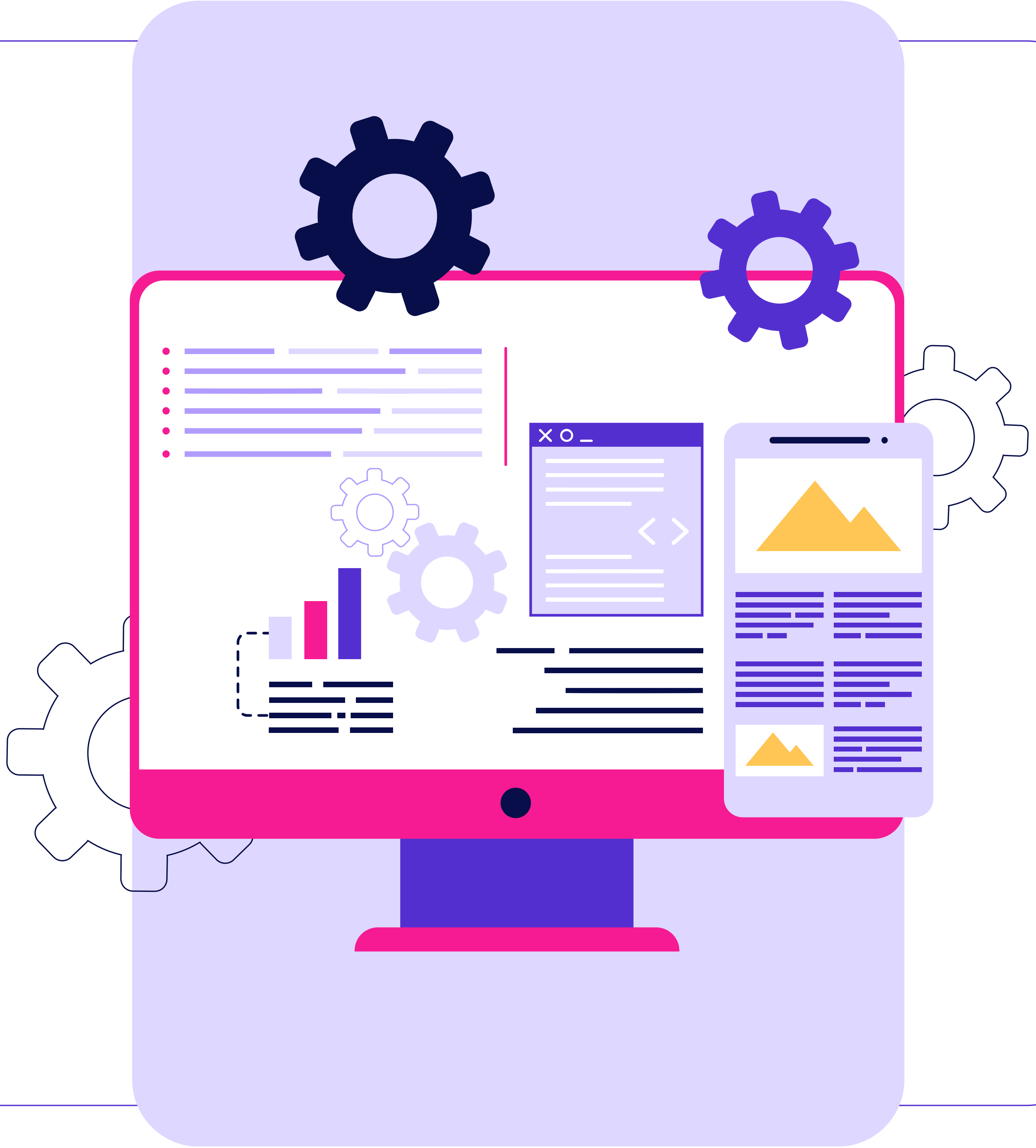
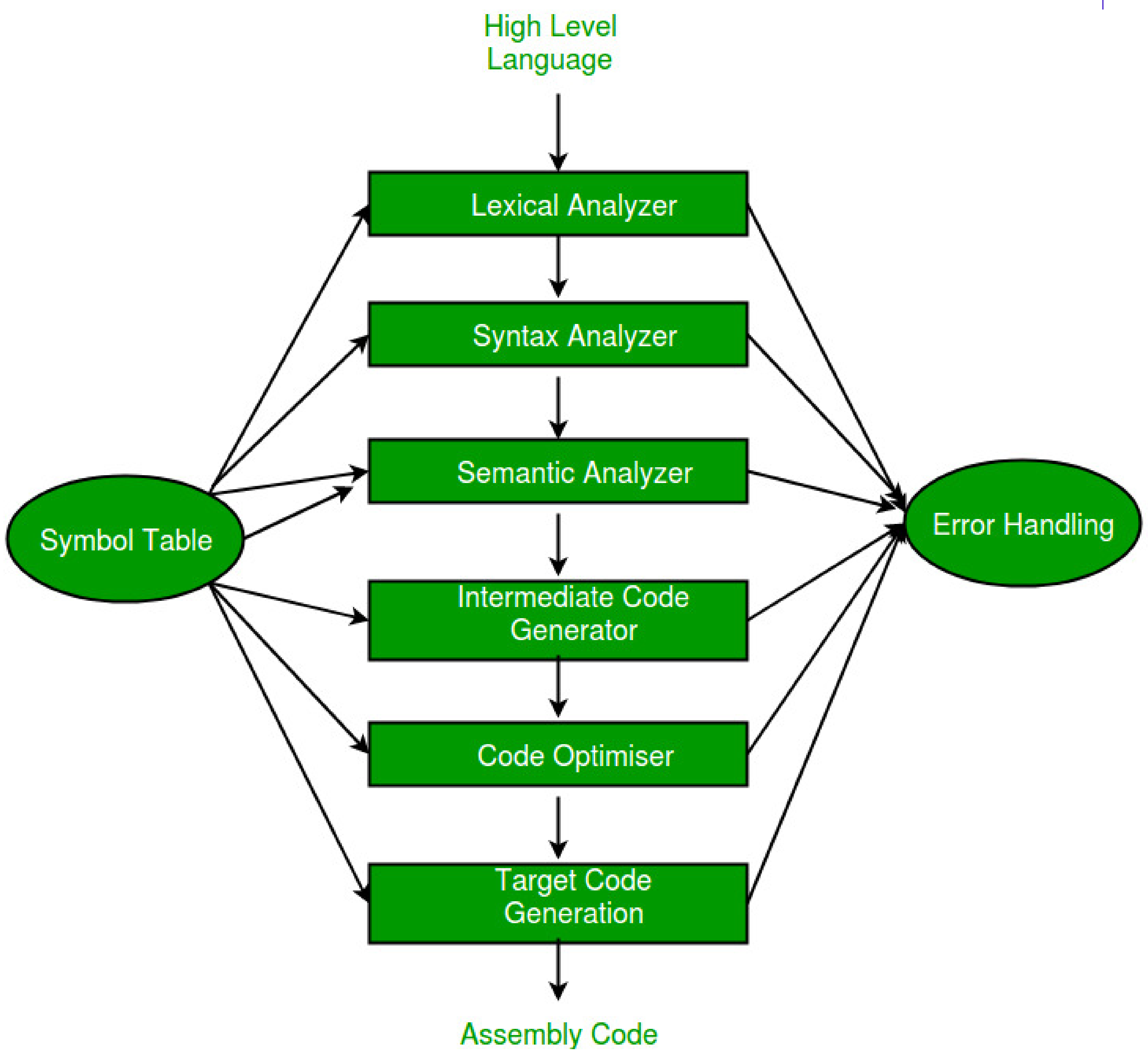
IT250 MINI PROJECT

Nithin S 221IT085
Ayush Kumar 221IT015
Jay Chavan 221IT020



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.



PHASES OF A COMPILER

01 Lexical Analyzer

02 Syntax Analyzer

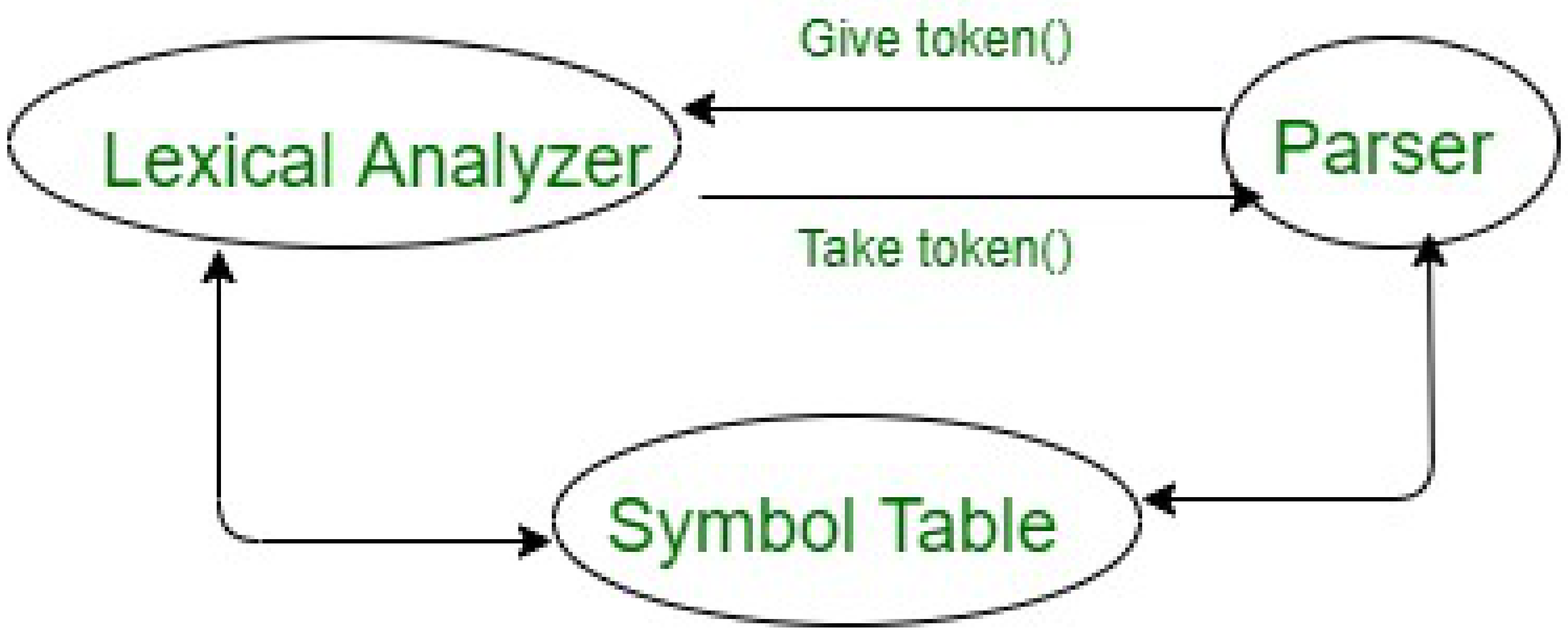
03 Semantic Analyzer

04 Intermediate Code Generator

05 Code Optimizer

06 Target Code Generator

LEXICAL ANALYZER



- Scans the Pure High Level Language Code Line by Line
- Takes Lexemes as input and produces Tokens using DFA for pattern matching
- Removes Comments and Whitespaces from the Pure High Level Code
- Helps in macro expansion in the Pure HLL Code
- Creates a Symbol Table

Input

```c
// test case to check loop statements

int main(){

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

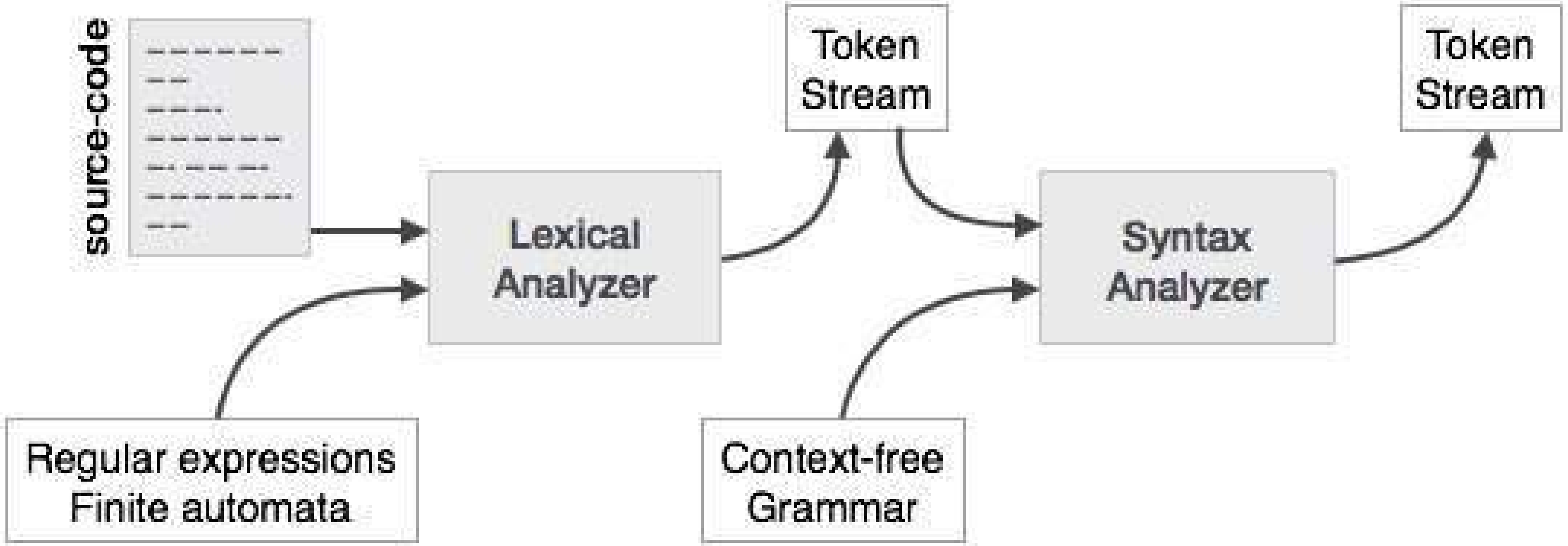
Output

Table:

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

```
MultiLineComment (0 lines):

SingleLineComment :
 test case to check loop statements
```

SYNTAX ANALYZER



- Checks for syntactic errors like missing semicolons, mismatched parentheses, etc.
- May involve resolving ambiguities in the grammar.
- Implements parsing algorithms such as LL, LR, or Recursive Descent.
- Constructs an Abstract Syntax Tree (AST) representing the hierarchical structure of the code.
- Handles language features like function prototypes, declarations, and definitions.

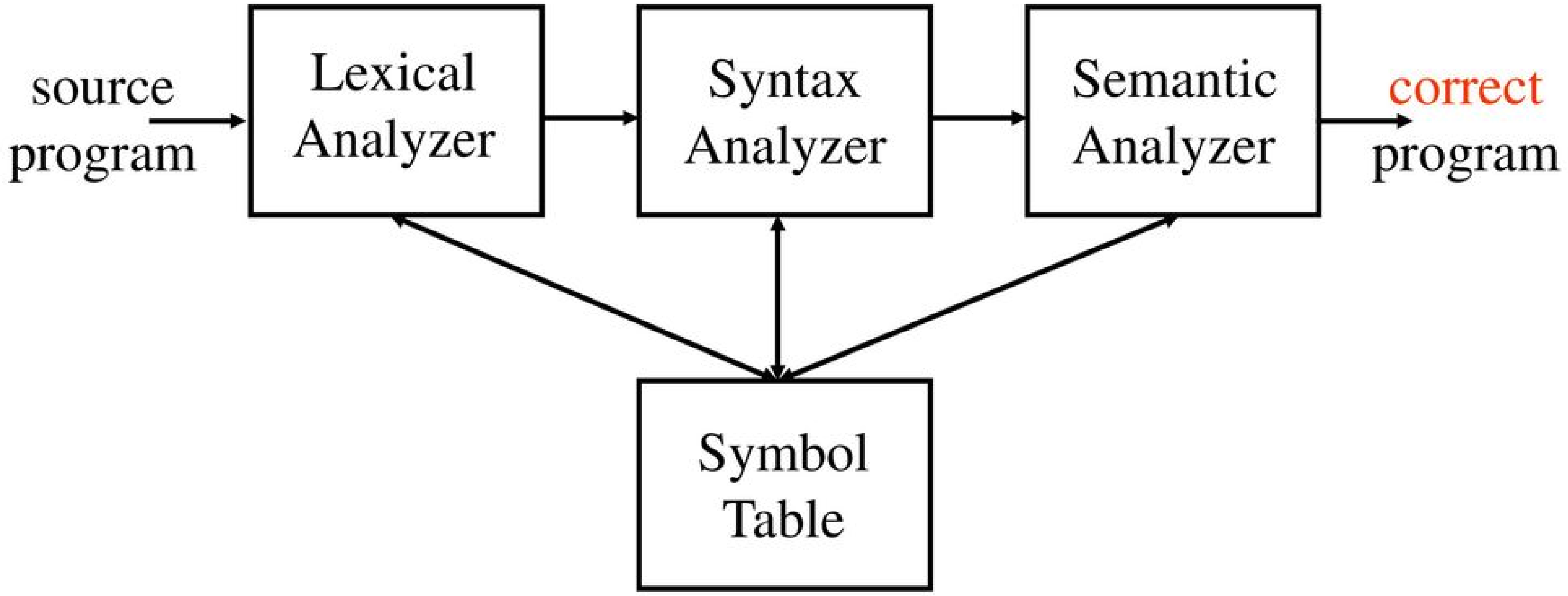
Input

```c
// test case to check loop statements

int main(){

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

Output

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/2_Syntax_Analyzer$ lex parseTree.l
nithin@nithin1729s:~/Codes/Projects/C Compiler 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;    }
      |           ^~~~~~~~
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/2_Syntax_Analyzer$ cc lex.yy.c y.tab.c
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/2_Syntax_Analyzer$ ./a.out < 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
                                                                              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   )   )
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/2_Syntax_Analyzer$
```

SEMANTIC ANALYZER



- Performs type checking to ensure operations are performed on compatible types.
- Handles scope and namespace resolution.
- Detects and reports semantic errors such as type mismatches or undeclared variables.
- Ensures that data types are used in a way consistent with their definition.
- Keeps a check that control structures are used in a proper manner. (example: no break statement outside a loop)

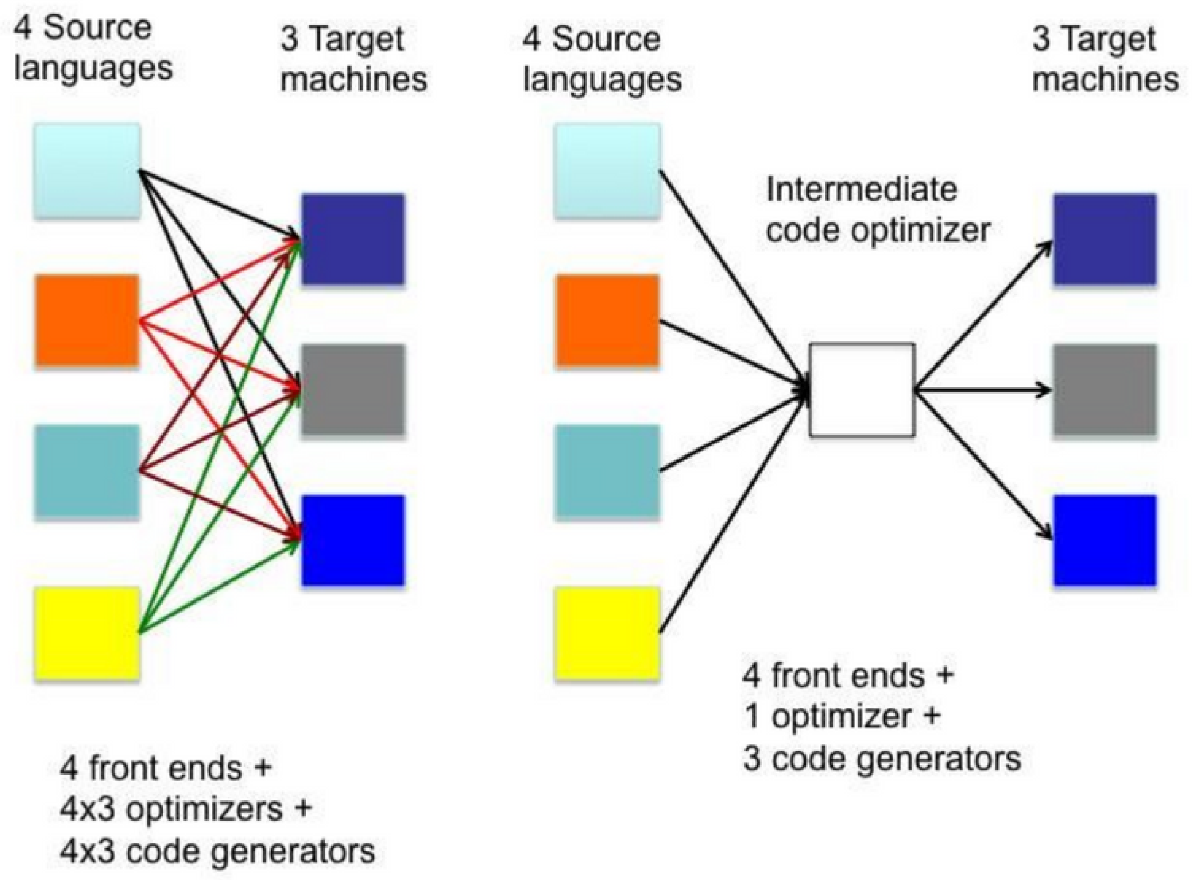
Input

```c
// test case to check loop statements

int main(){

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

Output

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$ ./a.out < 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
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$
```

INTERMEDIATE CODE GENERATOR



- Can generate different intermediate representations like Abstract Syntax Trees (AST), Quadruples, or Direct Abstract Graphs (DAG).
- Handles complex language constructs like loops and conditionals.
- Prepares the code for optimization by simplifying and restructuring it.
- Optimizes control flow structures like loops and conditional statements.
- Generates temporaries for intermediate results.
- Handles function calls and parameter passing mechanisms.
- Converts expressions into a more manageable form for optimization

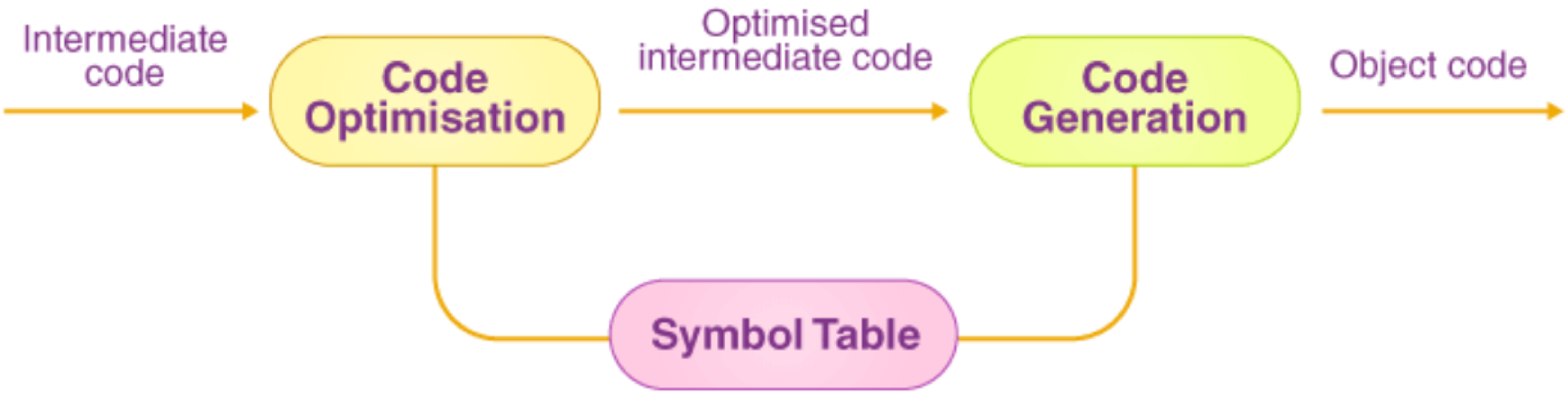
Input

```
// test case to check loop statements

int main(){

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

Output

```
nume = 3.45
i = 0
L0:
t0 = i < 10
ifFalse t0 goto L1
a = t0
t1 = i + 1
i = t1
goto L0
L1:
i = t1
```

CODE OPTIMIZER



- 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.

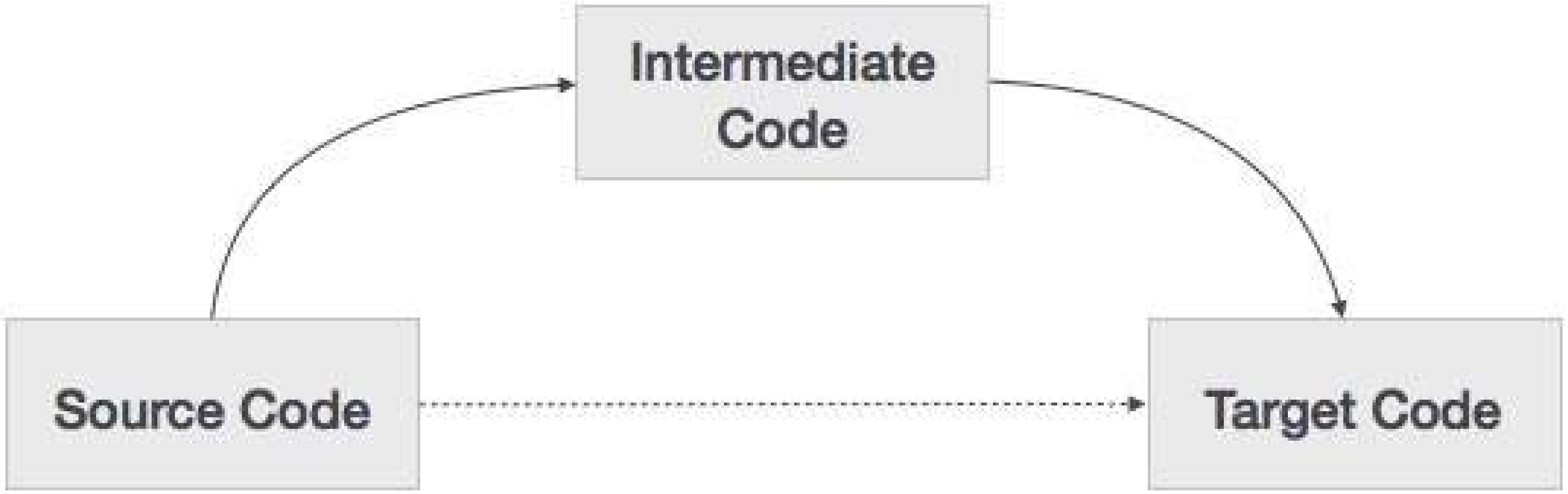
Input

```
// test case to check loop statements

int main(){

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

Output

```
nume = 3.45
i = 0
L0:
t0 = True
ifFalse t0 goto L1
a = t0
t1 = 1
i = t1
goto L0
L1:
i = t1
```

TARGET CODE GENERATOR



- Resolves addressing modes and memory layout considerations.
- Manages register allocation and spill code generation.
- Optimizes instruction selection and scheduling for the target architecture.
- Integrates platform-specific instruction sets and features.
- Handles platform-specific binary formats such as ELF or COFF.
- Manages symbol resolution and relocation for linking.
- Supports generation of position-independent code (PIC) for shared libraries.
- Integrates runtime support for exception handling and dynamic memory management.

Input

```
// test case to check loop statements

int main(){

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

Output

```
.text
L0:
MOV R0,=i
MOV R1,[R0]
CMP R1,#10
BGE L1
MOV R2,=i
MOV R3,[R2]
MOV R4,=t1
MOV R5,[R4]
ADD R5,#3,R1
STR R5, [R4]
MOV R6,=i
MOV R7,[R6]
MOV R8,#t1
STR R8, [R6]
B L0
L1:
MOV R9,=i
MOV R10,[R9]
MOV R11,#t1
STR R11, [R9]
SWI 0x011
.DATA
nume: .WORD 3.45
i: .WORD 0
a: .WORD t0
```

Conclusion

- With the lex and yacc tool one can create its own compiler, wherever one is required.

- It is basically procedural language compiler tools and to support object oriented one need to work on structure of C language to support object oriented which makes the compiler quite complex.

- To use lex and yacc on UNIX is easy as compared to other operating systems. Gcc is the basic compiler to generate the executable from lex and yacc compiled files.

- By studying these tools one can understand the basic structure of the compiler designed in C and go forward from it.

Future Scope

- Machine Learning Based Optimization
- Quantum Computing Compilation
- Heterogenous Computing Compilation
- High Level Synthesis
- Domain Specific Compilation

THANK YOU