CSB 353: Compiler Design

Compiler Design Project Report

Partial C Compiler

Submitted By: Branch: **CSE**

Kati Kalyani (201210026)

Chikkudu Charan Teja (201210013)

Dharavath Rohith (201210015)

Semester: **6th Sem**

Submitted To:

Dr. Shelly Sachdeva

Associate Professor

NATIONAL INSTITUTE OF TECHNOLOGY DELHI

<u>Department of Computer Science and Engineering</u>

2023

INDEX

ABSTRACT

The Partial C Compiler is a project aimed at developing a compiler for the C programming language that generates assembly code for a simple computer architecture. The project involves the design and implementation of a lexical analyzer, parser, semantic analyzer, intermediate code generator, and code generator.

Once the program has been successfully analyzed, the intermediate code generator generates an intermediate representation of the program, which is then used by the code generator to produce the final assembly code. The code generator maps the intermediate representation to a specific computer architecture, generating efficient code that can be executed on the target machine.

The Partial C Compiler project presents a number of interesting challenges, including the need to design efficient algorithms for parsing and semantic analysis, as well as the need to generate optimal assembly code for a simple computer architecture. To address these challenges, the project team will make use of established techniques and tools from the field of compiler design, such as finite automata, recursive descent parsing, and optimization algorithms.

The Partial C Compiler project has several potential applications, including the development of simple embedded systems that require custom software, and the education of students and programmers interested in learning about compiler design and implementation. By successfully completing the Partial C Compiler project, the project team will gain valuable experience in designing and implementing compilers, as well as a deeper understanding of the inner workings of the C programming language.

Functionalities

Features implemented in this project:

- Looping Constructs: It will support nested for and while loops.

  Syntax:

  ```
  int i;
  for(i=0;i<n;i++){
  }
  int x;
  while(x<10){ … x++}
  ```

- Conditional Constructs:

  if…else-if…else statements,

  with support of nested conditional statement

- Operators:

  ADD(+), MULTIPLY(*), DIVIDE(/), MODULO(%), AND(&), OR(|)

- Structure construct of the language,

  Syntax: struct pair{ int a; int b};

- Function construct of the language,

  Syntax: int func(int x)

- Support for a 1-Dimensional array.

  Syntax : char s[20];

Lexer/Lexical Analysis

- Token Generation
- Line Numbers
- Lexical Errors
- Record Lexemes
- Identify Keywords
- Symbol Table

Parser/Syntax Analysis

- Abstract Syntax Tree Construction
- Global & Local Variables Distinction
- Arrays/Pointers definition
- Define struct, union
- Shift Reduce, Reduce Reduce Error correction
- Valid Actions for grammar productions
- Error Handing and Error Recovery

#Intermediate Code Generator

- Generate three address code
- Quadruple format

- Insert Temporaries to Symbol Table

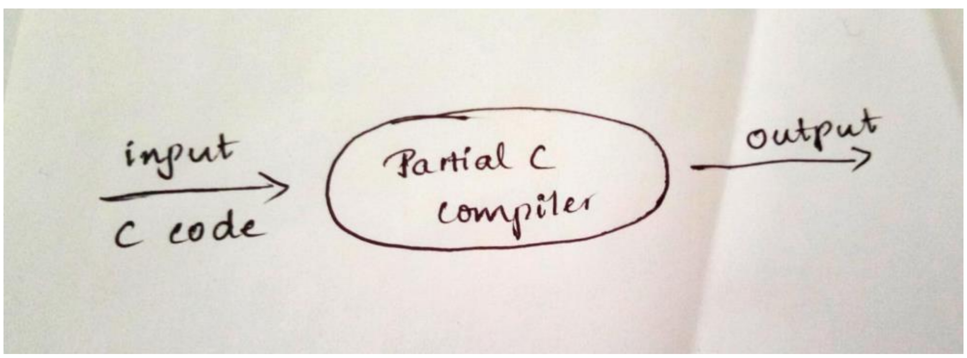
- AST

Intermediate Code Optimization

- Eliminate dead code

- Constant Folding, Propagation

WorkFlow



Following are the input and output codes for the above workflow:-

For While

Input:- **Output:-**

```
katikalyani@iampoor136: ~/work
#include<stdio.h>
int main ()
{
int n=5;
while(n<0)
{
n=n-1;
}
return 0;
}
~
```

```
        .file   "test1.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $5, -4(%rbp)
        jmp     .L2
.L3:
        subl    $1, -4(%rbp)
.L2:
        cmpl    $0, -4(%rbp)
        js      .L3
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 11.2.0-19ubuntu1) 11.2.0"
        .section        .note.GNU-stack,"",@progbits
```

For Loop

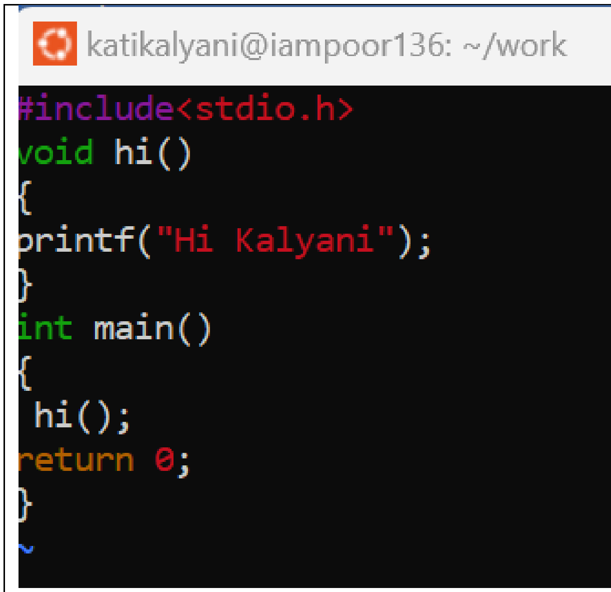
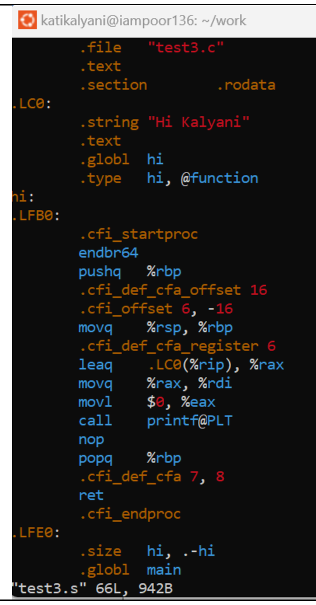
Input:- **Output:-**

```
katikalyani@iampoor136: ~/work
#include<stdio.h>
int main ()
{
for(int i=0;i<3;i++)
{
        printf("Kalyani\n");
}
return 0;}
~
```

```
katikalyani@iampoor136: ~/work
        .file   "test2.c"
        .text
        .section        .rodata
.LC0:
        .string "Kalyani"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        jmp     .L2
.L3:
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        call    puts@PLT
        addl    $1, -4(%rbp)
.L2:
        cmpl    $2, -4(%rbp)
        jle     .L3
        movl    $0, %eax
```

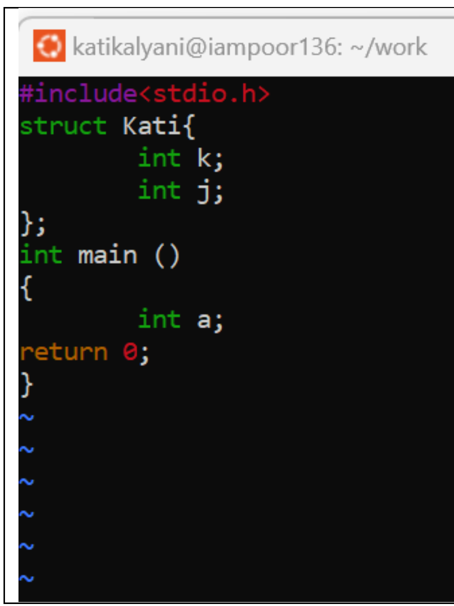
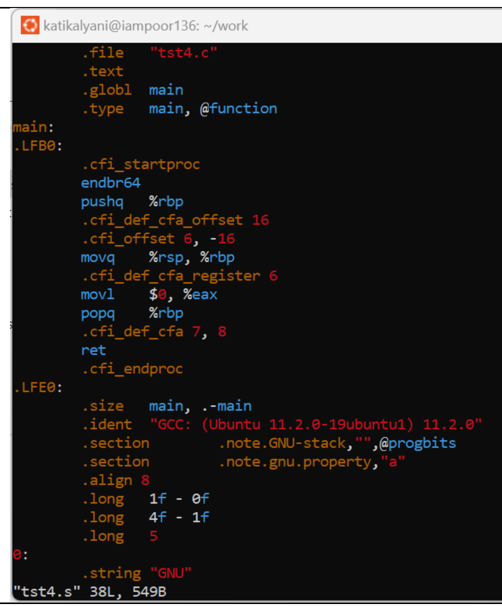
Function

Input:- **Output:-**


katikalyani@iampoor136: ~/work
```
#include<stdio.h>
void hi()
{
printf("Hi Kalyani");
}
int main()
{
 hi();
return 0;
}
~
```

katikalyani@iampoor136: ~/work
```
	.file	"test3.c"
	.text
	.section	.rodata
.LC0:
	.string "Hi Kalyani"
	.text
	.globl	hi
	.type	hi, @function
hi:
.LFB0:
	.cfi_startproc
	endbr64
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	leaq	.LC0(%rip), %rax
	movq	%rax, %rdi
	movl	$0, %eax
	call	printf@PLT
	nop
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	hi, .-hi
	.globl	main
"test3.s" 66L, 942B
```

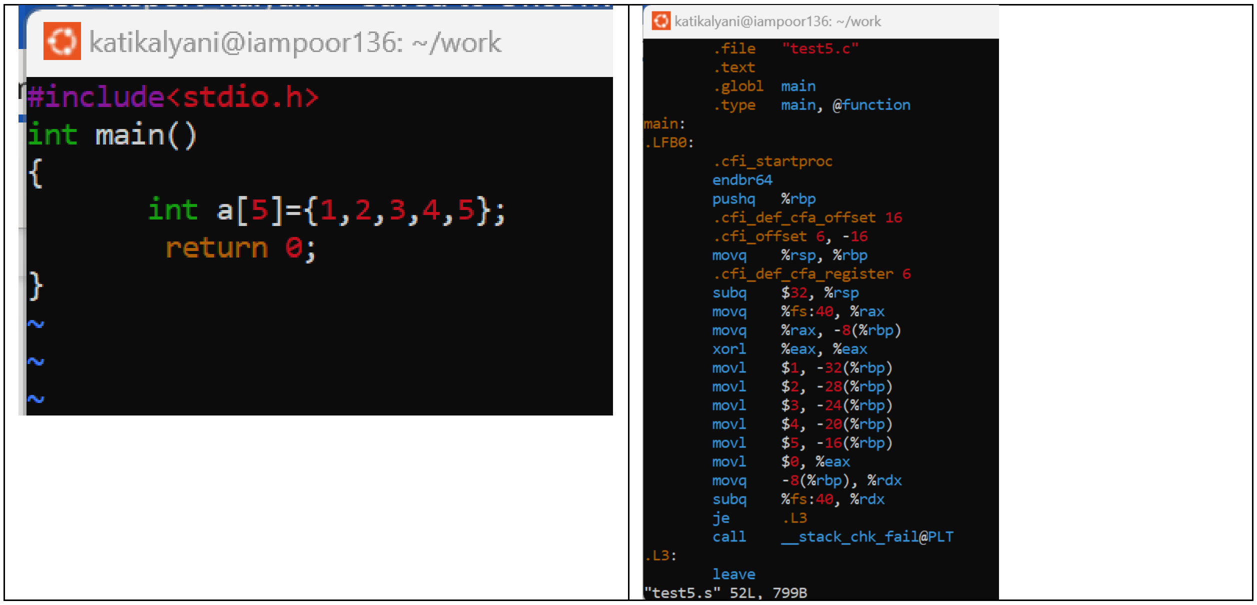
Struct Implementation

Input:- **Output:-**


katikalyani@iampoor136: ~/work
```
#include<stdio.h>
struct Kati{
	int k;
	int j;
};
int main ()
{
	int a;
return 0;
}
~
~
~
~
~
~
```

katikalyani@iampoor136: ~/work
```
	.file	"tst4.c"
	.text
	.globl	main
	.type	main, @function
main:
.LFB0:
	.cfi_startproc
	endbr64
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movl	$0, %eax
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	main, .-main
	.ident	"GCC: (Ubuntu 11.2.0-19ubuntu1) 11.2.0"
	.section	.note.GNU-stack,"",@progbits
	.section	.note.gnu.property,"a"
	.align 8
	.long	1f - 0f
	.long	4f - 1f
	.long	5
0:
	.string "GNU"
"tst4.s" 38L, 549B
```

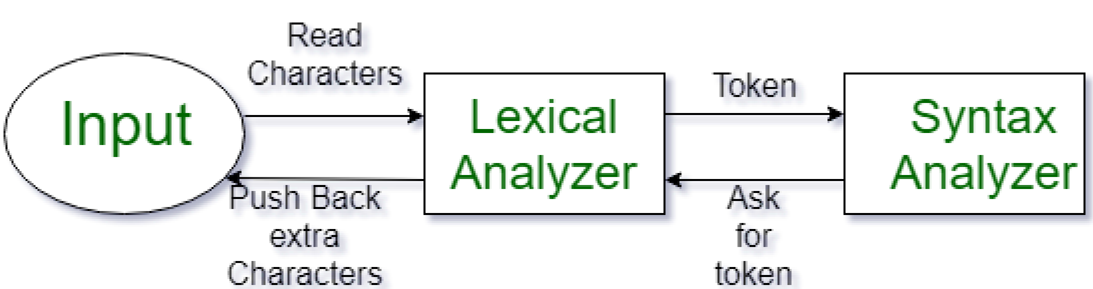
Array Implementation

Input:-



LEXICAL PHASE:-

Lexical analysis is the starting phase of the compiler. It gathers modified source code that is written in the form of sentences from the language preprocessor.



Input:-

```c
// Implicit Error that our Language doesn't support

#include<stdio.h>

int main() {
    char @hello;
    @hello = 'c';
}
```

Output:-

```
katikalyani@iampoor136:~/LEXICAL ANALYZER$ lex scanner.l
katikalyani@iampoor136:~/LEXICAL ANALYZER$ gcc lex.yy.c
katikalyani@iampoor136:~/LEXICAL ANALYZER$ ./a.out
====================================================================
// Implicit Error that our Language doesn't support     - SINGLE LINE COMMENT
#include<stdio.h>       -Pre Processor directive
int      - KEYWORD
main     - KEYWORD
(        - OPENING BRACKETS
)        - CLOSING BRACKETS
{        - OPENING BRACES
char     - KEYWORD
ERROR at line no. 6
@


SYMBOL TABLE

char     KEYWORD
int      KEYWORD
main     KEYWORD


CONSTANT TABLE
```

Syntactic Phase or Parsing Phase:-

The second phase of a compiler is syntax analysis, also known as parsing. This phase takes the stream of tokens generated by the lexical analysis phase and checks whether they conform to the grammar of the programming language. The output of this phase is usually an Abstract Syntax Tree (AST).

Input:-

\\wsl.localhost\Ubuntu\home\katikalyani\PARSER\test1.c - Dev-C++ 5.11

```c
#include<stdio.h>

int fun(char x){
    return x*x;
}

void main(){
    int a=2,b,c,d,e,f,g,h;

    c=a+b;
    d=a*b;
    e=a/b;
    f=a%b;
    g=a&&b;
    h=a||b;
    h=a*(a+b);
    h=a*a+b*b;
    h=fun(b);

    //This Test case contains operator,structure,delimeters,Function;
}
```

Output:-

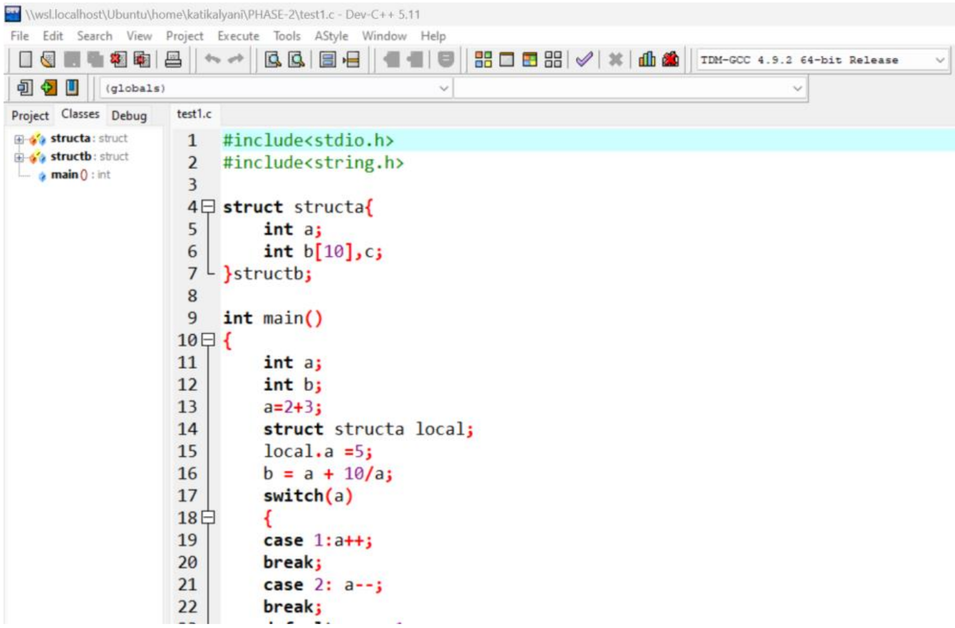
```
katikalyani@iampoor136:~/PARSER$ ./a.out
Status: Parsing Complete - Valid
                              SYMBOL TABLE
                              ------------

    SYMBOL |           CLASS |        TYPE |      VALUE |    LINE NO
--------------------------------------------------------------------------------

         a |      Identifier |         int |          2 |          8
         b |      Identifier |         int |            |          8
         c |      Identifier |         int |            |          8
         d |      Identifier |         int |            |          8
         e |      Identifier |         int |            |          8
         f |      Identifier |         int |            |          8
         g |      Identifier |         int |            |          8
         h |      Identifier |         int |            |          8
         x |      Identifier |        char |            |          3
      char |         Keyword |             |            |          3
       fun |      Identifier |         int |            |          3
    return |         Keyword |             |            |          4
       int |         Keyword |             |            |          3
      main |      Identifier |        void |            |          7
      void |         Keyword |             |            |          7


                              CONSTANT TABLE
                              --------------

      NAME |            TYPE
--------------------------------------------------------------------------------
         2 | Number Constant
```

```
katikalyani@iampoor136:~/PARSER$ lex scanner.l
katikalyani@iampoor136:~/PARSER$ yacc -d parser.y
parser.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
parser.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
katikalyani@iampoor136:~/PARSER$ gcc lex.yy.c y.tab.c -w
katikalyani@iampoor136:~/PARSER$ ./a.out
12 syntax error void
Status: Parsing Failed - Invalid
```

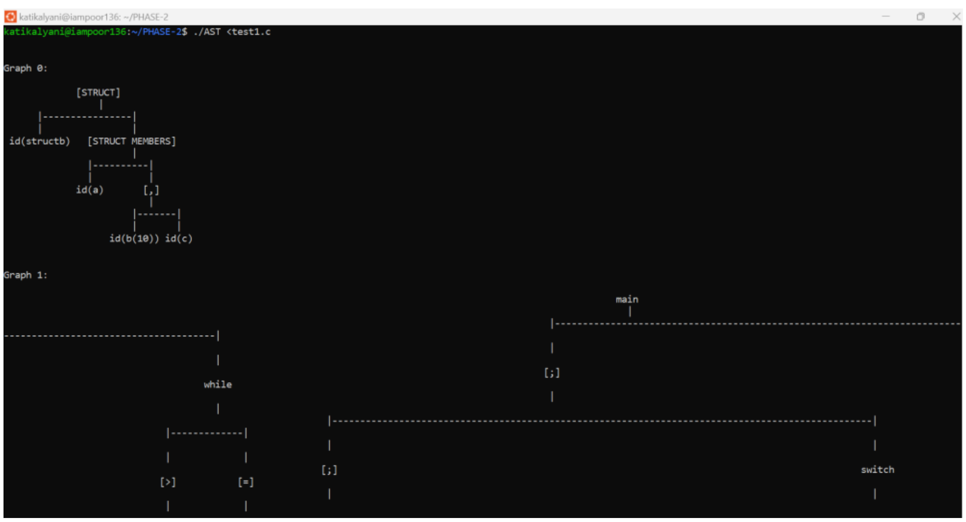
Semantic Analysis

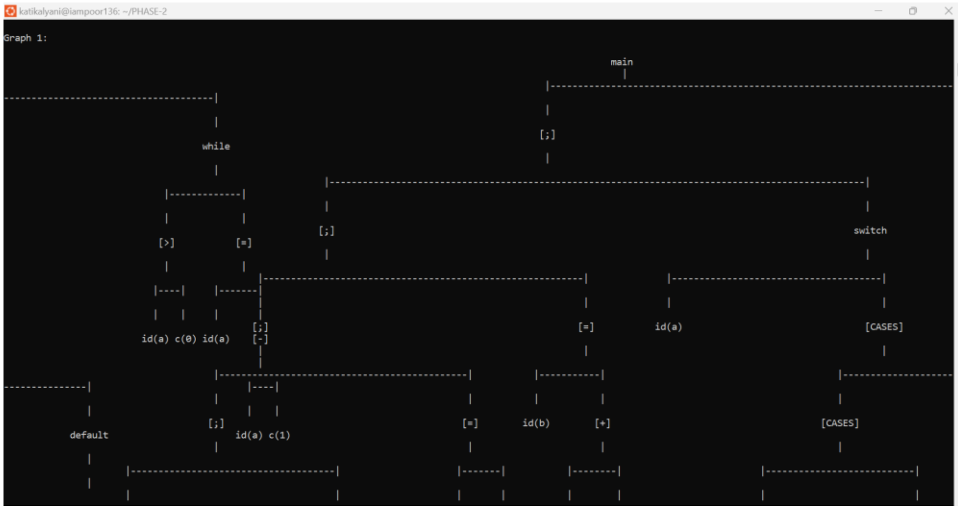
Semantic analysis is the task of ensuring that the declarations and statements of a program are Semantically correct, i.e that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used. Semantic analysis can compare information in one part of 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). Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures. Some of the functions of Semantic analysis are that it maintains and updates the symbol table, check source programs for semantic errors and warnings like type mismatch, global and local scope of a variable, re-definition of variables, usage of undeclared variables.

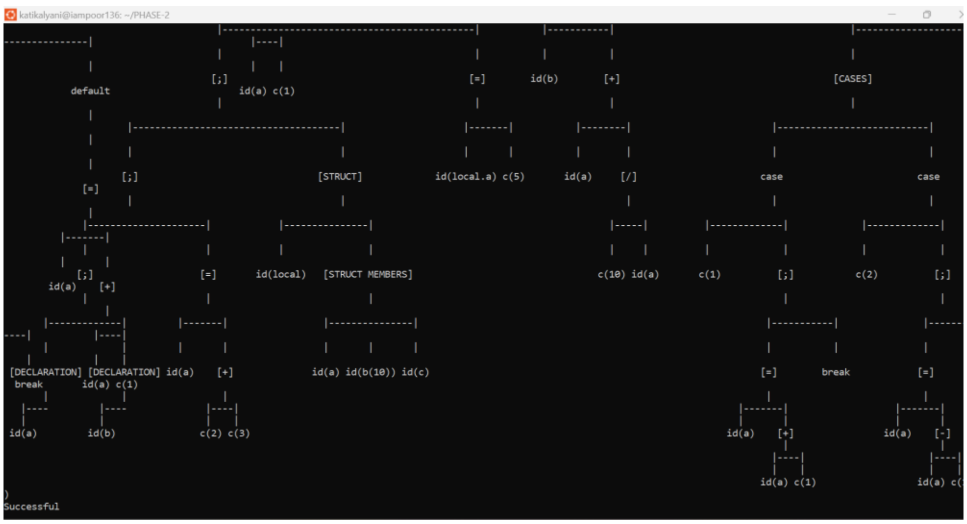
Input:-


\\wsl.localhost\Ubuntu\home\katikalyani\PHASE-2\test1.c - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Release
(globals)
Project Classes Debug
structa : struct
structb : struct
main () : int
test1.c
#include<stdio.h>
#include<string.h>
struct structa{
int a;
int b[10],c;
}structb;
int main()
{
int a;
int b;
a=2+3;
struct structa local;
local.a =5;
b = a + 10/a;
switch(a)
{
case 1:a++;
break;
case 2: a--;
break;

Output:-

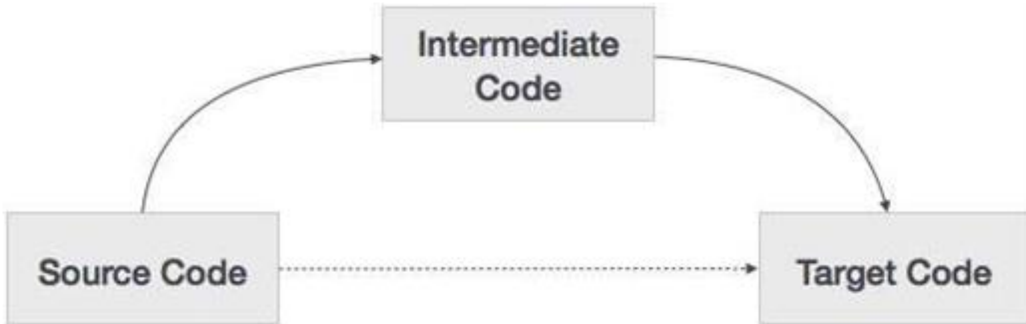

katikalyani@iampoor136: ~/PHASE-2
katikalyani@iampoor136:~/PHASE-2$ ./AST <test1.c
Graph 0:
[STRUCT]
id(structb)
[STRUCT MEMBERS]
id(a)
[,]
id(b(10))
id(c)
Graph 1:
main
[;]
while
[>]
[=]
[;]
switch





Intermediate Code Generation:

Intermediate code generator receives input from its predecessor phase, the semantic analyzer, in the form of an annotated syntax tree.



That syntax tree can then be converted into a linear representation. Intermediate code tends to be machine-independent code.

Three-Address Code – A statement involving no more than three references (two for operands and one for result) is known as a three-address statement. A sequence of three address statements is known as three address codes. Three address statement is of form x = y op z; here, x, y, z will have an address (memory location).

Input:-

```c
#include<stdio.h>
#include<string.h>

struct structa{
    int a;
    int b[10],c;
}structb;

int main()
{
    int a;
    int b;
    a=2+3;
    struct structa local;
    local.a =5;
    b = a + 10/a;
    switch(a)
    {
    case 1:a++;
    break;
    case 2: a--;
    break;
```

Output:-

```
katikalyani@iampoor136:~/PHASE-2$ ./a.out <test1.c
T0 = a * b
b = T0
L0:
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
a = T3
goto L0
L1:
T4 = b < = c
T5 = not T4
if T5 goto L3
a = 10
goto L4
L3:
a = 20
L4:
a = 100
i = 0
L5:
T6 = i < 10
T7 = not T6
if T7 goto L6
goto L7
L8:
T8 = i + 1
i = T8
goto L5
L7:
T9 = a + 1
a = T9
goto L8
L6:
T10 = x  < b
T11 = not T10
if T11 goto L9
x = 10
goto L10
```

katikalyani@iampoor136: ~/PHASE-2

```
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
a = T3
goto L0
L1:
T4 = b < = c
T5 = not T4
if T5 goto L3
a = 10
goto L4
L3:
a = 20
L4:
a = 100
i = 0
L5:
T6 = i < 10
T7 = not T6
if T7 goto L6
goto L7
L8:
T8 = i + 1
i = T8
goto L5
L7:
T9 = a + 1
a = T9
goto L8
L6:
T10 = x  < b
T11 = not T10
if T11 goto L9
x = 10
goto L10
L9:
x = 11
L10:
Input accepted.
```

katikalyani@iampoor136: ~/PHASE-2

--------------------Quadruples------------------------

Operator	Arg1	Arg2	Result
*	a	b	T0
=	T0	(null)	b
Label	(null)	(null)	L0
>	a	b	T1
not	T1	(null)	T2
if	T2	(null)	L1
+	a	1	T3
=	T3	(null)	a
goto	(null)	(null)	L0
Label	(null)	(null)	L1
<=	b	c	T4
not	T4	(null)	T5
if	T5	(null)	L3
=	10	(null)	a
goto	(null)	(null)	L4
Label	(null)	(null)	L3
=	20	(null)	a
Label	(null)	(null)	L4
=	100	(null)	a
=	0	(null)	i
Label	(null)	(null)	L5
<	i	10	T6
not	T6	(null)	T7
if	T7	(null)	L6
goto	(null)	(null)	L7
Label	(null)	(null)	L8
+	i	1	T8
=	T8	(null)	i
goto	(null)	(null)	L5
Label	(null)	(null)	L7
+	a	1	T9
=	T9	(null)	a
goto	(null)	(null)	L8
Label	(null)	(null)	L6
<	x	b	T10
not	T10	(null)	T11
if	T11	(null)	L9

```
katikalyani@iampoor136: ~/PHASE-2
Label       (null)      (null)      L0
>           a           b           T1
not         T1          (null)      T2
if          T2          (null)      L1
+           a           1           T3
=           T3          (null)      a
goto        (null)      (null)      L0
Label       (null)      (null)      L1
<=          b           c           T4
not         T4          (null)      T5
if          T5          (null)      L3
=           10          (null)      a
goto        (null)      (null)      L4
Label       (null)      (null)      L3
=           20          (null)      a
Label       (null)      (null)      L4
=           100         (null)      a
=           0           (null)      i
Label       (null)      (null)      L5
<           i           10          T6
not         T6          (null)      T7
if          T7          (null)      L6
goto        (null)      (null)      L7
Label       (null)      (null)      L8
+           i           1           T8
=           T8          (null)      i
goto        (null)      (null)      L5
Label       (null)      (null)      L7
+           a           1           T9
=           T9          (null)      a
goto        (null)      (null)      L8
Label       (null)      (null)      L6
<           x           b           T10
not         T10         (null)      T11
if          T11         (null)      L9
=           10          (null)      x
goto        (null)      (null)      L10
Label       (null)      (null)      L9
=           11          (null)      x
Label       (null)      (null)      L10
```

Code Optimization

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives :

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process

Input:-

```
katikalyani@iampoor136:~/PHASE-2$ python3 codeopt.py
Quadruple form after Constant Folding
-------------------------------------
* a b T0
= T0 NULL b
Label (null) (null) L0
> a b T1
not T1 (null) T2
if T2 (null) L1
+ a 1 T3
= T3 NULL a
goto (null) (null) L0
Label (null) (null) L1
<= b c T4
not T4 (null) T5
if T5 (null) L3
= 10 NULL a
goto (null) (null) L4
Label (null) (null) L3
= 20 NULL a
Label (null) (null) L4
= 100 NULL a
= 0 NULL i
Label (null) (null) L5
< i 10 T6
not T6 (null) T7
if T7 (null) L6
goto (null) (null) L7
Label (null) (null) L8
= 1 NULL T8
= 1 NULL i
goto (null) (null) L5
Label (null) (null) L7
= 101 NULL T9
= 101 NULL a
goto (null) (null) L8
Label (null) (null) L6
< x b T10
not T10 (null) T11
if T11 (null) L9
```

katikalyani@iampoor136: ~/PHASE-2

```
Label (null) (null) L0
> a b T1
not T1 (null) T2
if T2 (null) L1
+ a 1 T3
= T3 NULL a
goto (null) (null) L0
Label (null) (null) L1
<= b c T4
not T4 (null) T5
if T5 (null) L3
= 10 NULL a
goto (null) (null) L4
Label (null) (null) L3
= 20 NULL a
Label (null) (null) L4
= 100 NULL a
= 0 NULL i
Label (null) (null) L5
< i 10 T6
not T6 (null) T7
if T7 (null) L6
goto (null) (null) L7
Label (null) (null) L8
= 1 NULL T8
= 1 NULL i
goto (null) (null) L5
Label (null) (null) L7
= 101 NULL T9
= 101 NULL a
goto (null) (null) L8
Label (null) (null) L6
< x b T10
not T10 (null) T11
if T11 (null) L9
= 10 NULL x
goto (null) (null) L10
Label (null) (null) L9
= 11 NULL x
Label (null) (null) L10
```

katikalyani@iampoor136: ~/PHASE-2

```
Label (null) (null) L10


Constant folded expression -
--------------------
T0 = a * b
b = T0
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
a = T3
goto L0
T4 = b <= c
T5 = not T4
if T5 goto L3
a = 10
goto L4
a = 20
a = 100
i = 0
T6 = i < 10
T7 = not T6
if T7 goto L6
goto L7
T8 = 1
i = 1
goto L5
T9 = 101
a = 101
goto L8
T10 = x < b
T11 = not T10
if T11 goto L9
x = 10
goto L10
x = 11
```

```
katikalyani@iampoor136: ~/PHASE-2
if T11 goto L9
x = 10
goto L10
x = 11


After dead code elimination -
------------------------------
T0 = a * b
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
goto L0
T4 = b <= c
T5 = not T4
if T5 goto L3
goto L4
T6 = i < 10
T7 = not T6
if T7 goto L6
goto L7
goto L5
goto L8
T10 = x < b
T11 = not T10
if T11 goto L9
goto L10
katikalyani@iampoor136:~/PHASE-2$
```

Assembly Code Generation:

Target code generated using a python script which reads the icg file and stores it line by line.Line by line the ARM statements are generated and registers are chosen in a round robin fashion(R%13) to ensure that registers are correctly used. For branch conditions, the condition statements are skipped and taken care off when we encounter the ifFalse statement. When the ifFalse statement is encountered then the previous statement which will be the conditional statement is converted into an ARM CMP statement and the ifFalse is converted to a B statement based on the condition.

Input:-

```
func begin main
t0 = 3
t1 = 2
L0:
t2 = x <= y
IF not t2 GoTo L1
refparam "hello world"
refparam result
call printf, 1
t3 = x + 1
x = t3
GoTo L0:
L1:
func end
~
```

Output:-

```
katikalyani@iampoor136:~/CG$ cat icg.s
.text
L0:
MOV R0,=t2
MOV R1,[R0]
MOV R2,=L1
MOV R3,[R2]
MOV R4,=IF
MOV R5,[R4]
STR R5, [R4]
refparam result
MOV R6,=x
MOV R7,[R6]
MOV R8,=t3
MOV R9,[R8]
ADD R9,#7,R1
STR R9, [R8]
GoTo L0:
L1:
func end
SWI 0x011
.DATA
func: .WORD main
t0: .WORD 3
t1: .WORD 2
refparam: .WORD world"
call: .WORD 1
x: .WORD t3
```