23CS(AI&DS)7301

AUTOMATA AND COMPILER DESIGN

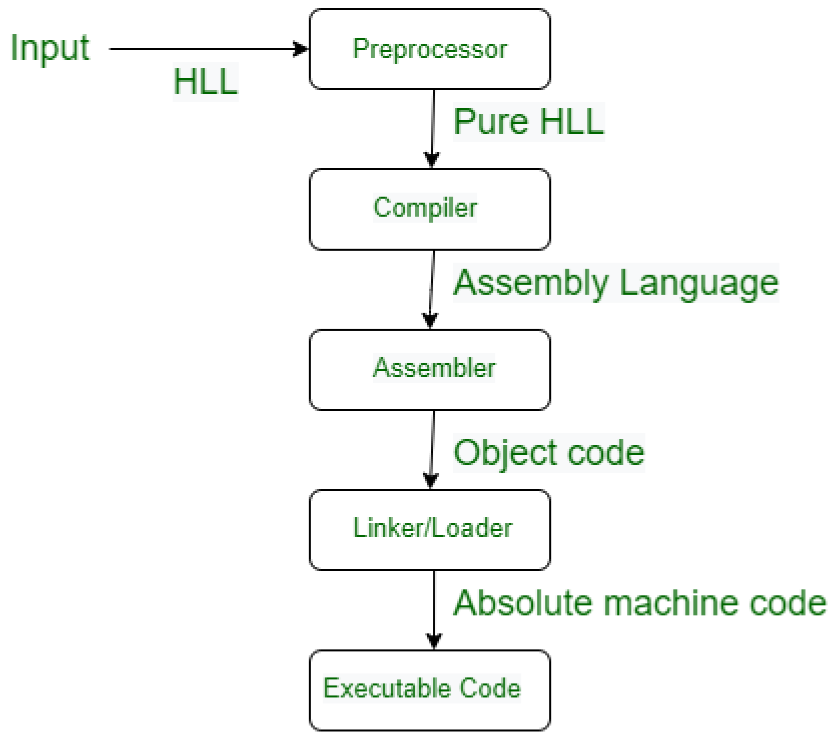
UNIT - 1

Structure of Compiler: Lexical Analysis, Syntax analysis, Intermediate Code generation, Code Optimization, Code generation, Bookkeeping, Error handling.

Formal Language and Regular Expressions: Languages, Definition Languages, regular expressions, Finite Automata DFA, NFA. Conversion of regular expression to NFA, NFA to DFA. Applications of Finite Automata to lexical analysis, lex tools. Implementation of Lezical Analyzer (LTC)

Language Processing System

The programs that are written in high-level languages (e.g., C, C++, Python, Java, etc.) are passed into a sequence of devices and operating system (OS) components to generate the desired machine code that can be understandable by machine, known as a **Language Processing System.** These systems help in converting **human-readable source code** into **machine-executable code** or other forms.
The different components of the Language processing system are given in the below diagram.



Language Processing System

Pre-processor

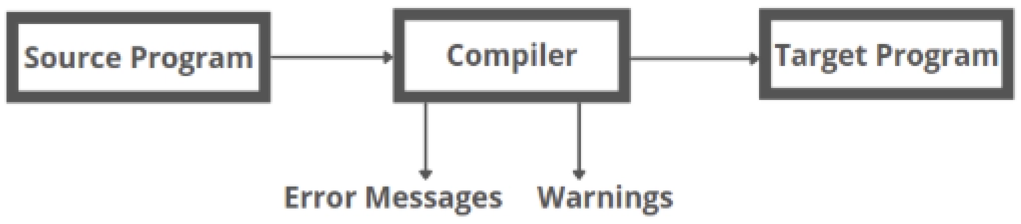
A source program can be broken down into modules, each of which is saved in its own file. A pre-processor is responsible for gathering the source program. All header files are included in the preprocessor, which also checks for a macro's presence. It accepts source code and outputs updated source code.

Macro: A macro is a chunk of code that has a name assigned to it. An interpreter or compiler replaces the name with the contents of the macro whenever it is used. Macros are used to automate the frequency with which sequences are run or to provide more powerful abstraction.

Example: #include<iostream> here # is a pre-processor directive. It copies all library functions of iostream (standard input-output) file into a program.

Compiler

A compiler is computer software that transforms source code written in one computer language (the source language) into another computer language (the target language).
The compiler is used for programs that translate source code from a high-level programming language (e.g., C++, Python, Java, etc.) to a lower-level language (e.g., assembly language or machine code).



Assembler

An **assembler** converts the assembly language code into machine code. The output of an assembler is known as an object file, which comprises a combination of machine instructions and the data needed to store them in memory.

Linker

A **linker** is a computer program that combines and links many object files to create one executable file. All of these files could have been created by different assemblers. A linker's primary function is to search and find referred modules in a program and establish the memory address where these codes will be loaded, resulting in absolute references in program instructions.

Loader

The **loader** is an operating system component that loads and executes executable files into the main memory. It calculates a program's size (instructions and data) and allocates RAM. It starts execution by initializing specific registers.

Executable Code

It's a low-level, machine-specific code that the machine can interpret. After the linker and loader have completed their tasks, the object code is eventually transformed into executable code.

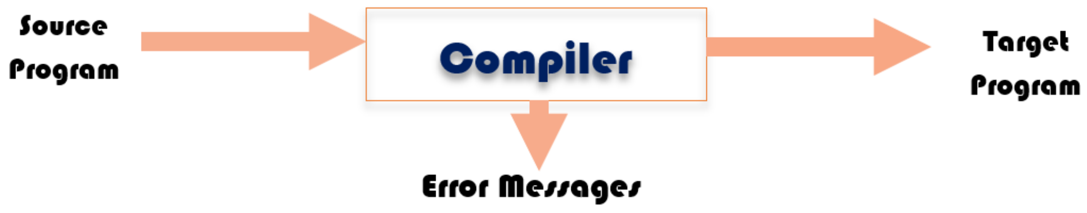
Types of Language Processors

The language processors can be any of the following three types:

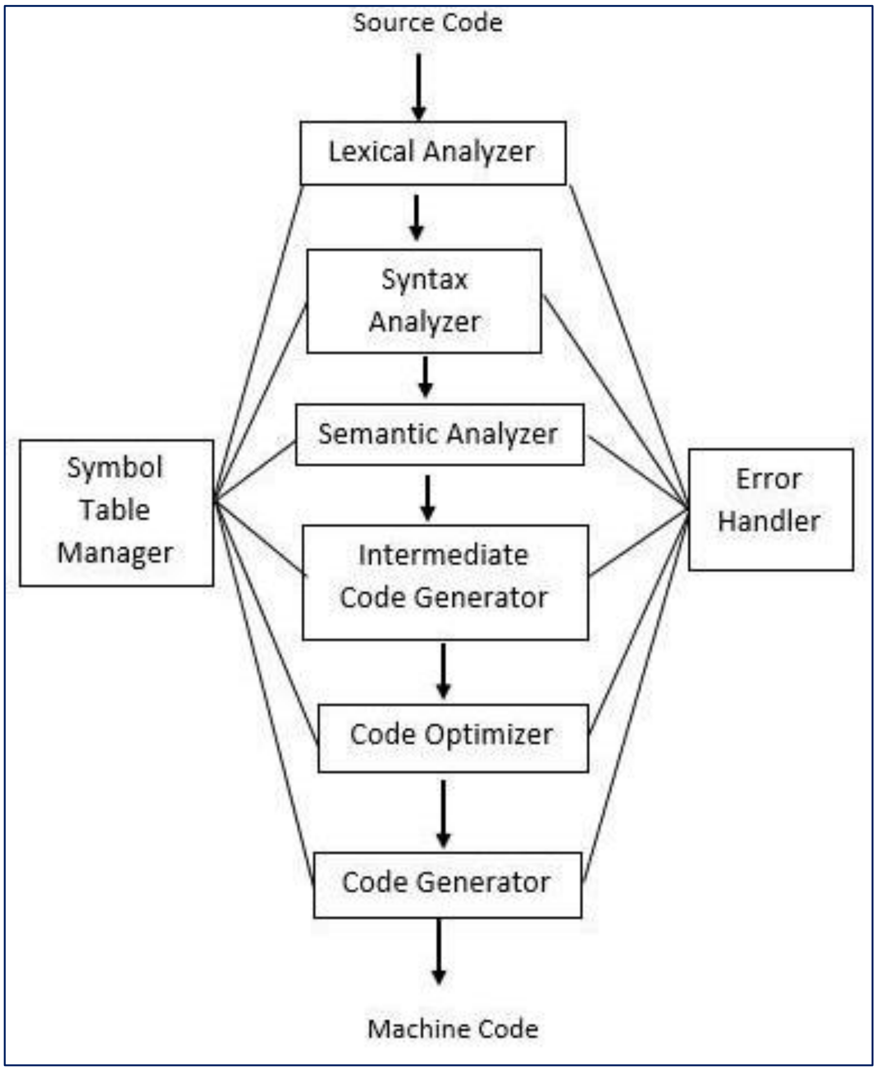
Aspect	Compiler	Interpreter	Assembler
Translation Process	Translates entire program	Translates and executes line by line	Translates assembly to machine code
Output	Generates standalone executable	No separate executable generated	Generates machine code
Execution	Separate execution step required	Directly executes source code	No execution, only translation
Feedback	Feedback after compilation	Immediate feedback during execution	No feedback during translation
Efficiency	Generally results in faster execution	Slightly slower due to on-the-fly translation	Efficient low-level control
Debugging	Requires recompilation for debugging	Easier debugging through step-by-step execution	Limited debugging capabilities
Examples	C, C++, Java	Python, Ruby, JavaScript	x86 Assembly, MIPS Assembly

Compiler

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



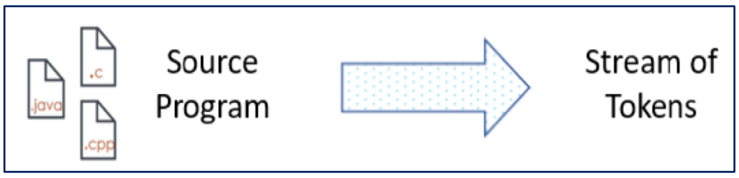
Phases of Compiler



Lexical Analysis

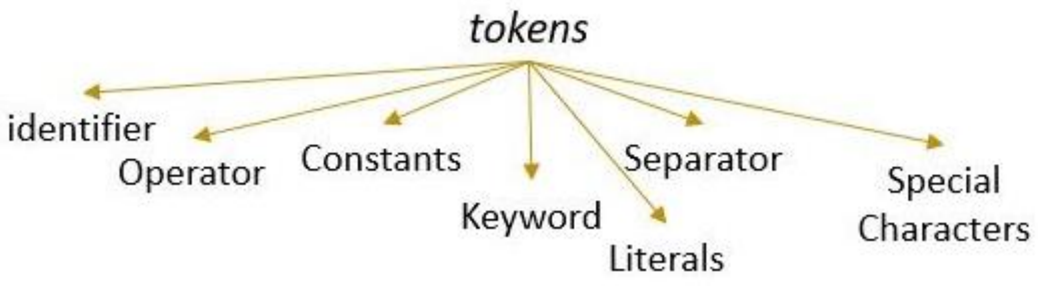
In a compiler, linear analysis is called lexical analysis or scanning:

- It is the first phase of a compiler
- Lexical Analyzer is also known as scanner
- Reads the characters in the source program from left to right and groups the characters into stream of Tokens.
- Such as Identifier, Keyword, Punctuation character, Operator.
- Pattern: Rule for a set of strings in the input for which a token is produced as output.
- A Lexeme is a sequence of characters in the source code that is matched by the Pattern for a Token.
- A token is the smallest individual meaningful part of the source code.



For example, consider the following assignment statement:

position = initial + rate * 60



Keyword(if,while, etc) special character(: # $ % etc), separator(( ) { } [ ] ; ,)

In lexical analysis, the characters would be grouped into the following tokens:

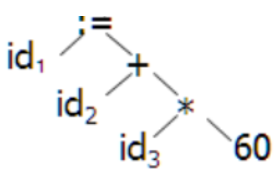
1. The identifier *position*.
2. The assignment operator(ASSIGN) =.
3. The identifier *initial*.
4. The operator +.
5. The identifier *rate*.
6. The Operator * .
7. The number or literal or constant 60.

The blanks separating the characters of these tokens would be eliminated during this phase.

Syntax Analysis

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

Example of parse tree:



Functions of Syntax Analysis:

- Tokenization: Breaks down the code into basic elements or tokens (e.g., keywords, variables, operators).
- Error Checking: Ensures that the code written adheres to the defined syntax rules of the programming language.
- Structure Organization: Converts code into a structured format, such as a parse tree or an abstract syntax tree, which represents the hierarchical relationship of code elements.

Limitations of Syntax Analysis

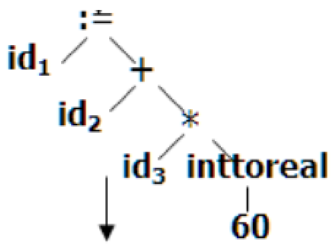
1. It cannot determine if the token is a valid token or not.
2. It cannot determine whether a token is used before or not.
3. It cannot determine whether the operation performed on tokens is valid or not.
4. It cannot tell whether the token was initialized or not.

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. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.

For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.



Functions of Semantic Analysis:

- Type Checking
- Scope resolution
- Function correctness
- Identifying Unused Variables and Unreachable Code
- Handling Overloading and Overriding

Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as 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.

Example:

$$\begin{aligned}
\mathbf{temp_1} &:= \mathbf{inttoreal(60)} \\
\mathbf{temp_2} &:= \mathbf{id_3 * temp_1} \\
\mathbf{temp_3} &:= \mathbf{id_2 + temp_2} \\
\mathbf{id_1} &:= \mathbf{temp_3}
\end{aligned}$$

The following three are commonly used intermediate code representations:

1. Postfix Notation
2. Three Address Code
3. Syntax Tree

Functions of Intermediate code generation:

- Portability: Intermediate code is platform-independent, making it easier to generate machine code for multiple architectures without rewriting the entire compiler.
- Optimization: Intermediate code provides an abstract, simpler representation, allowing the compiler to perform optimizations before generating machine-specific code.
- Reusability: The intermediate code can be reused for different target machines, simplifying compiler design and maintenance.
- Error Detection: It helps in early detection of errors, as transformations in intermediate code can reveal inconsistencies before reaching machine code generation.

Limitations of Intermediate code generation:

- Increased Compilation Time: Generating and processing intermediate code adds extra steps, slightly lengthening the overall compilation time.
- Memory Overhead: Storing and manipulating the intermediate code requires additional memory resources during the compilation process.

Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

Example:

$$\begin{aligned} \mathbf{temp_1} &:= \mathbf{id_3 * 60.0} \\ \mathbf{id_1} &:= \mathbf{id_2 + temp_1} \end{aligned}$$

There are various techniques used by most of the optimizing compilers, such as:

1. Common sub-expression elimination
2. Dead Code elimination
3. Constant folding
4. Copy propagation
5. Induction variable elimination
6. Code motion
7. Strength Reduction

Advantages of Code Optimization are:

- Improved Performance: Optimized code runs faster and consumes fewer system resources.
- Reduced Execution Time: Optimization reduces execution time, enhancing overall system responsiveness.
- Resource Efficiency: Optimized code consumes less memory and CPU cycles, leading to better resource utilization.
- Better User Experience: Faster response times and smoother execution improve user satisfaction.

Code Generation

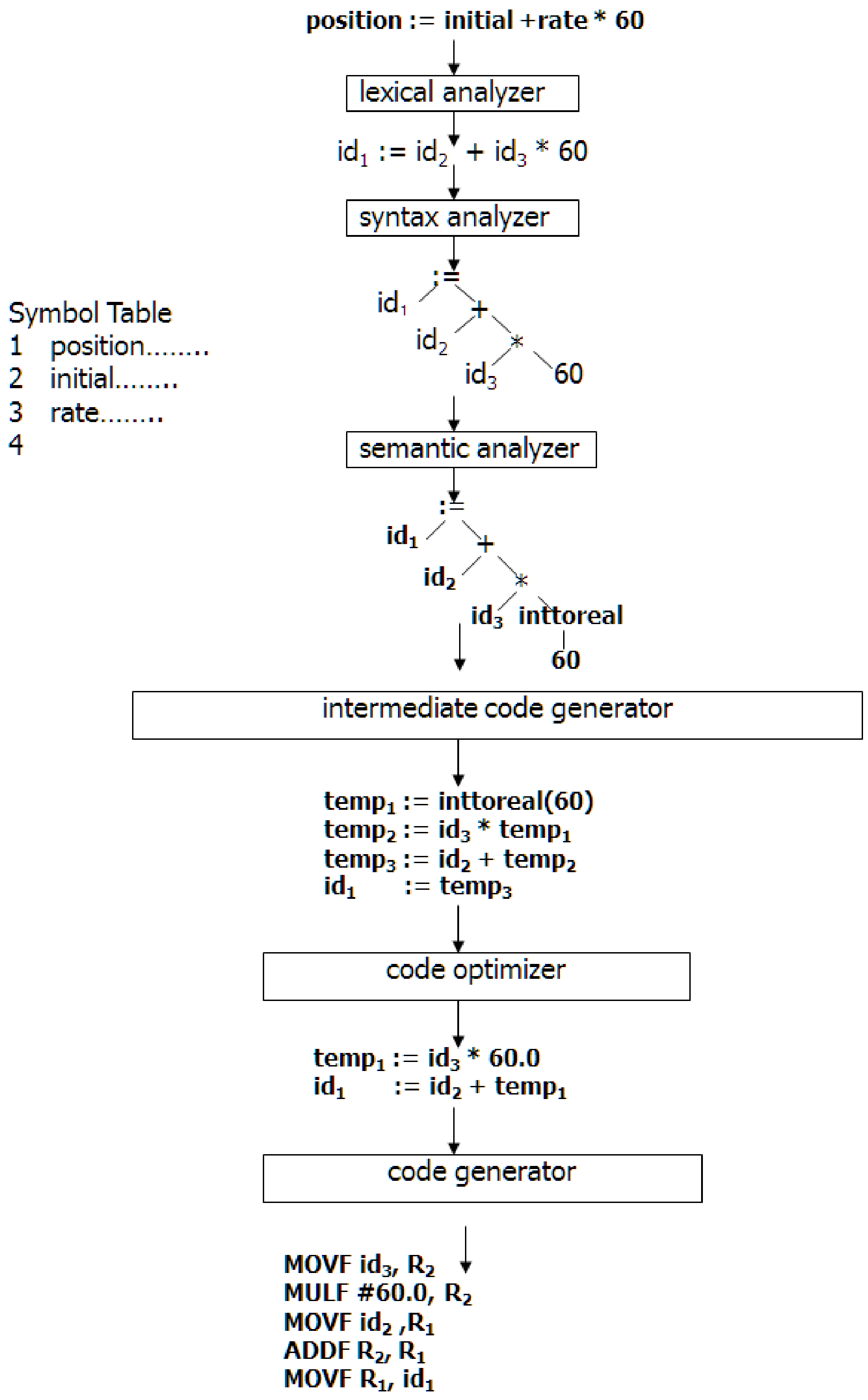
The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated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

Example:

MOVF id_3, R_2
MULF #60.0, R_2
MOVF id_2 ,R_1
ADDF R_2, R_1
MOVF R_1, id_1

Advantage of Code Generation:

Performing various operations on registers is efficient as registers are faster than cache memory. This feature is effectively used by compilers; however, registers are not available in large amount and they are costly. Therefore, we should try to use minimum number of registers to incur overall low cost.



Solve: 1) z = x * y   2) a = b*(c-d)*10

Bookkeeping or Symbol-Table Management:

- An essential function of a compiler is to record the identifiers used in the source program and collect its information.
- A symbol table is a data structure containing a record for each identifier with fields for attributes.(such as, its type, its scope, if procedure names then the number and type of arguments etc.,)
- The data structure allows finding the record for each identifier and store or retrieving data from that record quickly.

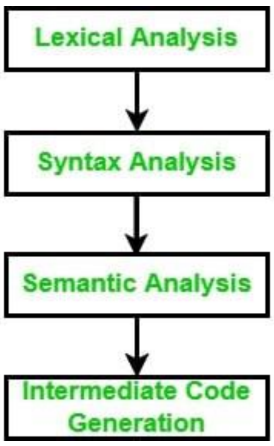
Error Handling and Reporting

- Each phase can encounter errors. After detecting an error, a phase must deal that error, so that compilation can proceed, allowing further errors to be detected.
- Lexical phase can detect error where the characters remaining in the input do not form any token.
- The syntax and semantic phases handle large fraction of errors. The stream of tokens violates the syntax rules are determined by syntax phase.
- During semantic, the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved. e.g. if we try to add two identifiers, one is an array name and the other a procedure name.

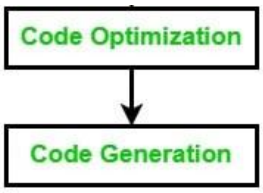
Phases in Compiler:

Generally, phases of compiler are divided into two parts:

1. Front End phases: The front end consists of those phases or parts of phases that are source language-dependent and target machine, independents. These generally consist of lexical analysis, semantic analysis, syntactic analysis, symbol table creation, and intermediate code generation. A little part of code optimization can also be included in the front-end part. The front-end part also includes the error handling that goes along with each of the phases.



2. Back End phases: The portions of compilers that depend on the target machine and do not depend on the source language are included in the back end. In the back end, code generation and necessary features of code optimization phases, along with error handling and symbol table operations are also included.



Passes in Compiler:

A pass is a component where parts of one or more phases of the compiler are combined when a compiler is implemented. A pass reads or scans the instructions of the source program or the output produced by the previous pass, which makes necessary transformation specified by its phases.

There are generally two types of passes

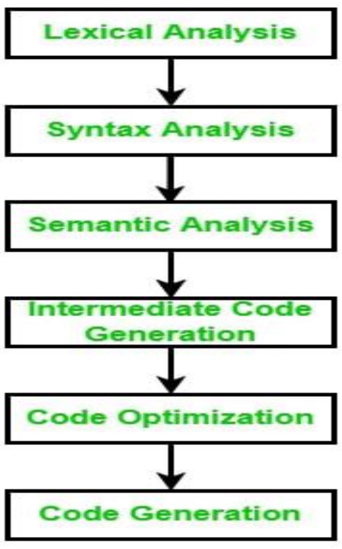
1. One-pass
2. Two-pass

Grouping

Several phases are grouped together to a pass so that it can read the input file and write an output file.

1. One-Pass - In One-pass all the phases are grouped into one phase. The six phases are included here in one pass.
2. Two-Pass - In Two-pass the phases are divided into two parts i.e. Analysis or Front End part of the compiler and the synthesis part or back end part of the compiler.

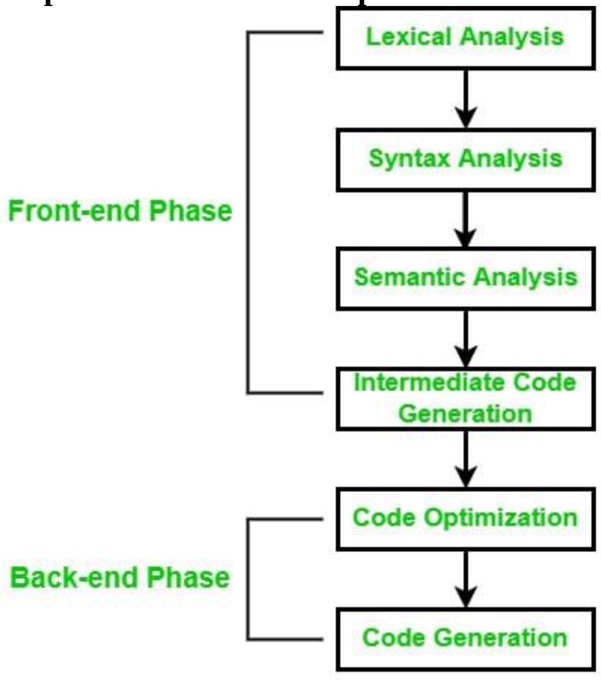
Purpose of One Pass Compiler



One- Pass Compiler

A one-pass compiler generates a structure of machine instructions as it looks like a stream of instructions and then sums up with machine address for these guidelines to a rundown of directions to be backpatched once the machine address for it is generated. It is used to pass the program for one time. Whenever the line source is handled, it is checked and the token is removed.

Purpose of Two-Pass Compiler



Two- Pass Compiler

A two-pass compiler utilizes its first pass to go into its symbol table a rundown of identifiers along with the memory areas to which these identifiers relate. Then, at that point, a second pass replaces mnemonic operation codes by their machine language equivalent and replaced uses of identifiers by their machine address. In the second pass, the compiler can read the result document delivered by the first pass, assemble the syntactic tree and deliver the syntactical examination. The result of this stage is a record that contains the syntactical tree.

Errors Encountered in the Analysis Phase of a Compiler:

1. Lexical Errors (Scanner Phase)

- Invalid characters or tokens not defined in the language.
- **Examples**:
  - Misspelled keywords (`fi` instead of `if`)
  - Use of illegal characters (e.g., `@` in a language where it's not allowed)
  - Unterminated strings: `"hello`

2. Syntax Errors (Parser Phase)

- Violations of the grammatical structure (rules) of the language.
- **Examples**:
  - Missing semicolons: `int x = 5`
  - Mismatched parentheses/brackets: `if (x > 0 { ... }`
  - Incorrect nesting of constructs

3. Semantic Errors (Semantic Analysis Phase)

- Code is grammatically correct but makes no sense semantically.
- **Examples**:
  - Type mismatches: `int x = "hello";`
  - Undeclared variables: `y = 10;` (when `y` is not declared)
  - Function called with wrong number/type of arguments

4. Symbol Table Errors

- Issues related to the declaration and scope of identifiers.
- **Examples**:
  - Redeclaration of a variable in the same scope
  - Use of a variable outside its scope

5. Type Checking Errors

- Inconsistent or illegal use of types.
- **Examples**:
  - Performing arithmetic on incompatible types (e.g., `string + int`)
  - Assigning `float` to `int` without explicit casting (in strongly typed languages)