INTRODUCTION TO COMPILERS

Compiler:

In general, compiler is a computer program that reads a program written in one language, which is called the source language, and translates it in to another language, which is called the target language. Traditionally, source language was a high level language such as C/C++ and target language was a low level language such as Assembly language/Machine level Language. So, in general compilers can be seen as translators that translate from one language to another.

Definition: A compiler is a translator which is simply a program that reads a program written in one language (source program) and translates it into an equivalent program in another language (target program).

As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

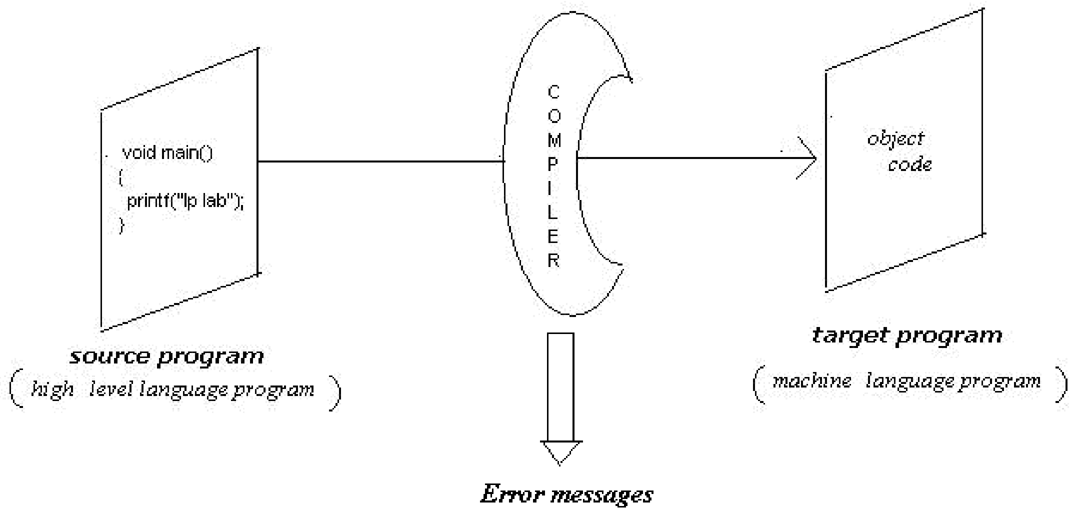


Fig: A compiler

Compilers are sometimes classified as single-pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform. Despite this apparent complexity, the basic tasks that any compiler must perform are essentially the same.

<u>**Design of a Compiler**</u>

Phase:

The term phase often comes up when you talk about compiler construction. Initially, compilers were every simple pieces of single, monolithic software written by one person for the compilation of a simple language. But when the source code of the language to be translated becomes complex and large, the compiler was broken down in to multiple (relatively independent) phases.

Definition: Phase is logically cohesive operation that takes one representation as input and gives another representation as output.

The advantage of having different phases is that the development of the compiler can be distributed among a team of developers. Furthermore, it improves the modularity and reuse by allowing phases to be replaced by improved ones or additional phases (such as further optimizations) to be added to the compiler.

As the design of any system software includes two phases namely Analysis and Synthesis phases, the compiler also can be divided into to analysis phase also called as back end of compiler and synthesis phase also called front end of compiler.

Together these phases further divided into 6 phases such as Lexical Analysis Phase, Syntax Analysis Phase, Semantic Analysis Phase, Intermediate Code Generation, Code Optimization and Code Generation.

All of the aforementioned phases involve the following tasks: • Symbol table Management and • Error handling mechanism.

STRUCTURE OF A COMPILER

A compilation process is so complex that is not reasonable, either from a logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step. For this reason, it is customary to partition the compilation process into a series of sub processes called phases as shown in the figure.

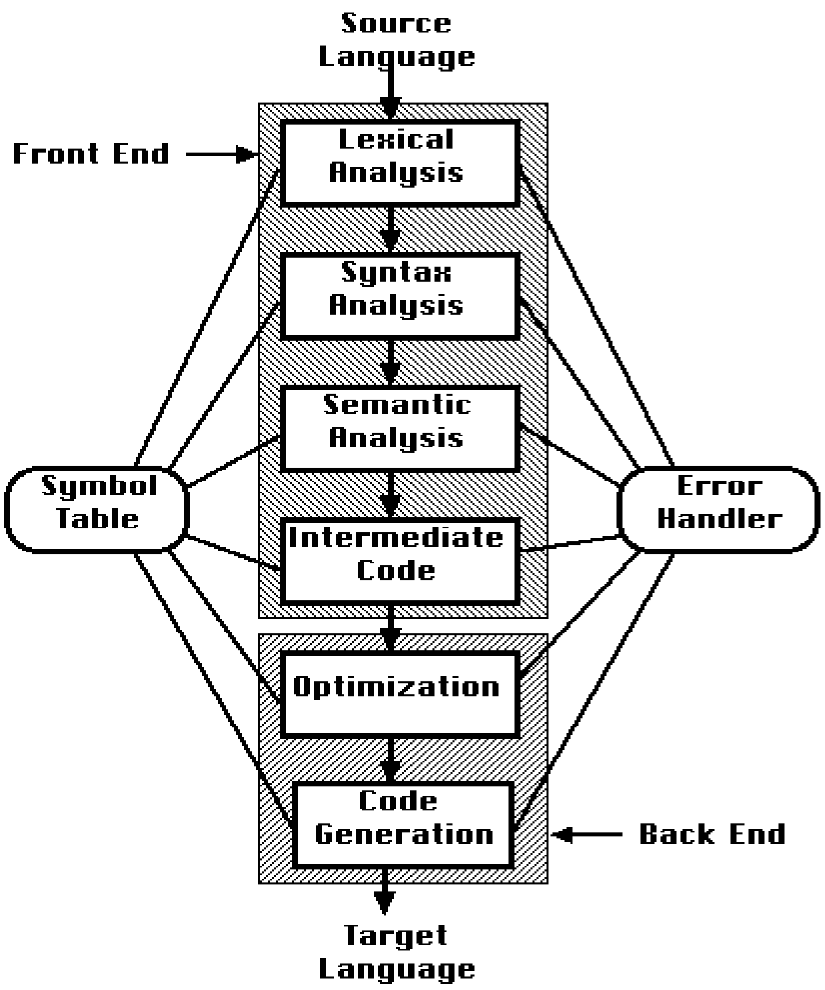


Fig: Phases of a compiler

Lexical Analysis

The lexical analyzer also called as scanner is the interface between the source program and the compiler. The lexical analyzer reads the source program one character at a time, carving the source program into a sequence of atomic units called tokens. Each token represents a sequence of characters that can be treated as a single logical entity. Identifiers, keywords, constants, operators, and punctuation symbols such as commas and parentheses are typical tokens.

Normally a lexical analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from it.

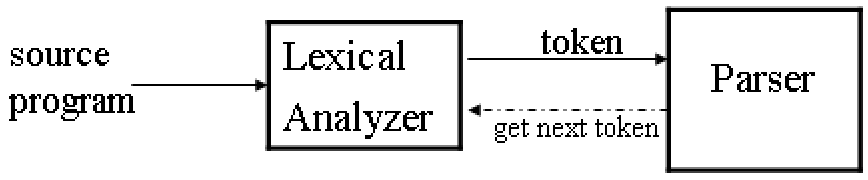


Fig: Lexical Analyzer

• Token: Token is a sequence of characters which some logical meaning that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc.

• Lexeme: Lexeme is instance of a token i.e., group of characters forming a token.

• Pattern: Pattern describes the rule that the lexemes of a token takes. It is the structure that must be matched by strings. In order to recognize a particular patterns, Lexical Analysis phase uses Regular Expressions as the notation by which it recognizes the tokens.

• Once a token is recognized the corresponding entry is made in the symbol table.

Input: stream of characters

Output: Token

Token Template: <token-name, attribute-value>

(eg.) c=a+b*5;

Lexemes and tokens

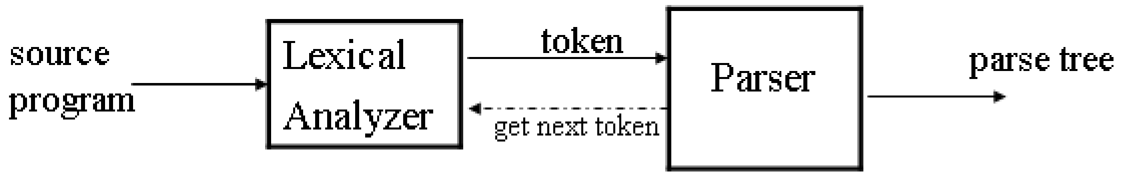
Lexemes	Tokens
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

Hence, <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>

<u>Syntax Analysis</u>:

Syntax Analyzer creates the syntactic structure of the given source program. This syntactic structure is mostly a *parse tree*. It is also known as *parser*. The syntax of a programming is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.

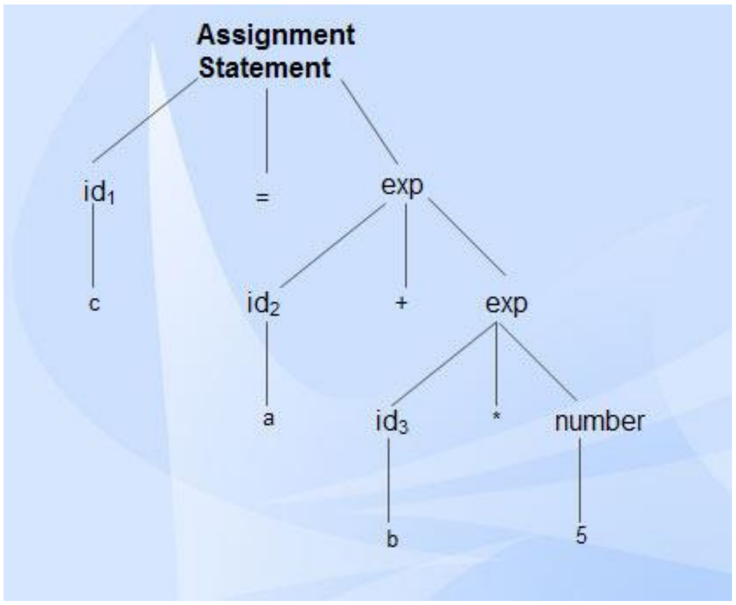
Parser will take the tokens produced by lexical analysis and combines by means of some rules specified in the form of context free grammars and form into a tree like representation called parse tree when there is no syntactical errors are there. If there are syntactical errors are found, they will be directed to error handling mechanism.



<u>Fig: A Parser</u>

Input: *Tokens*

Output: *Parse Tree*



Semantic Analysis:

Semantic analysis is the third phase of compiler in which it checks the meaning for the semantic consistency.

• Type information is gathered and stored in symbol table or in syntax tree.

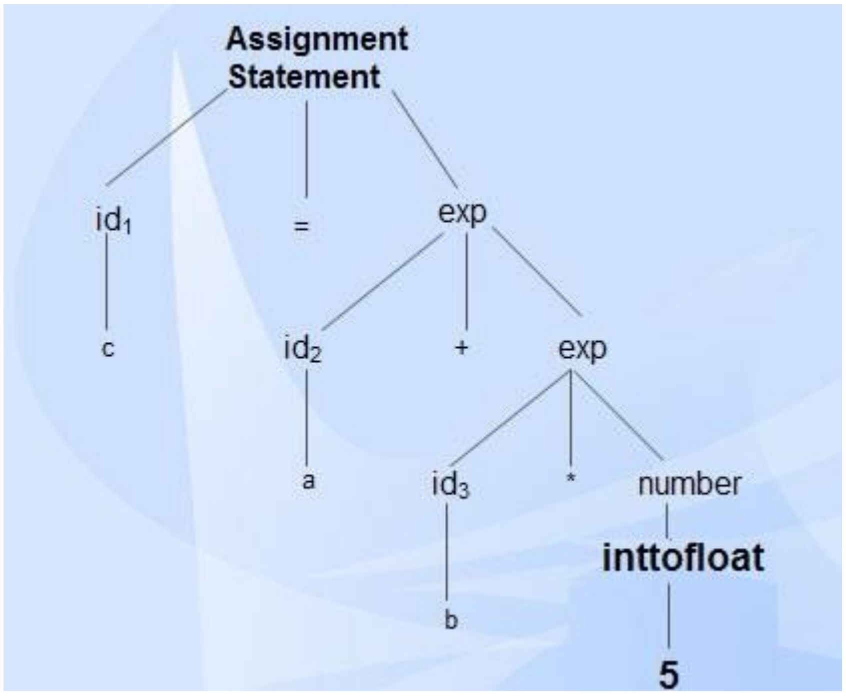
• Performs

- type checking.
- Variable declared before used
- Type Compatibility
- Formal/actual parameter agreement
- Resolve overload symbols etc.

Example: In an expression c = a + b;   if c & a is of type real, and b is of type integer, there is mismatch. If the compiler has the ability to covert the data types implicitly, then the b can be type casted to real internally by the compiler in this semantic analysis phase otherwise it reports an error as mismatch in data types.

Input: *Parse Tree*

Output: *Modified Parse Tree*



<u>Intermediate Code Generation</u>:

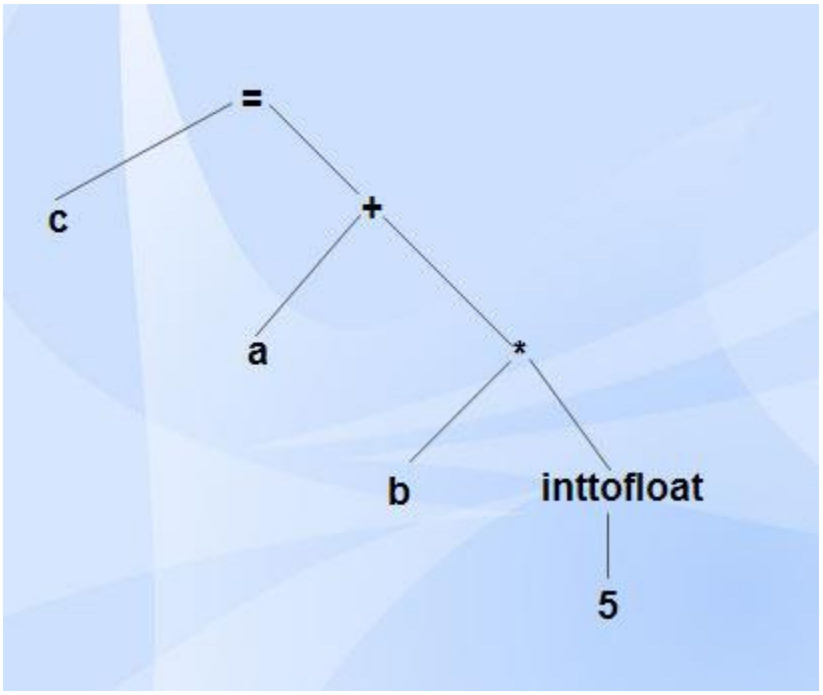
• Intermediate code generation produces intermediate representations for the source program which is as close to the target program and which is easier to develop from the source.

There are various intermediate representations are available in the literature and the most popular ones are:

o Postfix notation: where the operator place after the operand where the precedence's and associativity between operators are already taken care and is very easy to generate the target code.

Ex. A+B*C → ABC*+ OR (A+B)*C → AB+C*

o Syntax tree : Which is modified version of a parse tree in which the redundancies are eliminated in such a way the operators becomes the internal nodes and operands becomes the leaves.



Three Address Code:

One popular type of three address code is "Three Address Code".

Which is most commonly used form where each statement consists of maximum of 3 operands and 1 operation.

A typical three-address code statement is

A:=B op C

where A, B and C are operands and op is a binary operator

Ex.

t_1 = inttofloat (5)

t_2 = id_3* tl

t_3 = id_2 + t_2

id_1 = t_3

Code Optimization:

Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output without changing the meaning of the source program.

• It results in faster running machine code.

• It can be done by reducing the number of intermediate code for a program.

• This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.

• During the code optimization, the result of the program is not affected.

Code optimization is an optional phase. We should use code optimization if the optimized code has execution time less than that of execution time of intermediate code.

• To improve the code generation, the optimization involves

Local Optimization concerns those statements in a single block such as

o Deduction and removal of dead code (unreachable code).

o Calculation of constants in expressions and terms.

o Collapsing of repeated expression into temporary string.

o Removal of unwanted temporary variables.

Loop Optimization concerns those statements involves inner loops such as

- o Loop unrolling.
- o Moving code outside the loop.

And Global Optimization which concerns optimization based on different blocks of statements.

Ex.

t_1 = id_3* 5.0

id_1 = id_2 + t_1

Code Generation:

• Code generation is the final phase of a compiler.

• It gets input from code optimization phase and produces the target code or object code as result.

• Intermediate instructions are translated into a sequence of machine instructions that perform the same task.

• The code generation involves

- o Allocation of register and memory.
- o Generation of correct references.
- o Generation of correct data types.
- o Generation of missing code.

Ex.

MOV R_2, id_3

MUL R_2, # 5.0

MOV R_1, id_2

ADD R_1, R_2

STORE R_1, id_1

Symbol Table Management

- Symbol table is used to store all the information about identifiers used in the program.

- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.

- It allows finding the record for each identifier quickly and to store or retrieve data from that record.

- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

Example

int a, b; float c; char z;

Symbol name	**Type**	**Address**
a	Int	1000
b	Int	1002
c	Float	1004
z	char	1008

Example

```
extern double test (double x);

 double sample (int count)

{

  double sum= 0.0;

    for (int i = 1; i < = count; i++)

      sum+= test((double) i);

      return sum;

  }
```

Symbol name	Type	Scope
test	function, double	extern
x	double	function parameter
sample	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

Error Handling

• Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.

• In lexical analysis, errors occur in separation of tokens.

• In syntax analysis, errors occur during construction of syntax tree.

• In semantic analysis, errors may occur at the following cases:

(i) When the compiler detects constructs that have right syntactic structure but no meaning

(ii) During type conversion.

• In code optimization, errors occur when the result is affected by the optimization. In code generation, it shows error when code is missing etc.

Figure illustrates the translation of source code through each phase, considering the statement

c =*a+ b* * 5.

Error Encountered in Different Phases

Each phase can encounter errors. After detecting an error, a phase must some how deal with the error, so that compilation can proceed.
A program may have the following kinds of errors at various stages:

Lexical Errors

It includes incorrect or misspelled name of some identifier i.e., identifiers typed incorrectly.

Syntactical Errors

It includes missing semicolon or unbalanced parenthesis. Syntactic errors are handled by syntax analyzer (parser).
When an error is detected, it must be handled by parser to enable the parsing of the rest of the input. In general, errors may be expected at various stages of compilation but most of the errors are syntactic errors and hence the parser should be able to detect and report those errors in the program.

The goals of error handler in parser are:

- Report the presence of errors clearly and accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correcting programs.

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

o Panic mode.
o Statement level.
o Error productions.
o Global correction.

Semantical Errors

These errors are a result of incompatible value assignment. The semantic errors that the semantic analyzer is expected to recognize are:

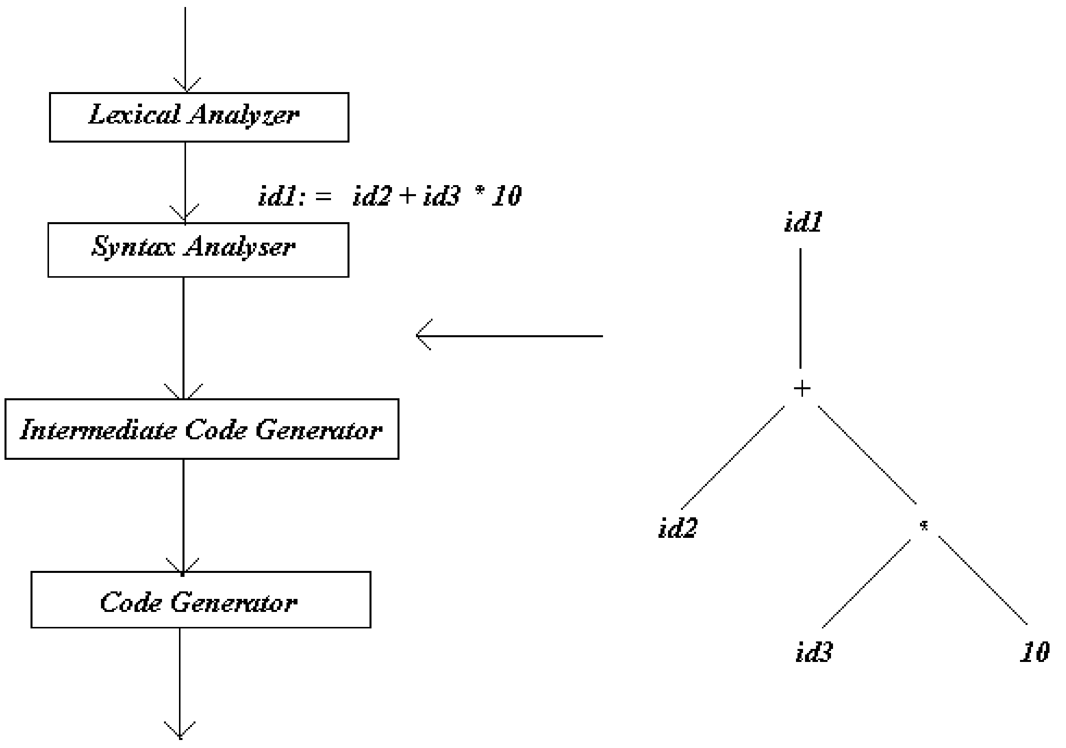
- Type mismatch.
- Undeclared variable.
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Logical errors

These errors occur due to not reachable code-infinite loop.

Consider the following example:

Position: = initial + rate * 10



```
MOV id3, R2
 MUL #10.0, R2
 MOV   id2, R1
 ADD R2, R1
 MOV R1, id1
```

Phase vs Pass in Compiler

Pass and Phase are two terms often used with compilers. Number of passes of a compiler is the number of times it goes over the source (or some form of representation of it). A compiler is broken down in to parts for the convenience of construction. Phase is often used to call such a single independent part of a compiler.

What is a Pass in a Compiler?

A standard way to classify compilers is by the number of "passes" it takes to complete the its construction. Usually, compiling is a relatively resource intensive process and initially computers did not have enough memory to hold such a program that did the complete job. Due to this limitation of hardware resources in early computers, compilers were broken down in to smaller sub programs that did its partial job by going over the source code (made a "pass" over the source or some other form of it) and performed analysis, transformations and translation tasks separately. So, depending on this classification, compilers are indentified as one-pass or multi-pass compilers.

As the name suggests, one-pass compilers compiles in a single pass. It is easier to write a one-pass compiler and also they perform faster than multi-pass compilers. Therefore, even at the time when you had resource limitations, languages were designed so that they could be compiled in a one-pass (e.g. Pascal). On the other hand, a typical multi-pass compiler is made up of several main stages. The first stage is the scanner (also known as the lexical analyzer). Scanner reads the program and converts it to a string of tokens. The second stage is the parser. It converts the string of tokens in to a parse tree (or an abstract syntax tree), which captures the syntactic structure of the program. Next stage is the that interpret the semantics of the syntactic structure. The code optimizations stages and final code generation stage follow this.

What is the difference between Phase and Pass in Compiler?

Phase and Pass are two terms used in the area of compilers. A pass is a single time the compiler passes over (goes through) the sources code or some other representation of it. Typically, most compilers have at least two phases called front end and back end, while they could be either one-pass or multi-pass. Phase is used to classify compilers according to the construction, while pass is used to classify compilers according to how they operate.