Introduction to Compiler

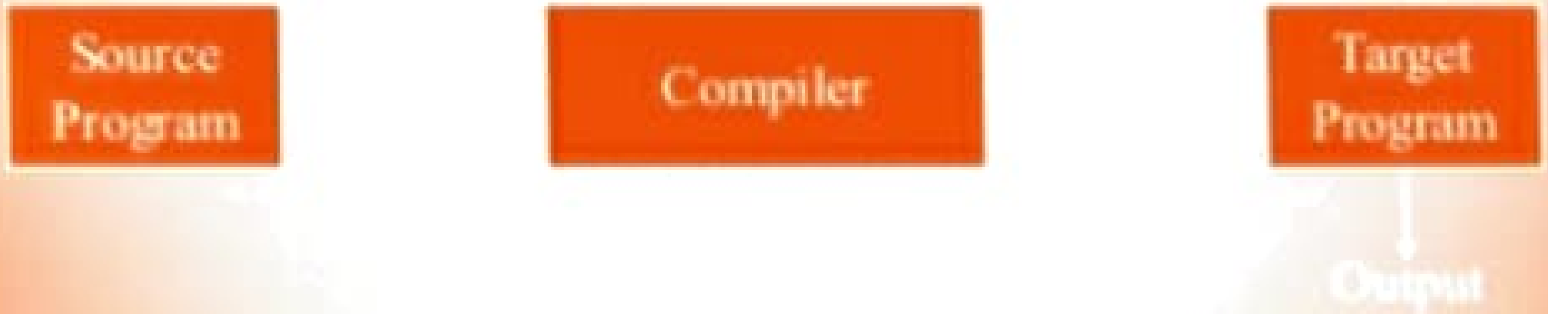
Why Use a compiler?

- All computers only understand machine language

- Therefore, high-level language instructions must be translated into machine language prior to execution

Compiler

- A compiler is a large program that can read a program in one language the *source* language - and translate it into an equivalent program in another language - the *target* language;
- An important role of the compiler is to report any errors in the source program that it detects during the translation process



- If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

Example

```
sum = 0;
for (x = 3; x < 5; x++)
 { cout << "x is " << x;
   cout << endl;
   sum += x;
   a *= b / 2;
```

Source Code

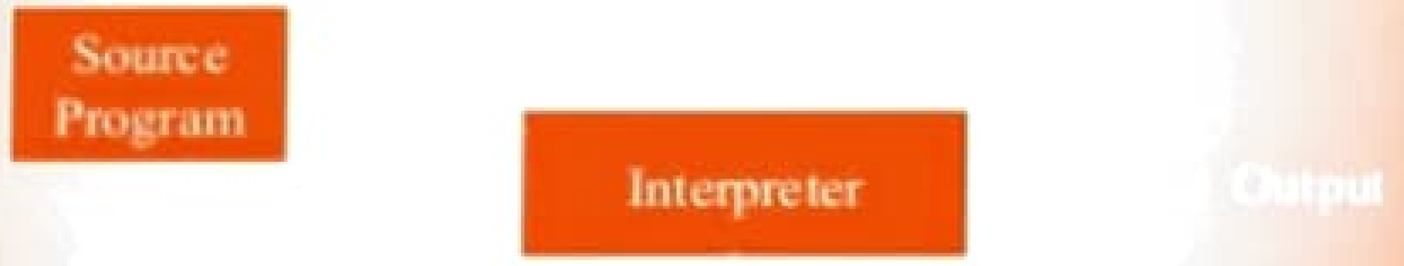
→

```
1 0 0 1 1 1 0 1
0 1 0 1 1 0 1 1
1 0 1 1 1 1 0 0
0 1 1 0 0 1 1 0
1 0 1 0 1 1 0 0
0 0 0 1 1 0 1 1
```

Target Code

Interpreter

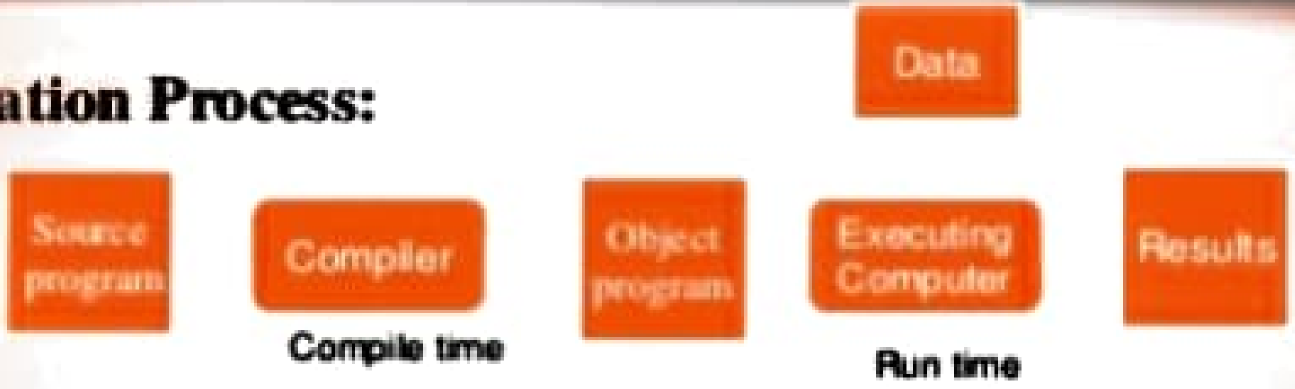
An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program or inputs supplied by the user



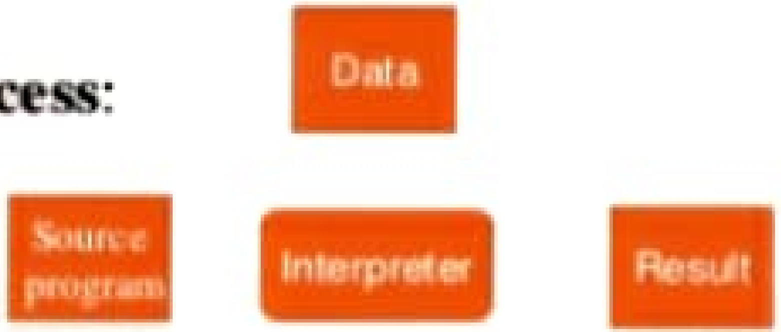
The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Working Process of Compilers Vs Interpreter

Compilation Process:



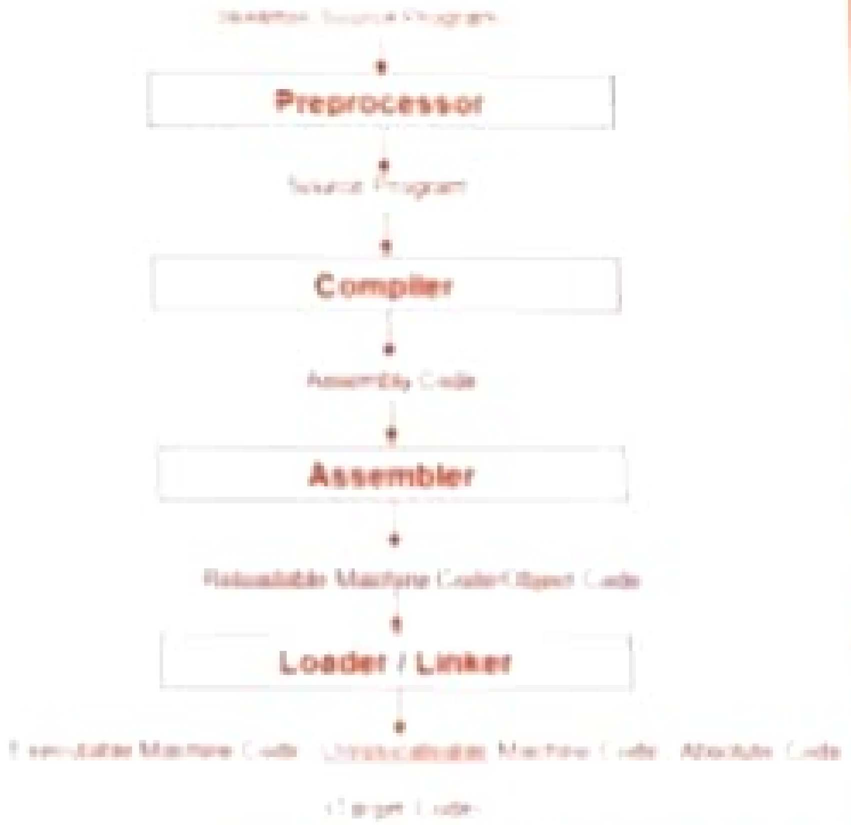
Interpretive Process:



Sr.	Compiler	Interpreter
1	Compiler Takes **Entire** program as input	Interpreter Takes **Single** instruction as input .
2	Intermediate Object Code is **Generated**	**No Intermediate Object Code is Generated**
3	Conditional Control Statements are Executes **faster**	Conditional Control Statements are Executes **slower**
4	**Memory Requirement** : More(Since Object Code is Generated)	**Memory Requirement is Less**
5	Program **need not be compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	Programming languge like C, C++ use compilers.	Programming language like Python, Ruby use interpreters.

Context of a Compiler

- The programs which assist the compiler to convert a skeletal source code into executable form make the context of a compiler and is as follows:
- **Preprocessor:**
  The preprocessor scans the source code and includes the header files which contain relevant information for various functions.
- **Compiler:**
  The compiler passes the source code through various phases and generates the target assembly code.



Cont....

- **Assembler:**

  The assembler converts the assembly code into relocatable machine code or object code. Although this code is in 0 and 1 form, but it cannot be executed because this code has not been assigned the actual memory addresses.

- **Loader/Link Editor:**

  It performs two functions. The process of loading consists of taking machine code, altering the relocatable addresses and placing the altered instructions and data in memory at proper location.

  The link editor makes a single program from several files of relocatable machine code. These files are library files which the program needs.

  The loader/link editor produces the executable or absolute machine code.

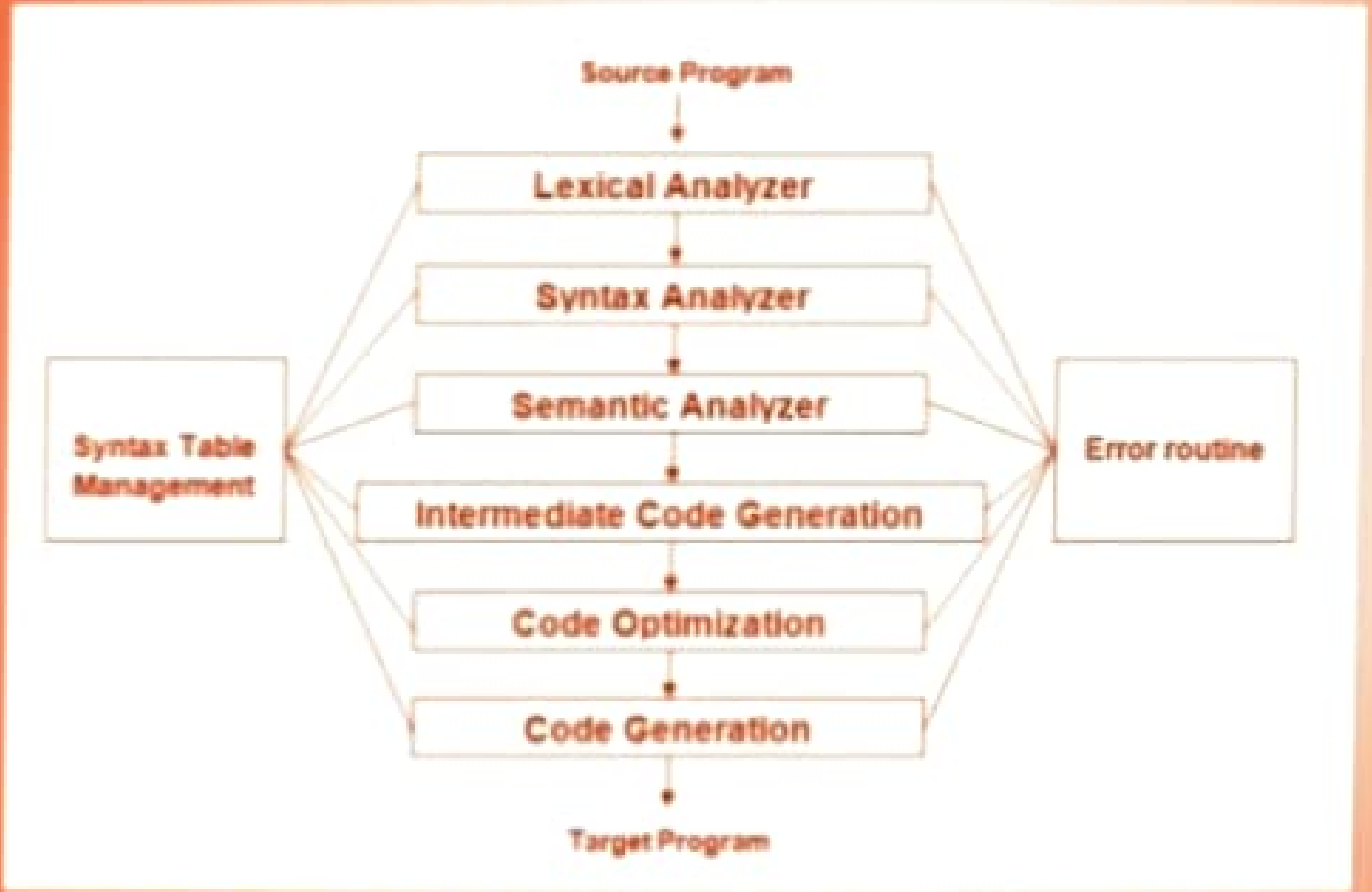
Phases of Compiler Design

A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below

There are two phases of compilation.

- Analysis (Machine Independent/Language Dependent)
- Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called '**phases**'.


Source Program
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generation
Code Optimization
Code Generation
Target Program
Syntax Table Management
Error routine

Phase-1: Lexical Analysis

- Lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called ***lexeme***
- For each lexeme, the lexical analyzer produces a **token** of the form that it passes on to the subsequent phase, syntax analysis

(token-name, attribute-value)

- Token-name: an abstract symbol is used during syntax analysis.
- attribute-value: points to an entry in the symbol table for this token.

Example:

$\text{newval := oldval + 12} \longrightarrow$ **Tokens:**

newval	Identifier
=	Assignment operator
oldval	Identifier
+	Add operator
12	Number

Lexical analyzer truncates white spaces and also removes errors.

LEXICAL ANALYSIS:

Source:

```
program Ex   (output); { comment }
begin writeln ('hi')   end.
```

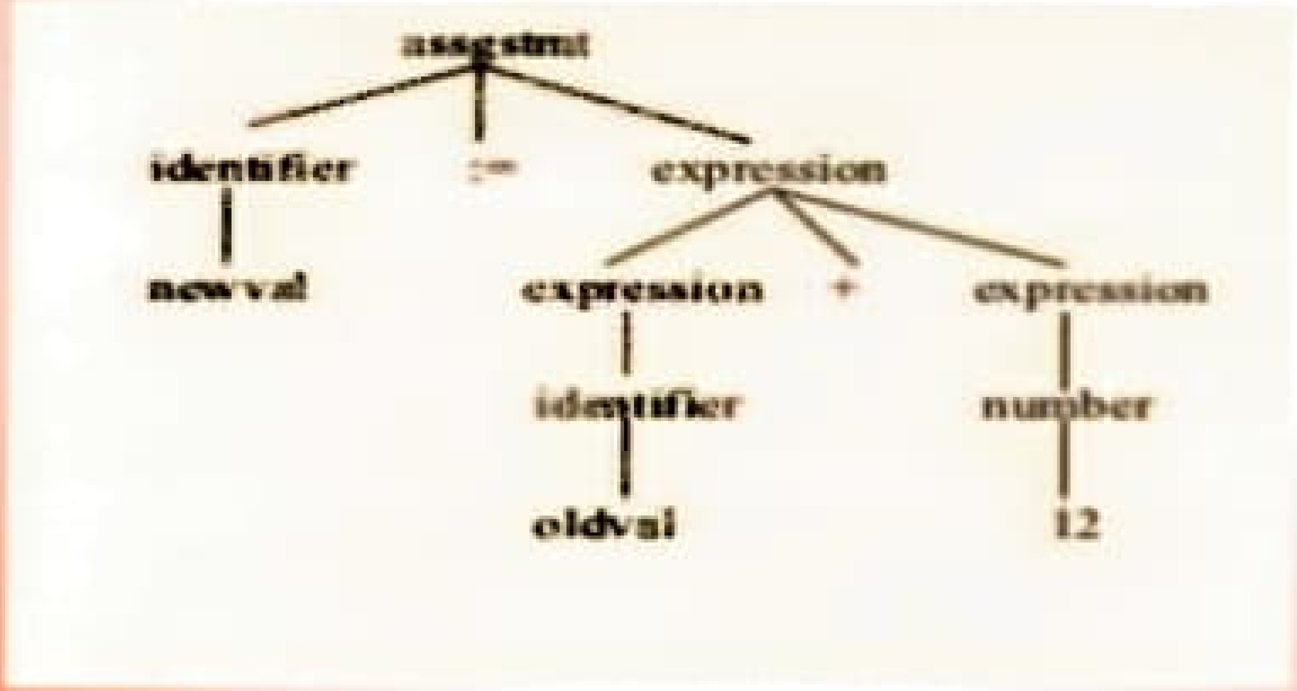
Output of scanner:

```
 1.   Identifier     program
 2.   White_space    ' '
 3.   IdentIfer      Ex
 4.   White_space    '   '
 5.   Punctuation    (
 6.   Identifier     output
 7    Punctuation    )
 8.   Punctuation    ;
 9.   White_space    ' '
10.   Comment        { comment }
11.   Identifier     begin
12.   White_space    ' '
13.   IdentIfier     writeln
14.   White_space    ' '
15.   Punctuation    (
16.   String         'hi'
17.   Punctuation    )
18.   White_space    '   '
19.   IdentIfier     end
20.   Punctuation    .
```

Phase-2: Syntax Analysis

- Also called **Parsing or Tokenizing**.
- The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation

Example:



assgstmt
identifier
:=
expression
newval
expression
+
expression
identifier
number
oldval
12

Phase-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.
- 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 programming 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.
- Example: **newval := oldval+12**

  The type of the identifier newval must match with the type of expression (oldval+12).

Example:

- **<u>Semantic analysis</u>**
  - **Syntactically correct, but semantically incorrect**

example:

sum = a + b;

```
int a;
double sum;
char b;
```

data type **mis**match

Semantic records	
a	integer
sum	double
b	char

Phase-4: Intermediate Code Generation

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation (a program for an abstract machine). This intermediate representation should have two important properties:

- it should be easy to produce and
- 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**.

This phase bridges the analysis and synthesis phases of translation.

Example:

$$\begin{tikzcd}[row sep=small]
\text{newval := oldval + fact * 1} \arrow[d] \\
\text{Id1 := Id2 + Id3 * 1} \arrow[d] \\
{}
\end{tikzcd}$$

```
Temp1     =     into real (1)
Temp2     =     Id3    *     Temp1
Temp3     =     Id2    +     Temp2
Id1       =     Temp3
```

Phase-5: Code Optimization

- The compiler looks at large segments of the program to decide how to improve performance
- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- Usually better means:
  - faster, shorter code, or target code that consumes less power.
- There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.
- Optimization cannot make an inefficient algorithm efficient - "only makes an efficient algorithm more efficient"

Example:

- The above intermediate code will be optimized as:

```
Temp1     =     Id3   *     1
Id1       =     Id2   +     Temp1
```

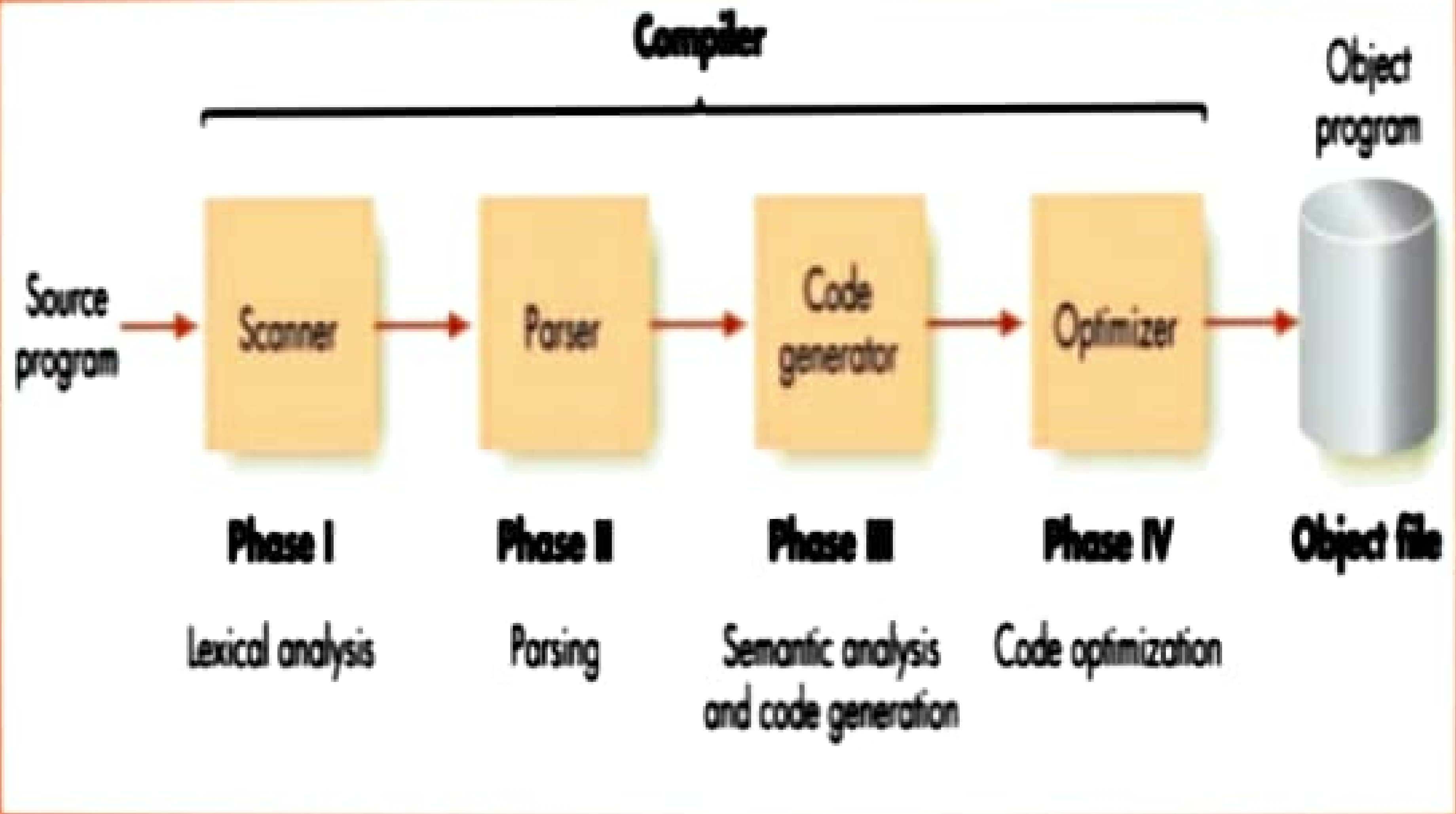
Phase-6: Code Generation

- The last phase of translation is code generation.
- Takes as input an intermediate representation of the source program and maps it into the target language
- If the target language is machine, code, registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.
- A crucial aspect of code generation is the judicious assignment of registers to hold variables.

Example:

Id1 := Id2 + Id3 * 1

```
MOV          R1,Id3
MUL          R1,#1
MOV          R2,Id2
ADD          R1,R2
MOV          Id1,R1
```


Compiler
Source program
Scanner
Parser
Code generator
Optimizer
Object program
Phase I
Phase II
Phase III
Phase IV
Object file
Lexical analysis
Parsing
Semantic analysis and code generation
Code optimization

Symbol-Table Management

- The symbol table is a data structure 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
- These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

new Val	id1 & attribute
old Val	id2 & attribute
fact	id3 &attribute

Error Handling Routine:

- One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler.
- Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message. Both of the table-management and error-Handling routines interact with all phases of the compiler.

One pass compiler

- One pass compiler passes through the source code of each compilation unit only once.
- Their efficiency is limited because they don't produce intermediate codes which can be refined easily.
- One pass compilers very common because of their simplicity.
- Check for semantic errors and generate code.
- They are faster then multi pass compilers.
- Also known as **Narrow compiler.**
- Pascal and C are both languages that allow one pass compilation.

Multi-pass compilers

- The input is passed through certain phases in one pass. Then the output of previous phases is passed through other phases in second pass and so on until the desired output is generated.
- It requires less memory because each pass takes output of previous phase as input.
- It may create one or more intermediate code.
- Also known as **wide compiler.**
- Modula-2 is a language whose structure requires that a compiler has at least two passes.

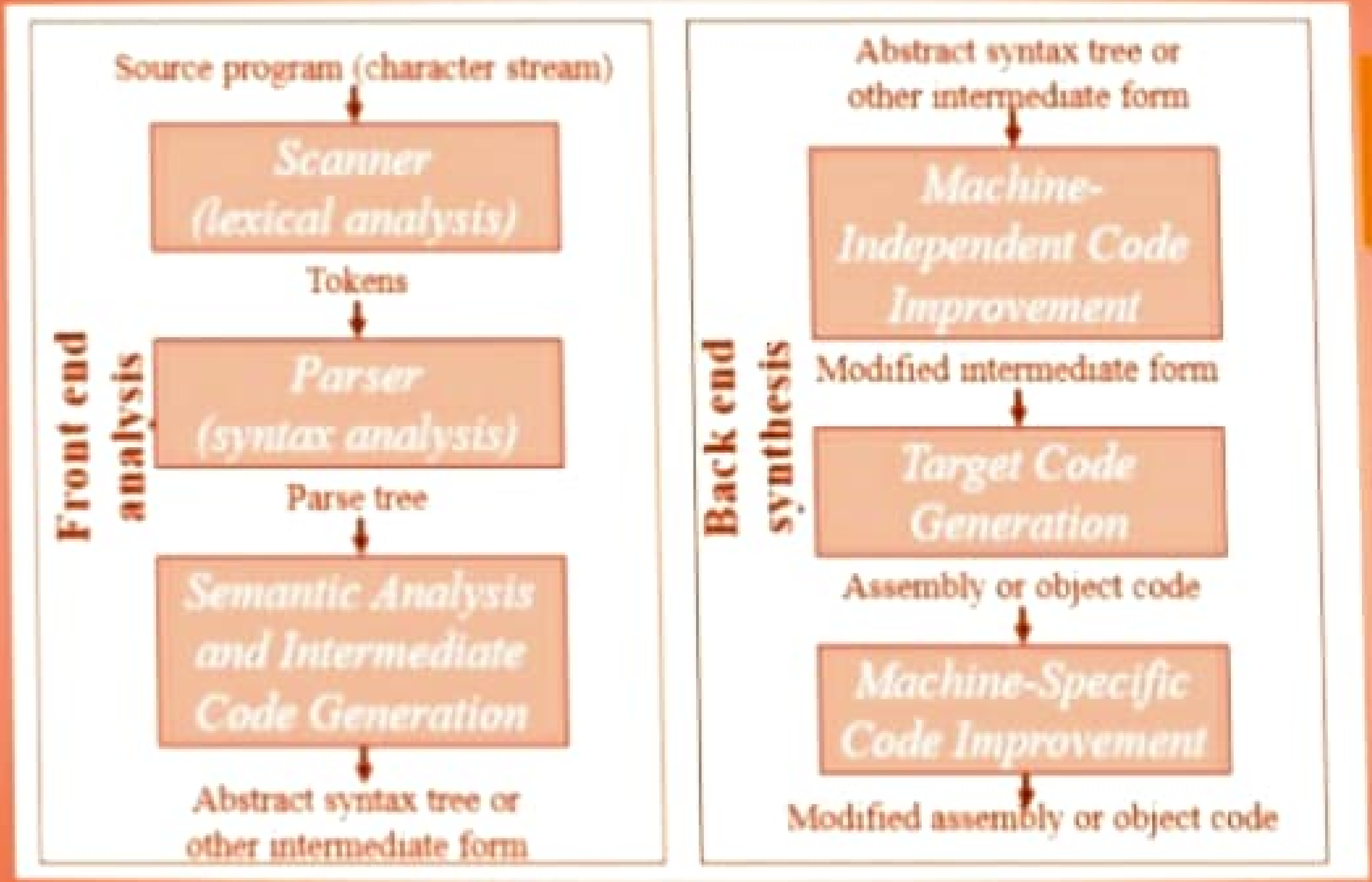
Front End vs Back End of a Compilers

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.
- A certain amount of code optimization can be done by front end as well.
- 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 **Analyzes** the source program and produces intermediate code while the back end **Synthesizes** the target program from the intermediate code.

Cont....

- The front end phase consists of those phases that primarily depend on source program and are independent of the target machine.
- Back end phase of compiler consists of those phases which depend on target machine and are independent of the source program.
- Intermediate representation may be considered as middle end, as it depends upon source code and target machine.


Front end analysis
Source program (character stream)
Scanner (lexical analysis)
Tokens
Parser (syntax analysis)
Parse tree
Semantic Analysis and Intermediate Code Generation
Abstract syntax tree or other intermediate form
Back end synthesis
Abstract syntax tree or other intermediate form
Machine-Independent Code Improvement
Modified intermediate form
Target Code Generation
Assembly or object code
Machine-Specific Code Improvement
Modified assembly or object code