Inference in prepositional Logic

1) Modus Ponens:

$$\frac{P \quad P \rightarrow Q}{Q}$$

2) Modus Tollens:

$$\frac{\sim Q \quad P \rightarrow Q}{\sim P}$$

3) Chain Rule

$$\frac{P \rightarrow Q \quad Q \rightarrow R}{P \rightarrow R}$$

4) Substitution

If S is a valid sentence. S' from S by consistent substitution of preposition in S is also valid.

5) Simplification

From P and Q, we infer P.

$$\frac{P \wedge Q}{P}$$

6) Conjunction rule.

$$\frac{P \quad Q}{\therefore P \wedge Q}$$

7) Transposition

$$\frac{P \rightarrow Q}{\therefore \sim Q \rightarrow \sim P}$$

8) Addition

$$\frac{P}{\therefore P \vee Q}$$

where Q is any other statement.

9) Disjunctive Syllogism

$$\frac{\sim P \quad P \vee Q}{\therefore Q}$$

10) Constructive Dilemma

$$\frac{(P \rightarrow Q) \wedge (R \rightarrow S) \quad P \vee R}{\therefore Q \vee S}$$

11) Destructive Dilemma

$$\frac{(P \rightarrow Q) \wedge (R \rightarrow S) \quad \sim Q \vee \sim S}{\therefore P \vee R}$$

Q. John is intelligent. John is intelligent implies John tops the class.

Sol

$$\frac{J \quad J \rightarrow T}{T}$$

J = John is intelligent
T = John tops the class

Q Apple is fruit and Mushroom is vegetable.

$$\frac{F \wedge V}{F}$$

F = Apples is fruit
V = Mushroom is vegetable.

Q
- John is not a religious person
- John goes to church implies john is religious

Sol

$$\frac{\sim R \qquad C \rightarrow R}{\sim C}$$

Q
- Smith is intelligent student.
- John is a good player.

Sol

$$\frac{I \qquad P}{\therefore I \wedge P}$$

Q David is an opinient boy.

$$\frac{D}{D \vee Q}$$

Q can be any sentence

Q
- Mohit is not laborious boy.
- Mohit is a laborious boy or Sam is honest boy

Sol

$$\frac{\sim M \qquad M \vee S}{S}$$

Q i) Sugar is sweet implies sweet is sugar and acid is sour implies sour is acid

Sol $(S \rightarrow \omega) \wedge (A \rightarrow C)$

Constructive Dilemma.

Q John scored is 85% implies john is intelligent and Smith scored 54% implies he is weak and John is not an intelligent or Smith is not weak

Sol $(P \rightarrow Q) \wedge (R \rightarrow S)$

$\sim Q \vee \sim S \rightarrow P \vee R$

Predicate Logic

It describes property of object in the form $P(x)$ where 'P' is the predicate and 'x' is the variable.

eg) Rohit helps Shyam.

⇒ helps (Rohit, Shyam)

Connectives

$\wedge, \vee, \rightarrow, \leftrightarrow, \sim$

Quantifiers

i) $\exists$

ii) $\forall$

eg) Everyone x is loyal to some y.

⇒ $\forall_x \exists_y \text{loyal}(x, y)$

Q.) i) Marcus was a man

⇒ man(marcus)

ii) Marcus tried to assassinate Ceasar.

⇒ assassinate (Marcus, Ceasar)

iii) All boys have worn Denim jeans.

~~$\forall x : P(x)$~~

$\forall_x$: boys $(x) \wedge$ worn (Denim, x)

iv) Some girls out of all wear black.

Q. John likes all kind of food.

Sol $\forall x : food(x) \longrightarrow likes(John, x)$

$\forall x : f(x) \longrightarrow l(J, x)$

$\forall x : \sim f(x) \vee l(J, x)$

$\Rightarrow \sim f(x) \vee l(J, x)$

Resolution: (for propositional logic)

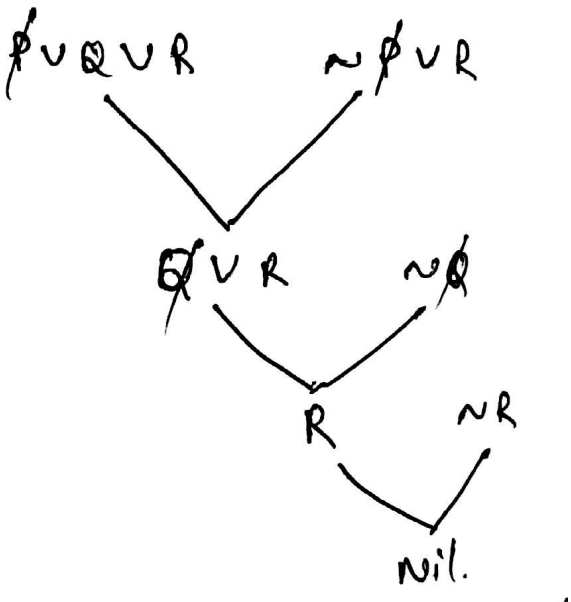
It is an iteration process at each steps two clauses called parent clause are compared, yielding a new clause ~~can be~~ that has been inferred from them. New Clause represent ways that two parent clause interact with each other.

eg) $A = P \vee Q \vee R$

$B = \sim P \vee R$

$C = \sim Q$

$D = \sim R$



Q. P

$\sim P \vee \sim Q \vee R$

$\sim S \vee Q$

$\sim T \vee Q$

T

Q. P

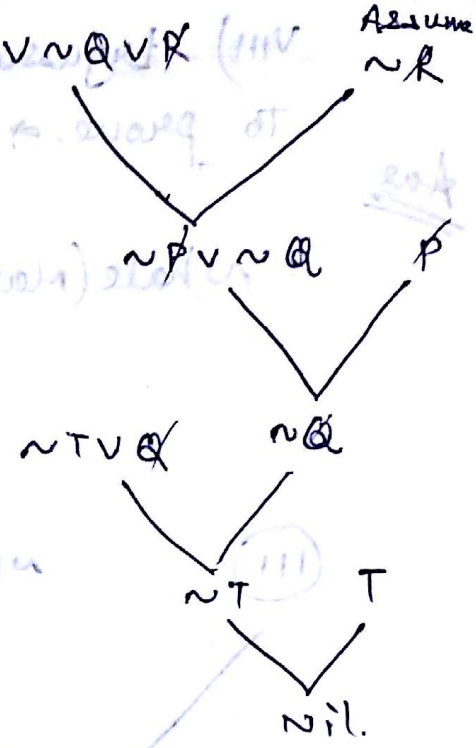
$\sim P \vee \sim Q \vee R$

$\sim S \vee Q$

$\sim T \vee Q$

T

So



Algorithm: (Resolution)

i) Convert all proposition into ~~clause~~ closure form

ii) Negate the proposition and convert the result to clause form

iii) Repeat for contradiction,

- Select any two parent clauses.
- Resolve them to contradiction
- If resolution is nil then stop.

Resolution: (for predicate logic)

i) man(Marcus)

ii) Pompeian(Marcus)

iii) ~Pompeian(x_1) V Roman(x_1)

iv) Ruler(ceasor)

v) ~Roman(x_2) V loyal to(x_2, ceasor) V hate(x_2, ceasor)

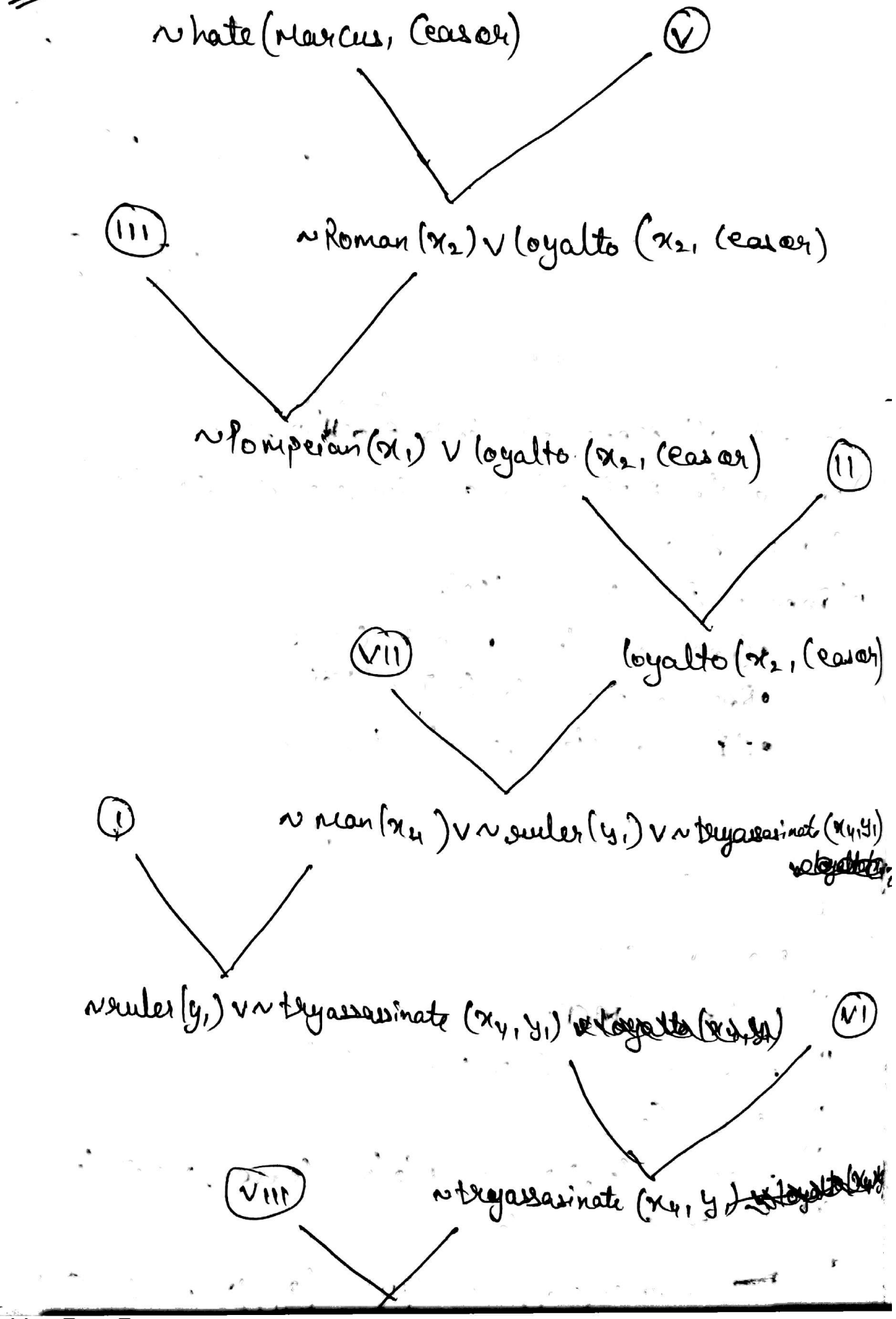
vi) loyalto(x_3, ff(x_3))

vii) ~man(x_4) V ~rules(y_1) V ~try assassinate(x_4, y_1) V

VIII) tryassassinate (Marcus, Ceasor)

To prove. ⇒ hate (Marcus, Ceasor)

Ans



Q

1.) ~Burn (x) V ~Woman (x) V Witch (x)

2.) ~Wood (x) V Burns (x)

3.) ~float (x) V Wood (x)

4.) float (x) V ~weight (x,y) V ~float (y)

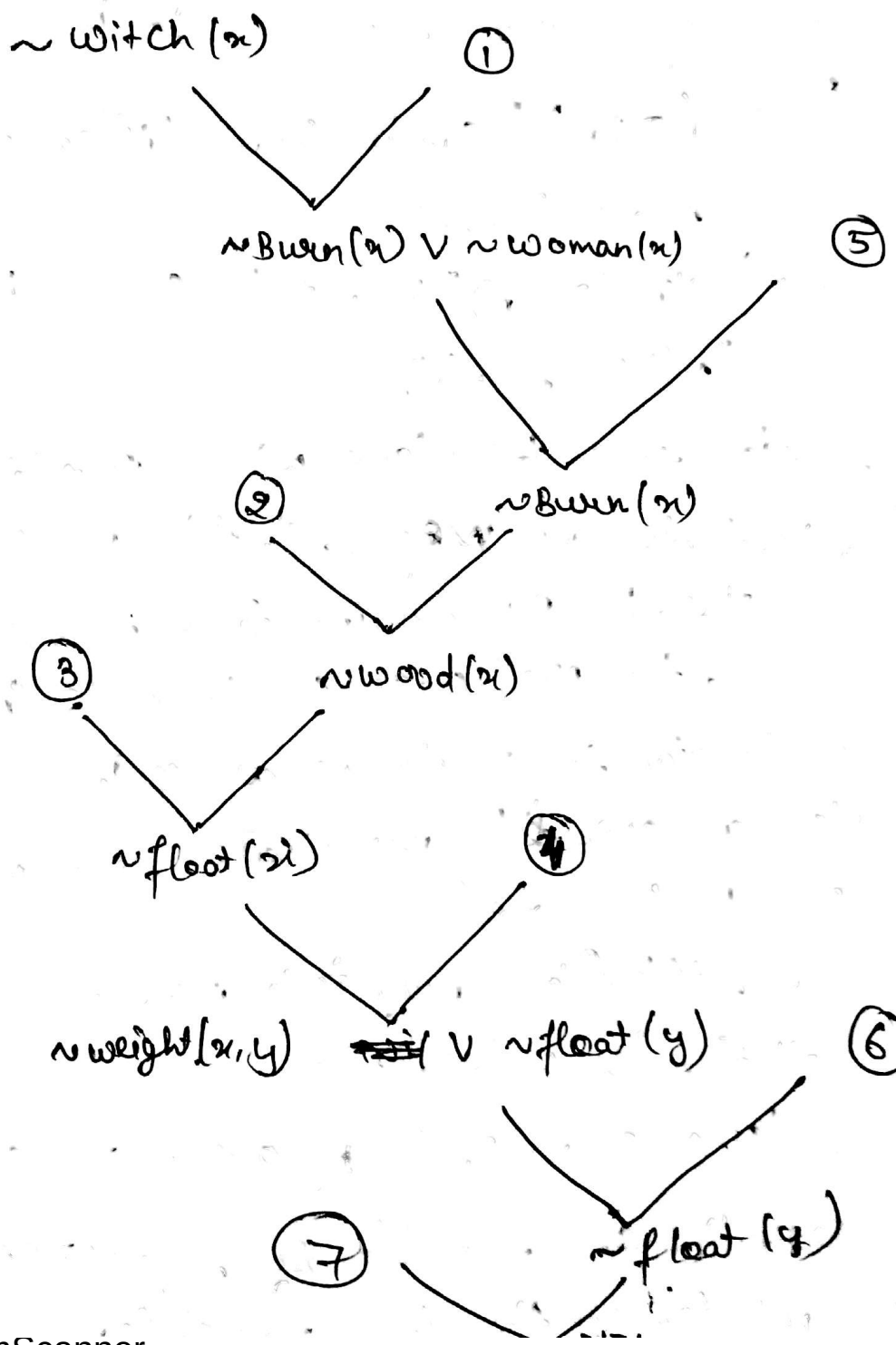
5.) Woman (x)

6.) weight (x, y)

7.) float (y)

To prove ≠ Witch (x)

80|



Unification:

Resolution is easily performed in proposition logic since it is easy to find out two literals that can not be true simultaneously.

eg) 'p' and '~p' can easily be contradicted

But in predicate logic, it is complicated since arguement also need to be considered

eg) "likes (John)" can be contradicted with "~likes (John)"

but "likes (John)" can not be contradicted with "~likes (Sam)".

Unification is a technique where predicated having variables are merged. In this process, we check whether by performing some substitution of variables, the predicates can be made identical the substitution for variables are try to generate identical and opposite predicate sets such that they can be combined.

Three substitution may take place-

1. Substitution by a constant.
2. Substitution of variable by another variable.
3. Substitution of a variable by a function that does not contain the same variable.

Q for the following sequence, unify the two clauses.

$P(x, f(z), mary)$ and $\sim P(x, y, z)$

Sol Substitute "z with "mary"

⇒ $z \uparrow mary$

$P(x, f(mary), mary)$ $\sim P(x, y, mary)$

Substitute "y" with "f(mary)"

⇒ $y \uparrow f(mary)$

$P(x, f(mary), mary)$ $\sim P(x, f(mary), mary)$

Q Perform Unification on the following sequence:-

① Knows (John, x) , ~Knows (John, Jane)

② Knows (John, x) , ~Knows (y, Bill)

③ Knows (John, x) , ~Knows (y, Mother (y))

④ Knows (John, x) , ~Knows (x, Eliza)

Sol ① Substitute $x \uparrow Jane$

Knows (John, Jane) , ~Knows (John, Jane)

② Substitute $x \uparrow bill$ & $y \uparrow John$

Knows (John, bill) , ~Knows (John, bill)

③ Substitute $y \uparrow John$

Knows (John, x) , ~Knows (John, Mother (John))

Substitute $x \uparrow$ Mother (John)

Knows (John, mother (John)) , ~Knows (John, mother (John))

(4) Replacement is not possible.

Algorithm

(1) If L_1 or L_2 ~~both~~ are both variables or constants then —

(a) If L_1 and L_2 are identical then return Nil.

(b) Else if L_1 is a variable then if L_1 occurs in L_2 then return fail else return L_2/L_1

(c) Else if L_2 is a variable then if L_2 occurs in L_1 then return fail else return L_1/L_2.

(d) Else return fail.

(2) If initial predicate symbol in L_1 & L_2 are not identical then return fail.

(3) If L_1 and L_2 have different no. of arguements then return fail.

(4) Substitute

(5) Stop.

Skolemization

It is a process of managing existential quantifi
Every existential quantifier whose arguments a
universal variable must be turned into a fun
term from quantifiers, whose scope include
that of existential quantifiers.

eg) $\forall x : \exists y$: fatherof (y, x)

$\Downarrow$

$\forall x$: fatherof $(s(x), x)$

① Determine which variables are existantial and which are universal.

② Replace each existantially quantified variables by a function. The arguements of that function are universally quantified variables

③ If two different universally quantified variables have same name, rename one of them.

④ Replace each universally quantified variables 'v' by simply 'v'

eg) $\forall x : V(x)$

$\Rightarrow v$

Syntax :

It is sequence of symbols that make up Valid programs

eg) `x=y+z;` it is a valid syntax in C.

`xy+-;` it is an invalid syntax in C.

By this example, we can say that syntax is way of writting expression, literals & identifi

the goal of Syntax :-

① How language designer specify the syntax.

② How can programmer learn the syntax.

③ How the compiler recognizes the syntax.

Elements of Syntax:

① **character set** : A-Z, a-z, 0-9, #, ;, &, $, etc

② **Identifiers**: Identifiers are named location in which we store certain values.

③ **operator**: +, -, *, &&, ||, ?, % . . .

④ **Keywords**: these are reserved words that have certain meaning.

⑤ **noise words**: For improving the readability, languages like COBOL have some unwanted words termed as noise words.

⑥ **Comments**: In order to understand the meaning of functions & statements, we are having concepts of comment.

⑦ **Blanks**: Blanks give proper readibility in programming language.

⑧ **Delimiters**: "{" ... "}"

⑨ **Free & fixed field formats**: 'C' language is having free format we can type our program anywhere on the console window whereas punch cards had fixed field formats.

(10) Expressions : $2+3*4 \Rightarrow 14$.

(11) Statements : Simple statements that follows the syntax.

Translators :

Translators are program that read the program in one language ie., source language and convert it into an equivalent program called as target language.

Types of translators :

(1) Assembler : It translates assembly language code into machine language.

(2) Compiler : It translates an input program of HLL into object program.

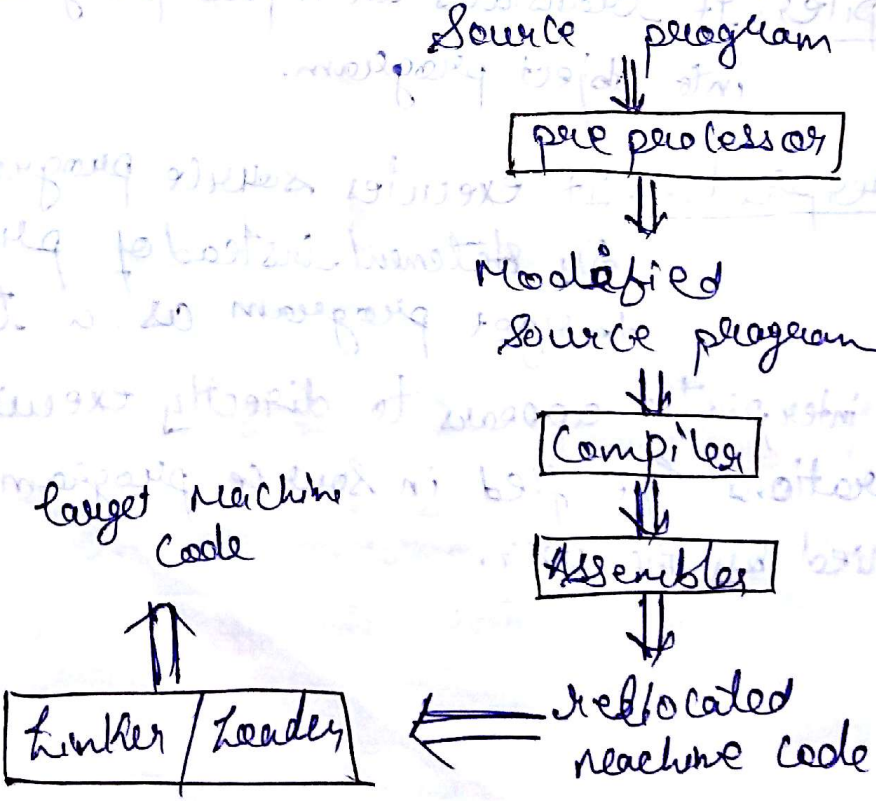
(3) Interpreter : It executes source program statement by statement instead of producing a target program as a translation

an interpreter appears to directly execute the operations specified in source program on input supplied by the user.

(2) Linker: It is a software which combines various pieces of code and data together to form a single executable code that can be loaded in memory. Large programs are often compiled in pieces so, the relocatable machine code may have to be linked together with other relocatable object files and library files into code that actually runs on the machine. Linker resolves external memory addresses where code in one file may refer to a location in another file.

(3) Loader: It loads the binary code in memory ready for execution they are responsible for: locating program in main memory every time it is being executed.

Overall language processing System.



Pre processor:

Source program can be divided into small module stored in seperate files. The task of collecting the source program is entrusted to the pre processor. It may also expand shortcuts called macros into source language statements.

Syntatic Criteria or use of Syntax:::

- **Syntax should be easily readable**
  - → there should be a provision of comments to enhance readability.
  - → provision of long identifier names.
  - → Name Constants, Constraints.
  - → Clearly understood control statements.
- **Writability**
  - → Redundancy should be removed. eg: global variable, functions, etc.
- **No ambiguity:**
- **verifiable:**
- **Ease of translation:** Translators should be able to translate the language easily.

Stages in Translation

Source Code

eg) position = initial + rate * 60

↓

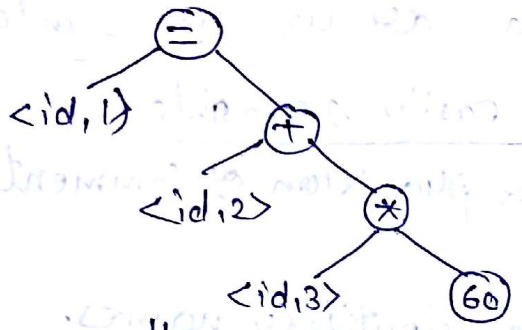
Lexical analyzer

↓

<id,1> = <id,2> + <id,3> * 60

↓

Syntax analyzer



↓

Semantic analyzer

↓

Intermediate code generator

↓

$t_1 = 60$

$t_2 = \langle id,3 \rangle * t_1$

$t_3 = \langle id,2 \rangle + t_2$

~~$t_4 = \langle id,1 \rangle +$~~

$\langle id,1 \rangle = t_3$

↓

Code optimization

↓

$t_1 = id3 * 60$

$id_1 = id_2 + t_1$

↓

Code generation

```
LDF   R2, id3
MULF  R2,R2,#60
LDF   R1, id2
ADDF  R1,R1, R2
STF   id1, R1
```

Stages of Translation

① **Analysis**: Analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of source program. If the analysis part detect that the source program is either syntactically ill or unsound then it must provide informative message so that user can take corrective action.
Analysis part also collects information about source program and stores it in a data structure called Symbol table, which is passed along with the intermediate representation to synthesis part.

② **Synthesis phase**: It constructs the desired target program from the intermediate representation and information in Symbol table.

① Analysis Steps

i) **Lexical analyzer**: It reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes or tokens. Output of token is

<token_name, attribute value>.

In token, the first component token_name is an abstract symbol ie, used during analysis and second component attribute value points to an entry in Symbol table for this token.

eg) <id, 1> in which id is an abstract symbol & '1' points to symbol table entry for position.

ii) Syntax Analyzer: Parser uses the first component of the the tok produced by the lexical analyzer. It create a tree like intermediate representation that depicts the grammatical structure of token stream.

iii) Symantic analizer: It uses the syntax tree and the information in symbol table to check the source program fo symantic consistency with the language definition. It also does type checking, expansi of macros, etc. where the compiler checks tha each operator has matching operands.

eg) Each array index has integer type value.

(2) Synthetic Steps

iv) Intermediate Code generation: It this a 3-addr code is generated, which consists of a sequence of assembly like instruction with three operands per instru each operand can act like a register. Compiler must generate a temporary name to hold the value computed by a 3 address instruct

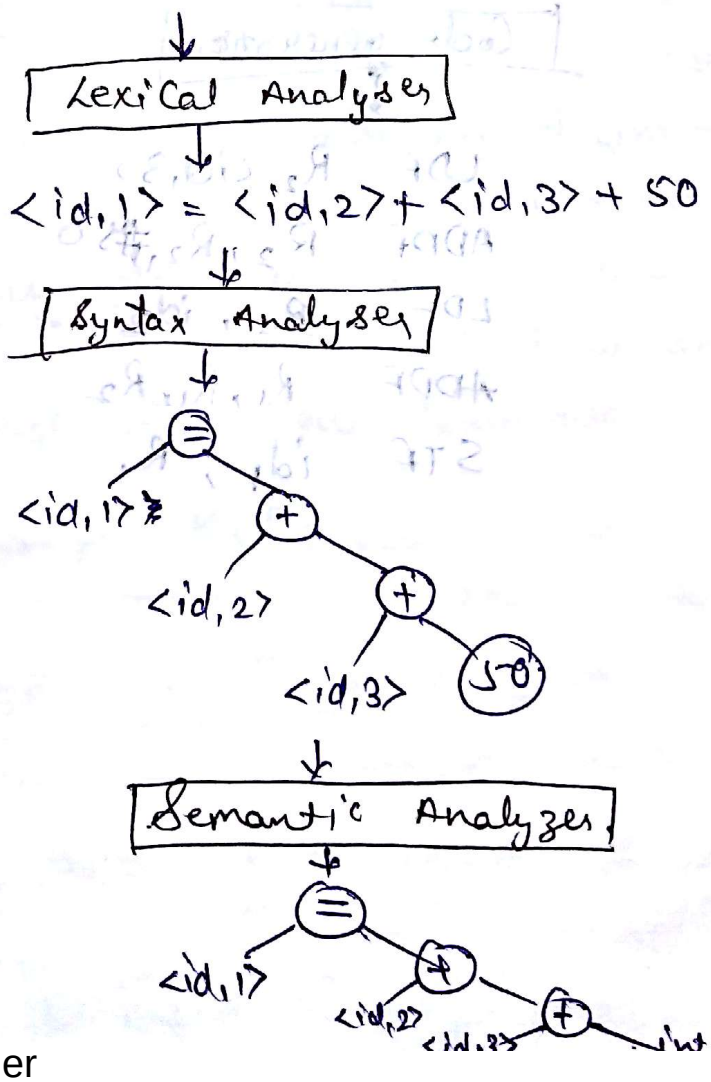
v) Code optimization: Machine independent code optimization phase attempts to improve the intermediate code so that better [illegible]

vi) **Code generation**: The first operand of each instruction specifies a destinat

the 'f' in each instruction tells that it deals with floating point numbers. the code loads the content of address <id,3> into register R_2, then multiplies it with floating point constant 60. "#" signifies that 60 is a intermediate constant. third instruction moves ~~"id"~~ <id,2> into register R_1 and fourth adds it to the value previously computed in register R_2. finally the value in register R_1 is stored into the address of <id,1> so, that code correctly implements assignment statement.

Q

$c = a + b + 50.$



↓

Intermediate Code Generation

↓

$t_1 = 50$

$t_2 = \langle id, 3 \rangle + t_1$

$t_3 = \langle id, 2 \rangle + t_2$

$\langle id, 1 \rangle = t_3$

↓

code optimization

↓

$t_1 = \langle id, 3 \rangle + 50$

~~$t_2 = \langle id, 2 \rangle + t_1$~~

$t_2 = \langle id, 2 \rangle + t_1$

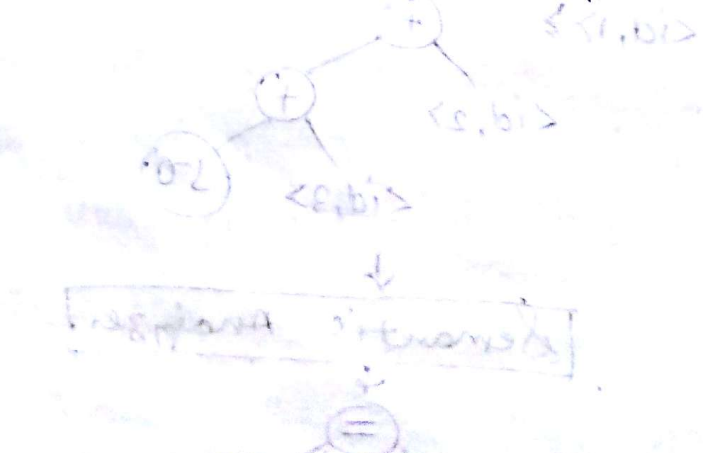
$\langle id, 1 \rangle = t_2$

↓

Code generation

↓

```
LDF   R2, <id,3>
ADDF  R2, R2, #50
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```



Elements of Syntax:

① **Character set**: The choice of character set is the first step to design the language. Characters set may be combination of A:Z, a-z, 0-9, ~, —, !

② **Identifiers**: String of letters or digits beginning with a letter or '_' is widely accepted. It is used for variable name. It may be of fixed or flexible length

③ **Operator symbol**: Generally, +, -, /, * are used for basic operations in many languages but in some languages like LISP we use "Plus".

④ **Keywords & reserved words**: Keyword is a word or identifier whose meaning is already defined in the lang., we can not use them as variable names

⑤ **Comments**: It makes the program readable and to facilitate documentation. A language may allow Comments

⑥ **Space or Blanks**: Blank spaces are used according to the need of language. It introduces readibility in the program

⑦ **Delimiters & brackets**: A delimiter is a syntactic element used to mark the beginning or end of

the statement or block. Brackets are paired delimiters.

BINDING

It is association of a program element to a particular property. It is choice of property from a set of property. The time during which this formulation takes place is called binding time.

Types of binding Or Stages of binding

(1) At Compile time : Some binding are done at Compile time and it remains static throughout program execution.

(2) Choosen by the programmer

i) Variable names of particular data type
eg) int a

ii) particular variable name Or marks
eg) marks = 0

iii) Structure
eg) struct emp {
int eid, age;
char name;
};

(3) Choosen by the translator

i) choice of array storage
ii) Relative location of object file
iii) Demand paging

c) Chosen by the loader

i) Program consists of several sub-progs that must be merged into a single executable program. Loader binds variables to addresses within the storage designated for each sub prog. However, this storage must be allocated actual addresses within physical computer that will execute the program.

(2) At execution time : Many bindings are performed during program execution

(a) At arbitrary points

i) Binding of variable to values at execution time
The source code add any two numbers. User enters first no. and then 2nd nos. Both the nos. are binded together by the '+' operator at execution time. ~~On entry to a~~

ii) On entry to a sub program or block.

eg) Binding of formal & actual parameters.

```
int sum(int a, int b)
{
    int c;
    c = a+b;
    return c;
}
```

```
int sum(int *a, int *b)
{
    int c;
    c = a+b;
    return c;
}
```

Categories of binding :

① Early binding or static binding.

② Late binding or dynamic binding.

1) Early binding or static binding.

Most of the bindings are made during translation time so it has early binding. It is inflexible because most bindings are performed at translation time, before the data are known.

It is difficult in C language to write programs that can adapt to a variety of different data dependent situations at execution time.

eg) Size of strings and type of variables must be fixed at translation time.

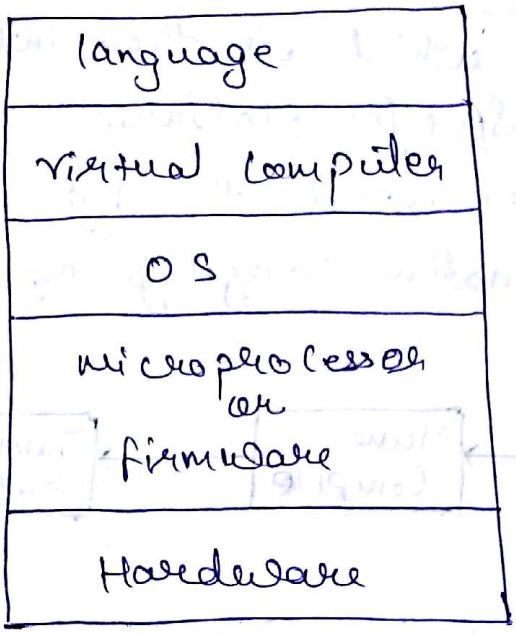
2) Late binding or Dynamic binding

Most of the binding are done until execution ti so it is known as late binding. Binding may be delayed during execution until the input data has b examined and the appropriate bindings for particular input da are determined

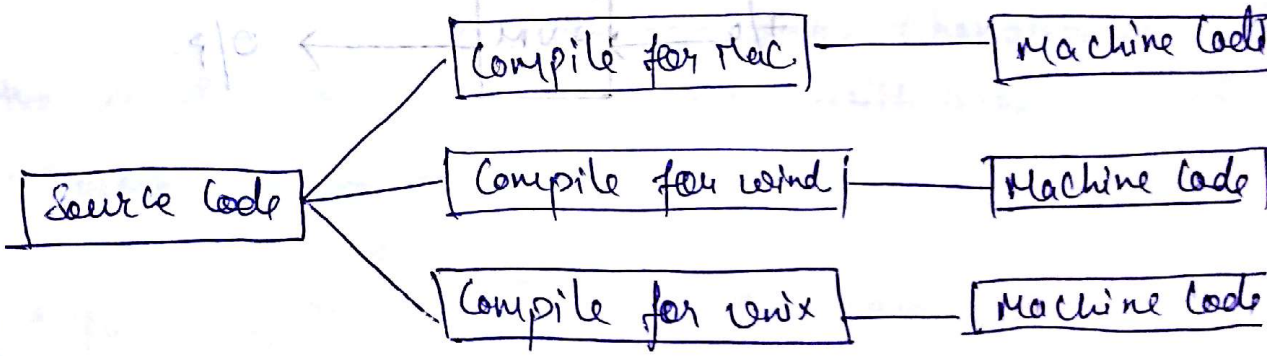
* C-language supports early binding whereas Lisp language support late binding.
* Early binding is efficient but not flexible
* Late binding is flexible but not efficient.

Virtual Machine

It is an abstract machine designed to be implemented on top of existing processes. It hides the underlying operating system.



Traditionally, for commercial software like MS-word had to be written almost independently for all different platforms



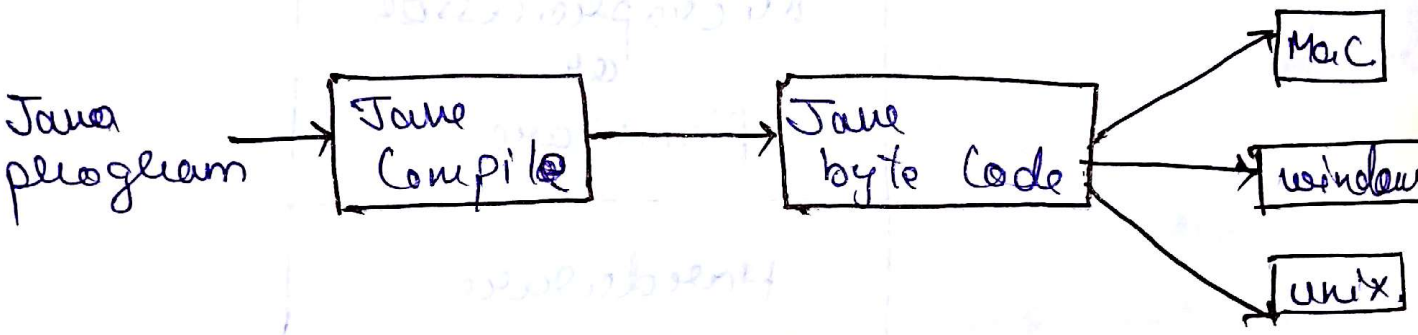
Java Virtual Machine

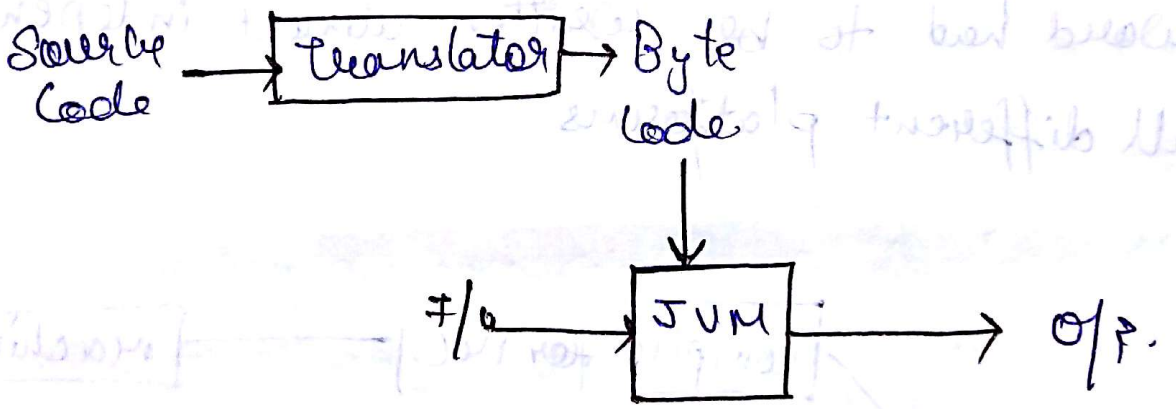
Java is platform independent using bit code. Source code after compilation gets converted into machine code. Machine code depends upon platform it is executed on

for different platform different machine code is produces which is called native executable code.

Java doesn't produce native executable code program written in Java are compiled into byte code which is then interpreted by JVM for a specific platform.

Interpreter reads the byte code and translate in into native language of host machine.





Language:

Medium of communication.

- Natural language (comm. b/w living beings)
- Programming language (used for coding)

Q what is programming language?

Programming Language is a mechanism used to instruct the computer for controlling the behaviour of the machine.

Role of Programming language

making computing convenience for people.
making efficient use of computing machine

Usage of programming language:

1) Programming Methods: Language designs are reflecting the changing need of the society and we are upcoming with good methods for writing large & complex programs.

2) Computer Capabilities: Computers have evolved from vaccum tubes to super-computer Operating System have also been inserted into them. This all has become just because of different programming language.

(3) Applications: Computers are being used in various fields and programming languages making it possible eg) Google.

(4) Implementation methods: The development of better implementation methods has affected the choice of features to include in a new language ~~design~~. designs.

(5) Theoretical Studies: By knowing the strength and weakness of language features, it influences the inclusion of these features in new language designs.

(6) Standardization: Programs can be transported from one computer to another

Q. Why do we study programming language?

Ans (1) to improve your ability to develop effective algorithms

Many languages provides features that when used properly may benefit the programmer but when used improperly may waste large amount of computer time

(2) to improve your use of your existing programming language:

By understanding how features in your language are implemented, you greatly increase your

ability to write efficient program.

eg) Arrays.

(3) To increase your vocabulary of useful programming Construct

⇒ language serves both as an aid and a constraints of thinking. By studying the constructs provided by a wide range of languages, a programmer increases his programming vocabulary.

(4) Allow better choice of programming language

⇒ Applications requiring numerical calculation can be ~~be~~ C-language, fortran, ADA, AI problems can be written in LISP, ML, prolog~~ramm~~. Internet application can be designed in perl or java

(5) to make it easier to learn a new language

Through knowledge of variety of programming language ~~and~~ constructs and implementation techniques allows the programmer to learn programming languages more easily when the need arises.

(6) To make it easier to design a new language

few programmers can be language developers or designers, because more you gain knowledge of different languages, the better understanding

of programming concepts you get.

Evolution of Computers

Year	Hardware	Method	Language
	I generation		
1951	Vaccum tubes	Manual	Machine language
	II generation		
1956	Transistors	Magnetic tape	Fortran, LISP
1961	Transistors	Magnetic disk.	COBOL, SNOBOL, JOVIAL
	III generation		
1966	Integrated Circuits.	Multiprogramming	BASIC, APL, SIMULA
	IV generation		
1971	LSI, VLSI	Structured programming	PASCAL, Prolog, C
1976	Distributed Computing	Real-time System	ML, Smalltalk
1981	Personal Computers	OOPS	TURBO, PASCAL

1986	Internet	client server computing	C++
	V generation		
1991	MPP (massive Parallel processing)	open systems	HTML
1996	world wide web (www)	E-commerce	Java, XML

Attributes of Good program:

1. Quality
2. Time
3. Cost

How software is build?

~~1) Requirement Phase:~~

* **Defining the system:**
  Eliciting Requirement from Customers.
* **Analyzing the defined system:**
  Study the defined system
* **Detailed system specification:**
  defining the components of system model based on the analysis.
* **Designing the system:**
  make DFD (data flow diagram), flowchart, ER-diagram.

* Coding: Write the code in Specific language.
* Testing: Determine the test cases and execute the program with the intent of finding the errors.
* Validation: Resulting solution needs to be validated against requirements given by the user.

Attributes of Good Language

* Clarity, Simplicity & Unity

Language is a means of expressing algorithm. It should provide clear simple & unified set of concepts for developing algorithms. Syntax must be simple so that programs may be written, tested and later understood and could be modified.
Readability of program in a language is also a central issue.

* Orthogonality

It refers to the attributes of being able to combine various features of a language in all possible combination which is meaningful. By orthogonality, it is easier to learn a language.
eg: Array and structures in C-language.

* Naturalness for the application

It should be possible to translate a program design directly into appropriate program statements that reflect the structure of the algorithm.

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

* Support for abstraction

Property of hiding the details. Language should allow data structures, data types and operations to be defined and maintain as self contained abstraction. C-language does not provide abstraction but C++ provides abstraction.

* Ease of program verification

A program may be proved correct by a formal verification method. It may be informally proved correct by desk checking. It may be tested by executed it with test input data & checking the output results against the specification.

* programming environment

Language should have a reliable and efficient environment. Special editors and testing packages speed the creation & testing of programs.

* Portability

It aims at transportability of the program from the computer on which they are developed to another computer.

* Cost

@ there must be efficient compiler, efficient register for allocation of program execution.

(b) Translation: Compiler should be fast and efficient.

(c) Creation, testing & use: Program must be design, coded and modify and used with a minimum investment of programmer time & energy.

(d) Maintainable: A language must be easy for a programmer to be repeatedly modified, repaired & extented by different programmers over a period of many years.

Programming Paradigms

: It defines methodologies of designing & implementing programs using key features and building blocks of a programming language.

1) Imperative or Procedural: Program consists of a sequence of statements and computer execute instructions sequentially. Execution of each statement causes the computer to change the value of one or more locations in its memory and to enter a new state. Memory consists of collection of cells and execution of statement can be represented as accessing memory location, combining these values in some way and storing the result in new location. Program development consists of building the successive machine takes needed to arive at the solution.

eg) C, C++, FORTRAN, ADA, ALGOL, PASCAL

2) Applicative or functional : In this, we consider the function that must be applied to initial machine state by accessing the initial set of variables & combining them in specific ways to get the answer. Program development proceeds by developing functions from previously developed functions to ~~more~~ build more complex function that manipulate the initial set of data until the final function can be used to compute an answer from initial data. Rather than looking at successive machine state of a computation, we consider the successive functional transformations that we must make on data to arrive at our answer. Once we have our final answer we apply this to the initial data to arrive at a result.

eg) $function_n(.....function_2(function_1(data))$

Languages are- LISP and ML.

LISP example:

1) append (L, [], L)

2) append (L[H|T], [H|NT])

3) Rule based programming: It executes by checking for the presence of certain enabling condition and when present executing an appropriate action. Execution is similar to imperative language except that statement are not sequential. Enabling condition determine

(Logic programming) (Declarative prog.)

order of execution.

Languages are- prolog

enabling condition$_1$ $\longrightarrow$ action$_1$

$\equiv$

enabling condition$_n$ $\longrightarrow$ action$_n$.

4) Object Oriented Programming: Object is the basic building block.

Object is characterized by state & behaviour.

State is specified by attribute and ~~attribute~~ behaviour is specified by method, Encapsulation, polymorphism, inheritance are the foundational aspects that give identity to this paradigm.

Languages are- C++, java.

5) Event driven Programming: Based on set of anticipated event.

Base system recognizes the events as they occu and coordinates necessary responses.

Languages are- Visual C++, Visual Basic.

6) Concurrent Programming: (parallel programming) this paradigm supports multi-threading. Segmen of same program can execute concurrently and Synchronization facilitates, cooperation among several threads.

Languages are- Occam

7) Distributed Programming: Synchronization & Semantics for message passing form the core support for implementing RPC & RMI

RPC - Remote Procedure Call
RMI - Remote Method Invocation

8) Database programming: This paradigm provides a structured way of framing the query. On RDBMS. It also provides framework for verifying & validating query results
Languages are - SQL, Oracle, etc.

Programming Environment:

It is the environment in which programs are created & tested. This is different from OS environ
It consists of set of support tools and command language for invoking the programs. This tool is used by the programmer during creation of the progra
following other are the tools of programming environme

1) Editor
2) Debugger
3) Verifier
4) Test Data generator.

Environment framework: It helps to provide faster, better and cheaper software.

It consists of infrastructure services that help the programmer to manage the development of the prog. This ~~from~~ framework supplies services such as data repotary, GUI, Security, communication servic. In Windows, we have Motif functions to display windo menus, Scroll bars, etc. to perform the function. We can move the mouse to appropriate picture or icon in order to start the execution. It allows user to be in direct control of determining what steps to perform. If computation face user can invoke editor & correct the program, if compilation succeeds user could invoke a loader and execute the program

Role of programming Environment:

1) Seperate Compiling: In order to execute large program which is made up of different components, we first compile and execute these components seperately and then merge them to form a complete program. It is ~~different~~ difficult to do seperate compilation of sub-programs or components because of following reasons

i) There might be a possibility that one sub-program uses the shared object which must be in consistent form so that it can be easily and efficiently used by other sub program.

ii) Another reason is when two or more sub-programs may be written in different language.

iii) It is considered as ~~seperate com~~ difficult job because there is no way to check the consistency of information about external sub-program and data that are redeclared in the sub-program. If declaration within sub program doesn't match with the actual structure of the external data then error will be appeared.

2) Testing & Debugging:

i) Execution trace features: Prolog and LISP allow statements and variables to be tagged for tracing during execution. Tagged variable is assigned a new value. Execution of program is interrupted and trace subprogram is called which prints a debugging information.

ii) Break points: Programmer can specify points in program as break points. When a break point is reached during execution, program execution is interrupted and control is given to the programmer. Programmer may inspect & modify value of variables and then restart the program from point of interruption.

iii) Assertion: assertion is conditional expression inserted as seperate statement in program. When assertion is enabled

compiler inserts code into compiled program to test the condition stated. During execution if condition fails, execution is interrupted and exception handler is invoked to print the message. After program is debugged, assertion is disabled