A Project Report

ON

Functional Semantic Analysis of C Program

BY

Aditya Agarwal – 1PI13CS009
Animesh Sahay – 1PI13CS027
Deepak Mahendrakar – 1PI13CS053
Gurleen Singh – 1PI13CS064

Guide
Mrs. Preet Kanwal
Assistant Professor,
PESIT-CSE
Bangalore

August 2015 – Dec 2015

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
PES INSTITUTE OF TECHNOLOGY
(An autonomous institute under VTU)
100 FEET RING ROAD, BANASHANKARI III STATE
BANGALORE-560085

PES Institute of Technology,
(An autonomous institute under VTU)

100 Feet Ring Road, BSK 3rd Stage, Bangalore-560085

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CERTIFICATE

Certified that the Special Topics: Mini Project work entitled **Functional Semantic Analysis of C Program** is a bonafide work carried out by **Deepak Mahendrakar 1PI13CS053, Aditya Agarwal 1PI13CS009, Animesh Sahay 1PI13CS027** and **Gurleen Singh 1PI13CS064** in partial fulfillment for the award of degree of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the academic semester August 2015 to December 2015.

Signature of the Guide Signature of the HOD

Prof. Preet Kanwal **Prof. Nitin V Pujari**

Aditya Agarwal (1PI13CS009)
Animesh Sahay (1PI13CS027)
Deepak Mahendrakar (1PI13CS053)
Gurleen Singh (1PI13CS064)

ABSTRACT

While Syntax Analyzer creates the parse tree, Semantic Analyzer will check actual meaning of the statement parsed in parse tree. Semantic Analysis can compare information in one part of a parse tree to that in another part (e.g. compare reference to variable agrees with its declaration, or that parameters to a function call match the function definition).
Semantic Analysis, also context sensitive analysis, is a process in compiler construction, usually after parsing, to gather necessary information from the source code. The information gathered by the Semantic Analyzer is used for Intermediate Code Generation.
Semantic Analysis usually includes type checking, checking variables must be declared before use.

ACKNOWLEDGEMENT

The successful completion of this task would be incomplete without the mention of people who constantly guided and encouraged us throughout the course of the project. We would like to express our heartfelt thanks to Prof. M.R. Doreswamy, founder of PES Institutions, Prof. D. Jawahar, Dr. K.N. Balasubramanya Murthy, Dr. K Sridhar our Principal for providing us with a congenial environment for carrying out the project. We express our gratitude to Prof. Nitin V Pujari, Head of the Department, Computer Science, PESIT, and our project guide Prof. Preet Kanwal for her constant support and invaluable advice without which this project would not have become a reality.

We are obliged to staff members of PESIT, for the valuable information provided by them in their respective fields. We are grateful for their cooperation period of our project. We thank our friends whose feedback helped us improve the project, and our parents for their unending encouragement and support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

COMPILER

Compiler is a translator program that translates a program written in (High Level Language) the source program and translate it into an equivalent program (Machine Level Language) the target program. During this process, the compiler will also attempt to spot and report obvious programmer mistakes (errors).

COMPILER DESIGN

A Compiler operates in phases. Conceptually, these phases operate in sequence (though in practice, they are often interleaved), each phase (except the first) takes the output from the previous phase as its input. Each phase is a logically interrelated operation that takes source program in one representation and produces output in another representation.

PHASES OF A COMPILER

- Lexical Analysis: Lexical Analyzer of Scanner reads the source program one character at a time, carving the source program into a sequence of automatic units called tokens.
- Syntax Analysis: The second stage of translation is called Syntax Analysis or parsing. In this, phase expressions, statements, declarations etc. are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.
- Intermediate Code Generation: An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.
- Code Optimization: This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.
- Code Generation: The last phase of translation is code generation. A number of optimizations to reduce the length of machine language program are carried out during this phase. The output of code generator is the machine language program of the specified computer.

- Assembly and Linking: The assembly-language code is translated into binary representation and addresses of variables, functions, etc. are determined.

SEMANTIC ANALYSIS

- Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.

CHAPTER 2

PROBLEM DEFINITION

2.1 Problem Definition

A major part of a compilation process is the semantic analysis. Since Semantic Analysis is a process in compiler construction, its job is to gather necessary semantic information from the source code.
The problem statement was to perform semantic analysis of a C program. Some of the cases that needed to be handled were overloading of function declaration, overloading of function definition, function declaration before call, type check of actual parameters against signature, redeclaration of variables within the same scope, variable declaration before assignment, variable type checked against value type and parenthesis matching.
Error message should be raised if the semantics of a C program were violated.

CHAPTER 3

LITERATURE SURVEY

3.1 Literature Survey

This book [1] talks about developing a generating-extension transformation, and describes specialization of the various parts of C, including pointers and structures. The book also investigates separate and incremental program analysis and transformations. Realistic programs are structured into modules, which break down inter-procedural analyses that need global information about functions.

The paper [2] talks about formal semantic basis for the termination analysis of logic programs. It talks about termination of a program P and goal G is determined by the absence of the infinite chain in the binary unfolding of P starting with G. The result is practical use as basing termination analysis on a formal semantics facilitates both the design and implementation of analyzers. The techniques are implemented using a standard CLP® library. The combination of an interpreter for binary unfolding and a constraint solver simplifies the design of the analyzer and improves its efficiency significantly.

CHAPTER 4

PROJECT REQUIREMENT DEFINITION

4.1 Functional Requirements:

Code model for:

- Running lex and yacc using Python's implementation of Ply.
- Graphical User Interface representation using the tkinter module of Python.
- Grammar rules for analyzing the semantics of a C Program.
- Global and local symbol table for storing the variable and function declaration in a global and local scope respectively.

CHAPTER 5

SYSTEM REQUIREMENTS SPECIFICATION

5.1 Software System Requirement:

Python IDLE:

IDLE (Integrated Development Environment or Integrated Development and Learning Environment) is an integrated development environment for **Python**, which has been bundled with the default implementation of the language since 1.5.2b1.

Ply Module (for lex and yacc):

PLY is an implementation of lex and yacc parsing tools for Python. It's implemented entirely in Python. It uses LR-parsing which is reasonably efficient and well suited for larger grammars.

PLY provides most of the standard lex/yacc features including support for empty productions, precedence rules, error recovery, and support for ambiguous grammars. PLY also provides very extensive error checking.

It was developed in 2001 for use in an Introduction to Compilers course where students used it to build a compiler for a simple Pascal-like language.

5.2 Hardware System Requirements

- CPU – 1.33 GHz Quad Core
- RAM – 256 MB (Running simple python programs).
- Hard Disk Space - < 5 MB.

CHAPTER 6

SYSTEM DESIGN

6.1 Phases of a compiler

A compiler operates in phases. A phase is a logically interrelated operation that takes the source program in one representation and produces output in another representation. The phases of a compiler are:

1) Analysis (Machine Independent/Language Dependent)
2) Synthesis (Machine Dependent/Language Dependent)

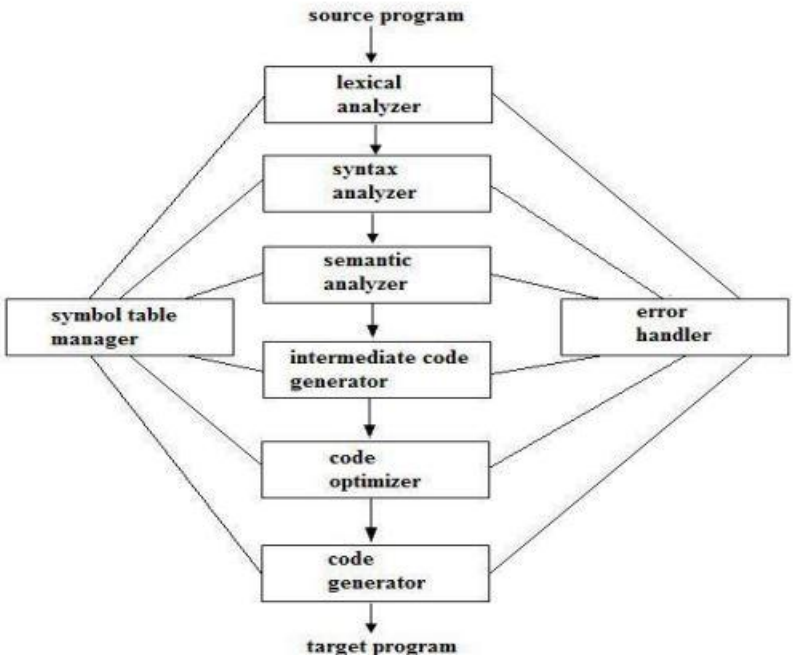

Figure-6.1

As shown in Figure 6.1 lexical analyzer, sytax analyzer, semantic analyzer, intermediate code generator, code optimizer and code optimizer are the phases of a compiler.
In addition to this the compiler also makes use of the symbol table manager and error handler.

6.2 Lex (Ply module)

Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). Lex is commonly used with the yacc parser generator. Lex is a Lexical Analyzer Generator that helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well

suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.
The Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized in our program the corresponding program fragment is executed. The recognition of the expressions is provided by a deterministic finite automata generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.
The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial look ahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.
Lex is a program generator designed for lexical processing of character input streams. It accepts out C program as input, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

6.3 Yacc Module (PLY for Yet another Compiler Compiler)

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.
The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.
Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the

basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

6.4 tkinter module (UI)

Tkinter is Python's de-facto standard GUI (Graphical User Interface) package. It is a thin object-oriented layer on top of Tcl/Tk.
Our program uses the tkinter module to provide the GUI to the user to enter the name of the file and the output gets printed in the console.

CHAPTER 7

7. PSEUDO CODE

Lexer tokens-

```
keywords = {
        'if':'KW_IF',
        'else':'KW_ELSE',
        'while':'KW_WHILE',
        'return':'KW_RETURN',
        'int':'T_INT',
        'float':'T_FLOAT'
    }

operators = {
        '=' : 'OP_EQU',
        '+' : 'OP_PLU',
        '-' : 'OP_MIN',
        '*' : 'OP_MUL',
        '/' : 'OP_DIV',
        '<' : 'RELOP_LT',
        '<=' : 'RELOP_LE',
        '>' : 'RELOP_GT',
        '>=' : 'RELOP_GE',
        '!=' : 'RELOP_NE',
        '==' : 'RELOP_EQ',
        '&&' : 'LOGIC_AND',
        '||' : 'LOGIC_OR'
    }

precedence = (
        ('left','LOGIC_AND','LOGIC_OR'),
        ('left','RELOP_EQ','RELOP_NE'),
        ('left','RELOP_GT','RELOP_LT','RELOP_GE','RELOP_LE'),
        ( 'left', 'OP_PLU', 'OP_MIN' ),
        ( 'left', 'OP_MUL', 'OP_DIV' ),
        ( 'nonassoc', 'UMINUS' )
)
```

```
markers = {
    ';' : 'MK_SC',
    ',' : 'MK_CM',
    '(' : 'MK_LPARAN',
    ')' : 'MK_RPARAN',
    '{' : 'MK_LBRACE',
    '}' : 'MK_RBRACE'
}

Id = r'[a-zA-Z][a-zA-Z0-9]*'
```

Parser grammar-

```
_S : S seen_eof

S : S CONSTRUCT seen_eoc
          | empty

CONSTRUCT : DT ID set_name Y

Y : MK_LPARAN seen_lparan set_isfunc FP MK_RPARAN seen_rparan F
    | VA_G MK_SC make_var_entry

VA_G : OP_EQU EXPR set_v_val
    | empty

F : MK_SC make_func_entry
    | MK_LBRACE seen_lbrace set_isdef make_func_entry reset_make_global_entry STMTS RCALL reset_isdef MK_RBRACE seen_rbrace

FP : P MK_CM FP
    | P

P : DT ID

STMTS : STMT STMTS
    | empty
```

STMT : KW_IF MK_LPARAN seen_lparan EXPR MK_RPARAN seen_rparan MK_LBRACE seen_lbrace STMTS MK_RBRACE seen_rbrace KW_ELSE MK_LBRACE seen_lbrace STMTS

MK_RBRACE seen_rbrace
| KW_WHILE MK_LPARAN seen_lparan EXPR MK_RPARAN seen_rparan MK_LBRACE seen_lbrace STMTS MK_RBRACE seen_rbrace
| DT ID set_is_dec_assignment YB MK_SC
| ID set_is_assignment OP_EQU VA_RHS check_assignment_semantics MK_SC
| FCALL MK_SC

YB : MK_LPARAN seen_lparan set_isfunc FP MK_RPARAN seen_rparan make_func_entry
| VA_L make_var_entry

VA_L : OP_EQU VA_RHS
| empty

VA_RHS : EXPR set_v_val
| FCALL check_assignment_semantics

FCALL : ID set_name set_iscall MK_LPARAN seen_lparan AP MK_RPARAN seen_rparan reset_iscall check_func_call_semantics

AP : AP MK_CM P2
| P2

P2 : ID
| NUM

RCALL : KW_RETURN EXPR MK_SC check_return_semantics
| empty

NUM : NUM_INT
| NUM_FLOAT

DT : T_INT set_type
| T_FLOAT set_type

Expression Grammar:
EXPR : EXPR OP_PLU EXPR
EXPR : EXPR OP_MIN EXPR
EXPR : EXPR OP_MUL EXPR
EXPR : EXPR OP_DIV EXPR
EXPR : EXPR RELOP_LT EXPR
EXPR : EXPR RELOP_LE EXPR
EXPR : EXPR RELOP_GT EXPR
EXPR : EXPR RELOP_GE EXPR
EXPR : EXPR RELOP_NE EXPR
EXPR : EXPR RELOP_EQ EXPR
EXPR : EXPR LOGIC_AND EXPR
EXPR : EXPR LOGIC_OR EXPR
EXPR : MK_LPARAN EXPR MK_RPARAN
EXPR : OP_MIN EXPR %prec UMINUS
EXPR : NUM
EXPR : ID

Errors Handled:

1. ParanMismatchError
2. VariableNotDeclaredError
3. VariableRedeclarationError
4. VariableTypeError
5. FunctionRedeclarationError
6. FunctionNotDeclaredError
7. InvalidOperandError
8. ReturnTypeMismatchError
9. AssignmentTypeMismatchError
10. FunctionOverloadingError

Function parameter check algorithm-
Check func scope in local and global scope
Check func param count with call count
For every actual parameter:
Check if actual parameter variables exist
Check if actual param type matches formal param type

CHAPTER 8

RESULTS DISCUSSION

The project enabled us to get a clear understanding of analyzing the Semantics of a C Program using Python's Ply and Lex Module.
We utilized this understanding to parse the C program which is provided as input and check for its semantic correctness like type checking of variables and return type of variables with method signature.
These functionalities were further enhanced by providing a User Interface using the Python's tkinter module. The user has the ability to upload the C Program for parsing and the appropriate output / error is printed on the screen.

CHAPTER 9

CONCLUSION

To write a program for a compiler for compiling high level Programming Languages like C, Java etc., one needs to program the various phases of a compiler like Syntax Analyser, Semantic Analyser, Lexical Analyser, Intermediate Code Generator etc.

Of these phases Semantic Analyser is an important phase. Semantic Analyzer will check actual meaning of the statement parsed in parse tree. After parsing, it gathers information from parsing that will be used by the intermediate code generator.

Semantic Analysis of a C Program involves checking for types of variables, matching return types of function against method signature, matching parenthesis etc.

CHAPTER 10
FURTHER ENHANCEMENTS

10.1 Handling Return type other than int and float

The program handles int and float as the return type of functions. The program can be modified to handle float and int as the return type. The formal parameters must also be declared int or float.

10.2 Variable Declarations

The program works for only one declaration of a variable per line in global space. The program can be modified to check for multiple declaration of variables in a line.

10.3 Type of Variables

The program works for int and float for the type of variables. It can be modified to handle other types as well.

CHAPTER 11

BIBLIOGRAPHY

11.1 Publications and Web Pages Referenced

1. Developing a generating-extension transformation, and describes specialization of the various parts of C
2. Formal semantic basis for the termination analysis of logic programs - http://www.sciencedirect.com/science/article/pii/S0743106699000060
3. Compiler Design Textbook - http://www.svecw.edu.in/Docs%5CCSECDLNotes2013.pdf
4. Python's Implementation of lex and yacc - http://www.dabeaz.com/ply/
5. Python's tkinter module - http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html