Journal of Software Engineering and Applications, 2020, 13, 77-90
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Intermediate Representation Using Graph Visualization Software

E. O. Aliyu[1], A. O. Adetunmbi[2], B. A. Ojokoh[3]

[1]Department of Computer Science, Adekunle Ajasin University, Akungba-Akoko, Nigeria
[2]Department of Computer Science, Federal University of Technology, Akure, Nigeria
[3]Department of Information Systems, Federal University of Technology, Akure, Nigeria
Email: aliyu.olubunmi@gmail.com, aoadetunmbi@futa.edu.ng, bolanleojokoh@yahoo.com

How to cite this paper: Aliyu, E.O., Adetunmbi, A.O. and Ojokoh, B.A. (2020) Intermediate Representation Using Graph Visualization Software. *Journal of Software Engineering and Applications*, **13**, 77-90.
https://doi.org/10.4236/jsea.2020.135006

Received: April 2, 2020
Accepted: May 6, 2020
Published: May 9, 2020

Abstract

In this paper, a method to initiate, develop and visualize an abstract syntax tree (AST) in C++ source code is presented. The approach is in chronological order starting with collection of program codes as a string and split into individual characters using regular expression. This will be followed by separating the token grammar using best first search (BFS) algorithm to determine node having lowest value, lastly followed by graph presentation of intermediate representation achieved with the help of graph visualization software (GraphViz) while former is implemented using python programming language version 3. The efficacy of our approach is used in analyzing C++ code and yielded a satisfactory result.

Keywords

Recursive Descent Parser, Best First Search, Intermediate Representation, Abstract Syntax Tree, Graph Visualization Software

1. Introduction

Intermediate representations (IR) do not exist in a vacuum. They are the stepping stone from what the programmer wrote to what the machine understands [1]. Drawings of compiler data structures such as syntax trees, control flow graphs, dependency graphs [2] are used for demonstration, debugging and documentation of compilers. For instance, [3] demonstrated using **Figure 1** the importance of intermediate representation in terms of portability and modularity.

That is, compiler for five languages and four target machines (left) without an IR caused front-end becomes cluttered with machine specific details and

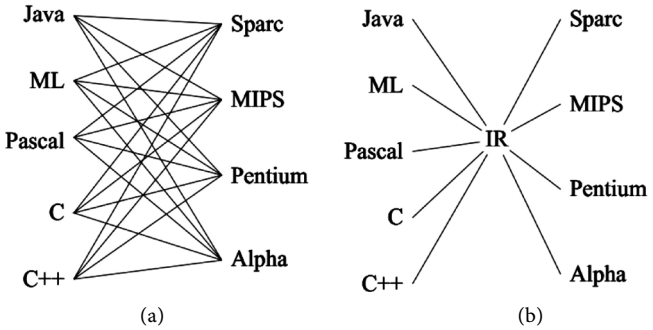

Figure 1. Compiler implementation in machine language. Source: Walker, 2003.

back-end becomes cluttered with source language specific details (that is, need separate compiler for each source language/target machine combination) while (right) with an IR need just "n" front-ends and "m" back-ends (meaning that, one can build a new front-end for an existing backend). In real-world compiler applications, such drawings cannot be produced manually because the graphs are automatically generated using graph layout algorithms such as JGraphX, JGraphT, JUNG, Prefuse and GraphViz [4]. In the area of debugging, compiler data structure has been used to overcome the problems faced by model checking in constructing a model for the system under consideration [5]. [6] stated that the main goal for the design of an intermediate representation is to support the optimizations using it. Therefore, this work is aimed at developing intermediate representation (that is, abstract syntax tree) at the level of parsing using graph visualization software. Abstract syntax trees (AST) can be used in program analysis and program transformation systems. [7] stated that AST has been known to provide a representation of programs suitable for flow-insensitive analysis such as type analysis, control flow analysis and pointer analysis because they ignore execution order of statements in a function or block. Program transformation on the other hand, is an automatic manipulation of source program [8]. It has been found useful in different applications including compiler construction, optimization, program synthesis, refactoring, software renovation and reverse engineering. Visualization tools provide a graphical description of a program [9]. Tools such as Dot draw directed graphs in a graphics format including Graphics Interchange Format (GIF), Portable Network Graphics (PNG), Scalable Vector Graphic (SVG), Portable Document Format (PDF) or PostScript [10]. However, when a program is parsed in C of Languages (CLANG), the result is either in a dump file or PDF.

This paper discusses how the grammar rule presented in [11] has been transformed into an intermediate representation (IR). Also, a parser using recursive descent parsing technique and best first search algorithm to generate an intermediate representation called abstract syntax tree (AST) with the help of graph visualization software to visualize the output of syntax analysis will be presented. [12] pointed out that visualization can be a useful teaching aid to examine the generated AST and gain a greater understanding of the underlying program. The

contribution of this paper is as follows:

1) initiate a parser using recursive descent and best first search algorithm for generating abstract syntax trees dot file based on defined grammar rules,

2) produce abstract syntax tree using graph visualization software for interpreting the dot file into portable network graphics format.

The next section will introduce review of related works in this field of study. In section three, background of intermediate representation, graph visualization software, recursive descent parser and best first search algorithm, parsing process methodology in section four while section five highlights the conclusion and future research work.

2. Related Works

Stalmans [12] presented visualization of abstract syntax trees for COCO/R. The idea is to help programmers with an integrated development environment (IDE) for COCO/R to understand the compiler generation process and visualize the output of syntax analysis.

A tool to support compiler process via java-based compiler-compiler in an interactive environment (JACCIE) was developed and presented by [13]. The idea is to aid automatic generation of compiler components in a visual debugging environment, displaying compiler components internal states which are hidden from users in conventional compilers.

To identify significant static analysis functionality provided in python program analyzer, [14] developed verification of program properties that automatically presents program flow and call graph using Graphviz. This visualization provides useful information to user extracting data such as macro definitions, variables, type definitions and function signatures from header files.

[15] introduced LLVM-based JIT compilation in genetic programming. Their intention was to improve the computational efficiency of genetic programming in understanding how just in time compilation (JIT) expressed in abstract syntax tree can be accomplished using the lower level virtual machine (LLVM) library.

A tool to secure Internet of Things (IoT) app or environment via static analysis system (SOTERIA) was developed and presented by [16]. The idea is to validate IoT app or environment for safety, security and functional properties by translating IoT source code into an intermediate representation using sensor-computation-actuator program structures.

However, in all the paper review, known of the paper clearly shows the process to explore the grammar structure as we have shown in this paper. The goal is to aid understanding of the underlying program in detecting assignment-in guard error including non-inclusion of brake and default in selective and iterative structures C++ codes.

3. Central Concepts of Intermediate Representation

According to [17] intermediate languages (ILs) are typically used in compiler

and compiler like applications (that is, static checkers). They are usually tree-like data types which represent some abstract syntax of a simple language. However, ILs generation system has been implemented in a number of tools such as java compiler compiler (JavaCC), another tool for language recognition (ANTLR), yet another compiler compiler (YACC), visible compiler compiler (VCOCO) respectively [12]. Although, there are several ways to generate an abstract syntax tree directly from the grammar, this paper discusses some of the techniques used in constructing an abstract syntax tree.

3.1. Abstract Syntax Tree

Quantitatively, an abstract syntax tree (AST) is a condensed version of parse trees. In the context of a compiler, the term AST is used interchangeably with syntax tree [18]. An abstract syntax tree on the other hand, ignores a significant amount of the syntactic information that a parse tree; which is a pictorial version of the grammatical structure of a sentence, would contain. **Figure 2** demonstrates graphically the parse tree and AST using arithmetic expression 2 + (2 + 2) relative to the grammar "expr = factor ((plus|minus) factor)) + while factor = int|(expr).

Each node of the tree denotes a construct occurring in the source code. [19] described formal notion of an abstract syntax tree (AST) as a labeled graph (N, E, μ_N) over the alphabet Σ_N; is a finite and directed graph, where N is a set of nodes, $E \subseteq N \times N$ is an edge relation between the nodes, and $\mu_N : N \to \Sigma_N$ is a node labeling function. A labeled tree is a labeled graph $T = (N, E, \mu_N)$ if it has a single root node root(T) for which we have the following: for each node $l \in N$ there exists exactly one path from the root to the node, that is, exactly one sequence $l_0, \cdots, l_n$, such that $l_0 = \text{root}(T)$, $l_n = l$ and $(l_{i-1}, l_i) \in E$ for $i = 1, \cdots, n$.

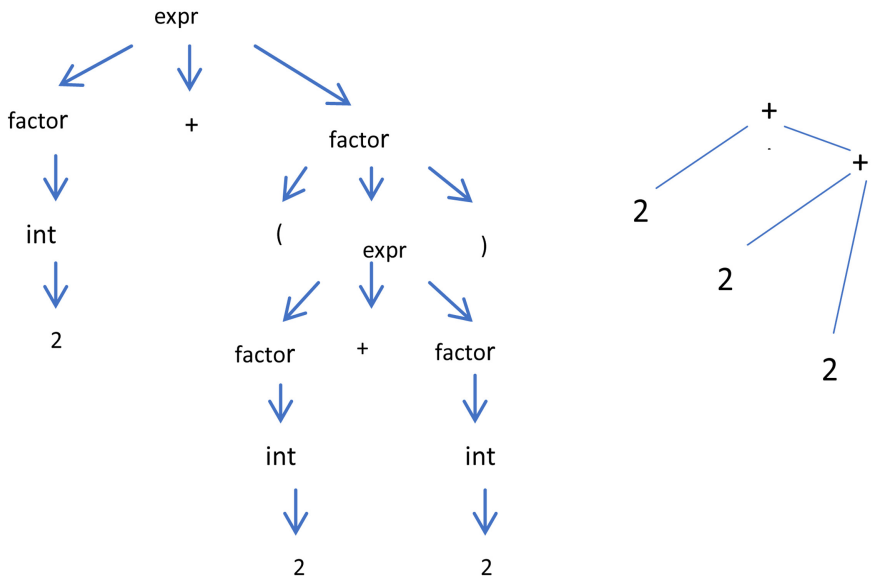

Figure 2. Parse tree format (left) and abstract syntax tree (right).

However, the formal notion can be exemplified by the anatomy of an AST as presented by [18].

By considering a single node within an AST, **Figure 3** shows that every node contains the terminal (that is, the token value) and pointers to its next sibling as well as its first child node.

3.2. Graph Visualization Software

Graphviz is an acronyms for Graph Visualization Software; a collection of libraries and utilities to manipulate and view graphs in a variety of output format such as simple text format (plain or plain-text), portable document format (PDF), jpeg, scalable vector graphic (SVG), portable network graphics (PNG) respectively [20] [21]. It can be used in a variety of contexts, such as software engineering, bio-informatics, internet and web structures and dynamic distributed communication services. [20] [21] [22] [23] stated that the algorithms of Graphviz concentrate on graph layout algorithm such as dot; a sugiyama-style hierarchical layout, neato; a symmetric layout algorithm based on stress reduction, osage; a layout algorithm for clustered graphs based on user specifications respectively, depending on the application type and the data being visualised. In this paper, Graphviz software was employed as a library, in particular, the dot algorithm, which produces a ranked layout of a graph, honoring the direction of the edges [21] [23]. The steps in dot layout comprises the following: initialize; initialization establish the data structures specific to the given algorithm, rank; after initialization, algorithm assigns each node to a discrete rank using an integer program to minimize the sum of the discrete edge lengths, mincross; rearranges nodes within ranks to reduce edge crossings, position; is the assignment of actual coordinates to the nodes, sameports is based on the edge attributes, by which certain edges sharing a node and all connect to the node at the same point, splines; edge representations are generated in this step while compoundEdges clipped to the bounding box of the specified clusters the spline generated in splines phase.

However, Graphviz model's node ranking in terms of linear integer program is given by:

$$\text{minimize} \sum_{(u,v)\in E} \omega(u,v)\left(y_u - y_v\right) \tag{1}$$

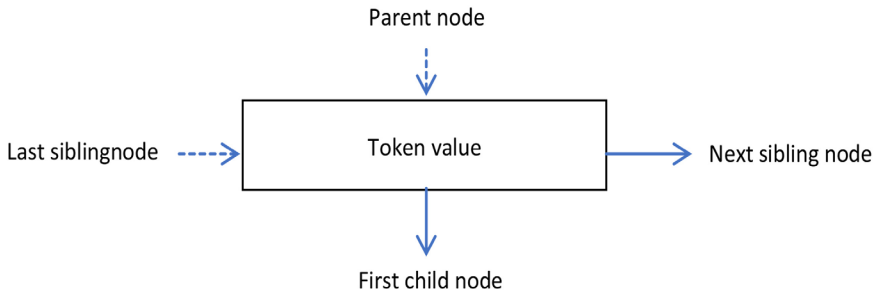

Figure 3. The anatomy of an AST node. Source: Joshi, 2017.

$$\text{Subject to } y_u - y_v \geq \partial(u,v) \ \forall (u,v) \in E \tag{2}$$

where y_u denotes the rank of node u and $\partial(u,v)$ is the minimum length of the edge. By default, ∂ is taken as 1, but the general case supports flat edges, when the nodes are placed on the same rank ($\partial = 0$), or the times when it is important to enforce a greater separation ($\partial > 1$). The weight factor $\omega(u,v)$ allows one to specify the importance of having the rank separation of two nodes be as close to minimum as possible.

3.3. Recursive Descent Parser (RDP)

Historically, there are two types of parsing: top-down parsing and bottom-up parsing. The former expands the non-terminals to match incoming tokens and directly construct a derivation (*i.e.* they construct derivations and parse tree from the root to the leaves) while the latter attempts to match an input with the right-hand sides of the grammar rules, when a match occurs, the right-hand side is replaced by, or reduced to, the nonterminal on the left. So, they construct derivations and parse tree from the leaves to the root. However, this paper would employ the use of top-down parsing techniques, to be precise, recursive-descent parsing (RDP) method for parser generator; an older method for constructing a parser by hand from a grammar that is very effective. By and large, it operates by turning the non-terminals into a group of mutually recursive procedures whose actions are based on the right-hand sides of the formal grammar while the right-hand sides are interpreted in the procedures as tokens are matched directly with input tokens as constructed by a lexer. This parsing technique may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. So, a form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing which uses a stack and a parsing table to parse the input and generate a parse tree [24].

Supposing an interpreter needs to process an arithmetic expression to evaluate additions like 2 + (2 + 2), after tokenization, RDP can be used to construct a parser as shown in **Figure 4** using python compiler version 3 for the grammar defined in **Figure 5**.

```
def exp (self):
self.factor()
      while self.current_token is in [+|-]:
            self.eat (self.current_token)
            self.factor ( )

def factor (self):
      if self.current_token is INT:
            self.eat(INT)
      else:
            self.eat (LPAREN)
            self.exp()
            self.eat (RPAREN)
```

Figure 4. RDP for **Figure 5**.

$$expr = factor\ ((plus|minus)factor))^{+}$$

$$factor = int|(expr)$$

Figure 5. Grammar rule for arithmetic expression.

3.4. Best First Search (BFS) Algorithm

In deciding branch to follow by avoiding backtracking to the beginning of the parsing process in the parser development, BFS algorithm; a heuristic that ranks alternatives in search algorithm at each branching step based on available information [25] was adopted. To do this, nodes is expanded as follows:

1) Search will start at the root node of the phrase structure P_s.

2) The node to be expanded next is selected on the basis of an evaluation function $P_r(n)$ where $n = L_x$, L_x is the valuation function specified for each syntactic domain as given in 3.

$$P_r(L_x) = \sum_{T_1}^{T_n} P_r\left(L_{xT_1}\right) = \text{lowest index\_val} \tag{3}$$

3) The node having lowest value of $P_r(n)$ is selected first that is; $\min P_r(n)$.

4) Lowest value of $P_r(n)$ indicates that goal is nearest from this node. That is, the lower the value, the higher the priority as presented in 4.

$$P_r(P_s) = \begin{cases} HP \text{ if } P_r(P_s) = \text{lowest value of } L_x \\ LP \text{ if } P_r(P_s) = \text{highest value of } L_x \end{cases} \tag{4}$$

where HP and LP denotes highest and lowest priority respectively.

Hence, the algorithm goes thus as shown in **Figure 6**.

This algorithm is implemented using priority queue created during tokenization to develop abstract syntax tree dot file; a configuration showing how the nodes is being visited to generate a specific abstract syntax tree in a portable network graphics (PNG) format.

4. Parsing Process Methodology

In transforming the action, a parser would take into more efficient program, this research establishes the following rules based on the ideas of Recursive Descent Parser (RDP) as depicted in **Table 1**.

Quantitatively, Lexer and Parser are usually combined to analyze the syntax of computer languages. This paper employed: 1) Regular Expression (RE) within python to collect the program code shown in **Figure 7**, as a string and split into individual characters. 2) Declaration of token type variables to represent the token was carried out as depicted in **Figure 8**. 3) Matching a RE with a character type is obtained and presented in **Table 2**.

For the parsing, the use of BFS algorithm in parser development is solely determined using a grammar rule presented in [11] where different methods within the parser class were defined, interpreting grammar programmatically to

```
Create an empty Priority Queue
  Priority Queue Pq;
Determine start point in Queue
  Priority start point = Pq[index of 0]
   i.e. the priority start point is the priority queue character at point zero
Loop through Priority Queue Until its empty
   u = Pq delete minimum value
if u is the goal (i.e. the last in queue) Then
   Exit
Else
  Foreach neighbor v of u
    if v is unvisited then
      mark v as visited
pq add v
 Then mark v as examined
```

Figure 6. BFS algorithm.

```
BLOCK 1:
 int main () {
  if (a == 2) {
    c = a + b;
  }
}
```

Figure 7. Program block 1.

```
C:\Users\HP\Desktop\Newest_Update_complete>python
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from parser_file import Lexer
>>> program_string = "int main ( ) { if ( a == 2 ) { c = a + b ; } }"
>>> lexer_reference=Lexer(program_string)
>>> lexer_reference.get_next_token()
Token(DECLARATION, 'int')
>>> lexer_reference.get_next_token()
Token(KEYWORD_ID, 'main')
>>> lexer_reference.get_next_token()
Token(LPAREN, '(')
>>> lexer_reference.get_next_token()
Token(RPAREN, ')')
>>> lexer_reference.get_next_token()
Token(LBRACE, '{')
>>> lexer_reference.get_next_token()
Token(IF_KEYWORD, 'if')
>>> lexer_reference.get_next_token()
Token(LPAREN, '(')
>>> lexer_reference.get_next_token()
Token(ID, 'a')
>>> lexer_reference.get_next_token()
Token(BOOL_OPRT, '==')
>>> lexer_reference.get_next_token()
Token(INTEGER_CONST, 2)
>>>
```

Figure 8. Token extraction within the command prompt.

generate AST. For instance, to verify a given sets of tokens over a defined grammar, the BFS require the logic (Grammar/Phrase Structure) and pattern

Table 1. Parsing rules for rapid coding.

Rule 1	Methods/Functions should be defined for all non-terminals
Rule 2	All terminals should be handled by an "eat" method
Rule 3	All "or" conditions within the grammar should be programmed using the if, else if, else program conditional statements.
Rule 4	All "*" or "+" depicting 0 or more and 1 or more occurrence of an EBNF rule should be handled within a while loop.

Table 2. Formalized pattern matching.

Token Type	Token Value	Formalization
DECLARATION	INT	TOKEN(DECLARATION, INT)
KEYWORD_ID	MAIN	TOKEN(KEYWORD_ID, MAIN)
LPAREN	(	TOKEN(LPAREN, ( )
RPAREN	)	TOKEN(RPAREN,))
LBRACE	{	TOKEN(LBRACE, {)
IF_KEYWORD	IF	TOKEN(IF_KEYWOR, IF)
LPAREN	**(**	**TOKEN(LPAREN, ( )**
ID	A	TOKEN(ID, a)
BOOL_OPRT	==	TOKEN(BOOL_OPRT, ==)
INTEGER_CONST	2	TOKEN(INTEGER_CONST, 2)
RPAREN	)	TOKEN(RPAREN,))
LBRACE	{	TOKEN(LBRACE, {)
ID	C	TOKEN(ID, c)
ASSIGN	=	TOKEN(ASSIGN, =)
ID	A	TOKEN(ID, a)
AE_OPRT	+	TOKEN(AE_OPRT, +)
ID	B	TOKEN(ID, b)
SEMI	;	TOKEN(SEMI, ;)
RBRACE	}	TOKEN(RBRACE, })
RBRACE	}	TOKEN(RBRACE, })

(Tokens). Hence, when the tokens are passed into the grammar then, a best first process is initiated, the token is searched for within the grammar based on the node having lowest value, if the token is found, it returns match, then collects the next token, and if the token is not found, it returns not match. The process continues until all the strings has been verified and parsed. Shown in Figure 9 is the parsing process for the program block 1in Figure 7.

Having automated the parsing process, AST class; a collection of several AST classes with each having a reference to each method of the parser class was implemented including AST visualizer parameters depicted in Table 3, while the nodegen command line that executes the process and in return, generates the dot file configuration and the picture format of the AST is given in Figure 10.

<program> ::=<type><function name><parameter><compound_statement

Pass 1:- Token(Declaration, int) ~ <type>(int | float | void | double | Boolean) **(Match)**

Pass 2:- Token(Keyword_ID, main) ~ <function name>(main | <id>) **(Match) …**

Pass 3:- TOKEN(LPAREN, ( ) ~ <parameter> (lparen<declaration>rparen ->lparen) **(Match) …**

Pass 4:- TOKEN(RPAREN,)) ~ <parameter> (lparen<declaration>rparen -><declaration> (<type><id> (<comma><type><id>)*)

<type>(int | float | void | double | Boolean) **(!match)** ↩

<id>([a-zA-Z_]([a-zA-Z0-9_])* **(!match)** ↩

rparen**(match) …**

Pass 5:- TOKEN(LBRACE, {) ~ <compound_statement> (lbrace<statement_list>rbrace ->lbrace) **(Match) …**

Pass 6:- TOKEN(IF_KEYWOR, IF) <compound_statement> (lbrace<statement_list>rbrace -><statement_list> (<statement>)*(

<statement>(<assignment_statement><semi> | <declaration_statement><semi> | <if_statement_block>)

-><assignment_statement> (<id> '=' <expr> -><id>([a-zA-Z_]([a-zA-Z0-9_])* **(!match)** ↩

'=' **(!match)** ↩

<expr>(<term> (['+'|'-'] <term>)* -><term>(<factor> (['*'|'/'] <factor>)* ->

(['+'|'-'] <factor>| <integer_const> | <real_const> | lparen<expr>rparen | <id>) ->'+' **(!match)** ↩

'-' **(!match)** ↩

<integer_const> (([0-9])+) **(!match)** ↩

<real_const>(<integer_const> ['.'] <integer_const> -><integer_const> (([0-9])+) (!match) ↩, lparen **(!match)** ↩

<declaration_statement> (<type><id> (<comma><type><id>)* -><type>(int | float | void | double | Boolean) (!match) ↩

<id>([a-zA-Z_]([a-zA-Z0-9_])* **(!match)** ↩

<if_statement_block>(<if_statement> (<else_if_statement>)* [<else_statement>] -><if_statement>(

if <if_block><compound_statement> ->if)**(match)…**

Pass 7:- TOKEN(LPAREN, ( ) ~ <if_block> (lparen<boolexpr> (['&&'|'||'] <boolexpr>)* rparen ->lparen) **(match)**

Figure 9. Parsing process for program block 1.

```
python ASTprogramModel.py textfile.cpp > pm.dot && dot -Tpng -o pm.png
pm.dot
```

Figure 10. Nodegen command line.

The nodegen command line contains the instruction format to generate abstract syntax tree model for the system. Argparser is imported into the python environment in which the Arg is the text-file to parse which is added to the program (*i.e.* the file) to execute. The extension is a dot extension as shown in Figure 11 which define the layered drawing of directed graphs while—Tpng is a flag

Table 3. AST visualizer parameters.

Specification Type	Parameter Set
Node Shape	Rectangle
Font Size	10px
Font Name	Courier
Line Height	0.4dpx
Shape Padding	10px
Path Size	0.4px
Title	AST Dependent

```
digraph astgraph {
labelloc="t";
 label="PROGRAM MODEL GENERATION"; fontsize=20;
 node [shape=rectangle, fontsize=12, fontname="Courier", height=.4,
padding=10];
ranksep=.4;
 edge [arrowsize=.4]
 node1 [label="CompoundStmt"]
 node2 [label="Type"]
 node3 [label="No_Op"]
 node4 [label="Compound"]
 node5 [label="IfStmt"]
 node6 [label="BoolOp"]
 node7 [label="Var Literal"]
 node8 [label="Integer Literal"]
 node6 -> node7
 node6 -> node8
 node5 -> node6
 node9 [label="Compound"]
 node10 [label="AssignStmt"]
 node11 [label="Var Literal"]
 node12 [label="BinOp"]
 node13 [label="Var Literal"]
 node14 [label="Var Literal"]
 node12 -> node13
 node12 -> node14
 node10 -> node11
 node10 -> node12
 node9 -> node10
 node5 -> node9
 node4 -> node5
 node1 -> node2
 node1 -> node3
 node1 -> node4
}
```

Figure 11. Dot File configuration program block 1.

that transform the dot file into portable network graphics (png) format (that is, to see what format dot support), having invoked from a command shell and presented in **Figure 12**.

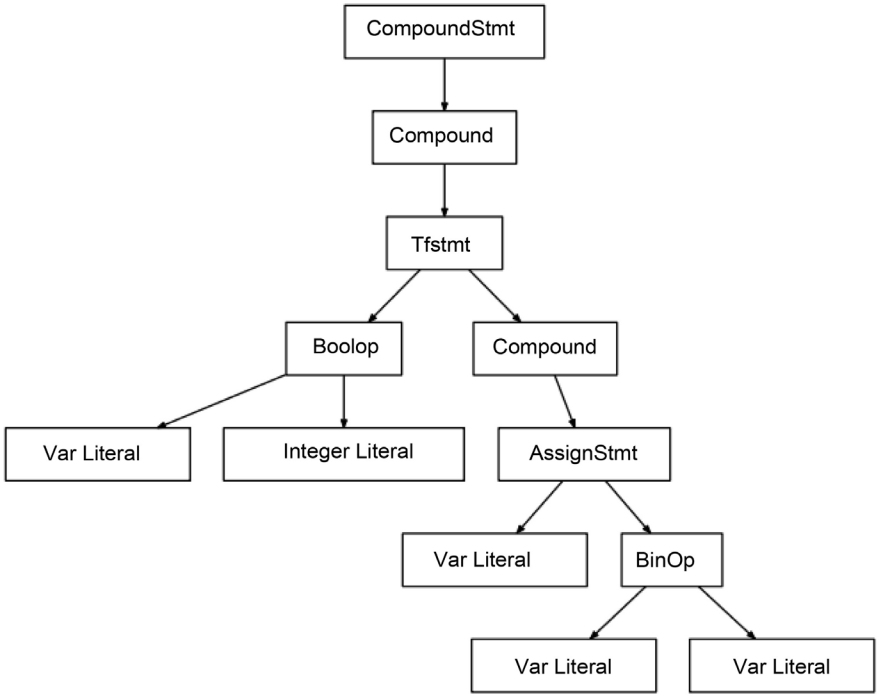

Figure 12. Program abstraction block 1.

5. Conclusions and Future Work

This paper discusses an intermediate representation approach developed. A lexer for transforming input characters into a stream of tokens, initiates a parser for generating abstract syntax tree dot file based on defined grammar rules and produces abstract syntax visualizer interpreting the dot file into portable network graphics format.

This paper observed better result due to portable network graphics transparency instead of portable document format for document sharing. Hence, the efficacy of our approach was established for analyzing C++ code.

Although, the research goal is to develop a model checker for detecting assignment in-guard error including emphasis on break and default keyword in C++ codes. Further research will employ visualization presented as a modeling formalism to detect operator's conformity error, non-inclusion of break and default keyword in selective and iterative structures. Second, different system performance to determine the reliability, validity of programs if optimal and mathematical model for calculating the schedule time for different program line of codes will be deduced.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Click, C. and Paleczny, M. (1995) A Simple Graph-Based Intermediate Representa-

tion. ACM SIGPLAN Workshop, San Francisco, CA, 35-49. https://doi.org/10.1145/202530.202534

[2] Sander, G. (1999) Graph Layout for Applications in Computer Construction. Theoretical Computer Science, Berkeley, CA, 175-214. https://doi.org/10.1016/S0304-3975(98)00270-9

[3] Walker, D. (2003) Intermediate Representation. Princeton University, Princeton, NJ, 1-21.

[4] Vaderna, R., Milosavljevic, G. and Dejanovic, I. (2015) Graph Layout Algorithms and Libararies: Overview and Improvements. Faulty of Technical Sciences, University of Novisad, Serbia.

[5] Strunk, E.A., Aiello, M.A. and Knight, J.C. (2006) A Survey of Tools for Model Checking and Model-Based Development. Technical Report, Department of Computer Science, University of Virginia, Charlottesville, VA.

[6] Braun, M., Buchwald, S. and Zwinkau, A. (2011) FIRM: A Graph-Based Intermediate Representation. Workshop on Intermediate Representations. Karlsruuhe Institute of Technology, Germany, 61-68.

[7] Moller, A. and Schwartzbach, M.I. (2019) Static Program Analysis. Computer Science, Aarhus University, Denmark.

[8] Visser, E. (2004) Program Transformation with Stratego/XT: Rules, Strategies, Tools and Systems. Technical Report, Institute of Information and Computing Sciences Utrecht University, Netherlands.

[9] Ade-Ibijola, A., Ewert, S. and Sanders, L. (2014) Abstracting and Narrating Novice Programs Using Regular Expressions. South Africa Institute of Computer Scientist and Information Technologist (SAICSIT), Centurion, South Africa. https://doi.org/10.1145/2664591.2664601

[10] Gansner, E.R., Koutsofios, E. and North, S. (2015) Drawing Graphs with Dot. https://graphiz.gitlab.io/_pages/pdf/dotguide.pdf

[11] Aliyu, E.O., Adewale, O.S., Adetunmbi, A.O. and Ojokoh, B.A. (2019) Requirement Formalization for Model Checking Using Extended Backus Naur Form. *I-Manager's Journal on Software Engineering*, **13**, 1-6.

[12] Stalmans, E.R. (2010) Visualisation of Abstract Syntax Trees for Coco/R. Computer Science of Rhodes University Grahamstowm, South Africa. https://pdfs.semanticscholar.org

[13] Kreb, N. and Schmitz, L. (2012) JACCIE: A Java-Based Compiler-Compiler for Generating, Visualizing and Debugging Compiler Components. *Elsevier Science of Computer Programming*, **79**, 101-115. https://doi.org/10.1016/j.scico.2012.03.001

[14] Kulkarni, A.A. (2013) Verification of Program Properties with Graphviz. Master Dissertation, College of Engineering, Shivajinagar, Pune.

[15] Gregor, M. and Spalek, J. (2017) Using LLVM-Based JIT Compilation in Genetic Programming. Department of Control and Information Systems, Faculty of Electrical Engineering, University of Zilina, Zilina, Slovak Republic. https://doi.org/10.1109/ELEKTRO.2016.7512108

[16] Celik, Z.B., McDaniel, Z.P. and Tan, G. (2018) SOTERIA: Automated IoT Safety and Security Analysis. USENIX Annual Technical Conference, 1-19

[17] Mount, S. (2013) A Language-Independent Static Checking System for Coding Conventions. Ph.D. Thesis, University of Wolverhampton, England.

[18] Joshi, V. (2017) Leveling up one's Parsing Game with ASTs. https://medium.com/basecs/leveling-up-onesparsing-game-with-asts-d7a6fc2400ff

[19] Fehnker, A., Brauer, J., Huuck, R. and Seefried, S. (2008) Goanna: Syntactic Software Model Checking. 1-6.

[20] Namratha, N. (2010) Visualization of Dataflow Models. A Dissertation in Eindhoven University of Technology, United Arab Emirates.

[21] Gansner, E.R. (2014) Using Graphviz as a Library (Cgraph Version). Graphviz Library Manual.

[22] Dogrosoz, U., Feng, Q., Madden, B., Doorly, M. and Frick, A. (2002) Graph Visualization Toolkits. *IEEE Computer Graphics and Application*, **22**, 30-37. https://doi.org/10.1109/38.974516

[23] Ellson, J., Gansner, E.R., Koutsofios, E., North, S.C. and Woodhull, G. (2003) GraphViz and Dynagraph-Static and Dynamic Graph Drawing Tools. Graph Drawing Software, AT and T Lab-Research, 127-148. https://doi.org/10.1007/978-3-642-18638-7_6

[24] Louden, K.C. (1993) Compiler Construction (Principles and Practice).

[25] Aliyu, E.O. (2019) Development of Model Checking Technique for Operators Conformity Error in Selective and Iterative Structures. PhD Thesis, Department of Computer Science, Federal University of Technology Akure, Ondo State, Nigeria.