University of sulaimani
College of science
Department of Computer Science

Compiler

The Structure of the Compiler

2024-2025

Mzhda Hiwa

Complier

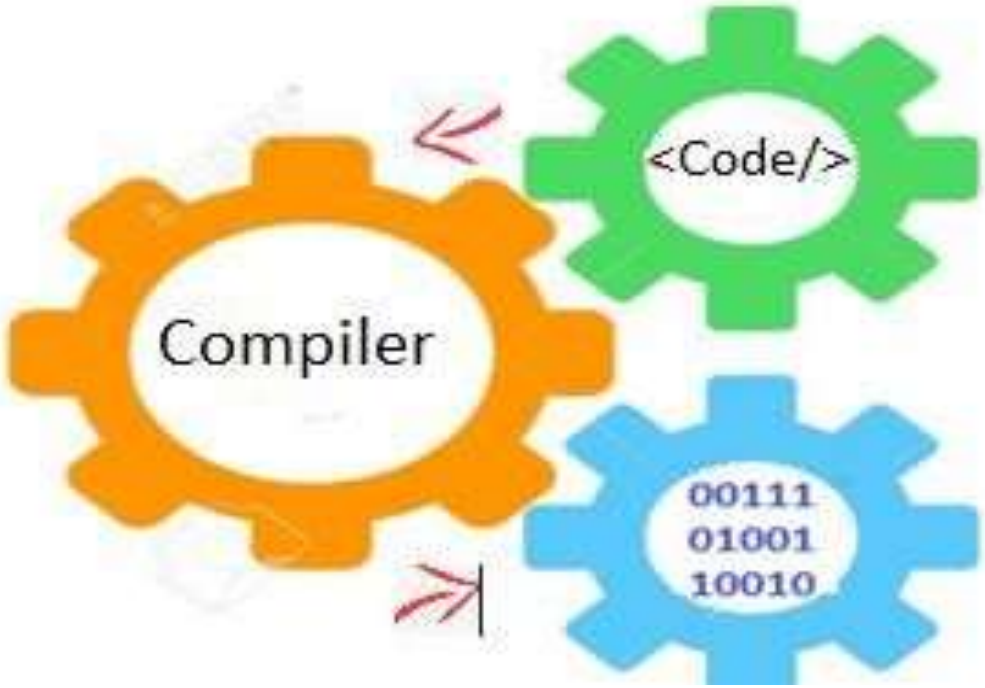
- A compiler is a computer program that translates a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers.

- All the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.

- The software systems that do this translation are called *compilers*.

Compiler..cont'd

Source Code



Executable Code

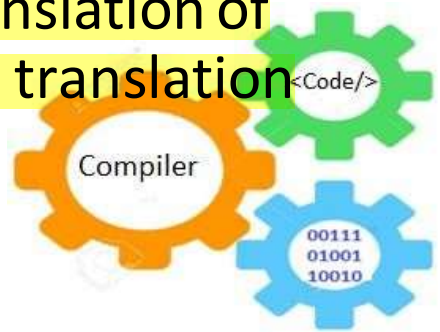
Interpreter and its Differences with Compiler

Interpreters also convert high level language into machine readable binary equivalents.

- There are some differences between compiler and interpreter:

1. The interpreter takes one statement then translates it and executes it and then takes another statement. While the compiler translates the entire program in one go and then executes it .

2. Compiler generates the error report after the translation of the entire page while an interpreter will stop the translation after it gets the first error.

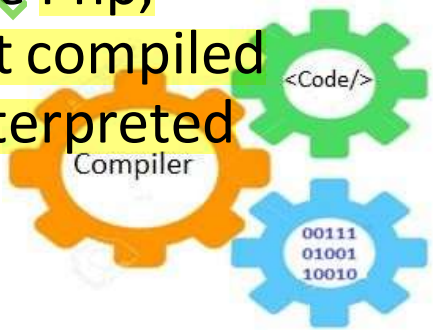


Difference Between Compiler and Interpreter ..cont'd

3. Compiler generates intermediate object code which further requires linking, hence requires more memory. While interpreter no intermediate object code is generated, hence are memory efficient.

4. The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs .

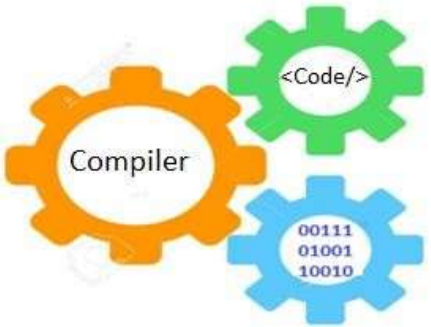
5. Examples of compiled programming languages are C, C++ ,Objective C Examples of interpreted languages are Php, Perl, JavaScript and Matlab, java programs are first compiled to an intermediate form called bytecodes, then interpreted by the interpreter.



Java Language Processors

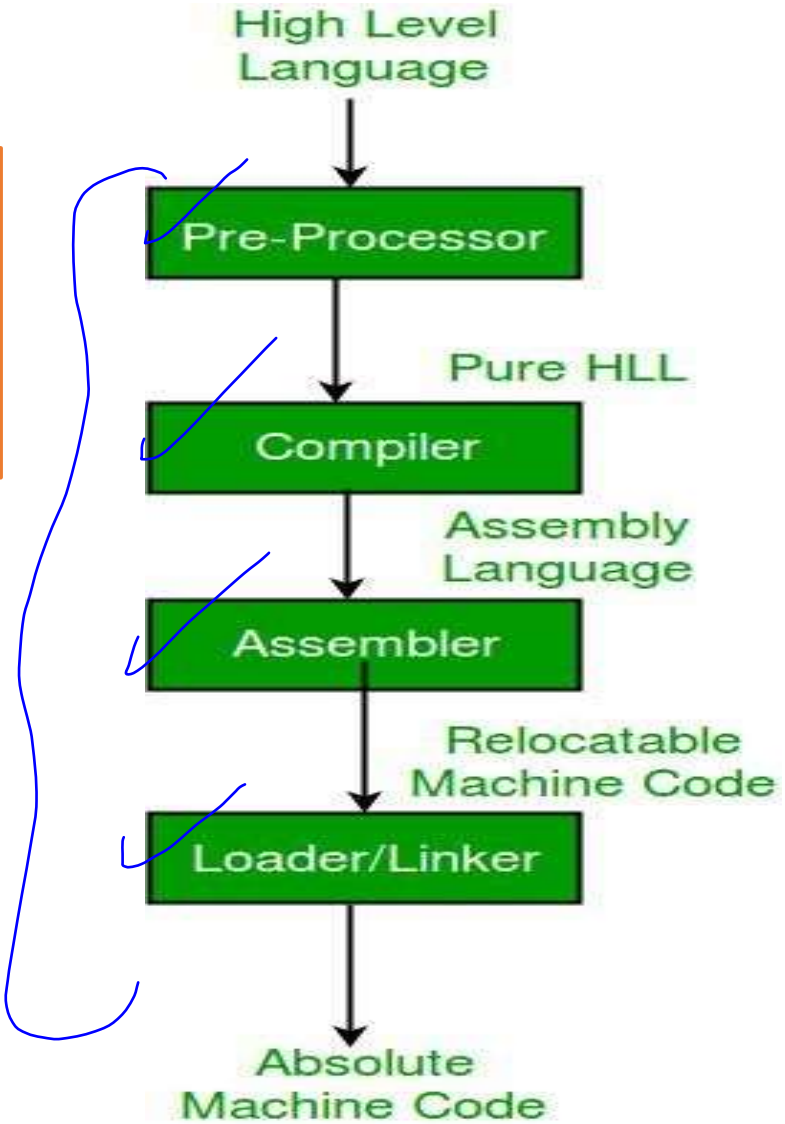
- Java language processors combine compilation and interpretation, A Java source program may first be compiled into an intermediate form called bytecodes. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

- In order to achieve faster processing of inputs to outputs, some Java compilers, called **just-in-time compilers**, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.



Steps for Language processing systems

In addition to a compiler, several other programs may be required to create an executable target program.



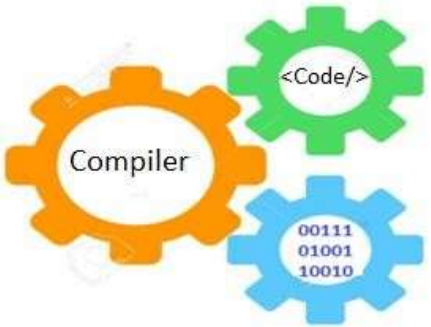
Preprocessor

Pre-processor is a program that processes the code before it passes through the compiler. Preprocessor scans source code and includes the header files which contains relevant information for various functions.

```
#define int x=10;
int main (){
int x= x ;
}
```

After Preprocessing

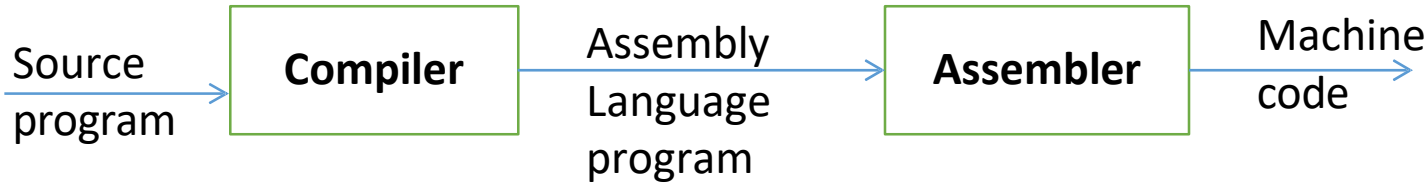
```
int main (){
int x=10;
}
```



Assembler

- Takes the assembly code produced by the compiler as input and produces the machine code as output.

- An assembler is a program that takes basic computer instructions(Assembly Language) and converts them into a pattern of bits that the computer's processor can use to perform its basic operations.

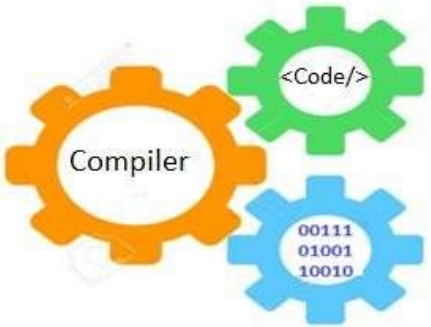


Linker

- Usually a longer program is divided into smaller subprograms called modules. And these modules must be combined to execute the program. The process of combining the modules is done by the **linker**.

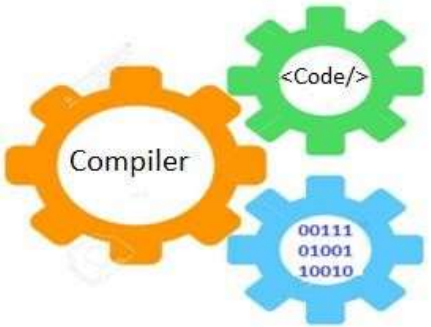
- **Linkers** links the libraries and functions which are essential for executing a program.

- If linker does not find a library of a function then it informs to compiler and then compiler generates an error.

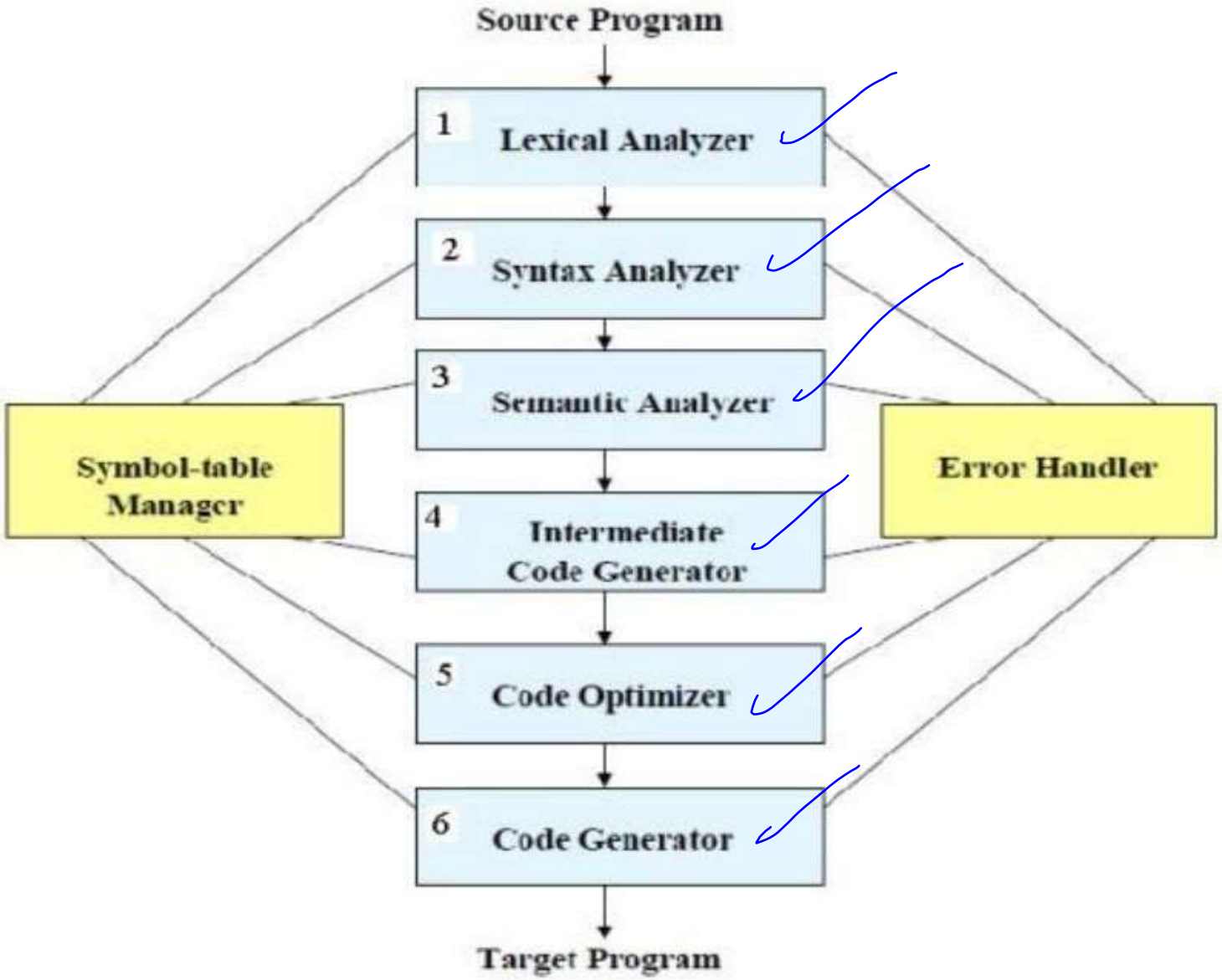


Loader

Loader is a program that loads machine codes of a program into the system memory. After the linker has done its work, the resulting "executable file" can be loaded by the operating system into central memory.



Phases of compiler (LSS ICG)

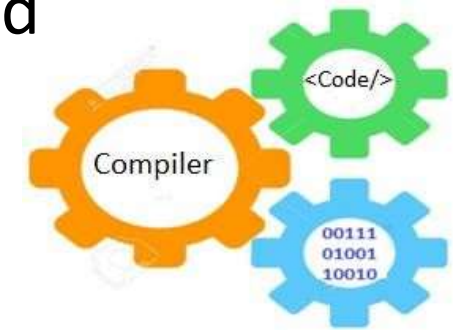


Lexical Analysis

- The lexical analyzer takes a source program as input, and produces a stream of tokens as output.

- A program or function that performs lexical analysis is called a **lexical analyzer**, **lexer**, **tokenizer**, or **scanner**.

- A **token** is a string of one or more characters that is meaningful as a group. The process of forming tokens from an input stream of characters is called **tokenization**.



Lexical Analysis..cont'd

- For example, suppose a source program contains the assignment statement

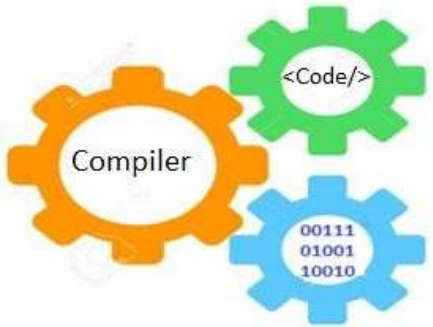
position = initial + rate * 60;

Tokens	Token type
position	Identifier
=	Assignment operator
initial	Identifier
+	Addition operator
rate	identifier
*	Multiplication operator
60	Integer literal ,number
;	Semicolon ,separator

2. Syntax Analysis

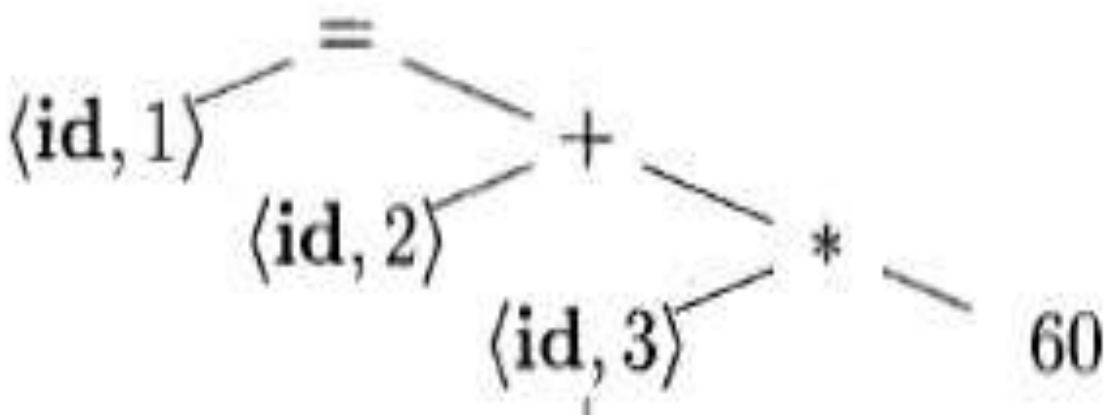
- The second phase of the compiler is syntax analysis or “parsing”.

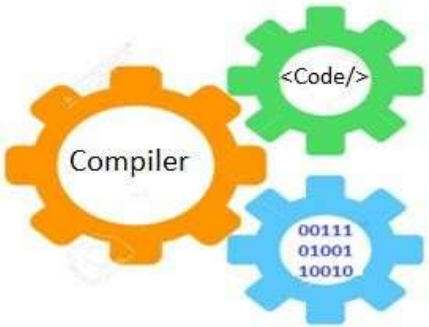
- Syntax analysis is based on the rules based on the specific programing language by constructing the parse tree with the help of tokens. It also determines the structure of source language and grammar or syntax of the language.



Syntax Analysis ..cont'd

- The tree structure produced in this phase is called *parse tree* or *syntax tree*.

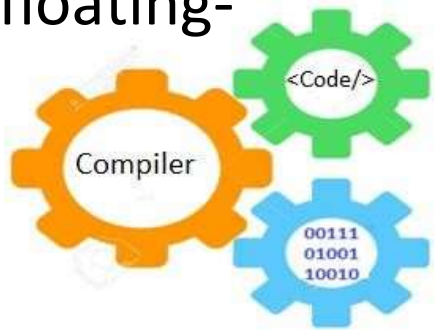




3. Semantic Analysis

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

- An important part of semantic analysis is **type checking,** where the compiler checks that each operator has matching operands. For example, many programing language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.



Semantic Analysis ..cont'd

- The Semantic Analyzer also check matching if…else statements, A variable is used but hasn't been initialized , Every variable is declared before used, etc.

- The language specification may permit some type conversions called **coercions**.

For example position = initial + rate * 60; Suppose that position, initial, and rate have been declared to be floating-point numbers, and that the 60 by itself forms an integer. The type checker in the semantic analyzer discovers that the operator * is applied to a floating-point number rate and an integer 60. In this case, the integer may be converted into a floating-point number.

4. Intermediate Code

During the translation the compiler may construct one or more intermediate code. This intermediate code should have two important properties:

1. It should be easy to produce.
2. It should be easy to translate into the target machine.

- The considered intermediate form called **three-address code**, which consists of a sequence of assembly-like instructions with three **operands per instruction**. Each operand can act like a **register**.

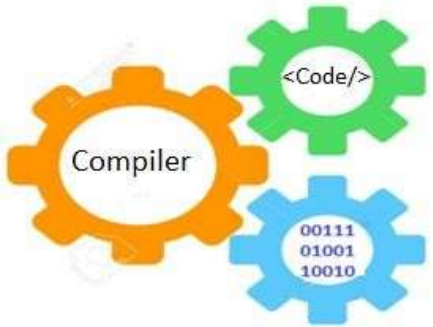
For example; id1= id2+id3*60

t1 = inttofloat(60)

t2 = id3 * t1

t3 = id2 + t2

id1 = t3



Code optimization

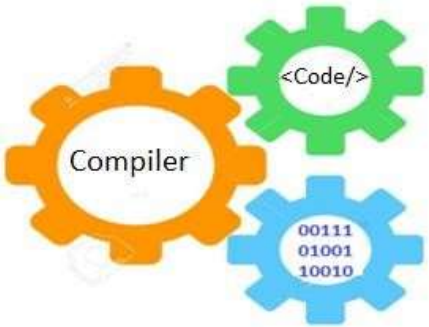
- It improves the Intermediate code in order to get a better target code in result. Usually better means faster, but other objectives may be desired, such as shorter code or target code that consumes less memory. Attempts to improve the intermediate code, so that faster-running machine code will result.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Can be performed by using two instructions:

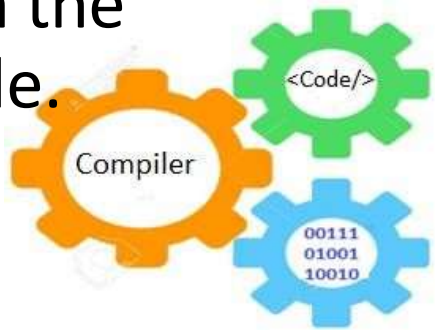
t1=id3*60.0

id1=id2+t1



6 Code Generation

- Final phase of the compiler.
- Generation of target code (in general, machine code or assembly code).
- Intermediate instructions are each translated into a sequence of machine instructions.
- Code generator takes the instructions from the previous phase to generates the target code.



Example of Code Generation

- **Ex:**

LDF R2, id3

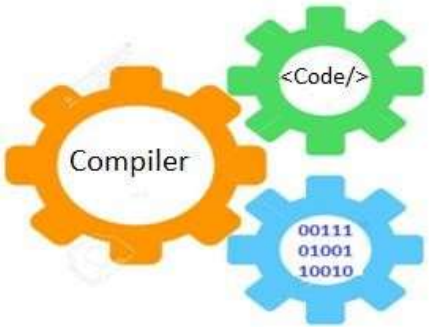
MULF R2, R2, #60.0

LDF Rl, id2

ADDF Rl, Rl, R2

STF idl, Rl

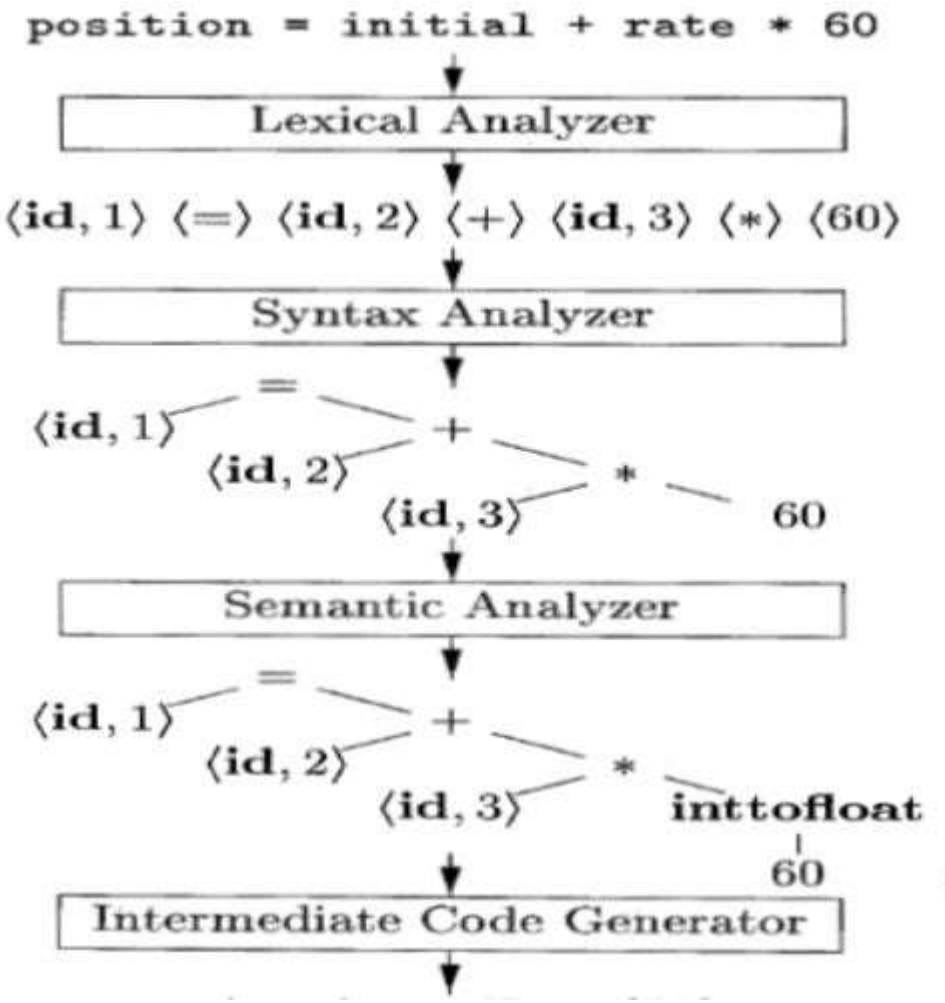
Note: the instructions are different according to the different types of microprocessor



Translation of a Statement

SYMBOL TABLE

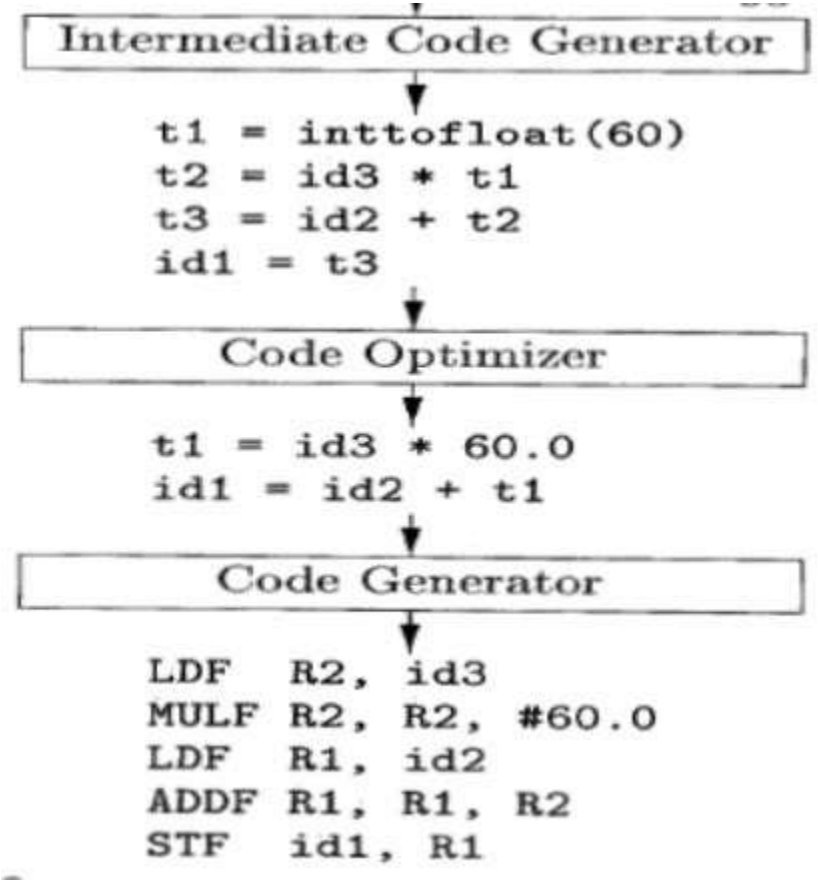
1	position	···
2	initial	···
3	rate	···
4		

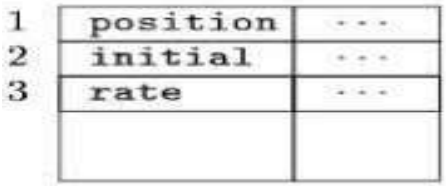


Translation of a Statement..cont'd

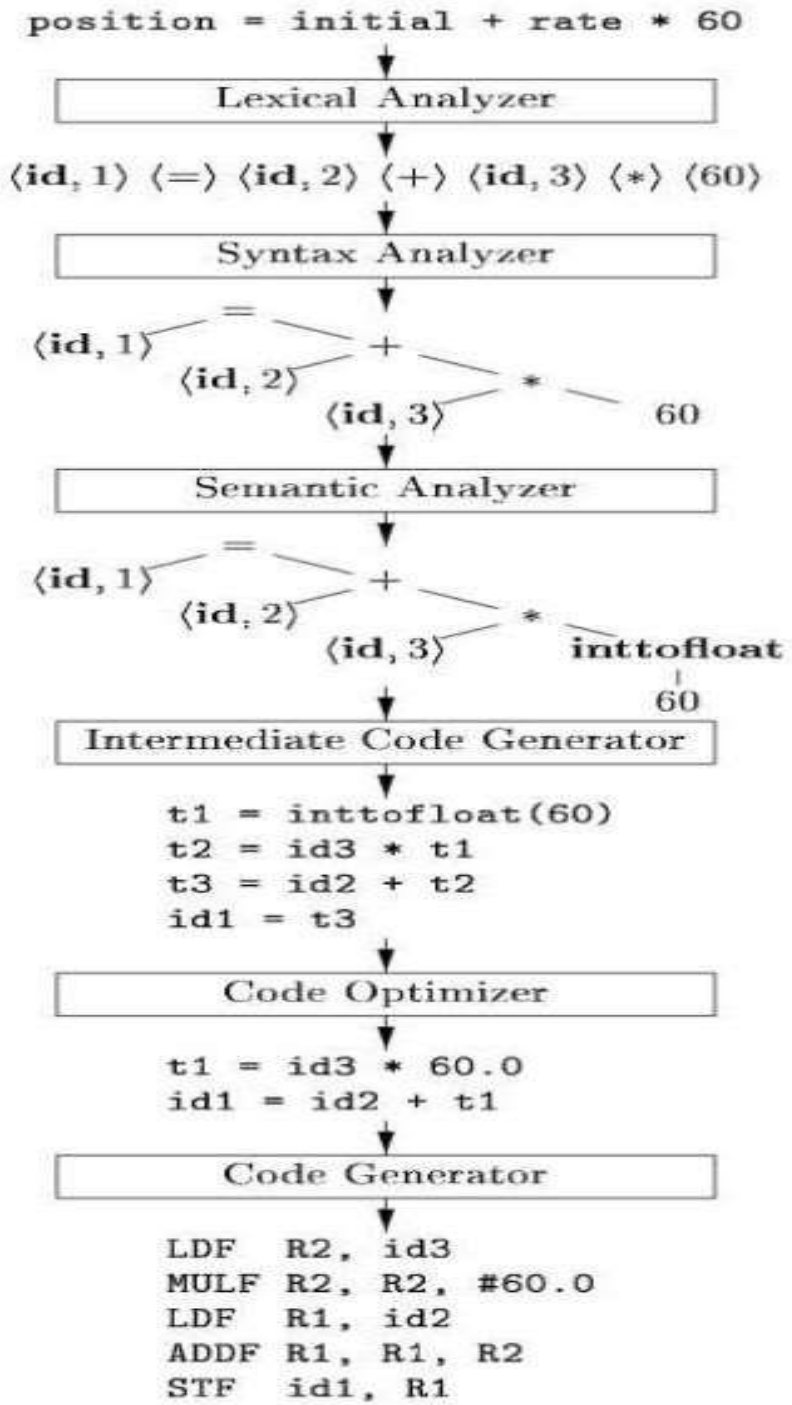
SYMBOL TABLE

1	position	···
2	initial	···
3	rate	···
4		





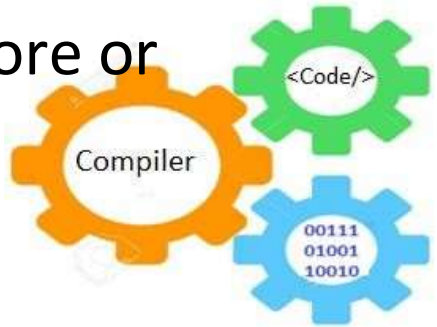
SYMBOL TABLE



Symbol Table

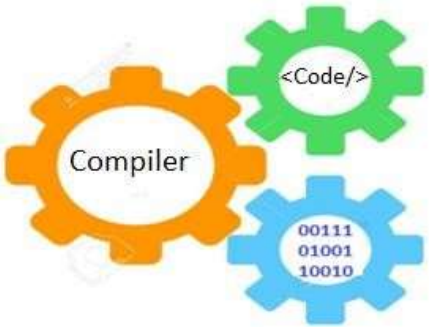
Symbol table is an important data structure created and maintained by **compilers** in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. **Symbol table** is used by both the analysis and the synthesis parts of a **compiler**.

The symbol table is containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.



Error Handler

- The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recover strategy and implement them to handle error. During this whole process processing time of program should not be slow.



Common Programming Errors can occur at many different levels

- **Lexical errors** include misspellings of identifiers, keywords, or operators ,and missing quotes around text intended as a string.
- **Syntactic errors** include misplaced semicolons or extra or missing braces; that is, '((" or ")." As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).
- **Semantic errors** include type mismatches between operators and operands. An example is a return statement in a Java method with result type void.
- **Logical errors** can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the programmer's intent.

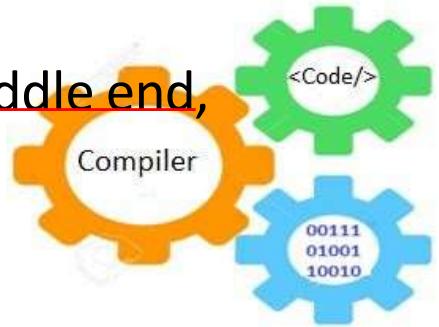
Front End vs Back End of a Compilers

- The phases of a compiler are collected into front end and back end.
- The FRONT END consists of those phases that depend primarily on the source program. These normally include Lexical and Syntactic analysis, Semantic analysis ,and the generation of intermediate code.

- The BACK END includes the code optimization phase and final code generation phase, along with the necessary error handling and symbol table operations.

- The front end Analyses the source program and produces intermediate code while the back end Synthesis the target program from the intermediate code.

- Intermediate representation may be considered as middle end, as it depends upon source code and target machine.



Front End vs Back End of a Compilers