

# A Multi-Pass Compiler with Code Optimized Abstract Syntax Tree

**Authors: M. O. Odim<sup>1</sup>; S. A. Arekete<sup>2</sup>; B. O. Oguntunde<sup>3</sup>; A. O. Lawal<sup>4</sup>**

Department of Computer Science, Redeemer's University<sup>1,2,3</sup>; Amazon Web Services (AWS)<sup>4</sup>  
*odimm@run.edu.ng<sup>1</sup>; areketes@run.edu.ng<sup>2</sup>; oguntunden@run.edu.ng<sup>3</sup>; alawalo@amazon.com<sup>4</sup>*

**DOI: 10.26821/IJSHRE.11.5.2023.110409**

## ABSTRACT

*This study proposes a multi-pass compiler with code optimized abstract syntax tree based on a subset of C# as the source and C constructs as the target, with the view to enhancing the understanding of compiler design, which conceptually has been a rather difficult discourse. A single-pass compiler allows a single traversal over the source code or intermediate form. This does not allow for a distributed compilation and modular development. On the other hand, the proposed scheme decouples the type-checker from the abstract syntax tree, thus separating them into independent segments that allows a multi-pass over the source code; this in turn provides a mechanism for modular development, which encourages distributed compilation. The proposed scheme constructs the abstract syntax tree before the type-checking, thus allowing another traversal on the intermediate form of the source code. An optimised tree was constructed after code optimisation had been conducted. The scheme is based on only small subset of C# constructs. Further study would be focused on adding more constructs.*

**Keywords: Compiler, Code optimisation, Abstract Syntax Tree, Multi-pass compiler.**

## 1. INTRODUCTION

Compiler construction is an essential part of applied computer science, which has been traditionally regarded as a complex piece of software. It is rather worrisome that little or no attention is being paid to the relevance and knowledge of compiler construction by students and young computer scientists nowadays. This perception stems mainly from traditional methods of teaching compilers as well as the lack of

available examples of small and functional compilers for real programming languages. Compiler construction requires a lot of practical engagements; however, teaching the course is based most times on methods that are too abstract for learning, which consequently, reduces the interest of the students. In addition, a number of students lack classical programming techniques and software programming capabilities required for in-depth knowledge of compiler construction [1]. The exclusion of compiler construction and language theory in many computing degrees has been reported in the literature [2], with the rationale that they are now no more vital to modern software engineering practice, regardless of the importance of compiler to computing in general. Compilers and operating systems provide the basic interfaces between a programmer and the machine. Compiler construction provides an extensively useful exercise in software engineering [3]; and hence cannot be considered to be irrelevant to modern software engineering practice. In view of the above, this study proposes a multi-pass compiler with code optimized abstract syntax tree (AST) as a tool for enhancing the art of learning compiler construction.

## 2. RELATED WORK

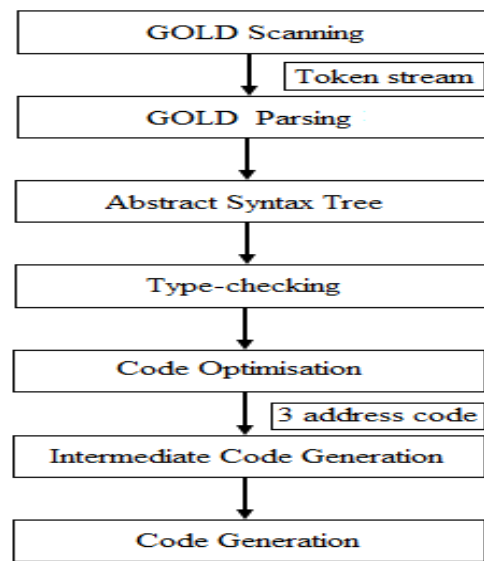
A couple of efforts seeking to enhance the understanding of compiler design have been reported in the literature. For instance, [4] designed a compilation simulator that could serve as an aid to learning compiler design which was composed of a virtual machine and its machine language, a simple high-level language and its compiler simulator. The simulator could recognise, carry out and describe the syntax and lexical analysis and the code generation phases of compilation. However, the language could only recognise digits and accept

single lower-case letters as variable names. An argument for the inclusion of at least some facets of compiler construction and language theory in computing degrees was made in [2]. An outline curriculum based around the recursive-descent methodology was proposed and a toolkit described that supports the delivery of this curriculum. Small languages, formal methods and object-orientation consolidation were identified as evidence of the applicability of compiler to teaching in a wider software engineering context. In [3] a compiler system for an adaptive computing to provide a general knowledge about compiler design and its implementation was reported. Although the paper concentrated on the implementation of a compiler, an outline that builds upon the compiler was also presented. [5] proposed and implemented a case-based and project-based learning environment for teaching important compiler design concepts. Adekoya in (Adekoya, 2011) developed a compiler that employed a single-pass compilation where the type-checker was an integral component of the AST generation process. The compiler does not perform any form of code optimisation. The proposed system is a multi-pass compiler where the type-checker is decoupled from the AST. Decoupling the type-checking module from the AST makes type-checking independent of the AST. Therefore, there can be multiple transversal on the intermediate form of the source code. This makes for the modularity of the system and allows distributed compilation. It also allows for modularized development. This study also allows a type of code optimisation on AST during type-checking. Further related work includes [6] which reported a large verification effort in constructing an initial fully trusted bootstrap compiler executable for a realistic system programming language and real target processor. The construction and verification process contain three activities: the verification of the compiling specification with respect to the language semantics and a realistic correctness criterion. Second, the implementation of the specification in the high-level source language following a transformational approach, and finally, the implementation and verification of a binary executable written in the compiler's target language. For the latter task, a realistic technique has been developed, which is based on rigorous a-posteriori syntactic code

inspection and which guarantees, for the first time, trusted execution of generated machine programs.

### 3. METHODOLOGY

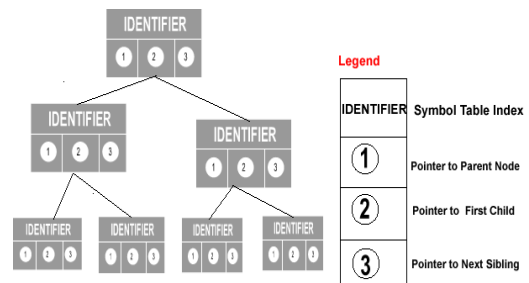
The proposed system is a multi-pass compiler where the type-checker is decoupled from the Abstract Syntax Tree (AST). Decoupling the type-checking module from the AST makes type-checking independent of the AST. Thus, there could be multiple transversal on the intermediate form of the source code. Figure 1 shows the process flow of the proposed scheme.



**Fig 1: Process flow of the proposed scheme**

#### 3.1 Abstract Syntax Tree

The syntax tree is a critical data structure for storing the intermediate form of the source programs. Figure 2 shows the Abstract Syntax Tree of the proposed system.



**Fig 2: AST Design**

#### 3.2 Generating the Abstract Syntax Tree

To illustrate the algorithm for generating the AST, an example grammar was employed as follows.

Grammar: Start Symbol = E

$E := E + T \mid T$

$T := T * F \mid F$

$F := i \mid (E)$

Given the input  $(i*i+i)$ , a bottom-up parse of the input would typically proceed as in Table 1.

**Table 1: AST Generation Procedure**

(i*i+i)		
	<b>Parsing action</b>	<b>AST Generation Procedure</b>
(F*i+i)	Reduce by $F := i$	F. node = new leaf (id, id.entry)
(T*i+i)	Reduce by $T := F$	T. node = F. node
(T*F+i)	Reduce by $F := i$	F. node = new leaf (id, id. entry)
(T+i)	Reduce by $T := T * F$	T. node = new node (*, T.node, F.node)
(E+i)	Reduce by $E := T$	E. node = T. node
(E+F)	Reduce by $F := i$	F. node = new leaf(id,id.entry)
(E+T)	Reduce by $T := F$	T. node = F. node
(E)	Reduce by $E := E + T$	E. node = new node (+, E.node, T. node)
F	Reduce by $F := (E)$	F. node = E. node
T	Reduce by $T := F$	T. node = F.node
E	Reduce by $E := T$	E.node = T.node
Accept		

The generation procedure in Table 1 explains the fact that terminals (operators and operands) are the only nodes that would ultimately reflect on the tree. The non-terminal nodes are used as temporary pointers only. The leaf nodes are the operands (identifiers, numbers, literals) and the internal nodes of the tree are the operators (+,\*). Traversing the abstract syntax tree involves starting with the node for the start symbol, E.

We use a sample C# program to illustrate the compilation and building of the tree as follows:

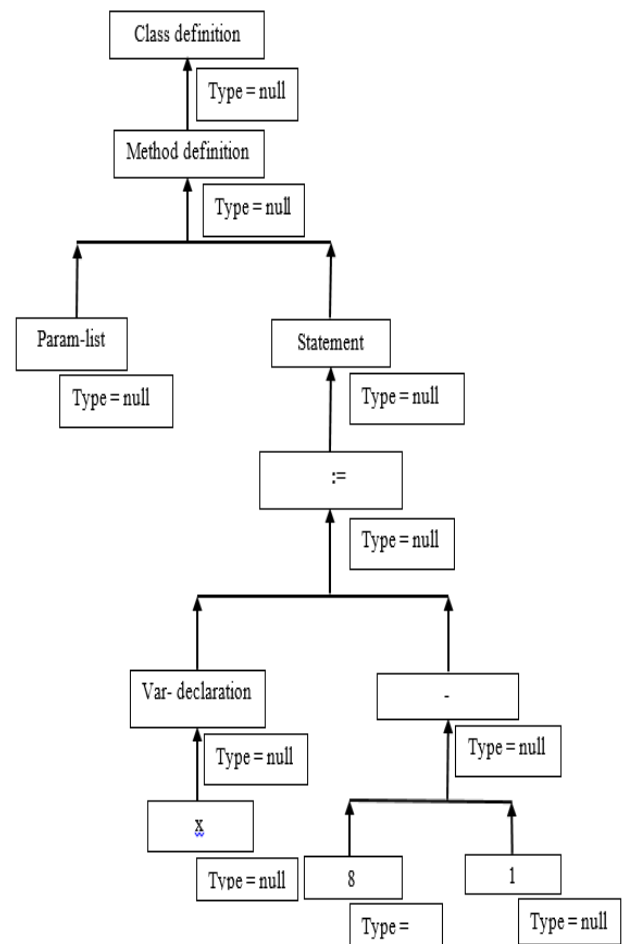
**Sample Program:**

```
public class Example{
    public int main(){
        int x;
        x = 8 - 1;
    }
}
```

The runtime sequence of how the tree is built and the order in which operators are added to the tree for are as follows:

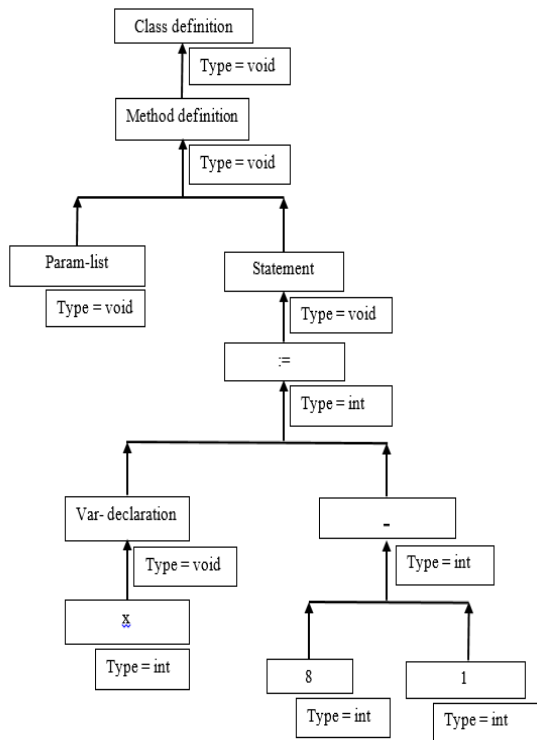
1. addOperatorNode("param-list", null).
2. addOperatoreNode("var-declaration", unary).
3. addOperatorNode("-", binary).
4. addOperatorNode(":=", binary).
5. addOperatorNode("statements", binary).
6. addOperatorNode("method-definition", nary).
7. addOperatorNode("class-definition", ternary).

Figure 3 shows the structure of the AST for the above sample program before type-checking.



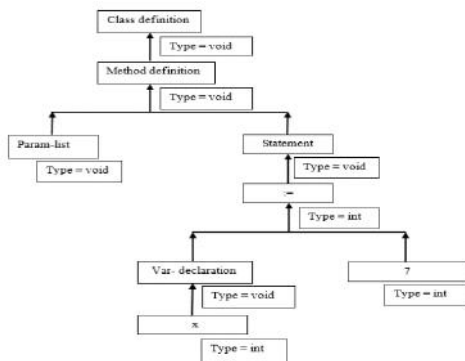
**Fig 3: The AST of a sample program before type-checking**

As shown in Figure 3, the type of the nodes is null before type-checking was done. Figure 4 shows the structure of the tree after type-checking.



**Fig 4: The AST of a Sample program after type-checking**

During type-checking, the data types are checked for type-compatibility. Figure 5 shows the structure of a code optimised AST after type-checking.



**Fig 5: The AST of a sample program after type-checking and code optimisation**

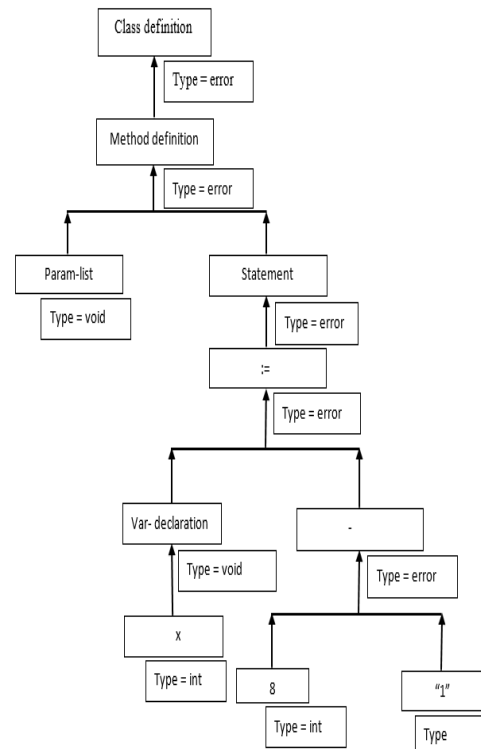
After type-checking, a code optimised AST is produced.

### 3.2 The structure of an AST tree with type error

Consider the program below which is syntactically incorrect,

```
public class Example{
    public int main(){
        int x;
        x = 8 - "1";
    }
}
```

The structure of the AST is shown in Figure 6.



**Fig 6: The AST of a sample program with type-error**

The tree above shows the AST of a program with a type error. The error is propagated upward and it returns error.

## 4. IMPLEMENTATION AND RESULTS

This section presents the implementation and result of the proposed scheme.

### 4.1 Implementation

There are two main parts to the system: the frontend and backend. The frontend of the system is the user-interface. The input from the users are entered via the frontend while the output from the system are also displayed to the user via this part. Other components in the user interface includes the menu structure, the program editor and the I-O Window. The backend, on the other hand, consists

of the computational logic of this system. The scanning code and the module that parses source programs are included in this unit. The backend also contains the Abstract Syntax Tree (AST) generation module, type-checker, 3-address code generation module and the C++ code generator. The backend also incorporates the functional modules involved in assembling, linking and execution of programs. The backend houses a number of third-party software libraries such as GOLD parser, GNU compiler collection and Java Runtime library. The key user-interface elements of this implementation are the menu structure, the program editor and the I/O window. Figure 7 depicts the user interface elements.



**Fig 7: IDE of the proposed scheme**

The File menu contains menu items needed to perform routine file management functions such as printing, opening and saving of files. With the file menu also comes an option for exiting the IDE – Integrated development environment.

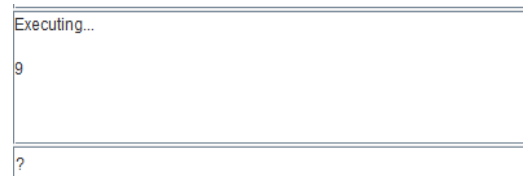
Users of the system need an interface for writing programs before proceeding with compilation and execution. The centrally located editor within the IDE provides this function. Below is a sample C# program written and typed using the editor.

```

public class Test {
    public int main() {
        int x ;
        int y ;
        x = 3;
        y = 6;
        Console.WriteLine ( x+y);
    }
}
    
```

**Fig 8: System Editor**

The I/O window displays message from the system. Some of these messages could be as a result of errors in the compilation process while some could be debugging messages from the system. Where debugging is turn on, other media could be used to send message to the users besides this window. The input-output window may also be used to get data inputs from the users of the system.



**Figure 9: Input-Output Window**

#### 4.1.1 Type-checker

The type-checker implemented in this system tests simple types. Type compatibility is enforced in every construct found in the source. Unlike the existing one, type-checking is decoupled from the abstract syntax tree construction. The philosophy behind this system of typing is partly to simplify implementation and also exploit the simplicity of syntax-directed translation. Figure 10 depicts a type checking for relational operators.

```

+-
*
if (oNode.n_identification.equals("<") || oNode.n_identification.equals(">")
    || oNode.n_identification.equals("==") || oNode.n_identification.equals("<=")) {
    Node oNode1 = typeCheckTree(oNode.n_firstChild);
    Node oNode2 = typeCheckTree(oNode.n_nextSibling);
    if (!operandsCompatibleRelationalOperations(oNode1, oNode2, null)) {
        oNode.iDataType = compiler.iDataType_error;
    }
    else {
        oNode.iDataType = compiler.iDataType_boolean;
    }
    oNode1 = null;
    oNode2 = null;
    return oNode;
}
    
```

**Fig10: Type-checking for relational operators**

Figure 10 was extracted from the code-base of type-checking module. It shows how the type-checker can be invoked to check the type compatibility of the two operands for relational operators.

#### 4.1.2 Code optimisation.

This system allows a type of code optimisation on the AST. This code optimisation performed on the AST is embedded in the type-checking module.

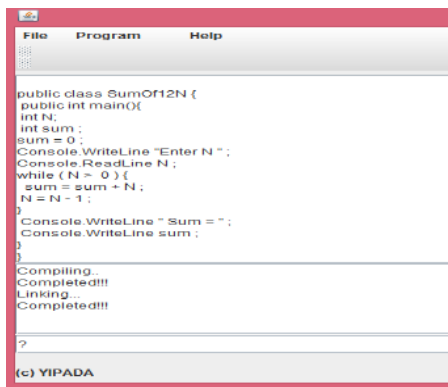
The Figure 11 was extracted from the code-base of the type-checking module. It shows how the code optimisation can be performed on numerical literals.

## 4.2 Result

When a program is compiled as shown Figure 12, at the backend, the GOLD parser which is scanner generator and parser generator scans the source language to obtain tokens. This is traditional function of a scanner generator. Figure 12 illustrates the compilation of a typical source program in C#.

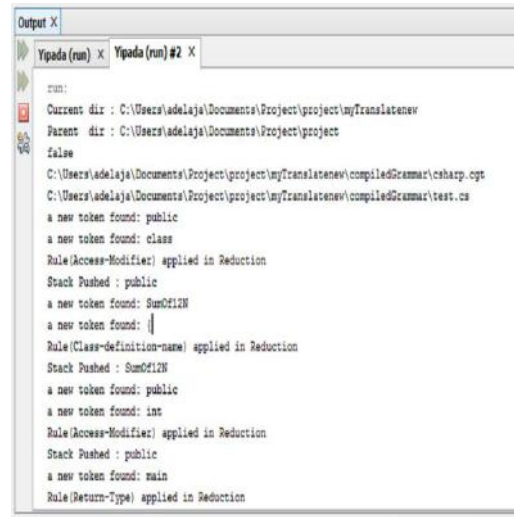
```
if (this.is_numeric_literal(oNode1) && this.is_numeric_literal(oNode2)) {  
    // coalesce/evaluate operands, set value of operator(oNode) and change status of oNode to terminal  
    oNode = evaluate_ArithMetic_operands(oNode.n_identification, oNode, oNode1, oNode2);  
    oNode.iDataType = greaterType(oNode1, oNode2);  
    Debug.println(Integer.toString(oNode.iDataType));  
    oNode.type = compiler.iTypeOfAstNode_NumberLiteral;  
    oNode.n_firstChild.n_nextSibling = null;  
    oNode.n_firstChild = null;  
    oSymbolTable.setDataTypesymbol(oNode.iSymbolTableIndex, Integer.toString(oNode.iDataType));  
}  
  
} else {  
    oNode.iDataType = greaterType(oNode1, oNode2);  
    oSymbolTable.setDataTypesymbol(oNode.iSymbolTableIndex, Integer.toString(oNode.iDataType));  
}  
  
private Node evaluate_ArithMetic_operands(String s_op_code, Node oNode, Node oNode1, Node oNode2) {  
  
    if (s_op_code == "+") {  
        if (oNode1.iDataType == compiler.iDataType_float || oNode2.iDataType == compiler.iDataType_float) {  
            // literals are float  
  
            oNode.n_identification = Float.toString(Float.valueOf(oNode1.n_identification)  
                + Float.valueOf(oNode2.n_identification));  
        } else if (oNode1.iDataType == compiler.iDataType_long || oNode2.iDataType == compiler.iDataType_long) {  
            // literals are long  
            oNode.n_identification = Long.toString(Long.valueOf(oNode1.n_identification)  
                + Long.valueOf(oNode2.n_identification));  
        } else {  
            // literals are integer  
            oNode.n_identification = Integer.toString(Integer.valueOf(oNode1.n_identification)  
                + Integer.valueOf(oNode2.n_identification));  
        }  
    }  
}
```

**Fig 11: Code optimisation**



**Fig 12: Compiling a program on the sum of N numbers**

The first segment of Figure 12 shows the source program in C# and the next segment indicates that the compilation using the developed compiler compiles successfully. Figure 13 shows the output of the scanner and parser.



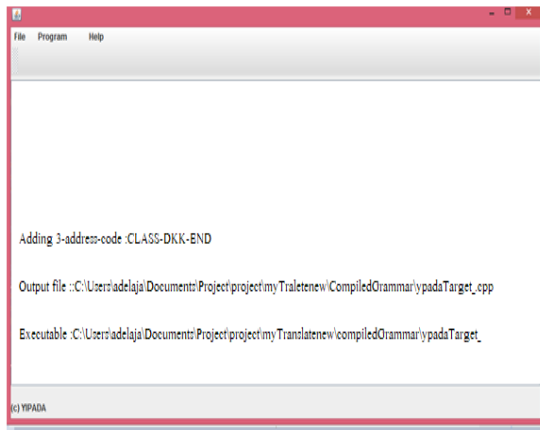
**Fig 13: Output of the Scanner and Parser.**

The system returns “a new token found” whenever the scanner successfully finds a new token as shown in Figure 13.

After scanning, parsing is also done by parser generator. Parsing is done by reduction of tokens by rules to get to the start symbol, since the parser generator is a bottom-up parser. When parsing has been successfully carried out, the system returns the rule that has been applied in reduction of that particular token. For example, in Figure 13, the parser reduces the token “public” and “class” by a rule called “Access-Modifier”. We note that the operation of the scanner and the parser are interleaved. Parsing of the program takes place incrementally. The parser will depend on the scanner for inputs which are tokens. After parsing, the compiler built an intermediate form of the source code. One intermediate form is the Abstract Syntax Tree (AST).

The existing system builds the Abstract Syntax Tree and performs type-checking while building the tree. This is shown in Figure 14.





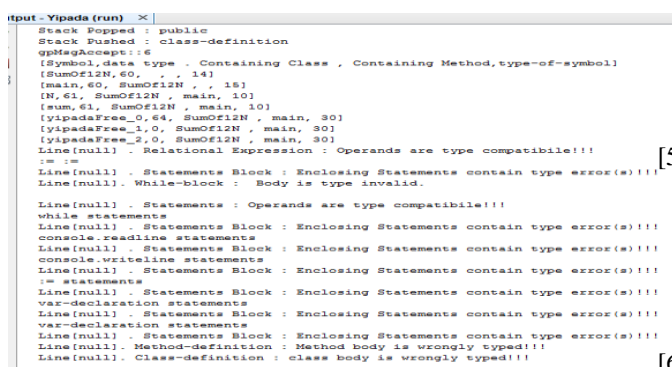
**Fig 18: The path to the target code generated**

The Figure 18 shows the path to the generated target code. If an incorrect program with a type error is compiled, the output depicted in Figure 19 presenting a source program with an incorrect type is displayed.



**Fig 19: A source program with an incorrect type**

The scanner generated the tokens successfully and the parser successfully reduced the tokens. The AST was also built successfully. However, on getting to type-checking, the type-checker prompted an error which was propagated to the top of the tree. Hence, compilation gets terminated at this phase (Figure 20).



**Fig 20: Output of an incorrect program**

## 5. CONCLUSION

A multi-pass compiler was designed, implemented and tested. The type-checker is independent of the Abstract Syntax Tree. A form of code optimisation on the AST was incorporated into the type-checker. The significance of this study is that it allows for modular development. This supports distributed compilation. Code optimisation helps in providing an optimised form of the source program and also helps in memory storage. However, the scanner/parser generator only simplifies the lexical and syntax analysis and accept only LALR [can this term be explained?] (1) grammars. The study only considered a very small subset of C# programming language. Future work would ideally focus on adding more constructs.

## 6. ACKNOWLEDGMENTS

Our thanks to the management and staff of Redeemer's University for providing the enabling laboratory, power and Internet facilities towards this study.

## 7. REFERENCES

- [1] A. Ghuloum, "An Incremental Approach to Compiler Construction.," *Scheme and Functional Programming.*, pp. 27-37, 2006.
- [2] A. C. Milne and E. V. McAdam, "Compilers, The Forgotten Subject?," *Innovation in Teaching and Learning in Information and Computer Sciences*, vol. 10, no. 2, pp. 32-40, 2015.
- [3] M. Jain, N. Sehrawat and N. Munsri, "Compiler System for Adaptive Computing," *International Journal of Computer Science and Mobile Computing*, vol. 3, no. 10, pp. 850-852, 2014.
- [4] M. O. Odim, A. Olawoye, P. Nwekeala and O. D. Akinola, "Compilation Simulator," *International Journal of Physical Sciences*, vol. 3, no. 2, pp. 14-20, 2008.
- [5] D. Kundra and . Sureka, "An Experience Report on Teaching Compiler Design Concepts Using Case-Based and Project-Based Learning Approaches," *IEEE Xplore*, 2017.
- [6] A. Dold, F. v. Henke and W. Goerigk, "A

- Completely Verified Realistic Bootstrap Compiler," *International Journal of Foundations of Computer Science*, vol. 14, no. 4, pp. 659-680, 2003.
- [7] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman , *Compilers: Principles, Techniques, and Tools*, 2nd, Ed., 2007.
- [8] A. Levitin, *Introduction to the Design and Analysis of Algorithm*, 3rd, Ed., 2012.
- [9] M. Marjan and Z. Viljem, *An Educational Tool for Teaching Compiler Construction.*, 2000.
- [10] N. Wirth, *Compiler Construction*, 2005.
- [11] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools.*, 2nd ed., 2007.
- [12] F. J. F. Benders, J. W. Haaring, T. H. Jaasen, D. Meffert and A. C. van Oostenrijk, *Compiler constructions: A practical approach*, 2003.
- [13] R. A. Brooker and D. Morris, "A general translation program for phrase structure languages," *ACM 9:1*, pp. 1-10, 1972.
- [14] . M. Torben, *Basics of compiler Design.*, Anniversary ed., 2010.
- [15] B. Richard , *Understanding and Writing Compilers: A do-it-yourself guide*, 2008.
- [16] P. D. Terry, *Compilers and Compiler Generators: An introduction with C++*, 1996.
- [18] D. Vermeir, *An Introduction to compilers DRAFT*, 2009.
- [19] L. YangSun and S. YunSik, "A Study on Verification and Analysis of Symbol Tables for Development of the C++ Compiler," *International Journal of Multimedia and Ubiquitous Engineering Vol 7*, p. 174, 2012.
- [20] F. L. Deremer, "Simple LR(k) grammars.," *ACM 14*, pp. 453-460, 1971.
- [21] J. P. Anderson, "A note on some compiling algorithms," *Communication ACM 7:3*, pp. 149 - 150, 1964.
- [22] B. Amit and . K. J. Brijendra , "Issues in Implementation of Parallel Parsing on Multi-core Machines," *International Journal of Computer Science, Engineering and Information Technology (IJCEIT)*,, vol. 4, no. 5, pp. 52-57, 2014.
- [24] A. A. Aaby, *Compiler Construction using Flex and Bison*, Walla Walla College, 2004.
- [25] O. A. Adekoya, "A Compiler for a subset of C#," 2011.
- [26] A. R. Adekoya, "A Compiler for a Subset of C#," Unpublished, University of Lagos, Akoka., 2011.