

# COMPUTER SCIENCE & INFORMATION TECHNOLOGY

## Compiler Design



Comprehensive Theory  
*with Solved Examples and Practice Questions*



**MADE EASY**  
Publications

[www.madeeasypublications.org](http://www.madeeasypublications.org)



### **MADE EASY Publications Pvt. Ltd.**

**Corporate Office:** 44-A/4, Kalu Sarai (Near Hauz Khas Metro Station), New Delhi-110016 | **Ph. :** 9021300500

**Email :** infomep@madeeasy.in | **Web :** www.madeeasypublications.org

## **Compiler Design**

© Copyright by MADE EASY Publications Pvt. Ltd.  
All rights are reserved. No part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photo-copying, recording or otherwise), without the prior written permission of the above mentioned publisher of this book.



**MADE EASY Publications Pvt. Ltd.** has taken due care in collecting the data and providing the solutions, before publishing this book. In spite of this, if any inaccuracy or printing error occurs then **MADE EASY Publications Pvt. Ltd.** owes no responsibility. We will be grateful if you could point out any such error. Your suggestions will be appreciated.

## **EDITIONS**

First Edition : 2015  
Second Edition : 2016  
Third Edition : 2017  
Fourth Edition : 2018  
Fifth Edition : 2019  
Sixth Edition : 2020  
Seventh Edition : 2021  
Eighth Edition : 2022

**Ninth Edition : 2023**

# CONTENTS

## Compiler Design

### CHAPTER 1

#### Introduction to Compiler ..... 2-15

1.1	Compiler .....	2
1.2	Compiler Stages .....	3
1.3	Grouping of Phases .....	9
1.4	Passes in a Compiler .....	9
	<i>Student Assignments</i> .....	10

### CHAPTER 2

#### Lexical Analysis ..... 16-28

2.1	Introduction .....	16
2.2	Tokens, Patterns and Lexemes .....	16
2.3	Recognition of Tokens .....	17
2.4	Attributes for Tokens .....	19
2.5	Lexical Errors .....	20
	<i>Student Assignments</i> .....	24

### CHAPTER 3

#### Syntax Analysis (Parser) ..... 29-81

3.1	Introduction .....	29
3.2	Prerequisites .....	30
3.3	Top-Down Parsing .....	34
3.4	LL(1) Parsing .....	36
3.5	Bottom-Up Parsing .....	45
3.6	Canonical LR Parsing (CLR) and LALR .....	54
3.7	Operator Precedence Parsing .....	62
3.8	Hierarchy of Grammar Classes .....	64
	<i>Student Assignments</i> .....	66

### CHAPTER 4

#### Syntax Directed Translation ..... 82-113

4.1	Introduction .....	82
4.2	Syntax-Directed Definition .....	82

4.3	Construction of Syntax Trees .....	91
4.4	Bottom-up Evaluation of S-Attributed Definitions .....	93
4.5	L-Attributed Definitions .....	94
4.6	Bottom-up Evaluation of Inherited Attributes .....	95
4.7	Intermediate Code Generation .....	96
4.8	Dependency Graph Generation using Semantic Rules (SDT) .....	98
4.9	Syntax-Directed Translation for Intermediate Code Generation .....	100
	<i>Student Assignments</i> .....	101

### CHAPTER 5

#### Intermediate Code Generation ..... 114-142

5.1	Introduction .....	114
5.2	Intermediate Representations .....	114
5.3	Basic Blocks and Flow Graphs .....	123
5.4	Peephole Optimization .....	134
	<i>Student Assignments</i> .....	137

### CHAPTER 6

#### Run-Time Environment ..... 143-159

6.1	Introduction .....	143
6.2	Storage Organization .....	144
6.3	Stack Allocation Space .....	146
6.4	Heap Allocation .....	149
6.5	Access to Non-Local Names (Scope of Variables) .....	149
6.6	Symbol Table Implementation .....	153
6.7	Parameter Passing .....	154
	<i>Student Assignments</i> .....	157



# Compiler Design

---

## GOAL OF THE SUBJECT

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space. In order to understand or construct the compiler one must be aware of its design principles. Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

## INTRODUCTION

In this book we tried to keep the syllabus of Compiler Design around the GATE syllabus. Each topic required for GATE is crisply covered with illustrative examples and each chapter is provided with Student Assignment at the end of each chapter so that the students get the thorough revision of the topics that he/she had studied. This subject is carefully divided into six chapters as described below.

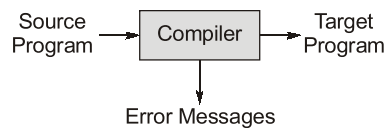
1. **Introduction to Compiler:** In this chapter we will introduce you to the Analysis-Synthesis model of compilation, various stages of Compilation (namely: lexical, syntax, semantic, intermediate code generation, code optimization and code optimization), grouping of various stages into analysis phase and synthesis phase, passes in compiler and finally we discuss the bootstrapping.
2. **Lexical Analysis:** In this chapter we will study Tokens, lexemes and their patterns and finally we discuss various ways of specifying tokens.
3. **Syntax Analysis (Parser):** In this chapter we introduce you the types of parsers, Top down parser: LL (1) parsing, Bottom up parsers: LR parser, SLR parser, CLR parser, LALR parser and Operator precedence parsing.
4. **Syntax Directed Translation:** In this chapter we will study about the Syntax directed definition, attributes (synthesized and inherited), Construction of Syntax trees, of S-Attributed Definitions, L-Attributed Definitions, Bottom up evaluation of inherited Attributes, dependency graph using SDT, SDT for intermediate code generation.
5. **Intermediate Code Generation:** In this chapter, we will study about the code generation for program and their various representations. We will also discuss basic blocks and flow graphs.
6. **Run Time Environment:** In this chapter we will study about the activation trees, control stacks, Storage organization, storage allocation strategies, scope of variables, Symbol table representations, parameter passing.



# Introduction to Compiler

## 1.1 Compiler

A compiler is a program that reads a program written in one language the source language-and translates into an equivalent program in another language-the target language. The translation process should also report the presence of errors in the source program.



### 1.1.1 Interpreter

An interpreter is a program that executes other programs. The interpreter and the underlying machine are combined into a virtual machine and simulate the execution of the input program on the virtual machine.

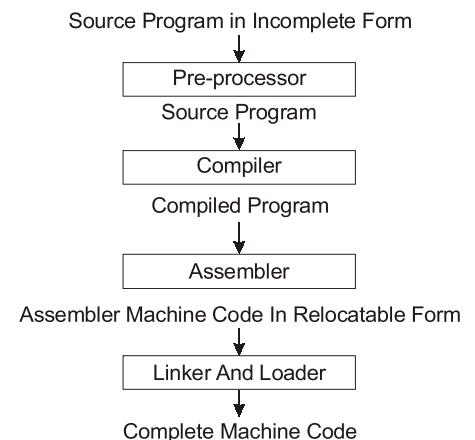
In general, the source language is any high level language and the object language is an assembly/machine language. We can show the general process of compilation by using following diagram.

Pre-processor is a program that processes its input data to produce output that is used as input to another program (compiler). The output is said to be a preprocessed form of the input data, which is generally used by compiler. Very basic work of pre-processor is to insert the files which are named in source program.

For *example*, let in source program, it is written `#include "stdio.h"`. Pre-processor replace this file by its contents in the produced output.

The basic work of linker is to merge object codes (that has not even connected), produced by compiler, assembler, standard library functions, and operating system resources.

The codes generated by compiler, assembler, and linker are generally re-locatable by its nature, mean to say, the starting location of these codes are not determined, means, they can be anywhere in the computer memory. Thus, the basic task of loader is to find/calculate the exact address of these memory locations.



### 1.1.2 The Analysis-Synthesis Model of Compilation

There are two parts of compilation:

- **Analysis:** The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- **Synthesis:** The synthesis part constructs the desired target program from the intermediate representation.

#### 1. Analysis of Source Program

In compiling, analysis consists of three phases:

1. **Linear/Lexical Analysis:** The stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning. The lexical analysis is performed by scanner. The working of the scanner is as follows:
  - The scanner reads characters from the source program.
  - The scanner groups the characters into lexemes (sequences of characters that “go together”).
  - Each lexeme corresponds to a token; the scanner returns the next token (plus maybe some additional information) to the parser.
  - The scanner may also discover lexical errors (e.g., erroneous characters).The definitions of what is a lexeme, token, or bad character all depend on the source language.
2. **Syntax/Hierarchical Analysis:** It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source program are represented by a parse tree.
3. **Semantic Analysis:** The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. An important component of semantic analysis is type checking. It may also annotate and/or change the abstract syntax tree (e.g., it might annotate each node that represents an expression with its type).

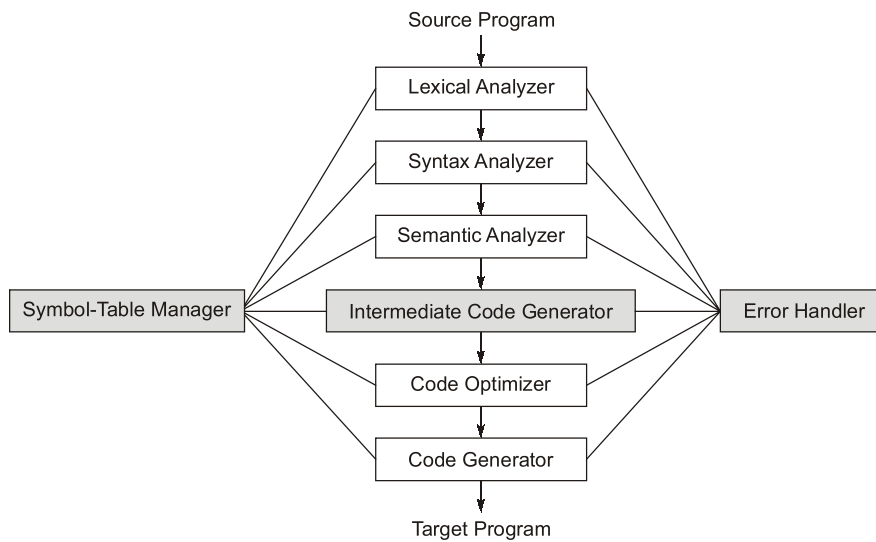
#### 2. Synthesis of Source Program

In compiling, synthesis consists of three phases:

1. **Intermediate Code Generation:** After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation should have two important properties; it should be easy to produce, and easy to translate into the target program. The intermediate code can have a variety of forms.
2. **Code Optimization:** In the optimization phase, the compiler performs various transformations in order to improve the intermediate code. These transformations will result in faster-running machine code.
3. **Code Generation:** The final phase in the compilation process is the generation of target code. This process involves selecting memory locations for each variable used by the program. Then, each intermediate instruction is translated into a sequence of machine instructions that performs the same task.

## 1.2 Compiler Stages

In general the process of compiler is divided into following six stages/phases. Each phase interact with symbol table administrator and error handling.



The first three phases, forming the bulk of the analysis portion of a compiler and the last three phases, forming the bulk of the synthesis portion of a compiler.

### 1. Lexical Analysis

It is a program which categorizes character sequence into lexemes and produces a sequence of tokens as an output for parser. Lexeme is a sequence of characters from the input program. Token is just a representative for the bodies that form the text of the program.

LEXEMES  $\xrightarrow{\text{Pattern Rule which tells when a sequence of characters make a token}}$  TOKEN

For example **integer** is a lexeme, i, n, t, e, g, e, r is the rule, and integer is its token. Similarly **madeeasy** is a lexeme, any sequence of alphanumeric starting with a letter is rule, and finally **madeeasy** becomes a token.

Let us consider the following example.

While (a >= b) a = a - 2;

We can represent it in the form of lexemes and tokens as under.

Lexemes	Tokens	Lexemes	Tokens
while	WHILE	a	IDENTIFIER
(	LPAREN	=	ASSIGNMENT
a	IDENTIFIER	a	IDENTIFIER
>=	COMPARISON	-	ARITHMETIC
b	IDENTIFIER	2	INTEGER
)	RPAREN	;	SEMICOLON

Let us also consider the following piece of code.

```

/* C program by madeeasy*/
main()
{
printf ("madeeasy/n");
}
  
```



Let us suppose that the length of words of memory is 4 bytes, and then the starting form of the input code is as follows:

/	*		C		p	r	o	g	r	a	m		b	y		m	a	d	e	e	a	s	y		*	/
	m	a	i	n	(	)		{		p	r	i	n	t	f	(	"	m	a	d	e	e	a	s	y	/
n	"	)	;		}																					

Lexical analyzer reads the input character one by one. Then according to the regulation of lexical grammar of C language, the character stream is grouped into different token and so on, it becomes the stream of tokens. In C language, the tokens can be KEYWORDS, IDENTIFIERS, INTEGERS, REAL NUMBERS, a notation of SINGLE CHARACTER, COMMENTS, and CHARACTER STRINGS, etc. For example, the lexical analyzer starts its work with reading the first word of the input. It reads the "/", it knows that this is not a letter, rather it is an operator. However, as expression with operator is missing, so it continues its reading to the second character to see if it is "\*" or not. If it is not "\*", then it is wrong. Otherwise it knows that the combination of "/" and "\*" forms the identification of the start of comment line, and all the character string before the identification of the end of comment line, i.e., the combination of "\*" and "/" is a comment.

**Example:** The number of tokens in the FORTRAN statement `DO 10 I = 1, 25` is seven.

KEYWORD = 'DO'; STATEMENT-LABEL = '10'; IDENTIFIER = 'I'; OPERATOR = '='; CONSTANT = '1'; COMMA = ',' and CONSTANT = '25'.

## 2. Syntax Analysis (Parsing)

It is a program that takes sequence of tokens as an input from lexical analysis phase and classify it as an expression with their expression-type. The expression is based on the rules of the source language. For example, consider the following piece of code written in source language.

$$x_1 = x_2 * x_3 + x_4$$

**Lexical analyzer converts this code in following frame:**

Identifier ( $x_1$ ) assignment operator (=) identifier ( $x_2$ ) binary operator (\*) identifier ( $x_3$ ) binary operator (+) identifier ( $x_4$ ). Let the rules of expression in source language is:

1. CONSTANT and IDENTIFIERS are expressions; and
2. If A and B are two expressions then A BINARY-OPERATOR B is also an expression.

Based on above rules, syntax analyzer performs the following action.

A = IDENTIFIER (2) and B = IDENTIFIER (3) – Both are expressions because from rule (1) IDENTIFIERS are expressions. Its type is "expression".

C = A BINARY-OPERATOR (1)B – It is an expression because from rule (2), if A and B are two expressions then A BINARY-OPERATOR B is also an expression. Its type is "expression".

D = IDENTIFIER (4) – It is expression because from rule (1) IDENTIFIERS are expression. Its type is "expression".

E = C BINARY-OPERATOR (2) D – It is an expression because from rule (2), if A and B are two expressions then A BINARY-OPERATOR B is also an expression. Its type is "expression".

Finally, we get:

id ASSIGNMENT-OPERATOR EXPRESSION – It is an expression and its type is "assignment".

The form of expressions can be best presented by a parse tree or a syntax tree.

**NOTE:** The main difference between lexical analyzer and syntax analyzer is that: lexical analyzer very rarely uses recursion while syntax analyzer almost always uses recursion.

### 3. Semantic Analysis

It takes input from syntax analysis phase generally in the form of parse table. It decides that whether the input has properly defined according to the rules of source language. It performs two actions: (1) type checking; and (2) type coercion based on type rules. In general, type rules are of two types: (1) expression-based type rules; and (2) assignment-based type rules.

Expression-based Types Rules		
Expression	What semantic analyzer determine	Type of expression
Constant	PROPER ACCORDING TO RULE	CONSTANT
Variable	PROPER ACCORDING TO RULE	VARIABLE
Operator	IF BINARY OR UNARY – PROPER ACCORDING TO RULE	OPERATOR

Expression-based Types Rules
Let A is a variable of TYPE(1) and B is properly defined expression of TYPE (2); and if we have TYPE(1) = TYPE(2), where TYPE(1) is an assignable type than A ASSIGN B is a properly defined assignment.

### 4. Intermediate Code Generation

Combination of lexical analysis, syntax analysis, and semantic analysis is called analysis phase of compiler. After analysis phase, the input source program is fragmented into a parse tree and a symbol table. Using information from these two, compilers starts the procedure of target/object code generation. Generally, code generation is done in two steps: (1) compiler generate code for general machine; and then (2) convert this general machine-based code to the code for particular machine by using a translator. A very general intermediate language is known as three-address code. Generation of code is recursive in nature, and for generating a code we use syntax tree; and therefore the procedure is also called syntax-directed translation.

For example consider the following piece of code:  $x_1 = x_2 * x_3 + x_2$

Its equivalent syntax tree is as follows.

Syntax Tree	Equivalent Generated Code
<pre> graph TD     Eq["="] --&gt; x1["x1"]     Eq --&gt; Plus["+"]     Plus --&gt; Mult["*"]     Plus --&gt; x2_2["x2"]     Mult --&gt; x2_1["x2"]     Mult --&gt; x3["x3"] </pre>	$t_3 = x_2$ $t_4 = x_3$ $t_1 = t_3 * t_4$ $t_2 = x_2$ $x_1 = t_1 + t_2$

where 't' denotes temporary variable.

### 5. Code Optimization

The code optimization is a process to improve the code so that the overall run time improves and space requirement reduces. There are lots of methods to improve the code.

### 6. Code Generation

It is the final stage of compiler process to generate the code for particular machine. Generally in this stage, compiler works on memory management and register allocation.

### 1.2.1 Symbol-Table Management

A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table. However, the attributes of an identifier cannot normally be determined during lexical analysis. The remaining phases enter information about identifiers into the symbol table and then use this information in various ways.

### 1.2.2 Error Detection and Reporting

Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.

#### Explain all the Necessary Phases and Passes in a Compiler Design?

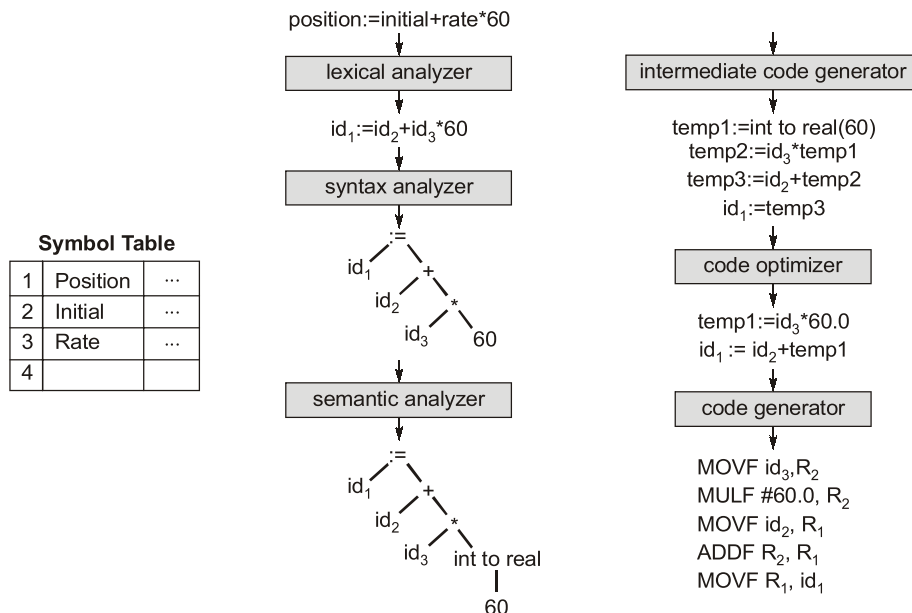
**Solution:** Since writing a compiler is a non-trivial task, it will be a good idea to structure the work. A representative work of doing this will be to split the compilation into several phases which operate in sequence (though in actual practice, they are often interleaved), each phase takes the result produced by its previous phase as its input.

#### 1. Analysis Phases

As translation progresses, the compiler's internal representation of the source program changes. We illustrate these representations by considering the translation of the statement

`position := initial + rate * 60`

Following figure shows the representation of this statement after each phase.



- **The Lexical Analysis Phase:** The lexical analysis phase reads the characters in the source program and groups them into streams of tokens; each token represents a logically cohesive sequence of characters, such as identifiers, operators, and keywords. The character sequence that forms a token is called a "lexeme." Certain tokens are augmented by the lexical value; that is, when an identifier like 'position' is found, the lexical analyzer not only returns id, but it also enters the lexeme position

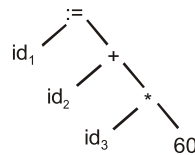
into the symbol table if it does not already exist there. It returns a pointer to this symbol table entry as a lexical value associated with this occurrence of the token `id`. In our example, tokens will be generated as:

The identifier position, The assignment symbol `:=`, The identifier initial, The plus sign, The identifier rate, The multiplication sign and The number 60.

Internal representation of our example after the lexical analysis will be  $id_1 := id_2 + id_3 * 60$

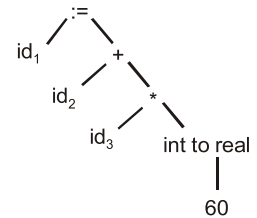
The subscripts 1, 2 and 3 are used for convenience; the actual token is `id`.

- **The Syntax Analysis Phase:** The syntax analysis phase imposes a hierarchical structure on the token stream, shown below



- **The Semantic Analysis Phase:** During the semantic analysis, it is considered that in our example all identifiers have been declared to be reals and that 60 by itself is assumed to be an integer. Type checking of syntax tree reveals that `*` is applied to a real, 'rate', and an integer, 60. The general approach is to convert the integer into a real. This has been achieved by creating an extra node for the operator `int to real` that explicitly converts an integer into a real.

Alternatively, since the operand of `int to real` is a constant, the compiler may instead replace the integer constant by an equivalent real constant.



## 2. Synthesis Phases

- **Intermediate Code Generation:** We consider an intermediate form called "three-address code". The source program of our example will appear in three-address code as

```

temp1 := int to real(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1    := temp3
  
```

- **Code Optimization:** A natural algorithm generates the intermediate code, using an instruction for each operator in the tree representation after semantic analysis, even though there is a better way to perform the same calculation, using the two instructions

```

temp1 := id3 * 60.0
id1    := id2 + temp1
  
```

There is nothing wrong with this simple algorithm, since the problem can be fixed during the code-optimization phase. That is, the compiler can deduce that the conversion of 60 from integer to real representation can be done once and for all at compile time, so the `int to real` operation can be eliminated. Besides, `temp3` is used only once, to transmit its value to `id1`. It then becomes safe to substitute `id1` for `temp3`.

- **Code Generation:** Using registers 1 and 2, the translation of the code of our example might become

```

MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
  
```

The first and second operands of each instruction specify a source and destination, respectively. The F in each instruction tells us that instructions deal with floating-point numbers. The # signifies that 60.0 is to be treated as a constant.

### 1.3 Grouping of Phases

**Front End:** The front end consists of those phases, or parts of phases, that depend primarily on the source language and are largely independent of the target machine. These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code. A certain amount of code optimization can be done by the front end as well. The front end also includes the error handling that goes along with each of these phases.

**Back End:** The back end includes those portions of the compiler that depend on the target machine, and generally, these portions do not depend on the source language, just the intermediate language. In the back end, we find aspects of the code optimization phases, and we find code generation, along with the necessary error handling and symbol-table operations.

### 1.4 Passes in a Compiler

A pass is a module in which portions of one or more phases of compiler are combined when a compiler is implemented. A pass reads or scans the instructions constituting the source program or the output produced by its previous pass, makes the necessary transformations specified by its phases, writes the result produced into an intermediate file which may then be read by a subsequent pass. Factors that influence the number of passes are: The structure of the source language and Environment in which compiler will operate.

Mathematical optimality criteria does not affect the number of passes in a compiler. A compiler can explicitly use following types of passes:

1. One-pass compilation technique
2. Two-pass compilation technique
3. Multi-pass compilation technique

#### 1.4.1 Purpose of One-Pass Compiler

A one-pass compiler generates a skeleton of machine instructions the first time it sees a stream of instructions, then appends the machine address for these instructions to a list of instructions to be backpatched once the machine address for it is generated. Backpatching is a technique of merging phases into one pass.

#### 1.4.2 Purpose of Two-Pass Compiler

A two-pass compiler uses its first pass to enter into its symbol table a list of identifiers (as identified in lexical analysis) together with the memory locations to which these identifiers correspond. Then a second pass replaces mnemonic operation codes (mnemonics are keywords as defined by the programming constructs of a language) by their machine-language equivalent and replaced uses of identifiers by their machine addresses or memory locations.

#### Advantages of Bootstrapping

- It is a non-trivial test of the language being compiled.
- By bootstrapping compiler developers only need to know the language compiled.
- Bootstrapping improves the compiler's back-end which improves not only general purpose programs but also the compiler itself.
- It is a comprehensive consistency check as it is able to reproduce its own object code.

## Summary



- Converting a program written in one language into an identical program written in another language is called compilation.
- Phases of compiler:
  1. **Lexical analysis:** Used to generate lexeme (Tokens)
  2. **Syntax analysis:** Used to generate syntax tree.
  3. **Semantics analysis:** Used to check syntax tree semantically correct or not.
  4. **Intermediate code generation:** Used to generate machine independent code.
  5. **Code optimization:** Improve the overall runtime and space of code.
  6. **Code generation:** Generate the code for particular machine.
- The first 3 phases of compiler are called front-end.
- The last 2 phases of compiler are called back-end.
- Pass is a module in which portions of one or more phases of compiler are combined when a compiler is implemented.
- Types of passes:
  1. One-pass compiler, 2. Two-pass compiler, 3. Multi-pass compiler
- In single pass compiler more memory is needed but time to complete compilation is less.
- In multipass compiler memory requirement is less but time to complete compilation is more.

Student's  
Assignments

**Q.1** A C identifier consists of zero or more letters, digits, and underscores, in any order, but may not begin with a digit. Which pattern will match one?

- (a)  $[A - z]^+[0 - 9]^*$
- (b)  $[A - Za - z\_ ]^+[0 - 9]^+$
- (c)  $[a - zA - Z0 - 9\_ ]^+$
- (d) None of the answers are correct

**Q.2** Consider the following statements:

1. A compiler is a translator from one language to another.
2. The output of a compiler needs to be in a low-level language.

Which of the following statements are true?

- (a) Only statement 1 is true
- (b) Only statement 2 is true
- (c) Both (1) and (2) are true
- (d) Neither of them are true

**Q.3** Consider the following statements:

1. There is no restriction on compiler that the input and output language must be different.
2. Interpreter takes code written in a particular programming language and executes it directly one statement (or expression, etc.) at a time.
3. Both interpreters and compilers could exist for a given language.

Which of the following statements are true?

- (a) 1 and 2
- (b) 1, 2 and 3
- (c) 2 and 3
- (d) Neither is true

**Q.4** Consider the following statements:

1. Compiler produce executable binary object file whereas an interpreter produce code, both executable can run many times.
2. Before translation, compiler and interpreter reads all of the input file.

Which of the following statements are true?

- (a) Only statement 1 is true
- (b) Only statement 2 is true
- (c) Both (1) and (2) are true
- (d) Neither of them are true



- Q.5** Consider the following statements:
1. C is a context-free language.
  2. C++ is a context-free language.
- Which of the following statements are true?
- (a) Only statement 1 is true
  - (b) Only statement 2 is true
  - (c) Both (1) and (2) are true
  - (d) Neither of them are true
- Q.6** Consider the following statements:
1. C++ is much harder to parse than C.
  2. The backend of the compiler handles program parsing.
- Which of the following statements are true?
- (a) Only statement 1 is true
  - (b) Only statement 2 is true
  - (c) Both (1) and (2) are true
  - (d) Neither of them are true
- Q.7** Consider the following statements:
1. Both compiler and interpreter perform syntactic and semantics analysis.
  2. Both compiler and interpreter perform optimizations on the input code.
  3. Both compilation and interpretation is an off-line process.
- Which of the following statements are true?
- (a) 1 and 2 only
  - (b) 1 and 3 only
  - (c) 2 and 3 only
  - (d) Neither of them are true
- Q.8** Consider the following statements:
1. Abstract syntax tree made in syntax analysis phase contain cycle.
  2.  $30 = x * 3$  contains lexical error.
- Which of the following statements are true?
- (a) Only statement 1 is true
  - (b) Only statement 2 is true
  - (c) Both (1) and (2) are true
  - (d) Neither of them are true
- Q.9** In a variable declaration, the variable has not previously been declared in the same step.
- (a) Compiler handles it in Scanner stage
  - (b) Compiler handles it in Semantic stage
  - (c) Compiler handles it in parser stage
  - (d) None of these
- Q.10** Consider the following statements:
- $S_1$ : A compiler performs code optimization; a preprocessor does not.
- $S_2$ : A compiler performs full syntactic and semantic analysis; a preprocessor does not.
- Which of the following statements is correct?
- (a)  $S_1$  and  $S_2$  are both true
  - (b)  $S_1$  is true,  $S_2$  is false
  - (c)  $S_1$  is false,  $S_2$  is true
  - (d)  $S_1$  and  $S_2$  are both false
- Q.11** Consider the following issue:
- I. Simplify the phases
  - II. Compiler efficiency is improved
  - III. Compiler works faster
  - IV. Compiler portability is enhanced.
- Which is/are true in context of lexical analysis?
- (a) I, II, III
  - (b) I, III, IV
  - (c) I, II, IV
  - (d) All of these
- Q.12** The task of the lexical analysis phase is
- (a) To parse the source program into the basic elements or token of the language.
  - (b) To build a literal table and an identifier table
  - (c) To build a uniform symbol table
  - (d) All of the above
- Q.13** Assembly code data base is associated with
- (a) Assembly language version of the program which is created by the code.
  - (b) A permanent table of decision rules in the form of patterns for matching with the uniform symbol table to discover syntactic structure.
  - (c) Consists of full or partial list or the token is as they appear in the program. Created by lexical analysis and used for syntax analysis and interpretation.
  - (d) A permanent table which lists all key words and special symbols of the language in symbolic form.
- Q.14** The term "environment" in programming language semantics is said as
- (a) Function that maps a name to value held there
  - (b) Function that maps a storage location to the value held there

- (c) Function that maps a name to a storage location
- (d) None of the above

**Q.15** A linker is given object modules for a set of programs that were compiled separately. What information need to be included in an object module?

- (a) Object code
- (b) Relocation bits
- (c) Name and locations of all external symbols defined in the object module
- (d) Absolute addresses of internal symbols.

**Q.16** Which of the following is used in syntax analysis?

- (a) Finite automata      (b) Push down automata
- (c) Regular grammar    (d) None of these

**Q.17** Identify the type of error (earliest phase) identified during compilation of the following program.

```
#include <stdio.h>
main ( )
{
    int gate, exam, rank;
    gate = exam = rank = 10.3;
    printf("%c", gate);
}
```

- (a) Lexical error      (b) Syntax error
- (c) Semantic error    (d) None of these

**Q.18** Symbol table is not modified during \_\_\_\_\_ phase of a compiler.

- (a) Lexical phase      (b) Semantic phase
- (c) Both (a) and (b)    (d) None of these

**Q.19** Choose the correct sequence of occurrence during compilation process

- (a) Character stream → parse tree → optimized code
- (b) Parse tree → token stream → intermediate code
- (c) SDT tree → parse tree → optimized code
- (d) Parse tree → 3 address code → character stream

**Q.20** Consider the following statements:

$S_1$ : Analysis phase of compiler includes code optimization stage.

$S_2$ : Synthesis phase of compiler is followed by analysis phase.

- (a)  $S_1$  is correct,  $S_2$  is not
- (b)  $S_2$  is correct,  $S_1$  is not
- (c)  $S_1$  and  $S_2$  both are correct
- (d)  $S_1$  and  $S_2$  both are incorrect

**Q.21** Which of the following is not a functionality of C compiler?

- (a) Identifying syntax error
- (b) Identifying tokens
- (c) Linking
- (d) None of these

**Q.22** Consider the following program segment:

```
main ( )
{
    int a, b;
    a = 5 + 8 +;
    printf("%d", a);
    /*&b = 5; */
}
```

The number of tokens present in the above program segment \_\_\_\_\_.

**Q.23** Consider the following statements:

- (i) Type checking is done during all the phases especially in syntax analysis phase.
- (ii) Target code generation phase is machine independent code generation.
- (iii) SDT with only synthesized attribute, always have a order of evaluation.
- (iv) Symbol table is constructed during analysis part of compiler

Which of the above is correct?

- (a) (i) and (ii) only      (b) (iii) and (iv) only
- (c) (ii) and (iv)      (d) (i) and (ii) only

**Q.24** Which of the following is correct about self-relocating program?

- (a) It cannot be made to execute in any area of storage other than the one designated for it at the time of its coding or translation.
- (b) It can itself perform the relocation of its address sensitive portions.
- (c) Consists a program and relevant information for its relocation.
- (d) None of the above



**Q.25** Consider the following statements:

$S_1$  : A lexeme is a sequence of characters in the source program that is matched by pattern for a token.

$S_2$  : The set of string described by a rule is called pattern associated with a token.

Which of the following is true?

- (a) Only  $S_2$
- (b) Only  $S_1$
- (c) Both  $S_1$  and  $S_2$
- (d) Neither  $S_1$  nor  $S_2$

**Q.26** Which of the following is true?

- (a) Symbol table is constructed during the analysis part of compiler i.e., (front end)
- (b) Type checking is done during syntax analysis phase.
- (c) Syntax directed definition with only synthesised attribute, always have a order of evaluation.
- (d) Both (a) and (c)

**Q.27** In a compiler the module that checks the token arrangement against the source code grammar is called \_\_\_\_\_.

- (a) Lexical analyzer
- (b) Syntax analyzer
- (c) Semantic analyzer
- (d) Code optimizer

**Q.28** Match the following errors corresponding to their phase:

**Group A**

- 1. Unbalanced parenthesis
- 2. Appearance of illegal characters
- 3. Undeclared variables

**Group B**

- A. Syntactic error
- B. Semantic error
- C. Lexical error
- (a)  $1 \rightarrow A, 2 \rightarrow C, 3 \rightarrow B$
- (b)  $1 \rightarrow B, 2 \rightarrow C, 3 \rightarrow A$
- (c)  $1 \rightarrow A, 2 \rightarrow B, 3 \rightarrow C$
- (d)  $1 \rightarrow B, 2 \rightarrow C, 3 \rightarrow A$

**Q.29** Type checking is normally done during

- (a) Lexical analysis
- (b) Syntax analysis
- (c) Semantic phase
- (d) Code optimization

**Q.30** Consider the following program:

```
1. main( )
2. {   int x = 10;
3.     if (x < 20;
4.         else
5.         y = 20;
6. }
```

When lexical analyzer scanning the above program, how many lexical errors can be produced?

**Q.31** Match **Group-I** and **Group-II** and select the correct answer using the codes given below the lists:

**Group-I**

- A. Token
- B. Pattern
- C. Lexeme

**Group-II**

- 1. Sequence of characters in the source program that matches the pattern of a token.
- 2. A pair consisting of a token name and an optional attribute value.
- 3. Description of the form that can be accepted.

**Codes:**

	A	B	C
(a)	1	2	3
(b)	2	3	1
(c)	3	1	2
(d)	1	3	2

**Q.32** Consider the following statements:

- I. The module that checks every character of the source text is called symbol table in a compiler.
- II. Keywords of a language are recognized during lexical analysis.
- III. Temporary variables are one of the contents of an activation record.

Which of the above statements is/are correct?

- (a) I and III only
- (b) I only
- (c) II and III only
- (d) I and II only

**Answer Key:**

- 1. (d)    2. (a)    3. (c)    4. (d)    5. (d)
- 6. (a)    7. (d)    8. (d)    9. (b)    10. (a)
- 11. (c)    12. (d)    13. (a)    14. (c)    15. (a, c)
- 16. (b)    17. (c)    18. (d)    19. (a)    20. (d)

21. (c) 22. (24) 23. (b) 24. (b) 25. (c) 19. (a)  
 26. (d) 27. (b) 28. (a) 29. (c) 30. (0)  
 31. (b) 32. (c)



### Student's Assignments

### Explanations

1. (d)

Since identifier may start with zero or more letters, digits and underscores, in any order but should not begin with digit.

Thus,  $[a - zA - Z]^+ [a - zA - Z_0 - 9]^*$

Hence, none of the option matches so option (d) is answer.

5. (d)

Both C and C++ are context sensitive language because context can make difference.

For **example**:

**int A** and **int a** are treated as two different variables.

9. (b)

Any declared variable is stored in a symbol table which is checked in the semantic phase of the compiler.

10. (a)

A preprocessor is a programs that processes its input data to produce output that is used as input to another program. While compiler performs full syntactic, semantic and code optimization etc.

12. (d)

Lexical analyzer performs the following tasks:  
 Reads the source program, scans the input characters, group them into lexemes and produces the token.

It also updates symbol table.

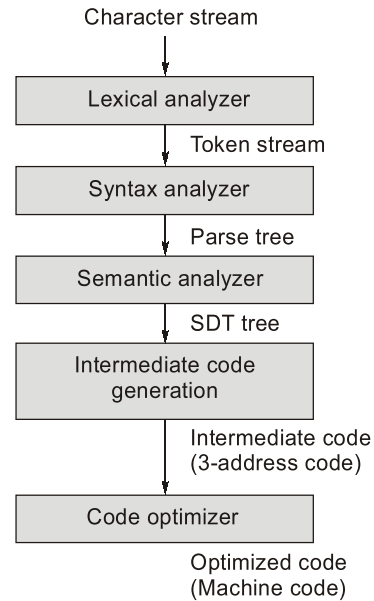
Hence option (d) is true.

16. (b)

Push down automata is used in syntax analysis phase.

18. (d)

Symbol table is modified by both lexical and semantic phase of a compiler.



20. (d)

Analysis phase {lexical analysis, syntax analysis, semantic analysis} is followed by synthesis phase {intermediate code generation, code optimizer, machine code generation}.

21. (c)

Linking is done by a linker after compilation process.

Compiler can identify token, generates compilation error which can be lexical, syntax or semantic.

22. (24)

```

1   2   3
main ( )
4
{
    5   6   7   8   9
    int a , b ;
    10  11  12  13  14  15  16
    a = 5 + 8 + ;
    17   18   19   20  21  22  23
    printf ( " %d" , a ) ;
    /* & b = 5; */
24
}
```

Total 24 tokens.