of a class.
2. **Classes**: Sets, concepts, collections or kinds of things.
3. **Attributes**: Feature or characteristics of a class.
4. **Relations**: Ways in which individual and classes can be related.

2.3 RDF-Triples

An RDF triple is the atomic data entity in the Resource Description Framework (RDF) data model. As its name indicates, a triple is a set of three entities that codifies a statement about semantic data in the form of subject–predicate–object expressions. Ontology is used to represent the domain of programming languages. Then the parser parses source code or bytecode and serializes it into RDF triples. All the statements and expressions are RDF-ized in this process. Semantic techniques can be used then to query over these triples [6].

The components of a triple, such as a statement, consist of a subject, a predicate and an object. This is similar to the classical notation of an entity–attribute–value model within object-oriented design, where this would be expressed as an entity, an attribute and a value. From this basic structure, triples can be composed into more complex models, by using triples as objects or subjects of other triples. We can represent knowledge in a machine-readable way with this format and address every part of an RDF triple individually via unique URIs. The assertion of an RDF triple says that some relationship, indicated by the predicate, holds between the things denoted by subject and object of the triple. RDF uses URI references to identify resources and properties. Certain URI references are given specific meaning by RDF. Specifically, URI references with the following leading substring are defined by the RDF specifications:

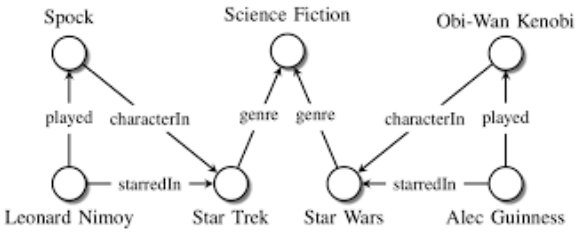


Figure 2.2: Knowledge Graph Example

http://www.w3.org/1999/02/22-rdf-syntax-ns# (conventionally associated with namespace prefix rdf:)

Used with the RDF/XML serialization, this URI prefix string corresponds to XML namespace names [XML-NS] associated with the RDF vocabulary terms.

2.4 Knowledge Graph

A knowledge graph is a structured representation of facts, consisting of entities, relationships, and semantic descriptions. Entities can be real-world objects and abstract concepts [7]. The characteristics of a Knowledge graph are:

- Descriptions have formal semantics and they can be processed in an efficient and unambiguous manner.
- Entity descriptions contribute to one another, forming a network, where each entity represents part of the description of the entities, related to it, and provides context for their interpretation.

A knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge. In other words, a knowledge graph is a programmatic way to model a knowledge domain with the help of subject-matter experts, data interlinking, and machine learning algorithms.

As we have discussed the theory related to our work, it's time to propose a framework to solve the problem.

Chapter 3

Proposed Framework

To solve the problem, we will use lexical analysis, syntax analysis and semantic analysis and finally generate knowledge graph. In this chapter, we discuss the tools and data structures we have used.

3.1 FLEX

FLEX is a tool for generating lexical analyzer. This is a tool for Linux operating system that can perform lexical analysis on the given C source code according to the regular expression rules provided.

There are three steps FLEX uses to generate tokens.

1. a file is created with $.l$ extension which is written in lex language. The lex compiler transforms it to C program, in a file that is always named $lex.yy.c$.
2. The C compiler compiles the $lex.yy.c$ file into an executable file called $a.out$. length.
3. The output file $a.out$ takes a source code file and produce a stream of tokens.

The lex program has 3 sections,

1. **Definition Section:** Enclosed using $\%\{\%\}$. This section contains variable declarations, import of header files, etc.
2. **Rules Section:** Enclosed using $\%\%Regexrules\%\%$. All the regular expressions are written here.
3. **User Code Section:** This section contain C statements and additional functions.

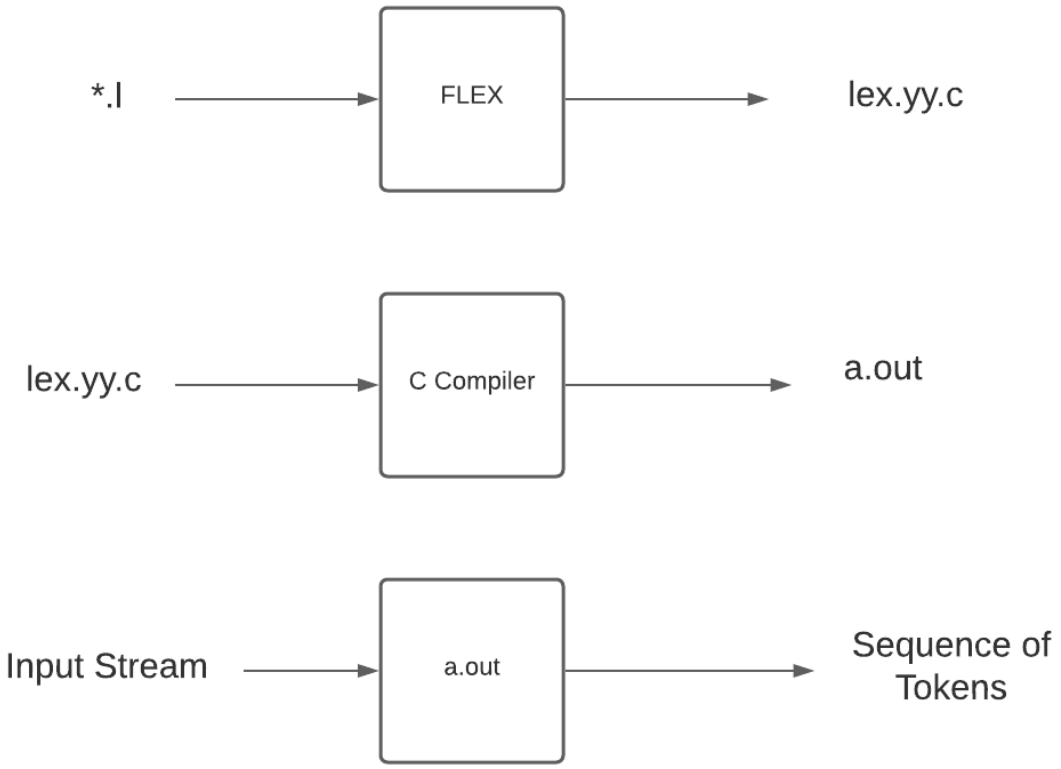


Figure 3.1: How FLEX Works

So, for a subset of C we write regular expressions and make tokens from the source code that can be used in the semantic analysis step.

3.2 YACC

Yacc (Yet Another Compiler-Compiler) is a computer program for the Unix operating system. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a LALR parser based on a formal grammar, written in a notation similar to Backus–Naur Form (BNF).

YACC input file is divided in three parts.

1. **Definition:** The definition part can include C code external to the definition of the parser and variable declarations, within $\%\{ \; and \; \%\}$.

2. **Rules:** The rules part contains grammar definition in a modified BNF form. For each grammar definition we write codes in C that checks for semantic errors.

3. **Auxiliary Routine:** The auxiliary routines part is only C code. It includes function definitions for every function needed in rules part. It can also contain the main() function

definition if the parser is going to be run as a program. The main() function must call the function yyparse().

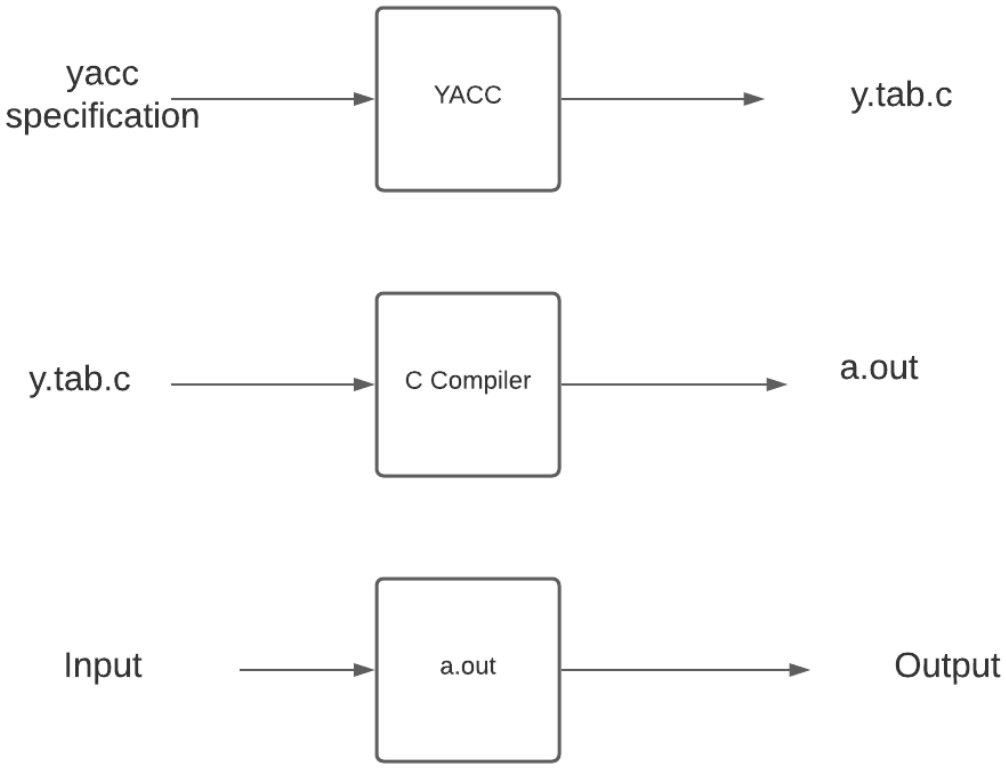


Figure 3.2: How YACC Works

After semantically analyzing the source code using YACC, we have used our own data structure to define the knowledge graph and RDF-triples from the source code.

3.3 Base Ontology

The base ontology contains base elements of the C source code like variables, functions, loops etc. They are the building block of of the ontology. It has been checked for satisfiability, incoherence and inconsistencies using the HermiT reasoner [8]. When the code is analyzed semantically, we assign the elements to the basic building blocks. The root is $Code\_Element$. This has three sub-classes,

- **Function:** Represents functions. This has the attributes like name, return type to hold the information about a function.
- **Type:** Type has two sub-classes,

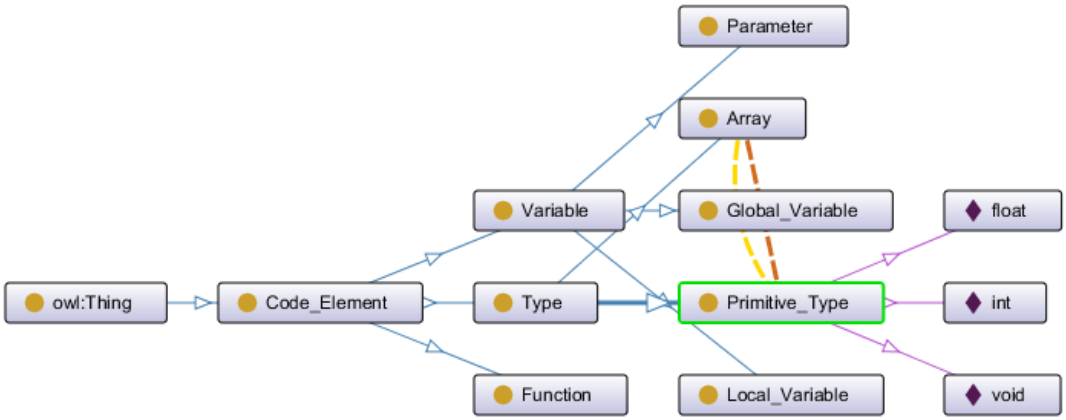


Figure 3.3: Base Ontology

  - **Array:** Represents all type of array.
  - **Primitive Type:** Represents the primitive types like int, $float$, etc.

- Variable: Variable has three types.
  - **Local Variable**
  - **Global Variable**
  - **Parameter**

There are some relationships between the classes,

- **hasArgument:** Relationship between a function and a variable.
- **hasReturnType:** Relationship between a function and types like int, $void$ etc.
- **hasVariableType:** Relationship between a variable and types like int, $float$ etc.
- **isArgument:** Relationship between a function and a variable. Inverse of hasArgument.
- **isReturnedBy:** Relationship between a function and a variable.
- **returns:** Relationship between a function and a variable. inverse of isReturnedBy.

3.4 Knowledge Graph Representation

In order to convert our source code into knowledge graph, we need to follow some kind of notation. In this research we have created a textual representation of knowledge graph. On each

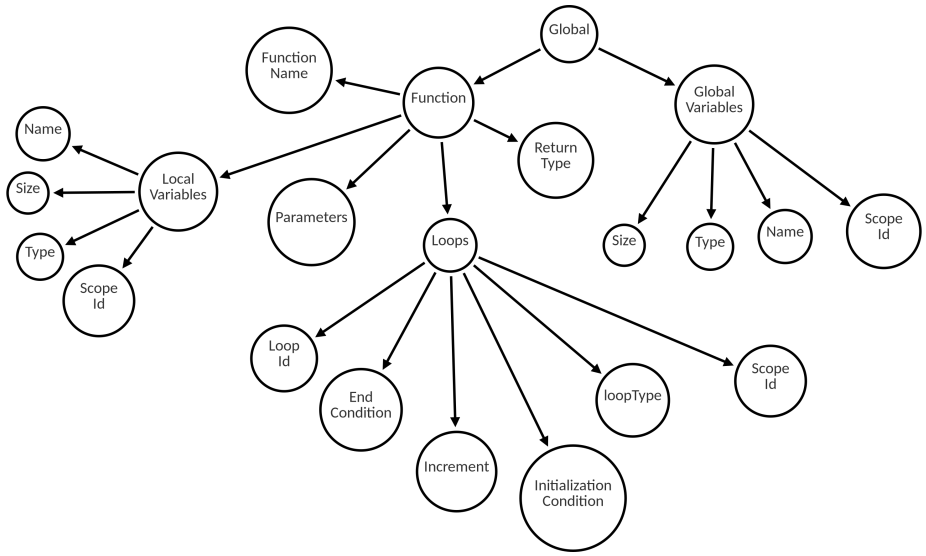


Figure 3.4: Knowledge Graph Representation of C source Code

line we have three attributes. The first and the third denotes vertices and the second denotes and edge. That is the first and the third are entities and the second one is the relationship between them. The knowledge graph is connected.

The Global Scope is considered as the root. This has relation with all the functions that are present in the code. Also all the global variables are related with this.

The functions are considered local. They have variables that are local and related to the functions. The loops are also considered as entities and has relation with the functions.

Each variable has name, type, size and scope id. The scope ids' are for denoting which scope the variable belongs to. The size is for array. Normal variables have size one.

Each function has a name, return type and, a list of parameters.

The loops has scope id that denotes which scope the loop belongs to. A scope may have more than one loops so to keep track of loop number, we have used loop id. The loop has type that tells us if this is a while loop or a for loop. Finally they have relation with three entities. Loop initiating condition, loop ending condition, and increment or decrement.

3.5 Data Structure Representation of the Knowledge Graph

In order to generate knowledge graph we need to keep track of the entities and relations between them. To do so while parsing, we use some simple data structures (Appendix A.1) defined by us. They are used inside the parser. When a grammar matches with a part of the source, We insert information about the entities using the data structures. Brief description of the data structures are as follows.

3.5.1 Structures

We have three structures. They are for keeping variables, functions and loops.

1. **variableDescription:** A variable has four attributes. Its name. The scope id is appended with the name to resolve ambiguity. Its data type, the scope it belongs to and a size. For array variables the size matters. Else it is one.
2. **functionDescription:** A function has name and return type. A list of parameters. The list contains the name of the variables. The variables are mapped to their name in a different data structure that will be discussed later. Also there is a list where we have kept the scopes from which this function has been called.
3. **loopDescription:** Loop has a scope id. A loop id to indicate a particular loop as a scope can obviously have more than one loop. It has a type that indicates if the loop is a *for loop* or a *while loop*. Finally it has three variables that denotes the initializing, ending and increment conditions.

3.5.2 Stores

We have some store classes that has been used to store the entities. Later after the parsing has been done, the stores are used to write the textual form of the knowledge graph.

1. **globalStore:** Stores the variable that are global and function names.
2. **variableStore:** Stores the variables using the structure for variable (Item 1). There is a map that has string and number as key, value pairs. That is each variable name is mapped to the index of the container where that variable is stored in the container so that we can update them in constant time.
3. **functionStore:** Stores the function information. It has a vector of the function description data structure (Item 2). A mapping of which function is in which index of the container to make sure efficient updates and lastly a map where the key, value pair are number and a container. The key are for scope id and the container to save the function calls.
4. **loopStore:** Stores the loops using the structure for variable (Item 3). There is a map that has string and number as key, value pairs. That is each loop name is mapped to the index of the container where that variable is stored in the container so that we can update them in constant time. There is also a map that tracks the count of loops in each scope and helps to assign unique loop ids.

Using these data structures we have generated knowledge graph and RDF-triples. In the next chapter we will show how our proposed framework performs.

Chapter 4

Experiments

We have experimented with a few codes to test our proposed framework. In this chapter we have shown our experimental setup and the outputs our framework gave after providing (Appendix A.3) as input. This has all types of variables, parameters and functions. Also more than one loop in the scopes, that covers more or less everything to demonstrate we have developed so far.

4.1 Experimental Setup

We have use a few tools for our experiment alongside our own code. These codes helped to compile our work.

- We have used **Ubuntu 18.04** as our operating system.
- For the lexical analyzer, flex has been used. The command *sudo apt-get install flex* installs the tool.
- For the semantic analyzer, YACC has been used. The command *sudo apt-get install bison flex* installs the tool that helps to compile our semantic analyzer code.
- Protege [1] has been used for creating the base ontology (Section 3.3). Also, **OntoGraf** that is integrated in Protege has been used to visualize the RDF-triples.
- This tool has been draw knowledge graph manually has been used.

```
<INT><ID,adjList><SEMICOLON,;><INT><ID,dfs><LPAREN,(><INT><ID,src><COMMA,,><FLOAT><ID,edgeCost><RPAREN,)><LCURL,{><INT><ID,val><
SEMICOLON,;><INT><ID,arr><LTHIRD,[><CONST_INT,118><RTHIRD,]><SEMICOLON,;><FOR><LPAREN,(><ID,val><ASSIGNOP,=><CONST_INT,5><SEMICO
LON,;><ID,val><RELOP,<><CONST_INT,7><SEMICOLON,;><ID,val><INCOP,++><RPAREN,)><LCURL,{><RCURL,}><RETURN><ID,val><SEMICOLON,;><RCU
RL,}><INT><ID,union_find><LPAREN,(><RPAREN,)><LCURL,{><INT><ID,k><COMMA,,><ID,cmp><SEMICOLON,;><ID,cmp><ASSIGNOP,=><CONST_INT,10
><SEMICOLON,;><FOR><LPAREN,(><ID,k><ASSIGNOP,=><CONST_INT,0><SEMICOLON,;><ID,k><RELOP,<><ID,cmp><SEMICOLON,;><ID,k><INCOP,++><RP
AREN,)><LCURL,{><RCURL,}><FOR><LPAREN,(><ID,k><ASSIGNOP,=><CONST_INT,4><SEMICOLON,;><ID,k><RELOP,>=><CONST_INT,0><SEMICOLON,;><I
D,k><INCOP,--><RPAREN,)><LCURL,{><RCURL,}><ID,dfs><LPAREN,(><CONST_INT,2><COMMA,,><CONST_FLOAT,2.7><RPAREN,)><SEMICOLON,;><RETUR
N><ID,cmp><MULOP,*><CONST_INT,10><SEMICOLON,;><RCURL,}><INT><ID,main><LPAREN,(><RPAREN,)><LCURL,{><INT><ID,x><COMMA,,><ID,y><SEM
ICOLON,;><INT><ID,i><COMMA,,><ID,j><COMMA,,><ID,k><SEMICOLON,;><ID,union_find><LPAREN,(><RPAREN,)><SEMICOLON,;><ID,dfs><LPAREN,(
><CONST_INT,5><COMMA,,><CONST_FLOAT,6.6><RPAREN,)><SEMICOLON,;><FOR><LPAREN,(><ID,i><ASSIGNOP,=><CONST_INT,0><SEMICOLON,;><ID,i>
<RELOP,<><CONST_INT,10><SEMICOLON,;><ID,i><INCOP,++><RPAREN,)><LCURL,{><INT><ID,v><SEMICOLON,;><IF><LPAREN,(><ID,i><MULOP,%><CON
ST_INT,2><RELOP,==><CONST_INT,0><RPAREN,)><LCURL,{><ID,y><ASSIGNOP,=><CONST_INT,5><SEMICOLON,;><RCURL,}><ELSE><LCURL,{><ID,y><AS
SIGNOP,=><CONST_INT,10><SEMICOLON,;><RCURL,}><RCURL,}><ID,i><ASSIGNOP,=><CONST_INT,10><SEMICOLON,;><WHILE><LPAREN,(><ID,i><RELOP
,<=><CONST_INT,10><RPAREN,)><LCURL,{><ID,i><INCOP,--><SEMICOLON,;><RCURL,}><ID,i><ASSIGNOP,=><CONST_INT,0><SEMICOLON,;><WHILE><L
PAREN,(><ID,i><RELOP,!=><CONST_INT,10><RPAREN,)><LCURL,{><ID,i><INCOP,++><SEMICOLON,;><RCURL,}><FOR><LPAREN,(><ID,i><ASSIGNOP,=>
<CONST_INT,0><SEMICOLON,;><ID,i><RELOP,<><CONST_INT,10><SEMICOLON,;><ID,i><INCOP,++><RPAREN,)><LCURL,{><RCURL,}><FOR><LPAREN,(><
ID,i><ASSIGNOP,=><CONST_INT,10><SEMICOLON,;><ID,i><RELOP,>><CONST_INT,0><SEMICOLON,;><ID,i><INCOP,--><RPAREN,)><LCURL,{><RCURL,}
><FOR><LPAREN,(><ID,i><ASSIGNOP,=><CONST_INT,0><SEMICOLON,;><ID,i><RELOP,<><CONST_INT,10><SEMICOLON,;><ID,i><ASSIGNOP,=><ID,i><A
DDOP,+><CONST_INT,2><RPAREN,)><LCURL,{><RCURL,}><FOR><LPAREN,(><ID,i><ASSIGNOP,=><CONST_INT,10><SEMICOLON,;><ID,i><RELOP,>><CONS
T_INT,0><SEMICOLON,;><ID,i><ASSIGNOP,=><ID,i><ADDOP,-><CONST_INT,2><RPAREN,)><LCURL,{><RCURL,}><RETURN><CONST_INT,0><SEM
ICOLON,;><RCURL,}>
```

Figure 4.1: Tokens Generated

4.2 Experiment Output

At first the code passes through the lexical analyzer where both syntax analysis and lexical analysis occurs. After this, a set of tokens are generated.

Then code is parsed and semantic analysis occurs. The parser looks for errors according to the grammars (Appendix A.2) provided.

While parsing we can save information about the source code. We can save them using the data structures we have defined. So we save the data and when it's time to exit we write out the RDF-triples in an owl file and the knowledge graph in a text file.

For the RDF-triple, we have used the base ontology (Section 3.3). We created it Protege [1] and kept it in the root directory of our project. We read the whole file in a string and append new information as we go through. Finally, when we are done parsing, we write it in an owl file. If we use **OntoGraf** to visualize it then we will get the following figure.

Simultaneously we generate the knowledge graph. It's in the textual representation, so we present a figure to visualize it. For better quality of the diagram, follow this link

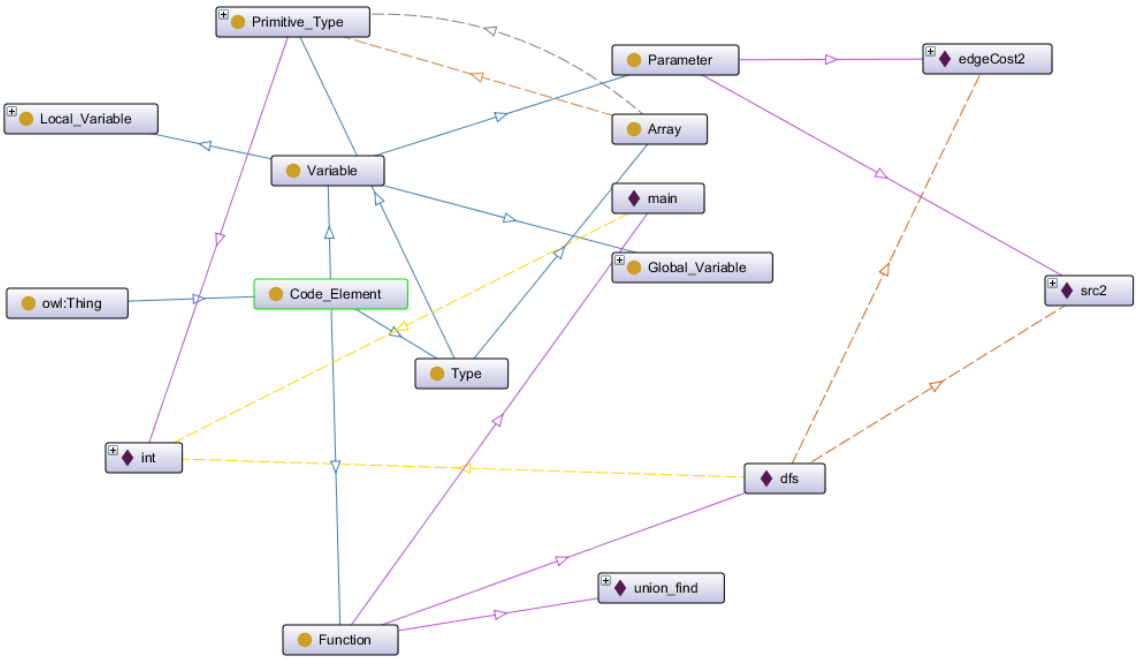


Figure 4.2: Generated RDF-triples

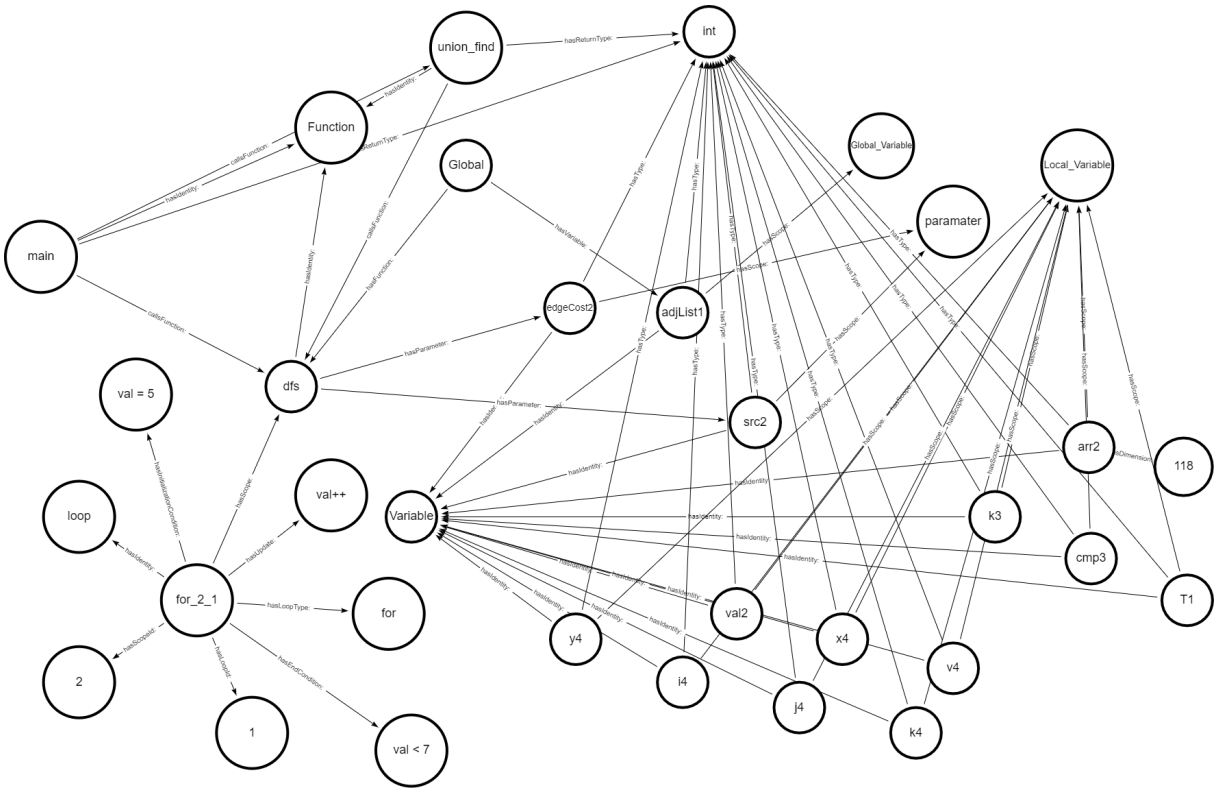


Figure 4.3: Generated Knowledge Graph

Chapter 5

Discussion

From our experiments we can see that, as our code gets bigger, the size of the knowledge graph and ontograf also gets bigger that causes a problem to graphicallly visualize them. But at the same the textual form will work fine if used as input for any further analysis programmatically. There is a scope for improvement so that those structure reflect the whole code in a better way. So, it is quite messy to get the idea from those structure. We have generated knowledge graph and RDF-triples only for a subset of C. So, this experiment can be extended for other entity of the C language. Also this experiment can be extended for other programming language like, C++, Java too.

References

[1] M. A. Musen, "The protégé project: a look back and a look forward," *AI matters*, vol. 1, no. 4, pp. 4–12, 2015.

[2] M. Atzeni and M. Atzori, "Codeontology: Rdf-ization of source code," in *International Semantic Web Conference*, pp. 20–28, Springer, 2017.

[3] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[4] P. Mika and H. Akkermans, "Towards a new synthesis of ontology technology and knowledge management," *Knowledge Engineering Review*, vol. 19, no. 4, pp. 317–378, 2004.

[5] D. Dermeval, J. Vilela, I. I. Bittencourt, J. Castro, S. Isotani, P. Brito, and A. Silva, "Applications of ontologies in requirements engineering: a systematic review of the literature," *Requirements Engineering*, vol. 21, no. 4, pp. 405–437, 2016.

[6] G. Klyne, "Resource description framework (rdf): Concepts and abstract syntax," *http://www. w3. org/TR/2004/REC-rdf-concepts-20040210/*, 2004.

[7] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu, "A survey on knowledge graphs: Representation, acquisition and applications," *arXiv preprint arXiv:2002.00388*, 2020.

[8] R. Shearer, B. Motik, and I. Horrocks, "Hermit: A highly-efficient owl reasoner.," in *Owled*, vol. 432, p. 91, 2008.

Appendix A

Codes

A.1 Utility Data Structure to Define and Store Knowledge Graph

```
#include<string>
#include<vector>
#include<map>

using namespace std;

string toString(int n)
{
        string temp;

        if(!n){
                return "0";
        }

        while(n){
                int r=n%10;
                n/=10;
                temp.push_back(r+48);
        }

        reverse(temp.begin(),temp.end());
        return temp;
```

```
23 }
24
25 class globalStore {
26     // relationship + invidiual name
27     vector<string> globals;
28 public:
29     void add(string s) {
30         globals.push_back(s);
31     }
32
33     string globalKnowledgeGraph() {
34         string ret = "";
35         for(string s : globals) {
36             ret += "Global␣" + s + "\n";
37         }
38
39         return ret;
40     }
41 };
42
43 struct variableDescription{
44     string var_name, var_type, var_scope;
45     int arraySz;
46
47     variableDescription(){}
48     variableDescription(string var_name, string var_type,
       ↪ string var_scope, int arraySz) {
49         this->var_name = var_name;
50         this->var_type = var_type;
51         this->var_scope = var_scope;
52         this->arraySz = arraySz;
53     }
54 };
55
56 class variableStore
57 {
58 public:
59     vector<variableDescription> variables;
60     map<string, int> mapping;
```

```
61
62      variableStore(){}
63
64      void addVariable(string var_name, string var_type, string
           ↪ var_scope, int arraySz) {
65          this->variables.push_back(variableDescription(var_name,
               ↪  var_type, var_scope, arraySz));
66          this->mapping[var_name] = variables.size() - 1;
67      }
68
69      void makeParameter(string var_name) {
70          this->variables[this->mapping[var_name]].var_scope = "
               ↪ Parameter";
71      }
72
73      string variableRDF()
74      {
75          string ret = "";
76          for(int i = 0; i < this->variables.size(); i++) {
77              ret += "\t<owl:NamedIndividual␣rdf:about=\"http://
                   ↪ www.semanticweb.org/acer/ontologies/2020/10/
                   ↪ Onto-C#" + this->variables[i].var_name + "
                   ↪ \">\n";
78              ret += "\t\t<rdf:type␣rdf:resource=\"http://www.
                   ↪ semanticweb.org/acer/ontologies/2020/10/Onto-
                   ↪ C#" + this->variables[i].var_scope + "\"/>\n"
                   ↪ ;
79              ret += "\t\t<rdf:type>\n";
80              ret += "\t\t\t<owl:Restriction>\n";
81              ret += "\t\t\t\t<owl:onProperty␣rdf:resource=\"http
                   ↪ ://www.semanticweb.org/acer/ontologies
                   ↪ /2020/10/Onto-C#hasVariableType\"/>\n";
82              ret += "\t\t\t\t<owl:allValuesFrom␣rdf:resource=\"
                   ↪ http://www.semanticweb.org/acer/ontologies
                   ↪ /2020/10/Onto-C#" + this->variables[i].
                   ↪ var_scope + "\"/>\n";
83              ret += "\t\t\t</owl:Restriction>\n";
84              ret += "\t\t</rdf:type>\n";
85              ret += "\t\t<hasType␣rdf:resource=\"http://www.
```

```
                ↪ semanticweb.org/acer/ontologies/2020/10/Onto-
                ↪ C#" + this->variables[i].var_type + "\"/>\n";
86
87              if(this->variables[i].arraySz)
88                  ret += "\t\t<Dimension␣rdf:datatype=\"http://
                        ↪ www.w3.org/2001/XMLSchema#integer\">" +
                        ↪ toString(this->variables[i].arraySz) + "
                        ↪ </Dimension>\n";
89
90              ret += "\t\t<Name␣rdf:datatype=\"http://www.w3.org
                    ↪ /2001/XMLSchema#string\">" + this->variables[
                    ↪ i].var_name + "</Name>\n";
91              ret += "\t</owl:NamedIndividual>\n\n";
92          }
93
94          return ret;
95      }
96
97      string variableKnowledgeGraph()
98      {
99          string ret = "";
100         for(int i = 0; i < this->variables.size(); i++) {
101             string name = this->variables[i].var_name;
102             ret += name + "␣hasIdentity␣Variable\n";
103             ret += name + "␣hasScope␣" +  this->variables[i].
                    ↪ var_scope + "\n";
104             ret += name + "␣hasType␣" + this->variables[i].
                    ↪ var_type + "\n";
105
106             if(this->variables[i].arraySz)
107                 ret += name + "␣hasDimension␣" + toString(this
                        ↪ ->variables[i].arraySz) + "\n";
108
109             ret += "\n";
110         }
111
112         return ret;
113     }
114 };
```

```
115
116 struct functionDescription{
117     string function_name, return_type;
118     vector<string> parameters;
119     vector<string> functionCallings;
120
121     functionDescription(){}
122     functionDescription(string function_name, string
        ↪ return_type) {
123         this->function_name = function_name;
124         this->return_type = return_type;
125     }
126 };
127
128 class functionStore
129 {
130 public:
131     vector<functionDescription> functions;
132     map<string, int> mapping;
133     map<int, vector<string>> functionCallings;
134
135     functionStore(){}
136
137     void addFunction(string function_name, string return_type)
        ↪ {
138         this->functions.push_back(functionDescription(
            ↪ function_name, return_type));
139         this->mapping[function_name] = functions.size() - 1;
140     }
141
142     void addParameter(string function_name, string param) {
143         this->functions[this->mapping[function_name]].
            ↪ parameters.push_back(param);
144     }
145
146     void addFunctionCall(int function_scope_id, string calling)
        ↪  {
147         this->functionCallings[function_scope_id].push_back(
            ↪ calling);
```

```
148     }
149
150     void assignFunctionNames(map<int, string> scopeMapping) {
151         for(auto itr = this->functionCallings.begin(); itr !=
              ↪ this->functionCallings.end(); itr++) {
152             int id = itr->first;
153             string name = scopeMapping[id];
154             for(string s : this->functionCallings[id]) {
155                 this->functions[this->mapping[name]].
                      ↪ functionCallings.push_back(s);
156             }
157         }
158
159     }
160
161     string functionRDF()
162     {
163         string ret = "";
164         for(int i = 0; i < this->functions.size(); i++) {
165             ret += "\t<owl:NamedIndividual␣rdf:about=\"http://
                  ↪ www.semanticweb.org/acer/ontologies/2020/10/
                  ↪ Onto-C#" + this->functions[i].function_name +
                  ↪  "\">\n";
166             ret += "\t\t<rdf:type␣rdf:resource=\"http://www.
                  ↪ semanticweb.org/acer/ontologies/2020/10/Onto-
                  ↪ C#Function\"/>\n";
167
168                 for(string s : this->functions[i].parameters)
169                 ret += "\t\t<hasArgument␣rdf:resource=\"http://
                      ↪ www.semanticweb.org/acer/ontologies
                      ↪ /2020/10/Onto-C#" + s + "\"/>\n";
170
171             ret += "\t\t<hasReturnType␣rdf:resource=\"http://
                  ↪ www.semanticweb.org/acer/ontologies/2020/10/
                  ↪ Onto-C#" + this->functions[i].return_type + "
                  ↪ \"/>\n";
172             ret += "\t</owl:NamedIndividual>\n";
173         }
174
```

```
175         return ret;
176     }
177
178     string functionKnowledgeGraph()
179     {
180         string ret = "";
181         for(int i = 0; i < this->functions.size(); i++) {
182             string name = this->functions[i].function_name;
183             ret += name + "␣hasIdentity␣Function\n";
184             ret += name + "␣hasReturnType␣" + this->functions[i
                ↪ ].return_type + "␣\n";
185
186                 for(string s : this->functions[i].parameters)
187                 ret += name + "␣hasParameter␣" + s + "\n";
188
189             for(string s : this->functions[i].functionCallings)
190                 ret += name + "␣callsFunction␣" + s + "\n";
191
192             ret += "\n";
193         }
194
195         return ret;
196     }
197 };
198
199 struct loopDescription {
200     int scopeId, loopId;
201     string endCondition = "", initializationCondition = "",
        ↪ inc_dec = "";
202     string scope, type; // type is For or While
203
204     loopDescription(){}
205     loopDescription(int loopId, int scopeId, string type) {
206         this->loopId = loopId;
207         this->scopeId = scopeId;
208         this->type = type;
209     }
210 };
211
```

```
212 class loopStore
213 {
214 public:
215     vector<loopDescription> loops;
216     map<string, int> mapping;
217     map<int, int> loopNumbering;
218 
219     loopStore(){}
220 
221     // returns the loops unique identity
222     string addLoop(int scopeId, string type) {
223         int loopId = loopNumbering[scopeId] + 1;
224         loopNumbering[scopeId] = loopId;
225 
226         loops.push_back(loopDescription(loopId, scopeId, type))
              ↪ ;
227 
228         string id = toString(scopeId) + "_" + toString(loopId);
229         mapping[id] = loops.size() - 1;
230 
231         return id;
232     }
233 
234     void addScopeNames(map<int, string> scopeMap) {
235         for(int i = 0; i < this->loops.size(); i++) {
236             this->loops[i].scope = scopeMap[this->loops[i].
                  ↪ scopeId];
237         }
238     }
239 
240     void addEndCondition(string loopName, string condition) {
241         this->loops[this->mapping[loopName]].endCondition =
              ↪ condition;
242     }
243 
244     void addInitialization(string loopName, string init) {
245         this->loops[this->mapping[loopName]].
              ↪ initializationCondition = init;
246     }
```

```
247
248     void addIncDec(string loopName, string val) {
249         this->loops[this->mapping[loopName]].inc_dec = val;
250     }
251
252     string loopKnowledgeGraph() {
253         string ret = "";
254         for(int i = 0; i < this->loops.size(); i++) {
255             string id = toString(this->loops[i].scopeId) + "_"
                ↪ + toString(this->loops[i].loopId);
256             string name = this->loops[i].type + "_" + id;
257
258             ret += name + "␣hasScopeId␣" + toString(this->loops
                ↪ [i].scopeId) + "\n";
259             ret += name + "␣hasLoopId␣" + toString(this->loops[
                ↪ i].loopId) + "\n";
260             ret += name + "␣hasIdentity␣Loop\n";
261             ret += name + "␣hasScope␣" + this->loops[i].scope +
                ↪  "\n";
262             ret += name + "␣hasLoopType␣" + this->loops[i].type
                ↪  + "\n";
263
264             if(this->loops[i].initializationCondition.length())
265                 ret += name + "␣hasInitializationCondition␣\""
                    ↪ + this->loops[i].initializationCondition
                    ↪ + "\"\n";
266
267             if(this->loops[i].endCondition.length())
268                 ret += name + "␣hasEndCondition␣\"" + this->
                    ↪ loops[i].endCondition + "\"\n";
269
270             if(this->loops[i].inc_dec.length())
271                 ret += name + "␣hasUpdate␣\"" + this->loops[i].
                    ↪ inc_dec + "\"\n";
272
273             ret += "\n";
274         }
275
276         return ret;
```

```
277         }
278 };
```

A.2 Grammar for the Parser

```
%{
#include<iostream>
#include<cstdlib>
#include<cstring>
#include<cmath>
#include "symbol.h"
#define YYSTYPE SymbolInfo*

using namespace std;

int yyparse(void);
int yylex(void);
extern FILE *yyin;

SymbolTable *table;

// what to do when error occurs
void yyerror(char *s){}


%}

%token IF ELSE FOR WHILE

%left
%right

%nonassoc


%%

start : program
        {

        }
        ;
```

```
38
39 program : program unit
40         | unit
41         ;
42
43 unit : var_declaration
44      | func_declaration
45      | func_definition
46      ;
47
48 func_declaration : type_specifier ID LPAREN parameter_list
    ↪ RPAREN SEMICOLON
49                 | type_specifier ID LPAREN RPAREN SEMICOLON
50                 ;
51
52 func_definition : type_specifier ID LPAREN parameter_list
    ↪ RPAREN compound_statement
53                 | type_specifier ID LPAREN RPAREN
                      ↪ compound_statement
54                 ;
55
56
57 parameter_list  : parameter_list COMMA type_specifier ID
58                 | parameter_list COMMA type_specifier
59                 | type_specifier ID
60                 | type_specifier
61                 ;
62
63
64 compound_statement : LCURL statements RCURL
65                    | LCURL RCURL
66                    ;
67
68 var_declaration : type_specifier declaration_list SEMICOLON
69                  ;
70
71 type_specifier  : INT
72                 | FLOAT
73                 | VOID
```

```
74                  ;
75
76 declaration_list : declaration_list COMMA ID
77                   | declaration_list COMMA ID LTHIRD CONST_INT
                        ↪ RTHIRD
78                   | ID
79                   | ID LTHIRD CONST_INT RTHIRD
80                   ;
81
82 statements : statement
83            | statements statement
84            ;
85
86 statement : var_declaration
87           | expression_statement
88           | compound_statement
89           | FOR LPAREN expression_statement
               ↪ expression_statement expression RPAREN
               ↪ statement
90           | IF LPAREN expression RPAREN statement
91           | IF LPAREN expression RPAREN statement ELSE
               ↪ statement
92           | WHILE LPAREN expression RPAREN statement
93           | PRINTLN LPAREN ID RPAREN SEMICOLON
94           | RETURN expression SEMICOLON
95           ;
96
97 expression_statement    : SEMICOLON
98                         | expression SEMICOLON
99                         ;
100
101 variable : ID
102          | ID LTHIRD expression RTHIRD
103          ;
104
105  expression : logic_expression
106            | variable ASSIGNOP logic_expression
107            ;
108
```

```
109 logic_expression : rel_expression
110                  | rel_expression LOGICOP rel_expression
                        ↪
111                  ;
112
113 rel_expression  : simple_expression
114                  | simple_expression RELOP simple_expression
                        ↪
115                  ;
116
117 simple_expression : term
118                   | simple_expression ADDOP term
119                   ;
120
121 term :  unary_expression
122      |  term MULOP unary_expression
123      ;
124
125 unary_expression : ADDOP unary_expression
126                  | NOT unary_expression
127                  | factor
128                  ;
129
130 factor  : variable
131         | ID LPAREN argument_list RPAREN
132         | LPAREN expression RPAREN
133         | CONST_INT
134         | CONST_FLOAT
135         | variable INCOP
136         | variable DECOP
137         ;
138
139 argument_list : arguments
140                           |
141                           ;
142
143 arguments : arguments COMMA logic_expression
144               | logic_expression
145               ;
```

```
146
147
148 %%
149 int main(int argc,char *argv[])
150 {
151
152         if((fp=fopen(argv[1],"r"))==NULL)
153         {
154                 printf("Cannot␣Open␣Input␣File.\n");
155                 exit(1);
156         }
157
158         fp2= fopen(argv[2],"w");
159         fclose(fp2);
160         fp3= fopen(argv[3],"w");
161         fclose(fp3);
162
163         fp2= fopen(argv[2],"a");
164         fp3= fopen(argv[3],"a");
165
166
167         yyin=fp;
168         yyparse();
169
170
171         fclose(fp2);
172         fclose(fp3);
173
174         return 0;
175 }
```

A.3 Code on Which We Experimented

```
int adjList;
int dfs(int src, float edgeCost){
    int val;
    int arr[118];
    for(val = 5; val < 7; val++){}

    return val;
}

int union_find()
{
    int k, cmp;
    cmp = 10;
    for(k = 0; k < cmp; k++){}
    for(k = 4; k >= 0; k--){}

    dfs(2, 2.7);

    return cmp * 10;
}

int main()
{
    int x, y;
    int i, j, k;

    union_find();
    dfs(5, 6.6);


    for(i = 0; i < 10; i++) {
        int v;
        if(i % 2 == 0){
            y = 5;
        }

        else{
```

```
38              y = 10;
39          }
40      }
41
42      i = 10;
43      while(i <= 10){
44          i--;
45      }
46
47      i = 0;
48      while(i != 10) {
49          i++;
50      }
51
52      for(i = 0; i < 10; i++){}
53      for(i = 10; i > 0; i--){}
54      for(i = 0; i < 10; i = i + 2){}
55      for(i = 10; i > 0; i = i - 2){}
56
57      return 0;
58  }
```

Generated using Undegraduate Thesis LaTeX Template, Version 1.4. Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.

This thesis was generated on Friday 5th March, 2021 at 4:34pm.