Report on

"Mini C Compiler(Looping Constructs)"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Akshaya J	**PES2201800013**
Jigya Shah	**PES2201800417**
Samyuktha Prakash	**PES2201800470**

Under the guidance of

Prajwala T. R.
Professor
PES University, Bengaluru

January – May 2021

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

1. Introduction

This project creates a mini C compiler specifically for looping constructs like for, while and do-while. Expressions are also handled by the compiler

The input is a c code and the output is an optimised code in 3 address code format.
Input:

```
void main()
{
        int i;
        do
        {
                i=2+3;
                i++;
        }while(i<10);
}
```

Output:

```
jigya@jigya-VirtualBox:~/Desktop/mini-C-compiler-master/Phase2$ ./a.out <tests/test7.c
func begin main
L0:
= 2   t0
= t0   i
= 3   t1
= t1   i
+ t0 t1 t2
= t2   i
+ i 1 t3
= t3   i
= 10   t4
= t4   i
< i t4 t5
ifnot t5 goto L1
goto L0:
L1:
func end

Status: Parsing Complete - Valid
                              SYMBOL TABLE
                              ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO |
-------------------------------------------------------------------------------------------------
         i |      Identifier |        int |         10 |          3 |
        do |         Keyword |            |            |          4 |
        t0 |       Temporary |            |            |          6 |
        t1 |       Temporary |            |            |          6 |
        t2 |       Temporary |            |            |          6 |
        t3 |       Temporary |            |            |          7 |
       int |         Keyword |            |            |          3 |
        t4 |       Temporary |            |            |          0 |
        t5 |       Temporary |            |            |          0 |
      main |        Function |       void |            |          1 |
     while |         Keyword |            |            |          8 |
      void |         Keyword |            |            |          1 |
        10 | Number Constant |            |            |            |
         2 | Number Constant |            |            |            |
         3 | Number Constant |            |            |            |
```

2. Architecture of the Language

Syntax Analyser or parser verifies that a string of token names can be generated by the grammar of the source language. The parser is expected to report any syntax errors in an intelligible manner and to recover from the commonly occurring errors to continue processing the remainder of the program. It detects the following types of errors:

1. Errors in structure
2. Missing operator
3. Misspelt keywords
4. Unbalanced parenthesis
5. Missing semicolon

Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not. Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other.

Semantic analysis typically involves the following tasks:

1. Type Checking - Data types are used in a manner that is consistent with their definition (i. e., only with compatible data types, only with operations that are defined for them, etc.)
2. Label Checking - Labels references in a program must exist.
3. Duplicate Checking - Checks if duplicate identifiers are defined

3. Context Free Grammar

```
program
: function_declaration;

variable_declaration
: type_specifier variable_declaration_list ';';

variable_declaration_list
: variable_declaration_list ',' variable_declaration_identifier | variable_declaration_identifier;

variable_declaration_identifier
: identifier  vdi;


vdi : ASSIGN_OP simple_expression | ;

type_specifier
```

```
: INT | CHAR | FLOAT  | DOUBLE
| LONG long_grammar
| SHORT short_grammar
| UNSIGNED unsigned_grammar
| SIGNED signed_grammar
| VOID
;

unsigned_grammar
: INT | LONG long_grammar | SHORT short_grammar | ;

signed_grammar
: INT | LONG long_grammar | SHORT short_grammar | ;

long_grammar
: INT  | ;

short_grammar
: INT | ;

function_declaration
: function_declaration_type;

function_declaration_type
: type_specifier MAIN '('  ')' statement  ;

statement
: expression_statment | compound_statement
| iterative_statements
| return_statement | break_statement
| variable_declaration

compound_statement
: '{'  statment_list  '}' ;

statment_list
: statement statment_list
| statement error ';'
| ;

expression_statment
: expression ';'
| ';'
| error ';'

iterative_statements
: WHILE '(' simple_expression ')' statement
| FOR '(' expression ';' simple_expression ';' expression ')'statement
| DO statement WHILE '(' simple_expression ')' ';';
```

```
return_statement
: RETURN ';' | RETURN expression ';'


break_statement
: BREAK ';';

expression
: mutable ASSIGN_OP expression
| mutable ADD_ASSIGN_OP expression
| mutable SUB_ASSIGN_OP  expression
| mutable MUL_ASSIGN_OP  expression
| mutable DIV_ASSIGN_OP expression
| mutable MOD_ASSIGN_OP expression
| mutable INC_OP
| mutable DEC_OP
| INC_OP mutable
| DEC_OP mutable
| simple_expression ;


simple_expression
: simple_expression OR_OP and_expression
| and_expression ;

and_expression
: and_expression AND_OP unary_relation_expression
| unary_relation_expression;


unary_relation_expression
: exclamation_OP unary_relation_expression
| regular_expression ;

regular_expression
: regular_expression relational_OPs sum_expression }
 | sum_expression  ;

relational_OPs
: GT_ASSIGN_OP  | LT_ASSIGN_OP  | GT_OP | LT_OP | EQ_OP | NE_OP  ;

sum_expression
: sum_expression sum_OPs term
| term;

sum_OPs
: ADD_OP
| SUB_OP ;
```

```
term
: term MULOP factor
| factor ;

MULOP
: MUL_OP | DIV_OP | MOD_OP ;

factor
: immutable
| mutable ;

mutable
: identifier;

immutable
: '(' expression ')'
| constant ;


constant
: integer_constant
| string_constant
| float_constant
| character_constant;
```

4. Design Strategy

A. Symbol Table Creation:

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities.

In this project, a simple symbol table is created having SYMBOL, CLASS, TYPE, VALUE and LINENO as columns.

The lexer file has the symbol table creation.

The symbol table has the identifiers, function name,temporary variables, keywords and constants.

B. Intermediate Code Generation

Intermediate Code Generation phase is the glue between the frontend and backend of the compiler design stages.

Many compilers use a medium-level language as a stepping-stone between the high-level language and the very low-level machine code. Such stepping-stone languages are called intermediate code. It provides lower abstraction from source level and maintains some high level information. Intermediate Code can be represented in many different formats depending whether it is language-specific or language-independent.

Most common independent intermediate representations are:

1. Postfix notation
2. Three Address Code
3. Syntax tree

In this project, the intermediate code is generated in **three address code format**.

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code.
It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in a temporary variable generated by the compiler.
The compiler decides the order of operation given by three address code.

C. Code Optimisation

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory).

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

1. The output code must not, in any way, change the meaning of the program.

2. Optimization should increase the speed of the program and if possible, the program should demand less number of resources.

3. Optimization should itself be fast and should not delay the overall compiling process.

There are four code optimisation strategies applied:

a. **Constant Folding:**
   Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime

b. **Constant Propagation**
   Constant propagation is the process of substituting the values of known constants in expressions. Constant propagation eliminates cases in which values are copied from one location or variable to another, in order to simply assign their value to another variable.

c. **Common Subexpression Elimination**
   common subexpression elimination (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value.

d. **Dead Code Elimination**
   dead code elimination is a compiler optimization to remove code which does not affect the program results

D. Error Handling

Multiple errors are handled, that is all the errors in the code are shown and parsing doesn't stop with only one error.

Some of the semantics errors handled are:

1. Type mismatch
   a. Return type mismatch.
   b. Operations on mismatching variable types.
2. Undeclared variable
   a. Check if a variable is undeclared globally.
   b. Check if a variable is visible in the current scope.
3. Reserved identifier misuse.
   a. Function name and variable name cannot be the same.
   b. Declaration of keyword as variable name.
4. Multiple declaration of a variable in a scope.
5. Duplicate declaration of identifiers are handled
6. Accessing an out of scope variable.

5. Implementation Details

A. Symbol Table Creation

A symbol table is generated in the Lexical Analyzer stage.
A hashing technique was used to implement the symbol table i.e. if there is a collision, then after the point of collision, the table is searched linearly in order to find an empty slot.
Two structures were maintained; one for symbol table and the other for constant table. Both the symbol table and the constant table were merged and displayed together as one.

B. Intermediate Code Generation

In order to generate 3 address code, an explicit stack was used.
Whenever an operator, operand or a constant was encountered, it was pushed to the stack.
Whenever reduction occurred, the codegen( ) function generated the 3 address code by creating a new temporary variable and by making use of the entries in the stack. After that it popped those entries from the stack and pushed the temporary variable to the stack so that it gets used in further computation

C. Code Optimisation

The code for code optimisation was implemented using the python programming language.

The Constant Folding and Constant Propagation takes the input from the file generated after the implementation of the intermediate code generation.
Hence, it takes input from icg.txt and gives the output through a file named cfg.txt

The Constant Subexpression Elimination also takes input from icg.txt and gives the output through a file named cse.txt.
Subexpressions which are repeated are removed and the temporary variable storing the eliminated subexpression is substituted later during the code whenever encountered.

The dead code elimination takes input from the cfp.txt and gives output as dce.txt.This eliminates the extra lines which have been stored in temporary variables.

D. Error Handling

Multi-line error recovery mechanism is handled using predefined functions in yacc like yyerrok() and predefined keywords like error.

The mechanism used to implement multi-line error recovery is a simple straightforward one, which makes use of a synchronising token (i.e. the semicolon ;).

E. Instruction to Build and Run the Code:

```
lex lexer.l
yacc -d parser.y
gcc y.tab.c lex.yy.c -ll -w
./a.out <testcase_file>
python cfp.py
python cse.py
python dce.py
```

The optimised code will be present in the dce.txt file.
After each of the optimisations the output is stored in a file.
The code is runs on Ubuntu

6. Results

The output is an optimized code that can only compile simple expressions, loops and if-else conditional statements written in C. The intermediate code is generated in three address code format, and given below are the limitations of our project.

Shortcomings:

a. This mini C compiler only handles looping constructs such as for loops, while loops and do-while loops. It does not handle switch cases and so on.
b. The dead code elimination only deals with removing unused temporary variables.
c. Global optimisation techniques are not implemented.
d. Existence of a reduce-reduce conflict.

7. Snapshots

Test1: for and while loop

```
jigya@jigya-VirtualBox:~/Desktop/mini-C-compiler-master/Phase2$ ./a.out <tests/test1.c
func begin main
= 0   t0
= t0   i
= t0   i
L0:
< i n t1
ifnot t1 goto L1
+ i 1 t2
= t2   i
L2:
= 10   t3
= t3   x
< x t3 t4
ifnot t4 goto L3
+ x 1 t5
= t5   x
goto L2:
L3:
goto L0:
L1:
func end

Status: Parsing Complete - Valid
                            SYMBOL TABLE
                            ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO |
-------------------------------------------------------------------------------------------------
         i |      Identifier |        int |          0 |          2 |
         n |      Identifier |        int |            |          2 |
         x |      Identifier |        int |         10 |          6 |
       for |         Keyword |            |            |          5 |
      char |         Keyword |            |            |          3 |
        ch |      Identifier |       char |            |          3 |
        t0 |       Temporary |            |            |          5 |
        t1 |       Temporary |            |            |          5 |
        t2 |       Temporary |            |            |          5 |
        t3 |       Temporary |            |            |          7 |
       int |         Keyword |            |            |          2 |
        t4 |       Temporary |            |            |          0 |
        t5 |       Temporary |            |            |          0 |
      main |        Function |       void |            |          1 |
     while |         Keyword |            |            |          7 |
      void |         Keyword |            |            |          1 |
        10 | Number Constant |            |            |            |
         0 | Number Constant |            |            |          _ |
```

dce.txt

```
i = 0
i = 0
L0: t1 = i<n
ifnot t1 goto L1
t2 = i+1
i = t2
L2: t3 = 10
x = 10
ifnot t4 goto L3
x = 11
goto L2: L3: goto L0: L1:


Eliminated 3 lines of code
```

Test 2: Simple operations

```
jigya@jigya-VirtualBox:~/Desktop/mini-C-compiler-master/Phase2$ ./a.out <tests/test2.c
func begin main
= 6   t0
= t0   a
= 2   t1
= t1   b
+ a b t2
= t2   c
* a b t3
= t3   d
/ a b t4
= t4   e
% a b t5
= t5   f
* b c t6
+ a t6 t7
= t7   g
func end

Status: Parsing Complete - Valid
                            SYMBOL TABLE
                            ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO |
-------------------------------------------------------------------------------------------------
         a |      Identifier |        int |          6 |          2 |
         b |      Identifier |        int |          2 |          2 |
         c |      Identifier |        int |            |          2 |
         d |      Identifier |        int |            |          2 |
         e |      Identifier |        int |            |          2 |
         f |      Identifier |        int |            |          2 |
         g |      Identifier |        int |            |          2 |
        t0 |       Temporary |            |            |          2 |
        t1 |       Temporary |            |            |          2 |
        t2 |       Temporary |            |            |          4 |
        t3 |       Temporary |            |            |          5 |
       int |         Keyword |            |            |          2 |
        t4 |       Temporary |            |            |          0 |
        t5 |       Temporary |            |            |          0 |
        t6 |       Temporary |            |            |          0 |
        t7 |       Temporary |            |            |          0 |
      main |        Function |       void |            |          1 |
      void |         Keyword |            |            |          1 |
         2 | Number Constant |            |            |            |
         6 | Number Constant |            |            |            |
```

dce.txt

```
a = 6
b = 2
c = 8
d = 12
e = 3.0
f = 3.0
g = 22


Eliminated 8 lines of code
```

Test 3: do-while loop

```
jigya@jigya-VirtualBox:~/Desktop/mini-C-compiler-master/Phase2$ ./a.out <tests/test3.c
func begin main
+ a b t0
= t0   c
+ a b t1
= t1   d
L0:
+ a 1 t2
= t2   a
= 10   t3
= t3   a
< a t3 t4
ifnot t4 goto L1
goto L0:
L1:
func end

Status: Parsing Complete - Valid
                            SYMBOL TABLE
                            ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO |
-------------------------------------------------------------------------------------------------
         a |      Identifier |        int |         10 |          4 |
         b |      Identifier |        int |            |          4 |
         c |      Identifier |        int |            |          4 |
         d |      Identifier |        int |            |          4 |
        do |         Keyword |            |            |          7 |
        t0 |       Temporary |            |            |          5 |
        t1 |       Temporary |            |            |          6 |
        t2 |       Temporary |            |            |          8 |
        t3 |       Temporary |            |            |          9 |
       int |         Keyword |            |            |          2 |
        t4 |       Temporary |            |            |          0 |
      main |        Function |        int |            |          2 |
     while |         Keyword |            |            |          9 |
        10 | Number Constant |            |            |            |
```

cse.txt

```
t0 = a+b
c=t0
t1 = t0
d=t1
L0: t2 = a+1
a=t2
t3=10
a=t3
t4 = a<t3
ifnot t4 goto L1
goto L0: L1:
```

dce.txt

```
t0 = a+b
c = t0
t1 = a+b
d = t1
L0: t2 = a+1
a = t2
a = 10
ifnot t4 goto L1
goto L0: L1:


Eliminated 2 lines of code
```

Test 4: function should be ‘main’

```
jigya@jigya-VirtualBox:~/Desktop/mini-C-compiler-master/Phase2$ ./a.out <tests/test4.c
syntax error at lineno: 1 at character: fun
Status: Parsing Failed - Invalid
```

Test 5: some Semantic errors

```
jigya@jigya-VirtualBox:~/Desktop/mini-C-compiler-master/Phase2$ ./a.out <tests/test5.c
func begin main
L0:
a
Undeclared
= 0   t0
= t0   a
> a t0 t1
ifnot t1 goto L1
Condition checking is not of type int
if
Undeclared
syntax error at lineno: 7 at character: (
Status: Parsing Failed - Invalid
b
Undeclared
Type mismatch
```

Test 6:Error showing multi line errors and switch not implemented

```
jigya@jigya-VirtualBox:~/Desktop/mini-C-compiler-master/Phase2$ ./a.out <tests/test6.c
func begin main
= 0   t0
= t0   i
= t0   i
L0:
= 10   t1
= t1   i
< i t1 t2
ifnot t2 goto L1
= 10   t3
= t3   j
< j t3 t4
syntax error at lineno: 4 at character: ;
Status: Parsing Failed - Invalid
syntax error at lineno: 4 at character: )
Status: Parsing Failed - Invalid
syntax error at lineno: 8 at character: switch
Status: Parsing Failed - Invalid
```

8. Conclusions

This project is written for a C compiler with only loops and simple expressions.
In the course of this project, the lexer phase was implemented first followed by the parser phase.
The context free grammar then generates the intermediate code, which is further optimised using some local optimisation techniques.

9. Further Enhancements

Our project handles only the main function as of now so we can further take user defined functions.Functions like printf doesn't work so we have to integrate the preprocessor directives for various inbuilt functions to work.