Introduction to Compiler:

1.1 INTRODUCTION OF LANGUAGE PROCESSING SYSTEM

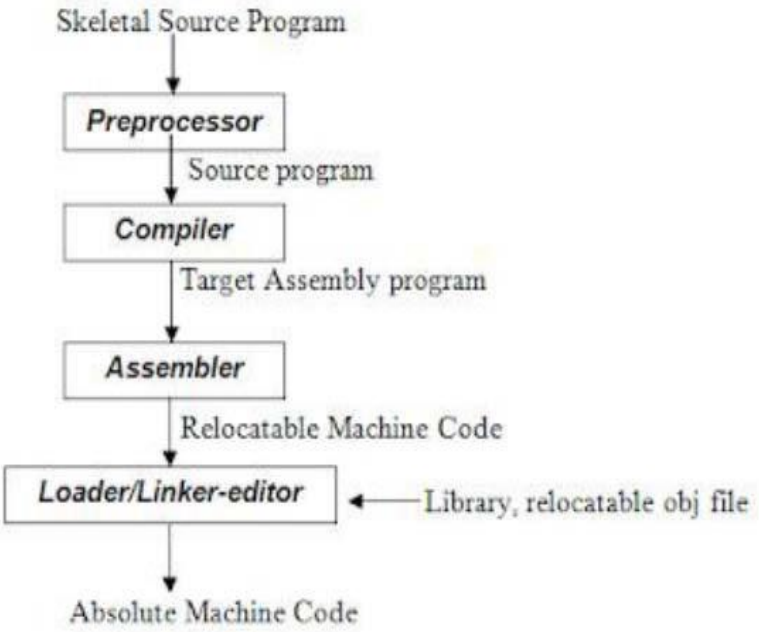

Fig 1.1: Language Processing System

Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

1. *Macro processing:* A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A preprocessor may include header files into the program text.
3. *Rational preprocessor:* these preprocessors augment older languages with more modern flow-ofcontrol and data structuring facilities.
4. *Language Extensions:* These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

COMPILER

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.

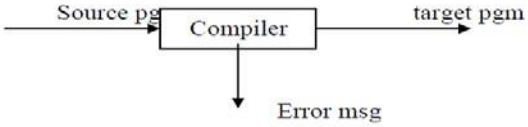

Fig 1.2: Structure of Compiler

Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.

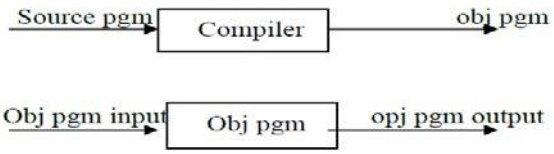

Fig 1.3: Execution process of source program in Compiler

ASSEMBLER

Programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program(in assembly language), the output is a machine language translation (object program).

INTERPRETER

An interpreter is a program that appears to execute a source program as if it were machine language.

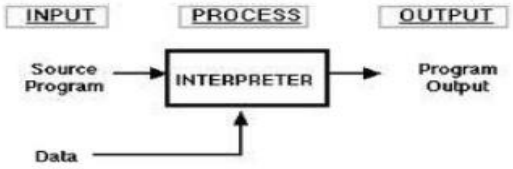

Fig1.4: Execution in Interpreter

Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

1. Lexical analysis
2. Synatx analysis
3. Semantic analysis
4. Direct Execution

Advantages:

Modification of user program can be easily made and implemented as execution proceeds.
Type of object that denotes a various may change dynamically.
Debugging a program and finding errors is simplified task for a program used for interpretation. The interpreter for the language makes it machine independent.

Disadvantages:

The execution of the program is *slower*.
Memory consumption is more.

LOADER AND LINK-EDITOR:

The assembler produces relocatable machine code as its output.

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The LINKER resolves external memory addresses, where

thee code in one file may refer to a location in another file. The LOADER then puts together all of the executable object files into memory for execution.

1.2 TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of the HLL specification would be detected and reported to the programmers. Important role of translator are:

1 Translating the HLL program input into an equivalent Machine Language program.
2 Providing diagnostic messages wherever the programmer violates specification of the HLL.

1.3 LIST OF COMPILERS

1. Ada compilers
2 .ALGOL compilers
3 .BASIC compilers
4 .C# compilers
5 .C compilers
6 .C++ compilers
7 .COBOL compilers
8 .Common Lisp compilers
9. ECMAScript interpreters
10. Fortran compilers
11 .Java compilers
12. Pascal compilers
13. PL/I compilers
14. Python compilers
15. Smalltalk compilers

1.4 STRUCTURE OF THE COMPILER DESIGN

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called **'phases'**.

Lexical Analysis:-

LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automic units called **tokens.**

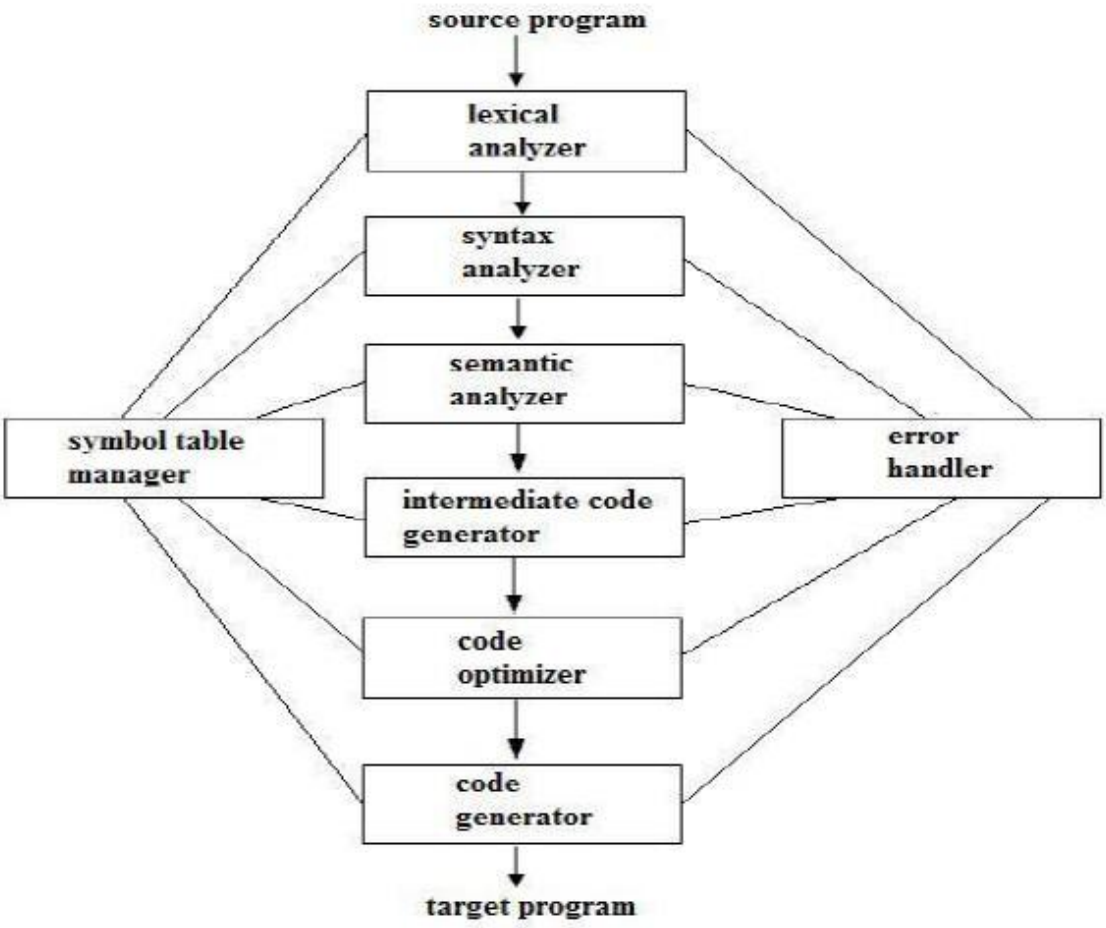

Fig 1.5: Phases of Compiler

Syntax Analysis:-
The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis.

The parser uses the tokens produced by lexical analyser to create a tree like intermediate representation that depicts the grammatical structure of token stream,

Syntax analysis is aided by using techniques based on formal grammar of the programming language.

A typical representation is a SYNTAX TREE in which each interior node represents an operation and the children of the node represent the argument of the operation.

Semantic Analysis: Semantic analyser used 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 the syntax tree or the symbol table for subsequent use during intermediate code generation.

Type checking and *type conversion* are other part of semantic analysis.

Intermediate Code Generations:-
An intermediate representation (machine independent)of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

Intermediate representation should have two important properties: It should be easy to produce and it should be easy to translate into target machine.

Code Optimization :-
This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-
The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Table Management (or) Book-keeping:- This is the portion to **keep the names** used by the program and records essential information about each. The data structure used to record this information called a 'Symbol Table'.

Error Handlers:-
It is invoked when a flaw or error in the source program is detected.

MORE ILLUSTRATION OF ALL THE ABOVE PHASES OF COMPILER:

The output of LA is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as **expression.** Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

The parser has two functions. It checks if the tokens from lexical analyzer, occur in pattern that are permitted by the specification for the source language. It also imposes on tokens a tree-like structure that is used by the subsequent phases of the compiler.

Example, if a program contains the expression **A+/B** after lexical analysis this expression might appear to the syntax analyzer as the token sequence **id+/id.** On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formulations rule of an expression. Syntax analysis is to make explicit the hierarchical structure of the incoming token stream by **identifying which parts of the token stream should be grouped**.

Intermediate Code Generation:-
The intermediate code generation uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instruction with one operator and a small number of operands. The output of the syntax analyzer is some representation of a parse tree. the intermediate code

generation phase transforms this parse tree into an intermediate language representation of the source program.

Code Optimization

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space. Its output is another intermediate code program that does the some job as the original, but in a way that saves time and / or spaces.

a. Local Optimization:-

There are local transformations that can be applied to a program to make an improvement. For example,

If **A** > **B** goto **L2**
Goto L3
L2:

This can be replaced by a single statement

If **A < B** goto **L3**

Another important local optimization is the elimination of common sub-expressions

A := B + C + D E := B + C + F

Might be evaluated as

T1 := B + C A := T1 + D E := T1 + F

Take this advantage of the common sub-expressions **B + C.**

b. Loop Optimization:-

Another important source of optimization concerns about **increasing the speed of loops**. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point, in the program just before the loop is entered.

Code generator :-

Code Generator produces the object code by deciding on the memory locations for data, selecting code to access each datum and selecting the registers in which each computation is to be done. Many computers have only a few high speed registers in which computations can be performed quickly. A good code generator would attempt to utilize registers as efficiently as possible.

Table Management OR Book-keeping :-

A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers. The data structure used to record this information is called as Symbol Table.

Error Handing :-

One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg. Both of the table-management and error-Handling routines interact with all phases of the compiler.

Example:

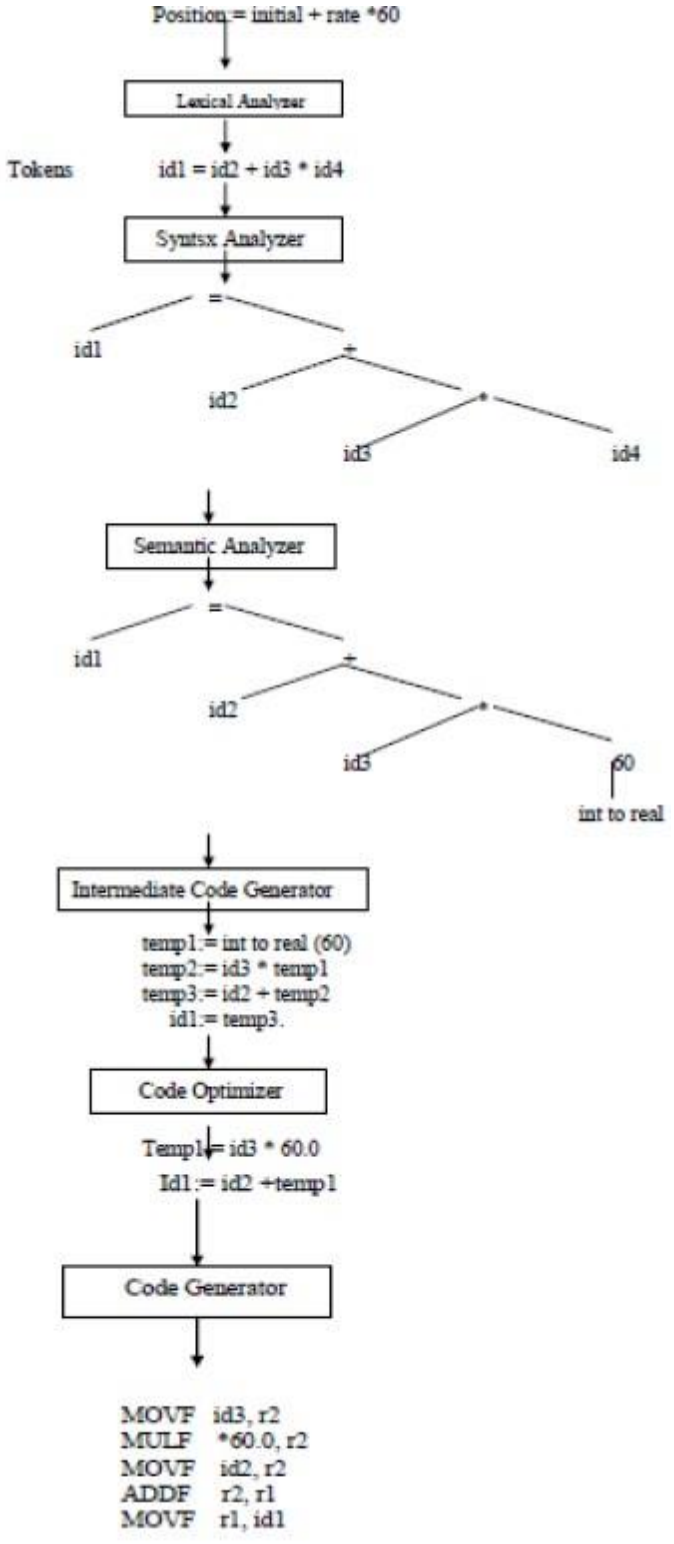

Fig 1.6: Compilation Process of a source code through phases