GATE

Be sure with Vidyalankar Since 1960

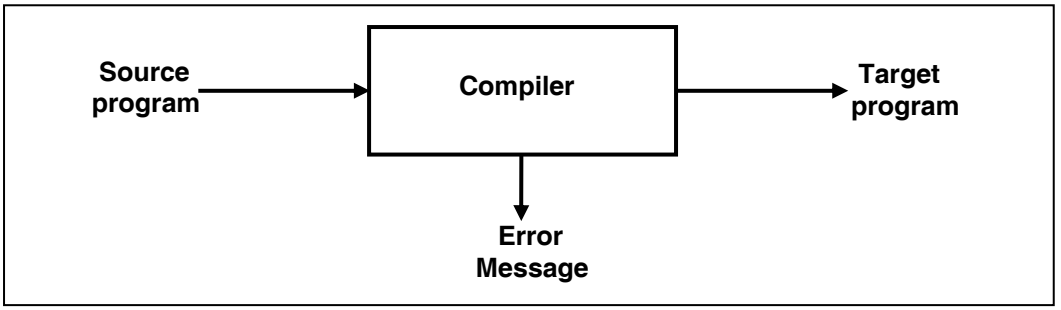
Pearl Centre, S.B. Marg, Dadar (W), Mumbai – 400 028. Tel. 4232 4232

CS : COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
Compiler Design

Index

COMPILERS

A compiler is a program that reads a program written, in one language i.e. the source language and translates it into an equivalent program in another language i.e. the target language.



There are two parts of compilation:

1) Analysis
2) Synthesis

1) **The Analysis** part breaks up the source program into constituent pieces and creates an intermediate representation on the source program.
2) **The Synthesis** part constructs the desired target program from the intermediate representation.

The Context of a Compiler

A source program may be divided into modules stored in separate files. A task of collecting the source program is sometimes entrusted to a distinct program, called a preprocessor. The target program created by the compiler may require further processing before it can be run.

A typical compilation is as shown below :

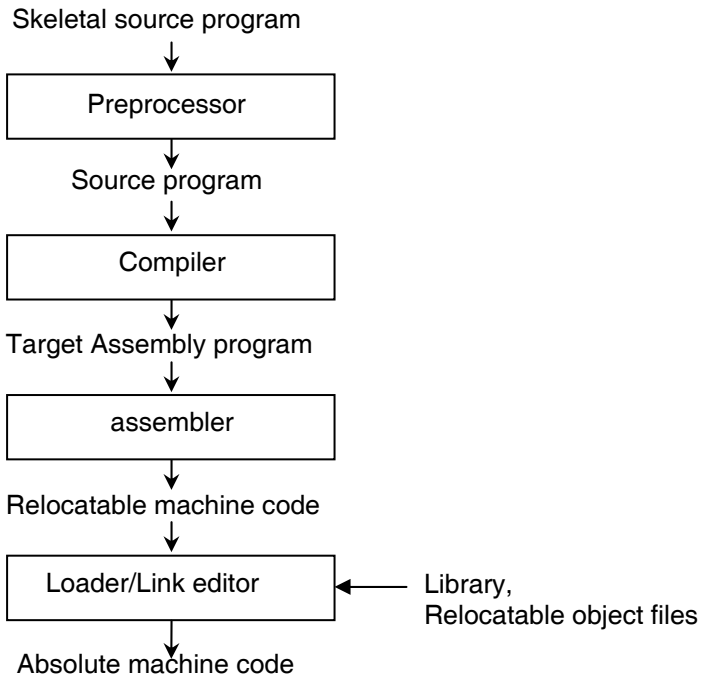


Fig: A language processing system

Analysis of the Source Program

In compiling, analysis consist of three phases :

1) **Linear Analysis** – In this analysis, the stream of characters making up the source program reads from left – to – right and grouped into tokens that are sequence of characters having a special meaning.
2) **Hierarchical Analysis** –In this analysis, characters or tokens are grouped hierarchically into nested collections with collective meaning.
3) **Semantic Analysis** – In this analysis, certain checks are performed to ensure that the components of a program fit together meaning fully.

(1) Linear Analysis: In a compiler, Linear Analysis is also called as Lexical Analysis or scanning.

Example.: In Lexical Analysis, the characters in the assignment statement:

$y = x + r * 69;$

would be grouped into following tokens.

i) The identifier y
ii) The assignment symbol
iii) The identifier x
iv) The plus sign
v) The identifier r
vi) The multiplication sign
vii) The number 69.

(2) Hierarchical Analysis: Hierarchical Analysis is called parsing 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 following figure represents the grammatical phases of the source program.

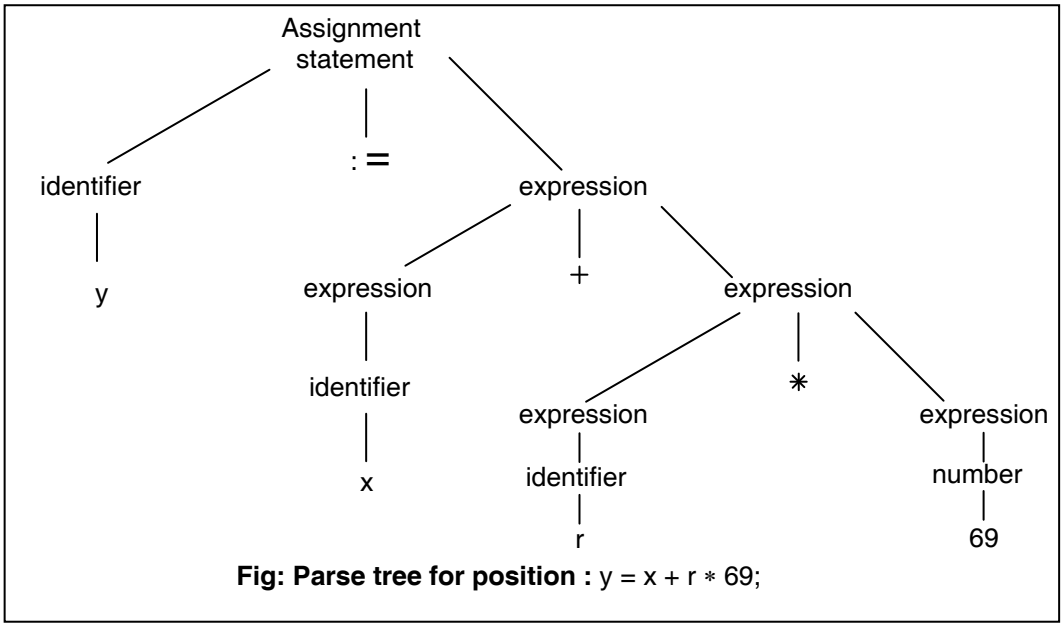


Fig: Parse tree for position : y = x + r * 69;

Lexical constructs do not require recursion. Context free grammars and a formalization of recursive rules can be used to guide syntactic analysis. The characters grouped are recorded in a table, called a symbol table and removed from the input processing of the next token.

(3) Semantic Analysis : The semantic analysis phase checks the source program for semantic error 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.

Many software tools that manipulate source programs first perform some kind of analysis. Some examples of such tools include :

1) *Structure editor*: A structure editor takes as input a sequence of commands to build a source program. The structure editor not only performs the text creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program. Thus, the structure editor can perform additional tasks that are useful in the preparation of programs.

2) *Pretty printers*: A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.

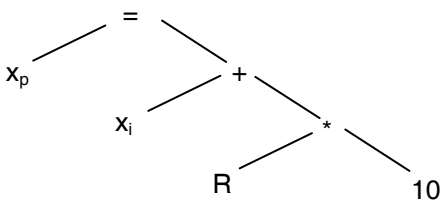
3) *Static checkers* : A static checker reads a program analyzes it, and attempts to discover potential bugs without running the program. The analysis portion is often similar to that found in optimizing compilers.

4) *Interpreters* : Instead of producing a target as a translation, an interpreter performs the operations implied by the sources program. For an assignment statement $d_p = d_i + R * 10$, for example, an interpreter might build a tree shown in figure below, and then carry out the operations at the nodes as it "walks" the tree.

For example :
Consider the following assignment statement

$$x_p = x_i + R * 10$$

Syntax Tree :



At the root, it would discover it had an assignment to perform, so it would call a routine to evaluate the expression on the right, and then store the resulting value in the location associated with the identifier x_p. At the right child of the root, the routine would discover it had to compute the sum of two expressions.

Interpreters are frequently used to execute command languages, since each operator executed in command language is usually an invocation of a complex routine such as an editor or compiler.

PHASES OF COMPILATION

The compilation process is partitioned into several subprocesses called the phases of compilation. A phase is a logically cohesive operation that takes one representation of a source program as input and produces another representation as output.

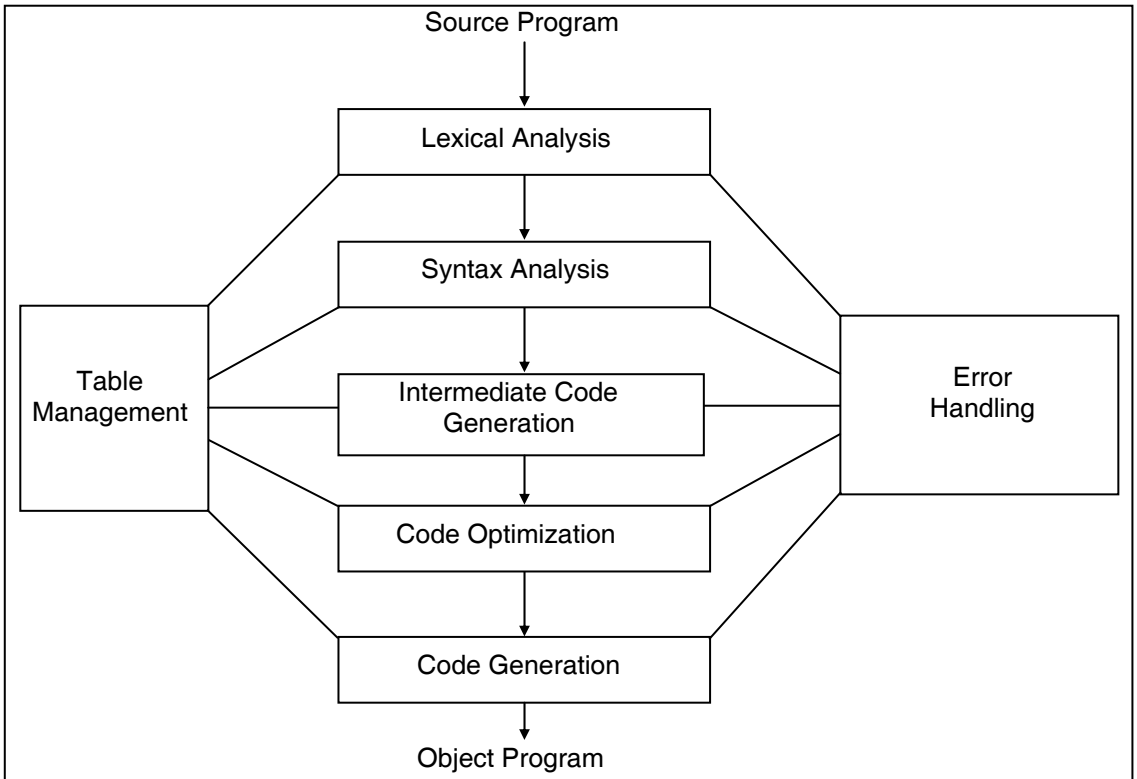
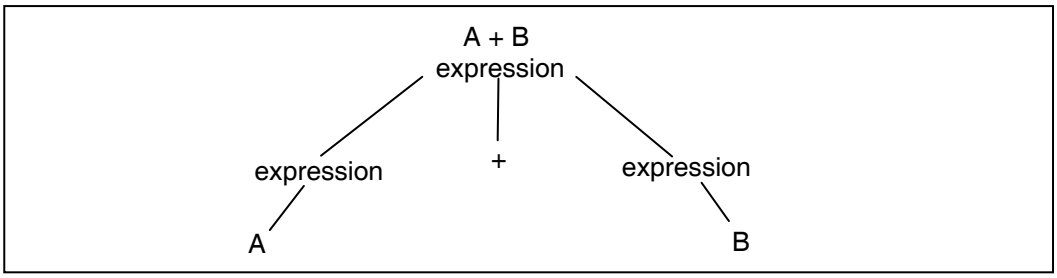


Fig. Phases of a compiler

i) **Lexical Analysis:** This scanning process separates the characters of the source language into groups, that logically belong together; these groups are known as **tokens**. Usually tokens are **keywords** like if, for, while, do etc., **identifier** like sum, max, prod etc., **operator symbols** like <, >, = and **punctuation marks** like ; , ' , " etc. The output of this phase will be a stream of tokens which will be passed to the next phase i.e. syntax analysis.

ii) **Syntax Analysis :** This is parsing phase. The syntax analysis groups the tokens together into syntactic structures. The syntactic structure can be regarded as a tree whose leaves are tokens.

Example



The syntax analyzer checks that the tokens appearing in its input (this input is output of lexical analysis) occur in patterns that are permitted by the source language.

iii) **Intermediate Code Generation :** The intermediate code generator uses the tree structure created by the parser to create a stream of simple instructions.

Example : Three address code form

One popular type of intermediate language is known as three address code. Instructions in this form will use at the most three addresses.

A typical three address code statement will be :

A = B OP C, where A, B, C are operands and OP is a binary operator.

In addition to statements that use arithmetic operators, an intermediate language needs unconditional and simple conditional branching statements.

i.e. **goto** 10 ;
if a > b **then**
goto 30 ;

Statements like **while–do , do–while, for, if–then–else–repeat–until** are translated into this lower level conditional and unconditional three address statements.

iv) **Code Optimization :** It is an optional phase designed to improve the intermediate code so that the ultimate object program runs faster and takes less space. Output of this phase will be another intermediate code that does the same job as the original but in a more convenient manner.

There are two types of optimization :

(a) Local Optimization
(b) Loop Optimization

(a) **Local Optimization** : These are local transformations applied to a program to attempt an improvement.

Example.

1) **if** A > B **then**
goto 10 ;
goto 20 ;
10 :
The above structure can be modified as follows :
if A < = B **then**
goto 20 ;
10 :

2) A = B + C + D;
K = B + C + M;
The above instructions can be modified as follows :
T = B + C;
A = T + D;
K = T + M:

(b) **Loop Optimization** : A typical loop improvement is to move a computation that produces the same result each time the loop is repeating, to a point in the program just before the loop is entered. Such a computation can be called as **loop invariant**.

v) **Code Generation :** The code generation phase will convert the code into a sequence of machine instructions.

Example

The instruction **A = B + C** may be converted into machine code sequence as follows :

LOAD B
ADD C
STORE A

Such a expansion of intermediate code into machine code usually produces an object program that contains many redundant **LOAD's** and **STORE's** and that utilizes the resources of the computer inefficiently. To avoid this redundant **LOAD's** and **STORE's,** a code generator may keep track of the run–time contents of registers.

vi) **Table Management :** It is a portion of the compiler which keeps track of the names used by the programs and records essential information about each thing, such as its type, etc. The data structure used to record this information is known as **Symbol Table**.
The information about data objects is collected by the lexical analyzer and syntax analyzer and entered into the symbol table.
Example :
When a lexical analyzer sees an identifier **MAX**, it may enter the name **MAX** into the symbol table if it is not already present. If the syntax analyzer recognizes a declaration **int MAX**, it will note the information i.e. the type of **MAX** in the symbol table.

vii) **Error Handling :** The error handler will be invoked when a bug in the source program is detected. It must warn the programmer by sending a proper message. The error message should allow the programmer to determine exactly where the errors have occurred. Errors can be encountered by all of the phases of the compiler.

Example

a) The lexical analyzer may be unable to proceed because the next token in the source program is misspelled.
b) While entering information into the symbol table the table management routine may find an identifier that has been declared more than once with contradictory attributes.
c) The syntax analyzer may be unable to infer a structure for its input because a syntactic error such as a missing parenthesis has occurred.
d) The intermediate code generator may detect an operator whose operands have incompatible types.
e) The code optimizer, doing control flow analysis, may detect that certain statements can never be reached.

Whenever a phase of the 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 inputs looking for subsequent errors.

COMPILER WRITING TOOLS

A number of tools have been developed specifically to construct compilers. These tools range from scanner and parser generator to complex systems, called **compiler–compilers**, **compilers–generator** or **translator writing system**, which produce a compiler from some form of specification of a source language and target machine.

The input specification for these systems may contain :
1) a description of the lexical and syntactic structure of the source language.
2) a description of what output is to be generated for each source language construct.
3) a description of the target machine.

The principal aids provided by existing compiler–compilers are :
(i) scanner generators
(ii) parser generators
(iii) code generators.

CROSS COMPILER

A compiler is characterized by three languages, its source language, object language and the language in which it is written. These languages may all be quite different.

Example

A compiler may run on one machine and produce object code for another machine. Such a compiler is often caused a **cross–compiler**.

Many microprocessors & minicomputers are implemented this way. They run on a bigger machine and produce object code for the smaller machine. A compiler being implemented in its own language. Suppose we have a new language **'L'**; which we want to make available on several machines, say **A & B**. For machine A, a small compiler $\mathbf{C_A^{SA}}$ that translates a subset **'S'** of language **'L'** into the machine or assembly code of **'A'**. We then write a compiler $\mathbf{C_S^{LA}}$ in the simple language **'S'**. This program, when run through $\mathbf{C_A^{SA}}$, becomes $\mathbf{C_A^{LA}}$, the compiler for the complete language **'L'**, running on machine **'A'**, and producing object code for **'A'**.

The process is shown below:

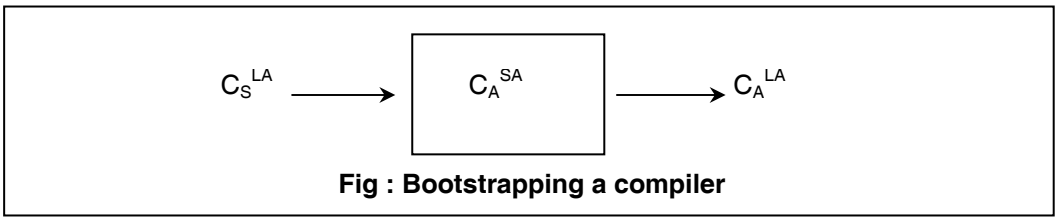


Fig : Bootstrapping a compiler

Using $\mathbf{C_L^{LB}}$ to produce $\mathbf{C_B^{LB}}$, a compiler for language **'L'** on **B**, is now a two–step process, as shown below:

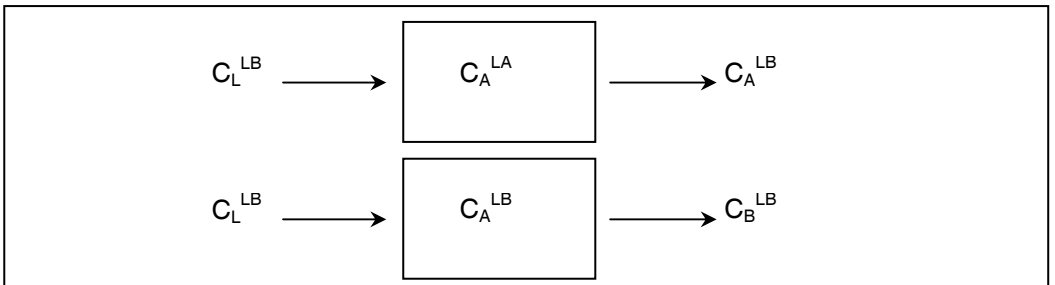


Fig : Bootstrapping a compiler to a second machine.

We first run C_L^{LB} through C_A^{LB} to produce C_A^{LB}. A cross–compiler for **'L'** which runs on machine **'A'** but produces code for machine **'B'**. Then we run C_L^{LB} through cross–compiler to produce the desired compiler for **'L'** that runs on machine **'B'** & produces object code for **B**.

THE GROUPING OF PHASES

Whatever we discussed up–til now deals with the logical organization of a compiler. In an implementation, activities from more than one phase are often grouped together.

Front and Back Ends :
More oftenly, the phases are collected into a front end and a back end

- The front end consists of those phases or parts of phases, that depends on the source language primarily and largely independent of the target machine.
  These normally includes –
  → Lexical and syntactic analysis
  → The creation of the symbol table, semantic analysis
  → The generation of intermediate code.
  → A certain amount of code optimization and error handling with each phase.
- The back end includes those portions of the compiler that depend on the target machine, and generally these portions do not depend on the source language, just the intermediate language.
  These normally include :
  → The code optimization phase
  → Code generation, along with the necessary error handling and symbol–table operations

Passes :
Several phases of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file. It is common for several phases to be grouped into one pass and for the activity of these phases to be interleaved during the pass.

For example
Lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass. If so, the token stream after lexical analysis may be translated directly into intermediate code.

Reducing the number of passes
It is always desirable to have relatively few passes, since it takes time to read and write intermediate files. On other hand, if we group several phases into one pass, we may be forced to keep the entire program in memory, because one phase may need information in a different order than a previous phase producing it. The internal form of the program may be considerably bigger than either the source program or the target program, so space may not be trivial.

Compiler–Construction Tools
The compiler writer, like any programmer, can profitably use software tools such as debuggers, version managers, profilers, and so on. Some general tools have been created for automatic design of specific compiler components. The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of a compiler.

The following is a list of some useful compiler–construction tools.

No.	Tool	Description
1	Parser generator	• These produce syntax analyzer, normally from input that is based on a context–free grammar
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 a finite automaton.
3	Syntax–directed translation engines	• These produce collection of routines that walk the parse tree, for generation of intermediate code. • The basic idea is that one or more " translation" are associated with each node of the parse tree, and each translation is defined in terms of translations 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 languages into the machine language for the target machine • The rules must include sufficient details so that we can handle the different possible access methods for data
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 every other part. • Different tasks of this nature can be performed by essentially the same routine with the user supplying details of the relationship between intermediate code statements and the information being gathered.

PROGRAMMING LANGUAGE CONSTRUCTS

High–level language

It allows a programmer to express algorithm in a more natural notation that avoids many of the details of how a special computer functions.

It also makes programming task simpler, but it also introduced some problems. That is, we need a program to translate high-level language into a language that machine can understand. That means it needs a compiler to be processed.

A compiler is a program that accepts a program written in a high-level language and produces an object program.

High level language is closer to the programmer than the hardware. Hence, programmer can do programming with no specific computer hardware to keep in mind. In this sense, it is **"hardware independent".**

It is at the highest level of the program language hierarchy. These are machine independent since close to programmer and it is easier to understand as well as to program.
Program length is smaller than most of the low-level languages.

Assembly language

Assembly language is in intermediate level between machine code and high level language.
It is very close to hardware and is therefore restricted by the capabilities of the hardware.
Hence, these are **machine dependent**.

The assembly programmer knows that computer has central processing unit **(CPU)** which has **ALU, CU** & few **CPU** registers.
Some features are:
(1) It has mnemonic.
(2) Addresses are symbolic, not absolute.
(3) Reading is easier than m/c language.
(4) Introducing data to program is easier.

But it requires an assembler to translate a source program into object code.
The program length is manageable but it is difficult to understand as well as program.

Machine Language

It is a low-level language and is at the lowest level of the program language hierarchy.
There is one-to-one correspondence between the assembly and machine instructions. The assembler program, converts the assembly language program into, machine language in a predefined instruction format.

The assembled machine language program can be stored as a file on the disk. At the time of execution, machine language program has to be brought into memory from disk and then loaded at appropriate locations. This is done by **loader**.

This language take a hypothetical machine operation code.

Example **ADD X, Y** in assembly language which is 0110 001110 010101 in machine language where 0110 for "add":, 001110 and 010101 are the addresses of X & Y.
Since there is machine level coding itself, hence no translator is required to convert the programs in the executable form and the programs are directly executed.

The programming in machine level is very difficult and complex to understand and implement. Hence the program length is also usually very long.

Examples:

High–level language:

example.

COBOL, C, FORTRAN, ALGOL etc.

COMPUTE C = A ∗ B/C

Assembly language:

example.

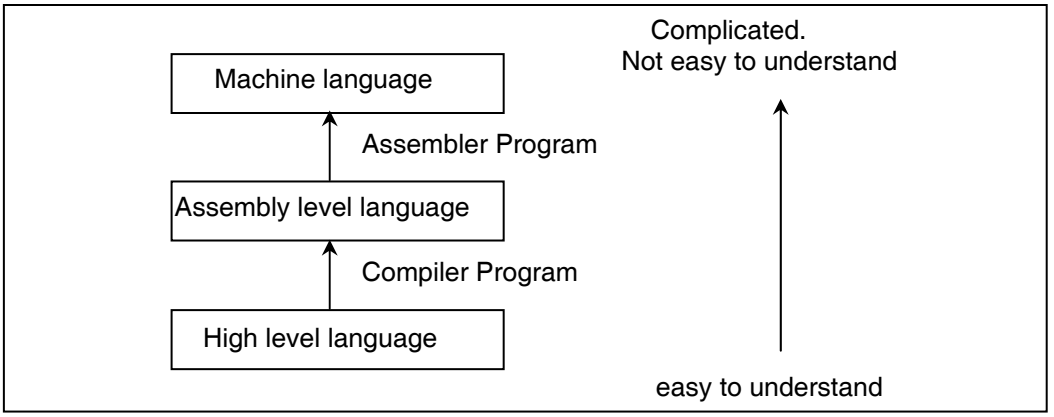
8085 and 8086 instructions sets.

```
L  3,  TEN        Load the number 10 into register 3
DC   F '10'       Define constant 10
```

Machine language:

Absolute address	Relative address	Hexa decimal	Instructions
48	0	58201398	L 2,904 (0, 1) – (Load reg. 2) from location (904) + C (reg 1) = 952

Hierarchy of languages:



High level programming languages:

A programming language is a notation with which people can communicate algorithms to computers and to one another. Some of the aspects of high–level languages which make them preferable to machine or assembly language are the following:

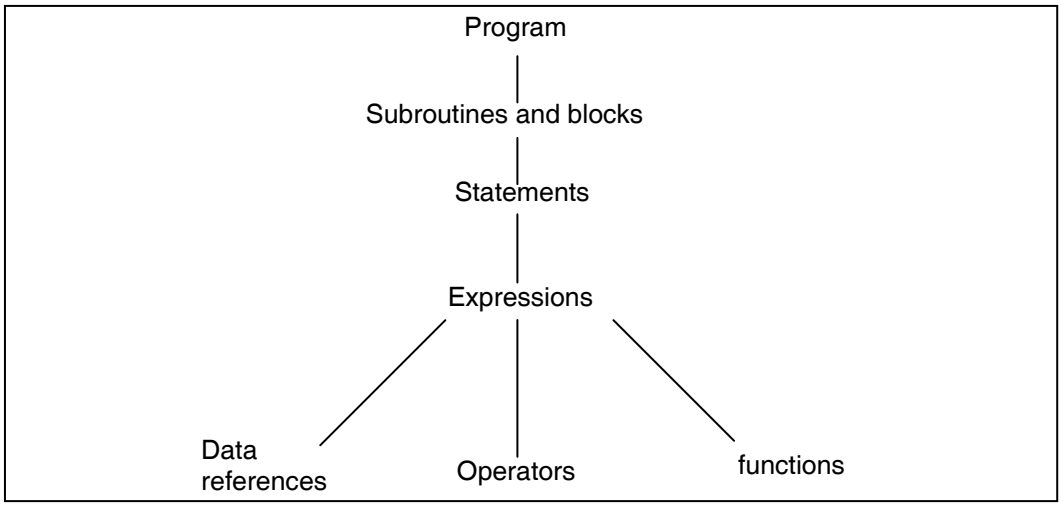
1) **Ease of understanding:**
   A high–level language program is generally easier to read, write and prove correct than an assembly language program, because a high level language usually provides a more natural notation for describing algorithms than assembly language.

2) **Naturalness**

3) **Portability**

4) **Efficiency of use:**
   However, particular high–level language features facilitate reliable programming. Some of the features are:
   → *data structures*
   → *Scope rules :* that allow modifications to be made to a piece of a program without affecting other portions of the same program.
   → *Flow of control:* Constructs that clearly identify the looping structures in a program.
   → *Subroutines:* allowing a program to be designed in small pieces.

Hierarchy of program elements:



Data Elements

One of the basic building blocks of a programming language is the set of primitive data elements to which operators can be applied and out of which more complex data structures can be built.

Various commonly used data elements are:

1) Numerical data
2) Logical data
3) Character data
4) Pointers
5) Labels

Various Data Types

Identifiers and Names

On the logical level, each programming language manipulates and uses object (or values), which are data elements such as integers or real or are more complex items such as arrays or procedure. It is customary to view the computer as consisting of an abstract store, consisting of cells in which a value of any type may be kept. Each cell has a name, which is usually a variable of the programming language. Each name, is denoted by an identifier, which is a string of characters.

Attributes
The attributes of a name determine its properties. The most important attribute of a name is its type, which determined what values it may have.

Declaration
The attributes of a name are determined implicitly, explicitly or in some cases by default. For example: in FORTRAN, any identifier beginning with I, J, … N is implicitly an integer. Generally, the attributes of a name can be set by means of an explicit declaration.

Binding Attributes to Names
The act of associating attributes to a name is often referred to as binding the attributes to the name. Most binding of Attributes is done during compilation (at compile time) when the attributes become known as the compiler reacts to declaration. This is called as static binding. Some languages allow dynamic binding, or binding of attributes while the program is running.

Data Structures
Various data structures are:

1) Lists
On the logical level, a list is either null or a data element followed by a list. The basic operations on lists are the insertion, deletion or substitution of an element, and the determination of whether a given element is on the list.

2) Trees
A tree T can be recursively defined as a collection of elements (nodes) with the following structural relationship.
a) One element r is called the root of T.
b) The remaining nodes can be partitioned into $K \geq 0$ subtrees T_1, T_2, … T_K such that root of T_i is a child of r.

The order is indicated by listing the children of each node from left to right in the sequence in which they are to appear.

3) Arrays
An array is a collection of elements of some fixed type, laid out in a K–dimensional rectangular structure. A measure of the distance along each dimension is called an index or subscript, and the elements are found at integer points from some lower limit to some upper limit along each dimension.
An element of an array is named by giving the name of the array and the values of its indices, as $A[i_1,...i_k]$. The upper and lower bounds and the total number of elements in an array may be known at compile time (a fixed size array) or determined at run time (an adjustable array).

a) **Fixed size one–dimensional array**
If the size of the array is known at compile time, then it is expedient to implement the array as a block of consecutive words in memory. If it takes K memory units to store each data element, then A[i], the i^{th} element of array A, begins in location

BASE + K * (i–LOW)

Where LOW → lower bound on the subscript
BASE → lowest numbered memory unit allocated to the array.

b) **Fixed size multidimensional Array**
Considers the 2×3 array as shown in fig. below

A[1,1]	A[1,2]	A[1,3]
A[2,1]	A[2,2]	A[2,3]

A 2−D array is normally stored in one of two forms, either row−major or column major.

c) **Adjustable arrays**
Many languages, such as ALGOL and PL/I, but not FORTRAN, permit the size of array to be specified dynamically i.e. at runtime. APL allows even the number of dimensions to vary at run time. In such situations, a dynamic storage allocation is used to provide the necessary storage.

4) Record Structures
An important class of data structures is the record structure, found in COBOL or PL/I. Logically, a record structure is a tree, with the fields of the record being the children of the root, the subfields being children of these, and so on. Record structures are implemented as a block of memory. To determine how much storage is necessary, and what the accessing functions for the various fields are, we must determine the width and offset for each field.

5) Character strings
These are one−dimensional arrays whose elements are characters. They may thus be represented as arrays. A data descriptor consisting of the number of characters currently in the string is essential and must appear with the string if the length varies.

6) Stacks
A stack is a linear list that we operate only at the beginning or top end. We may push a new element onto the stack, making it the first element on the list.
Example: pushing D onto the list A, B, C yields the list D, A, B, C. We may also pop elements from the top by removing them. If we pop the stack A, B, C we get B, C.

- Any program, regardless of the language used, may be used as specifying a set of operations that are applied to certain data in certain sequence. Basic difference among languages exists in the type of data allowed, in the type of operations available.
- This Mechanism provided for controlling the sequence in which the operations are applied to the data.
- The data storage areas of an actual computer, such as the memory, register and external media, usually have a relatively simple structure as sequence of bits groups into bytes or words.
- Data storage of virtual computer for a programming language tends to have a more complex organization with arrays, stacks and other data structure.
- As different data structures occupy different size of memory, the declaration of the data is essential. Some of the data objects that exist during program execution is programmer defined.

- They will be variable, constant, arrays, files etc. Other data object will be system defined, they will be data objects that the virtual computer set up for housekeeping operation during program execution and not directly accessible by the user. **Example**. Run time storage stack, subprogram activation records, file buffers etc.
- If we observe the cause of a program, some data objects exist at the beginning of execution. Some data objects are destroyed during execution other exists until the program terminates. Each object has the lifetime during which it may be used to store the data values.
- A data object participates in various binding during its lifetime. While the attributes of data object is invariant during its lifetime, the binding may change dynamically.
- The most important attributes and bindings are based on data type, its location, value the data possess, name of the data and components it have.
- **The data object** is elementary if it contains a data value that is always manipulated as a unit. It is a data structure if it is an aggregate of other data objects.
- **Arrays** are the most common data structure in programming language.
  The attributes of the arrays are
  1) Number of component.
  2) Data type of each component.
  3) Subscripts to be used to select each component.
- The array might be one dimensional (linear) or two dimensional or matrix type. There might also be multidimensional array.
- A **data structure** composed of a fixed number of components of the different type is usually termed a record. Both record and array are form of the fixed length linear data structure, but records differs in two ways.
  1) The component of record may be heterogeneous or mixed data type.
  2) The component of record are named with symbolic names rather than indexed with subscripts.
- All the data types used are used to evaluate the expression. First the expression should be parsed to check its validity, then by referring the appropriate data expression should be evaluated.
- To do the control structure over the program, we are required to do the control in some way. One of them is the recursion, where the loop is calling the procedure or function within itself to do the recursion over them.

Parameter passing Methods

Communication between two procedures, one of which calls the other, is done through global variables and parameters of the called procedure.

Example–1

```
Integer procedure DIVIDE (X,Y) integer X, Y;
if Y = 0 then
return 0
else
return X/Y
```

In above **example**, **DIVIDE** is the procedure. X and Y are the formal parameters or Formals.

Example–2

A: = **DIVIDE (B, C)**

Here B and C are the actual parameters or actuals.

There are three common methods of parameter passing :

1) Call–by–Reference
2) Call–by–value
3) Call–by–name

1) **Call–by–Reference**

In this mechanism, the address of the actual parameter is passed to called function.
If the parameter is an expression, its value is stored in temporary locations.
If the parameter is an array element its address is computed at the time of call.
Call by reference is also known as call–by–address or call–by–location.
When a parameter is passed by reference, the calling program passes a pointer to the subroutine which contains r–value of each actual parameter.
The address of the actual parameter is put in a known place determined by the language implementation. Therefore the calling procedure knows where to put the address of actual parameters (pointer) and the called procedure knows where to find them.
If expression is having l–value, then the expression is evaluated in a new location and the l–value of that location is passed.

Example

Procedure **SWAP (X, Y)** integer X_1, Y_1

begin

Integer TEMP;
TEMP: = X_1;
X_1: = Y_1;
Y_1 = TEMP;

end

Fig: The procedure SWAP

X and Y are the formal parameter & SWAP is the procedure.
SWAP (X, Y), where X & Y are actual parameter.

2) **Call–by–value**

In this, the values of actual parameter are passed to the called function. These values are assigned to the corresponding formal parameter.
Passing of values occur in one direction only.
It is commonly used in built in functions of the language
This is simplest possible method of parameter passing.
The actual parameter are evaluated and their r–values are passed to the subroutines. These are determined by the language implementation.
In call–by–value, it is not possible to change the values in the calling program, unless the user explicitly pass pointer as actual parameters.
The formal parameters are declared as pointers and are used in the called procedure.

Example:
The call procedure SWAP (X, Y) would be equivalent to the sequence of steps as follows:

$T_1 := X$

$T_2 := Y$

$TEMP := T_1$

$T_1 := T_2$

$T_2 := TEMP$

Where T_1, T_2 are names local to SWAP.
A generalization of call–by–value is copy–restore (copy–in, copy–out or value–result) linkage.

3) **Call–by–Name**
Every occurrence of a formal parameter in the body of the called functions is replaced by the name of the corresponding actual parameter. Call–by–Name is hard to implement.
The actual parameter is unevaluated until they are needed. Any local name of the called procedure cant have the same identifier as a name mentioned in the actual parameter. It must be given a distinct identifier before the substitution of actual formals. The actual parameters must be surrounded by parentheses. These are necessary to preserve their integrity.

4) **Call–by–result value**
The capabilities of call by result value is extended by copying the value of the formal parameters back to corresponding actual parameters at return.

STORAGE MANAGEMENT IN COMPILER

In storage management, there are number of elements to which storage must be allocated in order to execute the object program. The storage is required for the object program and the user–defined data structure, variables & constants. The temporaries are required for expression evaluation and parameter transmission. The space is allocated for input/output buffers.

There are two types of storage allocation possible :
(1) Static storage Allocation
(2) Dynamic storage Allocation

(1) **Static storage Allocation**
The storage allocation is done statically, when the compiler decides the size of every data item. If recursive procedure calls are not permitted, then the space for all programs and data can be allocated at compile time.
Advantage of static Allocation
(i) It is easy to implement.
(ii) No run–time support is required as the library routine are loaded with object program.

Example:
FORTRAN uses the static allocation method. In this, each subroutine is separately compiled and the space is required for array computation. There is no recursion in **FORTRAN**, as each subprogram can store its return address in a private location. Total space required for a program is summation of the space needed for the subprograms, their data and linkage information & library routines used.

(2) **Dynamic storage Allocation**
The dynamic storage allocation is used when a program contains either recursive procedures or data structures whose size is adjustable.
There are two types of dynamic allocations:

i) **Stack Allocation:**
Stack allocation is useful for handling recursive procedures. Each stack has a Stack pointer. When a procedure is called, it places its data on top of a stack and increments the stack pointer. Similarly when procedure returns, it pops its data from the stack and decrement the stack pointer.
Stack storage allocation scheme is not used to handle arbitrary allocation and release of storage. The last–In–First–Out **(LIFO)** mode of operation handles the basic storage requirements of a block–structured language.
All fixed sized storage required by variables declared in one procedure into a single chunk of storage in called an 'activation record'.
The diagram of 'activation record' is as follows:

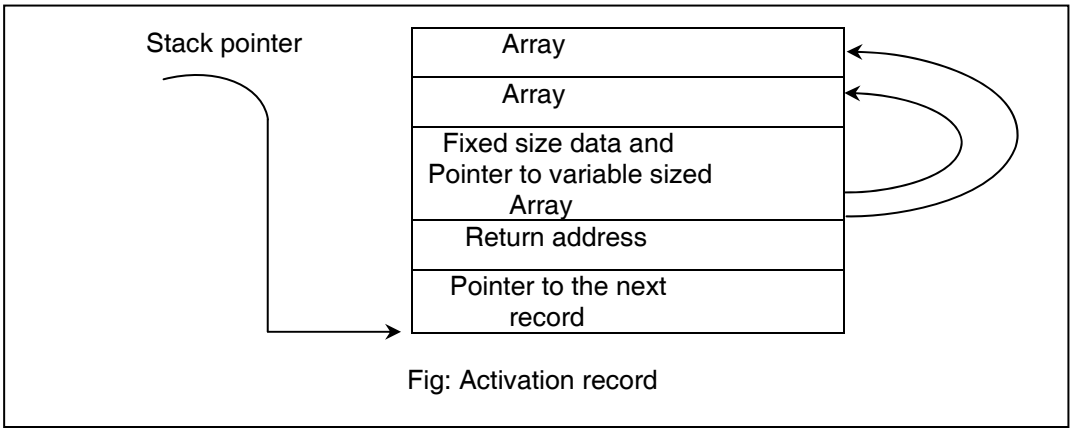


Fig: Activation record

ii) **Heap Allocation:**
Heap allocation is useful for implementing data whose size varies as the program is running.

Examples:
Strings in **SNOBOL** or Lists in **LISP**
It takes a large block of memory and dividing it into variable length blocks, out of which some of the blocks are used for data.

In **LISP & SNOBOL** languages, data is constantly being created, destroyed and modified in size. It is inconvenient to place variable length data on stack.
Heap is nothing but a run–time organization.

The diagram of **heap allocation** is as follows:

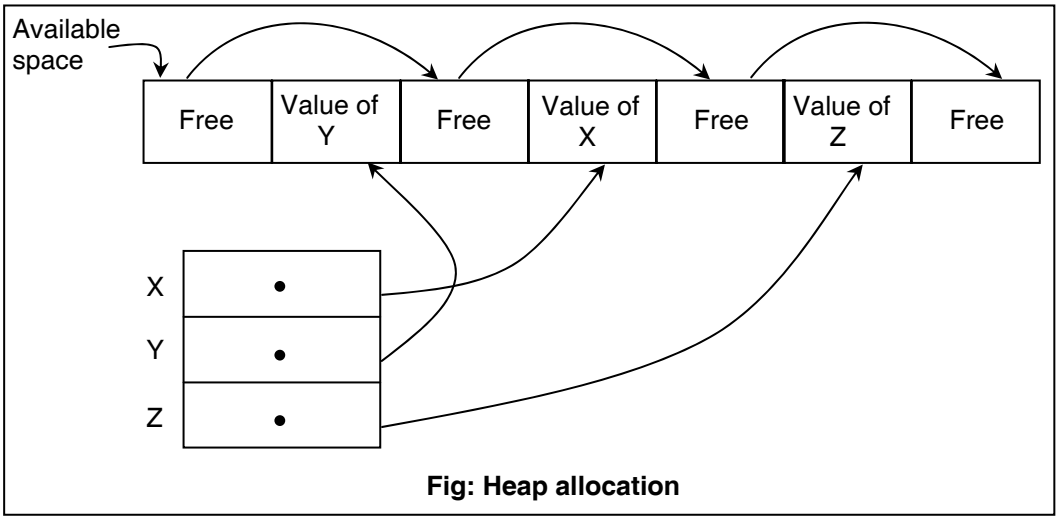


Fig: Heap allocation

X, Y, Z are the pointers from fixed locations. These fixed locations might be allocated statically or they might be on a stack. These pointers points to blocks of memory in heap, and the value of each name including the data descriptor give the block length which is kept in the block.

Each free block contains a pointer to the next free block and information regarding how long the free block is.

COMPARISON OF COMPILER AND INTERPRETERS

	Compiler	Interpreter
1	It converts the whole program written in HLL into its equivalent machine code in one go and execute it if the program is error free.	It converts line by line of the source program written in **HLL** into its equivalent machine code in one go and executes it if the program is error free.
2	Speed of execution is very fast	The speed of execution is slow
3	The translation is done of the entire program	The translation of program is done line by line
4	It saves a lot of time when the program has to be executed repeatedly	It takes a lot of time when the program has to be executed repeatedly
5	It requires a large memory space	It requires less memory space
6	It creates an object file.	It does not create an object file.
7	Software cost is high	Software cost is less.
8	The code size of the compiler is much larger	The code size of the interpreter is smaller
9	Program needs to be compiled only once and can be executed repeatedly	For every new run of the program, the program needs to be translated.
10	**example C** compiler	**example**. **BASIC** interpreter.

LMR (LAST MINUTE REVISION)

- **Compiler**
  A compiler is a program that reads a program written, in one language i.e. the source language and translates it into an equivalent program in another language i.e. the target language.

There are two parts of compilation

Analysis	Synthesis
Breaks up the source program into constituent pieces and creates an intermediate representation.	Constructs the desired target program from the intermediate representation.

- Analysis of the source program

Linear Analysis	Hierarchical Analysis	Semantic Analysis
In this, the stream of characters making up the source program reads from left – to right and grouped into tokens	In this, Characters or tokens are grouped hierarchically into nested collection with collective meaning.	In this, Certain checks are performed to ensure that the components of a program fit together

- **Phases of Compilation**
  The compilation process is partitioned into several subprocesses called the phases of compilation
  i) *Lexical Analysis* : This scanning process separates the characters of the source language into groups, that logically belong together ; these groups are known as tokens. Usually tokens are keywords like if, for , while, do etc. identifier like sum, max etc. operator symbols like < , >, = and punctuation marks like ; , ', " etc.
  ii) *Syntax Analysis* : The syntax analysis groups the tokens together into syntactic structures. The syntactic structure can be regarded as a tree whose leaves are tokens
  iii) *Intermediate code generation* : The intermediate code generator uses the tree structure created by the parses to create a stream of simple instructions.
  Example : Three address code form $\Rightarrow$ A = B OPC where A, B, C are operands and OP is a binary operator.
  iv) *Code optimization* : It is an optional phase designed to improve the intermediate code so that the ultimate object program runs faster and takes less space. Output of this phase will be another intermediate code.

Two types of optimization

Local optimization	Loop optimization

v) *Code Generation* : The code generation phase will convert code into a sequence of machine instructions.
Example : The instruction A = B + C
LOAD B
ADD C
STORE A

vi) *Table management* : It is a portion of the compiler which keeps track of the names used by the programs and records essential information about each thing such as its type etc. The data structure used to record this information is known as symbol table.

vii) *Error handling* : The error handler will be invoked when a bug in the source program is detected. It must warn the programmer by sending a proper message. The error message should allow the programmer to determine exactly where the errors have occurred. Errors can be encountered by all of the phases of the compiler.

- **Cross Compiler**
  A compiler may run on one machine and produce the object code for another machine. Such a compiler is called a cross – compiler.
- **High – level language**
  → It allows a programmer to express algorithm in a more natural notation that avoids many of the details of how a special computer functions.
  → A compiler is a program that accepts a program written in a high–level language and produce an object code.
  → It is hardware independent.
- **Assembly Language**
  → It is an intermediate level between machine code and high level language.
  → It is machine dependent.
- **Machine language**
  → It is a low–level language and is at the lowest level of the program language hierarchy.
  → The assembler program converts the assembly language program into, machine language in a predefined instruction format
- **Interpreter**
  → It converts line by line of the source program written in high level language into its equivalent machine code in one go and executes it if the program is error free
  → The speed of execution is slow and it requires less memory space.
  Errors can be encountered in all the phases of compiler errors
  – Syntactic Errors – Semantic Errors – Lexical Errors
- **Data Structures**
  Lists, Trees, Arrays, Records, character strings, stacks are various data structures available.
- **Parameter Passing Methods**
  – Call–by–reference – Call–by–value
  – Call–by–Name – Call–by–result value

❑ ❑ ❑ ❑ ❑ ❑

ASSIGNMENT – 1

Duration : 45 Min. **Max. Marks : 30**

Q1 to Q6 carry one mark each

1. If ABC translator is used for translation of COBOL into assembly language then in this case ABC is called as _____
   (A) Assembler (B) Compiler
   (C) Interpreter (D) None of these

2. In _____, certain checks are performed to ensure that the components of program fit together meaningfully.
   (A) Linear analysis (B) Hierarchical analysis
   (C) Semantic analysis (D) Sentential analysis

3. During _____, the compiler tries to detect construction that have the right syntactic structure but no meaning to the operation in solved.
   (A) Syntax analysis (B) Semantic analysis
   (C) Both (A) and (B) (D) Neither (A) nor (B)

4. _____ is the processing of every statement in a source program , or its equivalent representation, to perform a language processing function (a set of language processing function)
   (A) Analysis pass (B) Synthesis pass
   (C) Language processor pass (D) None of these

5. Which of the following is a semantic action ?
   (A) Checking semantic validity of constructs in synthesis phase
   (B) Determining the meaning of synthesis phase
   (C) Constructing an intermediate representation
   (D) all of the above

6. _____ are designed to combine the main advantages of interpreters and compilers.
   (A) Optimizing compilers (B) Incremental compilers
   (C) Single pass compilers (D) None of these

Q7 to Q18 carry one mark each

7. Which of the following statements is incorrect?
   (A) Executing a program written in a high – level programming languages is basically a two–steps process i.e. the source program must first be compiled i.e. translated into the object program. Then the resulting object program is loaded into memory and executed.
   (B) Interpreter are often smaller than compilers and facilitate the implementation of complex programming language constructs.
   (C) In most of the command languages, one communicates directly with the operating system with no prior translation at all.
   (D) None of these

8. Select the correct statement:
   (A) A phase is a logically cohesive operation that takes as input one representation of the source program and produces output as an another representation.
   (B) A pass reads the source program or the output of the previous pass makes the transformations specified by its phases, and writes output into an intermediate file, which may then be read by a subsequent pass
   (C) A multipass compiler can be made to use less space than a single pass compiler.
   (D) All of the above

9. When the user types 'while' the structure editor performs which of the following function?
   (i) The editor supplies the matching do
   (ii) Reminds the user that a conditional statement must come between them.
   (iii) Jump from a begin or left parenthesis to its matching and/or right parenthesis
   (iv) If in case any output, it produces same output as analysis phase of a compiler analysis.

   (A) (i), (ii), (iii) (B) (i), (ii), (iii), (iv)
   (C) (ii), (iii) (D) (i), (ii), (iii)

10. Consider the following situation
    Many programming language definitions require a compiler to report an error every time a real number is used to index an array, some language specification may permit some operand corrections.
    The given situation is handled in _____

    (A) Syntax analysis (B) Semantic analysis
    (C) Linear analysis (D) Hierarchical analysis

11. Which of the following statement is true ?
    (i) A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
    (ii) When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table with its corresponding attributes.

    (A) (i) (B) (ii)
    (C) Both (i) and (ii) (D) Neither (i) nor (ii)

12. Which of the following are properties of intermediate code generation ?
    (i) Intermediate code should be easy to produce
    (ii) Intermediate code should be easy to translate into the target program
    (iii) Each instruction has at most one operator in addition to the assignment

    (A) (i), (ii), (iii) (B) (ii), (iii)
    (C) (i), (iii) (D) (ii), (i)

13. Which of the following is a function of preprocessors
    (i) produce input to compilers
    (ii) file inclusion
    (iii) Language extensions
    (iv) Macro processing

    (A) (ii), (iv) (B) (i), (iv), (iii)
    (C) (ii), (iii), (iv) (D) (i), (ii), (iii), (iv)

14. In implementation point of view, front end normally consists of _____
(i) Lexical analysis (ii) Syntax analysis
(iii) Semantic analysis (iv) Intermediate code generation
(A) (i), (ii) (B) (i)
(C) (i), (iii) (D) (i), (ii), (iii), (iv)

15. Match the following :

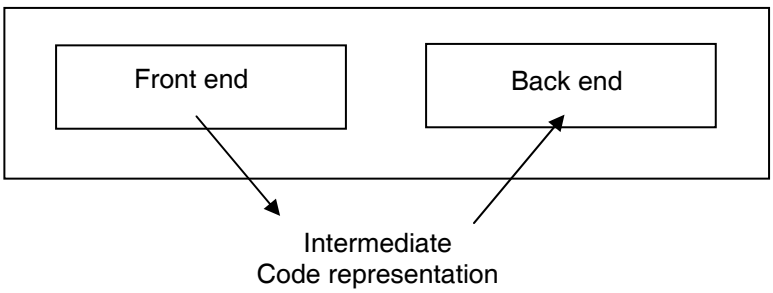
1) A language translator	(i) bridges the specification gap between two PLs.
2) A preprocessor	(ii) bridges an execution gap to the machine language of a computer system.
3) A language migrator	(iii) language processor bridges an execution gap

(A) (1)–iii (2)–ii (3)–i (B) (1)–ii (2)–iii (3)–i
(C) (1)–i (2)–iii (3)–ii (D) None of these

16. A _____ provides general purpose facilities required in most application domains, such a language is independent of specific application domains and results in a large specification gap which has to be bridged by an application designer.
(A) problem oriented language (B) procedure oriented language
(C) object oriented language (D) (A) and (B)

17. Which of the following is incorrect?
i) Lexical constructs do require recursion
ii) Context free grammar and a formalization of recursive rules can be used to guide syntactic analysis.
(A) (i), (ii) (B) (i)
(C) (ii) (D) Neither (i) nor (ii)

18. Consider the following, two pass schematic for language processing.



The desirable properties of intermediate code are
(i) ease of use
(ii) efficient processing algorithm for construction and analyzing the intermediate code
(iii) intermediate code must be compact
(A) (i), (ii) (B) (ii), (iii)
(C) (i), (ii), (iii) (D) (i), (iii)

❑ ❑ ❑ ❑ ❑ ❑

TEST PAPER – 1

Duration : 30 Min. **Max. Marks : 25**

Q1 to Q5 carry one mark each

1. Which of the following is correct statement about structure editors?
   (i) A structure editor takes as input a sequence of commands to build a source program.
   (ii) The structure editor performs the text creation and modification function of an ordinary text editor.
   (iii) The structure editor analyses the program text, putting an appropriate hierarchical structure on the source program.
   (A) (i), (ii), (iii) (B) (i), (iii)
   (C) (ii), (iii) (D) (ii)

2. _____ uses the Hierarchical structure determined by the syntax–analysis phase to identifythe operators and operands of expressions and statements.
   (A) Scanning (B) Semantic analysis
   (C) Linear analysis (D) None of these

3. For achieving aim of optimizing compilers requires _____
   (i) additional analysis of the program structure and the definition of program variable.
   (ii) detect the possibilities of removing loop invariant code out of a loop
   (iii) eliminating redundant expression evaluation
   (A) (i), (iii) (B) (i), (ii)
   (C) (ii), (iii) (D) (i), (ii), (iii)

4. Which of the following statement is correct?
   (A) A cross complier is a compiler which runs on one machine and generates code for another machine.
   (B) The difference between a cross compiler and a normal compiler is in terms of the code generation only.
   (C) Cross –compilers are widely used for mini and micro –compilers
   (D) All of above

5. _____ can detect errors where the characters remaining in the input do not form any token of the language.
   (A) Syntax analysis phase (B) Semantic analysis phase
   (C) Lexical analysis phase (D) None of these

Q6 to Q15 carry two marks each

6. Match the following :

1) Text formatters	i)	translates a predicate containing relational and Boolean operators into commands to search database for records satisfying that predicate.
2) Silicon compilers	ii)	takes input that is a stream of characters most of which is text to be typeset, but some of which includes commands to indicate paragraphs, figures or structures.
3) Query interpreters	iii)	a source language that is similar or identical to conventional programming language.

(A) 1 – i, 2 – iii, 3 – ii (B) 1 – ii, 2 – iii, 3 – i
(C) 1 – iii, 2 – ii, 3 – I (D) 1 – iii, 2 – i, 3 – ii

7. Consider the following declaration

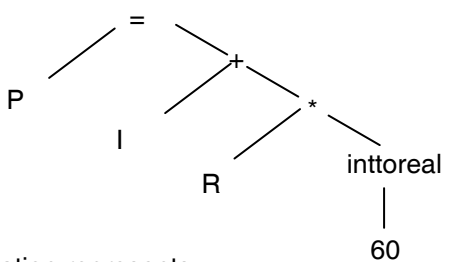
```
for (i  = 0 ;  i <= 5 ; i++)
{       c = c + i ;
}
```

The certain process applied to above code, and due to it we get output as follows
for (i = 0 ; < = 5++) { c }

Then, that process may be _____

(A) Linear analysis (B) Hierarchical analysis
(C) Semantic analysis (D) None of these

8. Consider the following declaration



The above declaration represents :
(A) Hierarchical analysis but it is not valid one
(B) Syntax analysis but is not valid one
(C) Semantic analysis with valid conversion from integer to real
(D) None of these

9. In implementation point of view, back end normally consists of _____
(i) Code generation
(ii) Code optimization
(iii) Error handling
(iv) Symbol table operations.

(A) (ii), (iv) (B) (ii), (iii), (iv)
(C) (i), (ii), (iii), (D) (i), (ii), (iii), (iv)

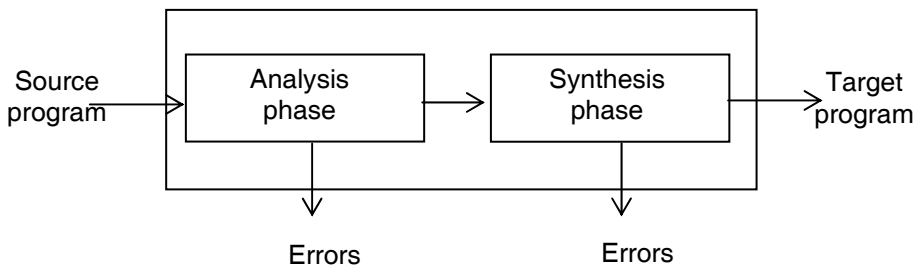
10. Match the following :

1)	Parser generators	(i)	Takes a collection of rules that define the translation of each operations of the intermediate language into the machine language for the target machine.
2)	Syntax directed translation engines.	(ii)	Produce syntax analyzers, normally from input that is based on a context–free grammar .
3)	Scanner generators	(iii)	Produces the collections of routines that walk the parse tree.
4)	Automatic Code generators	(iv)	Automatically generate lexical nalyzers, normally from a specification based on regular expressions.

(A) (1)–ii (2)–iii (3)–iv (4)–i (B) (1)–iii (2)–ii (3)–i (4)–iv
(C) (1)–i (2)–ii (3)–iv (4)–iii (D) (1)–ii (2)–iii (3)–i (4)–iv

11. Consider the statement
percent –profit := (profit *100)/ cost –price
in some programming language.
The which of the following statement is incorrect ?
(A) Lexical analysis identifies :=, * and /as operators, 100 as a constant and the remaining strings as identifiers.
(B) Syntax analysis identifies the statement as an assignment statement with percent–profit as the left hand side and (profit*100) cost–price as the expression on the right hand side.
(C) Semantic analysis determines the meaning of the statement to be the assignment of $\frac{\text{profit} \times 100}{\text{cost} - \text{price}}$ to percent–profit.
(D) None of these

12. Consider a following language processor



The language processing can be performed on a statement –by– statement basis that is, analysis of a source statement can be immediately followed by synthesis of equivalent target statements.

This may not be feasible due to
(i) Forward references
(ii) Issues concerning memory requirements and organization of a language processor

(A) (i) (B) (ii)
(C) Both (i) and (ii) (D) Neither (i) nor (ii)

13. Which of the following statement is incorrect?
(i) A forward reference of a program entity is a reference to the entity which precedes its definition in the program
(ii) While processing a statement containing a forward reference, a language processor does not posses all relevant information concerning the referenced entity. This does not create any difficulties in synthesizing the equivalent target statements.
(iii) The problem created due to process can be solved by postponing the generation of target code until more information concerning the entity becomes available

(A) (i) (B) (ii)
(C) (iii) (D) (i), (iii)

14. Consider the following translation statement

P : = i + r ∗ 60

Then which of the following statement is correct about lexical analysis phase?
(A) When an identifier r is found, then lexical analyzer generates a token say id
(B) When an identifier r is found, then lexical analyzer enters the lexeme r into the symbol table if is not already there.
(C) The lexical value associated with occurrence of id (token generated due to r) points to the symbol table entry for r.
(D) (A) (B) (C)

15. Which of the following situations, the error handler can easily handle?
i) The lexical analyzer may be unable to proceed because the next token in the source program is misspelt.
ii) While entering information into the symbol table, the table management routine may find an identifier that has been declared more than once with contradictory attributes.
iii) The intermediate code generator may detect an operator whose operands have incompatible types.

(A) (i), (ii) (B) (ii), (iii)
(C) (i), (ii), (iii) (D) (i), (iii)

❑ ❑ ❑ ❑ ❑ ❑