Compiler Design

For

Computer Science
&
Information Technology

By

www.thegateacademy.com

✆080-40611000

Syllabus for Compiler Design

Lexical Analysis, Parsing, Syntax-Directed Translation, Runtime Environments, Intermediate Code Generation.

Previous Year GATE Papers and Analysis

GATE Papers with answer key

thegateacademy.com/gate-papers

Subject wise Weightage Analysis

thegateacademy.com/gate-syllabus

Contents

	Chapters	Page No.
#1.	**Introduction to Compilers**	**1 – 15**
	• Compilers	1 – 2
	• Analysis of the Source Program	3 – 5
	• The Phases of a Compiler	5 – 12
	• Lexical Analyzer	12 – 14
	• Specification of Tokens	14 – 15
#2.	**Parsing**	**16 – 72**
	• Syntax Analysis	16
	• The Role of the Parser	16 – 17
	• Context-Free Grammar	18 – 21
	• Writing a Grammar	22 – 26
	• Top-Down Parsing	26 – 37
	• Bottom-Up Parsing	37 – 40
	• Operator-Precedence Parsing	40 – 49
	• LR Parsers	50 – 65
	• Parser Generators	65 – 72
#3.	**Syntax Directed Translation**	**73 – 100**
	• Syntax Directed Translation	73 – 74
	• Syntax Directed Definitions (SD-Definitions)	74 – 79
	• Construction of Syntax Trees	79 – 84
	• Bottom-Up Evaluation of S-Attributed Definition	85 – 86
	• L-Attributed Definitions	86 – 90
	• Top-Down Translation	90 – 94
	• Bottom-Up Evaluation of Inherited Attributes	94 – 97
	• Run Time Environment	97 – 100
#4.	**Intermediate Code Generation**	**101 – 122**
	• Intermediate Code Generation	101
	• Intermediate Language	101 – 107
	• Issues in the Design of a Code Generator	107 – 110
	• Target Machine	110 – 113
	• Code Optimization	113 – 117
	• The Principal Sources of Optimization	117 – 122
	Reference Books	**123**

www.thegateacademy.com

"The will to do springs from the knowledge that we can do."
.... James Allen

CHAPTER 1

Introduction to Compilers

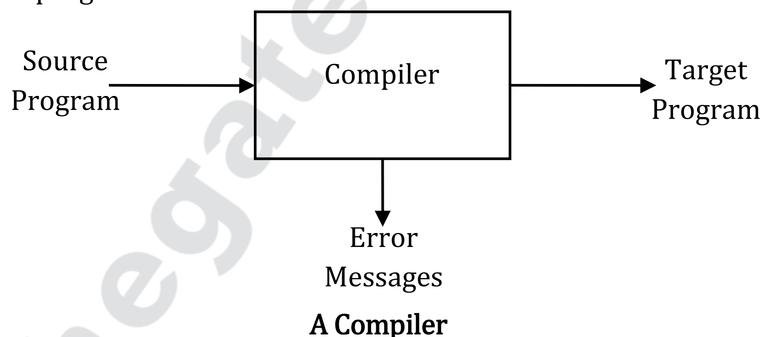
Learning Objectives

After reading this chapter, you will know:

1. Compiler
2. Analysis of the Source Program
3. The Phases of a Compiler
4. Lexical Analyzer
5. Specification of Tokens

Compiler

A compiler is a program that reads a program written in one language – the source language – and translates it into an equivalent program in another language – the target language (see Fig. Shows below. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.



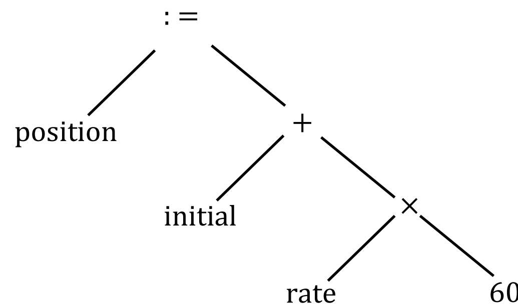
A Compiler

Compilers are sometimes classified as single pass multi-pass, load and go, debugging or optimising 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. By understanding these tasks, we can construct compilers for a variety of source languages and target machines using the same basic techniques.

The Analysis-Synthesis Model of Compilation

There are two parts of compilation: Analysis and Synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. Out of the two parts, synthesis requires the most specialized techniques.

During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called as tree. Often, a special kind of tree called as Syntax tree is used, in which each node represents an operation and the children of node represent the argument of the operation. For example, a syntax tree for an assignment statement is shown in below.

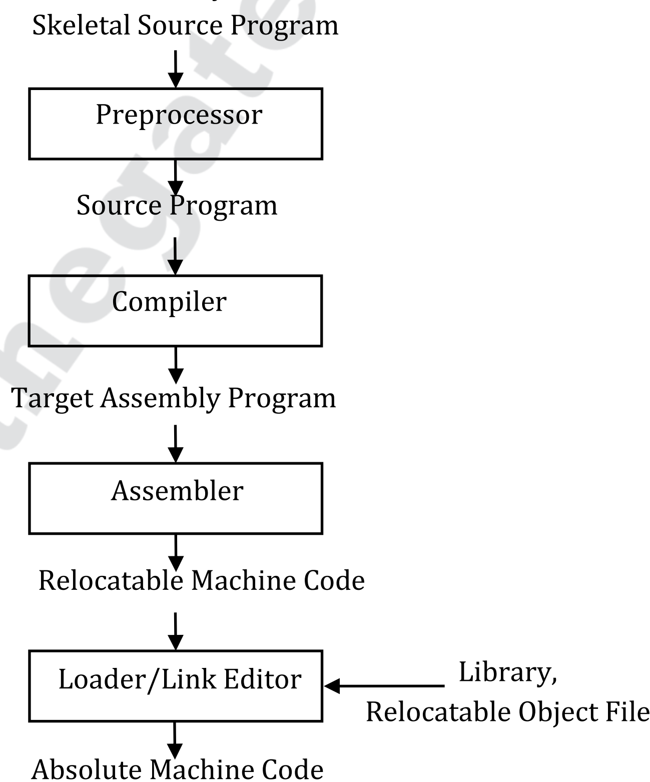


Syntax Tree for Position: = Initial + Rate ∗ 60.

The Context of a Compiler

In addition to a compiler, several other programs may be required to create an executable target program. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a distinct program, called as **Preprocessor**. The preprocessor may also expand shorthands, called macros, into source language statements.

Following below figure shows a typical "compilation." The target program created by the compiler may require further processing before it can be run. The compiler in below figure creates assembly code that is translated by an assembler into machine code and then linked together with some library routines into the code that actually runs on the machine.



A Language-Processing System

Analysis of the Source Program

1. Linear or Lexical analysis, in which 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.
2. Hierarchical or Syntax analysis, in which characters or tokens are grouped hierarchically into nested collections with collective meanings.
3. Semantic analysis, in which certain checks are performed to ensure that the components of a program fit together meaningfully.

Lexical Analysis

A **token** is a string of characters, categorized according to the rules as a symbol (e.g. IDENTIFIER, NUMBER, COMMA, etc.). The process of forming tokens from an input stream of characters is called **tokenization** and the lexer categorizes them according to symbol type. A token can look like anything that is useful for processing an input text stream or text file.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parenthesis as tokens, but does nothing to ensure that each '(' is matched with a ')'.

In a compiler, linear analysis is called lexical analysis or scanning. For example, in lexical analysis the characters in the assignment statement

Position: $=$ initial $+$ rate $*$ 60

would be grouped into the following tokens:

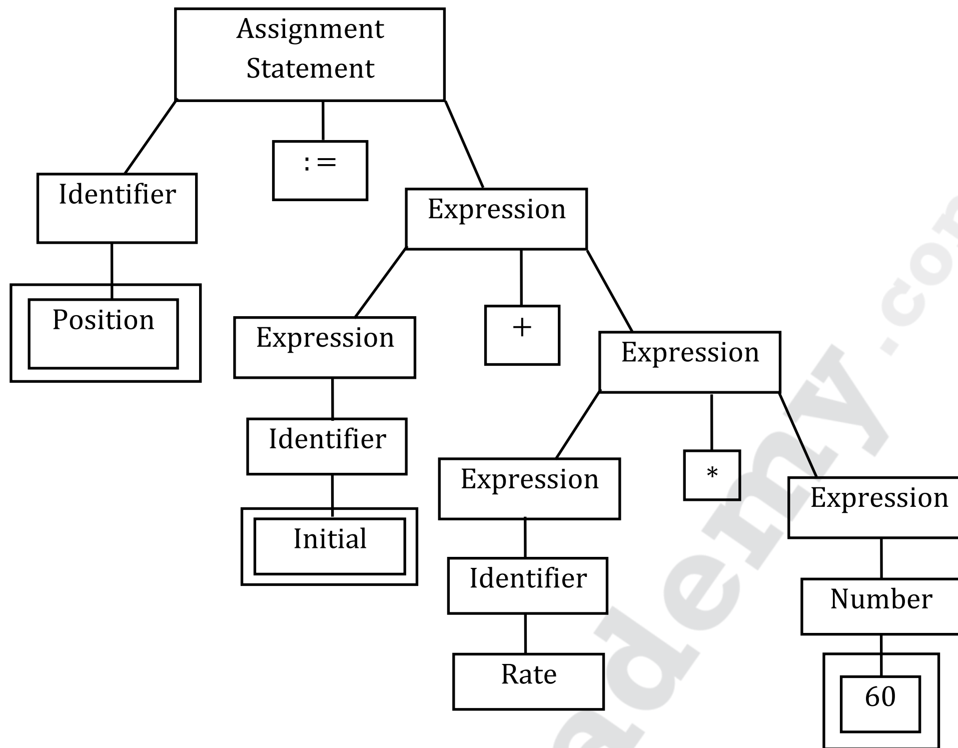
1. The identifier **position**.
2. The assignment symbol **: =**
3. The identifier **initial**.
4. The plus sign **+**
5. The identifier **rate**.
6. The multiplication sign
7. The number **60**

   The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

Syntax Analysis

Hierarchical analysis is called parsing or syntax 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 such as the one shown in figure below.

www.thegateacademy.com



Parse Tree for Position: = Initial + Rate ∗ 60

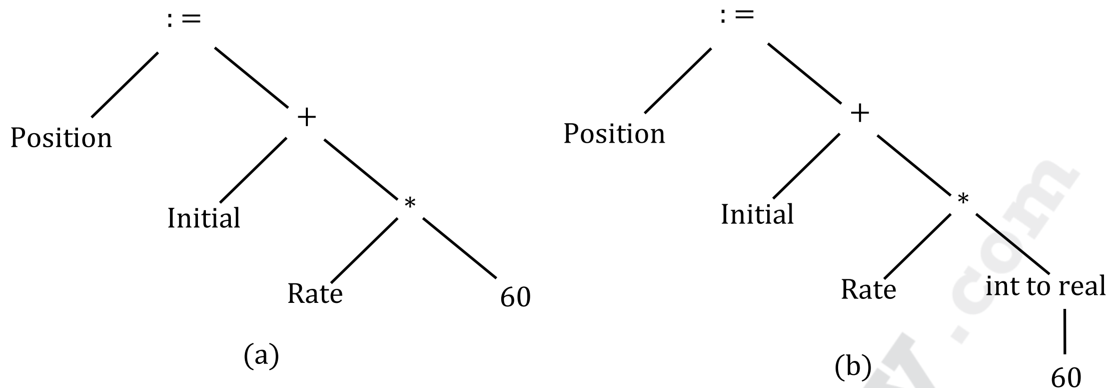
In the expression initial + rate ∗ 60, the phrase rate ∗ 60 is a logical unit because the usual conventions of arithmetic expressions tell us that multiplication is performed before addition. Because the expression initial + rate is followed by a ′∗′ it is not grouped into a single phrase by itself in Fig. above.

The hierarchical structure of a program is usually expressed by recursive rules. For example, we might have the following rules as part of the definition of expressions:

1. Any identifier is an expression.
2. Any number is an expression.
3. If expression1 and expression2 are expressions, then so are
   expression1 + expression2
   expression1 ∗ expression2
   (expression1)

   Rules (1) and (2) are non-recursive basic rules, while (3) defines expressions in terms of operators applied to other expressions. Thus, by rule (1), initial and rate are expressions. By rule (2), 60 is an expression, while by rule (3), we can first infer that rate∗60 is an expression and finally that initial + rate∗60 is an expression.

www.thegateacademy.com



Semantic Analysis Inserts a Conversion from Integer to Real

The parse tree for position describes the syntactic structure of the input. A more common internal representation of this syntactic structure is given by the syntax in Fig. above (a). A syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes, and the operands of an operator are the children of the node for that operator.

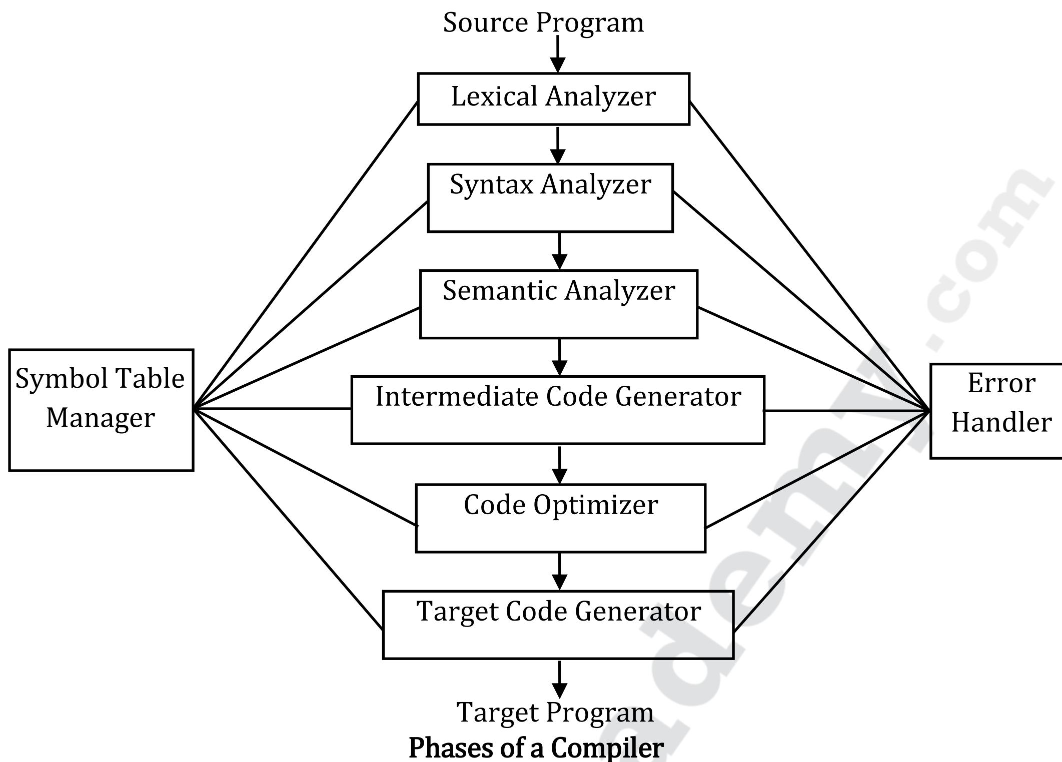
Semantic Analysis

The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.
An important component of semantic analysis is type checking.
Here the compiler checks that each operator has operands that are permitted by the source language specification. For example, many programming language definitions require a compiler to report an error every time a real number is used to index an array. However, the language specification may permit some operand corrections, for example, when binary arithmetic operator is applied to an integer and real. In this case, the compiler may need to convert the integer to a real.

The Phases of a Compiler

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another. A typical decomposition of a compiler is shown in Fig. below. In practice, some of the phases may be grouped together, and the intermediate representations between the grouped phases need not be explicitly constructed.

www.thegateacademy.com



Phases of a Compiler

The first three forming the bulk of the analysis portion of a compiler, were introduced in the last section. Two other activities, symbol-table management and error handling, are shown interacting with the six phases of compilation lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and code generation. Informally, we shall also call the symbol-table manager and the error handler "phases."

The 6 phases divided into 2 Groups

1. **Front End**: Depends on stream of tokens and parse tree
2. **Back End**: Dependent on Target, Independent of source code

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.

Symbol table is a Data Structure in a Compiler used for Managing information about variables & their attributes.

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. A compiler that stops when it finds the first error is not as helpful as it could be.

The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase.

The 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 -------------- (1.1)

Lexical Analyzer

1. The lexical analysis phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an identifier, a keyword (if, while, etc.), a punctuation character, or a multi-character operator like : = .The character sequence forming a token is called the lexeme for the token.

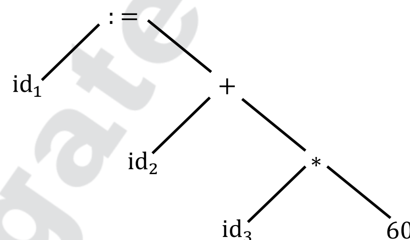
   Certain tokens will be augmented by a "lexical value". For example, when an identifier like rate is found, the lexical analyzer not only generates a token, say id, but also enters the lexeme rate into the symbol table, if it is not already there. The lexical value associated with this occurrence of id points to the symbol-table entry for rate.

   In this section, we shall use id_1, id_2, and id_3 for position, initial, and rate, respectively, to emphasize that the internal representation of an identifier is different from the character sequence forming the identifier. The representation of assignment statement (1.1) after lexical analysis is therefore suggested by:

   $id_1 = id_2 + id_3 * 60$ -------------- (1.2)

Syntax Analysis Phase

1. The syntax Analysis Phase: The syntax analysis phase imposes a hierarchical structure on the token stream, shown below



2. 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 integer into a real