

***Report on***

**“C++ Mini Compiler for ‘while’, ‘for’, ‘if’, ‘if-else’ constructs”**

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

***Compiler Design Laboratory***

**Bachelor of Technology**

**in**

**Computer Science & Engineering**

***Submitted by:***

|  |  |
| --- | --- |
| **Lamya Bhasin**  **Sujeeth V**  **Pratik B** | **PES1201701244**  **PES1201700958**  **PES1201700272** |

*Under the guidance of*

|  |
| --- |
| **Kiran P**  Professor  PES University, Bengaluru |

**January – May 2020**

**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.** |
|  | **INTRODUCTION (Mini-Compiler is built for C++ language. Provide sample input and output of your project)** | **01** |
|  | **ARCHITECTURE OF LANGUAGE:**   * **What all have you handled in terms of syntax and semantics for the chosen language.** | **02** |
|  | **LITERATURE SURVEY (if any paper referred or link used)** | **03** |
|  | **CONTEXT FREE GRAMMAR (which you used to implement your project)** |  |
|  | **DESIGN STRATEGY (used to implement the following)**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** * **TARGET CODE GENERATION** |  |
|  | **IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following):**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE (internal representation)** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ASSEMBLY CODE GENERATION** * **ERROR HANDLING *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** * **Provide instructions on how to build and run your program.** |  |
|  | **RESULTS AND possible shortcomings of your Mini-Compiler** |  |
|  | **SNAPSHOTS (of different outputs)** |  |
|  | **CONCLUSIONS** |  |
|  | **FURTHER ENHANCEMENTS** |  |
| **REFERENCES/BIBLIOGRAPHY** | |  |

**CHAPTER 1: INTRODUCTION**

C++ was the language chosen as the basis of this mini compiler. Simple constructs from the

language were implemented. The frontend of the compiler including Symbol table

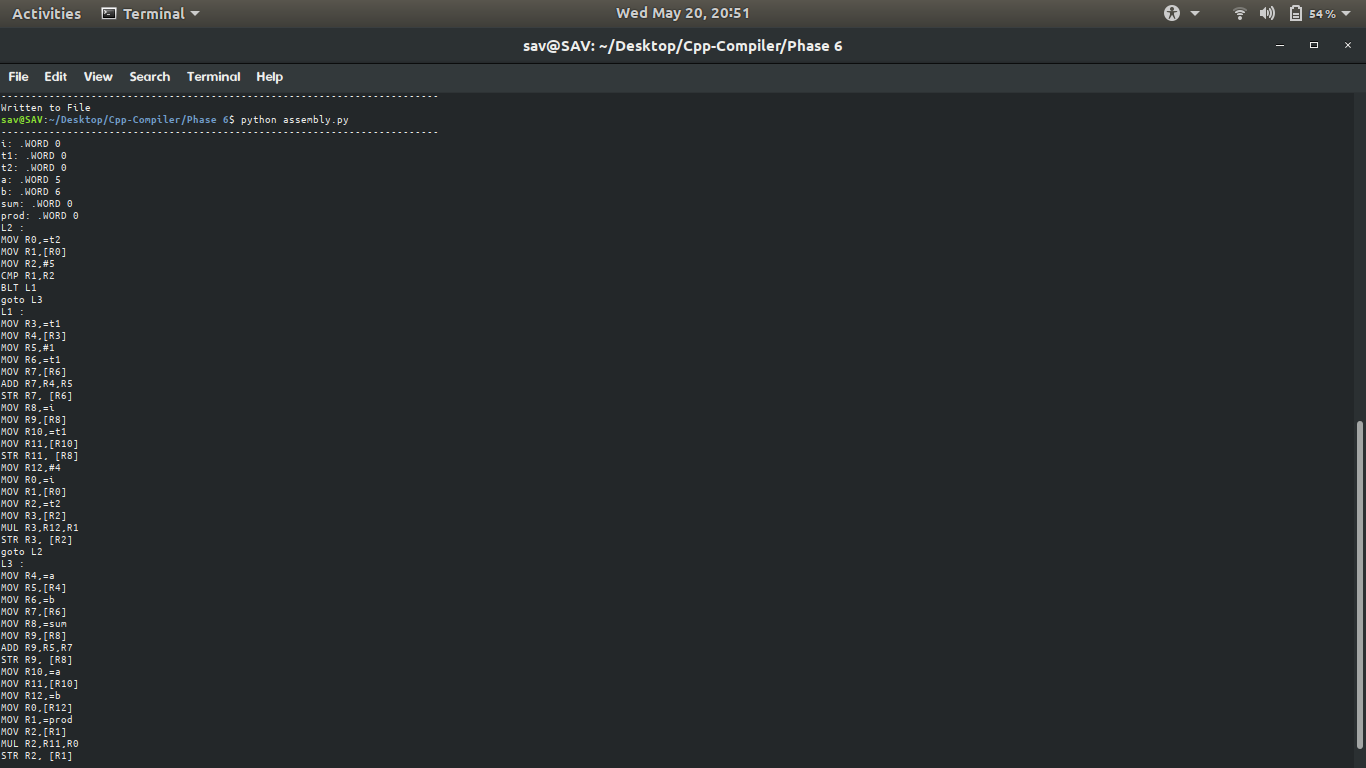
generation, Abstract Syntax tree construction, Intermediate Code generation and Code

Optimization was implemented using lex and yacc. The language used to design the compiler is C.

Sample input:



Sample output:



Compiler for the following constructs:

do while​ loop

1. while loop
2. for loop
3. if statement
4. if else statements

Given source program in C can be translated to a symbol table, abstract syntax tree, intermediate code, and optimized intermediate code.

**CHAPTER 2. ARCHITECTURE**

* -  Used lex to create the scanner for our language.
* -  Used yacc to implement grammar rules to the token generated in the  scanner phase.
* The following are the operators and special characters implemented in our

programming language:

* -  Binary operators: + - \* / %
* -  Unary operators: ! ~ + - ++  and prefix)

- shorthand operators: +=, -=, \*=, /=

* -  Ignore comments and white-spaces
  1. -  Single line comments starting with //
  2. -  Multi-line comments enclosed within /\* ...... \*/
* -  Types: int, float, char, double, long
* -  Constructs ‘for’ and ‘while’ loop and ‘if’ and ‘if-else’
* -  Includes functions definitions and function calls.
* -  No conflicts and errors in our code/grammar.
* -  Error handling related to scope and declaration.
* Warnings and Error recovery:
* Errors:

- Redefinition of identifiers within the same scope

- Use of undeclared identifiers

- Invalid operands to ‘%’. Implicit conversion to INT

- Syntax errors based on the specified grammar.

* -  All the errors and warnings are displayed along with line number
* -  scenario when same variable name is used is handled
* -  Code Optimizations techniques used :
  1. -  Common Subexpression elimination
  2. -  Constant folding
  3. -  Dead Code Elimination
* assembly code generation: line by line arm command generation using conditional statement if-else and helper functions to identify the labels and operations.

**CHAPTER 3. LITERATURE SURVEY**

* 3.1 Course material shared for Compiler Design Course (especially ICG and Code optimisation)
* 3.2 <https://www.tldp.org/HOWTO/Lex-YACC-HOWTO.html#toc1> - Working of lex and yacc

3.3 https://www.lysator.liu.se/c/ANSI-C-grammar-y.html - Helped us write the grammar for our compiler

3.4 https://softwareengineering.stackexchange.com/questions/165543/how-to-write-a-ver y-basic-compiler - A reference link for writing the code and taking ideas.

* **CHAPTER 4. CONTEXT-FREE GRAMMAR**

|  |
| --- |
| * main\_block : declaration main\_x '(' ')' multiple\_expressions * ; * body : '{' multiple\_expressions '}' body * | '{' multiple\_expressions '}' * | expression * | logical\_expression * | if * | error ';' * | for * | while * | printf * | '{' '}' * | return\_x '0' ';' * ; |

if: if\_x '(' extended\_logical\_expression ')' body

;

for : for\_x '(' assignment\_expression\_with\_null ';' extended\_logical\_expression\_with\_null ';' extended\_logical\_expression\_with\_null ')' body

;

while : while\_x '(' logical\_expression ')' body

| while\_x '(' expression ')' body

;

extended\_logical\_expression\_with\_null:extended\_logical\_expression

|

;

assignment\_expression\_with\_null:assignment\_expression

|

;

multiple\_expressions: logical\_expression ';' multiple\_expressions

// | expression ';' multiple\_expressions

| multiple\_expressions multiple\_expressions

| body

;

logical\_expression: comparision\_expression logicalop extended\_logical\_expression

| assignment\_expression logicalop extended\_logical\_expression

| RHS logicalop extended\_logical\_expression

;

extended\_logical\_expression:logical\_expression

| comparision\_expression

| assignment\_expression

| RHS

;

expression: comparision\_expression

| assignment\_expression ';'

| declaration\_statement ';'

| RHS

;

comparision\_expression:RHS comparisionop RHS

;

assignment\_expression: declaration\_statement assignmentop RHS

| identifier assignmentop RHS

;

declaration\_statement: declaration identifier

| declaration\_statement ',' pointers identifier

;

printf: printf\_x '(' '"' identifier '"' ')' ';'

| printf\_x '(' '"' identifier '"' multiple\_ids ')' ';'

;

multiple\_ids: ',' identifier multiple\_ids

| ',' identifier

;

pointers : pointers '\*'

|

;

RHS: '(' RHS ')'

| RHS '+' RHS

| RHS '-' RHS

| RHS '/' RHS

| RHS '\*' RHS

| RHS '%' RHS

| F

;

F: identifier

| '\*' identifier

| '&' identifier

| '~' identifier

| '!' identifier

| identifier unary

| unary identifier

| number

;

**CHAPTER 5. DESIGN AND IMPLEMENTATION**

-  Symbol Table : Symbol table is a data structure that tracks the current bindings of identifiers for performing semantic checks and generating code efficiently. We have implemented the symbol table as an array of structures. The members of the structures include variable name, scope, data type and value. Every new token encountered in the program is entered into the symbol table.

* Abstract Syntax Tree : This tree is constructed as the input is parsed. Each node of this tree contains a pointer to left, a pointer to right and a member for a string.
  + 1. -  Intermediate Code Generation : The intermediate code is generated on the fly , as we parse the code and check its grammar , the intermediate code is generated. The representation of ICG is done by using quadruple table. The quadruple table consists of operator, first parameter, second parameter and the result.
* -  Code Optimization : To increase efficiency the code optimization is done on the generated ICG. We have implemented constant folding, common subexpression elimination and dead code elimination.
* - Assembly Code: The optimised code is read from its file. For each line the appropriate arm code is generated using simple if-else statements and helper functions. Thereby, the corresponding assembly code is generated and written to the assembly code file (.s file).
* -  Error Handling : In case of syntax error, the compilation is halted, and an error message along with the line number where error occured is displayed. Semantic errors such as division by zero, multiple declaration of the same variable, invalid assignment, scope errors are also explicitly pointed out. All of which are specified as production rules within the grammar. Type checking is done and if the types do not match in the expressions, a warning is displayed and implicit conversion is done to recover from errors.

**CHAPTER 6. IMPLEMENTATION**

* -  The tools we have used for implementing the code are lex and yacc.
* -  The lex file has all the tokens specified with the help of regular expressions  and the yacc file has grammar rules with corresponding actions.
* -  As the code is being parsed, the tokens are generated and comments and extra  spaces are ignored. For every new variable encountered, it is entered into the  symbol table along with its attributes.
* -  Semantics Analysis uses available information in the table to check for  semantics i.e. to verify that expressions and assignments are semantically  correct ( type checking ) and update it accordingly.
* -  The scope check is done by having a variable which increments on every  level of nesting. In this manner, the scope is checked for each variable and  error messages are displayed if anything is used out of scope.
* -  Integer, float and char literal constants.
* -  Variables must be declared and can only be used in ways that are acceptable  for the declared type.
* -  Once parsing is successful, we generate an abstract tree and it is shown in  pre-order manner.
* -  The intermediate code generation also happens on the fly.
* -  After generating intermediate code, optimization is done by doing dead code  elimination, constant folding and common subexpression elimination.
* -  Code optimisation uses information present in symbol table for machine  dependent optimization.
  + - 1. - Assembly code is generated by reading the optimised code from its file, splitting the lines, identifying the type of labels and functions by using simple if-else statements and generating the appropriate ARM commands for each of the lines and thereby generating the entire assembly code.

Commands to execute the code:

lex plex.l

yacc -d plex.y

gcc lex.yy.c y.tab.c -ll -ly -o plex.o

lex lexicalanalyzer.l

yacc -d parser.y

gcc lex.yy.c y.tab.c -ll -ly -o lexicalanalyzer.o

./plex.o < input.cpp

./plex.o < input.cpp

python optimize.py icg.txt

**CHAPTER 7. RESULTS and possible shortcomings**

* + - 1. A mini compiler that can compile the chosen constructs was obtained.
* Shortcomings are that time taken could be improved.
* **CHAPTER 8. SCREENSHOTS of various outputs**
* Symbol table creation from input
* 
* AST creation from given input
* 
* 
* 
* - optimisations

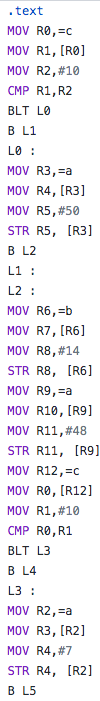


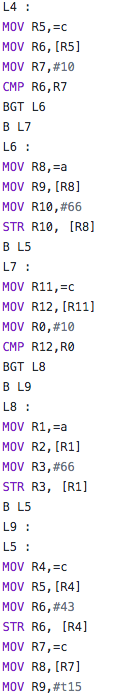


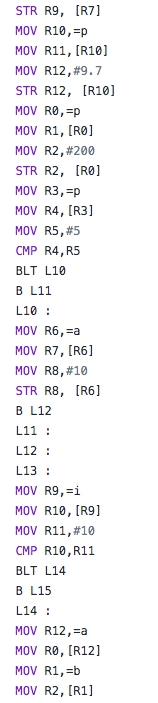




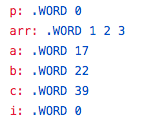
* assembly code output







****

****

**CHAPTER 9. CONCLUSIONS**

A compiler for C++ was thus created using lex and yacc.In addition to the looping and conditional constructs specified, basic building blocks of the language (functions, declaration

statements, assignment statements, etc) were handled.

This compiler was built keeping the various stages of Compiler Design, ie, Lexical Analysis, Syntax Analysis, Semantic Analysis and Code Optimisation in mind.

As a part of each stage, an auxillary part of the compiler was built (Symbol Table, Abstract Syntax Tree and Intermediate Code). Each of these components are required to compile code successfully.

In addition to this, basic error recovery has also been implemented.

Through this process, all kinds of syntax errors and certain semantic errors in a C++ program can be caught by the compiler.

Optimizations to improve this have also been implemented.

Assembly code generated by generating the appropriate arm codes.

**CHAPTER 10. FURTHUR ENHANCEMENTS**

**-** Functionality for other looping and conditional constructs can be implemented.

* - Conditional jumps like goto, continue and break.
* -  More data types can be accomodated.

- The compiler can be constructed to recover from more kinds of errors

**REFERENCES**

* a. Compilers – Principles, Techniques, and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
* b. https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-des ign/
* c. http://web.cs.wpi.edu/~kal/courses/compilers/
* d.https://www.tutorialspoint.com/compiler\_design/compiler\_design\_intermedia  te\_c ode\_generations.htm