



***Report on***

**“A Mini Compiler for C”**

*Submitted in partial fulfillment of the requirements for Sem VI*

***Compiler Design Laboratory***

**Bachelor of Technology  
in  
Computer Science & Engineering**

*Submitted by:*

<b>Maanvi Nunna</b>	<b>01FB16ECS187</b>
<b>Malaika Vijay</b>	<b>01FB16ECS189</b>
<b>Nandagopal N.V</b>	<b>01FB16ECS221</b>

*Under the guidance of*

**Madhura V**  
Assistant Professor  
PES University, Bengaluru

**January – May 2019**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING  
FACULTY OF ENGINEERING  
PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)  
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

## TABLE OF CONTENTS

Chapter No.	Title	Page No.
1.	INTRODUCTION (Mini-Compiler is built for which language. Provide sample input and output of your project)	01
2.	ARCHITECTURE OF LANGUAGE: <ul style="list-style-type: none"> <li>What all have you handled in terms of syntax and semantics for the chosen language.</li> </ul>	02
3.	LITERATURE SURVEY (if any paper referred or link used)	03
4.	CONTEXT FREE GRAMMAR (which you used to implement your project)	
5.	DESIGN STRATEGY (used to implement the following) <ul style="list-style-type: none"> <li>SYMBOL TABLE CREATION</li> <li>ABSTRACT SYNTAX TREE</li> <li>INTERMEDIATE CODE GENERATION</li> <li>CODE OPTIMIZATION</li> <li>ERROR HANDLING - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).</li> </ul>	
6.	IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following): <ul style="list-style-type: none"> <li>SYMBOL TABLE CREATION</li> <li>ABSTRACT SYNTAX TREE (internal representation)</li> <li>INTERMEDIATE CODE GENERATION</li> <li>CODE OPTIMIZATION</li> <li>ERROR HANDLING - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).</li> <li>Provide instructions on how to build and run your computer</li> </ul>	
7.	SNAPSHOTS (of different outputs)	
8.	RESULTS AND possible shortcomings of your Mini-Compiler	
9.	CONCLUSIONS	
10.	FURTHER ENHANCEMENTS	
REFERENCES/BIBLIOGRAPHY		

## **INTRODUCTION**

The goal of this project is to develop a construct-specific mini-compiler for C. The primary constructs handled are the iterative constructs -

- while
- do-while

This mini-compiler is written in Python with ply (Python Lex and Yacc), a lex and yacc parsing tool for Python. The essential features of PLY are:

- it is implemented entirely in python
- it uses LR parsing that is well suited for large and complex grammars, specifically LALR(1)
- it supports features like empty productions and precedence rules
- it doesn't do anything more or less than provide the basic lex/yacc functionality
- PLY also doesn't provide inbuilt functions for construction abstract syntax trees or tree traversal, giving the user the freedom to construct them as they wish

Overview of PLY:

It consists of two separate modules `lex.py` and `yacc.py` both of which are found in the `ply` module. The `lex.py` is used to convert the input text into a set of tokens specified by a collection of regular expression rules. The `yacc.py` is used to parse the context free grammar specified in the program to identify the correct language syntax. `Lex.py` provides an external interface in the form of a `token()` function that returns the next valid token on the input stream. `Yacc.py` calls this repeatedly to retrieve tokens and invoke grammar rules.

The main difference between `yacc.py` and Unix `yacc` is that `yacc.py` doesn't have a separate code-generation process. PLY relies on introspection to build its lexers and parsers. In Unix `yacc` you need a separate input file that is later converted into the source file, whereas in PLY the specifications given to it are through the python programs. Hence, you don't need additional source files or extra compiler construction steps. Lastly, since the generation of parsing tables is expensive, PLY caches the results and saves them to a file. If no change is detected, the parsing tables are read from the cache, else they are regenerated.

## **ARCHITECTURE OF THE LANGUAGE**

This mini-compiler is built to support a bare-bones version of the language C.

The grammar written for this version of C handles the following language components -

1. Iterative constructs - while and do-while
2. Arithmetic and logical expressions
3. Declarative statements (variable declarations of primitive and compound types)
4. Function calls

The Lexical Analysis phase is designed to recognize syntactic units of C. This includes keywords, operators, C style identifiers, signed and unsigned integral and floating point values, as well as real numbers in scientific notation. Ply Lex provides support for defining keywords and reserved words as tokens. Other tokens such as identifiers are specified by means of regular expressions.

The tokens returned by the Lexical Analysis phase are then parsed using the grammar written for this version of C. In this phase, the tokens are parsed to match the appropriate C constructs. We add error checking mechanisms to the parsing process i.e. a global error production is defined to notify the user of errors with their corresponding line numbers.

## CONTEXT FREE GRAMMAR

**start** : PRE MAIN stmt\_grp

**stmt\_grp** : OPEN\_CURLY\_BRACES CLOSE\_CURLY\_BRACES  
| OPEN\_CURLY\_BRACES new\_node stmt\_list CLOSE\_CURLY\_BRACES

**stmt\_list** : stmt  
| stmt stmt\_list

**stmt** : sel\_stmt  
| iter\_stmt  
| asgn\_expr  
| stmt\_grp  
| var\_decl  
| expr\_stmt

**expr\_stmt** : expr SEMICOLON

**expr** : bin\_expr  
| paren\_expr  
| primary\_expr\_id  
| primary\_expr\_const  
| un\_expr  
| fun\_call

**bin\_expr** : expr PLUS expr  
| expr MINUS expr  
| expr STAR expr  
| expr FORWARD\_SLASH expr  
| expr PERCENTAGE expr  
| expr POWER expr  
| expr RIGHT\_SHIFT expr  
| expr LEFT\_SHIFT expr  
| expr GREATER\_THAN expr  
| expr LESSER\_THAN expr  
| expr GREATER\_THAN\_EQUAL expr  
| expr LESSER\_THAN\_EQUAL expr

- | expr EQUAL\_TO\_EQUAL expr
- | expr NOT\_EQUAL\_TO expr
- | expr OR\_OR expr
- | expr AND\_AND expr
- | expr AND expr
- | expr OR expr

**paren\_expr** : OPEN\_PARANTHESES expr CLOSE\_PARANTHESES

**primary\_expr\_id** : ID

**primary\_expr\_const** : CONST

**fun\_call** : ID OPEN\_PARANTHESES param CLOSE\_PARANTHESES

**param** : primary\_param  
| primary\_param COMMA param

**primary\_param** : key\_w\_param  
| expr  
| empty

**key\_w\_param** : ID EQUAL\_TO expr

**un\_expr** : pre\_op  
| post\_op

**pre\_op** : PLUS\_PLUS ID  
| MINUS\_MINUS ID  
| EXCLAMATORY\_MARK ID

**post\_op** : ID PLUS\_PLUS  
| ID MINUS\_MINUS

**var\_decl** : qualifier signed\_unsigned specifier type list SEMICOLON

**qualifier** : CONST  
| VOLATILE

| empty

**specifier** : SHORT

| LONG

| LONG\_LONG

| empty

**signed\_unsigned** : SIGNED

| UNSIGNED

| empty

**type** : basic

| userdef

**basic** : INT

| CHAR

| DOUBLE

| FLOAT

**userdef** : struct

| union

**struct** : T\_STRUCT ID OPEN\_CURLY\_BRACES decl\_list CLOSE\_CURLY\_BRACES

**decl\_list** : var\_decl SEMICOLON decl\_list

| SEMICOLON

**union** : T\_UNION ID OPEN\_CURLY\_BRACES decl\_list CLOSE\_CURLY\_BRACES

**list** : primary\_list

| primary\_list COMMA list

**primary\_list** : ID

| ID OPEN\_SQUARE\_BRACES expr CLOSE\_SQUARE\_BRACES

| ID EQUAL\_TO expr

| ID OPEN\_SQUARE\_BRACES expr CLOSE\_SQUARE\_BRACES EQUAL\_TO .  
OPEN\_CURLY\_BRACES initialiser CLOSE\_CURLY\_BRACES

**initialiser** : ID COMMA initialiser

| CONST COMMA initialiser

| ID

| CONST

**sel\_stmt** : IF OPEN\_PARANTHESES expr CLOSE\_PARANTHESES stmt ELSE stmt

| IF OPEN\_PARANTHESES asgn\_expr CLOSE\_PARANTHESES stmt ELSE stmt

| IF OPEN\_PARANTHESES expr CLOSE\_PARANTHESES stmt

| IF OPEN\_PARANTHESES asgn\_expr CLOSE\_PARANTHESES stmt

**asgn\_expr** : ID asgn\_op expr SEMICOLON

**asgn\_op** : EQUAL\_TO

| PLUS\_EQUAL\_TO

| MINUS\_EQUAL\_TO

| STAR\_EQUAL\_TO

| FORWARD\_SLASH\_EQUAL\_TO

| PERCENTAGE\_EQUAL\_TO

| LEFT\_SHIFT\_EQUAL\_TO

| RIGHT\_SHIFT\_EQUAL\_TO

| AND\_EQUAL\_TO

| OR\_EQUAL\_TO

**iter\_stmt** : WHILE OPEN\_PARANTHESES expr CLOSE\_PARANTHESES stmt

| DO stmt\_grp WHILE OPEN\_PARANTHESES expr CLOSE\_PARANTHESES  
SEMICOLON



## **DESIGN STRATEGY**

### **The Symbol Table**

The symbol table is designed as a tree of scope specific tables. We define each node in the tree as a scope specific table. Each node is designed as an object of class "Scope". Each scope contains a reference to its parent scope (required to traverse the ancestors of the current scope to check for the existence of identifiers in enclosing scopes), a list of its child nodes, and a dictionary of tokens. The tree structure facilitates the maintenance of the hierarchy of block defined in a C program

### **Abstract Syntax Tree**

The Abstract Syntax tree is designed using objects of type AST\_Node defined in the following section. Each node has information about the type of the node i.e. the C construct it represents, and what it's children are. Children could be the operands of an operator, or the statements contained in a block of statements, parameter lists for function calls etc. Some internal nodes of the tree do not represent C constructs, but are required to maintain the sequence of operations/statements in a line of code, as is detailed in examples to follow. The tree is constructed during parsing and is used as the basis for Intermediate Code Generation.

### **Intermediate Code Generation**

Intermediate code would be generated by traversing the AST because the AST maintains the structural information of the source code. The design is to traverse the AST in a depth-first left-to-right manner and generate intermediate instruction units. This way, the order in which the intermediate code is generated matches with the order of instructions in the source code. Intermediate code is expected to follow the three address format. Use of temporaries is also necessary for complicated expressions or compound statements for which there cannot be a single intermediate instruction. These temporaries may also be referenced in other instructions, introducing opportunity for a number of optimisations - some of which are covered in the next section.

## Code Optimisation

Code optimisations chosen for the scope of this project include - Constant propagation and copy propagation.

**Constant propagation** : This type of optimisation is applicable in cases where a variable is assigned a constant value and used later in the code. This provides opportunity for optimisation where the variable can merely be replaced with the constant value it was originally assigned, provided that it is not modified. This saves run time as well as memory space.

**Copy propagation** : This type of optimisation is comparable to constant propagation. Unlike constant propagation, copy propagation attempts to optimise on variables that are merely copies of other variables. Thus, if the copy of a variable is used later in the code, it can be eliminated and the source of the copy can be used directly. This is again only applicable when the copied value is not changed through the course of execution. This too saves run time as well as memory space.

## Error Handling

Productions were introduced in the grammar itself to explicitly invoke certain errors that cannot otherwise be detected in the lexical/semantic phases. Particularly, verifying if a variable is declared which can be either - in the given scope or in an associated parent scope. When this is violated, an explicit error is called to report the undeclared variable.

## IMPLEMENTATION DETAILS

### Symbol Table

We define a class of type “Scope”, instances of which represent symbol tables for the scope in which the object was created. Each Scope object has a reference to the parent scope, a list of child scopes, and a dictionary of identifiers found in that scope, as well as those declared elsewhere but are used in the current scope.

This dictionary uses the tuple of the identifier and a flag as a key i.e. a key is defined as (ID, flag). This flag indicates whether the variable is local to this scope or was declared in an enclosing scope.

The values associated with each key are defined as a dictionary with the following fields

- parent\_reference - holds a reference to the parent scope (required for direct access to the symbol table node containing an identifier not declared in the current scope)
- scope\_number - an identifier for a scope
- line\_no - the line number where the identifier is encountered
- qualifier - whether a variable was declared as const/volatile etc.
- signed\_unsigned - whether a variable was declared to be signed/unsigned
- specifier - whether a variable is defined to be long/short etc.
- use\_count - to count the number of times a variable is used

Each time a variable is encountered, the local symbol table is checked. If it is contained in this table, nothing is done. Else, the parent scopes are checked. If found, an entry is made into the symbol table along with the indication that the variable is not local to this scope.

When a variable is declared, it is added to the symbol table, along with the indication that it is local to this scope.

## **Abstract Syntax Tree**

We define a class of type AST\_Node, instances of which represent nodes in the AST. The node is composed of the following fields

- parent - a reference to the parent node
- children - a list of children
- construct - the C / grammar construct the node represents
- value - the value held by a terminal/constant
- array - boolean flag indicating whether a terminal is an array or not
- length - length of the array, if the node represents an array

The tree is constructed during parsing. Each occurrence of a C construct corresponds to the addition of a node to the AST. Since the parse is bottom up, we accumulate the children of a node via a stack that is implicit in Ply's parse routine. The accumulated children of a node are added to the parent node once the reduction of the rule corresponding to that construct is made.

### **Intermediate Code Generation**

The intermediate code follows the three address code format and this project uses quadruples for representation of the generated intermediate code. Since the AST maintains structural information of the source code being compiled, it serves in the generation of intermediate code. By parsing the AST in a depth first- left to right manner, the ICG preserves order of occurrence of instructions but can appropriately insert and reorder instructions where necessary. Temporaries are assigned while maintaining continuity of instructions and following order of evaluation. A unique number is allotted to each node, giving unique ids to the temporaries generated in the intermediate code. Branches are handled using labels which are assigned to AST nodes during the AST creation

### **Code Optimisation**

Code optimisations chosen for the scope of this project include - Constant propagation and copy propagation.

#### **Constant propagation :**

For implementation of this optimisation, a lookup table was created. This was a dictionary in python with the key being the variable which is being constant propagated, and the value being its constant RHS. This was done by defining specific rules while parsing the AST which detect assignment productions which have constant values on the right side tree.

Henceforth, all occurrences of the variable on the RHS of an assignment are replaced in place with the value it was assigned to the key in the lookup table.

**Copy propagation :** This type of optimisation is comparable to constant propagation in its implementation as well. Unlike constant propagation, however, copy propagation poses the added challenge of detecting changes to the source of the copy. Like the implementation for constant propagation, a dictionary is used as

a lookup table. The difference is that when entries for the value field of the copied variable are changed in the symbol table, the copy propagation is aborted as the copier variable no longer represents the same value as the source of the original copy.

## Error Handling

The following error handling mechanisms are used to catch and report errors in the input program -


1. Error Productions - We report undeclared variables by querying the hierarchy of symbol tables. If the identifier is not found in any enclosing scope, an error production is called which reports the error along with its corresponding line numbers. We report undeclared variables by querying the hierarchy of symbol tables. If the identifier is not found in any enclosing scope, an error production is called which reports the error along with its corresponding line numbers.
2. The Abstract Syntax Tree, once build is traversed to check for type errors. These errors are caught by querying the symbol table and finding the corresponding type information of the variable. If a mismatch is found, and if a safe type conversion cannot be made, a Type Error is raised.

## SCREENSHOTS



```
1 #include<stdio.h>
2
3 int main()
4 {
5     int a;
6     while(a<5)
7     {
8         a = a+1;
9     }
10 }
```

Source code



```
LABEL 7 :
T1 = a < 5
IF_FALSE T1 GOTO LABEL 3
T2 = a + 1
a = T2
GOTO LABEL 7
LABEL 3 :
```

Intermediate code

```

stmt_list None 2 1
[
  var_declaration None 2 1
  {
    var_type int 0 3
    {
    }
    decl_list None 1 4
    {
      terminal a 0 6
      {
      }
    }
  }
  stmt_list None 1 2
  {
    while None 2 5
    {
      operator < 2 7
      {
        terminal a 0 9
        {
        }
        terminal 5 0 10
        {
        }
      }
      stmt_list None 1 8
      {
        assign = 2 11
        {
          terminal a 0 12
          {
          }
          operator + 2 13
          {
            terminal a 0 14
            {
            }
            terminal 1 0 15
            {
            }
          }
        }
      }
    }
  }
}

```

Abstract syntax tree

```

a x
#include<stdio.h>

int main()
{
  int d,s,b;

  d=d+1;
  //constant propagation
  s=5;
  d=s+1;
  //copy propagation
  b=d;
  f=b+2;
}

```

Source code

```

T1  =  d + 1
d   =  T1
T3  =  5 + 1
d   =  T3
T5  =  d + 2
f   =  T5

```

Optimised code

## **Results**

This project succeeds in producing intermediate code with some optimisations, albeit restricted to the constructs mentioned before- while, do while, expressions, initialisations, function calls, statement blocks, if else blocks etc.

## **Conclusion**

This project saw the implementation of the different phases of a compiler for a limited set of programming constructs. It highlights the challenges of the compilation process and brings to light the complexities of the different processes involved. This project is also a testament to the simplicity of using python-ply over lex-yacc for implementation of the compiler.

## **Further Enhancements**

The scope of this project was limited in that it focused only on some programming constructs and aspects of the compilation process. Further enhancements could be to work on a more general purpose compiler with less of these limits which were set in the interest of time for the completion of the project. The optimisations that were chosen could be expanded upon and the intermediate code could be made to handle a broader set of constructs.

## **References**

Python ply official documentation page:

<https://www.dabeaz.com/ply/>

Python ply official github page:

<https://github.com/dabeaz/ply>