UNIT-1
Introduction to Compiler

A compiler is a translator software program that takes its input in the form of program written in one particular programming language and produce the output in the form of program in another language.

Features of Compiler:

→ A compiler is a translator the converts the high-level language into the machine language.

→ High-level language is written by a developer and machine language can be understood by the processor.

→ Compiler is used to shao errors to the programmer.

→ Compiler executes a program into two parts: In first, source program is compiled into object program, In second, object program is translated into target program.

$$\begin{tikzcd}
{} \arrow[r, "\text{Source program}"] & \boxed{\text{Compiler}} \arrow[r, "\text{Object program}"] & {}
\end{tikzcd}$$

$$\begin{tikzcd}
{} \arrow[r, "\text{Object program}"] & \boxed{\text{Assembler}} \arrow[r, "\text{Target program}"] & {}
\end{tikzcd}$$

⊛Compiler Structure:

A compiler operates in phases. A phase is a logically inter-related operation that takes source program in one representation and produces output in another representation. There are two phases of compilation:

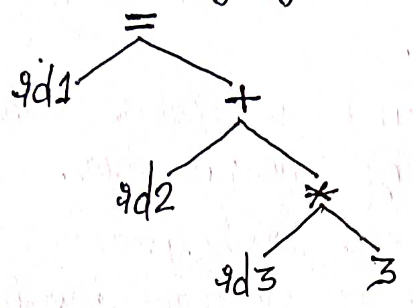
1) Analysis phase: In analysis part, an intermediate representation is created from the given source program. This part is called front end of the compiler. This part consists of mainly four phases:

i) Lexical Analysis: Lexical analysis or scanning is the process where the source program is read from left-to-right and grouped into tokens. Tokens are sequences of characters with a collective meaning. In any programming language tokens may be constants, operators, reserved Words etc. The Lexical Analyzer takes a source program as input, and produces a stream of tokens as output.

Example:
Input string: c=a+b*3
Tokens: id1= id2+id3*3

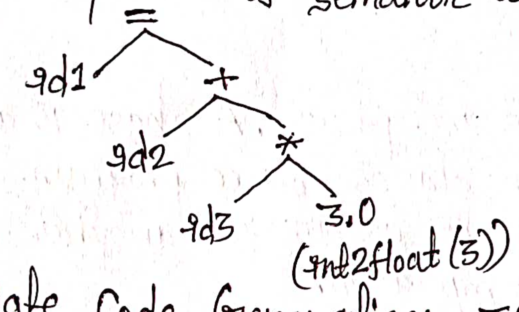
ii) Syntax Analysis: In this phase, the syntax analyzer takes the token produced by lexical analyzer as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the token is syntatically correct or not, according to the rules that define the syntax of the source language.

Example:



iii) Semantic Analysis: In this phase, the semantic analyzer checks the source program for semantic errors and collects the type information for the code generation. Semantic analyzer checks whether they form a sensible set of instructions in the programming language or not. Type-checking is an important part of semantic analyzer.

Example:



iv) Intermediate Code Generation: If the program syntatically and semantically correct then intermediate code generator generates a simple machine independent intermediate language. The intermediate code should be generated in such a way that it can easily be translated into the target machine code.

Example:

```
t1 = 3.0;
t2 = id3 * t1;
t3 = id2 + t2;
id1 = t3;
```

2) Synthesis phase: In synthesis part, the equivalent target program is created from intermediate representation of the program created by analysis part. This part is also called back end of the compiler. This part consists of mainly two phases:

i) Code Optimization: It is used to improve the intermediate code so that the output of the program could run faster and takes less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution without wasting resources.

Example:
```
t2 = id3*3.0;
id1 = id2+t2;
```

ii) Code Generation: Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language.

Example:
```
MOV R1, id3
MUL R1, #3.0
MOV R2, id2
ADD R1, R2
MOV id1, R1.
```

⊛. Symbol Table: Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phase of compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its type, its position in storage, and any other relevant information.

⊛. Error Handling: Whenever an error is encountered during the compilation of the source program, an error handler is invoked. Error handler generates a suitable error reporting message regarding the error encountered. Errors can be encountered at any phase as:

Lexical analysis phase: due to misspelled tokens, unrecognized characters etc.
Syntax analysis phase: due to syntatic violation of language.
Intermediate code generation: due to incompatibility of operands type for operator.
Code optimization phase: due to some unreachable statements.

Block Diagram:

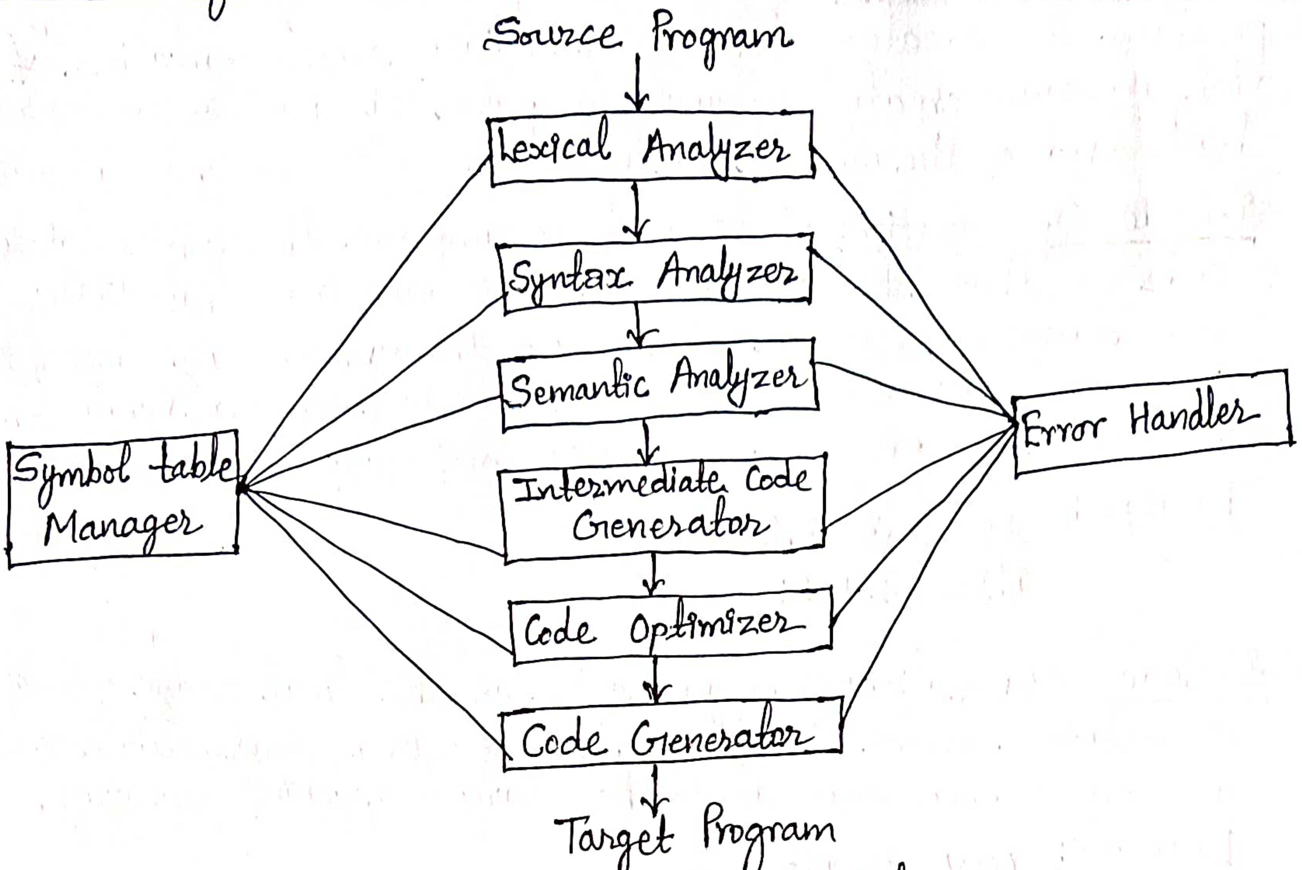


Fig: Phases of a Compiler

⊛. Compiler vs Interpreter:

Compiler	Interpreter
i) Compiler is a translator which takes input i.e, high-level language, and produces an output of low-level language.	i) An Interpreter is a program that translates a programming language into a comprehensible language.
ii) Compiler scans the whole program in one go.	ii) Interpreter translates one statement at a time.
iii) Errors are shown at the end together as it scans full code in one go.	iii) Errors are shown line by line as it scans code one line at a time.
iv) It does not require source code for later execution.	iv) It requires source code for later execution.
v) It converts source code into object code.	v) It does not convert source code into object code instead it scans it line by line.

⊛. Simple or One pass Compiler vs. Multi-pass Compiler:

(Simple) One pass compiler	Multi-pass compiler
i) In a one pass compiler all the phases are combined into one pass.	i) In multi-pass compiler different phases of compiler are grouped into multiple phases.
ii) Here intermediate representation of source program is not created.	ii) Here intermediate representation of source program is created.
iii) It is faster than multi-pass compiler.	iii) It is slightly slower than one pass compiler.
iv) It is also called narrow compiler.	iv) It is also called wide compiler.
v) Pascal's compiler is an example of one pass compiler.	v) C++ compiler is an example of multi-pass compiler.

⊛. Preprocessor: A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, agumentation, file inclusion etc.

Functions:

i) Macro processing: A preprocessor may allow a user to define macros that are short hands for longer constructs.

ii) File inclusion: A preprocessor may include header files into the program text.

iii) Rational preprocessor: These preprocessors augment older languages with more modern flow-of-control and data structuring facilities.

iv) Language Extensions: These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro.

⊛. **Macros:** A macro stands for macroinstruction is a programmable pattern which translates a certain sequence of input into a present sequence of output. Macros can make tasks less repetitive by representing a complicated sequence of keystrokes, mouse movements, commands, or other types of input.

Features of macro processor:

→ It represents a group of commonly used statements in the source programming language.

→ Using macro instructions programmer can leave the mechanical details to be handled by macro processor.

→ Macro processor designs is not directly related to the computer architecture on which it runs.

→ Macro processor involves definition, invocation and expansion.