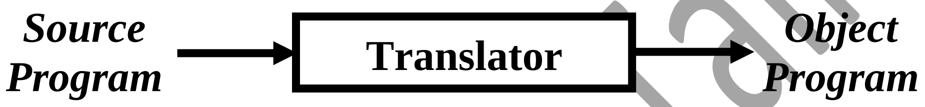
Compiler and Translators

Introduction:

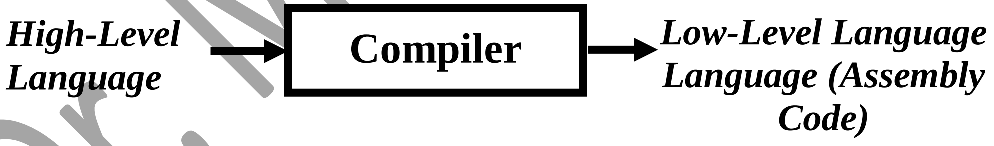
The purpose of these lectures is to get the acquainted with the compiler construction basics and to study the algorithms required to design a compiler.

Translator is a program that takes as input a program written in one programming language (the source language) and produces as output a program in another language (the object or target language).



Compiler is a translator that translates a high-level language program such as FORTRAN, PASCAL, C^{++}, to low-level language program such as an assembly language or machine language.

Note: The translation process should also report the presence of errors in the source program.

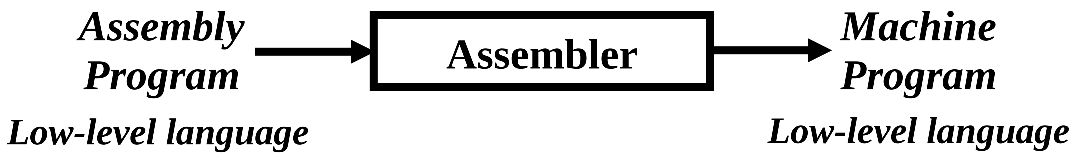


Others Translators:

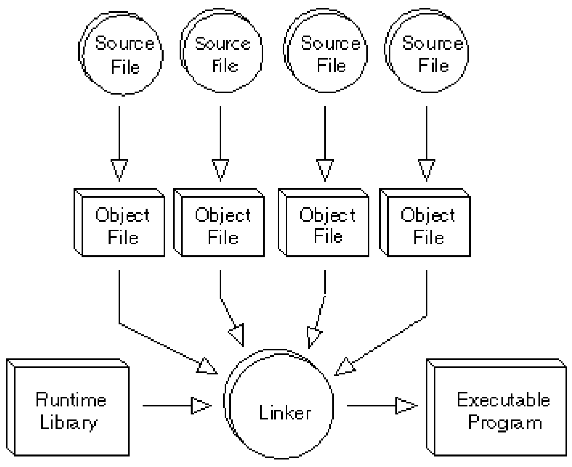
There are another kinds of translators including:

1- An ***Interpreter*** is a translator that effectively accepts a source program and executes it directly, without, producing any object code first. It does this by fetching the source program instructions one by one, analyzing them one by one, and then "executing" them one by one. It is need smaller space (advantage), but it is slower (disadvantage).

2- An ***Assembler*** is a translator that translates the assembly language program (mnemonic program) to machine language program.



3- A ***Linker*** or ***Link editor*** is a computer program that takes one or more object files generated by a compiler and combines them into a single executable program.



4- A ***Loader*** is the part of an operating system that is responsible for loading programs. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution. Loading a program involves reading the contents of the executable file containing the program instructions into memory, and then carrying out other required preparatory tasks to prepare the executable for running.

The ***cousins of compilers*** are assemblers, and loaders and link editors.

Why do we Need Translations?

If there are no translators then we must programming in machine language. With machine language we must be communicate directly with a computer in terms of bits, registers, and very primitive machine operations. Since a machine language program is nothing more than a sequence of 0's and 1's, programming a complex algorithm in such a language is terribly tedious and fraught with mistakes.

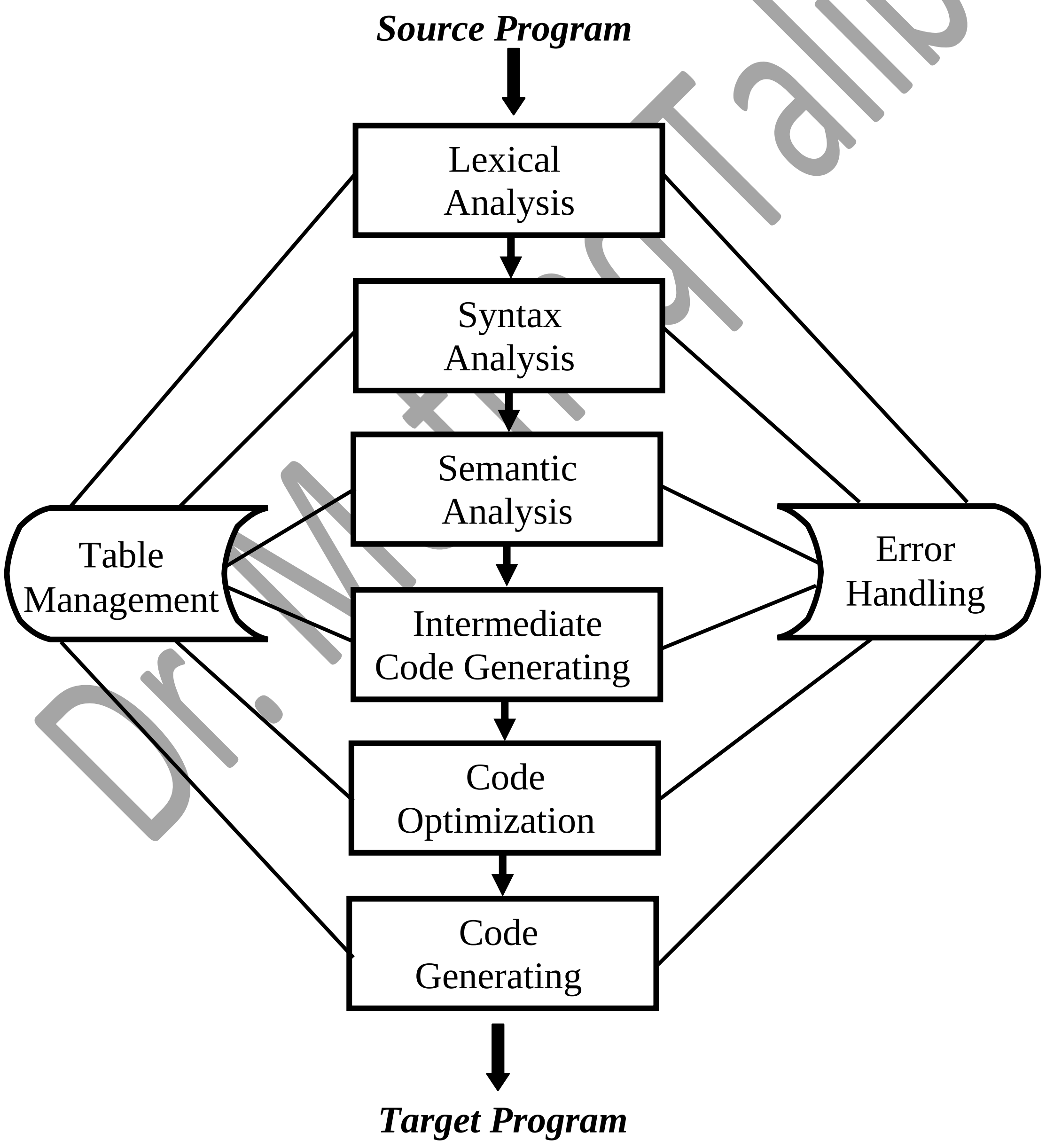
Why we Write Programs in High-Level Language?

We write programs in high-level language (advantages of high-level language) because it is:

1) **Readability**: high-level language will allow programs to be written in the same ways that used in description of the algorithms.
2) **Portability**: High-level languages can be run without changing on a variety of different computers.
3) **Generality**: Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many languages.
4) **Brevity**: Programs expressed in high-level languages are often considerably shorter (in terms of their number of source lines) than their low-level equivalents.
5) It's easy in the **Error checking** process.

The Structure of Compiler

The process of compilation is **so complex** to be achieved in one single step, either from a logical point of view or from an implementation point of view. For this reason it is partition into a series of subprocesses called Phases. The typical compiler consists of several phases each of which passes its output to the next phase plus symbol table manager and an error handler as shown below:



Phases of a Compiler.

Lexical Analyzer (Scanner): Separates characters of the source language program into groups (sets) that logically belong together. These groups are called Tokens. Tokens are Keywords, Identifiers, Operator Symbols, and Punctuation Symbols. The output of the analyzer is stream of tokens, which is passed to the syntax analyzer.

Tokens: groups of characters of source language program logically belong together. For example:

Tokens =

For, If, Do	{Keyword}
x, a1, sum	{Identifiers}
3, 44.2, -53 , "Book"	{Constants}
>, +, =	{Operator Symbols}
; , . ,'	{Punctuation Symbols}

Syntax Analyzer (Parser): Groups tokens together into syntactic structures called ***Expressions***. Expressions might further be combined to form Statements. Often the syntactic structure can be regarded as a ***Tree*** whose leaves are the tokens. The interior nodes of the tree represent strings of tokens that logically belong together.

Semantics: is a term used to describe "meaning", and so the constraint analyzer is often called the static semantic analyzer, or simply the semantic analyzer. The output of the syntax analyzer and semantic analyzer phases is sometimes expressed in the form of a **Abstract Syntax Tree** (AST). This is a very useful representation, as it can be used in clever ways to optimize code generation at a later phase.

Intermediate Code Generator: Create a stream of simple instructions. Many style of intermediate code are possible. One common style uses instructions with one operator and a small number of operands.

Code Optimization: Is an optional phase designed to improve the intermediate code so that the ultimate object program runs faster and/or takes less space.

Code Generation: In a real compiler this phase takes the output from the previous phase and produces the object code, by deciding on the memory locations for data, generating code to access such locations, selecting registers for intermediate calculations and indexing, and so on.

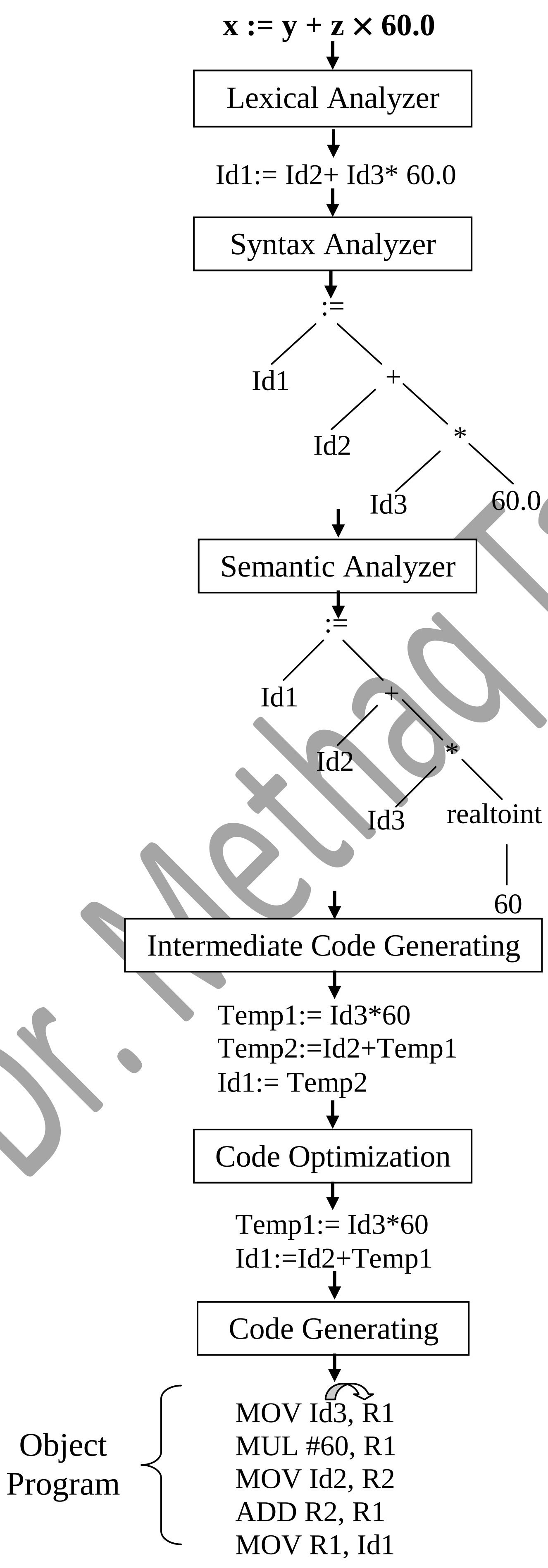
Table Management: Portion of the compiler keeps tracks of the name used by the program and records essential information about each, such as its type integer, real, … etc. the data structure used to record this information is called a ***Symbolic Table***.

Error Handler: One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error messages should allow the programmer to determine exactly where the errors have occurred. Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message.

Example of the Compilation Process

Consider the following sentence is segment from source program:

$$\mathbf{x := y + z \times 60.0}$$



```
Temp1:= Id3*60
Temp2:=Id2+Temp1
Id1:= Temp2
```

```
Temp1:= Id3*60
Id1:=Id2+Temp1
```

```
MOV Id3, R1
MUL #60, R1
MOV Id2, R2
ADD R2, R1
MOV R1, Id1
```