Report on

"Compiler Design Project - C Compiler"

Submitted in partial fulfilment of the requirements for **Semester VI**

Compiler Design Project

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Vishnu A S	**PES1201800192**
Ashay G	**PES1201801767**
Girish GN	**PES1201802101**

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

Ch. No	Title	Page No.
1	INTRODUCTION	3
2	ARCHITECTURE OF LANGUAGE	3
3	CONTEXT FREE GRAMMAR	3
4	DESIGN STRATEGY • SYMBOL TABLE CREATION • INTERMEDIATE CODE GENERATION • CODE OPTIMIZATION • ERROR HANDLING	12
5	IMPLEMENTATION DETAILS : • SYMBOL TABLE CREATION • INTERMEDIATE CODE GENERATION • CODE OPTIMIZATION • ERROR HANDLING	12
6	RESULTS AND POSSIBLE SHORTCOMINGS	13
7	CONCLUSIONS	14
8	FURTHER ENHANCEMENTS	14
9	SNAPSHOTS	15

Constructs :

- **If else**
- **do while**

Introduction and Problem Statement

To implement a Compiler for C using Lex, Yacc and C as the language to code the compiler in. The input could be any valid C Program with Functions, Conditions and Looping constructs. The output would be the expected output from “gcc” when run on a standard Unix shell.

Architecture of the Language

The syntax is the same as C, for all the constructs that we are focusing on. We have made sure to implement the semantics of the language as close to C as possible. We have handled all errors and border cases for every construct in our language and have added many error checking mechanisms too.

Grammar

We used a regular Backus Normal Form Grammar to develop the entire language. Most of the productions inspired by ISO C and a few constructs that we created to provide some novelty to our project.

The grammar has been pasted below:

preprocessing-token:

    #include header-name

    #define literal literal

declaration-statement:

    attribute declaration-specifier

function-definition:

    keyword identifier (identifier-sequence) function-body

function-body:

    compound-statement

token:

identifier

keyword

literal

operator-token

punctuator

header-name:

< string-path >

" string-path "

identifier:

identifier-nondigit

identifier digit

statement:

labeled-statement

expression-statement

compound-statement

selection-statement

iteration-statement

jump-statement

labeled-statement

identifier : statement

compound-statement:

{ statement-seq }

statement-seq:

statement

statement-seq statement

selection-statement:

if (condition) statement

if (condition) statement else statement

condition:

expression

type-specifier-seq declarator = assignment-expression

iteration-statement:

while (condition) statement

for (for-range-declaration : for-range-initializer) statement

for-range-declaration:

attribute-specifier-seqopt type-specifier-seq declarator

for-range-initializer:

expression braced-init-list

jump-statement:

break ;

continue ;

return expressionopt ;

return braced-init-listopt ;

expression:

multiplicative-expression

additive-expression

relational-expression

equality-expression

logical-and-expression

logical-or-expression

conditional-expression

assignment-expression

multiplicative-expression:

pm-expression

multiplicative-expression * pm-expression

multiplicative-expression / pm-expression

multiplicative-expression % pm-expression

additive-expression:

multiplicative-expression

additive-expression + multiplicative-expression

additive-expression - multiplicative-expression

relational-expression:

shift-expression

relational-expression < shift-expression

relational-expression > shift-expression

relational-expression <= shift-expression

relational-expression >= shift-expression

equality-expression:

relational-expression

equality-expression == relational-expression

equality-expression != relational-expression

logical-and-expression:

Inclusive-or-expression

logical-and-expression && inclusive-or-expression

logical-or-expression:

logical-and-expression

logical-or-expression || logical-and-expression

conditional-expression:

logical-or-expression

logical-or-expression ? expression : assignment-expression

assignment-expression:

Conditional-expression

logical-or-expression assignment-operator initializer-clause

throw-expression

assignment-operator:

=

*=

/=

%=

+=

-=

>>=

<<=

&=

^=

|=

identifier-nondigit:

nondigit

nondigit:

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x

y

z

id-expression:

identifier

Identifier-digit:

0

1

2

3

4

5

6

7

8

9

keyword:

break

char

continue

double

else

false

float

for

if

int

long

return

true

void

punctuator:

{

}

[

]

#

(

)

;

:

?

literal:

integer-literal

character-literal

floating-literal

string-literal

boolean-literal

integer-literal:

decimal-literal integer-suffixopt

decimal-literal:

nonzero-digit

decimal-literal digit

nonzero-digit:

1

2

3

4

5

6

7

8

9

c-char:

any member of the source character set except the single quote ', backslash \, or new-line character

escape-sequence

universal-character-name

escape-sequence:

simple-escape-sequence

simple-escape-sequence:

\'

\"

\\

\n

\t

sign:

+

-

digit-sequence:

digit

digit-sequence digit

string-literal:

s-char-sequence

s-char-sequence:

s-char

s-char-sequence s-char

s-char:

any member of the source character set except the double-quote ", backslash \, or new-line character

escape-sequence

universal-character-name

boolean-literal:

false

true

Design Strategies and Implementation Details

Symbol Table Generation

We are using a linked list of structures to implement our Symbol Table. It is created on the Heap and is called to a print function just before the compilation ends. The print function outputs a formatted symbol table to STDOUT.A node of the symbol table has the following structure.

1. Line
2. Name
3. Scope (0 for Global, -1 for Main)
4. Value
5. ID
6. Data Type

Which is displayed in a tabular form during the end of the program to represent the symbol table that has been generated during the first phase.

We store the entire table in a dynamic list-queue data structure.

Intermediate Code Generation

Intermediate Code Generation is implemented using various functions and data structures that are used to generate and store the Intermediate Codes. We have a list of structures of type Quadruple to store the Quadruples generated by the compiler. Several functions are used to accomplish the required processing. We then use a print function that neatly formats the IC and prints it onto STDOUT.

Code Optimisation :

We have done 5 optimisations in our project.They are :

- Eliminate dead code
- Subexpression elimination

- Constant Folding
- Constant Propagation
- Loop Invariant code outside loop

These were done using python language as it was easy to split the lines and it had various inbuilt methods which we can make use of to optimise the intermediate code.This give out Optim_ICG.txt file which contains the optimized the code

Error Handling

Various Errors are handled in our compiler such as syntactic and semantic errors.The yyerror function handles the errors and gives out respective error messages with the respective line no's.

Results

Our Compiler has been able to compile and generate code for the Sample Input files pretty accurately. It can detect a plethora of errors and satisfactorily compile and produce optimal code. We are confident that by the end of the semester, we will be able to build a reputable compiler for C.

Possible Shortcoming

The compiler we built is a mini-compiler and doesn't entirely mimic or compile all C code. We haven't implemented STLs that make up the majority of C. And the functions we have generated have been optimized specifically for the current language and grammar that has been elaborated on in this document. The generated code may be a bit buffed up compared to a highly optimized version of the same, generated by an Official C Compiler.

Conclusions

We can conclude that a satisfactorily accurate compiler can be build using Lex and Yacc for a number of different languages spreading across multiple genres. We can conclude that the various phases of a standard compiler can be built and implemented using these tools and by following all regulations, a standard compiler can be built for almost any language.

Further Enhancements

We have included two construct specified and plan to add a few more techniques we have learned during the duration of this course that may not be present in the standard C compiler.

Build and Run :

lex lexer.l
yacc -d parser.y
gcc y.tab.c -ll -ly
./a.out input.c
python optimization.py

Snapshots 1: Sample Input with no errors - input.c

```c
#include <stdio.h>
int p = 0;
int main()
{
    int k = 10;
    int a = 10;
    int b = 40;
    int c,e,g,d;
    a = b * c + g;
    d = b * c * e;
    a = 5;
    float qw;
    char c2[100];
    int h = a;
    p--;
    ++p;
    --p;
    if(p<5)
        a=10;
    else
        a=20;
    int i=4;
    do {
        i--;
        int t=b;
    }
    while(i>1);
    int l;
    a=p;
    int uy = a+b*l;
    h=15;
    int y;
    h = y;
}
```

Output :

Intermediate Code Generation Without Optimisation:

```
ashay@Ashays-MacBook-Pro compiler % sh run.sh
p =  0
k =  10
a =  10
b =  40
t0 = b * c
t1 = t0 + g
a =  t1
t2 = b * c
t3 = t3 * e
d =  t3*e
a =  5
h =  a
p = p - 1
p = p + 1
p = p - 1
if  p<5  goto L0
goto L1
L0 :
a =  10
goto L2
L1 :
a =  20
L2 :
i =  4
L3 :
i = i - 1
t =  b
if  i>1  goto L3
goto L4
L4 :
a =  p
t4 = b * l
t5 = a + t4
uy =  t5
h =  15
h =  y
```

Symbol table :

Line	Name	Scope	value	id_type	datatype
2	p	0	p+1	IDENT	int
5	k	-1	10	IDENT	int
6	a	-1	p	IDENT	int
7	b	-1	40	IDENT	int
8	c	-1		IDENT	int
8	e	-1		IDENT	int
8	g	-1		IDENT	int
8	d	-1	t3*e	IDENT	int
9	t0	-1	b*c	TEMP	TEMP
9	t1	-1	t0+g	TEMP	TEMP
10	t2	-1	b*c	TEMP	TEMP
10	t3*e	-1	t3*e	TEMP	TEMP
12	qw	-1		IDENT	float
13	c2	-1		ARRAY	char
14	h	-1	y	IDENT	int
22	i	-1	i+1	IDENT	int
25	t	-1	b	IDENT	int
28	l	-1		IDENT	int
30	t4	-1	b*l	TEMP	TEMP
30	t5	-1	a+t4	TEMP	TEMP
30	uy	-1	t5	IDENT	int
32	y	-1		IDENT	int
34	main	0		FUNCT	int

Output :

Optimisations :

```
Loop Invariant Code Motion
__________________________
()
p =  0
k =  10
a =  10
b =  40
t0 = b * c
t1 = t0 + g
a =  t1
t2 = b * c
t3 = t3 * e
d =  t3*e
a =  5
h =  a
p = p - 1
p = p + 1
p = p - 1
if  p<5  goto L0
goto L1
L0 :
a =  10
goto L2
L1 :
a =  20
L2 :
i =  4
L3 :
i = i - 1
t =  b
if  i>1  goto L3
goto L4
L4 :
a =  p
t4 = b * l
t5 = a + t4
uy =  t5
h =  15
```

```
_____________________
()
Dead Code Elimination
_____________________
()
p =  0
a =  10
b =  40
t0 = b * c
t1 = t0 + g
a =  t1
a =  5
p = p - 1
p = p + 1
p = p - 1
if  p<5  goto L0
goto L1
L0 :
a =  10
goto L2
L1 :
a =  20
L2 :
i =  4
L3 :
i = i - 1
t =  b
if  i>1  goto L3
goto L4
L4 :
a =  p
t4 = b * l
t5 = a + t4
uy =  t5
h =  15
h =  y
()
```

```
_________________________
()
Constant Folded Expression
_________________________
()
('p', '=', '0')
('a', '=', '10')
('b', '=', '40')
('a', '=', 't1')
('a', '=', '5')
('p', '=', -1)
('p', '=', 0)
('p', '=', -1)
('IF', 'p<5', 'GOTO', 'L0')
('GOTO', 'L1')
('L0', ':')
('a', '=', '10')
('GOTO', 'L2')
('L1', ':')
('a', '=', '20')
('L2', ':')
('i', '=', '4')
('L3', ':')
('i', '=', 3)
('t', '=', '40')
('IF', 'i>1', 'GOTO', 'L3')
('GOTO', 'L4')
('L4', ':')
('a', '=', -1)
('uy', '=', 't5')
('h', '=', '15')
('h', '=', 'y')
ashay@Ashays-MacBook-Pro compiler %
```

```
_________________________
()
Constant Folding Quadruples
_________________________
()
('=', '0', 'NULL', 'p')
('=', '10', 'NULL', 'a')
('=', '40', 'NULL', 'b')
('*', '40', 'c', 't0')
('+', 't0', 'g', 't1')
('=', 't1', 'NULL', 'a')
('=', '5', 'NULL', 'a')
('=', -1, 'NULL', 'p')
('=', 0, 'NULL', 'p')
('=', -1, 'NULL', 'p')
('IF', 'p<5', 'GOTO', 'L0')
('GOTO', 'L1')
('LABEL', 'L0')
('=', '10', 'NULL', 'a')
('GOTO', 'L2')
('LABEL', 'L1')
('=', '20', 'NULL', 'a')
('LABEL', 'L2')
('=', '4', 'NULL', 'i')
('LABEL', 'L3')
('=', 3, 'NULL', 'i')
('=', '40', 'NULL', 't')
('IF', 'i>1', 'GOTO', 'L3')
('GOTO', 'L4')
('LABEL', 'L4')
('=', -1, 'NULL', 'a')
('*', '40', 'l', 't4')
('+', '-1', 't4', 't5')
('=', 't5', 'NULL', 'uy')
('=', '15', 'NULL', 'h')
('=', 'y', 'NULL', 'h')
```

Optimized Intermediate Code :

```
 1  p = 0
 2  a = 10
 3  b = 40
 4  a = t1
 5  a = 5
 6  p = -1
 7  p = 0
 8  p = -1
 9  IF p<5 GOTO L0
10  GOTO L1
11  L0 :
12  a = 10
13  GOTO L2
14  L1 :
15  a = 20
16  L2 :
17  i = 4
18  L3 :
19  i = 3
20  t = 40
21  IF i>1 GOTO L3
22  GOTO L4
23  L4 :
24  a = -1
25  uy = t5
26  h = 15
27  h = y
28
```

Snapshots 2 : Sample Input with errors - input1.c

```c
#include <stdio.h>
int p = 0;
int main()
{
    int k = 10;
    int a = 10;
    int b = 40;
    int c,e,g,d;
    a = b * c + g  //error no semicolon
    d = b * c * e;
    a = 5;
    float qw;
    char c2[q]; //error q not defined
    int h = a;
    p--;
    ++p;
    --p;
    if(p<5)
        a=10;
    else
        a=20;
    int i=4;
    do {
        i--;
        int t=b;
    }
    while(i>1);
    int l;
    a=p;
    int uy = a+b*; //error
    h=15;
    int y;
    h = y;
    h = o; //error 'o' not defined
}
```

Output :

```
ashay@Ashays-MacBook-Pro CD-Project % sh run.sh
Error: syntax error on line number 10
Error: syntax error on line number 13
Error: syntax error on line number 30
Error on 34, Assignment RHS not declared
p =  0
k =  10
a =  10
b =  40
t0 = b * c
t1 = t0 + g
a =  t1
a =  5
h =  a
p = p - 1
p = p + 1
p = p - 1
if  p<5  goto L0
goto L1
L0 :
a =  10
goto L2
L1 :
a =  20
L2 :
i =  4
L3 :
i = i - 1
t =  b
if  i>1  goto L3
goto L4
L4 :
a =  p
h =  15
h =  y
h =  $
```

Symbol Table :

Line	Name	Scope	value	id_type	datatype
2	p	0	p+1	IDENT	int
5	k	-1	10	IDENT	int
6	a	-1	p	IDENT	int
7	b	-1	40	IDENT	int
8	c	-1		IDENT	int
8	e	-1		IDENT	int
8	g	-1		IDENT	int
8	d	-1		IDENT	int
9	t0	-1	b*c	TEMP	TEMP
10	t1	-1	t0+g	TEMP	TEMP
12	qw	-1		IDENT	float
14	h	-1	$	IDENT	int
22	i	-1	i+1	IDENT	int
25	t	-1	b	IDENT	int
28	l	-1		IDENT	int
32	y	-1		IDENT	int
35	main	0		FUNCT	int

4 Errors Found