Q01.

⇒ Attributes of a good programming language.

↳ Clarity, Simplicity and unity:

→ Should provide clear, simple and unified set of concepts.

→ Syntax of a language affect the ease with which a program may be written, tested and later understood and modified.

↳ Orthogonality:

→ refers to the attribute of being able to combine various features of a language in all possible combinations with every combination being meaningful.

→ Orthogonal languages are easier to learn and programs are easier to write.

↳ Naturalness of the application:

→ It should be possible to translate algorithms directly into appropriate program statements

→ Languages should provide appropriate data structures, operations, control structures and syntax for the problem to be solved.

↳ Support for abstraction:

→ programmer may use 'them' in other parts of the program only knowing their abstract properties without concern for the details of implementation.

↳ Ease of program verification:
→ There are many techniques for verifying that a program correctly performs its required functions such as; formal verification method, desk checking and so on.
→ Simplicity of syntatic and Semantic structure is a primary aspect that tends to simplify program verification.

↳ Programming Environment
→ Presence of an appropriate programming environment may make a technically weak language easier to work with.
→ It includes factors such as; language editors, language debugging facilities, versioning features etc.

↳ Portability of programs:

↳ Cost of use:
(i) Cost of program execution → Repeated execution
(ii) " " " translation → Important to have fast and efficient Compiler
(iii) " " " Creation, testing and use.
(iv) " " " Maintenance

⇒ Differentiate Hardware and firmware Computer. Explain in detail about translators and Software Simulation.

Hardware Computer	Firmware Computer
i) Physical device composed of electronic circuits, chips and other tangible components	i) That combines hardware and software elements with software embedded in the hardware.
ii) Executes instructions and perf-orms computations directly using its physical Components	ii) Executes instructions and performs computations using a combination of hardware and preloaded software.
iii) Limited flexibility and custom-ization	iii) offerers more flexibility and customization through firmware updates.
iv) Upgrades typically involves repla-cing or adding physical compo-nents.	iv) Upgrades can be performed by updating the embedded software without altering the hardware.
v) Retains its functionality even if the power is turned off.	v) Consists of non-volatile memory that retains instructions and settings even if the power is off.
vi) Eg: CPUs, GPUs, RAM Motherboard, HDD	vi) Embedded Systems, BIOS

→ Translators:

↳ Could be designed to translate programs written in high-level language into equivalent programs in the machine-level language of the actual computer

↳ In general, translator accepts program in some source language and produce functionally equivalent programs in another object language as output.

↳ Several Specialized types of translators are assembler, Compiler loader or link editor and preprocessors or ● macroprocessors

① Assembler:

↳ Is a translator whose object language is machine language but whose source language is assembly language.

② Compiler:

↳ Is a translator whose source language is high-level language and whose object language is close to the machine language of an actual computer either being an assembly language or some variety of machine language.

③ Loader or Link editor

↳ Is a translator whose object object language is actual machine code and whose source code is almost identical. It makes a single file from several files of relocatable machine code.

(iii) preprocessor or a Macroprocessor

↳ Is a translator whose language is an extended form of some high-level language whose object language is the standard form of the same language.

→ Translation of a high level source language into executable machine language programs often involves more than one translation step.

→ Moreover, the compilation step may involve a number of steps that progressively translates the program onto various intermediate forms before producing final object program.

→ Software Simulation (Software Interpretation)

↳ Rather than translating the high-level language programs into equivalent program in object language, software simulator executes the input program directly.

↳ SS is a computer whose machine language is the high level language. In such case, we can say that the host computer creates a virtual machine simulating the high-level language.

↳ When the host computer is executing the high-level language program, it is not possible to tell if the program is being directly executed or is converted to low-level machine language.

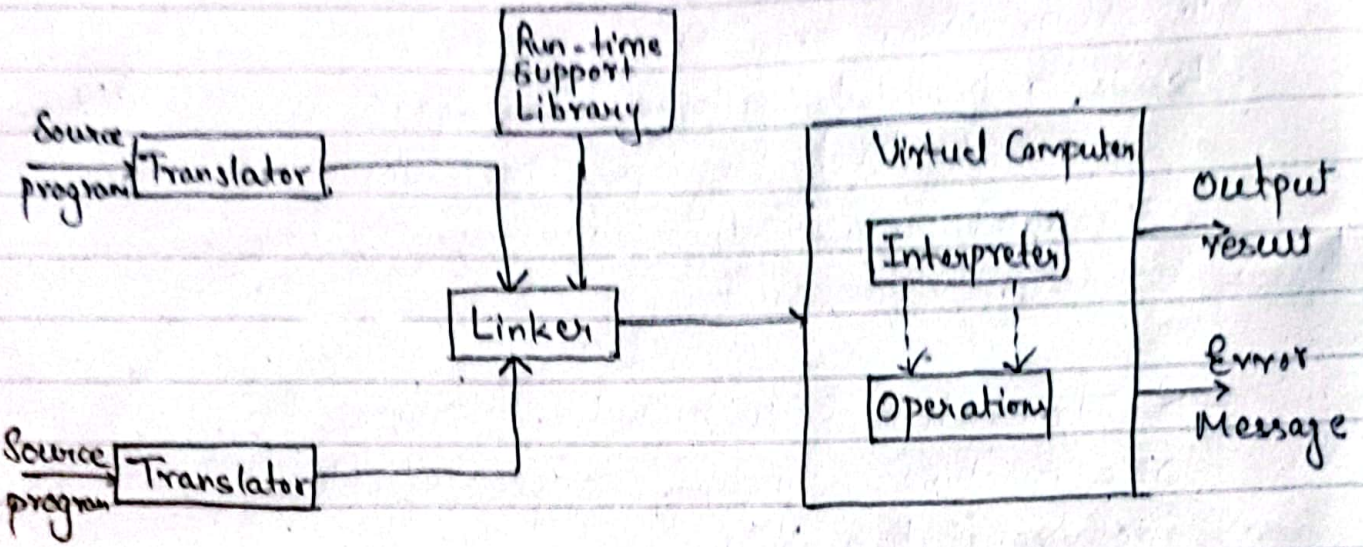


Fig: Structure of typical language implementation.

⇒ Language Standardization

↳ A programming language code can be validated and evaluated using three approaches:

i) Read the definition in the language reference manual typically published by the vendor of your local ~~computer~~ Compiler.

ii) Write a program on your computer to see what happens

iii) Read the definition in the language standard.

→ Option (ii) is probably the most common option

→ (i) is also done but iii) is rarely used.

→ i & ii are tied to a particular implementation, But is that implementation correct?, what if we want to move our 50,000 line program to another computer that has a compiler by a different vendor? will the program compile? If not, why not? Is it legal for vendors to add features to the

language?

→ To address this concern, most language has standard definitions and all the implementation should follow this standard. Standards come in two flavors:

① Proprietary Standards:

→ These are definitions by the company that developed and own the language.

→ These do not work the languages that have become popular and widely used.

② Consensus Standards:

→ These are documents produced by organizations based on an agreement by relevant participants.

→ These are major methods to ensure uniformity among several implementations of a language.

→ Each country typically has one or more organizations assigned with a role of developing standards. Eg: ANSI, IEEE, ISO

→ Three issues needs to be addressed to use this standards effectively:

① Timeliness;

↳ when do we Standardize a language?

↳ One would like to Standardize the language early enough so that there is enough experience in using it but not so late that it encourages many incompatible implementations

ii) Conformance;

↳ A program is conformant if it only uses features defined in the Standard.

↳ A conforming compiler is that, when given a conformant program, produces an executable program that produces the correct output.

iii) Obsolescence:

↳ As our knowledge and experience of programming evolve, new computer architecture require new language featur

↳ One a language is standardized, it seems quaint a few years later.

⇒ Explain different language paradigms in detail

↳ A programming paradigm is a way or style of programming

↳ programming paradigm is defined as a set of principles, ideas, design concept and norms that defines the manner in which the program code is written and organized

↳ Is a framework that defines how the programmer can conceptualize and model complex problems to be solved.

↳ Two main types:

① Imperative:

→ Consists of a set of program Statement and each Statement directs the computer to perform specific task.

→ Execution of the statement is decided by the control flow statements.
→ It focuses on how it is do be done.
→ Eg: Instead of giving friend simply the address, we would give detail step by-step to reach the place.

3 types of imperative:
(i) procedural PL → Consists set of function, each performing a specific task. encap. inher. abstr. Polym
(ii) OOP → Uses program components as objects. Uses concept of
(iii) Database query language → Use to manipulate data stored in database.

(ii) Declarative Style:
→ focuses on the logic of the program and the end result
→ approach is pretty much straight forward and focuses on what is to be done
→ Control flow is not the important element of this paradigm
→ Eg: you simply give your friend the address and let him figure out how to reach home.

2 types:
(i) functional: attempts to solve problems by composing Mathematical functions as program components. foundation for FP is lambda Calculus.
(ii) Logical: Based on logic and control: Logic means facts and rules while control means order of rules.

⇒ Binding and Binding time:

→ Binding of a program element to a particular characteristic or property is simply the choice of the property from a set of possible properties.

→ Time during program formulation or processing when this choice is made is termed as binding time of that property to that element

→ the properties of the program elements are either fixed by the definition of the language or its implementation.

Classes of binding times:

① Execution time (run time):

↳ Many bindings are performed during execution.

Eg:

Binding variable to its values

Binding variable to particular storage location.

Subcategories:

① On entry to a subprogram or block:

In most languages, bindings are restricted to occur only at the time of entry to a block during time of execution

Eg: binding of formal to actual parameters and the binding of formal parameters to storage location may occur only in entry to a block in C & C++

(ii) At arbitrary points during execution.
Some binding may occur at any point during execution of program.
Eg: In binding a variable to values, languages like LISP, ML permit such binding to occur.

(ii) Translation time (compile time):
Sub categories:
(i) Binding chosen by programmer
→ the programmer consciously produces some decisions concerning choices of variable names, types etc. that describes during translation.

(ii) Binding chosen by translator
→ Some bindings are chosen by translator without a direct programmer requirement.
Eg: relative area of a data object in the Storage designated for a phase is usually managed without intervention of programmer

(iii) Binding chosen by loader
→ program generally includes multiple subprograms that should be combined into a single executable program. The translator generally binds variables to addresses within the storage designated to each block.

(iii) Language implementation time.
→ Some aspects of a language ~~time~~ definition may be same for all programs that are run using a particular implementation of a language, but may vary bet' implementat

(iv) Language defn time:
→ Most of the structure of programming language is fixed at the time the language is defined.

⇒ Define Syntax Explain general syntatic criteria in detail. List differed syntatic elements of a programming language

→ Syntax represents fundamental rules of programming language to write functioning code.

→ Syntax of a PL is what a program looks like.

→ Syntax provides significant information needed for a understanding a program and provides much needed information toward translation of source program into object program.

→ Language syntax alone is insufficient to unambiguosly speify the structure of statement.

Criteria:

① Readability:
→ A readable program has structure of the algorithm.
→ Readable program is often said to be self-documenting
→ Readability is enhanced by language features such as:
(i) natural statement formats
(ii) structured formats (statement)
(iii) Naming convention
(iv) noise words
(v) Mnemonic operator Symbols.

(ii) Writability:
→ Same as readability.

(iii) Ease of verifiability:
→ Concept of program correctness or program verifi-cation. The program should be easily verifiable to be mathematically proved correct.

(iv) Ease of translation:
→ program should be easy to translate into executa-ble form.
→ Relates to the needs of the translator that processes the written program.
→ programs becomes harder to translate as the no. of syntatic constructs increase.

(v) Lack of ambiguity:
→ Ambiguity is a central problem in every language design
→ A language defn ideally provides unique meaning for every syntatic construct.
→ Ambiguous construction allows two or more definition

Eg:

→ If bool- exp then st_1 then st_2

If bool- exp then st_1

Here the interpretation is clearly defined for two stmts separately but when two forms are combined then

if $bool\text{-}exp_1$ then if $bool\text{-}exp_2$ then st_1 else st_2

Syntatic elements:

→ Character set : Alphabets of the language
→ Identifiers : name given to variable, functions etc.
→ Operator symbols. (+, -, /, *, %)
→ keywords : identifiers for syntaxes or statements.
→ Nois words : provides improved readability
→ Comments : " " "
→ Spaces :
→ Delimiters and Brackets : used to denote the beginning and end of syntatic constructs.
→ Free - and fixed field formats : Free field if a syntax can be written anywhere on an input line without regarding positi -oning.
→ Expressions : functions that returns a value.
→ Statements : Describe a task to be performed.

⇒ Storage Management. Different elements that required Storage during program execution

→ Storage management is the process of allocating and managing memory resources during program executing
→ It involves organizing, tracking and deallocating memory to efficiently store and retrieve data for execution of program.

Elements:

→ **Variables:**
- ↳ are used to store data temporarily during program execution
- ↳ are typically stored in computer's memory and can be accessed and modified by program

→ **Arrays:**
- ↳ are used to store multiple elements of same types.
- ↳ are typically stored in contiguous memory locations for efficient access and manipulation of elements.

→ **Objects:**
- ↳ Objects require memory allocation to store their member variables and methods.
- ↳ Can be dynamically created and destroyed during program execution

→ **Pointers:**
- ↳ are variable that store memory addresses
- ↳ They allow program to indirectly access & manipulate data by referencing memory locations.

→ **Data Structures:**
- ↳ DS like linked list, trees, queues and stack require memory allocation to store their element & maintain their structure.

→ **Heap and Stack:**
- ↳ The stack is used for local variables, function calls and managing program flow.
- ↳ Heap is used for dynamically allocated memory.

→ **Libraries and frameworks:**
- ↳ when using libraries or frameworks, extra memory may be required their DS, config and runtime states.

⇒ Concept of Heap Storage Management with reference to variable and fixed size element.

↳ Heap Storage management refers to the allocation and deallo-cation of memory on heap, which is a region of memory used for dynamic memory allocation.

↳ It allows program to request memory at run-time for variable sized data structures or objects.

↳ When dealing with elements of fixed size, the concept involves allocating and deallocating memory blocks of a consistent size.

↳ This is useful when the program requires a fixed-size data structure such as array or collection of objects of same size.

Working Mechanism:

① Allocation:

→ The program requests a block of memory from the OS.

→ The size of block is determined by multiplying the fixed size of each element by desired no. of elements

→ Eg: if each element is 4 bytes and we need space for 100 element then the required block size is 400 bytes.

② Deallocation:

→ When program no longer needs the allocated memory, it should deallocate to free up resources.

→ Deallocation involves returning the memory block to OS for future allocations.

(iii) Memory Mgmt functions:

(i) Malloc(): In C, Malloc() is used to allocate block of memory.

(ii) free(): " " free() " " " deallocate block of memory.

⇒ Variable Sized element:

↳ When dealing with variable-sized elements such as arrays that can grow or shrink in size heap storage management is crucial.

(i) Allocation:

↳ Typically Malloc() in C and 'new' in C++ is used to request a block of memory from heap.

↳ The size of block is determined dynamically based on required size of the element.

(ii) Deallocation:

↳ When we are done using variable-size element, we need to deallocate the memory to prevent memory leaks.

↳ To deallocate the memory we use free() in C or delete in C++.

(iii) Resizing:

↳ In some cases variable-sized element may require resizing during program execution.

↳ Eg: an array may need to grow or shrik to accomodate more or fewer elements. In such cases we typically allocate a new memory block, copy the existing data to the new block and deallocate the old block. 'realloc()' can be used to handle resizi

(iv) Fragmentation:

↳ Heap Storage Management for variable-sized element can be prone to fragmentation.

↳ Fragmentation occurs when heap becomes fragmented to small, non-contiguous block of free memory, making it challenging to find a contiguous block of memory for allocation.

↳ To mitigate fragmentation, techniques like memory pooling and custom memory allocators can be used.

⟹ Exception and Exception handler. Explain about system exception and programmer defined exception.

↳ In programming, exception is an event that occurs during the execution of program, which disrupts the normal flow of a program.

↳ Are generally caused by unexpected or erroneous conditions that aries at run time such as invalid inputs division by 0.

↳ When an exception occurs, the program's normal execution is interrupted and an object is created to represent the specific type of exception that has occured.

↳ Object contains:

(i) Nature of the exception

(ii) Error message

(iii) Stack trace showing the sequence of method calls that led to exception

↳ To handle these exceptions or error, PL provides ~~called~~ constructs called exception handlers or catch blocks.

↳ Exception handler is a piece of code designed to handle a particular type of exception.

Steps:

① Throwing an exception
→ when exception condition occure, it throws an exception
→ This is typically done using "throw" statement that automatically throws an exception when it occurs.

② Catching the exception
→ Exception handler is a block of code that is capable of catching and processing a specific exception. It is typically enclosed within "try-catch" block. Where "try" block executes an instruction and if error or exception occurs "catch" block executes.

③ Handling Exception
→ when exception is thrown, the program searches for compatible exception handler within the try-catch block. If a matching handler is found, the exception is caught and the corresponding catch block is executed. Catch block can perform error handling actions such as logging the exception, displaying error message.
→ the program terminates and error is not handled.

④ Propagating Exception:
→ If an exception is thrown but not caught, it propagates up the call stack looking for appropriate exc. handler at higher levels. The process continues until an handler is found if not

⇒ Explain Type equivalence in detail. Types of storage management techniques. Explain

↳ Two types are equivalent if an operand of one type in an expression is substituted for one of the other type without coercion. Also if two operands are of same type also can be called type equivalence.

↳ Type equivalence is an strict form of type compatibility i.e. compatibility without coercion.

Two approaches for defining type equivalence:
① Name type equivalence
② Structure type equivalence

① Name type Eq:
↳ Means that two variables have equivalent types if they are defined either in the same declaration or in declarations that use same type name.
↳ Name type is easy to implement but is more restrictive
Eg: Under a strict interpretation whose type is a subrange of the integers would not be equivalent to an integer type variable
↳ To use name equivalence, all types must have names. Most languages allow user to define types that are anonymous i.e. they do not have names.

(a) Structure type eq:

- is more flexible than name type equivalence but is more difficult to implement
- Under this, only two type names must be compared to determine equivalence.
- The entire structures of the two types must be compared and this ~~takes a lot~~ is not simple

Eg:

```
typedef age ~~a1, a2~~;
typedef roll_no;
age a1, a2;
roll_no r1, r2;
```

Here,

a_1 and a_2 are name type equivalent

a_1 and r_1 are structure type equivalent

Types of Storage Management techniques

(1) Static:

- Is used to handle allocation and deallocation of memory at compile time.
- Size and location of these variables are determined in advanced and remain constant throughout the program's execution
- Is commonly used where variables are typically declared outside function or any block, at the the global scope or as static variables within functions.
- Memory is allocated before the program starts and is deallocated automatically when the program terminates

(ii) Stack:

↳ also known as automatic Storage Management, that uses Stack data Structure to manage memory.

↳ Memory for local variable, function parameters and function call frames is allocated and deallocated automatically as functions are called and return.

↳ follows a LIFO discipline where recently allocated memory is to be the first one to deallocated.

↳ Is an efficient and deterministic approach since memory allocation & deallocation happens automatically and in an predictable manner.

(iii) Heap:

↳ Involves dynamic memory allocation and deallocation at run-time

↳ Allows more flexible memory usage, as memory blocks can be allocated and deallocated as needed during execution.

↳ Is used when there is typically variable-sized data such as dynamic arrays.

↳ Malloc calloc -

⇒ Elementary Data type? Describe briefly approaches to Specification and implementation of Elementary data type.

↳ An elementary data object contains a single data value

↳ A class of such data objects where various operations are performed or defined is called Elementary data type.

→ Data type Specification

⇒ Attributes:

↳ Basic attributes are data types and names

↳ Some of the attributes may be stored in descriptor as part of data object during program execution

↳ others may be used to determine the storage represen-tation of data object

⇒ Values:

↳ Type of data object determines the set of possible values that it may contain

↳ Eg: Integer data type determines a set of integer values that may serve as the values for data object of this type.

↳ Set of value defined by an elementary data type is usually an ordered set with a least value and a greatest value

⇒ Operations:

↳ Set of operations defined for a data type determine how data objects of that type may be manipulated

↳ Operations may be primitive or programmer-defined

↳ primitive operations are specified as part of ~~program definitions~~ language definition

↳ programmer-defined operations are specified in the form of subprograms or method declaration.

↳ Operation that has two input and one output is formed as binary operation

↳ " " one input " " " " " "

" unary operation.

⇒ SubTypes.

↳ if a datatype is part of a larger class, we say it is a subtype of the larger class

↳ the larger class is a supertype of this data type.

→ Data type implementation

↳ implementation consists of a storage representation for data objects and values of that type & a set of algorithms or procedures that define the operations of the type.

⇒ Storage representation:

↳ Storage for elem. data type are strongly influenced by the underlying computer that will execute the program

Eg: Integers are represented as with integer representation for numbers used in underlying hardware.

↳ Two methods to treat attributes

(i) Determined by compilers and not stored in descriptors during execution

(ii) Stored in descriptor as part of data object at run.

↳ Storage representation is usually described in terms of the size of the block.

⇒ Implementation of operations:

↳ Each operations defined on data objects can be implemented in one of these 3 ways.

(i) Directly as hardware operation

(ii) As a procedure or function sub program

(iii) As an inline code sequence.

⇒ Explain type checking. Explain Static and dynamic type Checking.

↳ Type checking is a process performed by compiler or interpreter to verify that the operations performed on data are compatible with the types of data involved.

↳ It ensures that only valid operations are performed and helps catch potential type related errors in a program

Static:

↳ Is a type checking technique that is performed at compile time.

↳ The compiler analyzes the program's source code and checks the type of variables, expressions and function calls throughout the program.

↳ The compiler verifies if the types are used correctly and detects error before the program is executed.

```
Eg:  int a = 5;
     String name = "John";
     int result = a + name   // Error incompatible
                                types.
```

Dynamic:

↳ Is a process performed during runtime.

↳ type checking occurs during program execution, typically when a operation involving data types are encountered.

↳ If the operation is not supported or valid an error is raised which may terminate the program.

Eg:

```
X = 5
name = "John"
result = X + name;
```

⇒ Type conversion and Coercion

↳ Type conversion and coercion are techniques used to change the type of a value from one data type to another.

↳ They are commonly used in PL to obtain compatibility and perform operations on different types of data

↳ Although type conversion and coercion have same function they have two different concepts.

Type Conversion:

↳ also known as type casting is an explicit operation where a value is explicitly converted from one type to another.

↳ The programmer explicitly specifies the desired type conversion using language-specific function.

Eg:

```
let x = 10          → Number
y = String(x)       → String type
```

Coercion:

↳ also known as implicit type conversion, is an automatic conversion that occurs when values of different types are uses together in an expression or operation.

↳ The PL automatically converts one or both of the values to a compatible type to perform operation

Eg:

```
var x = 5
var y = "10"
var result = x + y = "510"
```

⇒ Explain Attribute grammars with Example.

↳ Attribute grammars are used to define semantic model of a programming language.

↳ The idea is to associate a function with each node in the parse tree giving the semantic content of that node.

↳ The grammars are used to pass semantic information around syntax tree.

↳ are created by adding attributes or functions to each rule in a grammar.

↳ Attributes ~~are~~ maybe of two types:

① Inherited:

→ Is a function that relates nonterminal values in a tree with nonterminal values higher up in the tree.

② Synthesized:

→ Relates left-hand side non-terminal to values of right-hand side non-terminals.

→ Consider a grammar:

$E \rightarrow T \mid E+T$

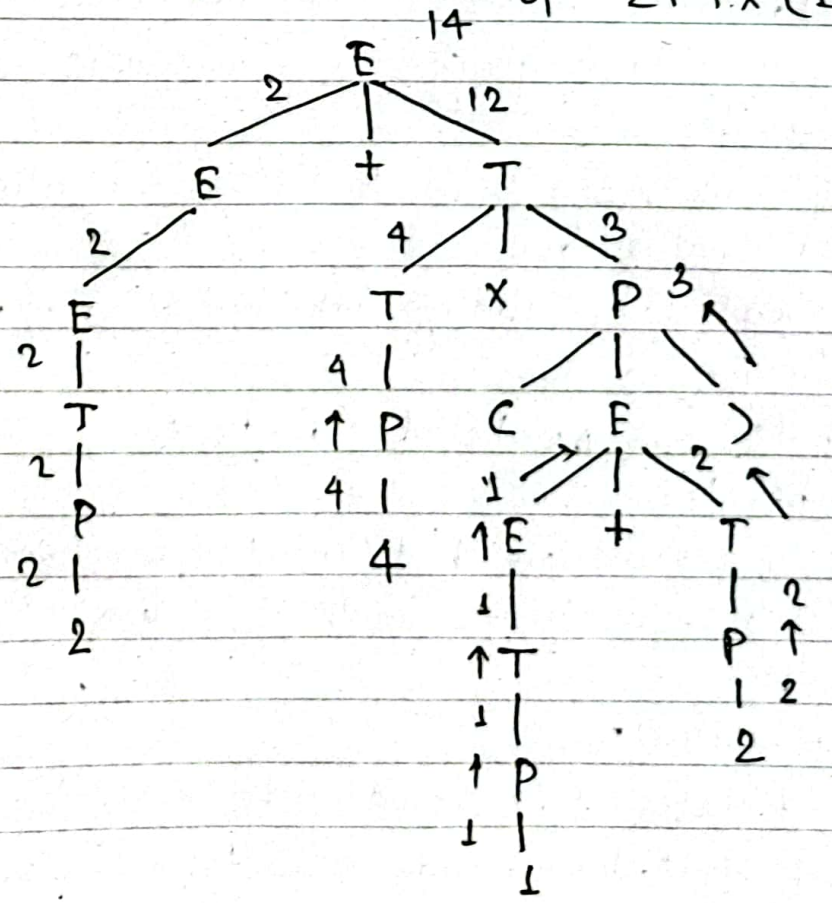
$T \rightarrow P \mid T \times P$

$P \rightarrow I \mid (E)$

we can define the semantics of this language by a set of relationship among non-terminals.

Production	Attribute
$E \rightarrow E+T$	value (E_1) = Value (E_2) + value (T)
$E \rightarrow T$	value (E) = value (T)
$T \rightarrow T \times P$	value (T_1) = value (T_2) × value (P)
$T \rightarrow P$	value (T) = value (P)
$P \rightarrow (E)$	value (P) = value (E)
$P \rightarrow I$	value (P) = value of number I

Attribute tree of $2+4 \times (1+2)$



⇒ Scheduled Sub program and its techniques

↳ A scheduled sub program refers to a function or block that is executed to a predetermined schedule or at a specific time.

↳ it is a way to automate execution of a code in certain interval of time or at a specific point in time.

↳ are often used to automate repeatitive task, perform maintenance operations or handle background processes.

Scheduling techniques

① Scheduling sub program to execute before or after other sub program.

call B after A.

② Scheduling Subprog.. " " after a condition satisfies

Call B If y=7 and z<0

③ " " on basis of a simulated time scale

Call B at time = Current_time +50

④ Scheduling sub program according to a priority designation

Call B with priority 6

⇒ Levels of Sequence control?

↳ Sequence control refers to the various ways in which the flow of execution is controlled within a program

↳ The levels determine the order in which instructions or statements are executed.

① Statement-level:

↳ refers to sequential execution of individual statements within a program.

↳ Statements are executed one after another as they appear in the source code.

↳ Each statement is executed and completed before moving to another

Eg:

St1 → 1st

St2 → 2nd

St3 → 3rd

② Control Structure-level:

↳ involves using control structures such as loops and conditional statements to determine order of execution of blocks of code

Eg:

```
If condition 1:
    St1
    St2
Else:
    St3
    St4
```

(iii) function-level
↳ Control of program flow through function calls and returns.
↳ functions encapsulate blocks of code and can be invoked from other parts of the program

Eg:

function A:
St1
St2

Main program:
St3
St4
Call function A:
St5
St6

⇒ Define referencing environment? Local and non-local referencing environment and its types

↳ Referencing environment also known as environment or a scope is a conceptual structure that defines the visibility and accessibility of variables, functions & entities within a program.
↳ It determines how variables are referenced and resolved during program execution.
↳ referencing environment keeps track of the bindings betn names & corresponding values.

① Local referencing Env.

↳ also refered as local scope or a lexical scope defines the visibility and accessibility of variables within a specific block.

↳ are typically associated with functions, methods or blocks of code

↳ variable declared within a local environment are only accessible within that scope

Eg:

```
function add()
    x = 10
    print(x)
add()
```

② Non-local referencing Env.

↳ also referred as non-local scope or an enclosing scope.

↳ it occurs when an inner scope references variables from outer scope

↳ are created when functions or blocks are nested and the inner scope can access variable from outer scope

Two types of non-local referencing environments:

① Enclosing function Scope:

↳ In this case, a function is nested with another function and the another function can access variable from the function it is enclosed in

Eg:

```
function A():
    x = 10
    function B():
        print(x)
    B();
A();
```

(iii) Modular-level scope:

↳ occurs when a module or file-level ~~or global variable~~ scope is accessed by function defined within that module

Eg:

```
X = 10
function A();
    print(x);
A();
```

⇒ Differentiate formal and actual parameters. Describe each Method of parameter transmission.

Formal Parameters:

↳ also known as formal arguments or parameters, are placeholders or variables defined in the function declaration or definition.

↳ They represent the value that function receives when it is called

Eg:

```
function greet (name)
    print (" Hi I am " + name)
greet("John")
```

Actual Parameters:

↳ also known as actual ~~eg~~ arguments or parameters are the values that are passed to a function when it is called.

↳ They provide the actual data or variables that are used to fulfill functions formal parameters.

Eg:

```
function add (5, 10)
    sum = 5 + 10
add();
```

Methods of parameter transmission

① Pass by value:

↳ the actual parameter is copied and assigned to the formal parameter.

↳ changes made to the formal parameter within the function do not affect the value of actual parameter.

Eg:

```
function increment(x):
    x = x+1
    print(x)
function main():
    num = 5
    increment(num)
    return 0
```

Here the function increases the value of x but does not affect the value of "num" in main function.

② pass by reference:

↳ the actual parameter's memory address is passed to the formal parameter.

↳ Any changes made to the formal parameter will affect the value of actual parameter

```
function incr(int&x)
    x = x+1
    print(x)
function main()
    int num = 5
    incr(num)
    return 0.
```

Here,
'num' is passed to 'incr()' using an 'f'. The function increments the value of 'x' and as 'num' and 'x' refers to same memory location, the value of 'num' also changes.

(ii) Pass by pointer:

↳ the memory address of actual parameter is passed to the formal parameter, which is declared as a pointer

↳ This method allows modifying the value of actual parameter through the pointer.

Eg:

```
function incr (int *ptr)
    *ptr = *ptr + 1
    print (*ptr)
function main ()
    int num = 5
        incr (&num)
        retn 0
```

The function increaments the value of *ptr within the function & as 'ptr' holds the memory address of 'num' the value of 'num' is modified.

⇒ Need of Studying PPL? or PL

(i) To improve ability to develop effective algorithms.
(ii) To improve use of your existing programming language
(iii) To increase your vocabulary of useful programming constructs.
(iv) To allow a better choice of programming language
(v) To make it easier to learn new language
(vi) To make it easier to design new language

⇒ Effects of programming environment on language design.
↳ programming environment has affected language design primarily in two major areas:
(i) feature aiding separate compilation & assembly of progra from components.
(ii) feature aiding program testing and debugging.

(1) Separate Compilation:
↳ This features require the language to be structured so that individual subprogram can be separately compiled and executed and then later merged without change in the final program

↳ Separate compilation is made difficult because when compiling one sub program the compiler may need information from another sub program.

↳ To provide information about separately compiled subprograms. either:

① ~~The may b~~ information may be redeclared within the sub program

or

② it may prescribe a particular order of compilation to require compilation of each sub-program to be preceded by compilation of the specification of all called subprograms.

or

③ it may require presence of a library containing the relevant specification during compilation so that the compiler may retrieve them as needed

↳ Separate compilation also effects language design in the use of shared names. If several groups are writing portions of a large program, it is often difficult to ensure that the names used by each group are distinct.

↳ A common is to find same names of subprograms and other parts of programs during assemb'y of the program.

④ Testing and debugging:

↳ Most of the languages contain some features to aid program testing and debugging. Eg:

① Execution trace features:

→ Many languages provide features that allow particular statements and variables to be tagged for tracing during execution.

(ii) Breakpoints:

↳ programmer can specify points in the program as breakpoint
↳ when a breakpoint is reached during execution of a program, the program interrupts and the control is given to the programmer at a terminal.
↳ The programmer may inspect and modify values and then restart the program from that point.

(iii) Assertions:

↳ Conditional expression inserted as a separate statement in a program.
↳ During program execution, if the condition fails, the execution is interrupted and exception handler is invoked to print a message or take actions.

(iv) Environment framework

↳ Supplies services such as, data repository, GUI, security and communication services.
↳ programs are written to use this service.

(v) Job control and process languages

↳ Job Control Language (JCL) is a scripting language that describe jobs to the OS that runs in IBM mainframe computers.
↳ JCL acts as interface betn your programs and Mainfram OS.
↳ for every job we submit, we must tell OS where to find the input, how to process the output and what to do with the resulting output.

⇒ Significance of grammar in programming language development.

↳ Grammar plays a significant role in development of PL.

↳ It serves as a foundation for defining the syntax and structure of a programming language.

↳ Significance can be summarized as:

① Syntax Definition:

→ It defines valid combination of symbols, keywords and operators that make up language statement and expressions.

→ By adhering to the grammar rules, developers can write code that the compiler or interpreter can understand.

② Parsing & compilation:

→ parsing is the process of analyzing the input code and constructing parse tree based on grammar rules.

→ A well-defined grammar ~~rules~~ ensures that the parsing process is unambiguous.

③ Language consistency:

→ A grammar helps maintain consistency & uniformity in the PL.

→ It ensures code written follows a consistent set of rules, conventions making it easier to understand and maintain

(iv) Error Handling:

↳ Grammar aids in error detection and reporting during the compilation process.

→ when code violates the grammar rules, syntax errors can be identified and appropriate errors message can be generated to assist developers in identifying and resolving the issue.

⇒ Synthesized and inherited attributes with Eg:

Inherited:

↳ are attributes that are passed down from the parent nodes to the child nodes during top-down traversal of Syntax tree.

↳ The value of an inherited attribute at a node is derived from the value of its parent node(s).

Eg:

If_Stmt → If (exp) then stmt else stmt
| if (exp) then Stmt

Using inherited attribute;

if_ Stmt.type = expr.type
expr.type = Bool
Stmt1.type = Stmt2.type = Void

⇒Coroutine:

↳ Co-routine is a PL construct.

↳ Co → Co-operative

routine → functions

So Co-routine can be seen as co-operative functions.

Suppose we have a function that calls another function.

Eg:

```
function(){                          anotherfunction(){
  anotherfunction();  ------------>      St1
}                     <------------      St2
                                         St3
                                       }
```

from the 'function()' when we call 'anotherfunction()' the control doesn't return to 'function()' unless and until 'anotherfunction()' ends.

This behavior is changed using coroutines

Eg.

```
function(){..                        Anotherfunction(){
  Anotherfunction()  ------------>      .1
Okay I'll do 'a'     <------------  // I'm done now, will do later
Hey why don't you continue ---->  [ Ok
                                    1, 2, 3 ]
  Okay I'll do 'b' and 'c' <---  It's your turn now
you now finish rest  ---------->  [ Ok
  of your work                      4, 5, 6 ]
  'd'                                  } // return
  'e'
}
```

⇒ Assignment and initialization with their purpose and example.

↳ Assignment and initialization are concepts used in programming language to assign values to a variable.

Assignment:

↳ process of assigning a value to a variable

↳ involves taking a value and storing it in a variable or overwriting any previous value held by the variable.

↳ Assignment operator is ('=')

purpose:

↳ is to update the value of a variable, allowing it to hold different values at different points in program's execution

↳ ~~vari~~ are commonly used to store and manipulate data, track state and perform calculations within a program.

Initialization:

↳ process of giving initial value to a variable when it is declared.

↳ It set value to a specific variable before it is used or assigned a different value later in program

purpose:

↳ Is to ensure that a variable is well-defined and predictable initial value.

↳ It helps to avoid issues such as accessing uninitialized variables that could lead to unidentified behavior, bugs or unexpected program outcomes.
↳ Is especially important when variables are used in calculations or comparisions immediately after declaration

⇒ Inheritance: derived class, abstract class, object, message and polymorphism

Inheritance:
↳ fundamental concept of OOP. It allows ~~a~~ ~~cat~~ classes to inherit properties and ~~and~~ behaviors from other other classes enabling code reusability, Modularity and creation of hierarchies of related class.

↳ It establishes relationship between classes often refered to as base class and derived class.
↳ Derived class inherits characteristics from the base class and can ~~can~~ extend or modify them as needed

Benefits:
① Code reuse
② Modularity
③ polymorphism
④ Overriding and extension

⇒ Abstract class
↳ Is a class that cannot be instantiated and is typically used as a base for other classes.
↳ It serves as a blue-print or template for derived classes to inherit common properties and behaviors

Key features:

① Cannot be instantiated:
→ Cannot be directly instantiated using "new" keyword.
→ Attempting to create an instance of an abstract class will result in compilation error.

② Can contain abstract Methods:
→ Can declare more than one abstract Methods.

③ May contain concrete Methods

④ Used for creating inheritance Hierarchy

⑤

⇒ Abstract Data types, information hiding, Encapsulation

Abstract Data type:
↳ Abstraction is a view or representation of an entity that includes only the most significant attributes.
↳ It is a weapon against programming complexity.
↳ Two fundamental type of abstraction process abstraction, data abstraction
↳ abstract data type is an enclouse that includes only the data representation of one specific data type & the sub program only provide the operation for that type.

① Built-in types as Abstract Data types:
→ All built-in data types are abstract data types.
→ Eg: floating-point data type provides the means to create variables to store floating-point data.

② User-defined types as ADTs:
↳ Should provide the same characteristics as those of language-defined types such as floating point type
↳ representation of objects of user-defined type is hidden from program units that use the type so only direct operations possible on those objects.

⇒ Information Hiding:
↳ key concept of abstraction is information hiding.
↳ It hides the data representations and only provided operation are available

Benefit:
↳ Application code cannot manipulate the underlying representation of objects directly thus increasing the integrity.
↳ Reduces range of code and number of variables.
↳ name conflicts less likely.

Encapsulation:

↳ When size of the program goes beyond a few thousand lines, two practical problem become evident.

① having such a large program appear as single collection of sub programs that does not impose an adequate level of organization to keep it intellectually manageable

② cost of recompilation after each modification is significant. There is a need to find ways to avoid recompilation of parts of programs that do not change.

↳ The solution is to organize programs into collections of logically related code and data each of which can be compiled without recompilation of rest of the program. (Encapsulation)

↳ Encapsulation are often placed in libraries and made available for reuse in programs.

⇒ What is BNF? why BNF is important while designing PL. what is parse tree?

↳ Two classes of grammar useful in compiler technology

① BNF (Context-free Grammar)

② Eg: Regular Grammars

① (Bakus-Naur form) Grammars: BNF

↳ was developed by John Backus and refined by Peter Naur

→ CFGs and BNF are equivalent in power but differ only in notations.

→ BNF grammar is composed of set of rules which define a programming language.

→ Used to specify the syntatic rules of many computer languages

→ The production in this grammar have a single nonterminal on the LHS.

→ Instead of listing all the productions separately we can combine all those with the same nonterminal symbol.

→ Instead of using $\rightarrow$, in production, we use ::=

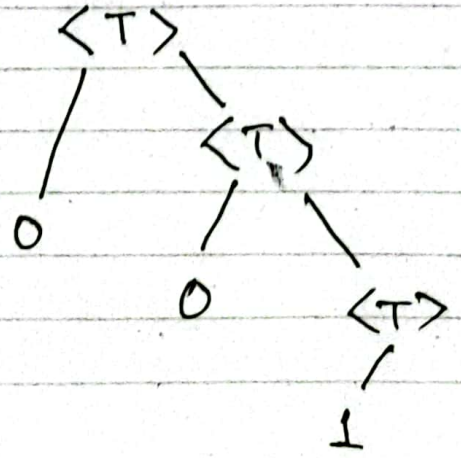
→ we enclose all nonterminal symbols in brackets <>, and we list all the right hand sides of production in the same statement, separating them by bars. (|)

Eg:

$A \rightarrow Aa$, $A \rightarrow a$, and $A \rightarrow AB$ can be combined into

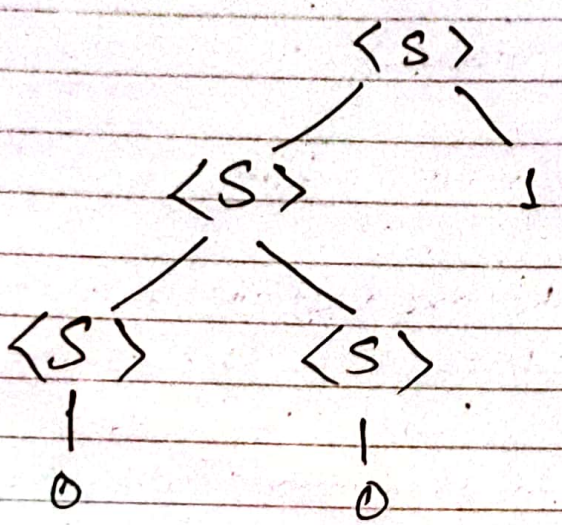
$\langle A \rangle ::= \langle A \rangle a \mid a \mid \langle A \rangle \langle B \rangle$

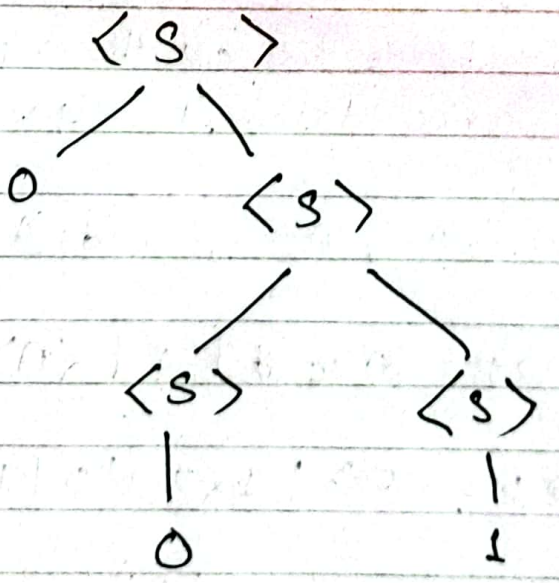
$\langle T \rangle ::= 0\langle T \rangle \mid 1\langle T \rangle \mid 0 \mid 1$



⇒ ambiguity:

⟨S⟩ ::= ⟨S⟩⟨S⟩|0|1





Denotational Semantics

↳ formal way of expressing the semantic definition of a programming language.

↳ The idea is to map every syntatic entity in a PL into some appropriate mathematical entity.

↳ Each Denotational semantic definition consists of

(i) Syntatic categories
(ii) BNF defining ~~syntat~~ structure of syntatic categories
(iii) value domains (the mathematical entities to be mapped)
(iv) Semantic functions
(v) Semantic Equation

In general, there will be one ~~syntatic~~ Semantic function for each syntax category & one semantic Eqn for every production in the syntax grammar

(i) Syntatic Categories

N: Numeral, D: digit

(ii) BNF

Numeral ::= Digit | Numeral digit
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(iii) Value domain

Number = {0, 1, 2, ---, 9}

(iv) Semantic function:

value : Numeral → Number
digit : Digit → Number

(v) Semantic Equation

Value [[N, D]] = plus (times (10, Value [[N]]), digit [[D]])
Value [[D]] = digit [[D]]
digit [[0]] = 0 and so on.

⇒ **Data objects:**

→ is the run-time grouping of one or more pieces of data in a virtual computer.

Two types:

① **programmer defined:**

↳ programmer explicitly creates and manipulates through declarations and statements in the program

② **System defined:**

↳ Virtual computer sets up for housekeeping during program execution and are not directly accessible to the programmer, such as, run-time storage ~~Stacks~~ Stacks, Sub program activation records etc.

↳ Data object is a container for the data value i.e. a place where data is stored and retrieved.

↳ is characterized by set of attributes.

↳ Attributes determine number, type and logical organization of data values.

↳ A data value might be single no. character or possibly a pointer to another data object.

↳ A data object is usually represented as storage in computer memory & data values as pattern of bits.

↳ Some data objects exists at the beginning of program execution while some gets created dynamically during execution.

↳ Some are destroyed during execution; others persist until the program terminates.

→ Elementary data object : if it contains data value that is always manipulated as a uni

→ Data structure : if it is an aggregate of other data objects.

Attributes and bindings of Data object :

① Types : Depends on the type of data values associated with data object.

② Location : Binding to a storage location in memory where it not directly modifiable by programmer but may be changed by storage management routines of Virtual Computer

③ Value : Binding as the result of assignment operation

④ Name : Binding to one or more names which the object may be referenced to during program execut

⑤ Component : Binding of a data object to one or more data objects of which it is a component is often represented by a pointer value and it may be modified by a change in the pointer.