Evolution of Programming Language

Shrawan - 07

Chapter - 1

Programming Language :

- A programming language is a formal language that specifies a set of instructions that can be used to produce various kinds of output.

- Programming language is a ~~formal~~ language that is intended for the expression of computer programs and that is capable of expressing any computer program.

over the last fifty years, there was thousands of programming languages introduced, belonging to several programming Paradigms. Despite of so many language, there are not manly languages that survived for more than ten years. The most important programming paradigm are

(i) Imperative paradigm
(ii) Object-oriented paradigm
(iii) functional paradigm
(iv) Logic paradigm

Challenges of Programming Education :

Acquiring and developing knowledge about programing are highly complex process. Novice (beginner) programmers have to overcome a wide range of difficulties. Programming courses are regarded as difficult, and often have very low passing rates.

There are five overlapping domains that student should acquire in an introductory course:

1. General orientation:
   The capabilities and applications of programs.
2. The notional machine:
   An abstract model of the computer used for executing programs
3. Notation:
   The syntax (format) and semantics (meaning) of a particular programming language.
4. Structures:
   The structuring of basic operations into schemas and plans.
5. Pragmatic
   The skills of planning, developing, testing, debugging, documenting etc

Main goals

- become familiar with the fundamental concepts of computer science.
- Develop proficiency in an engineering problem solving and design methodology.
- Understand the importance of advanced information technologies.

Main Objectives

- Use computers and application software as tools to solve problems
- Analyze, design, build and test operational solutions.
- Acquire the foundation of algorithmic processes.
- Learn to exploit the educational and professional resources available on the internet and world wide web.
- Develop a framework for considering the ethical implications of advanced information Technology

It is generally accepted that it takes about ten years of experience to turn a novice into an expert programmer, but there seems to be very few metrics for what constitutes a good programmer. Some of the relevant question are:

- Is it important to write efficient code or readable code?
- Is it more important to have a clumsy bug-free code or to have an elegant algorithm?
- Does it matter if student's programming is not entirely pure, as long as it meets all the functional requirements for the task?

Programming Paradigms

In computer science, several programming paradigms can be recognized. Moreover, The four main problem solving approaches: programming paradigms (प्रतिमान) are recognized as fundamental. Each of these approaches involves a distinct way of thinking and each is supported by a range of programming languages

- Imperative
- Object-oriented
- Functional
- Logical paradigm

a. Imperative Programming :

- The imperative programming paradigm is based on the von Neumann Architecture of computers, introduced in 1940's.
- Von Neumann architecture is the dominant computer hardware architecture which consists of a single sequential CPU separate from memory, and with data piped between CPU and memory
- This is reflected in the design of imperative languages, with:
  a. States: representing memory cells with changing values
  b. Sequential orders - reflecting the sequential CPU.
  c. Assignment statements - reflecting piping

Imperative programs are sequence of direction (orders) for performing an action. IP is characterized by programming with states and command which modify these states. Imperative programming languages provide a variety of commands inorder to structure the code and to manipulate the states

In Imperative Programming, a sequence of command can be named and name is used to invoke the sequence of command, called subprogram, procedure or a function. When IP combines with subprogram called procedural

Example: FORTRAN (1954), Cobol (1959), Pascal (1970), C (1971), Ada (1979)

b. Object-Oriented paradigm

- OOP is a generalization of imperative programming.
- The conceptual model is developed from simulation of event.
- If real world phenomena are simulated, then there should be an object for each entity involved in the phenomena. Object is an entity encapsulating data and related operations.
- In real world, objects interacts so OO programming uses message passing to capture interaction between objects.
- A programming language ~~uses messag~~ supporting this concept and using objects is called object-based. OOP support addition features:
  - Abstract data type definitions are used to define properties of class of objects.
  - Inheritence is a mechanism that allows definition of one abstract data type by deriving it from ~~other~~ an existing abstract data type - the newly defined type inherits the properties of parent type.
  - Inclusion polymorphism allows a variable to refer to an object of a class or an object of any of its derived
  - Dynamic binding of function calls supports the use of polymorphic function; the identity of a function applied to a polymorphic variable is resolved dynamically based on the type of the object ~~ob~~ referred to by the variable.
- Smalltalk (1969)
- C++ (1983)
- Java (1995)

c. Functional Programming

- The functional programming paradigm is based on the theory of mathematical functions, more precisely on the lambda-calcus.
- It allows a programmer to think about a problem, nature of problem rather than sequential nature.
- It is mainly motivated and developed by these qn.
  a. What is the proper unit of program decomposition?
  b. How can language best support program composition from independent components

Three main sets of component
a. data objects - such as list or an array.
b. built-in function - manipulating the basic data objects.
c. functional forms - for building new function

functional programming language are called applicative since the function are applied to their arguments, and non-procedural or declarative since the definitions specify what is computed and not how it is computed.

Eg: LISP (1958) ✓
ML (1973)
Scheme (1975)
Miranda (1982)
Haskell (1987)

d. Logic Programming

- The logic programming paradigm is based on first order predicate calculus.
- focus on declarative description rather than decomposition of the problem to be solved.
- The problem description used by an inference engine to find a solution.
- Consists of
  a. Axioms: defining facts about objects
  b. rules : defining ways for inferencing new facts
  c. goal statement - defining a theorem, potentially provable by a given axioms and rules.

Logic programming is characterized by programming with relations and inference. The programmer is responsible for specifying the basic logical relationships and doesnot specify the manner in which the inference rules are applied. Logic languages are usually more demanding in computation resources than procedural and object oriented languages

Logic paradigm examples are:
Prolog (1970)
Gödel (1994)
curry (1997)

Importance of study of programming language

- Programming language remains the central tool for problem solving in computer science.
- The progress of computer science can be trace in the progress of progress of programming language.
- Some Importance of knowing evolution of programming language is highlighted below:

a. Influence of languages on problem solving
- The structure of language defines the boundaries of thoughts - The Sapir-Whorf hypothesis (controversial)
- Although no programming language (PL) can prevent us from finding a certain solution to a problem, a given language can influence the quality of our programs

b. Benefits for all computer scientists
- by studying evolution, will aid you in choosing the best language for your application
- you learn the motivation for and the use of the most important facilities found in modern programming language.

c. Benefits for language designer
All enginering design is cumulative process, we will learn by analyzing succes and failures of the design of the past

d. Benefits for hardware Architects
hardware architect will gain insight into the ways machine may better support languages.

e. Benefits for language implementer
will gain insight into the motivation

f. Benefits for system designer
S/W software designer (OS & DBMS) will learn principles and techniques applicable to all human interface. Many S/W tolls including OS command language

g. Benefits for Software managers
- better able to make the decision if you know the cost of designing, extending a language.

Attributes of a Good Language

It is a programmer who ultimately decides which language will survive and which will die.

Reasons for a language to survive in this tough competitive world:

1. Clarity, Simplicity and Unity:
- A programing language provides a medium (माध्यम) to conceptual thinking of new algorithm and also to execute your thought process onto real coding statement.
- For algorithms to be implemented on language: It basic need is that the language is quite clear, simple and unified
- It is desirable to have a minimum number of different concepts so that combining multiple concept won't be that complex in natural. It must be simple and regular as possible such attribute called conceptual integrity

- Nowadays, Readability is major concern. The syntax of language effects the ease with which programs are written, tested and later used for knowledge / research purpose.

- A complex syntax language may be easy to write program in, but it proves to be difficult to read and debug the code later. Example, APL is complicated that even own developer finds their code difficult to understand after 1-2 months.

- The language should be simple enough to understand and point out errors.

2. Orthogonality (statistically independent)
Orthogonality refers to the attribute of being able to combine various features of a language in all possible combinations, with every combination being meaningful.
Example : conditional if ( )
logical &&
if (— && —) combining two

3. Naturalness of the application.
A language needs a syntax that, when applied properly, allows the program structure to reflect the logical structure what a programmer wanted it to. The language should provide appropriate data structures, operations, control structures and a natural syntax for the problem to be solved.
Example: implementation of pile of plate: Stack

4. Support for Abstraction
Many times language fail to implement many real life problemss into programs. There is always a gap between abstract data structure and operation
Example: consider a situation where scheduling is to be done for college student for attending a lecture in a class section, teacher. It can't be done by C, but easily accomplish using C++.

5. Ease of Program verification
The reliability of programming language written in a language is always a central concern. Sometime, verification done with random values of inputs and obtaining corresponding output

6. Programming Environment
Environment which is technically weak may get a bad response of programmer rather than a language that has less facility than former but its environment is technically good. Special editors, testing package are considered as good featured of programming Environment

7. Portability of programs
A language which is widely available and does not support different features on different computer system, which may have different hardware, is considered as good language.
ex: C, C++ are portable in nature

orthogonality principle:
Independent function should be controlled by independent mechanisms

8. Cost of use.
it means evaluation of any programming language but includes the cost of:

(i) Program Execution
total amount which has been used to implement the program. design, optimizing compilers, data allocation register etc are under PE.

(ii) Program Translation
compilation done more than execution. it is important to have a speed and efficient compiler to handle this job.

(iii) Cost of program creation, Testing and Use.
Cost which a programmer charges for his code of creating project with specified features

(iv) Cost of program Maintainance
Maintaince includes rectification of error propagated in real time, the updating of program as need of time.

Program Design Notations
Notations are for programmer to reduce the complexity in programming and understanding the program.
Some of these notations helped the programmer to design the
- Memory Layout
- Control flow
  - Flow diagrams (later Flowcharts)
- Mnemonics
  - helps to remember instruction codes like assembly language today

* Shortcoming Of earlier computer hardware:

a. Floating point Arithmetic:
Floating point is harder to support by hardware due to its complex representation. The earliest built-in floating point was provided by IBM 704 in 1953. Before that, it had to be simulated using following steps
> Manual scaling: numbers were multiplied by scale factor
> Multiply by constant factor
> Use integer processor
> Manually scale back result
> Complicated and error-prone process

b. Indexing:
Array is one of the most common data structures used in computer programming. Array uses indexing to store and retrieve data from consecutive memory locations. 'Adding a variable index quantity to a fixed address inorder to access the element of an array. Early computer doesnot support. They used address modification by:
> Altering the program's own data accessing instruction.
> It also consumed much of the scarce memory with this address modification code.
> For this reason it was common to use subroutine to perform indexing
It was every error prone process.

→ This justified the use of pseudocode interpreter

Pseudocode :

It is an instruction code that is different than that provided by real machine. It offered - Floating-Point and indexing which was actually not supported by earlier generation of computers. It provided entirely new instruction set not offered by real h/w.

* Need of Pseudocode :

During 1st gen, programmer need to know about the hardware specification of every machine that the program was intended for. In 1950's for IBM 650 which has following characteristics.

> No programming language was available
> Memory was only a few thousand words
> Stored program and data on rotating drum
> Instruction included address of next instruction so that rotating drum was under next instruction to execute and no full rotations were wasted.

* Pseudocode Interpreter :

Pseudocode Interpreter is an interpretive subroutine (The subroutine is an important part of any computer system's architecture) developed to run the pseudocode. They are used for saving memory since the pseudocode is more compact because implements a virtual computer which allows us to use functionality with its own data types (eg: floating point) and operations (eg indexing). It is an example of 'Automation principle' of programming language.

Pseudo-code interpreters were commonly used to perform floating-point operations and indexing, and this simulated instruction not provided by hardware.

→ Pseudocode interpreter (a primitive, interpreted programming language) implements :
- A virtual computer
- New instruction set
- New data structures

• A virtual computer is
- higher level than actual hardware
- Provides facilities more suitable to application
- Abstracts away hardware details.

* Pseudocode follows two basic principles of Programming
1. The Automation Principle:
Automate meachanical, tedious or error-prone Activities.

2. The Regularity Principle:
~~The~~ Regular rules, without exceptions, are easier to learn, use, describe and implements.

* Compiling Routines also used for implementing Pseudocode. Grace Murray began to use programs to extract subroutines from libraries and combine them (a process called Compiling) under the direction of pseudocode. Since this process was done once, at compilation time, do not need interpretation.

* Design of a Pseudo-code:

For the sake of example: we will assume that we are designing this pseudo-code for a computer with 2000 words of 10 digit memory; this was the capacity of 650 and is a reasonable assumption for machines of that vintage.

The design of Pseudo-Code based on the capabilities and constraints on the first generation computers. ~~In 1950~~ so lets begin by making a list of some of function our programmers are expected accomodate on in 1950, programmers are expected this but not by hardware of that time are:

- Floating-point arithmetic $(+, -, \times, \div, \sqrt{\ })$
- Floating-point comparision $(=, \neq, <, >, \leq, \geq)$
- Indexing
- Transfer of control
- Input-output

Hardware Assumption

The IBM 650 servers

> 1 word: 10 decimal digits + 1 sign

> 2000 byte memory

- 1000 for data
- 1000 for program

The design of Pseudocode must follow:

Impossible Error Principle

Making errors impossible to commit is preferreble to detecting them after their commision

* Instruction have three operands

The arithmetic operation's instruction consist of 3 operands and could have two or three or four addresses

- Two addresses $X + Y \rightarrow X$ (X, Y)
- Three addresses $X + Y \rightarrow Z$ (X, Y, Z)
- Four addresses $X + Y \times Z \rightarrow W$ (X, Y, Z, W)

op opn1 opn2 dest

↳ destination (Z)

↳ operands (X, Y)

↳ operation (instance: '+1' = addition)

Eg. +1 010 150 200

would add the content of location 010 to the contents of location 150 and store the result in location 200.

1 word can be enough to ~~opera~~ specify 3 operand instruction. operation contain a sign (+, −, etc) and 1 digit. It supports 20 operations.

* Orthogonal design increase Regularity

- operations should be more intuitive (सहज) than machine code
- Use the sign to get more orthoganility

+1 010 150 200 → add
−1 010 150 200 → subtract

} Orthoganility due to (+, −)

Orthogonality Principle

Independent function should be controlled by independent mechanism.

The orthoganility principle is a corollary (पहले prove करो) of Regularity Principle.

~~Q. Why~~

~~Oper~~

* Instruction format

Operand src1 src2 dst

Eg $X+Y \rightarrow Z$ (+1 010 150 200)

This instruction instructs to 'Add values at location 010 and 150 and save it to location 200.

To follow orthogonal design subtract operation should be '-1'

* Arithmetic operation – Comparisions

Arithmetic do mathematical operations.

~~Lo~~ Comparision alter the control flow of program

	+	−
1	+	−
2	x	÷
3	square	sq. root
4	if = goto	if ≠ goto
5	if ≥ goto	if < goto

for example

if $x \bar{=} y$ then goto Z

+4 200 201 035

if content of 200 = content of 201 goto 035 location

* Moving

Data could be moved by 'adding 0' to an address but that operation is expensive so new operation should be assigned to move data.

+1 150 000 280

moves the contents of location 150 to location 280

* Indexing and loops

→ To perform indexing, we will need the address of the array and the index variable, thus consuming two of three address fields in the instruction. Therefore, the only operations we can perform directly on array elements are to move them to or from other locations. we can use the codes +6 and −6 to move from or to an array: X i Z and Y y i

The formats of these operations are

+6 XXX iii ZZZ | −6 XXX yyy iii

Eg: If there is a 100-element array beginning at location 250 in data memory, and location 050 contains 17, then

+6 250 050 803

will move the content of location 267 (250+17) to location 803

−6 722 250 050

will move the contents of location 722 to location 267

→ Looping through the elements of an array requires
- iterator variable (array index i)
- upper bound (n)
- Address of beginning of loop (d)
- " +7 iii nnn ddd'

The operation increments location iii and loop to instruction ddd if the result is less than the contents of nnn

* Input - Output

A program needs to read data from input and write data to output.

> Needs only a memory location to read from or write to.

> Read: "+8 0000 0000 dst'
"-8 000 000 src'

Language design must follow

The Abstraction People

Avoid requiring something to be stated more than once; factor out the recurring Pattern.

* Complete Pseudo-code Operations

	+	-
0	Move MOVE	
1	+ ADD	- SUB
2	* MULT	÷ DIV
3	x^2 SQR	$\sqrt{x}$ SQRT
4	= EQ	≠ NE
5	≥ GE	< LT
6	GetArray GETA	PutArray PUTA
7	Incr & Test Loop	Label LABL
8	Read READ	Print PRNT
9	STOP STOP	Trace TRAC

Program Structure

we know how to write individual instruction, but we have not designed a means of constructing the program as a Whole.

- The simplest solution to this problem is to have interpreter read intialization cards and their Content in consecutive memory location

Thus the structure of a program is:

Intial data values
+ 99999999999 (10 आंकड़े 9)
Program instructions
+99999999
Input Data

we have used a card containing the "flag value" +99999999 to separate the intial value from the program and the program from the input data. → The loader reads in the intial data values and stores them in consecutive location (starting with 000) in the program memory

The loader doesnot read the input data; this is read by the user's program whenever executes a +8 instruction

Therefore a general structure of a program is

1. declaration
2. executable instruction
3. Input data

1. What is program structure ?

- The overall form of the program, with particular emphasis on the ~~invalid~~ individual components of the program and the interrelationships between these components. Program are frequently referred to as either well structured or poorly structured.
- with a well structured program, the division into components follow some recognized principle such as information hiding, and the interfaces between components are explicit and simple.
- By contrast, with a poorly structured program the division into components is largely arbitary and interfaces are implicit and complex.
- At a finer level, a well-structured program employs - appropriate data structures
  - program units
  - control flow.

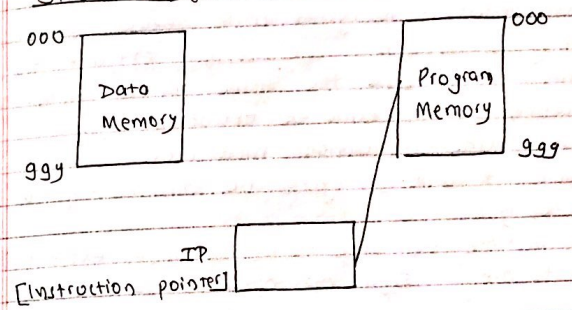
* Implementing the Interpreter : Interpreter for Pseudocode program can be implemented as :
  - Model interpreter behaviour ~~to~~ after manual execution
  - Cheat: Implement using a high level language
  - we have to stimulate the hardware in software

Second option is impossible as HLL ~~is~~ was not developed in 1960s. so third one is good for us.

Data Structures

Data structures are needed to stimulate the IBM 650 which contain
- Data memory
- Program memory
- Instruction pointer

Structure of the Interpreter :



[Fig: Interpreter Data structure]

* Read execute cycle is the heart of an iterative interpreter

→ What will we do is 'read the next instruction to be executed [IP points the address of next instruction], determine the operation encoded by the instruction, and then perform that operation. when execution is completed, it is called read-execute cycle

1. Read the next instruction
2. Decode the instruction
3. Execute the operation
4. Continue from step 1

The IP must be updated each time an instruction is read

Typical code for step 1 is
instruction := program [IP]
IP := IP + 1

* Decoding Instruction by extracting their parts
- Pseudocode is designed with regular structure, decoding is simple; we simply extract the sign operation code. and the three address field. The destination address could be extracted by:
dest := abs (instruction) mod 1000
(where x and y gives the remainder of dividing x by y)
we assume that the names of these extracted parts are sign, op, opnd1, opn2, and dest.

The operation types are
- Select operations
  switch statement (case - statement)
- Arithmetic Operations
  straight forward.
- Control flow
  IP may also need to be altered.

Labeling
- Labeling is the aid to the coding of the Programs. labels make program readable however they are not converted into object codes.
- Labels are used for replacing absolute addressing in the programs particularly in loops and branching.

we can define a label in Pseudocode using label definition operator (-7)
-7 OLL 000 000
defines the statement number or label, LL
only 100 numeric labels are possible (00-99)

* Interpreting Labels:
- Interpretator do not look through all instruction from beginning of program.
- It creates a label table with absolute address for labels and bind address.

Label	Location
20	001
40	005
50	009

Much faster; compiler do it this way

The Labeling Principle
Do not require users to know absolute number or addresses. Instead associate labels with number or addresses

* Data Labels:
we could use labels for variables by constructing a symbol table.
In initial data values
+0 sss nnn 000
± dddddddddd (10 अंक)
- declare n values of d referenced by symbol s
- symbolic notation:
VAR sss nnn
± dddddddddd (10 अंक)
- n = 1 : simple variable

The IP must be updated each time an instruction is read

Typical code for step 1 is
instruction := program [IP]
IP := IP + 1

* Decoding Instruction by extracting their parts
- Pseudocode is designed with regular structure, decoding is simple; we simply extract the sign operation code, and the three address field. The destination address could be extracted by:
dest := abs (instruction) mod 1000
(where x and y gives the remainder of dividing x by y)
we assume that the names of these extracted parts are sign, op, opnd1, opn2, and dest.

The operation types are
- Select operations
  switch statement (case - statement)
- Arithmetic Operations
  straight forward.
- Control flow
  IP may also need to be altered.

Labeling
- Labeling is the aid to the coding of the programs. labels make program readable however they are not converted into object codes.
- Labels are used for replacing absolute addressing in the programs particularly in loops and branching.

we can define a label in Pseudocode using label definition operator (-7)
-7 OLL 000 000
defines the statement number or label, LL
Only 100 numeric labels are possible (00-99)

* Interpreting Labels:
- Interpretator do not look through all instruction from beginning of program.
- It creates a label table with absolute address for labels and bind address.

Label	Location
20	001
40	005
50	009

Much faster; compiler do it this way

The Labeling Principle
Do not require users to know absolute numbers or addresses. Instead associate labels with number or addresses

* Data Labels:
we could use labels for variables by constructing a symbol table.
In initial data values
+0 sss nnn 000
± dddddddddd (10 अंकों)
- declare n values of d referenced by symbol s
- Symbolic notation:
VAR sss nnn
± dddddddddd (10 अंकों)
- n = 1 : simple variable

Debugging :

Debugging always has to be done and it can be facilatated by debugging by printing instruction executed in order. Interpreter can trace the execution of the program.

Security Principle

No Program that violates the definition of the language or its own intended structure, should escape detection.

Phenomenology of Programming language:

Obviously, Programming language, even simple ones such as Pseudo-code, are tools, so it will be worthwile to investigate them from this perspective Phenomenology

1. Tools are ampliative and reductive

ampliative : extending, adding to, थप्नु

Reductive : shrinking, subtractive to, कम्नी

→ Ihde contrasts the experience of using your hands to pick fruit with that of using stick to knock the fruit down.

→ Stick is ampliative; it extends your reach to otherwise inaccessible fruit.

→ It is reductive too; your experience of the fruit is mediated by stick, for you do not have direct experience of grasping the fruit and tugging (तान्ने) off the branch. You cannot feel if the fruit is ripe before you pick it.

- "Technological utopions" tends to focus on the ampliative aspect - the increased reach and power - and to ignore the reductive aspect. ~~when~~
- "Technological dystopions" tend to focus on the reductive aspect - the loss of direct, sensual experience - and to diminish the practical advantage of the tool.
- Both are reduced focuses upon different dimensions of the human technological experience.
- we should acknowledge every tools have positive and negative aspector. As Ihde says, "all technology is non-neutral"

These observations apply directly to programming languages. In the earliest days, computer were programmed directly with patch-cords (directly with hardware, Picking your own fruit!), But Pseudocode are more distancing amplifying programmers ability to write correct code, but reducing their contact with and control over hardware.

कुनै पनि चिजले केहि सुविधा थपेको हुन्छ भने त्यसले गर्दा केहि कमिपनि ल्याएको हुन्छ। Pseudo-code मा hardware संगको interaction is ampliative किनकी यसले गर्दा hardware independent, machine को information नभएपनि हुन्छ तर hardware संग पनि control गर्नको लागि अझ powerful हुने थियो, so it is ampliative and reductive as well.

2. Fascination and fear are common reactions to New tool.

→ कुनै पनि नया चिज आउदा, यसले रोचकता र डर दुवै जिगाउँछ।

Programming languages elicited the two typical responses to a technology: fascination and fear

- Utopians tend to become fascinated with the ampliative aspects of the new tool, they eager to use it.
- Dystopians fear the reductive aspect of the tool which may seem dangerous.

Ideally, greater familarity with a technology allows us to ~~go~~ grow beyond these reactions, for the benefits and limitations of a technology.

3. With mastery, objectification becomes Embodiment

Fortran

* History and Motivation.

- In early 1950's, there was little sympathy for the idea of a high-level programming language.
- Simple assembler and libraries of subroutines were the accepted tools
- An elegant algebric language is developed by Laning and Zierler of MIT in 1952.
- John Backus of IBM had design one system, called Speedcoding, lack of floating point and indexing.
- Backus concluded that programming costs could be decrease if mathematical notation can be used.
- He proposed these ideas to his management at IBM in January 1954. as a FORTRAN Formula Translation

timeline

1954: Project Approved

1957: Fortran : First version released

1958: FORTRAN II and III

Still many dependency on IBM 704

[hardware related to IBM704]

1962: FORTRAN IV

- 'ANS FORTRAN' by American National standard Institute.
- Breaks machine dependence

• we'll look at 1966 ANS FORTRAN

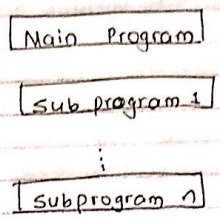
Goals: Decrease Programming cost

Efficiency

Backus and his team (9 others) began the design and programming of FORTRAN in early 1955 and released the system in April 1957 after 18 years of work.

Design: Structural Organization

1. Programs are divided into Disjoint subprograms
   - Pseudocode lacks subprogram, FORTRAN I system supports subprogram facility. The overall structure of a FORTRAN is mainprogram and zero or more subprograms
   - For example:



2. Constructs are either Declarative or Imperative
   - Pseudocode programs are divided into two parts
     - declarative : data areas, length, intial values.
     - Imperative: commands to be executed at runtime.
   - In FORTRAN, declarative constructs are nonexecutable statements and imperative constructs are executable statement
   - Declarative constructs are like declarative sentence which state facts about program, used at compile time
   - Imperative constructs are like imperative sentence, they give command, used at runtime.

3. Declaration include Bindings and Intializations
- In Pseudocode, we saw declarations performed three functions:
- They allocated memory
- They attached name to that memory
  called binding a name an area of memory.
- They intialized the contents

- FORTRAN ~~II~~ ~~fn~~ same
Example
DIMENSION DTA (900)
causes the loader to variable DTA to allocate 900 words
- To declare and intialize we used, DATA
DATA DTA, SUM / 900*0.0, 0.0
would intialize the array DTA to 900 zeroes and the variable SUM to zero

4. Imperatives are either computational, control flow or input-output.
- It includes 3 kinds of statement
1. Computational statement
- such as arithmetic and move operation
- Assignment statement is
  AVG = SUM / FLOAT(N)
- Algebric notation was important contribution

2. Control-flow statement
IF statements, conditional, comparision
DO statement, loop
GOTO statement, unconditional transfer

3. Input-Output
- namely READ and PRINT
- FORTRAN I had 12 statements for performing or controlling input-output of a total of only 26 statements

5. A program goes through several stages to be run : Compilation
- The series of stages are
  1. Compilation
  2. Linking
  3. Loading
  4. Execution

- Compilation translates individual FORTRAN subprogram into relocatable object code
- linking, links the different libraries
- loading, program is placed in computer memory
- execution, control of computer is turned over in the program in memory.

6. Compilation involves three phases
1. Syntatic Analysis:
   Compiler must classify the statement, constructs and extract their parts.
2. Optimization:
   FORTRAN has sophisticated optimizer which is equivalent as professional programmer.
3. Code synthes:
   The final task of compilation is to put together the ~~pro obj~~ parts of the object code in relocatable format.

Control Structures :

- Control structure is a constructs in the language that govern the flow of control of the program
- FORTRAN program mostly includes collections of assignment and control statement.
- FORTRAN was originally designed for IBM 704 and similar to the instruction set of IBM 704
- FORTRAN sometimes called an "assembly language for IBM 704
- 1st generation is machine dependents.

1. Arithmetic IF Statement

- IF (e) n1, n2, n3

evaluates the expression e and branches to n1, n2 or n3 depending on value of e as -ve, 0 or +ve respectively.

- ~~later,~~ arithmetic IF was not very satisfactory later, a more conventional If-Statement was added.

```
IF (X.EQ. A(I))    K = I-1
```

Arithmetic IF are violations of the Portability feature, It is machine dependent features.

> **The Portability Principle**
> Avoid features or facilities that are dependent on a particular computer or a small class of computers

2. GOTO : workhorse of control flow

In FORTRAN, GOTO statements is the raw material from which control structure are built.

- two way branch : Combination of IF and GOTO

```
      IF (condition) GOTO 100
            ... case for condition false...
      GOTO 200
100         ... case for condition true...
200   ...
```

- It is like If-then-else or conditional statements
- although the false and true branches are altered.
- If we wish to put true and false branch respectively, we must negate the condition Like:

```
      IF (.NOT. condition)) GOTO 100
```

- This use of IF-Statement divides the control flow into two cases, for dividing it into more than two cases, a Computed GOTO is provided

```
      GOTO (10, 20, 30, 40), I
10          ... handle case 1
      GOTO 100
20          ... handle case 2
      GOTO 100
30          ... handle case 3
      GOTO 100
40          ... handle case 4
100
```

Transfer to the statement 10, 20, 30 or 40 if I is 1, 2, 3, or 4 respectively. It is like switch-case. [N-way branching]

Note: GOTO (L1, L2, L3, L4), I
Transfer control to the Lk, if I contains k

3. LOOPS

Loops can be implemented by combination of If- and GOTOs

- trailing-decision loop could be: (while do)

```
100   ... body of loop
      if (loop not done) GOTO 100
```

- leading decision (repeat-until in Pascal)

```
100   if (loop done) GOTO 200 ~~(repeat un~~
      .. body of loop
      GOTO 100
200   ...
```

These are indefinite iterations

Information:
In Pascal, if we see while, we can instantly say it is leading decision loop but in FORTRAN, if we see if (without looking closely) it can be trailing or leading loop, conditional, selection or mid decision loop

The Structure Principle : Dijkstra
The static structure of a program should correspond in a simple way to the dynamic structure of the corresponding computations.

this principle means, should be able to visualize the behaviour of program easily from its written form.
Example: हेर्दा हुँदा उन सकुन पर्यो कि यो loop हो र यसको block यति, simplified हुनपर्यो लेखेको कार्यहरू

4. Computed and Assigned GOTOs : Confused

Computed GOTOs:
GOTO (L1, L2 ... Ln), I

where Li are statement number and I is any integer variable. The computed GOTO transfers to statement number Lk if I contains k.
- computed GOTO are usually implemented by a jump table.
- The compiler stores addresses (Lk हरू) in an array in memory and then compiles code to use I as an index into this array.

Assigned GOTO
ASSIGN 20 TO N
GOTO N, (L1, L2, ..., Ln)

where N is an integer variable. This statement transfers to the statement whose address is in the variable N; this is an indirect GOTO. The list of statement label is not actually necessary since N has an address to perform jump. Most compilers do not check.
The assigned GOTO must assign the address of statement say X.

The effect of ASSIGN 20 TO N will put the address of statement 20, say 347 into N.

* Computed and assign GOTO could be confused

① If programer is expected to use ~~computed~~ assigned goto but she used computed as

```
ASSIGN 20 TO N
  :
GOTO (20, 30, 40, 50), N
```

Say, N has 347, Computed GOTO attemp to jump to 347 index, which is out of range.

② Consider the opposite error, expected to use computed, but programmer used assigned GOTO as

```
I = 3
  :
GOTO I, (20, 30, 40, 50)
```

Since, assigned GOTO expects the address of a statement to be stored in variable I, it will transfer to address 3.

Cause of the Problems

GOTO (L1, L2, L3), I
GOTO I, (L1, L2, L3)

This is the violation of Syntatic Consistency principle.

Syntaltic Consistency Principle

Things that look similar should be similar and ~~things that look different should be different~~

Example:

* → for multiplication
** → for exponentiation

Confusing in GOTO will not caught by compiler

Defense in Depth Principle (syntatic checking)

If an error gets through one line of defense, then it should be caught by the next line of defense (type checking in this case)

5. DO Loop

DO-Loop provides a simple method of constructing a counted Loop sometimes called a definite iteration

```
      DO 100 I = 1,N
        A(I) = A(I)*2
100   CONTINUE
```

execute the statements between DO and CONTINUE with I taking on the values 1 through N

I is called controlled variable.

equivolent to:

```
I = 1
100  IF (I .EQ. N) GOTO 200
       A(I) = A(I)*2
       I = I+1
       GOTO 100
200  ...
```

(i) The DO-Loop is higher level :
- it allows programmer to state what they want rather than how to do ?
- Higher level construct as this is safer to use because they remove error

The Impossible error Principle
Making error impossible to commit is preferable to detecting them after their commision

(ii) Loop can be Nested
- Do loop are constructed hierarchically

```
      DO 100   I=1,M
        :
      DO 200   J=1,N
        :
200     Continue
        :
100     continue
```

but must be properly nested

(ii) Do Loop is highly Optimized
- do-loop can be partly attributed to its structure. Controlled variable and its intialization, increments are all stated explicitly along with the extent of the loop.
- IF and GOTO are more expensive and confusing.

Preservation of Information Principle
The language should allow the representation of information that the user might know and the compiler ~~might~~ need.

6. Subprogram (SubRoutine)
- FORTRAN I ommitted what is now the most imp.
- It has libraries of Input/Output, Mathematical like SIN, COS as Pseudocode did.
- It miss to let the programmer define their own subprogram so FORTRAN II added : so it is called late addition.
- Subprograms define procedural abstraction because:
  - repeated code can be abstracted out, variables formalized
  - allow large program to be ~~form~~ modularized.
- Instead of repeatating ~~same~~ sequence, we can define procedural abstraction once and call for it to be used every time.

```
SUBROUTINE name(formal)
    ---body---
RETURN
END
```

where name is a function name to be bound
formals is a list of the name of formal parameters also called dummy variable - formals bounds to actual

```
SUBROUTINE DIST(D, X, Y)
  D = X-Y
  IF (D.LT. 0)   D = -D
  RETURN
  END
```

```
fun1(X)
  X=100+X;
  return X;
}
```

X sum = ABS00 SO

y = fun1(sum)

This subroutine can be invoked by a CALL statement as:

```
CALL DIST ( DIST1, X1, X2)
:
CALL DIST (DIFFER, POSX, POSY)
```

- Callee is the subprogram being called.

(i) Subprogram allow large program to be modularized
- Saves writing
- total difficulty of design process can be decreased, each subprogram act. debug independent

The Abstraction Principle
Avoid requiring something to be stated more than once; factor out the recurring pattern.

(ii) Subprogram encourage libraries
- Separate compilation
- independent to each other
- they can be separately translated to relocatable object code so that it later combined by linker.
- Programming in Small

(iii) Parameters are usually passed by Reference
- Actual parameters copied to formal - copy rule.
- In FORTRAN, all parameters normaly are passed by reference
- when statement D = X-Y is calculated, DIFFER will be modified as well.
- Formal parameter gives the callee read-write access to the actual parameter in the caller

Note: Caller : जुन method बाट subroutine call गरियो।
Callee : जुन method लाई call गरियो।

- The copies would occupy too much space
- It would take too much time to substitue
- FORTRAN permits subprogram to be used for input or output or both.

(iv) Pass by Reference is always Efficient
- Pass by reference always passes just the reference (address of location) to the actual parameter,
- Since address is little information, which is passed to caller to callee

Example: Suppose actual parameters were array elements
CALL DIST (D(I), POS(I), POS(J))
- The address of POS(I) is passed, calculated from the address of first elements of POS and the value of index I

How about entire Array ?
Example: calculate average of an array.

```
FUNCTION AVG( ARR, N)
  DIMENSION ARR(N)
  SUM = 0.0
  DO 100 I = 1,N
      SUM = SUM + ARR(I)
100   Continue
  AVG = SUM / FLOAT(N)
  RETURN
  END
```

So callee can access the array just by having address of first element. ARR is bound to first element of array.

The Impossible Error Principle

See previou

- having to pass the array size means programmer can have wrong size.

(v) Pass by reference has dangerous consequence

- Parameter can be used for both input and output
- It can be dangerous if ~~output~~ actual parameter is input variable that updated by subprogram and later original value required.
- It may introduce a bug, difficult to trace
- SUBROUTINE SWITCH(N)

```
    N=3
    RETURN
    END
    CALL SWITCH(2)
```

The literal 2 (constant), memory location contents now 3 by switch subroutine.

I = 2+2 will be 6. difficult to debug.

(VI) Pass by value is preferable (copy-restore)

- Above problem can be solved by letting the programmer ~~decide~~ to allow to specify whether the parameter is used for input or output.
- Value of the actual parameter is copied into the formal argument at subprogram entry, and the result, or final value of formal, is copied into actual parameter at subprogram exit.
- Both operation are done by caller
- If actual is constant, second operation will omit
- If actual is variable/array, then only value copied

Subprogram : Activation Record

What happens when a subprogram is invoked?

- Transmit parameters by value or reference
- Save caller's status
- Enter the subprogram
- Restore caller's state
- Return to caller.

- When we enter callee, How can we return back to the caller, a subprogram can be called from different caller.
- When callee executes its RETURN statement, it will know whom to return
- Since subprograms may be separately completed, not possible to know register used by subprograms.

Before Invocation

- Place parameters into callee's activation record
- Save caller's status
  - ↳ Save content of register ↳ Save instruction pointer
- Save pointer to caller's activation record in callee's activation record.
- Enter the subprogram

After returning from subprogram

- Restore instruction pointer to caller's
- Return to caller
- caller needs to restore its state (register)
- If subprogram is a function, return value must be made accessible.

Contents of Activation record

- Parameter passed 'PAR'
- IP (resumption address),
- Dynamic link (address of caller's AR 'DL'
- Temporary area for storing registers TMP

Data Structure

- FORTRAN is a scientific programming language
- It has Primitives and Array data structures
- FORTRAN II include type
  - DOUBLE PRECISION
  - COMPLEX
  - LOGICAL (boolean)

(i) Primitive are scalars

FORTRAN supports primitive data types as Integers, floating point number, Double precision floating point, Complex number, boolean values but lacks of text processing.

Representations

- basic addressable units of storage was a word (32 bit)
- Integers represented as binary number with a sign bit
- 4 arith operation, test for zero, tests the sign bit are operation for integer
- Floating point, using scientific notation characteristics + mantissa (m)

sm	sc	c_7	...	c_0	m_{21}	m_{20}	...	m_1	m_0

value represent is $m \times 2^c$, • $m < 1$

(ii) The Arithmetic operators are overloaded

- Integers are subset of real, and real of complex.
- Since, computer representation are different and for real world similarity, conventional writing should be allow for programmer. so that arithmetic operation could be used for mix data type
  $x + 1$, x is real and 1 is integer
- $2 + 3 \cdot 1 = ?$ How to handle? typecasting (coercion)
  FLOAT(2) + 3·1
- Thats why it is called arithmetic operators are overloaded (x+1, Y+Y, 2+3)

So $x ** (1/3) = ?$

Since $1/3 = 0$,

So, $1/3.0 = 0.333$

(iii) Hollerith Constant

- It is a type integer, early form of character string.

Example: 6HCARMEL represents 6 character string 'CARMEL'

- Second class citizen
  - No operations allowed
  - can be read into an integer variable
  - violates the Regularity principle.
- Problems: Integer represent a Hollerith constant may be altered, which makes no sense
- weak typing: No type checking is performed something of one type (hollerith) to be used where another is expected (integer)
- Security loophole: meaningless operation is adding 1 to hollerith.

Constructor : Array

- method to built complex data structures from primitive
- FORTRAN has only array data structure.

  DIMENSION DATA, CORD(10,10), num(100)

  Started from 1
- FORTRAN doesnot required intiolized
- Maximum 3 dimension limitation (FORTRAN 77 has 7)
- Static in nature
- It violates Regularity principle (called zero-One-Infinity principle)

Representation

- Contiguous region of memory
- indexing by adding index value to base address.
- simple, intuitive representation

Dimension A(n)

A(1)
A(2)
⋮
A(n)

DIMENSION A(m,n)

ELEMENT	ADDRESS
A(1,1)	A
A(2,1)	A+1
A(m,1)	A+m-1
A(1,2)	A+m
A(m,2)	A+2m-1
A(m,n)	A+nm-1

- coloumn-major order

$\alpha\{A[i,j]\} =$
$\alpha\{A(1,1)\} + (j-1)m + i - 1$

- one address to each slot

- optimization
  - intialize controlled variable to address of arrayelement
  - increment the address itself
  - Dereference controlled variable to get any arrayelement

Name Structure

- what do name structures structure?
  - Name of course!, organizing names
- It includes declarations or binding constructs
- In Pseudocode

  VAR SUM 1

  +0000000000 (10 ~~stat~~)

  bound identifier SUM to a location memory and intialize that location to 1.
- Primitives bind names to object.

  INTEGER I, J, K
  - Allocate integers I, J and K and binds the names to memory location (one word each)
  - Declare, name, type, storage
  - This is not intialization, but done by DATA statement.

① Declarations are Non-executable statements

- It provides information for the compiler, linker and loader
- Static allocation is done, type in declaration is used to determine amount of storage
- It will make entries in symbol table for I, J, K
- Unlike, GOTO, IF, which are executable

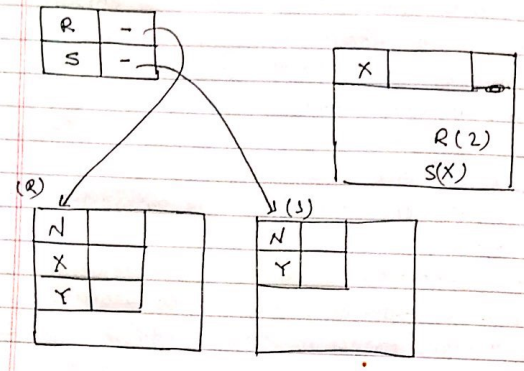
* Optional Declaration

- automatic declaration of variable ie uses a variable that is not declared
- If FORTRAN automatically declare, what type ??
- I = I + 1, I can be INTEGER, REAL, COMPLEX
- Solves by 'I through N' rule: Any variable whose names begin with I through N are declared integers, remains are reals

- COUNT = COUNT + 1 is happened but expected to COUNT = COUNT + 1 and COUNT will be declared as real type automatically. No DATA St. to initialize. hence difficult in debugging.
- Bad practices: Encourage funny names (KOUNT, ISUM, XLENGTH), instead of declaring.

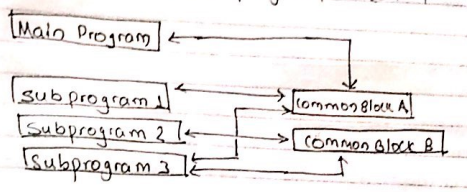
(ii) Variables name are local in scope
(iii) Subprogram names are Global in scope

Contour Diagram:



Once we have subprogram
- we need to find a way to share data by parameter
- Sharing data with just parameter is difficult.
- Common block allow more flexibility.
- provide a set of subprogramms that collectively manage a shared database.
- The ability to have more than one name for the same memory location: very flexible

example of common Block
- common block permits subprogram to have common variable, to share data areas.
- SYMTAB में subprogram होता।



Common blocks:
Common block permits subprogram to have common tables to share data areas as shown in fig above.

EQUIVALENCE allows sharing within subprogram.
- when FORTRAN was developed, scarce of memory, it is mandatory to optimize the memory, by sharing.
- Since dynamic memory allocation is not supported and memory is scarce, FORTRAN has EQUIVALENCE

DIMENSION INDATA(1000), RESULT(8000)
EQUIVALENCE INDATA(1), RESULT(3)

The first element of INDATA is to occupy the same memory location as first element of RESULT

- Nowadays, larger memory so that economization of storage is no longer.
- virtual memory system safely manage the real memory.
- Modern prog lang support dynamic memory allocation

- Equivalence is only be used when usage of INDATA and RESULT do not overlap.
- Allows access to different data types (float as if it was Integer).

Syntactic structures

(i) Language are Defined by Lexics and Syntax

- Syntax of a lang. is way that words and symbols are combined to form the statement and expression (Grammatical Notation)

  Eg: DO 100 i = 1,n

  we saw that a Do-loop has the word Do followed by a statement number, followed by an integer variable, followed by symbol '=', followed by integer expression, followed by symbol ',' and another integer expression

  Syntax: DO <label> <var> = <exp>, <exp>
- The Lexics of a language is the way in which characters (ie letter, number, signs) are combined to form words and symbol.

  Eg: Identifier must begin with a letter, followed by no more than 5 letters or digits.

- Syntactic Analysis means
  - Lexical analyzer called Scanner
  - Syntactic analyzer proper called parser

(ii) Fixed format lexics

- Pseudocode is a fixed format lexics, card oriented.
- Particular colums had particular meaning.
- 7-72 columns was devoted for actual statement

Columns	Purpose
1-5	statement number
6	continuation
7-72	statement
73-80	Sequence number

(iii) Ignoring blanks everywhere is a mistake

- Fortran ignored spaces, not just white spaces

DIMENSION INDATA(10000), RESULT(8000)

Equivalence to:

DIMENSIONINDATA(10000), RESULT(8000)

- Lexing and parsing is very difficult.

DO 20 I = 1,100 could be

DO20I = 1.100 (mistake ~~है~~ में DO20I treat as a variable) Defense in depth principle.

(iv) No reserved words

- FORTRAN allows any ~~itent~~ identifier to be declared as variable
- FORTRAN allow variable named IF

  DIMENSION IF(100)
- How do you read this

  IF(I-1) = 1 2 3 array.

  are likely to be IF (I-1), 1, 2, 3

(v) Algebric Notation was important Contribution

$$\frac{-b + \sqrt{b^2 - 4AC}}{2a}$$

(-B + SQRT(B**2 - 4*A*C)) / (2*A)

Operators Need precedence

precedence : priority of operator

1. exponentiation
2. * and / } left to right
3. + and -

$b^2 - 4ac \Rightarrow (b^2) - (4ac)$

$ab^2 \Rightarrow a(b^2)$

FORTRAN ~~was~~: has linear syntactic Organization

Why FORTRAN is considered as very Successful

- heavily optimized programming language
- Scientific and mathematics problem → solves
- wide application area