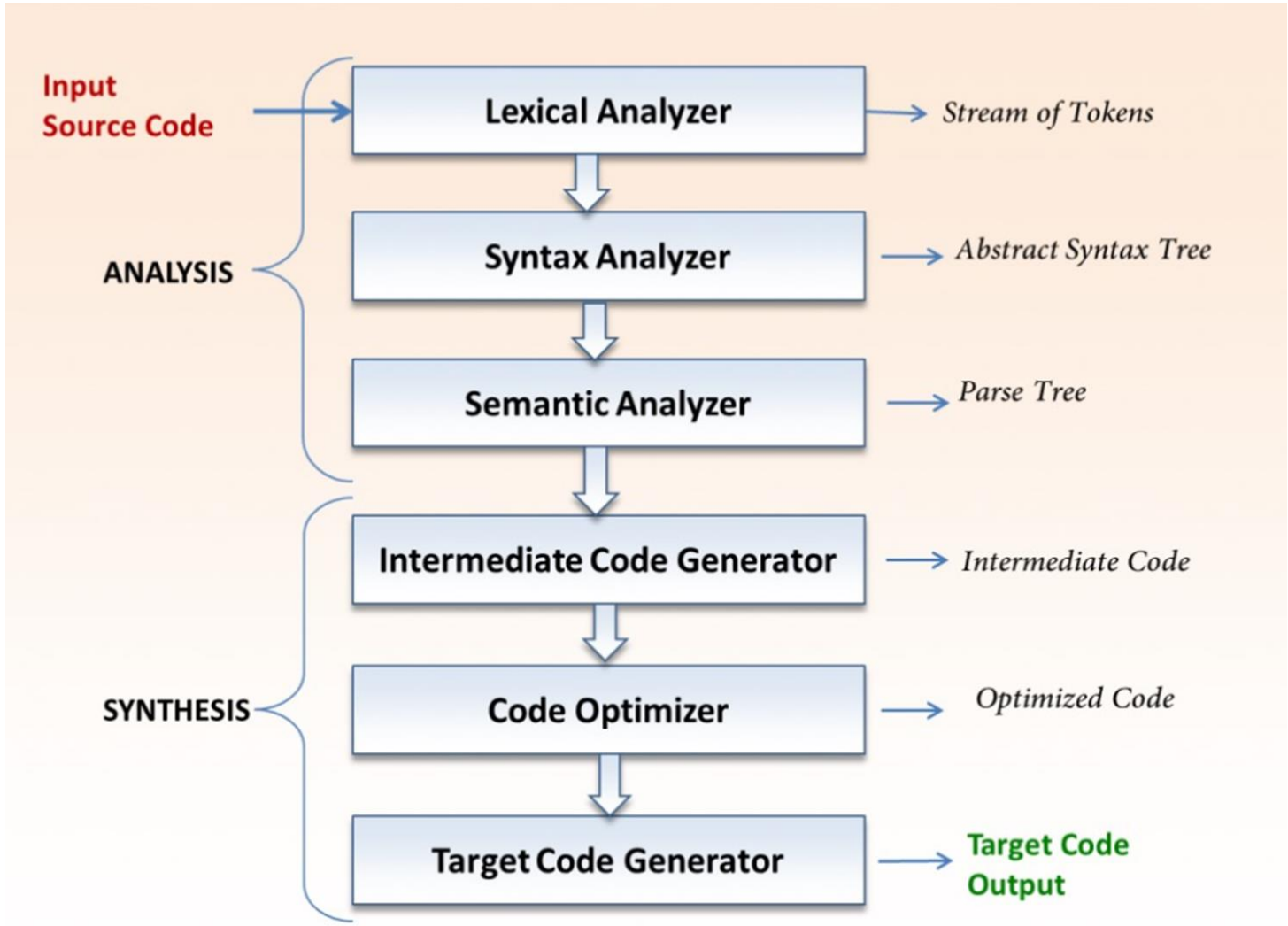
Visual Compiler Simulator: A Comprehensive Tool to Learn Compilation

- Team: VI-T121
- Subhanshu Raj (Team Leader)
- Shruti Malik,
- Raj Vikram Singh,
- Vishal Siwach,

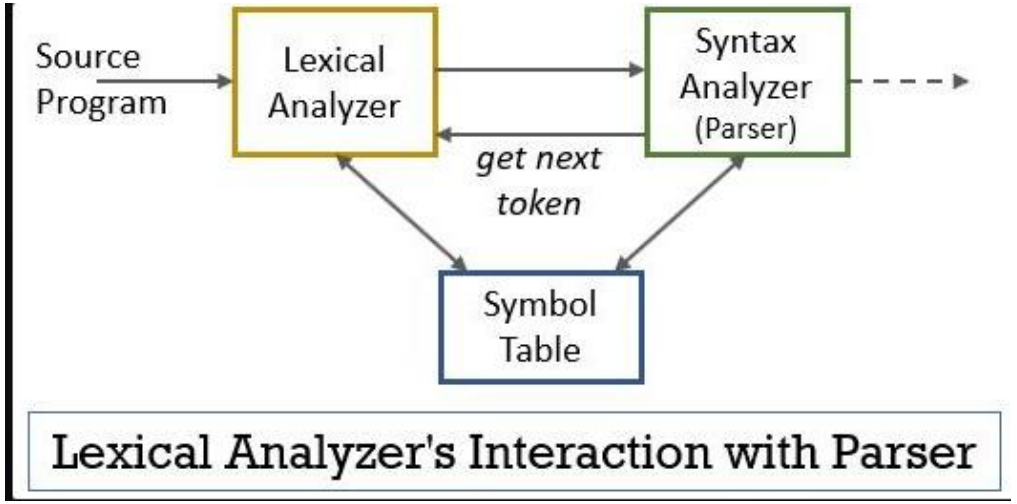
COMPILER

- A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses.



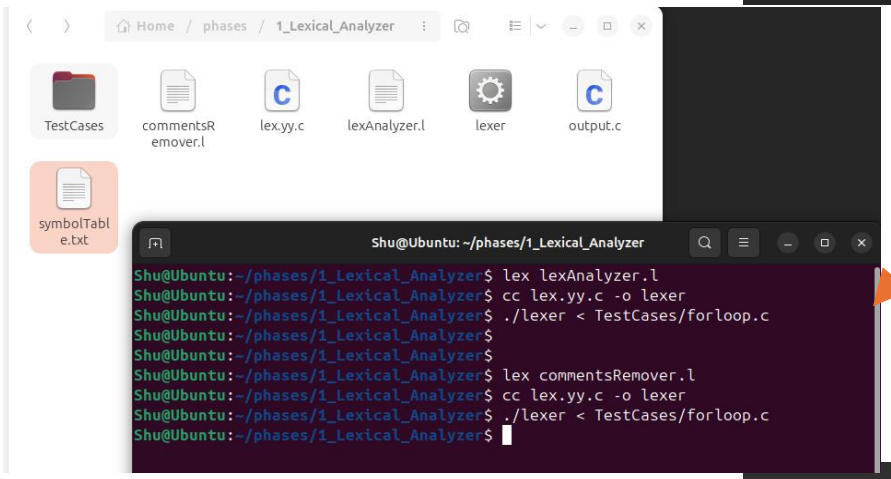
LEXICAL ANALYZER(-by Subhanshu Raj)

- It reads the pure high-level language (HLL) code one line at a time.
- It breaks the code into words (called lexemes) and turns them into tokens using a pattern-matching method called DFA (Deterministic Finite Automaton).
- It removes comments and extra spaces from the code.
- It helps to expand macros used in the code.
- It creates a symbol table to keep track of variable names, functions, and other identifiers.



Input

```c
// test case to check loop statements
#include<stdio.h>
int main(){

        int i, a, b;
        int nume=3.45;
        for(i = 0;  i < 10; i++){
                a=i;
        }
        i=1;
}
```



Code Run

Output.c

Open

```c

#include<stdio.h>
int main(){

        int i, a, b;
        int nume=3.45;
        for(i = 0;  i < 10; i++){
                a=i;
        }
        i=1;
}

```

Symbol Table

Table:

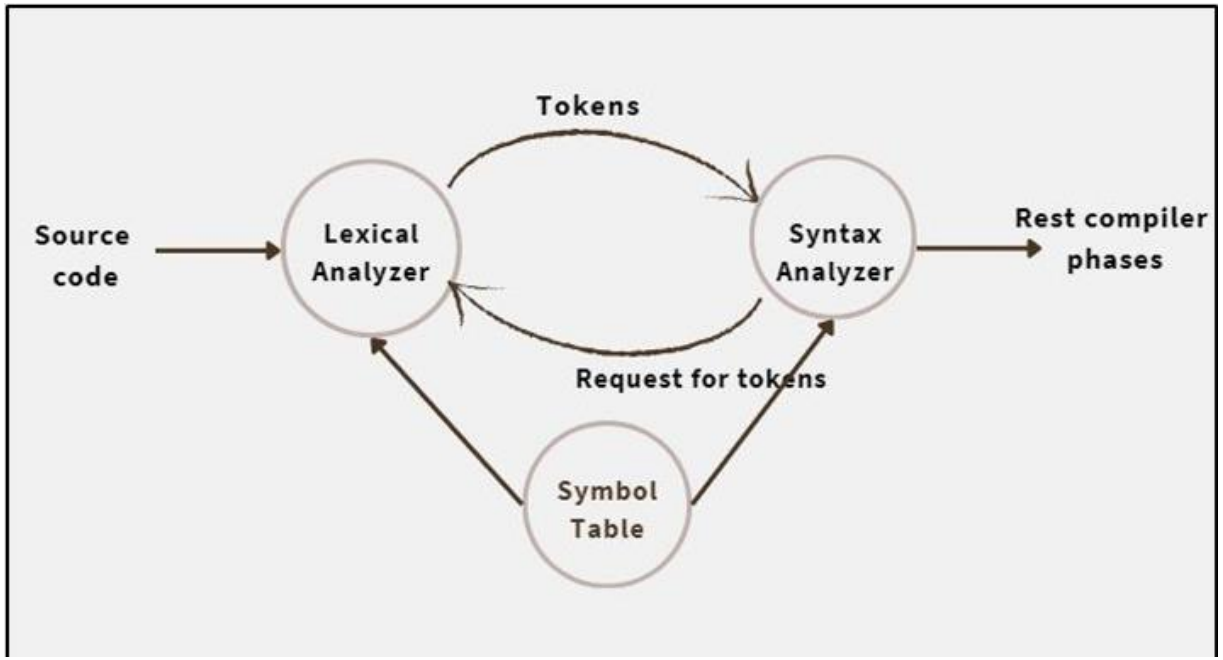
Lexeme	Token	Attribute Value	Line Number
#include<stdio.h>	Preprocessor Statement	0	2
int	Keyword	1	3
main	Procedure	2	3
{	Punctuator	3	3
int	Keyword	1	5
i	Identifier	4	5
,	Punctuator	5	5
a	Identifier	6	5
,	Punctuator	5	5
b	Identifier	7	5
;	Punctuator	8	5
int	Keyword	1	6
nume	Identifier	9	6
=	Assignment Op	10	6
3.45	Float Constant	11	6
;	Punctuator	8	6
for	Keyword	12	7
(	Punctuator	13	7
i	Identifier	4	7
=	Assignment Op	10	7
0	Integer Constant	14	7
;	Punctuator	8	7
i	Identifier	4	7
<	Relational Op	15	7
10	Integer Constant	16	7
;	Punctuator	8	7
i	Identifier	4	7
+	Arithmetic Op	17	7
+	Arithmetic Op	17	7
)	Punctuator	18	7
{	Punctuator	3	7
a	Identifier	6	8
=	Assignment Op	10	8
i	Identifier	4	8
;	Punctuator	8	8
}	Punctuator	19	9
i	Identifier	4	10
=	Assignment Op	10	10
1	Integer Constant	20	10
;	Punctuator	8	10
}	Punctuator	19	11

MultiLineComment (0 lines):

SingleLineComment :
 test case to check loop statements

SYNTAX ANALYZER (-by Shruti Malik)

- It checks for syntax errors in the code, like missing semicolons or unmatched brackets.
- It may need to clear up confusing grammar rules in the code.
- It uses parsing techniques like LL, LR, or Recursive Descent to understand the code structure.
- It builds an Abstract Syntax Tree (AST), which shows how different parts of the code are connected in a tree-like format.
- It deals with language features such as function declarations, definitions, and prototypes.



Input

```c
// test case to check loop statements
#include<stdio.h>
int main(){

        int i, a, b;
        int nume=3.45;
        for(i = 0;  i < 10; i++){
                a=i;
        }
        i=1;
}
```

Code Run

```
Shu@Ubuntu:~/phases/2_Syntax_Analyzer$ lex parseTree.l
Shu@Ubuntu:~/phases/2_Syntax_Analyzer$ yacc -d parseTree.y
parseTree.y:787.11-18: warning: POSIX Yacc does not support string literals [-Wyacc]
  787 |         | "INC_OP"      {       unaryop = 5;    }
      |           ^~~~~~~
parseTree.y:788.11-18: warning: POSIX Yacc does not support string literals [-Wyacc]
  788 |         | "DEC_OP"      {       unaryop = 6;    }
      |           ^~~~~~~
Shu@Ubuntu:~/phases/2_Syntax_Analyzer$ cc lex.yy.c y.tab.c -o parser
Shu@Ubuntu:~/phases/2_Syntax_Analyzer$ ./parser < TestCases/forloop.c

Line:6: 'float' to 'int'



ST

   SYMBOL               NAME      TYPE        SCOPE       LINE #             VALUE
 identifier               i       int           1             5                1
 identifier               a       int           1             5                0
 identifier               b       int           1             5                -
 identifier            nume       int           1             6                3
```

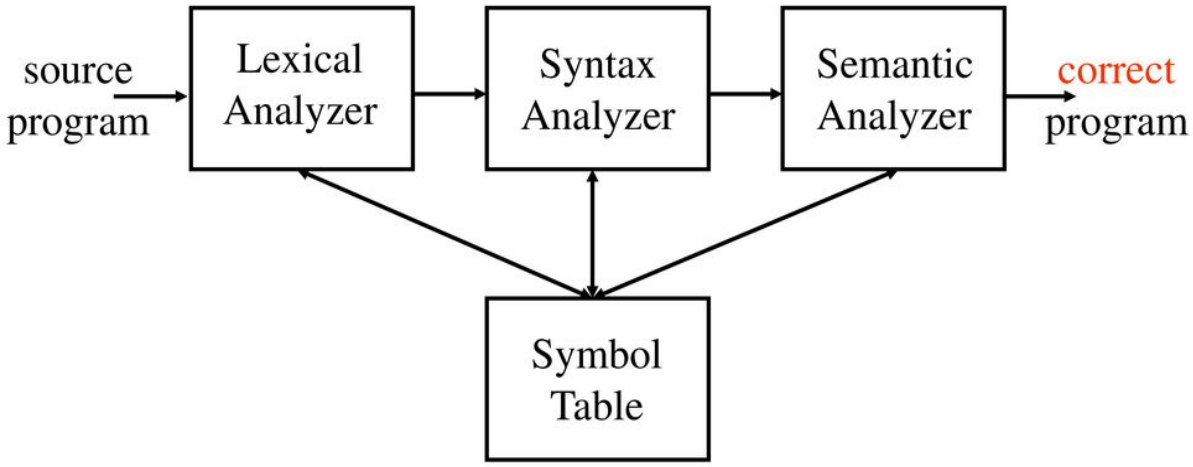
Parse Tree Visualization

- Parse tree will be printed in the terminal with its preorder traversal.

```
Parse Tree
                                                                    main

                                                                stmt

                                                        stmt            =

                                                =           for         i           1

                                        i           0           ++          =

                                                <           i           a           i

                                    i           10


Preorder Traversal of Parse Tree:
main      ( stmt      ( stmt      ( =   i   0   )     ( for     ( ++     ( <   i   10   )   i   )     ( =   a   i
)   )   )     ( =   i   1   )   )
Shu@Ubuntu:~/phases/2_Syntax_Analyzer$
```

SEMANTIC ANALYZER(-by Raj Vikram)

- It checks data types to make sure that operations (like addition or comparison) are done between compatible types (e.g., you can't add a number to a string).
- It manages scopes and namespaces, so that each variable or function is used only where it's allowed.
- It finds errors in meaning (called semantic errors), like using a variable that was never declared or using the wrong type.
- It makes sure data types are used correctly, based on how they were defined.
- It checks that control structures (like if-else, loops) are used properly — for example, making sure there is no break statement outside of a loop.



Input

```
// test case to check loop statements
#include<stdio.h>
int main(){

        int i, a, b;
        int nume=3.45;
        for(i = 0;  i < 10; i++){
                a=i;
        }
        i=1;
}
```

Printing Symbol Table

- In this phase, we extract necessary semantic information from the source code which is impossible to detect in parsing.

Code Run

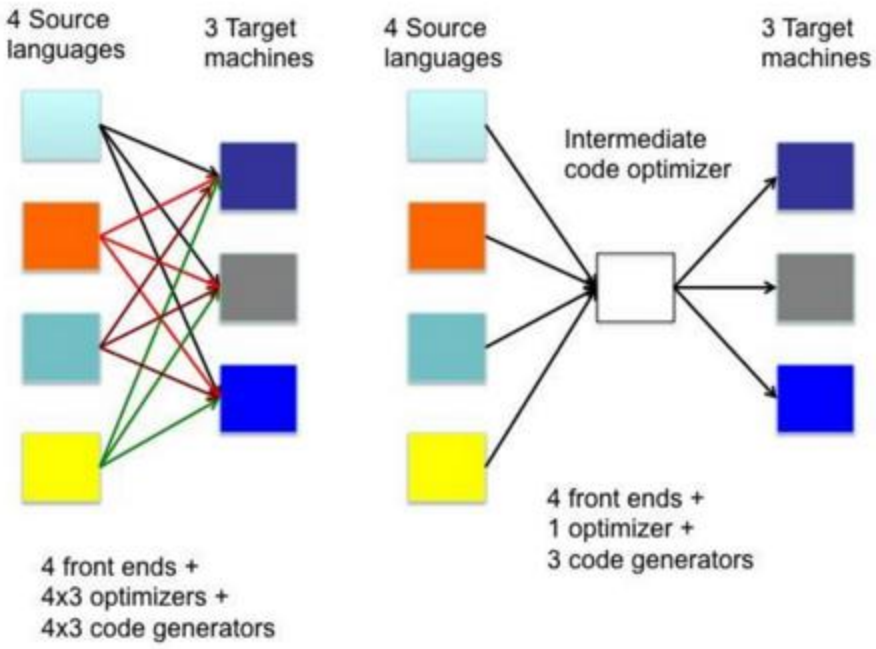
```
Shu@Ubuntu: ~/phases/3_Semantic_Analyzer
Shu@Ubuntu:~/phases/2_Syntax_Analyzer$ cd ..
Shu@Ubuntu:~/phases$ cd ~/phases/3_Semantic_Analyzer
Shu@Ubuntu:~/phases/3_Semantic_Analyzer$ lex scanner.l
Shu@Ubuntu:~/phases/3_Semantic_Analyzer$ yacc -d parser.y
parser.y:50.1-7: warning: POSIX Yacc does not support %expect [-Wyacc]
   50 | %expect 1
      | ^~~~~~~
Shu@Ubuntu:~/phases/3_Semantic_Analyzer$ cc lex.yy.c y.tab.c -o parser
Shu@Ubuntu:~/phases/3_Semantic_Analyzer$ ./parser <TestCases/forloop.c
PASSED: Semantic Phase
```

```
                                PRINTING SYMBOL TABLE

symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |     Identifier |        int |            |          5 |           99999 |              -1 |
          b |     Identifier |        int |            |          5 |           99999 |              -1 |
          i |     Identifier |        int |          1 |          5 |           99999 |              -1 |
        for |        Keyword |            |            |          7 |            9999 |              -1 |
       main |       Function |        int |            |          3 |            9999 |              -1 |
       nume |     Identifier |        int |       3.45 |          6 |           99999 |              -1 |
        int |        Keyword |            |            |          3 |            9999 |              -1 |
                              -------------------------------


                                PRINTING CONSTANT TABLE

constant name |   constant type
_____________________________________________________________________________
       3.45 | Floating Constant
         10 | Number Constant
          0 | Number Constant
          1 | Number Constant
Shu@Ubuntu:~/phases/3_Semantic_Analyzer$
```

Printing Constant Table

INTERMEDIATE CODE GENERATOR(-by Vishal Siwach)

- It can create intermediate versions of the code like AST (Abstract Syntax Tree), Quadruples, or DAG (Directed Acyclic Graph) to help with further processing.
- It understands and processes complex parts of the code, like loops and if-else statements.
- It simplifies and restructures the code to get it ready for optimization.
- It improves how control flow (like loops and conditions) works to make the code run more efficiently.
- It creates temporary variables to store intermediate results during computation.
- It takes care of function calls and how parameters are passed in them.
- It rewrites expressions into a simpler or cleaner form so they can be optimized more easily.



Input

```
// test case to check loop statements
#include<stdio.h>
int main(){

        int i, a, b;
        int nume=3.45;
        for(i = 0;  i < 10; i++){
                a=i;
        }
        i=1;
}
```

Output will be in ICG.txt file

- We are trying to generate language independent three-address code for a given source program which is lexically, syntactically and semantically correct

Code Run

```
Shu@Ubuntu: ~/phases/4_Intermediate_Code_Generator
Shu@Ubuntu:~/phases/4_Intermediate_Code_Generator$ lex ICG.l
Shu@Ubuntu:~/phases/4_Intermediate_Code_Generator$ yacc -d ICG.y
ICG.y:783.11-18: warning: POSIX Yacc does not support string literals [-Wyacc]
  783 |          | "INC_OP"      {       unaryop = 4; }
      |           ^~~~~~~
ICG.y:784.11-18: warning: POSIX Yacc does not support string literals [-Wyacc]
  784 |          | "DEC_OP"      {       unaryop = 5; }
      |           ^~~~~~~
ICG.y: warning: 120 shift/reduce conflicts [-Wconflicts-sr]
ICG.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
Shu@Ubuntu:~/phases/4_Intermediate_Code_Generator$ cc lex.yy.c y.tab.c -o parser
Shu@Ubuntu:~/phases/4_Intermediate_Code_Generator$ ./parser <TestCsese/forloop.c
bash: TestCsese/forloop.c: No such file or directory
Shu@Ubuntu:~/phases/4_Intermediate_Code_Generator$ ./parser <TestCase/forloop.c
bash: TestCase/forloop.c: No such file or directory
Shu@Ubuntu:~/phases/4_Intermediate_Code_Generator$ ./parser <TestCases/forloop.c
Shu@Ubuntu:~/phases/4_Intermediate_Code_Generator$
```

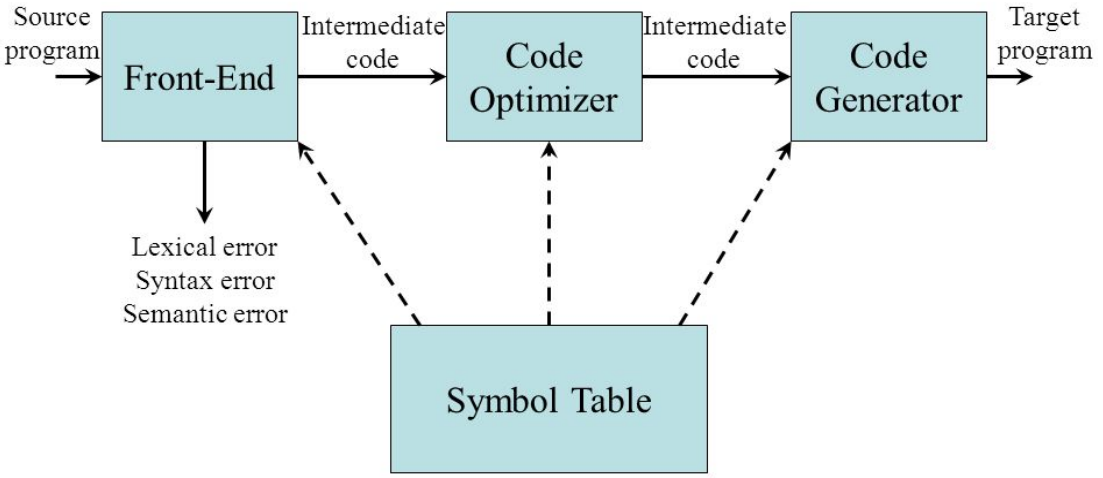
Open

icg.txt

```
1  i = 0
2  a = 0
3  i = 0
4  L0:
5  t0 = i < 10
6  ifFalse t0 goto L1
7  a = t0
8  t1 = i + 1
9  i = t1
10 goto L0
11 L1:
12 i = t1
```

CODE OPTIMIZER(- by Shruti Malik)

- Exploits data locality for memory access optimization.
- Applies loop transformations such as loop unrolling and loop fusion.
- Utilizes profile-guided optimization for performance improvements.
- Considers instruction scheduling to minimize pipeline stalls.
- Incorporates inline expansion to reduce function call overhead.
- Implements loop vectorization for exploiting SIMD (Single Instruction, Multiple Data) instructions.
- Applies interprocedural optimizations across multiple translation units.
- Considers speculative execution and branch prediction strategies.



Paste ICG generated in previous phase in input.txt

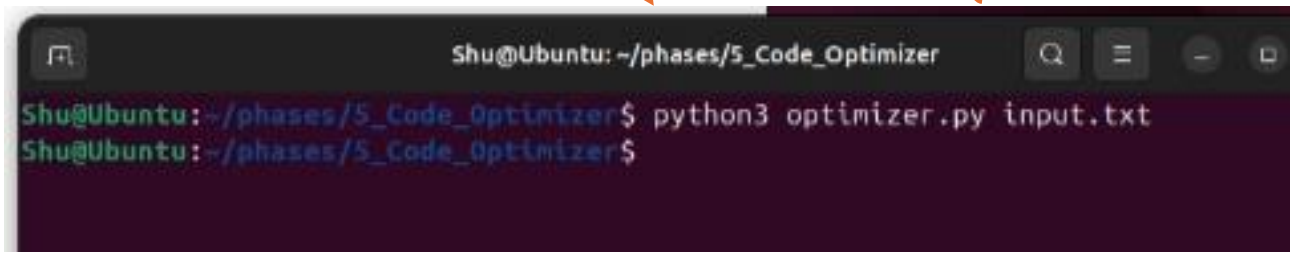
Output will be in optimized_icg txt file

- In this phase, the machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

5_Code_Optimizer > input.txt

```
1   i = 0
2   a = 0
3   i = 0
4   L0:
5   t0 = i < 10
6   ifFalse t0 goto L1
7   a = t0
8   t1 = i + 1
9   i = t1
10  goto L0
11  L1:
12  i = t1
13
```

Code Run



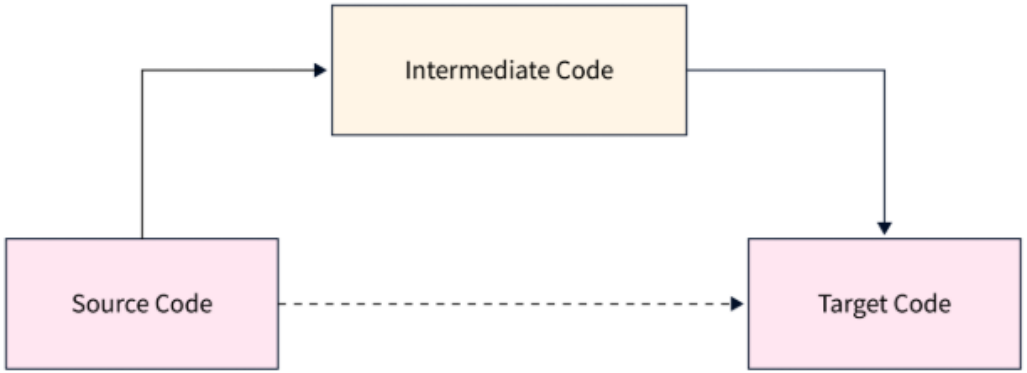
optimized_icg.txt

5_Code_Optimizer > optimized_icg.txt

```
1   i = 0
2   a = 0
3   i = 0
4   L0:
5   t0 = True
6   ifFalse t0 goto L1
7   a = t0
8   t1 = 1
9   i = t1
10  goto L0
11  L1:
12  i = t1
13
```

TARGET CODE GENERATOR(-by Subhanshu Raj)

- Chooses the right way to access memory and place data.
- Allocates registers and handles extra variables if registers are full.
- Picks and arranges instructions best suited for the CPU.
- Uses CPU-specific features and instructions.
- Supports binary formats like ELF or COFF for different systems.
- Handles symbols and memory addresses for linking.
- Can create position-independent code for shared libraries.
- Adds runtime support for things like errors and memory use.



Paste optimized_icg.txt generated in previous phase in icg.txt

icg.txt

```
1  i = 0
2  a = 0
3  i = 0
4  L0:
5  t0 = i < 10
6  ifFalse t0 goto L1
7  a = t0
8  t1 = i + 1
9  i = t1
10 goto L0
11 L1:
12 i = t1
```

Code Run

```
Shu@Ubuntu: ~/phases/6_Target_Code_Generator
Shu@Ubuntu:~/phases/6_Target_Code_Generator$ python3 assembly.py icg.txt
Compiling.....
Assembly code dumped to:  icg.s
Shu@Ubuntu:~/phases/6_Target_Code_Generator$
```

Output will be in icg.s

- The final phase in our compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces semantically equivalent target programs.

6_Target_Code_Generator > ASM icg.s

```
1   .text
2   MOV R0,=i
3   MOV R1,[R0]
4   MOV R2,#0
5   STR R2, [R0]
6   L0:
7   MOV R3,=i
8   MOV R4,[R3]
9   CMP R4,#10
10  BGE L1
11  MOV R5,=a
12  MOV R6,[R5]
13  MOV R7,#t0
14  STR R7, [R5]
15  MOV R8,=i
16  MOV R9,[R8]
17  MOV R10,=t1
18  MOV R11,[R10]
19  ADD R11,#9,R1
20  STR R11, [R10]
21  MOV R12,=i
22  MOV R0,[R12]
23  MOV R1,#t1
24  STR R1, [R12]
25  B L0
26  L1:
27  MOV R2,=i
28  MOV R3,[R2]
29  MOV R4,#t1
30  STR R4, [R2]
31  SWI 0x011
32  .DATA
33  i: .WORD 0
34  a: .WORD 0
```

Thanks

Q&A

- Open the floor for questions.
- Thank you for your attention.

-Subhanshu Raj
-Shruti Malik
-Raj Vikram Singh
-Vishal Siwach