Report on

C Compiler

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Atmik Ajoy	**PES1201800189**
Chethan U Mahindrakar	**PES1201801126**
Dhanya Gowrish	**PES1201800965**

Under the guidance of

Prof Preet Kanwal
Associate 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 Compiler has been developed for the **C programming language**

- **Input to compiler:**
  - C program
  - May start with optional header file declaration

```c
#include <stdio.h>

int main()
{
	int a = 20;
	int b=3;

	while(a<50)
	{

		printf("in while");
		a=a+1;

	}



	return 0;

	b=100;

}
```

- **Output of Compiler:**
  - Symbol table with the required information
  - 3 Address Code of the program
  - 3 Address Code in Quadruples format
  - Optimized code (on enabling optimization)

```
a = 20

b = 3

L1:

t0 = a < 50
iffalse t0 goto L2
t1 = a + 1

a = t1

goto L1

L2:

b = 100

Statements after line number 19 is DEAD CODE
Parsing done

------------------------------------------------Symbol Table------------------------------------------------

Sl.No   Identifier      Scope           Value           Type            Storage
-------------------------------------------------------------------------------------------------------------

1       a               1               1               INT             4
2       b               1               100             INT             4
3       main            0               0               MAIN FUNCTION - INT             0
-------------------------------------------------------------------------------------------------------------


 OP             ARG1            ARG2            RESULT

=               20              NULL            a
=               3               NULL            b
Label           NULL            NULL            L1
<               a               50              t0
iffalse         t0              NULL            L2
+               a               1               t1
=               t1              NULL            a
goto            NULL            NULL            L1
=               100             NULL            b
```

2. ARCHITECTURE OF LANGUAGE

- Procedural Language
- General Purpose Language
- Statically typed language
- Middle-Level Language
- It is highly portable as programs written in C can run on any system with close to none changes
- It is easy to extend in the sense that it is easy to add and remove functionality to and from.
- Can include multiple header files and gain functionality from them
- Implements BODMAS for arithmetic expression evaluation
- Starts execution of the program from main function , main function is mandatory

3.CONTEXT FREE GRAMMAR

```
start : Function start
        | PREPROC start
        | Declaration start
        |
        ;

Function : Type ID '('')' compound_statement
        ;



Type : INT
        | FLOAT
        | VOID
        ;

compound_statement : '{' stmt '}'
        ;

stmt : Declaration stmt
        | while stmt
        | RETURN consttype ';' stmt

        | RETURN ';'
```

```
        | ';'
        | PRINT '(' STRING ')' ';' stmt
        | ID INC';' stmt
        | ID DEC';' stmt
        | INC ID';' stmt
        | DEC ID';' stmt
        | switch stmt
        | compound_statement stmt
        |
        ;

switch : SWITCH'('ID')' '{'case'}'
        ;

case : CASE consttype ':'stmt break case
        | default
      ;



break : BREAK';'
      |
      ;

default : DEFAULT ':'stmt break
        |
        ;
while : WHILE '(' E ')' compound_statement
        ;
```

```
assignment1 : ID '=' E
        | ID ',' assignment1
        | consttype ',' assignment1
        | ID
        | consttype ;
consttype : NUM
        | REAL
        ;

Declaration : Type ID '=' E ';'
        | assignment1 ';'
        | Type ID ';'
        | Type ID '[' consttype ']' ';'
        | ID '[' assignment1 ']' ';'
        | error
        ;

array : ID '[' E ']'
        ;

E : E '+' T
  | E '-' T
  | T
  | ID E
  | ID GE  E
  | ID EQ E
  | ID NEQ E
  | ID AND E
  | ID OR E
```

```
    | ID '<' E
    | ID '>' E
    | ID '=' E
    | array
    ;
T : T '*' F
    | T '/' F
    | F
    ;
F : '(' E ')'
    | ID
    | consttype
    ;
```

4. DESIGN STRATEGY

SYMBOL TABLE CREATION

- Symbol table is generated for the program:
    - The symbol table contains the following fields:
        - Identifier Name
        - Scope of the identifier
        - Value associated with identifier
        - Type of the identifier
        - Storage required for the type
- The symbol table is represented as an array of structures.
- Each entry in the symbol table is associated with an element in the array of structures.
- We use the array of structures to represent multiple identifiers

INTERMEDIATE CODE GENERATION

- The Intermediate Code Representation implemented is **Three Address Code (3AC).**
- The 3AC generated is also represented in Quadruples format

- Optimized 3AC is also represented
- Appropriate actions are included in the grammar, to generate 3AC
- 3AC code generation for arithmetic and boolean expressions uses stack for generation
- Quadruple created for every 3AC stored in seperate array of structures (dedicated for quadruples)

CODE OPTIMIZATION

- **Constant Propagation:**
  - In an arithmetic expression, when an identifier is encountered on the right hand side, constant propagation is implemented
  - The approach involves searching the symbol table for the concerned identifier, in descending order of scope i.e most recent scope
  - identifiers are stored in symbol table and lookup is done on array of structures

- **Constant Folding:**
  - In an arithmetic expression, constant folding is implemented by evaluating every sub-expression within the expression in real time
  - The folded value is updated in the symbol table

- **Identification of Loop Invariant Code:**

    - Loop invariant code is identified by checking relevant loop variant variables.
    - A 2 dimensional character array is used to store the loop-dependant variables

- **Dead Code Identification**
    - Removal of statements after return
    - If the main() function does not return any value or does not print anything, the entire program is labelled as dead code

ERROR HANDLING:

- Panic Mode Recovery is performed in scanner, when a character does not match any pattern in the lex file
- In the yacc file, errors are identified through actions, and printed out with:
    - Appropriate error message
    - Line number of the error occurring

- Some of the errors that are handled:
    - Main function return type mismatch
    - Variable type mismatch
    - Variable being used before it is declared
    - Variable being used out of scope
    - Re-declaration of a variable in the same scope

4. IMPLEMENTATION

- **SYNTAX VALIDATION AND SYMBOL TABLE CREATION:**

Lexical Analysis:

- Removal of comments of the forms:
  - Single Line: ***//***
  - Multi Line : /* */
- Ignoring of whitespaces, tabs and newlines
- Generation of tokens
- yylval is used to record the value of each lexeme scanned
- yylineno is used records the lexeme position in terms of line number in the C program file

- Recognizing the variable types i.e int, float and returning the correct token and the appropriate field of yylval
- Values associated with variables, are returned appropriately in yylval
- For each character that cannot be matched to any token pattern in the lex file, Panic Mode Recovery is used.

Syntax Analysis:

- In the parser file, we verify if the sequence of tokens forms a valid sentence, given the definition of our grammar
- The language supports main function, headers, variables of various types (including single dimension arrays), arithmetic expressions, boolean expressions, pre and postfix expressions, while and switch constructs for the C language
- Symbol table is generated for the program:
    - The symbol table contains the following fields:
        - Identifier Name
        - Scope of the identifier
        - Value associated with identifier
        - Type of the identifier
        - Storage required for the type

- Variables with the same identifier names may be present in different scopes, this is reflected in the symbol table.

PHASE 2 - SEMANTIC ANALYSIS

- Writing appropriate rules for checking semantic validity

- Checking that variables must be declared before use, and can only be used in ways that are acceptable for the declared type.
- Checking if new declarations do not conflict with earlier ones,using scope.
- Handling errors with respect to scoping and declarations.
- Handling main() function return type validation. i.e int main() should return an integer.
- Symbol Table:
  - The symbol table is represented as an array of structures.
  - Each entry in the symbol table is associated with an element in the array of structures.
  - There may be variables of the same identifier name present in different scopes
  - Scope in C language is represented within { } .
  - Values of identifiers in the symbol table, is done considering scope as well.
  - If a variable is re-declared in a new scope, a new entry is created in the symbol table with relevant scope
  - If the variable is re-initialized in a new scope, the existing relevant entry is updated with the new re-initialized value.

- **INTERMEDIATE CODE GENERATION**
  - The Intermediate Code Representation implemented is **Three Address Code (3AC).**
  - The 3AC generated is also represented in Quadruples format
  - Appropriate actions are included in the grammar, to generate 3AC

    - For expressions, stack based actions are used.
    - For example, for the expression : x= 1+2,
      the RHS components i.e ‘1’, ‘+’ and ‘2’ are pushed on the stack
      stack[top]=stack[top-2] ‘+’ stack[top-1].

  - For switch and while constructs, Labels are generated by using relevant actions in the grammar for the constructs.
  - Temporaries and Label suffixes are incremented appropriately
  - The 3AC is represented as Quadruples

- **CODE OPTIMIZATION**

  - **Constant Propagation:**
    - In an arithmetic expression, when an identifier is encountered on the right hand side, constant propagation is implemented
    - The approach involves searching the symbol table for the concerned identifier, in descending order of scope i.e most recent scope

  - **Constant Folding:**

    - In an arithmetic expression, constant folding is implemented by evaluating every sub-expression within the expression in real time
    - The folded value is updated in the symbol table

- **Identification of Loop Invariant Code:**
    - Loop invariant code is identified by checking relevant loop variant variables.
    - We keep track of the variables used in the loop condition, and make relevant checks with every statement in the loop body.
    - Any statement that does not use any of the condition variables is identified as loop invariant code.

- **Dead Code Identification**
    - Removal of statements after return
    - If the main() function does not return any value or does not print anything, the entire program is labelled as dead code
    - This identification is performed by relevant actions in the grammar

ERROR HANDLING:

- Panic Mode Recovery is performed in scanner, when a character does not match any pattern in the lex file
- In the yacc file, errors are identified through actions, and printed out with:

    - Appropriate error message
    - Line number of the error occurring

- Some of the errors that are handled:
    - Main function return type mismatch
    - Variable type mismatch
    - Variable being used before it is declared
    - Variable being used out of scope
    - Re-declaration of a variable in the same scope

- The return type of the main() function is noted, and the value returned by the return statement is compared to this type. If there is a type mismatch, error is printed with “Return Type Mismatch” message.
- Functions are implemented to check the scope of the variable/identifier and the type of the variables/identifier
- If the type of the variable on the left hand side of the expression does not match the type of the value on the right hand side, then a “Type Mismatch” error is displayed.
- Functions are implemented to check for the existence of the identifier/variable of the appropriate scope, in the symbol table
- If a variable is used, and the variable has no previous entry in the symbol table in the allowed scopes, then a “Undeclared Variable” error has occurred.
- Similarly, if a variable is declared, and there already exists an entry in the symbol table for the same identifier in the same scope, then the variable has been redeclared. A “Variable Redeclared” error is displayed.

INSTRUCTIONS TO RUN FILE:

command:

$ sh run.sh

run.sh file:

```
lex parser.l
yacc parser.y
gcc y.tab.c -ll -w
./a.out test.c
```

To change the file name to give as input to the compiler, change the test.c in the run.sh to the .c file name of your choice.

5. RESULTS AND SNAPSHOTS

1. **Constant Propagation, Constant Folding, Expression Evaluation demo**

```c
#include <stdio.h>

int main()
{
	int a = 20;

	int b = a + 10;

	{
		int b = 5;
	}

	int c = 40;

	int d = (c/a + b - 10/2)*5;


	return 0;

}
```

Output:

```
a = 20

t0 = 20 + 10

b = t0

b = 5

c = 40

t1 = 40 / 20

t2 = t1 + 30

t3 = 10 / 2

t4 = t2 - t3

t5 = t4 * 5

d = t5
Parsing done
-------------------------------------------------------Symbol Table-------------------------------------------------------

Sl.No   Identifier       Scope            Value            Type             Storage
-----------------------------------------------------------------------------------------------------------------------------

1       a                1                20               INT              4
2       b                1                30               INT              4
3       b                2                5                INT              4
4       c                1                40               INT              4
5       d                1                135              INT              4
6       main             0                0                MAIN FUNCTION - INT              0
-----------------------------------------------------------------------------------------------------------------------------


 OP              ARG1             ARG2             RESULT

=               20               NULL             a
+               20               10               t0
=               t0               NULL             b
=               5                NULL             b
=               40               NULL             c
/               40               20               t1
+               t1               30               t2
/               10               2                t3
-               t2               t3               t4
*               t4               5                t5
=               t5               NULL             d
```

2. Unoptimized 3AC - Expression Evaluation

```c
#include <stdio.h>

int main()
{

        int d = (40/20 + 30 - 10/2)*5;


        return 0;

}
```

Output:

```
t0 = 40 / 20

t1 = t0 + 30

t2 = 10 / 2

t3 = t1 - t2

t4 = t3 * 5

d = t4
Parsing done

------------------------------------------------------Symbol Table----------------------------

Sl.No   Identifier       Scope          Value          Type            Storage
-----------------------------------------------------------------------------------------------

1       d                1              135            INT             4
2       main             0              0              MAIN FUNCTION - INT             0
-----------------------------------------------------------------------------------------------



 OP             ARG1            ARG2            RESULT

/               40              20              t0
+               t0              30              t1
/               10              2               t2
-               t1              t2              t3
*               t3              5               t4
=               t4              NULL            d
```

3. Switch Construct 3AC

```c
#include <stdio.h>

int main()
{
        int a = 30;
        int b=3;

        switch(a)
        {
                case 10: b=1;

                case 20: b=2;
                break;


                case 30: b=3;
                        break;

                default: b=4;


        }

        return 0;

}
```

```
a = 30

b = 3


L1:
t0 = a == 10
iffalse t0 goto L2
b = 1
goto L3

L2:
t1 = a == 20
iffalse t1 goto L4
L3:
b = 2
goto next1

L4:
t2 = a == 30
iffalse t2 goto L5
b = 3
goto L5

L5:
b = 4

next1:
Parsing done
-------------------------------------------------------Symbol Table-------------------------------

Sl.No   Identifier      Scope           Value           Type            Storage
--------------------------------------------------------------------------------------------------

1       a               1               30              INT             4
2       b               1               4               INT             4
3       main            0               0               MAIN FUNCTION - INT             0
--------------------------------------------------------------------------------------------------


 OP             ARG1            ARG2            RESULT

=               30              NULL            a
=               3               NULL            b
iffalse         t0              NULL            L2
=               1               NULL            b
goto            NULL            NULL            L3
iffalse         t1              NULL            L4
=               2               NULL            b
goto            NULL            NULL            next1
iffalse         t2              NULL            L5
=               3               NULL            b
goto            NULL            NULL            L5
=               4               NULL            b
```

4. While Construct - 3AC

```c
#include <stdio.h>

int main()
{
        int a = 20;
        int b=3;

        while(a<50)
        {

                printf("in while");
                a=a+1;

        }




        return 0;

        b=100;

}
```

```
a = 20

b = 3

L1:

t0 = a < 50
iffalse t0 goto L2
t1 = a + 1

a = t1

goto L1

L2:

b = 100

Statements after line number 19 is DEAD CODE
Parsing done

-------------------------------------------------------Symbol Table----------------------------------------

Sl.No   Identifier      Scope           Value           Type            Storage
------------------------------------------------------------------------------------------------------------


1       a               1               1               INT             4
2       b               1               100             INT             4
3       main            0               0               MAIN FUNCTION - INT             0
------------------------------------------------------------------------------------------------------------



 OP             ARG1            ARG2            RESULT

=               20              NULL            a
=               3               NULL            b
Label           NULL            NULL            L1
<               a               50              t0
iffalse         t0              NULL            L2
+               a               1               t1
=               t1              NULL            a
goto            NULL            NULL            L1
=               100             NULL            b
```

5. While Construct - Loop Invariant Code Identification

```
#include <stdio.h>

int main()
{
        int a = 20;
        int b=3;

        while(a<50)
        {

                printf("in while");

                a=a+1;

                int b=5;

                b=1+a;

                b=a+1;

                b= c + a;

                b = c + 5;

                int c = 5 + a;

                int d = a+7;

                int f = a;

        }



        return 0;

        b=100;

}
```

```
a = 20

b = 3

L1:

t0 = a < 50
iffalse t0 goto L2
t1 = a + 1

a = t1

b = 5

LOOP INVARIANT: 15
t2 = 1 + a

b = t2

t3 = a + 1

b = t3

t4 = c + a

b = t4

t5 = c + 5

b = t5

LOOP INVARIANT: 23
t6 = 5 + a

c = t6

t7 = a + 7

d = t7

f = a

goto L1

L2:

b = 100

Statements after line number 36 is DEAD CODE
Parsing done
```

-----Symbol Table-----

Sl.No	Identifier	Scope	Value	Type	Storage
1	a	1	1	INT	4
2	b	1	100	INT	4
3	b	2	5	INT	4
4	c	2	5	INT	4
5	d	2	7	INT	4
6	f	2	0	INT	4
7	main	0	0	MAIN FUNCTION - INT	0

OP	ARG1	ARG2	RESULT
=	20	NULL	a
=	3	NULL	b
Label	NULL	NULL	L1
<	a	50	t0
iffalse	t0	NULL	L2
+	a	1	t1
=	t1	NULL	a
=	5	NULL	b
+	1	a	t2
=	t2	NULL	b
+	a	1	t3
=	t3	NULL	b
+	c	a	t4
=	t4	NULL	b
+	c	5	t5
=	t5	NULL	b
+	5	a	t6
=	t6	NULL	c
+	a	7	t7
=	t7	NULL	d
=	a	NULL	f
goto	NULL	NULL	L1
=	100	NULL	b

6. CONCLUSIONS

- This project gave us a good understanding of the workings of a compiler
- We have understood the intricate implementation details of certain optimizations and this has helped us solidify our concepts.