SISAJOTI Mohan.

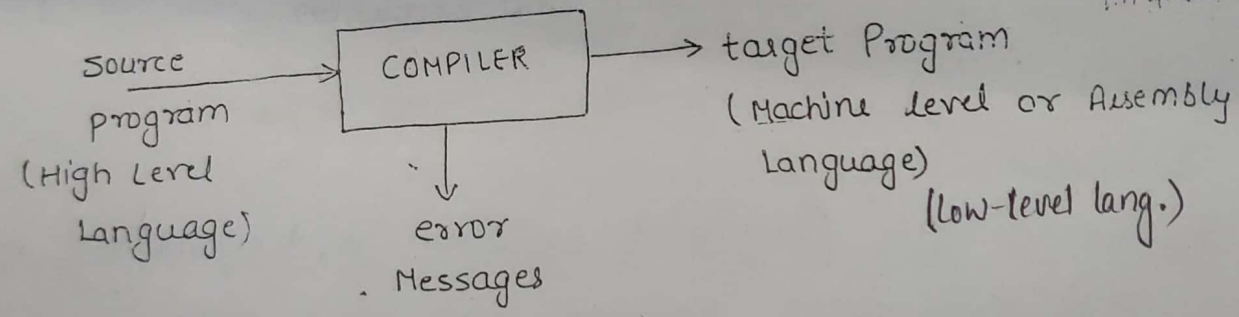
(2)

Translator are also program which can translate prog. written into source language to target language

Compilers and Translators

A translator is a program that takes as input a program written in one programming language (the source language) and produces as output a program in another language (the object or target language)

If a source language is a high-level language such as FORTAN, PL/I, or COBOL, and the object language, then is a low-level language such as machine or assembly language such a translator is called a compiler.



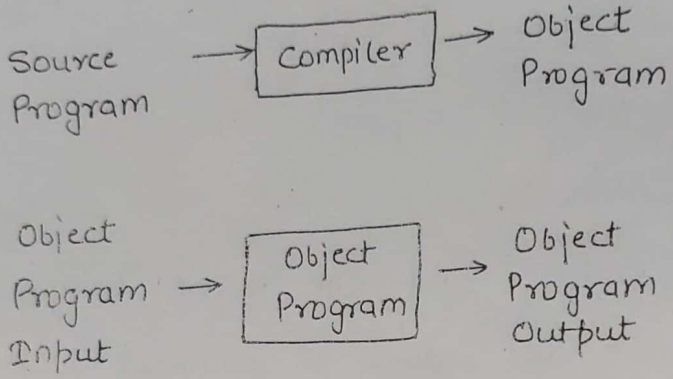
The first compiler FORTAN was developed in 1957 by Backus. Today compilers can be built with much less effort.

Compilers are sometimes ~~called~~ classified as: single pass, multi-pass, load and go, debugging or optimizing depending on how they have been constructed.

single pass
multiple pass
load & go
debugg
optimizing

Executing a program written in a high-level language is basically a two-step process. The source program

must first be compiled, that is translated into the object program. Then the resulting program i.e. object program is loaded into memory and executed.



Compilation and Execution.

Interpreter

Certain other translators transform a programming language into a simplified language, called intermediate code, which can be directly executed using a program called an interpreter.

Interpreters are often smaller than compilers. and facilitate the implementation of complex programming language constructs.

The main disadvantage of interpreters is that the execution time of an interpreter is that the execution time of an inter-preted program is usually slower than that of a corresponding compiled object program

HLL → [compiler] → Assembly lang → [Assembler] → m/c lang.

Assembler

If a source language is assembly language and the target language is machine language, then the translator is called an assembler.

The term preprocessor is sometime used for translators that take programs in one high-level language into equivalent program in another high-level-language.

Why do we Need Translators?

Since it is difficult to write programs in machine language so we make use of high-level language. But the computer can only understand the language of 0's and 1's or machine language, so we need translators to convert program written in high level language to low-level language to make the machine understand it.

Symbolic Assembly language

The most immediate step away from machine language is symbolic assembly language. In this language, a programmer uses mnemonic names for bothes operation codes & data address. E.g. ADD X, Y in assembly language, but in machine code, this will something like 0110 001110 010101
Add X Y

utes can't execute a program written in assembly. i.e. That program has to be first translated to machine language, which the computer can understand. The program that perform this translation is the assembler.

Macros

Many assembly languages provides a "macro" facility whereby a macro statement will translate into a sequence of assembly languages statements. Thus, macro facility is a text replacement capability. and these statements are translated into machine code.

e.g.

```
MACRO    ADD2   X,Y
         LOAD   Y
         ADD    X
         STORE  Y
ENDMACRO
```

first statement gives the name ADD2 to the macro and defines two dummy aguments, known as formal parameters, X and Y. The next three statements define the macro i.e. they give its translation.

High Level Languages

A High-level language makes the programming task simples, but it also introduce some problems. like we need a program to translate the high-level language into a language that machine can understand. Thus,

A compiler is used for this translation. But A compiles is a more complex program to write than an assembler. High-level language allows a programmer to express algorithms in more natual notation.

The Structure of a Compiler

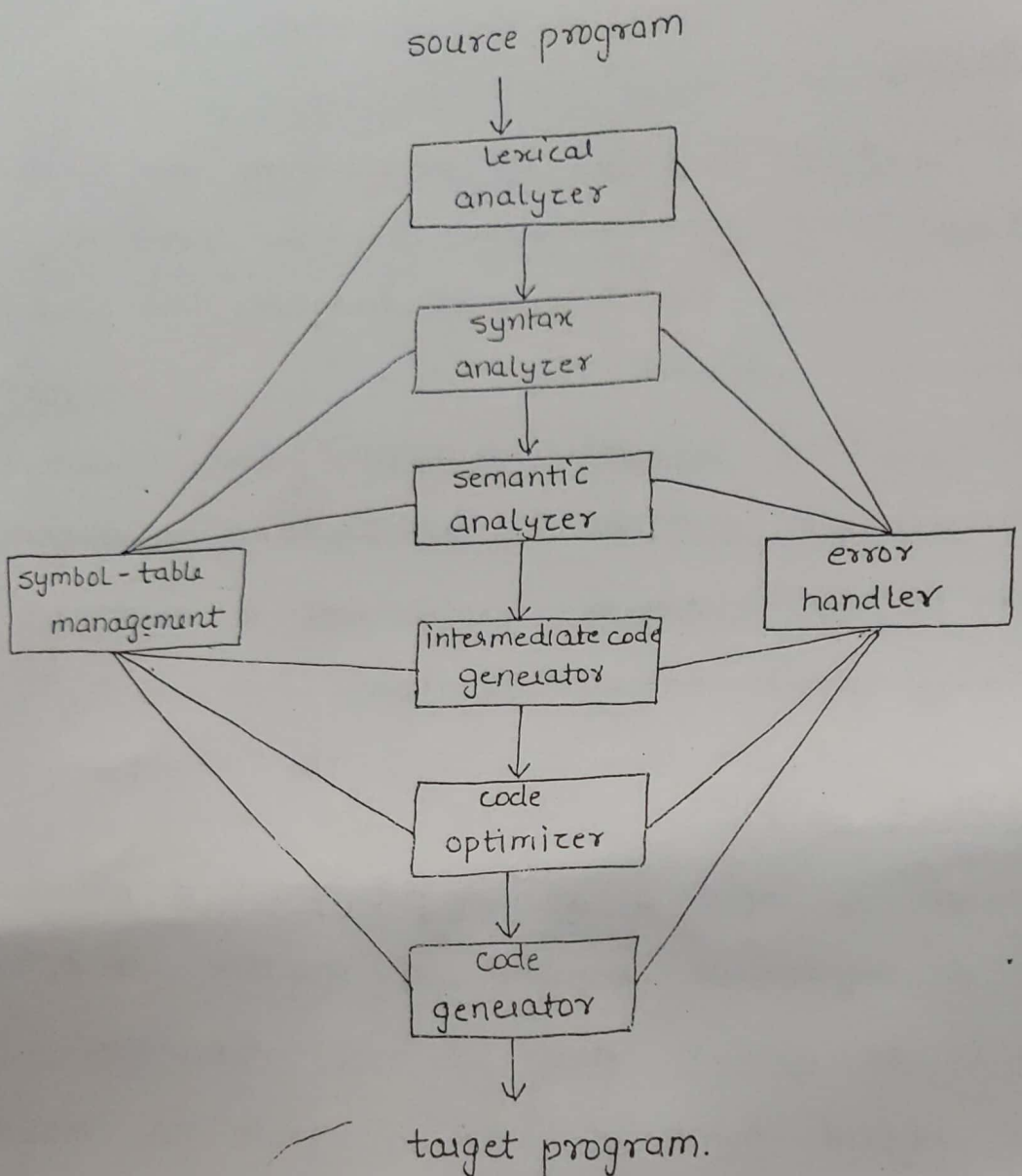
A compiles takes as input a source program and produce as output an equivalent sequence of machine instructions.

This process is so complex that is not reasonable, either from logical point of view or from an implementation point of view (if we consider compilation process is in one single step).

for this reason, it is customary to partition the compilation process into a series of subprocess called phases as shown in fig.

Phase

A phase is a logically cohesive operation that takes as input one representation of the source program and produce as output in another representation.



Phases Of a Compiler

1. Lexical Analyzer –

First phase, called the lexical analysis, or scanner, linear analysis seprates characters of the source language into groups are called tokens.

Eg The usual tokens are keywords, such as DO or IF, identifiers, such as X or NUM, operator symbols such as <= or +, and punctuation symbols such as parentheses or commas.

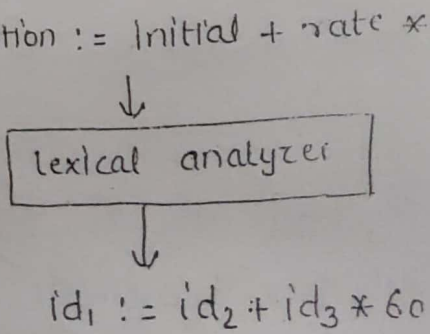
The output of the lexical analyzer is a stream of tokens, which is passed to the next phase, the syntax analyzer, or parser.

for eg. For 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.



Lexical Phase

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

2. Syntax analyzer

The syntax analyzer groups tokens together into syntactic structures.

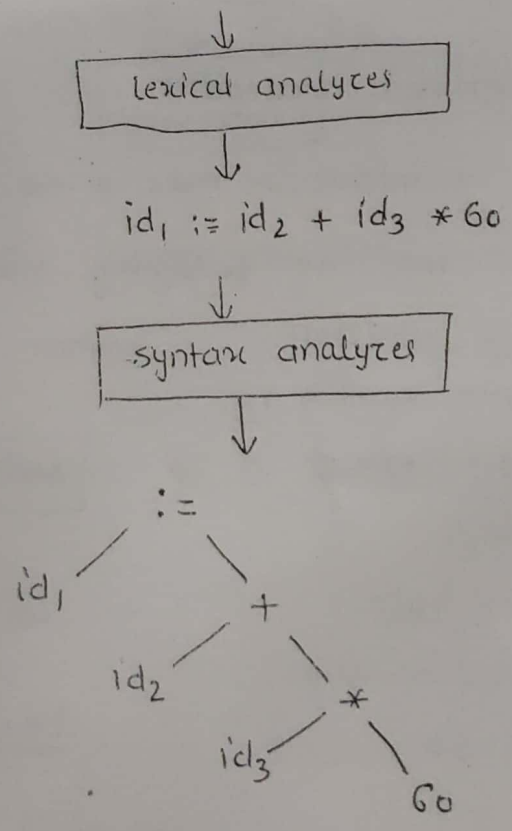
This phase is also called as a Hiesarchical analysis, or passing or syntax analysis. It involves grouping the tokens of the source program into grammatical phases that are used by the compiler to synthesize output.

~~The grammatical phrases that are used by the compiler~~

The grammatical phrases of the source program are represented by a parse tree.

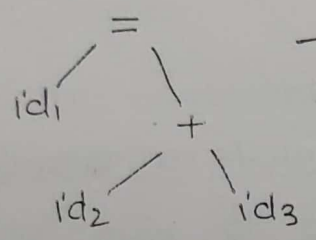
for e.g.

position := initial + rate * 60



Syntax analysis

Another Example, the three tokens representing A+B might be grouped into a syntactic structure called an expression. Expressions might further be combined to form statements. Often the syntactic structure can be regarded as a tree whose leaves are the tokens.



→ Syntax tree
This is also 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

Context-free grammers are a formalization of recursive rules that can be used to guide syntactic analysis

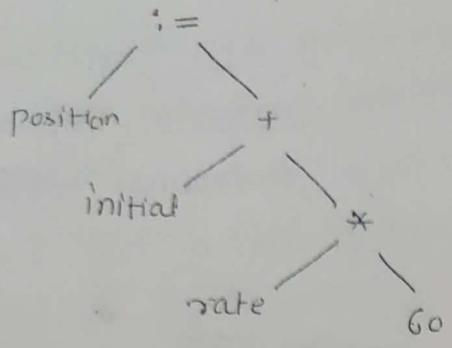
8. ~~Intermediate Code Generation~~

3. Semantic Analysis

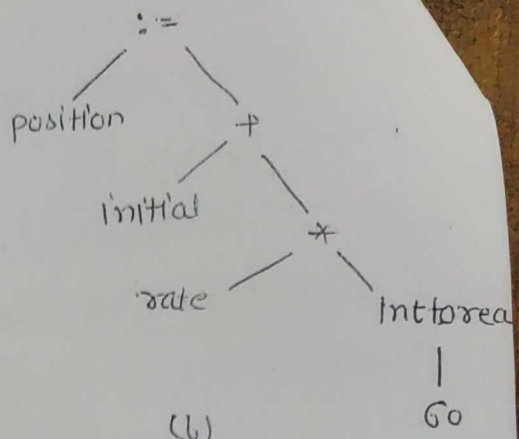
The semantic analysis phase checks the source program for ① semantic errors and gathers ② type information for subsequent code-generation phase. It uses ③ the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements

The another feature of semantic analysis is the ④ type checking. for e.g. many programming language definitions require a compiler to report an error every time a real no. is used to index an array.

e.g. is shown on next page.



(a)



(b)

Semantic analysis inserts a conversion from int. to real

Type Inside a machine, the bit pattern representing an int. is generally different from the bit pattern for a real

Suppose, that all identifiers in above fig. have been declared to be reals and that 60 by itself is assumed to be an int. Type checking of above fig. reveals that * is applied to a real, rate and an integer 60.

The general approach is to convert the int. to real. This can be achieved by creating extra node for the operator inttoreal.

4. Intermediate Code Generation :

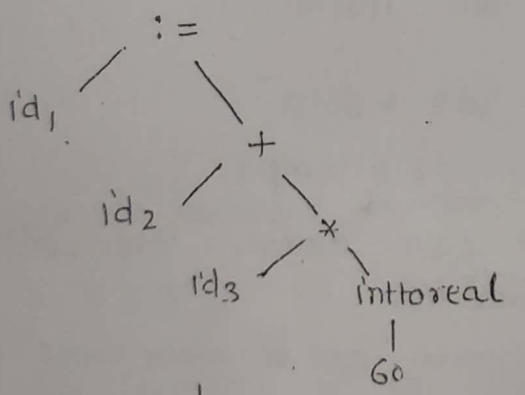
After syntax and semantic analysis, some compilers generate an intermediate representation of the source program. The intermediate code generation phase transform the parse tree into an intermediate-language representation of the source program.

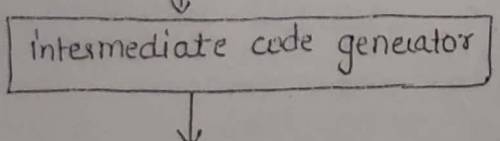
This intermediate representation should have two important properties :-

i) It should be easy to produce and

2) It should be easy to translate into target program.

The intermediate code can have a variety of forms e.g. a three address code (TAC) representation for the below fig. is show






```
temp1 := inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1   := temp3
```

where temp1, temp2 and temp3 are compiler generated

5. Code Optimization

In this phase, the compiler performs various transformation in order to improve the intermediate code. These transformation will result in fast running machine code.

i.e.
The code optimization phase attempts to improve the intermediate code. Some optimizations are trival.

for e.g.
a natual algorithm generates the intermediate code like

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 * temp2
id1   := temp3
```

even though there is a better way to perform the same calculation using two instructions.

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

In above, compiler can deduce the conversion of 60 from result, conclude that
int. to real representation can be done once and for all at compile time, so the inttoreal operation can be eliminated. and similarly only one tempory variable temp3 is used only once.

Local Optimization

There are "local" transformations that can be applied to a program to attempt an improvement.

- for eg., in Below fig., we saw two instances of jumps over jumps in the intermediate code, such as.

```
L1: if A>B goto L2
    goto L3
L2: T1 := 2*B
    T2 := T1 - 5
    if A ≤ T2 goto L4
    goto L3
L4: A := A+B
    goto L1
L3:
```

```
if A>B goto L2
goto L3                    — (1)
L2:
```

This sequence could be replaced by the single statement

```
if A ≤ B goto L3           —(2)
```

Sequence (1) typically replaced in the object program which

1. Compare A and B to set the condition codes,
2. jump to L_2 if the code for > is set, and
3. jump to L_3

Sequence (2), which

1. Compare A and B to set the condition code
2. jump to L_3 if the code for < or = is set.

if we assume A>B is true half the time, then for (1), we execute (1) and (2) all the time and (3) half the time, for an average of 2.5 instructions.

But for (2), we always execute two instructions, a 20% savings.

Loop Optimization

This optimization concerns speedups of loops. loops are especially good targets ~~at~~ for optimization because programs spend most of their time in inner 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. Then this computation is done only one each time the loop is entered, rather than once for each iteration of the loop. Such a computation is called loop invariant.

6. Code Generation

The final phase of the compiler is the generation of target code.

The code-generation phase converts the intermediate code into sequence of machine instructions. A simple e.g. of statement A := B + C into the machine code sequence

```
LOAD  B
ADD   C
STORE A
```

Such a straightforward macro-like expansion of intermediate code into machine code usually produces a target program that contains many redundant loads and stores and that utilizes the resource of the target machine inefficiently.

To avoid these redundant loads and stores, a code generator might keep track of the run-time contents of registers.

Another e.g., using registers 1 and register 2, e.g.

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

The first and second operands of each instruction specifies a source and destination, respectively.

The F in each instruction tells us that instructions deal with floating-point numbers.

This code moves the contents of the address id3 into register 2 then multiplies it with the real constant 60.0. The # signifies that 60.0 is to be treated as a constant.

Bookkeeping or Symbol-Table Management

An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. (1)

These attributes may provide information about the storage allocated for an identifier, its type, its scope) and in the case of procedures or function names, such things as the no and types of its arguments, the method of passing each argument (e.g. by reference), and its type returned, if any.

A symbol table or book-keeping 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.

The information about (identifiers) data objects are collected by the early phases of the compiler - lexical and syntactic phases - and entered into the symbol-table.

For e.g., when a lexical analyzer sees an identifier MAX, it may enter the name MAX into the Symbol Table if it is not already there, and produce as output a token whose value component is an index to this entry of symbol-table.

If a syntax analyzer recognices a declaration integer MAX, the action of the syntax analyzer will be to note in the symbol table that MAX has type "integer". No intermediate code will generate for this statement.

Error Handling.

One of the most imp. functions of a compiler is the Detection and reporting of errors in the source program.

The error messages should allow the programmer to determine exactly where the errors have occured.

Errors can encountered by virtually all of the phases of a Compiler. for e.g,

1. The lexical analyzer may be unable to proceed because the next token in the source program is missplaced.
2. The syntax analyzer may be unable to infer a structure for its input because a syntactic errors such as a missing parenthesis has occured.

3) The intermediate code generator may detect an operator whose operands have incompatible types.

4) The code optimizer, doing control flow analysis, may detect that certain statements can never be reached.

5) The code generator, may find a compiled-created constant that is too large to fit in a word of the target machine.

6) while entering info^n into the symbol table, the book-keeping routine may discover an identifier that has been multiply declared with contradictory attributes.

Whenever a phase of a compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message.

Once the error has been noted, the compiler must modify the input to the phase detecting the error, so that the latter can continue processing its input, looking for subsequent errors.

EXERCISE - 2

Compiler - Construction Tools', →

Some general tools have been created for the automatic design of specific compiler components. These tools use specialized languages for specifying and implementing the component, and many use algorithms that are quite sophisticated. The most successful tools are those that hide the details of generation algorithm and produce components that can be easily integrated into the remainder of a compiler.

Some useful tools are :→

1. Parser generators
2. Scanner generators
3. Syntax - directed translation engines.
4. Automatic code generators.
5. Data - Flow Engines.

1. Parser generators : → These produce syntax analyzers, normally from input that is based on context-free grammer. In early compilers, syntax analysis consumed not only a large fraction of the running time of Compiler, but a large fraction of the intellectual.

effort of writing a compiler.
This phase is now considered one of the easiest to implement

2.) Scanner generators :→ These automatically generate lexical analyzers, normally from a specification based on regular expressions.
The basic organization of the resulting lexical analyzer is in effect of finite automaton.

3.) Syntax-directed translation engines :→ These produce collections of routines that walk the parse tree, generating intermediate code ~~stuff~~
The basic idea is that one or more "translations" are associated with each node of the parse tree, and each translation is defind in terms of translation at its neighbor nodes in the tree.

(4.) Automatic code generators :→ Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine. The rules must include

~~PROGRAM 8~~

~~AIM :-~~

#include <iostream.h>

sufficient detail that we can handle the different possible access methods for data. eg., variables may be in registers, in a fixed location in memory, or may be allocated a position on a stack. The basic technique is "template matching".

(5.) Data-flow engines :- Much of the information needed to perform good code optimization involves "data-flow analysis", the gathering of information about how values are transmitted from one part of a program to each other part.

Compiler: Brief Overview of Compilation Process.

Elaborate on the compilation process with appropriate examples.

Design of a Compiler

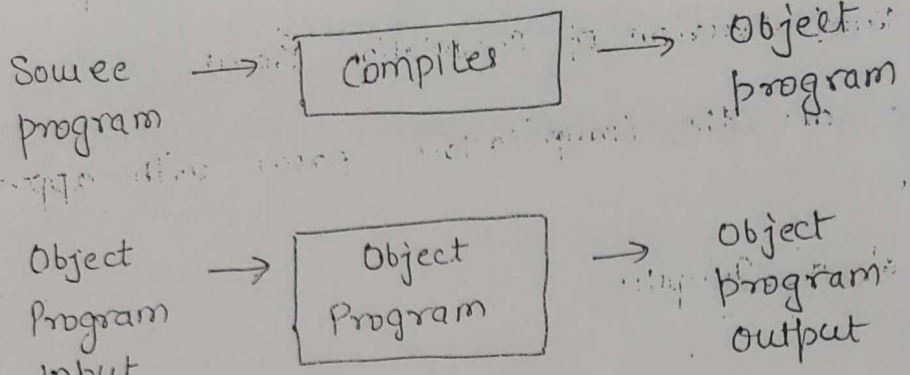
Ans:

A translator is a program that takes as input a program, written in one programming language (source language) and produces as output a program in another language (object/target language). If a source language is a high-level language such as fortran, PL/I or COBOL and the object language a low level language such as (assembly language or machine language) then such a translator is called a compiler.

Program Execution:

Executing a program written in a high-level programming language is basically a two step process.

1) Source program must first be compiled i.e. translated into the object program then the
2) resulting object program is loaded into a m/m and executed.



Compilation & Execution

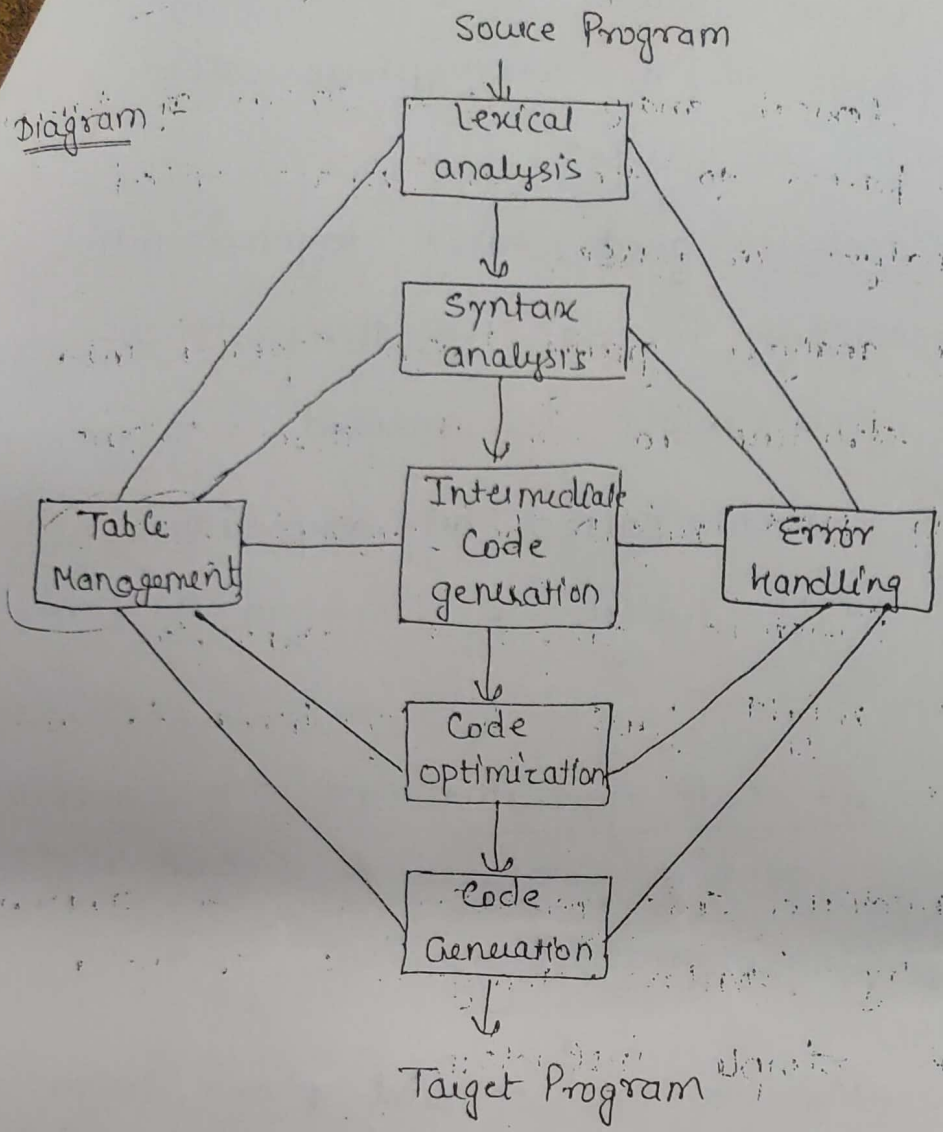
Structure of a Compiler

def A compiler take as input a source program and produces as o/p an equavilent sequence of machine instruction.

phases This process is so complex it is not reasonable to consider the compilation process as occuring in one single step, for this reason it is to partion the compilation process into a series of subprocesses called phases.

phase A phase is a logically cohesive operation that takes as input one representation of a source program and produces as output another representation

Diagram:-



Phases of a Compiler

Review of phases:

1) The first phase, called the lexical analyzer or scanner, seprates characters of sourc Language into groups that logically belongs together, these groups are called tokens.

The usual tokens are keywords, such as do or if, identifiers, such as X or NUM, operator symbols such as <= or + and punchuation symbols parantheis and commas.

2) The o/p of lexical analyzer is a stream of tokens, which is passed to the next phase called Syntax analyzer or parser.

The syntax analyzer groups taken together into syntactic structure for e.g, the three tokens representing A+B might be grouped into a syntactic structure called an expression. Expressions might further be combined to form statements.

3) The intermediate code generator uses the structure produced by syntax analyzer to create a stream of simple instruction.

4) Code optimization is an optional phase design to improve the intermediate code so that ultimate object program runs faster and/or takes less space

5) Final phase, code generation, produces the object code by deciding on the m/m locations for data, selecting code to access each datum, and selecting the registers in which each computation is to be done.

This is most difficult part of compiler design.