Introduction to Compiler Phases

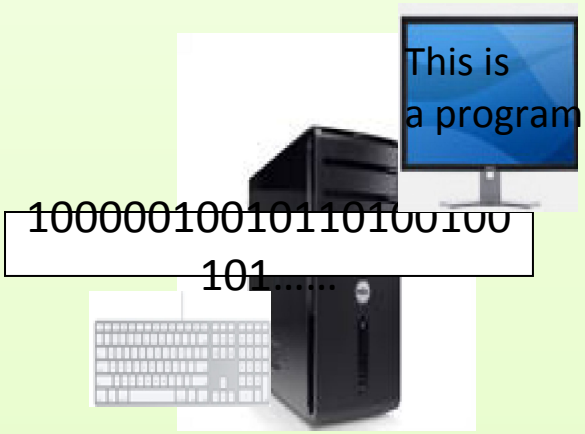
Translator

- Programs written in high-level languages need to be translated into low-level (machine code) for processing and execution by the CPU. This is done by a translator program.

- There are two types of translator program:
  - Compilers
  - Interpreters

Why Use a compiler?

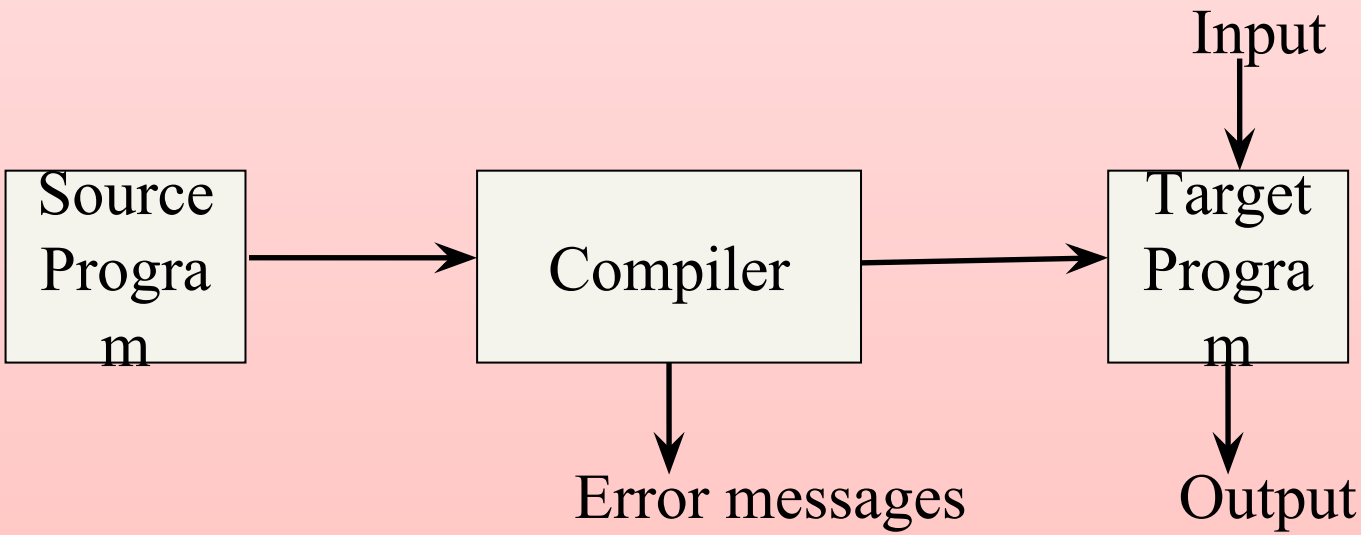
- Compiler is a program which takes one language as input and translate it into an equivalent another language.



- Compiler is divided into two parts: **Analysis** and **Synthesis**

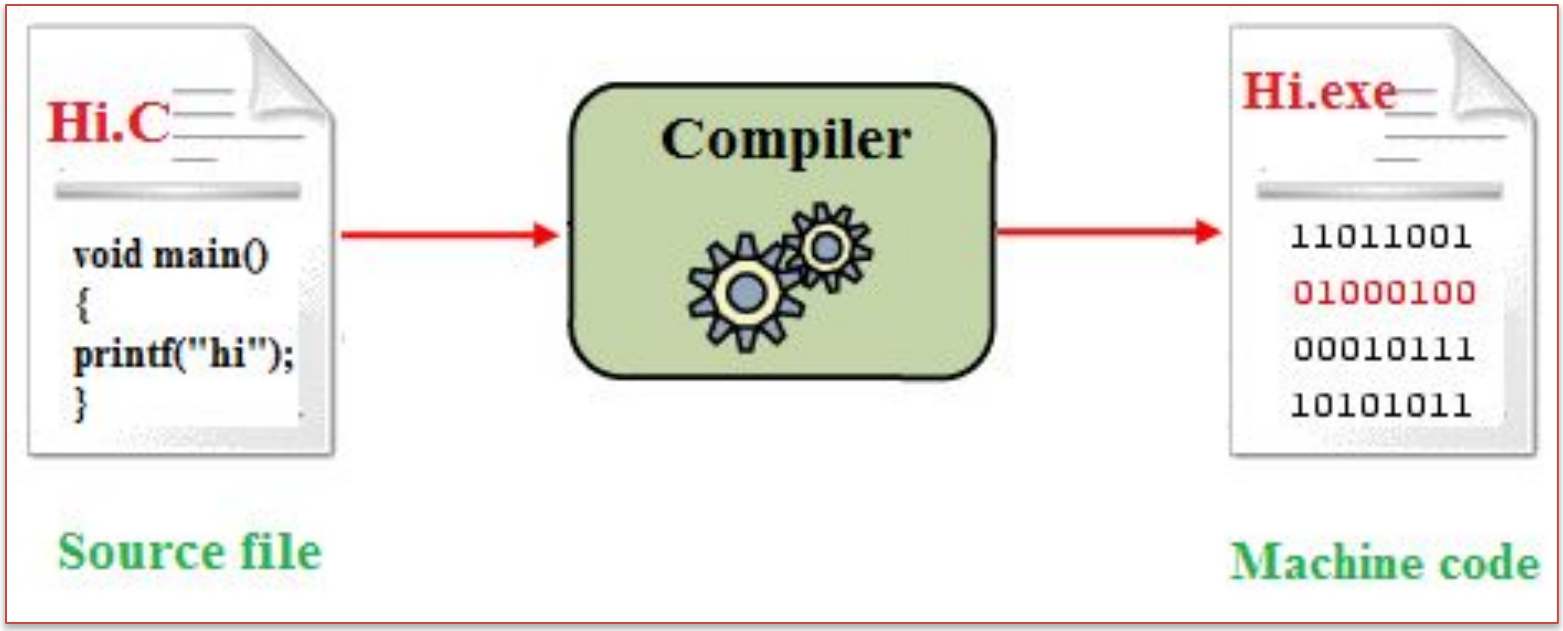
Compiler

- 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



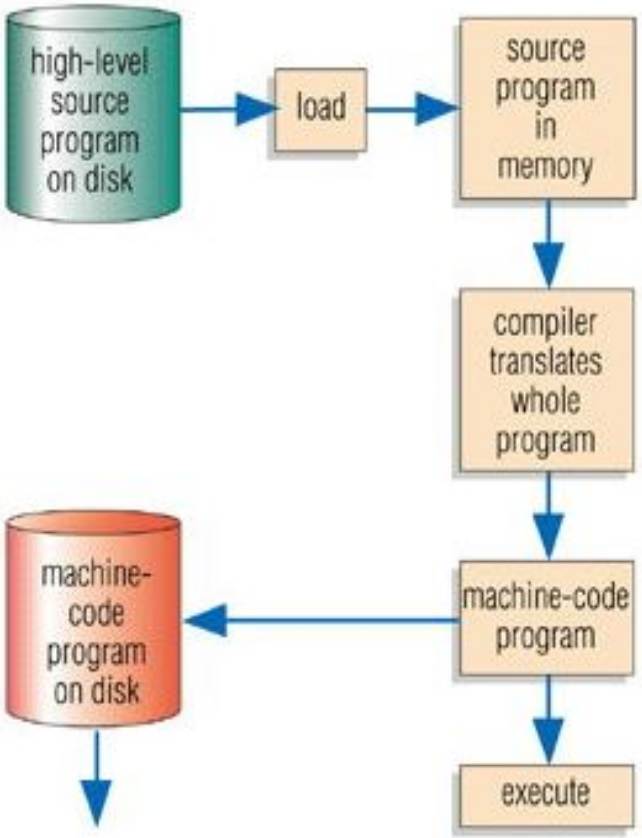
- If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

Example


Hi.C
void main()
{
printf("hi");
}
Compiler
Hi.exe
11011001
01000100
00010111
10101011
Source file
Machine code

Compilers

A Compiler program translates the whole program into a machine code version that can be run without the compiler being present.



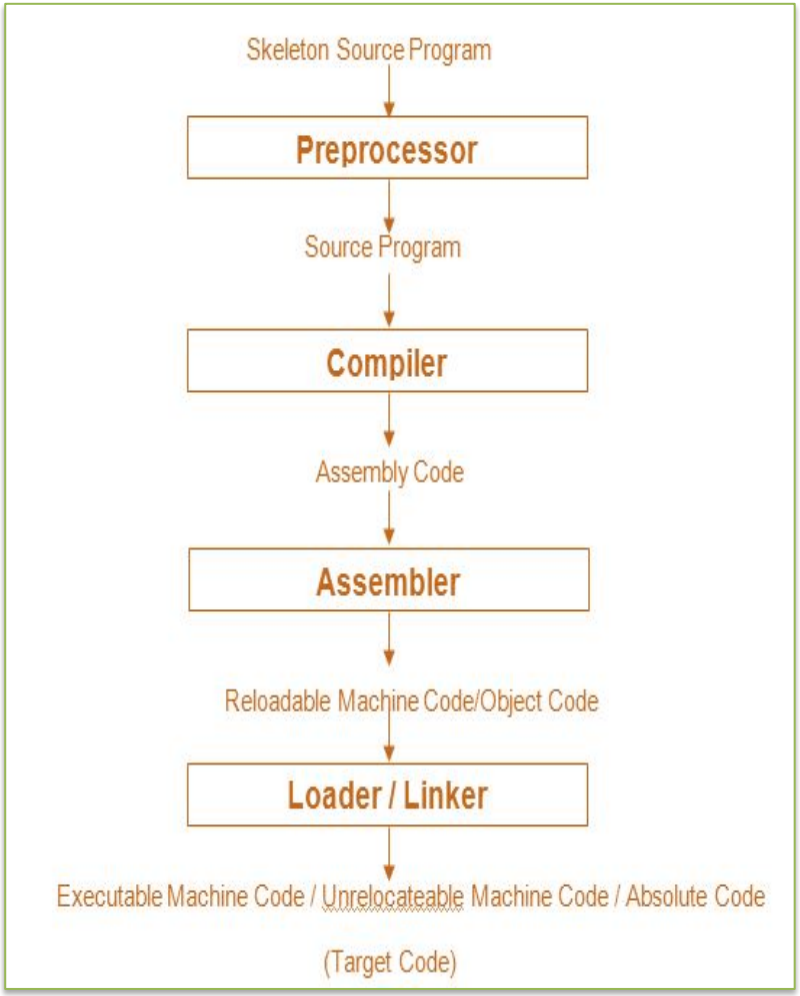
Advantage: program runs fast as already in machine code, translator program only needed at the time of compiling

Disadvantage: slow to compile as whole program translated

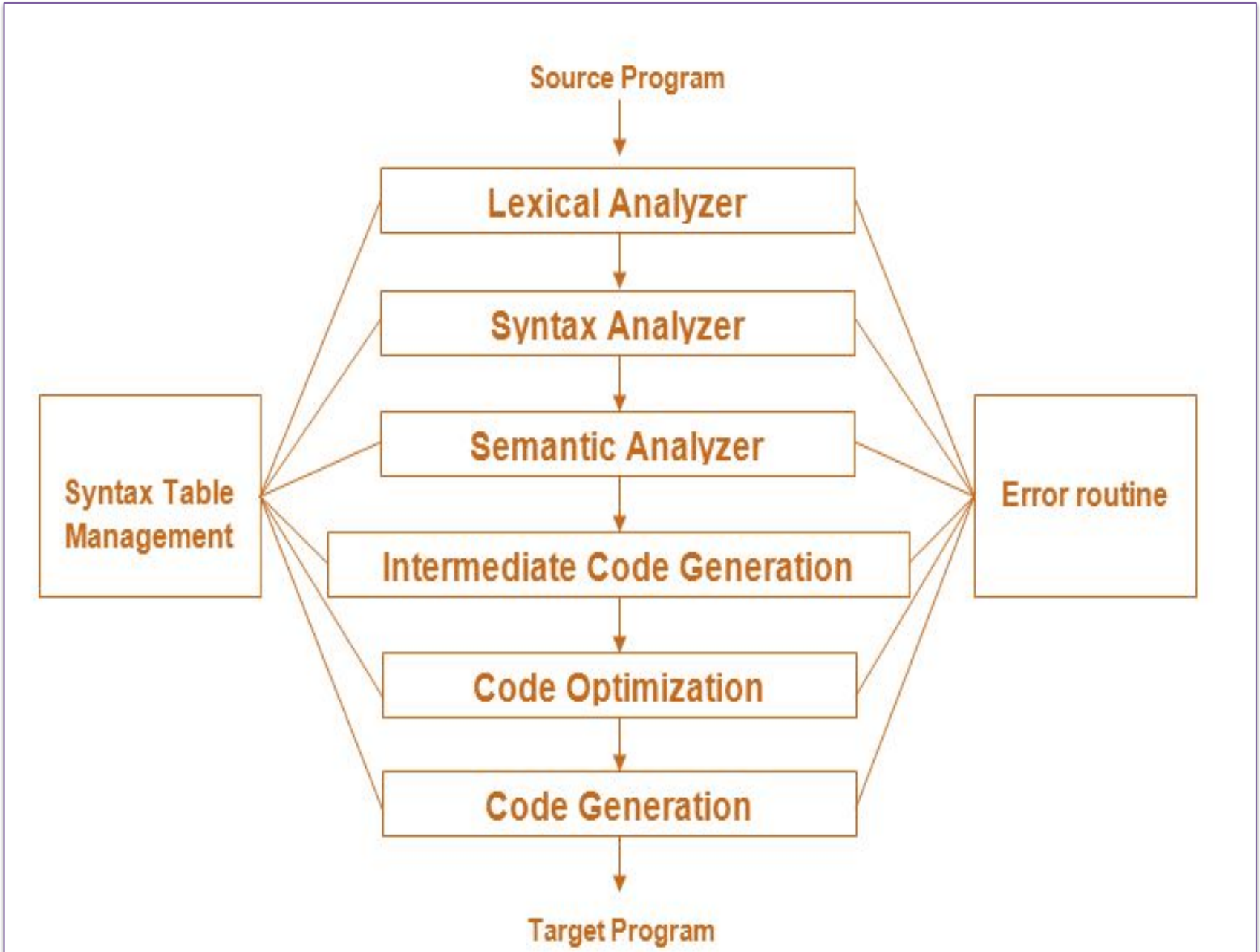
Context of a Compiler

- 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:**

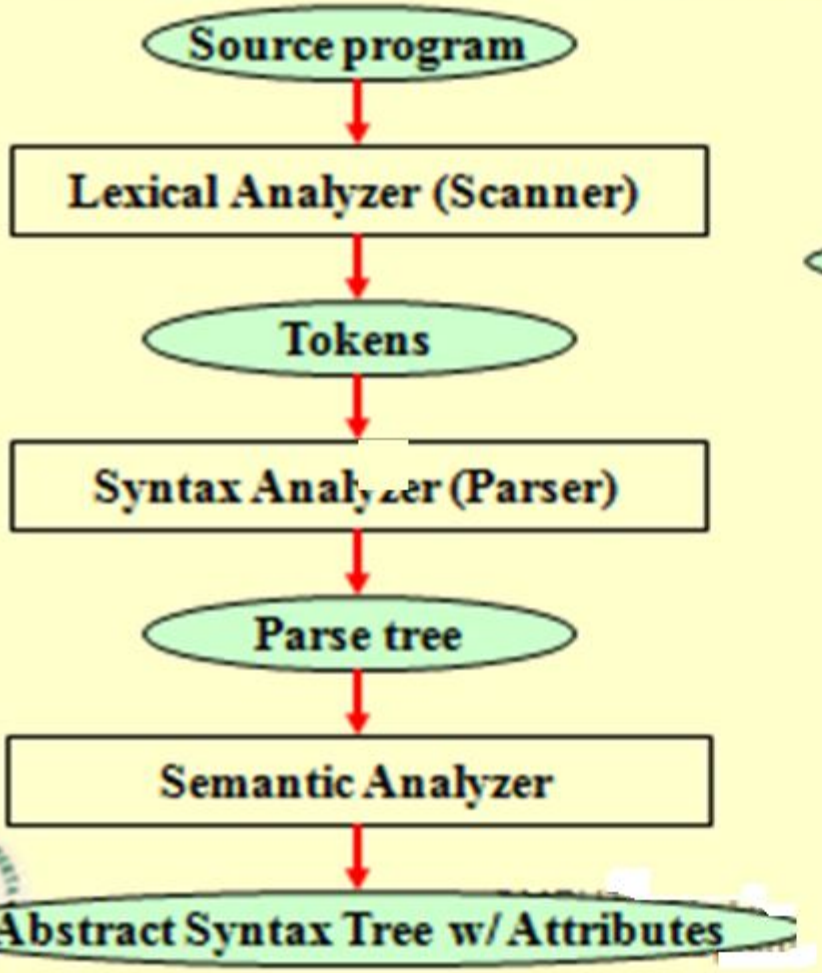
  The preprocessor scans the source code and

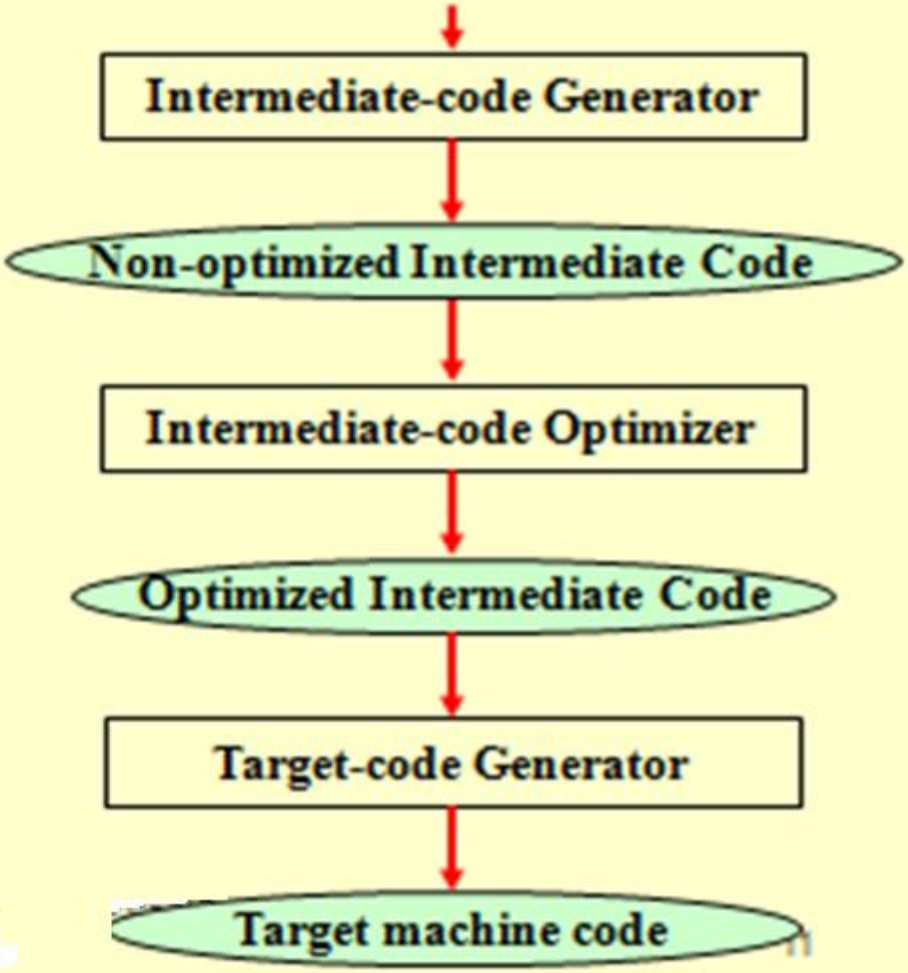


Phases Of Compiler


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

Phases of a Compiler





Phases of Compilers

Analysis Phase:

Breaks the source program into constituent pieces and creates intermediate representation. The analysis part can be divided along the following phases:

- The specification consists of three components:
  1. ***Lexical rules*** which govern the formation of valid lexical units in the source language.
  2. ***Syntax rules*** which govern the formation of valid statements in the source language.
  3. ***Semantic rules*** which associate meaning with valid statements of the language.

Phases of Compilers

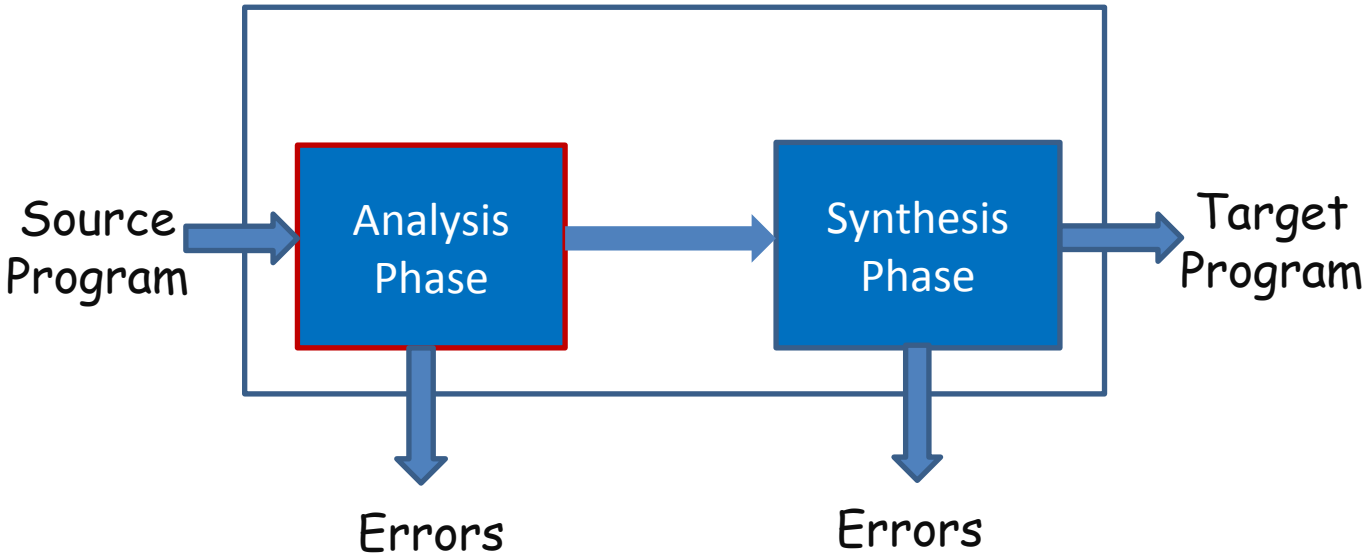
- Consider the following example:

percent_profit = (profit * 100) / cost_price

- **Lexical units** identifies =, * and / operators, 100 as constant, and the remaining strings as identifiers.
- **Syntax analysis** identifies the statement as an assignment statement with percent_profit as the left hand side and (profit *

Phases of Compilers

Language Processor



- Analysis of source statements can not be immediately followed by synthesis of equivalent target statements due to following reasons:
  1. Forward References
  2. Issues concerning memory requirements and organization of a LP

Phases of Compilers

Synthesis Phase

- The synthesis phase is concerned with the construction of target language statements which have the same meaning as a source statement.

- It performs two main activities:
  1. Creation of data structures in the target program (**memory allocation**)
  2. Generation of target code (**code generation**)

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

Phase-1: Lexical Analysis

Definition: *Lexical analysis* is the operation of dividing the input program into a sequence of *lexemes* (*tokens*).

Distinguish between

- *lexemes* – smallest logical units (words) of a program.
  Examples – `i`, `sum`, `for`, `10`, `++`, `"%d\n"`, `<=`.
- *tokens* – sets of similar lexemes.
  Examples –
  identifier = {`i`, `sum`, `buffer`, ...}
  int_constant = {`1`, `10`, ...}
  addop = {`+`, `-`}

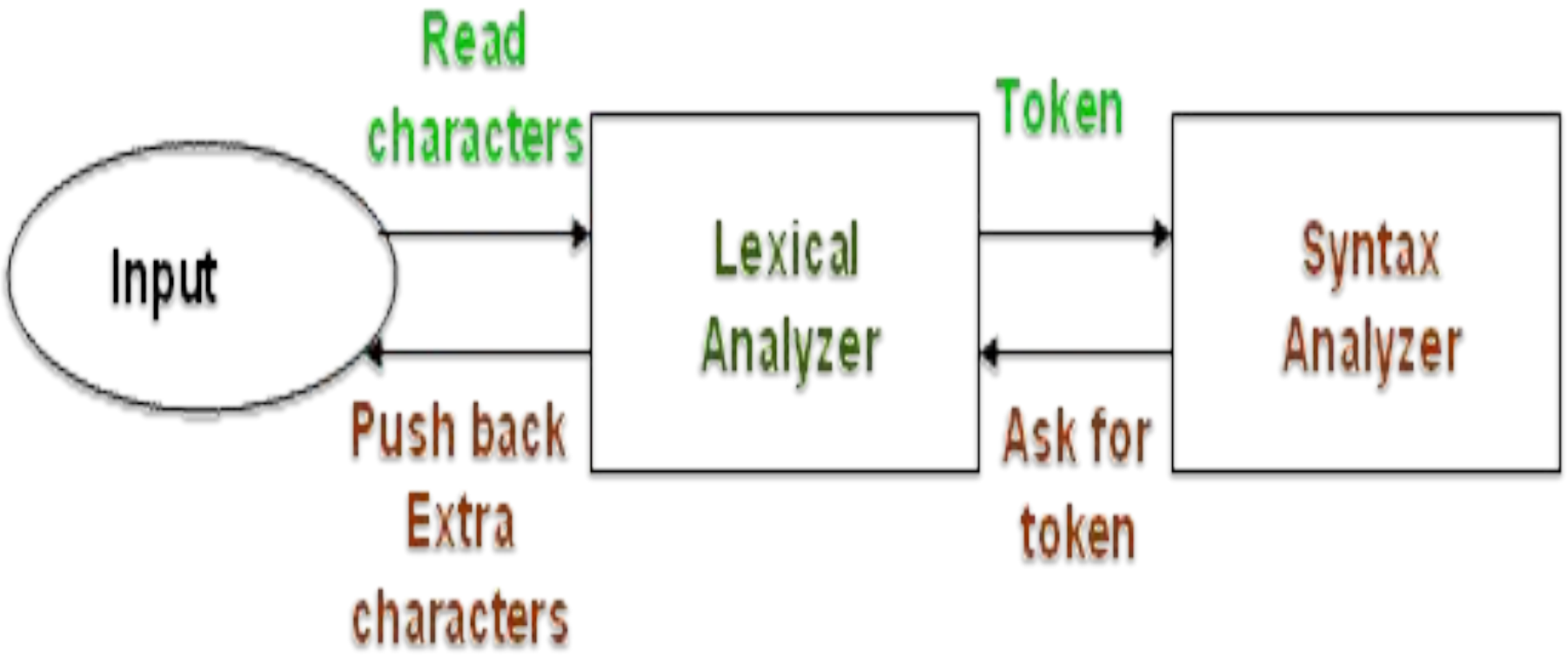
Phase-1: Lexical Analysis

Things that are not counted as lexemes –

- white spaces – tab, blanks and newlines
- comments

These too have to be detected and ignored.

Role of the Lexical Analyzer



Tasks of Lexical Analyzer

- Reads source text and detects the token
- Stripe out comments, white spaces, tab, newline characters.
- Correlates error messages from compilers to source program

Approaches to implementation

. Use assembly language- Most efficient but most difficult to implement

Example:

newval := oldval + 12

Tokens:

- newval Identifier
- = Assignment operator
- oldval Identifier
- + Add operator
- 12 Number (Constant)

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

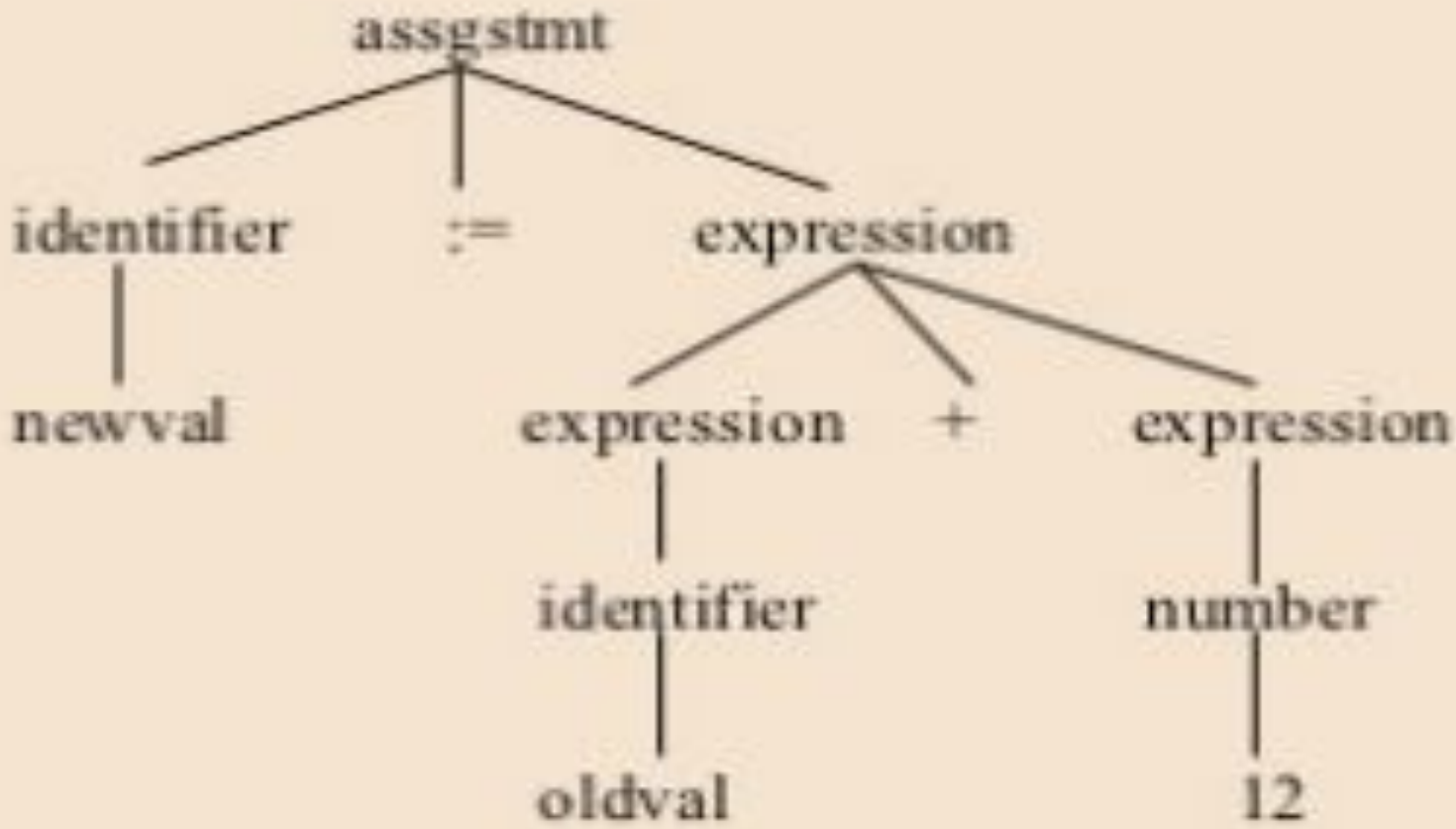
Syntax Analyzer

- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.

- Ex: We use BNF (Backus Naur Form) to specify a CFG

assgstmt -> identifier := expression

Example:


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

The Analysis-Synthesis Model

```
temp1 := inttoreal(10)
temp2 := id2 * temp1
temp3 := id1 + temp2
id1 := temp3
```

code optimization

```
temp1 := id2 * 10.0
id1 := id1 + temp1
```

code generation

```
MOVF id2, R2
MULF #10.0, R2
MOVF id1, R1
ADDF R2, R1
MOVF R1, id1
```

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

Example:

- Semantic analysis
  - Syntactically correct, but semantically incorrect

example:

sum = a + b;

int a;
double sum;
char b;

data type mismatch

Semantic records	
a	integer
sum	double
b	char

Example:

newval := oldval + fact * 1

Id1 := Id2 + Id3 * 1

Temp1 = int to real (1)

Temp2 = Id3 * Temp1

Temp3 = Id2 + Temp2

Id1 = Temp3

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

Example:

newval := oldval + fact * 1

Id1 := Id2 + Id3 * 1

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

Example:

- The above intermediate code will be optimized as:

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

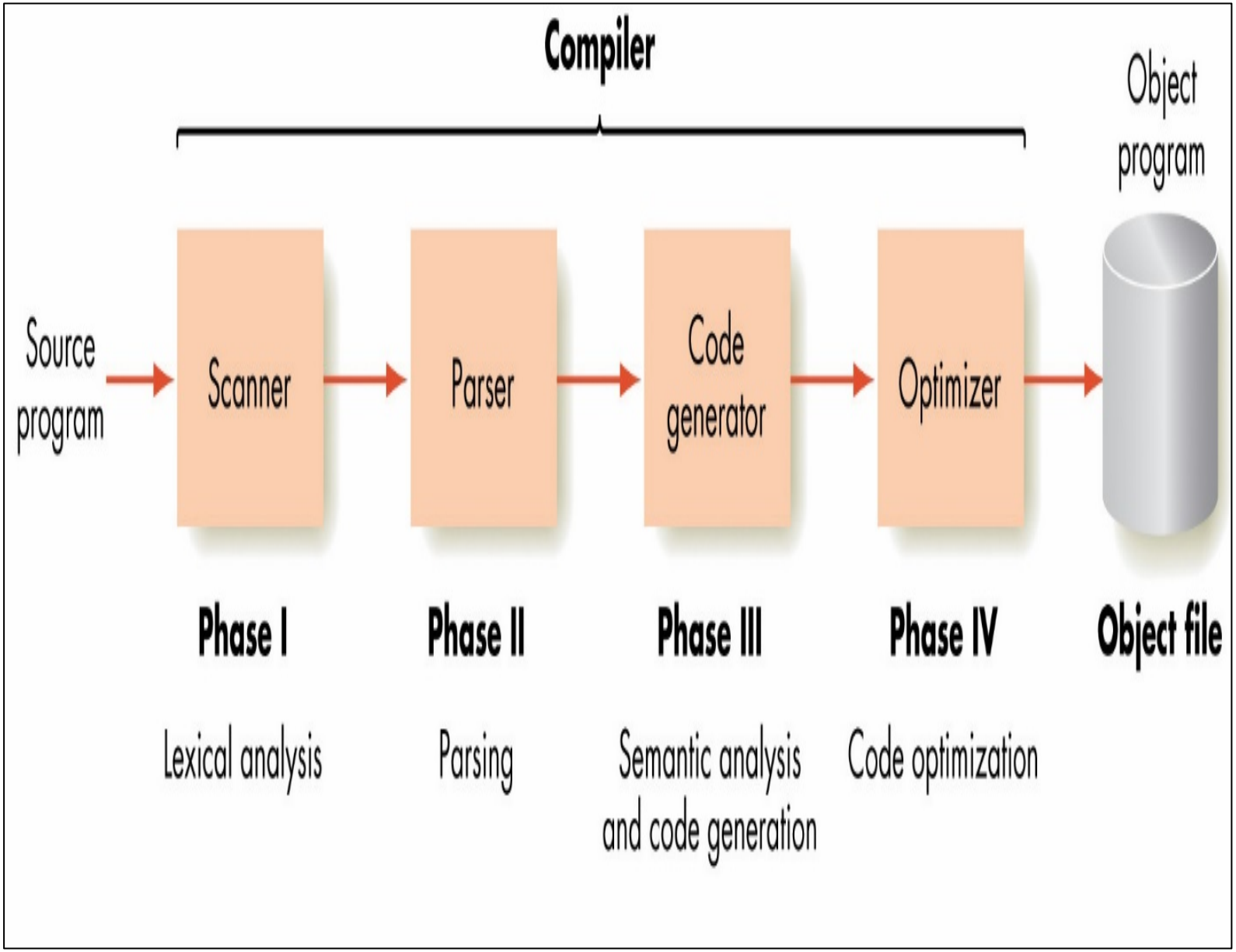
Phase-6: Code Generation

- **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**

Example:

Id1 := Id2 + Id3 * 1

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


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

Symbol-Table Management

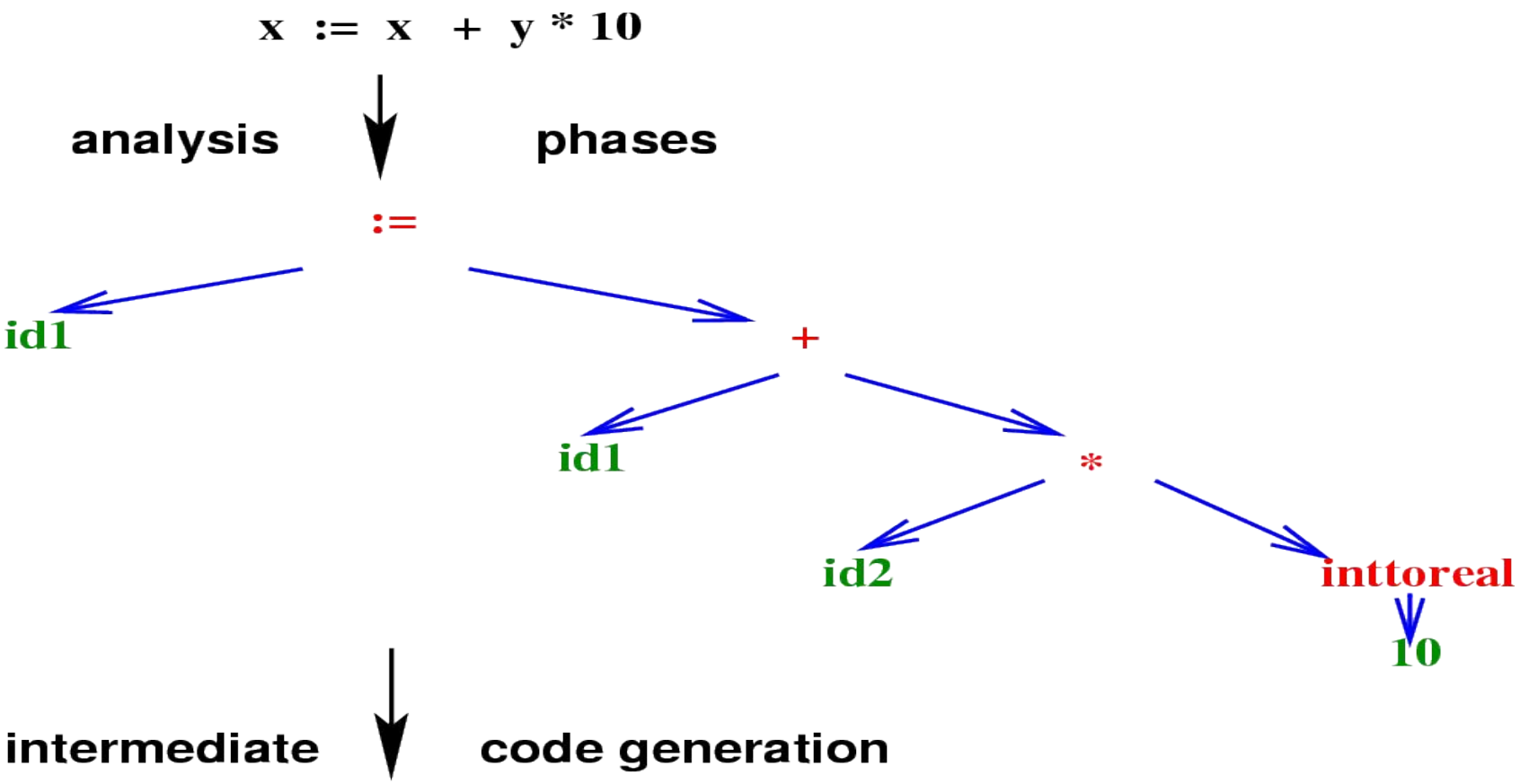
- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.
- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly
- These attributes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

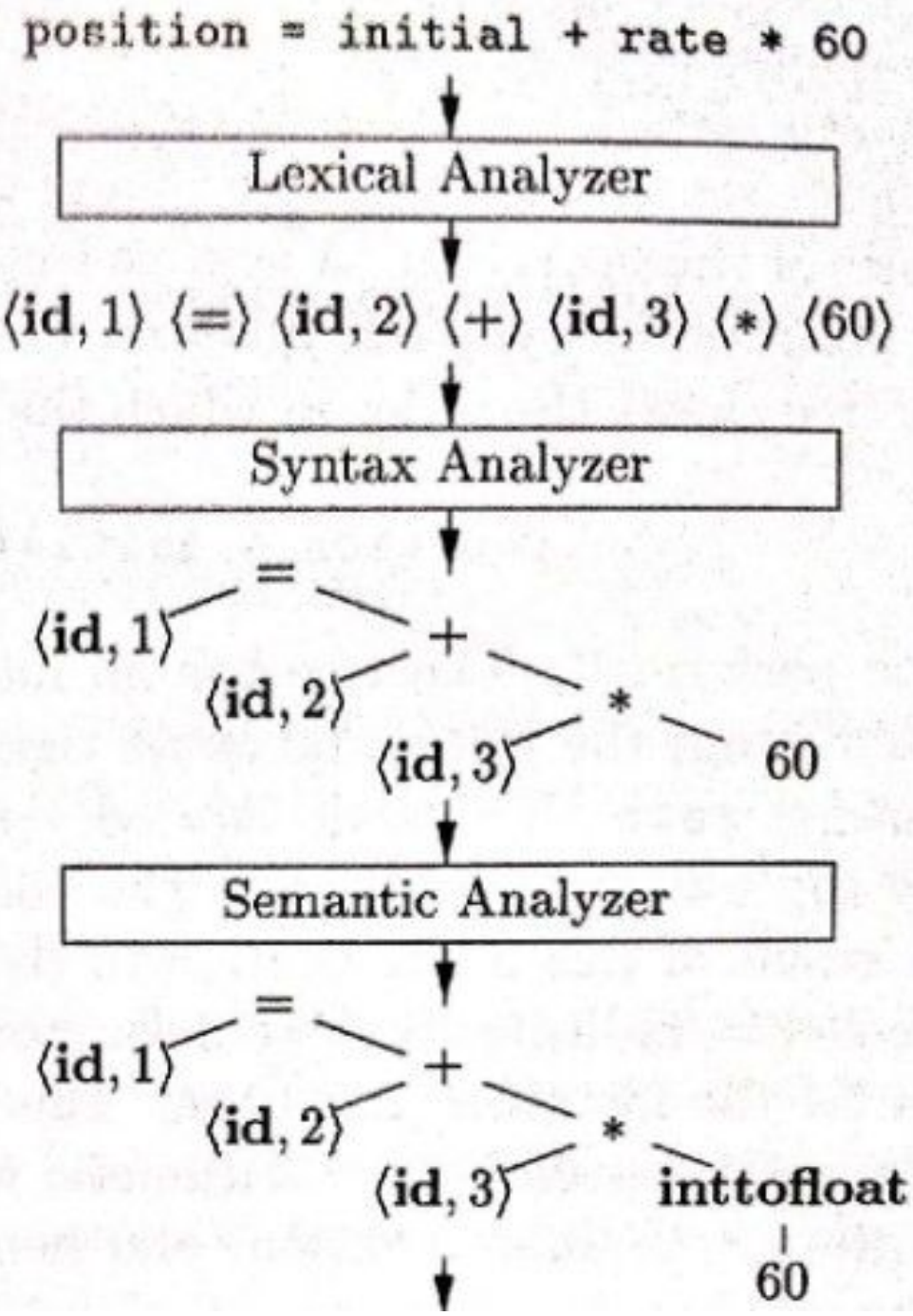
The Analysis-Synthesis Model



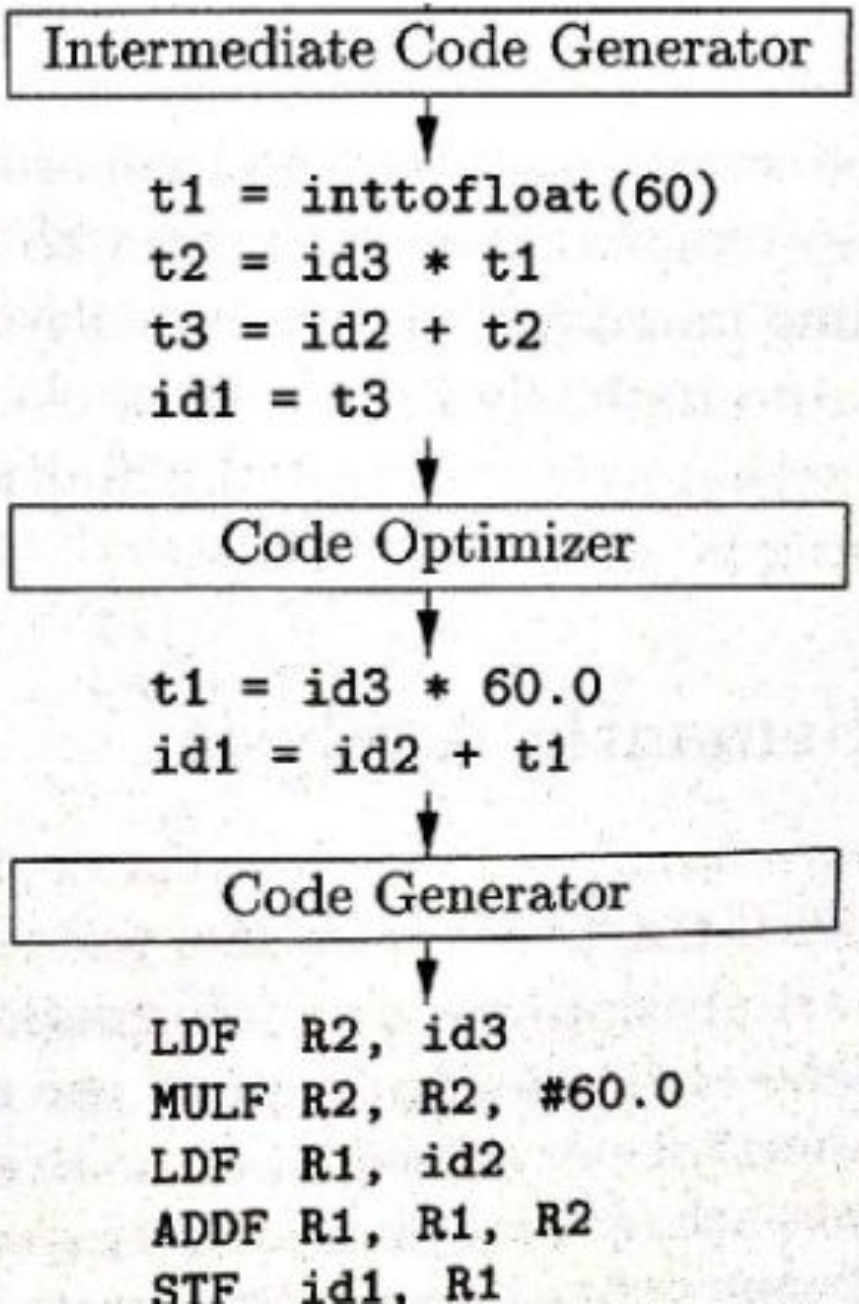
Example

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

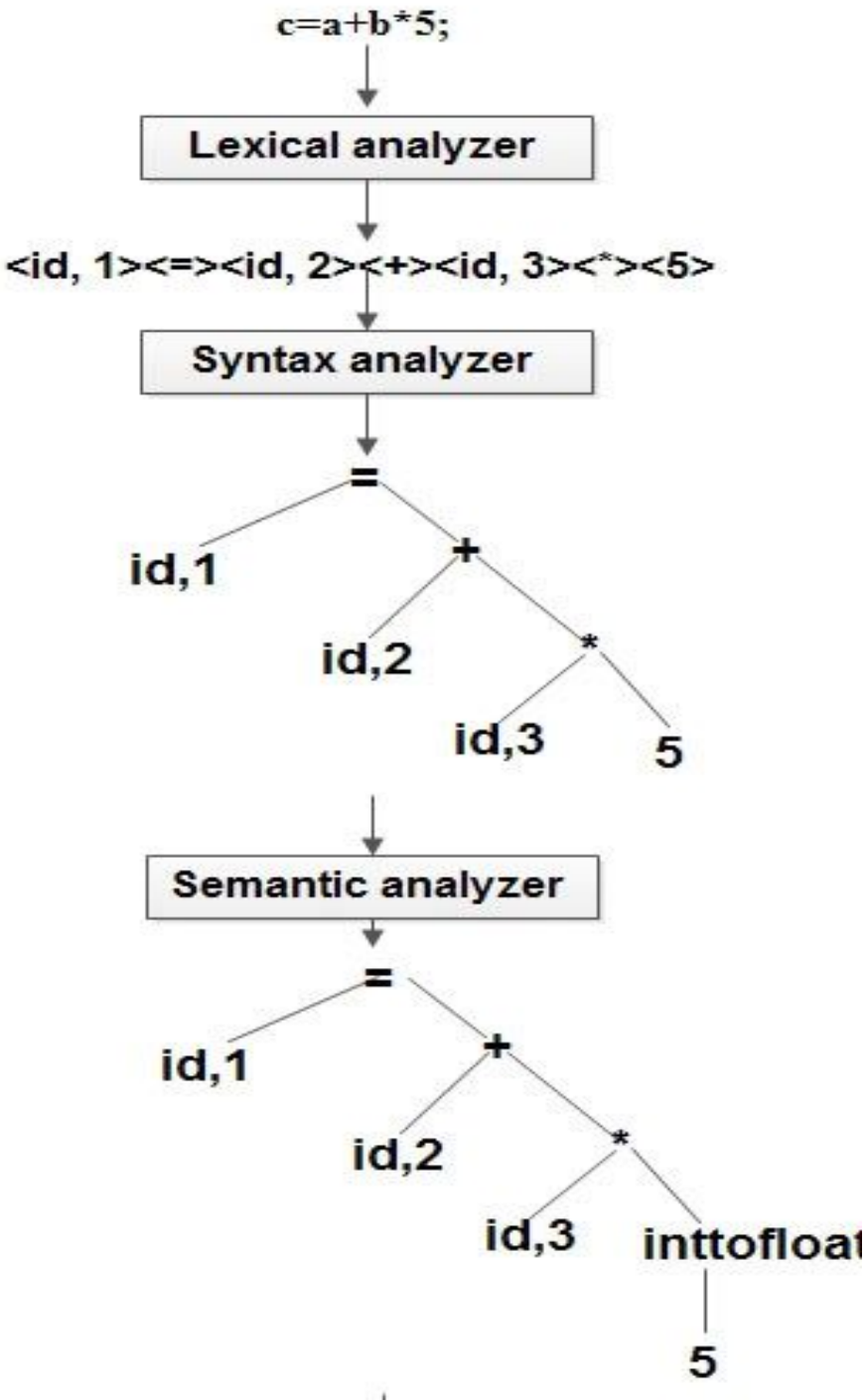
SYMBOL TABLE



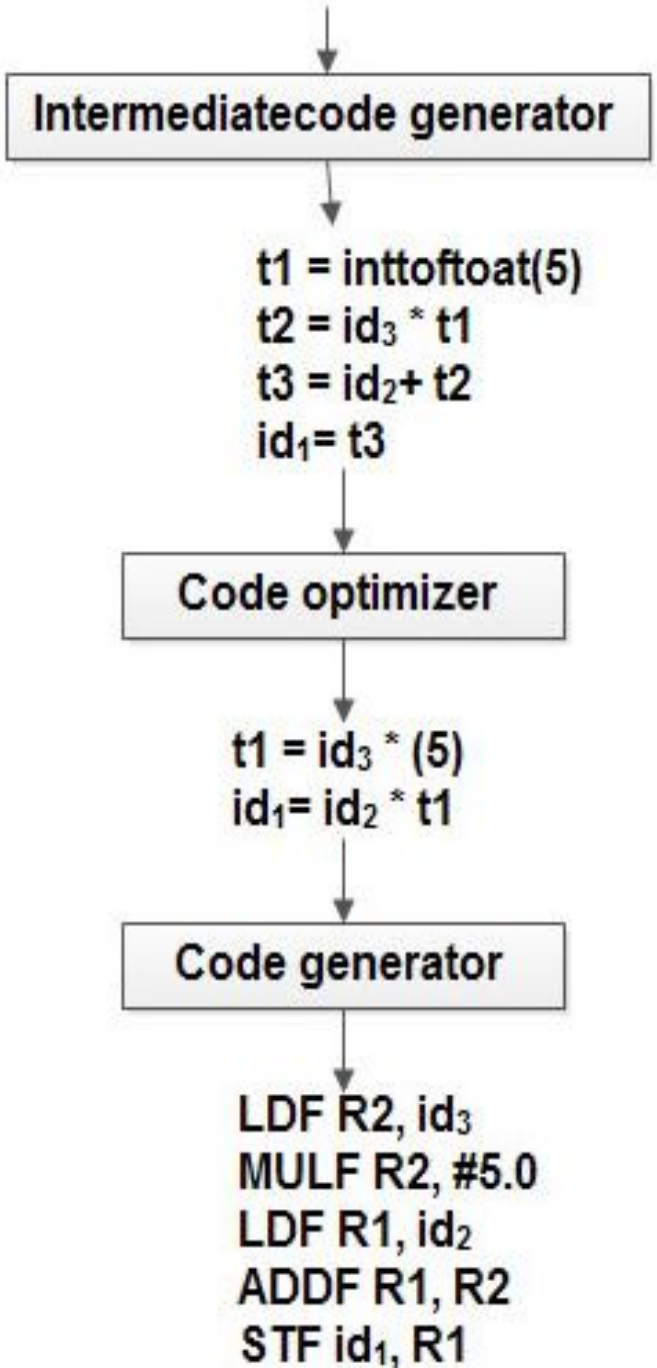
Example


Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1

Example
(c=a+b*5)

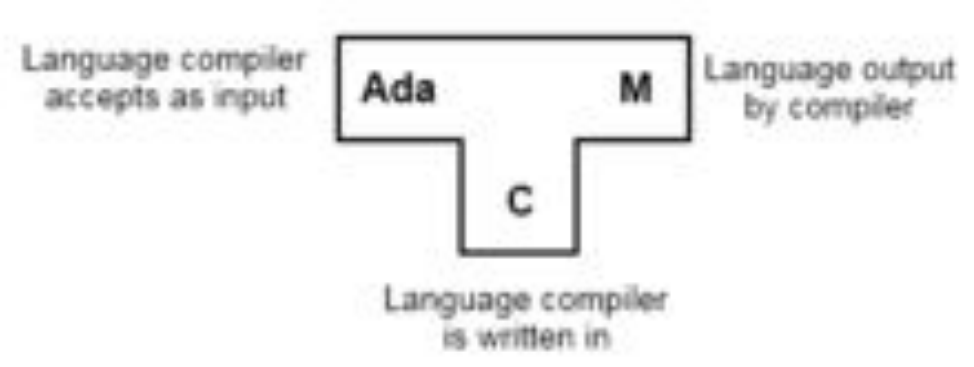


Example
(c=a+b*5)



Types of Compiler

- Native Compiler



- Cross Compiler
- Compilers-Compiler
- Transpiler

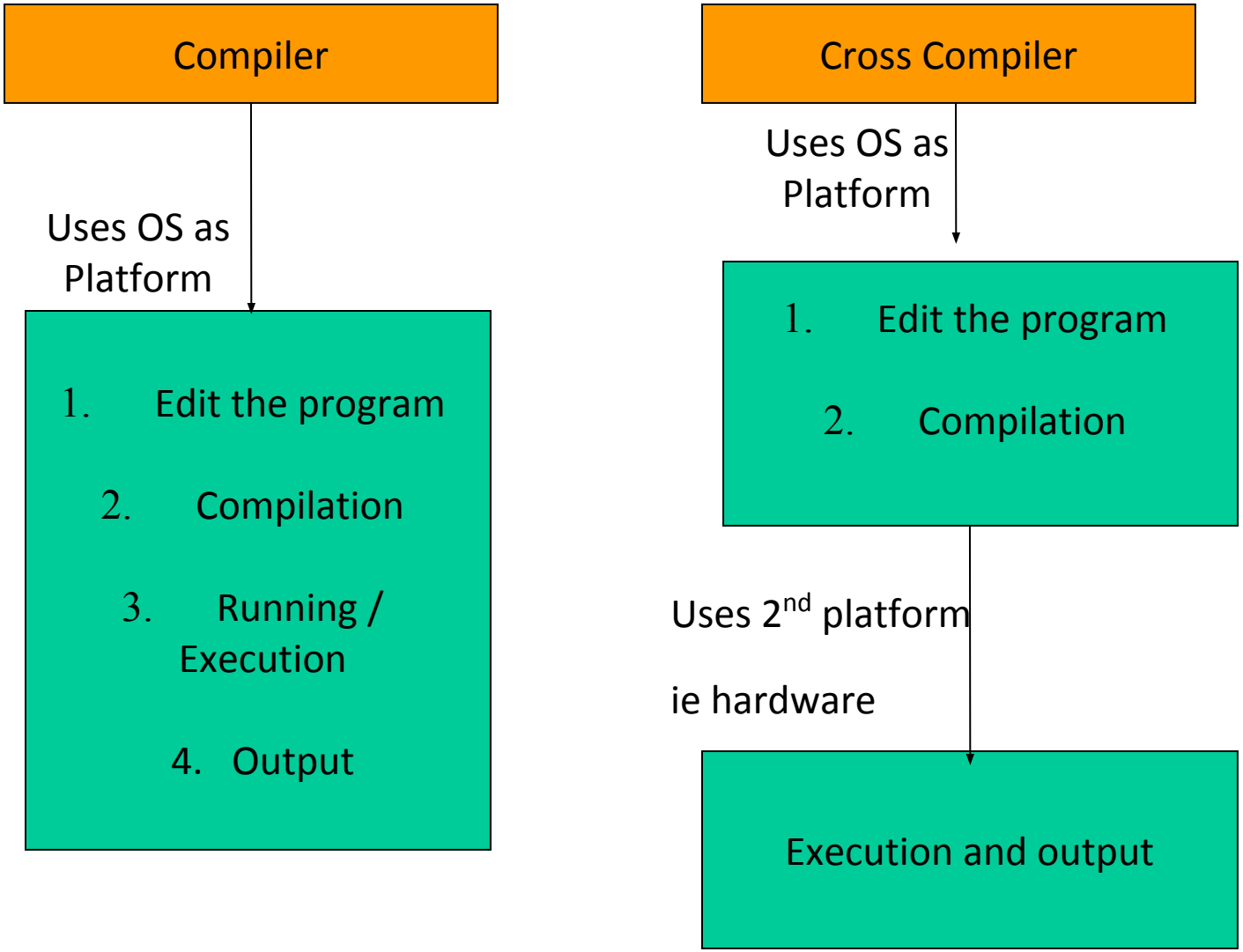
Cross Compiler

A compiler that runs on one computer platform and produces code for another is called as

“Cross compiler”.

A compiler produces a series of files called as object files.

COMPLIERS vs. CROSS COMPLIERS



Compiler-Compiler

- It is a Compiler Generator
- It Generates the Compiler for a Programming Language defined by attribute grammar.
- Language to compile itself is the essence of bootstrapping.

Uses of cross compiler

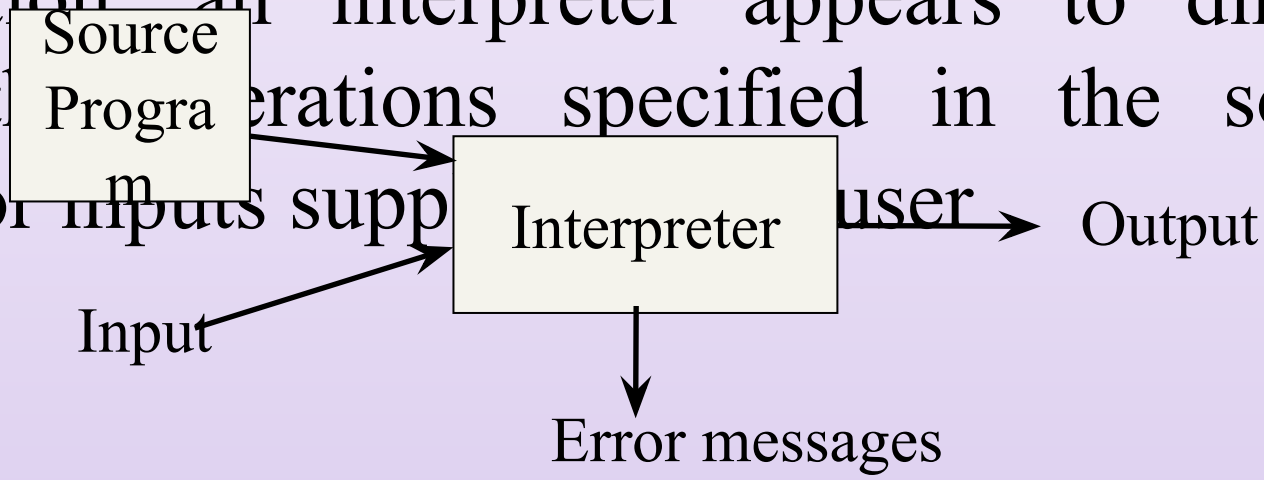
- Embedded computers where a device has extremely limited resources.
- Compiling for multiple machines. For example, a company may wish to support several different versions of an operating system or to support several different operating systems.

Transpiler

- A **source-to-source Compiler**, **transcompiler** or **transpiler** is a type of compiler that takes the source code of a program written in one programming language as its input and produces the equivalent source code in another programming language.
- For example, a source-to-source compiler may perform a translation of a program

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 on inputs supplied by the user



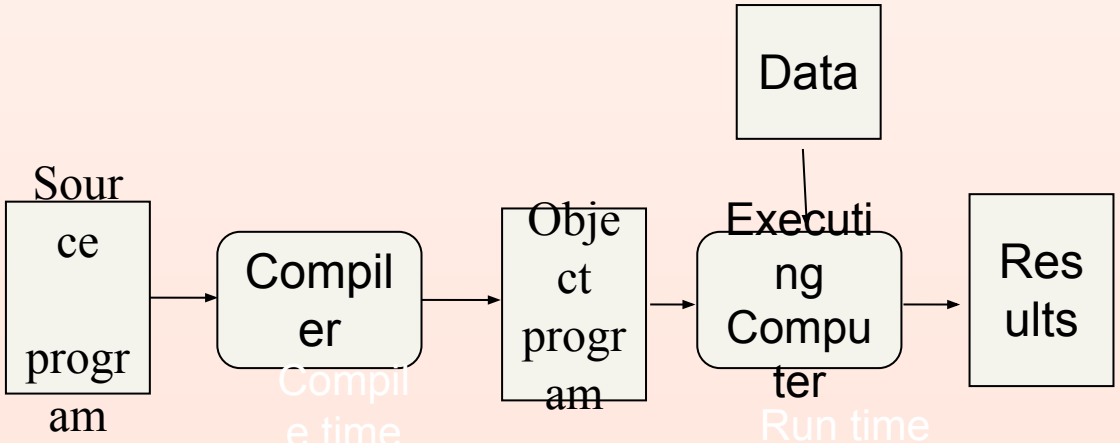
	Compiler	Interpreter
Scanning	**Compiler** scans the **entire program** first and then translates into machine code.	**Interpreter** scans and translates the program **line by line** to equivalent machine code.
Error Report	Compiler gives you **list of all errors** after compilation of whole program.	Interpreter stops the translation at the error generation and will continue when error get solved.
Machine code	Compiler converts the entire program to the machine code when all the errors are removed, **execution**takes place.	Each time the program is executed; every line is check for error and then converted into equivalent machine code.
Debugging	Compiler is **slow** for debugging.	Interpreter is good for **fast** debugging.
Execution Time	Compiler takes **less** execution time.	Interpreter takes **more** execution time.
Speed	Compiler is **faster.**	Interpreter is **slower** in compare to compiler.
After Modification	If you make any modification in program you have to recompile entire program i.e. scan the whole program every time after modification.	If you make any modification and if that line has not been scanned then no need to **recompile** entire program.

#	COMPILER	INTERPRETER
1	Compiler works on the complete program at once. It takes the **entire program** as input.	Interpreter program works line-by-line. It takes **one statement at a time** as input.
2	Compiler generates intermediate code, called the o**bject code or machine code.**	Interpreter does not generate intermediate object code or machine code.
3	Compiler executes conditional control statements (like if-else and switch-case) and logical constructs **faster than interpreter.**	Interpreter execute conditional control statements at a much **slower speed**.
4	**Compiled programs take more** memory because the entire object code has to reside in memory.	Interpreter does not generate intermediate object code. As a result, **interpreted programs are more memory efficient.**
5	Compile once and run anytime. Compiled program does not need to be compiled every time.	Interpreted programs are interpreted line-by-line every time they are run.

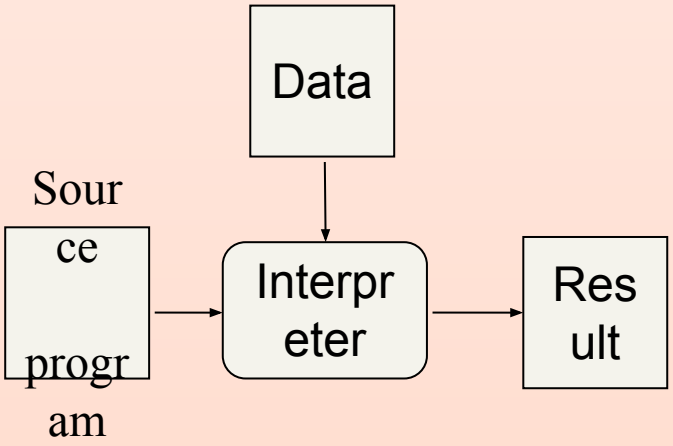
#	COMPILER	INTERPRETER
6	Errors are reported after the **entire program is checked** for syntactical and other errors.	Error is reported as soon as the first error is encountered. Rest of the program will not be checked until the existing error is removed.
7	A compiled language is more difficult to debug.	Debugging is easy because interpreter stops and reports errors as it encounters them.
8	Compiler does not allow a program to run until it is completely error-free.	Interpreter runs the program from first line and stops execution only if it encounters an error.
9	Compiled languages are more efficient but difficult to debug.	Interpreted languages are less efficient but easier to debug. This makes such languages an ideal choice for new students.
10	**Examples** of programming languages that use compilers: C, C++, COBOL	**Examples** of programming languages that use interpreters: BASIC, Visual Basic, Python, Ruby, PHP, Perl, MATLAB, Lisp

Working Process of Compilers Vs Interpreter

Compilation Process:



Interpretive Process:



Summary

- Compiler front-end: lexical analysis, syntax analysis, semantic analysis
  - Tasks: understanding the source code, making sure the source code is written correctly
- Compiler back-end: Intermediate code generation/improvement, and Machine code generation/improvement
  - Tasks: translating the program to a

THANK YOU