Compiler Design

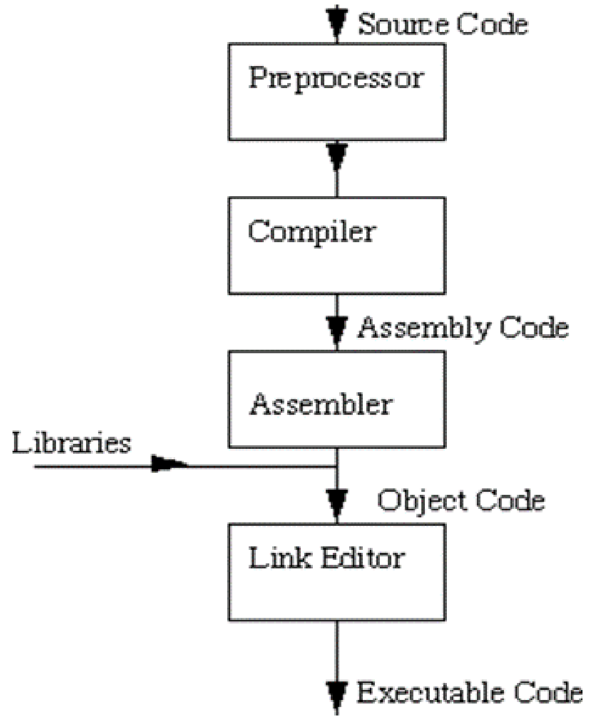
Language Processor

- A **language processor** is a software program designed or used to perform tasks such as processing program code to machine code.
- Types of Language Processor are:
  - Interpreter
  - Translator

Interpreter

- An interpreter is a language processor. Instead of producing a target program as a single translation unit, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Language Processing System

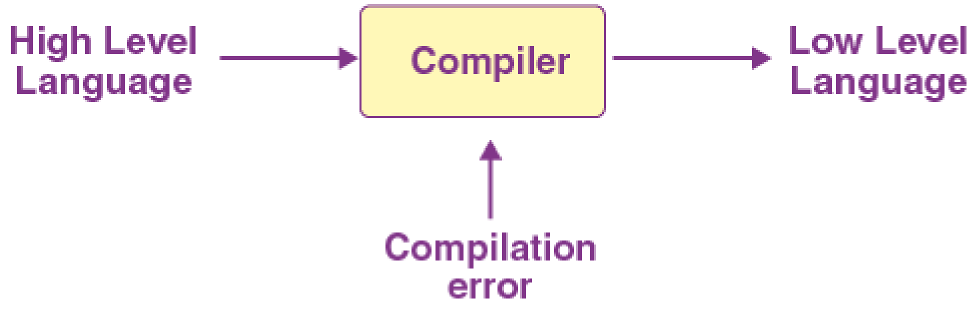


Translator

- Translator translates the high-level program code into machine code, allowing the computer to read and understand what tasks the program needs to be done in its native code.
- An assembler and a compiler are examples of translators.

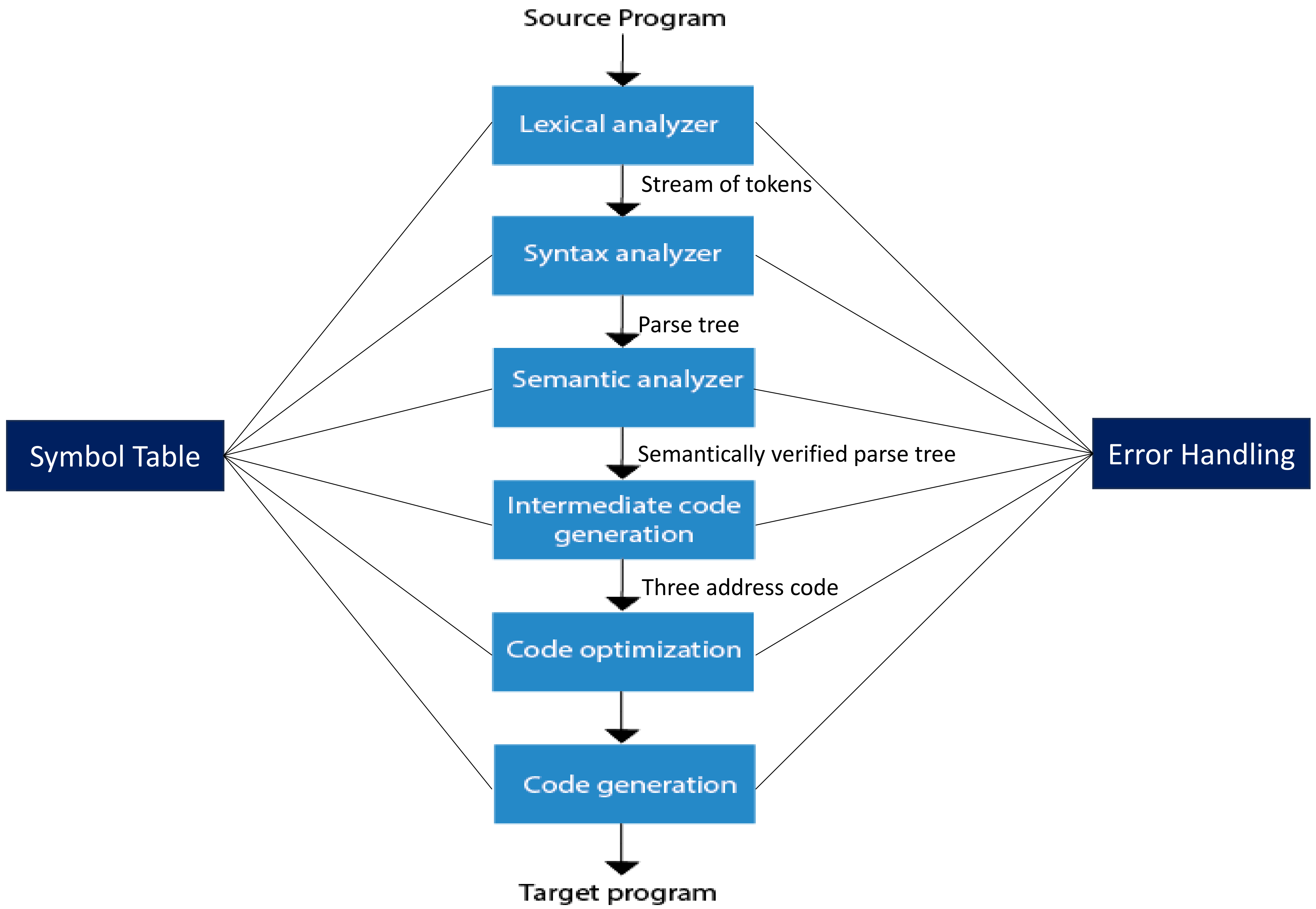
Compiler

- A compiler is a computer program that translates the source code composed in a high-level programming language into the machine code.



Compiler Phases

- The compilation process contains the sequence of various phases.
- Each phase takes source program in one representation and produces output in another representation.
- Each phase takes input from its previous stage.

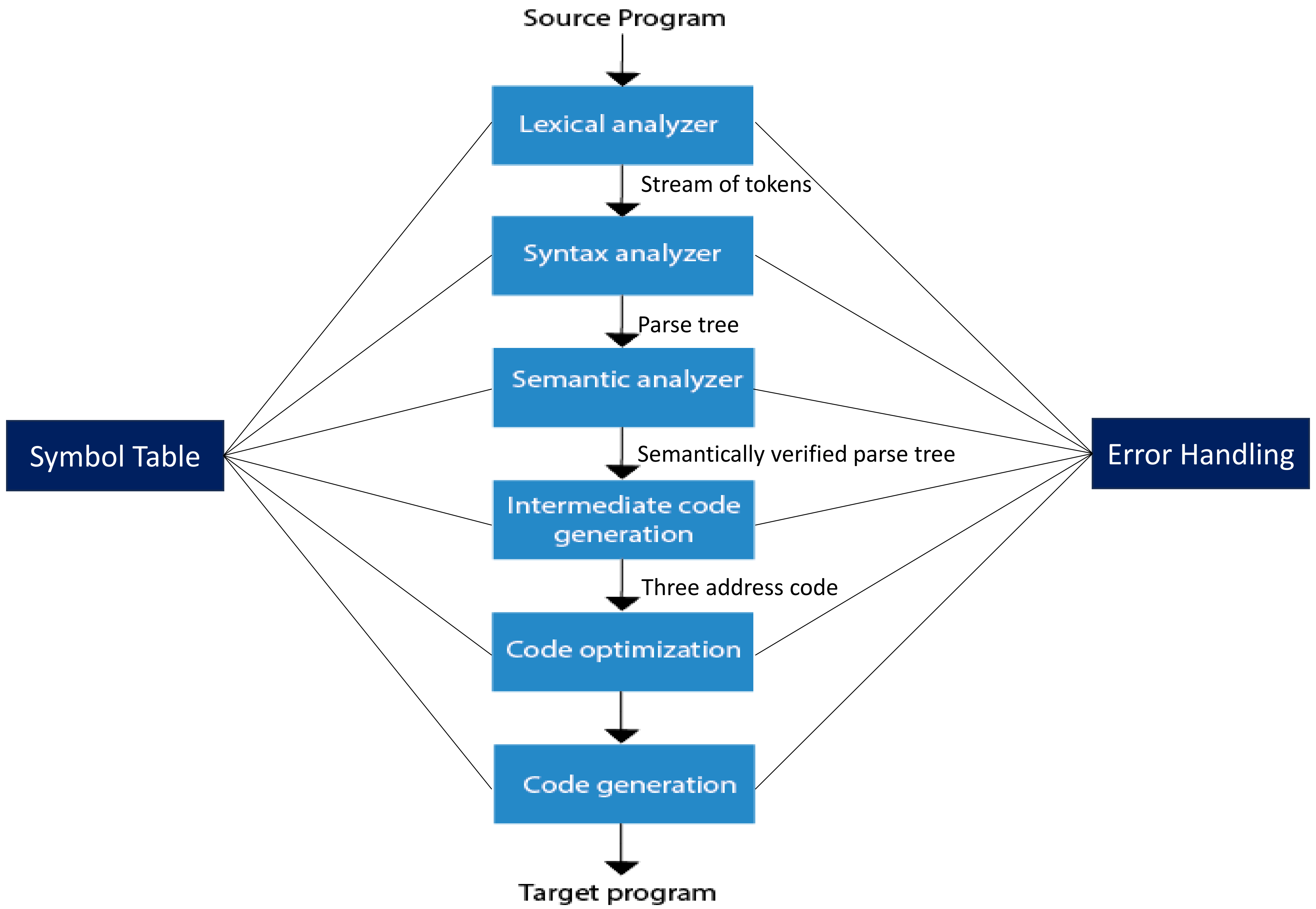

Source Program
Lexical analyzer
Stream of tokens
Syntax analyzer
Parse tree
Semantic analyzer
Semantically verified parse tree
Intermediate code generation
Three address code
Code optimization
Code generation
Target program
Symbol Table
Error Handling

Pass

- A pass is a collection of phases that convert the input from one representation to a completely deferent representation.
- Each pass makes a complete scan of the input and produces its output to be processed by the subsequent pass.

FRONT-END & BACK-END OF A COMPILER

- The phases of a compiler conceptually divided into two parts as:
  - Front-End Compiler
  - Back-End Compiler
- This division is due to the dependency of the compiler on source language or on the target machine. This model is called an Analysis & Synthesis model of a compiler.
- The **Front-end of the compiler** consists of phases that depend primarily on the Source language and are largely independent on the target machine. For example, front-end of the compiler includes Scanner, Parser, Creation of Symbol table, Semantic Analyzer, and the Intermediate Code Generator.
- The **Back-end of the compiler** consists of phases that depend on the target machine, and those portions don‘t dependent on the Source language, just the Intermediate language. In this we have different aspects of Code Optimization phase, code generation along with the necessary Error handling, and Symbol table operations.


Source Program
Lexical analyzer
Stream of tokens
Syntax analyzer
Parse tree
Semantic analyzer
Semantically verified parse tree
Intermediate code generation
Three address code
Code optimization
Code generation
Target program
Symbol Table
Error Handling

Lexical Analysis

- Lexical analyzer phase is the first phase of compilation process.
- It takes source code as input.
- It reads the source program one character at a time and converts it into meaningful lexemes.
- Lexical analyzer represents these lexemes in the form of tokens.

Example:

```
Res = x+y*3
```

Lexical Analyzer

- Res = x+y*3 (Source code)   id1 = id2+id3*3

Tokens

Res	Identifier
=	Assignment operator
X	Identifier
+	Arithmetic operator
Y	Identifier
*	Arithmetic operator
3	Number

Syntax Analysis

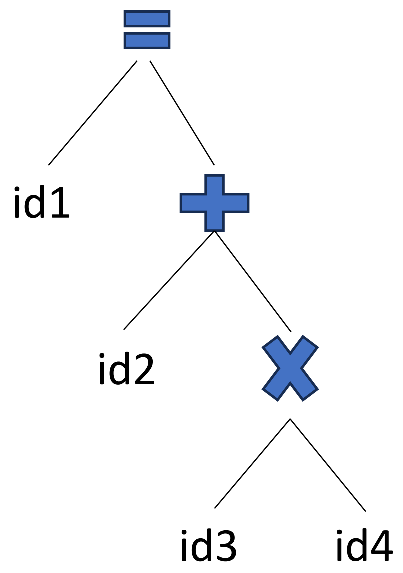
- Syntax analysis is the second phase of compilation process.
- It takes tokens as input and generates a parse tree as output.
- In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

Example:

Res = x+y*3

Syntax Analyzer

- Res = x+y*3 (Source code) id1 = id2+id3*3



Semantic Analysis

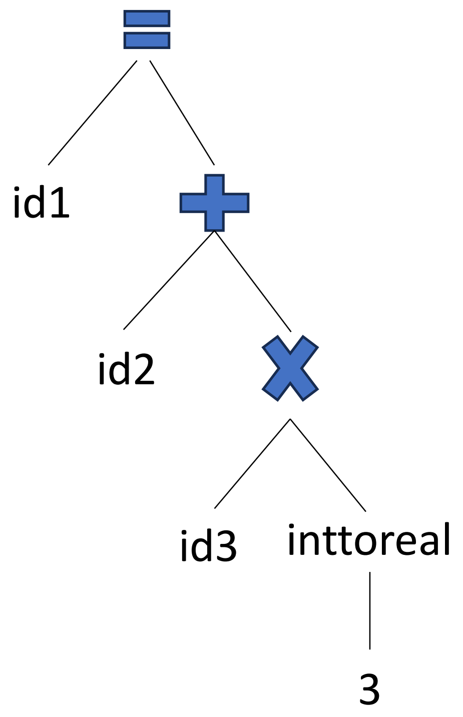
- Semantic analysis is the third phase of compilation process.
- It checks whether the parse tree follows the rules of language.
- Semantic analyzer keeps track of identifiers, their types and expressions.
- The output of semantic analysis phase is the annotated tree syntax.

Example:

Res = x+y*3

Semantic Analyzer

- Res = x+y*3 (Source code) id1 = id2+id3*3



Intermediate code generation

- In the intermediate code generation, compiler generates the source code into the intermediate code.
- Intermediate code is generated between the high-level language and the machine language.
- The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

Example:

Res = x+y*3

Intermediate code generator

- Res = x+y*3 (Source code)

id1 = id2+id3*3.0

```
temp1 = inttoreal(3)
temp2 = id3*temp1
temp3 = id2+temp2
id1 = temp3
```

Code Optimization

- Code optimization is an optional phase.
- It is used to improve the intermediate code so that the output of the program could run faster and take less space.
- It removes the unnecessary lines of the code.
- It arranges the sequence of statements in order to speed up the program execution.

Example:

Res = x+y*3

Code Optimization

- Res = x+y*3; (Source code)

id1 = id2+id3*3.0

```
temp1 = id3*3.0
id1 = id2+temp1
```

Optimization Examples

```
1) int a=5, b= 10;
   for(int k=1;k,=100;k++)
   {
        S = a+k;
        T=b*20;
   }
2) x = num1*num2+count;
    y = num1*num2+index;
3) for(int i=1; i<=10; i++)
    for(int j=1;j<=30;j++)
            Arr[i][j] = i+j;
```

Code Generation

- Code generation is the final stage of the compilation process.
- It takes the optimized intermediate code as input and maps it to the target machine language.
- Code generator translates the intermediate code into the machine code of the specified computer.

Example:

Res = x+y*3

Code generation

- Res = x+y*3; (Source code) id1 = id2+id3*3.0

```
LDF R2, id3
MULF R2, R2, #3.0
LDF  R1, id2
ADDF R1, R1, R2
STF id1, R1
```

Symbol Table Management

- A symbol table contains a record for each identifier with fields for the attributes of the identifier.
- This makes easier for the compiler to search the identifier record and retrieve it quickly.
- The symbol table also helps in the scope management.
- The symbol table interact with all the phases and symbol table update correspondingly.
- **Example**:

```
int fun(int a, int b)
{
   int res;
   res = a+b;
   return(res)
}
```

Symbol Table		
Name	**Type**	**Scope**
fun	function	global
a	int	function parameter
b	int	function parameter
res	int	local

Example

```
void fun()
  {
  int x1;
  int y1;
      {
       int x2;
       int y2;
      }
   int x3;
     {
      int x4;
      int y4;
     }
  }
```