1. Evolution of Programming Language.

Programming language: A programming language is a language that is intended for expression of computer programs & capable of expressing any computer program. Many of these languages are used for special purposes. i.e. for editing text, conducting transactions with a bank or generating reports etc.

1.1 Importance of Study:

Importance of learning Programming Language.

→ Programming languages are important for students in all discipline of computer science because they are primary tools of the central activity of computer science: "programming".

→ To improve your ability to develop effective algorithms and to improve your use of your existing programming languages.
- OO features, recursion.
- call by value, call by reference.

→ To increase your vocabulary of useful programming construct.

→ To allow better choice of programming language.

→ To make it easier to design a new language.

→ To understand & describe syntax & semantics of programming language.

→ To understand data, data types & basic statements. (string, character, float, integer, Boolean).

2) Characteristics of good programming Language.

a) Clarity, simplicity & unity: The programming language should have clear approach to achieve the goal, simple to understand & all the features / libraries should have a unique approach to implement. Syntax should be easier & understandable.

b). Orthogonality: This is explained as, the programming language should provide many different ways to achieve the same goal. (Eg: in C we can parse tree using "for loop" iteration & function recursion). So, it provides different ways to achieve the same goal where developer chooses the better one to be implemented.

c) Support for abstraction: Programming language should provide the data abstraction mechanism. Data abstraction means to fetch the necessary data from object.

d). Ease of program debugging: It should have excellent debugging facility to debug errors. It also should have capability of program reviewing and warning generation.

e). Programming environment: To save the time of coding, programmer should prefer to use 3GT (third generation tools) like eclipse, Android Studio, visual Studio that supports automatic code generation & thus saves much coding time.

f). Portability of programs: Program should be portable, i.e. it means the same program should be able to work on all platforms without any prior changes.

g). Cost of using: Developer should focus on less costly programming language in relation to usage. Eg: Java is free & open source, but .NET need visual studio IDE to be purchased so have costing along with it.

* History and Motivation.

1. Programming is Difficult.

→ As soon as first computer were built, it became obvious that programming was very difficult.

→ Necessity of dealing with many different details at one time.

→ Programs may contains millions of Lines of code, management of them was difficult.

→ Storage problem was also obvious, and also very slow.

→ Complicated / Calculation of arithematic expression was difficult too.

→ Error-prone.

2. Many Program Design Notations were Developed.

→. Due to the above situation, programming led to the development of program design notations, like flow Charts.

→ Flow chart was used to simplify complexity of the programming which helps to design memory layout and control flow of the program.

3. Floating point and Indexing were Simulated.

→. The earliest computers did not have built in floating point operations. Most of the machines were in scientific and numerical computations, which required wide range of magnitudes. For this manual scaling was used, where the numbers were multiplied by scale factors in order to keep them within the range of the integer facilities of the computer. This was also difficult which led to the development of floating point subroutines, which perform basic floating point operations (addition, subtraction, multiplication, division, square root etc).

Indexing was missing in earliest computers. They didn't have facility of index registers and index addressing mode.

4. Pseudo-Code Interpreters were invented:
- It is simpler version of programming code before it is implemented in specific programming language.
- Pseudo-code - an instruction code that is different from that provided by machine & probably much better.
- It had its own set of data types (floating point) & operation (indexing) in terms of real computers.
- own memory means along with time saving techniques Everybody felt this was good idea and many pseudo-code interpreters were born.

They followed 2 principle

(1) Automation Principle
"Automate mechanical, tedious, or error-prone activities."
It provided facilities more suitable to the application and eliminated many details from programming.

(2) The Regularity Principle
"Regular rules, without exceptions, was easier to learn, use, describe, & implement."
The virtual computer was more regular, simpler to understand through the absence of special cases."

(5) Compiling Routines were also used:
→ At the same time, another approach was being used for implementing pseudo-code - "compiling routines."
→ Compiling routines were used, which was faster.
→ "compiler- special program that translates a programming language's source code into machine code, bytecode or another programming language."

* Phenomenology of Programming Language.

1. Tools are ampliative and Reductive
   (आवर्धनात्मक) (न्यूनीकरणात्मक)

→ Stick is ampliative, experience of fruit - reductive - it is mediated by stick.

- Technological utopians tends to focus on ampliative aspect.
  - Ampliative aspects → The increased reach and power to ignore reductive aspect.
  - Reductive aspects → The loss of direct, sensual, experience and to diminish the practical advantage of tool.

→ But both of them reduced focuses upon different dimensions of the human technological experience. We should acknowledge the essential ambivalence (mixed feeling) of our experience of the tool: positive in some aspects, negative in others.

— In earlier days, computer were programmed directly with patch cords. Programming was direct, some program ner criticized. Pseudo code (even more distancing. Helped in writing code but reduced their contact with control over the machine.

2. Fascination and fear are Common Reaction to New Tools.
   - Utopians tend to become fascinated with ampliative aspects of new tools, so they embrace new technology & were eager to promote it.
   → Dystopian fear the reductive aspect of the tools. (higher level fear for inefficiency of tools).

3. With Mastery, Objectification Becomes Embodiment. (attached with body.)

→ With Mastery, good tools becomes transparent. When you encountered new tools (new programming language, it is experienced as an object, something to be learned about. As you gain skill, it becomes transparent and you can program. You require full evaluation of programming languages.

4) Programming Languages Influence focus and Action.

→ Tools influences focus and action. It influences focus by making some aspects of the situation salient and by hiding others; it influences actions by making some easy and others awkward.

Eg of three writing technologies: a dip pen, an electric type writer and a word processor.

Chapter 2: Emphasis on Efficency - FORTRAN.

2.1. History and Motivation.

1) Automatic Coding was Considered Unfeasible.

- In early 1950, there was little sympathy for the idea of a high level programming language.
→ Many programmer did not like the idea of decimal numbers in programming.
→ John Backus of IBM designed one system, Speedcoding, which became popular. (included floating point numbers and indexing).
→ For the generation of code with effcency and to even make it cheaper, in Late 1953, Backus proposed the ideas to his management at IBM and in January 1954, with one assistant. He began to work on FORTRAN

2). Preliminary FORTRAN was designed.

→. By November 1954, Backus and three associates had produced a preliminary external specification for the "IBM Mathematical FORmula TRANslating system, FORTRAN.

3). The FORTRAN Compiler was Successful.

→. Backus began to work on the design and programming of FORTRAN in early 1955 and released the system in April 1957. It got accepted in 1958.
→ Used some sophisticated optimization which later helped deliver the efficency by the FORTRAN.

4). FORTRAN has been revised Several times.

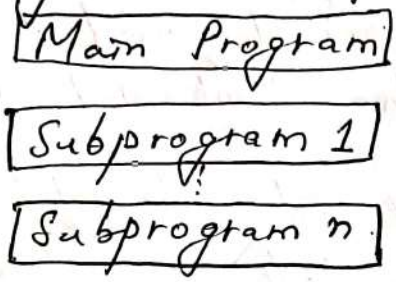
→ Experience of FORTRAN system led to propose FORTRAN II 1957.
→ In 1958, FORTRAN II was designed & implemented.
→ In 1962, FORTRAN IV was designed.
→ In 1977, a new ANS FORTRAN developed, also known as FORTRAN 77.
→. After 12 years of development process a new standard, commonly known as FORTRAN 90 was produced.

2.2. Design: STRUCTURAL ORGANISATION.

1. Programs are Divided into Disjoint Subprograms.

→ Pseudo-code lacked the idea of subprograms (eg: procedures, functions & subroutines). In FORTRAN, they were included.

→ The program contains one main program and zero or more sub-programs. Example:



2) Constructs are either Declarative or Imperative:

→ The pseudo-code programs were divided into 2 parts, which were i) a declarative part = which describes data areas, lengths & initial values.

ii) an imperative part = contained commands to be executed which programs obeys at run-time.

→ In FORTRAN also, both of the above parts were present but named differently.

→ Declarative constructs was called nonexecutable statements.

→ Imperative constructs was called executable statements.

3) Declarations include Bindings and Initializations.

→ FORTRAN performed 3 main functions (Same as pseudocode)

(a) They allocated an area of memory of specified size.

(b) They attached a symbolic name to that area of memory. This is called binding a name to an area of memory.

(c) They initialized the contents of that memory area.

→ Declaration. Eg: DIMENSION DTA (900) → causes loader to allocate 900 words and to bind DTA to this area.

→ Data Declaration was used for initialization. for example:

DATA DTA, SUM /900*0.0, 0.0/
this would initialize the array DTA to 900 zeros and the variable SUM to zero.

4). Imperatives are either Computational, Control-flow or Input-Output.

→ FORTRAN provides imperatives statements like
@ Computational statements → arithematic and move operations.
(b) Control-flow statements → comparision & looping statements.
(c) Input-Output statements → READ & PRINT.

Eg: Arithematic : Avg = sum / float(N).
Control-flow : if (condition) then
body of if
endif
Input-Output : read *, variable
print *, variable.

(5) A program goes through several stages to be RUN.

→ Series of stages.
(i) Compilation - translated individual FORTRAN subprograms into relocatable object code. Addresses assigned at load time.
(ii) Linking: It address the need for incorporating libraries of already programmed, debugged and compiled programs.
(iii) Loading: It is the process in which the program is placed in computer memory.
(iv) Execution: The control of the computer is turned over to the program in memory.

(6) Compilation involves three phases.

→ Determines the efficiency of final program.
(1) Syntactic analysis: The compiler must classify the statements and constructs of FORTRAN and extract their parts.
(2) Optimization: Efficiency is primary goal of FORTRAN. The experienced programmer performed this task.
(3). Code synthesis: To put together the parts of object code instructions in relocatable format.

Control structures.

① **Control Structures Govern Primitive Statements.**

→ govern the flow of the program. (same as Pseudo code).
→ direct controls to various primitive computational & input output instructions such as ADD READ.

② **Control Structures were based on IBM 704 Branch Instructions.**

→ originally designed as a programming language for the IBM 704 computer.
→ Sometimes called assembly language for IBM 704.
→ Initially machine dependency feature was included in FORTRAN which violates the portability principle.
→ Portability principle. avoid features or facilities that are dependent on a particular computer or a small class of computers.
→ Eg: The Arithmetic IF Statement:
if (e) n_1, n_2, n_3, evaluates e and branch to n_1, n_2, n_3 depending on the result of the evaluation, i.e is −, 0 or + respectively.
- compares the value of accumulator with a value in storage and then branches to one of 3 locations.

→ The arithmetic IF was not very satisfactory for many reasons.
- Diffult to keep the meaning of 3 labels and in fact 2 seem similar.

→ Later more conventional Logical IF statement was added. For example:

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

↳ uses EQ for equality relation.

③ **The GOTO is the workhorse of Control Flow.**

→ In FORTRAN, GOTO statement is the raw material from which control structures are built. Two way branch is often implemented with logical IF statement & a GOTO

P.T.O.

if (condition) GOTO 100
... case for condition false ..
GOTO 200
... case for condition true .....

- We can see that this corresponds to the if-then-else or conditional statement of newer programming languages.

→ The use of IF statement divides the control flow into 2 cases. That cases uses ① a computed GOTO. Eg.
GOTO (L1, L2, L3 ... Ln), I where Li are the statement label numbers & I is an integer variable.
If I = 3,
GOTO (10, 20, 30, 40, 50), I → In this case control is passed to label number 30 (∵ I = 3 = 3rd value)

→ Concept of selection statements was introduced.
for eg: if-then-else
Loop statements.
while-do → indefinite iterations.

④ It is difficult to correlate Static and Dynamic Structures.

→ It was possible to implement almost any control with it - those that are good but also bad as well.
→ The control structure should be understandable to the users (good appearance to the users). दिखने में अच्छा हो।
→ The undisciplined used of GOTO statements permits the contruction of very intricate control structures, which are hard to understand.

⑤ The Computed and Assigned GOTOs are easily Confused.

- Two statements is FORTRAN computed GOTO and assigned GOTO, illustrate the pitfalls that await the language designer.
- We have seen computed GOTO above already.
- FORTRAN also uses Assigned GOTO for branching to a number of different statement.

P.T.O.

Eg: GOTO N (L1, L2, L3 ... Ln)
where N is integer → This statement transfer to the value whose address is in variable N.

Assign 20 to N (Put the address of statement no. 20 in integer variable N)
GOTO N, (20, 30, 40) Transfer control to label 20.

- The effect of ASSIGN 20 TO N is to put the address of statement number 20 in the integer variable N. which is completely different from N=20.
- In general, address of statement number 20 will not be 20. (You have to put some other number into N.
- They looked confused (computed & assigned GOTO). & identical.
- Let us look the case of writing a computed GOTO where an assigned GOTO is intended.

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

→ The assign statement will assign the address of statement number 20 to N.
→ The Computed GOTO will then use this as an index to jump table. (20, 30, 40, 50). In this case, the above 20 will be out of range.
→ The program moves to unpredictable memory location.
→ Syntactic Consistency Principle should be followed i.e, "Things that look similar should be similar & things that look different should be different."
→ "Defense in depth principle": "If an error gets through one line of defense, then it should be caught by the next line of the defense."

① The Do-Loop is more Structured than the GOTO.
- Do loop provides simple method of constructing a counted loop (definite iteration)

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

⇒ is a command to execute the statements between Do and corresponding CONTINUE with I taking on the values 1 through N.

→ The variable which changes are controlled variable (I).

→ The variable which change are controlled variable (I) and the statements between Do & CONTINUE is called body of loop.

(7) The Do loop is higher level

- The Do loop is called higher level because it allows programmers to state what they want rather than how to do it.

→ Incorrect testing for loop is corrected before.

→ Do Loop illustrates "The impossible error principle" → Making errors impossible to commit is preferable to detect them after their commission"

(8) The Do-Loop can be Nested.

→ Do-loop can nested, i.e., constructed hierarchically.

```
Do 100 I = 1, M
   :
   DO 200 J = 1, N
   :
   CONTINUE
```

CONTINUE

→ should be nested correctly. Regular flow of control.

(9) The Do Loop is highly Optimized.

→ Preservation of Information principle:

"The language should allow the representation of information that the users might know & that the compiler might need."

→ faster indexing, incremented & tested directly, controlled variable etc.

10) Subprograms were an Important, Late Addition.

→ Fortran I prohibited the use of subprograms. But in FORTRAN II the deficiency was solved with the addition of SUBROUTINE and FUNCTION declarations.

→ Subprograms define procedural abstraction because
- repeated code can be abstracted out
- variables formalized
- allow large program to be modularized.

→ Instead of repeating large 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. formal is a list of name of formal parameters also called dummy variable - formal bounds to actuals.

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

This subroutine can be invoked by a Call statement, i e the program fragment can be written as

```
:
CALL DIST (DIST1, X1, Y1)
:
CALL DIST (DIFFER, POSX, POSY).
```

In the first case we bind D to DIST1, X to X1 & Y to Y1. In the second we bind D to DIFFER, X to POSX & Y to POSY. By this means the variables in the caller's environment can be accessed by referring to the formal parameters in the callee's environment (the callee is the subprogram being called)

11) Subprograms allow large programs to be modularized:
- Saves writing
- total difficulty of ~~of~~ design process can be decreased. larger problems are break down to smaller problems, subprograms can be written, debugged and read as independent unit.

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

12. Subprograms encourage Libraries:
- Separate compilation
- independent to each other.
- they can be seperately translated to relocatable object code by the compiler & later combined by linker.

13. Parameters Are Usually Passed by Reference.
→. This technique always passes just the reference to the actual parameters i.e, address
→ Since references are usually small (one word or less), this means that very little information is passed from caller to callee.
→ The technical term used in programming languages for an address of a location in memory is a reference because it refers to a memory location.
→ Pass by Reference is always efficient
→ Suppose that the actual parameters were array elements, for example:
CALL DIST (D(I), POS(I), POS(J))
Here address of the POS(I) is easily computed from the address of the first element of POS and the value of the index I. Here all the caller has to do is, compute the addresses of the array elements, D(I), POS(I) & POS(J) & pass this to caller.

→ It is necessary to pass the size of array to the subprogram, unfortunately sometimes wrong array size may be passed which causes programmer to abort the program or program may behave mysteriously. It violates "The Impossible Error Principle."
"Making errors impossible to commit is preferable to detecting them after their commission."

14. Pass by Reference has Dangerous Consequences. (imp)

- The most serious problems with passing all parameters by reference result from the assumption that all parameters can potentially be used for both input & output.
→ This can be dangerous if the actual parameter is a variable since it means that an input variable may be inadvertently updated by a subprogram which introduce bug in the program that is difficult to find. This is the side effect of the subprogram call.

→ Eg: SUBROUTINE SWITCH (N)
N = 3
RETURN
END

Here switch writes 3 in to its parameters. Therefore, the invocation CALL SWITCH(I) would result in 3 being stored in I.

- Now consider the invocation CALL SWITCH(2) which will assign address of CONSTANT 2, which contains 3 i.e, when subroutine assigns 3 to its parameter, it will overwrite the value 2 in the literal table. The problem with this is that the compiler will compile a program so that it loads the contents of this location whenever it needs a constant 2, even though that location now contains 3. So, if the program now executed the assignment I=2+2, the value stored in I would be 6. We have used the SWITCH procedure to change the "constant".
- Thus you can imagine the debugging difficulties.

15) Pass by Value - Result is Preferable.

→ Pass the value by making a copy of actual parameters to the formal parameters.

16) Subprograms are implemented Using Activation Records.

→ Problem: Saving state of caller (program counter and Register) transmitting parameters.

- All of the information must be stored in a caller private data area before the callee is entered. This data area is often called an Activation Record because it holds all information Relevant to subprogram.

→ CALL STEPS:

1. Place the parameters in the callee's activation record.
2. Save the state of the caller in the caller's activation record (including the point at which the caller is to resume execution).
3. Place a pointer to the caller's activation record in the callee's activation record.
4. Enter the callee at its first instruction.

→ RETURN STEP:

1. Jump to resumption address in Callers' Activation Record (found using dynamic link).
2. When caller regains the control, restore the rest of the state of its execution from its Activation Record.

DESIGN: DATA STRUCTURES (General idea imp.; not detail)

1) Data Structures were suggested by Mathematics.

→ FORTRAN – is a scientific language. Therefore data structuring methods included in FORTRAN were those most familiar to scientific & engineering applications of mathematic: Scalars and arrays.

2). The Primitives are Scalars:

→ The numeric scalars are the primary data structure primitives provided in FORTRAN.

→ The FORTRAN II language included a type DOUBLE PRECISION and arithmetic operations on numbers of the type like (Complex numbers, Logical, floating point numbers twice as large as the normal floating numbers).

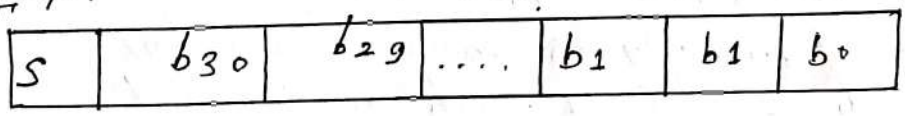
3). Scalar Types are represented in Different Ways:

→ Computers of the early 1960's were mostly word oriented that is the basic addressable unit of storage was a word.

→ Integer, floating point and logical values all normally occupied one word.

→ Character also typically would fit on one word.

→ Integer were represented as a binary number with a sign bit:



i.e, 32-bit pattern.

→ The number represented by this pattern is

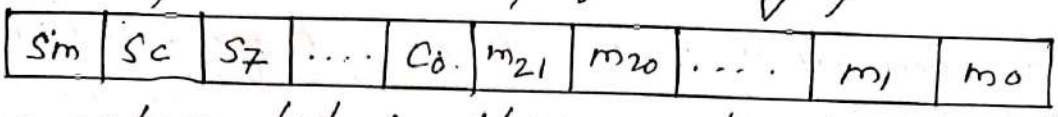
$$(-1)^S \sum_{i=0}^{30} b_i 2^i$$

→ Operation on integer: Arithmetic of $e^n (+, -, /, *)$
- test for zero
- test of the sign bit.

→ Float: represented as coefficient & power of 10.

eg: $-3.2 * 10^{12}$

→ The typical representation of floating point number is



The value represented by this number is $m \times 2^C$, where m is mantissa,

$$m = (-1)^{Sm} \sum_{i=0}^{21} m_i \cdot 2^i - 22$$

and C is the characteristic where

$$C = (-1)^{SC} \sum_{i=0}^{7} C_i \cdot 2^i$$

→ Operation of FORTRAN follow closely those typically implemented on a computer; i.e.
- → Arithmetic operation
- → Comparisons
- → Absolute values.

47. <u>The Arithmetic Operators are Overloaded:</u>

→ Overloading: One name, multiple meaning.

→ '+' can add integer, real, complex, double precision.

→ Old FORTRAN: Real numbers / variable X to be added to integer variable I, Necessary to convert I to float or X to integer (IFIX). Eg: X + FLOAT(I)

I = IFIX(X).

→ Later version allowed mixed mode expressions, that is expression of more than one mode, or type. Such as expression is I = X + I, which is interpreted to mean

I = IFIX(X + FLOAT(I))

→ We say that I has been coerced from integer to real and X+I has been coerced from real to integer.

→ For example: if we let X be a real variable or expression, and I be an integer variable or expression, then the FORTR -AN coercian rules are

X + I ⇒ X + (FLOAT(I)

I + X ⇒ FLOAT(I) + X

X - I ⇒ X - FLOAT(I)

X = I ⇒ X = FLOAT(I)

I = X ⇒ I = IFIX(X)

→ Newer FORTRAN permits mix-mode expression but with surprising results.

→ For example: If we write X**(1/3) to compute cube root of a real variable X, we will get wrong answer. This is because 1/3 is an integer division, which is truncating and returns 0. Thus X is raised to the zeroth power. Therefore we must write X**0.333333333 or X**(1.0/3) to ensure the exponent is a real number.

5) The Integer Type is Overworked.

→ In FORTRAN, the integer type required to do double duty i.e, it represents both integers and character strings.

→ Hollerith constant is considered to be of type integer. It was early form of what is now called a character string. For example. 6HCARMEL represents 6-character string 'CARMEL'.

6 - Literal integer
H - Hollerith
CARMEL → number of characters specified by the literal integer.

→ So, FORTRAN permits character strings to be read into integer or real variables and permits Hollerith constants to be used as parameters where integers are expected.

Eg:
```
      FUNCTION ISUCC(N)
      ISUCC = N+1
      RETURN
      END
```

FORTRAN permits us to write

N = ISUCC(6HCARMEL)

even though this make no sense. It is the security loophole and example of weak typing.

6) The Data Constructor is the Array.

→ Linguistic method is used to build complex data structure from the primitives are called the constructors. for data structures.

→ In FORTRAN, the only data structure constructor is the array.

7) The ~~Data Structure is the Array~~ Array are Static & Limited to Three Dimensions.

→ In FORTRAN arrays are declared by means of a DIMENSION statement. For example

DIMENSION DTA(100), COORD(10,10)

declares DTA to be a 100 element array (with subscripts in the range 1-100) and COORD to be a 10×10 array (with subscripts in the range 1-10).

→ Because the size is fixed it is called static.

→ Limited to 3 dimension which violates (zero-one infinity) principle)

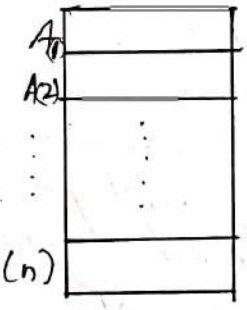
→ DIMENSION changed, it will be difficult to tell which of these numbers need to be changed and which do not.

8) Array Implementation is Simple and Efficient.

→ Consider a first one-dimensional array A declared by

DIMENSION A(n)

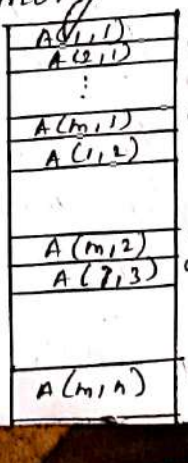
where n represents an integer denotation. The layout of the memory is



→ If we let $\alpha\{A(1)\}$ mean address of the memory of A(1), then A(2) is $\alpha(A(1)+1$, A(3) is $\alpha(A(2)+2$ and so on.

Similarly $A(n) = \alpha\{A(1)\}+n-1$. In general we have $\alpha\{A(i)\} = \alpha\{A(1)\}-1+i$ → This is called addressing equation of the array A.

→ Now derive equation for two dimension array. Suppose A is dimension declared by A(m,n), then the FORTRAN arranges array in memory in column order, that is, with the columns occupying adjacent memory locations as shown in figure where $\alpha = \alpha\{A(1,1)\}$



→ From the figure, we can see that, the address of A(I, J) is $[\alpha+(J-1)m+I-1]$

→ ∴ Address of 2 Dimensional array is $[\alpha\{A(I,J)\} = \alpha\{A(1,1)\}+(J-1)m+I-1]$

9) FORTRAN Arrays Allow Many Optimizations:

→ It is crucial that FORTRAN make optimum use of the index registers. Lets take a simple Do-Loop.

```
    Do 20 I = 1, 100
    SUM = SUM + A(I)
20  CONTINUE
```

→ The above implementation of this loop is to initalize to 1 an index register and increment the index register on each successive iteration.

→ We can compute address of the array by straightfoward method which we have done previously like

$\alpha\{(A(1)\} - 1 + IR$

→ The loop can be summarized by the code as

```
        Initialize IR to 1;
        Compute the address α{A(1)} - 1 + IR;
Loop:   Fetch the contents of this and add to SUM;
        Increment IR;
        If IR is less than or equal to 100, goto loop.
```

→ But it contains unnecessary addition. Smart programer would initalize the index register to the address of the first of the array and then using the index register as an indirect address for the array element.

```
        Initialize IR to α{A(1)};
Loop:   Fetch the contents of the location whose address
          is in IR and add to SUM,
        Increment IR;
        If IR is less than or equal to α{A(100)},
            goto Loop.
```

→ Such optimization was implemented in the first FORTRAN compiler.

2.5: DESIGN: NAME STRUCTURES:

1). The Primitives bind Names to Objects:

→ The purpose of the name structures are to organize the names that appear in the program.

```
INTEGER I, J, K
```

declares I, J, K to be the integer variables. This means that storage must be allocated for these variables (one word for each) and the names I, J, K must be bound to the addresses of the location.

→ This declaration doesnot provide the initialization function.

→ The other important ~~import~~ function fulfilled by this declaration is to specify the type of variables namely, INTEGER.

2). DECLARATIONS ARE NONEXECUTABLE.

→ The type in the declaration (eg: INTEGER) is used to determine the amount of storage that must be allocated for the variables.

→ Since this allocation is done once, before the program is executed & never changes, it is called static allocation.

→ Integer I, J, K make entries in the symbol table for each of I, J, K. These entries will contain the location of these variables in memory & their types.

Name	Type	Location
⋮	⋮	⋮
I	INTEGER	0245
J	INTEGER	0246
K	INTEGER	0247
⋮	⋮	⋮

3). Optional Variable Declarations are Dangerous:

→ If a programmer uses a variable but never declares it, the declaration will be automatically supplied to the compiler, by using "I through N" rule.

→ This mean that any variables whose names began with I through N are declared integers and all others are declared reals.

→ This rule has the unfortunate consequences since programmers pick obscure names.

→ For example, if the programmer write COUNT = COUMT+1, (M instead of N), then since the variable COUMT has not been declared, the compiler would automatically declare it as a real & this case, i.e, it will have the garbage value (Memory में पहले से पड़ा value)
→ The result will be strange and inexplicable and it will be difficult to debug.
→ Violates the Security Principle.

4) Environment determine Meanings.
→ The meaning of the sentence often depends on the context in which it is stated: words can have different meaning depending on their context.
→ The same logic applies here too.
X = COUNT(I) may have many meanings.
X = may be either real or integer.
COUNT = may be either function or array.

5). Variable names are Local in Scope:
→ Variable those defined inside subprograms are visible only inside that subprograms, which follows "Information Hiding".

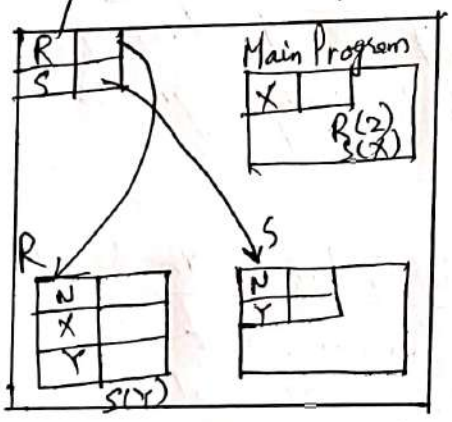
6). Subprograms Names are Global in Scope:
→ Till now we came to know that FORTRAN specifies that subprogram names are visible throughout the program, they are thus said to be global.
→ In contrast we say that variable names are local to subprograms in which declared.
→ Thus names fall into 2 broad classes, depending on visibility. This property is called scope of a name.
→ Subprogram name have global scope
Local Variables name has local scope.

Example:

```
Main Program
Integer X   (*)
:
CALL R(2)
:
CALL S(X)
:
END
SUBROUTINE R(N)
REAL X, Y   (*,*)
:
CALL S(Y)
:
END
SUBROUTINE S(N)
INTEGER Y   (***)
:
END.
```

∴ Eg: of variable scope in FORTRAN.

→ The relationships can be depicted in contour diagram.



→ R can see the bindings of S and its own Y, but it cannot see the Y in S or the X in main program.

→ Contour diagram are valuable aid to visualize scopes.

7) It is Difficult to Share Data with just Parameters.

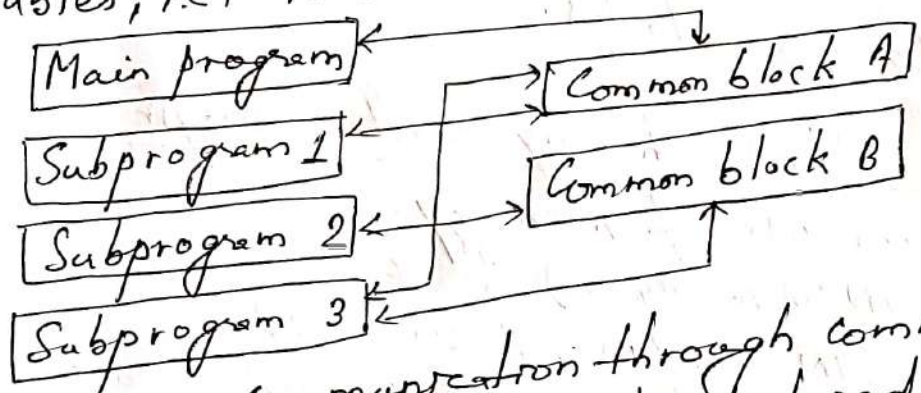
→ The way to pass information from one subprogram to another is through parameters. But it is sometimes very inconvincing to do so. If you enter a declaration for an array whose name is in NM, location AVAIL, type code INTCOD & dimensions M & N, we write

CALL ARRAY2 (NM, AVAIL, INTCOD, M, N, NAMES, LOC, TYPE, DIMS).

which is not very readable. Four extra parameters clutter of the call.

8) Common Blocks allow sharing between Subprograms.

→ Common block permit subprograms to have common variables, i.e. to share data areas.



∴ Subprograms communication through common blocks.

→ Every common block must be declared in every program unit in which the information stored therein is needed.

8) Common permits Aliasing, which is Dangerous.

→ While using common blocks, we see to that each subprograms includes an identical specification of the SYMTAB COMMON block. But all the specification will not agree.

COMMON / SYMTAB / NM(100), WHERE(100), MODE(100), SIZE(100).

Here VAR's use of WHERE(I) would be equivalent to Arrays use of LOC(I). But unfortunately this could occur accidently.

→ The programmer must reverse the LOC and TYPE array in one subprograms.

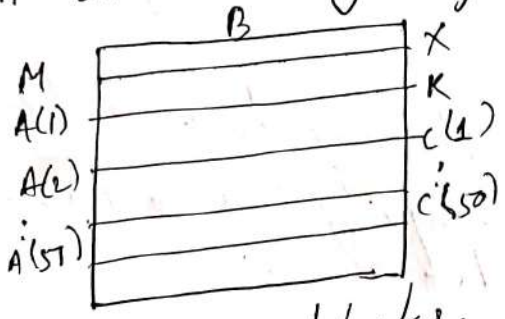
COMMON / SYMTAB / NAMES(100), TYPE(100), LOC(100), DIMS(100).

→ The problem is more serious. For example:

COMMON /B/ M, A(100) — one subprogram.

COMMON /B/ X, K, C(50), D(50) → second subprogram.

→ We see that location called M in the first subprogram is called in X in second program. These two variables don't even have the same type. If programmer stores an integer in M & then uses X as a floating point number, the result will be glibberish. It is security loophole.



∴ Aliasing in Common blocks.

⇒ EQUIVALANCE allows sharing within Subprograms.

→ To make optimum use of memory.

Eg: DIMENSION INDATA (10,000) RESULT (8000)

EQUIVALANE (INDATA(1), RESULT(1).

This states that first element of INDATA is to occupy the same memory location as the first element of RESULT.

* Design of a pseudo-code.

1) Basic capabilities must be decided.

→ For scientific computation what type of data (floating point number, array of floating point numbers) and what function;?, indexing control flow, looping should be determined first.

Our pseudo-code must accomodate

- Floating point arithematic
- Floating point comparisons
- Indexing
- Transfer of control
- Input - Output.

X

- FORTRAN :

SYNTACTIC STRUCTURE

(ASSIGNMENTS)

3: ALGOL-60 : GENARALITY and HIERARCHY

3.1: History and Motivation:

1) An international language was needed:

→ It became apparent to some computer scientists that a single, universal, machine-independent programming language would be valuable. The main problem was "portability".

2) The ALGOL -58 Language was Developed:

→ The European members brought several years of work on algebraic language design and Americans brought their experence in implementing pseudo-codes and other programming system.

The objectives of new language as stated in ALGOL-58 report were

i) New language should be as close as possible to standard mathematical notation and be readable with little further explanation.

ii) It should be possible to use it for the description for computing processes in publications.

iii). The new language should be mechanically translatable into machine languages/programs.

3) A Formal Grammer was used for Syntactic Description.

→ At one of the conferences on Information Processing held by UNESCO in Paris in June 1959, John Backus presented a description of ALGOL using formal syntactic notation he had developed. But Peter Naur, did not agree with John's idea and began to work on a more precise method of describing syntax. Peter prepared some samples of a variant of Backus notation. Thus this notation was developed for the ALGOL-60 Report and it is now known as BNF, Backus-Naur Form, reflecting the contribution of both men.

4) ALGOL-60 was designed:

→ 13 members of ACM and GAMM met in Paris for 6 days in January 1960 and prepared a final report. The resulted language was ALGOL-60 & published in May 1960.

→ The resulting language was remarkable for its generality and elegance.
→ It was "more of a racehorse than a CAMEL".
→ Few errors were corrected and final "Revised Report on the Algorithmic Language ALGOL-60" was published in the 'Communications' in January 1963.

5). The Report is a paradigm of Brevity and clarity:
→ While other programming languages were stretched upto hundreds to thousands pages, ALGOL-60 Report was only 15 pages long.
→ It was possible since it used BNF notation, which provided simple, concise, easy-to-read, precise method of describing ALGOL's syntax.
→ The committe decided to use clear, precise and unambiguous English language descriptions, which resulted in a report that was readable by potential users, implementers & language designers.

3.2 DESIGN: STRUCTURAL ORGANISATION

1) ALGOL programs are hierarchically structured:
→ One of the primary characteristics and important contributions of ALGOL is its use of hierarchical structure.
→ ALGOL program is composed of number of nested environments. It decrease GOTO statements.

2). Constructs are either declarative or Imperative.
→ The constructs of ALGOL-60 can be divided into 2 categories
i) Declarative - binds names to objects (variable & procedure)
ii) Imperative - do the work of computation.
→ Three kinds of declaration — a) Variable declarations
b) Procedure declarations
c) Switch declarations.
→ Variable declaration - datatypes - integer, real, Boolean.
Eg: integer i, j, k; real Array Data[-50:50]

→ ALGOL uses term procedure to refer to a subprogram. Procedure declarations are required to specify the types of their formal parameters. eg:

```
real procedure dist (x1, y1, x2, y2);
    real x1, y1, x2, y2;
dist := sqrt ((x1-x2)↑2 + (y1-y2)↑2)
```

→ Switch declarations serves same function as the FORTRAN computed GOTO.

3). Imperatives are Computational and Control Flow:

→ Two classes of imperative constructs in ALGOL-60,
  i) Computational
  ii) Control Flow.

→ There are no input/output constructs in ALGOL-60. The input-output be handled by library procedures.

→ Arithmetic ⇒ (+, −, /, *)
  Relational (Boolean o/p) ⇒ =, >, ≤
  Boolean Operators ⇒ (∧, ∨, ¬)
  Assignments (:=)

4). ALGOL has the familiar Control Structures:

→ The goto statement, if-then-else, the for-loop etc.

5) The Compile time, run time distinction is important:

→ FORTRAN was completely static, ie, all data areas are allocated and arranged by compiler.

→ In case of ALGOL, Dynamic array and recursive procedure is possible.

→ ALGOL data structure have later binding time (ie, name is bound to a memory location at run-time rather than compile time).

5) The Stack is the central Run-time Data Structure:

→ ALGOL uses stack to hold activation record for procedures & blocks.

→ Pushing & popping these activation record → dynamic allocation & deallocation is achieved

3.3: DESIGN: NAME STRUCTURES:

1. The Primitives Bind Names to Objects:

↪ The name structures are same as FORTRAN with one major difference.

→ In FORTRAN a variable name is statically bound to a memory location, whereas in ALGOL we will find that a single variable may be bound to a number of different memory location and these bindings can be changed during run-time.

2) The Constructors is the Block:

→ One of the important contributions of ALGOL-58 was the concept of compound statement.

→ This allows sequence of statements to be used wherever one statement is permitted.

→ For instance, although one statement would normally form the body of a for-loop, such as

```
for i := 1 step 1 until N do
    sum := sum + Data[i]
```

several statements can form the body if they were surrounded by begin-end brackets.

```
for i := 1 step 1 until N do
  begin
    if Data[i] > 1000 000 then Data[i] := 1000000;
    Sum := Sum + Data[i];
    Print Real (sum)
  end.
```

3). Blocks Define Nested Scopes:

→ ALGOL provides programmer to define any number of scopes nested to any depth.

→ ALGOL-60 avoids the redeclaration of scope; this is accomplished with a block.

```
begin declarations; statements end
```

Ex:

```
begin
    real x, y;
    real procedure cosh(x); real (x);
        cosh := (exp(x) + exp(-x))/2;
    procedure f(y, z);
        integer y, z;
    begin real array A[1:y];
        :
    end.
    begin integer array count[0:99];
        :
    end
    :
end.
```

4) Blocks simplify constructing large programs:

→ In FORTRAN COMMON was designed to allow sharing data structures among a group of subprograms. But the problem with COMMON is that, COMMON declaration must be repeated in each subprogram which is wasteful and a potential source of errors.

→ ALGOL block structure solves this problem since the symbol table arrays can be factored out into a block that surrounds the symbol table management procedures

→ Since FORTRAN COMMON blocks must be redeclared in every subprogram that uses them, there is a possibility that these declaration may be mutually inconsistent which can cause bugs that are difficult to find. In ALGOL, shared data structures are defined once, so there is no possibility of inconsistency.

"Violation of Information Hiding Principle"

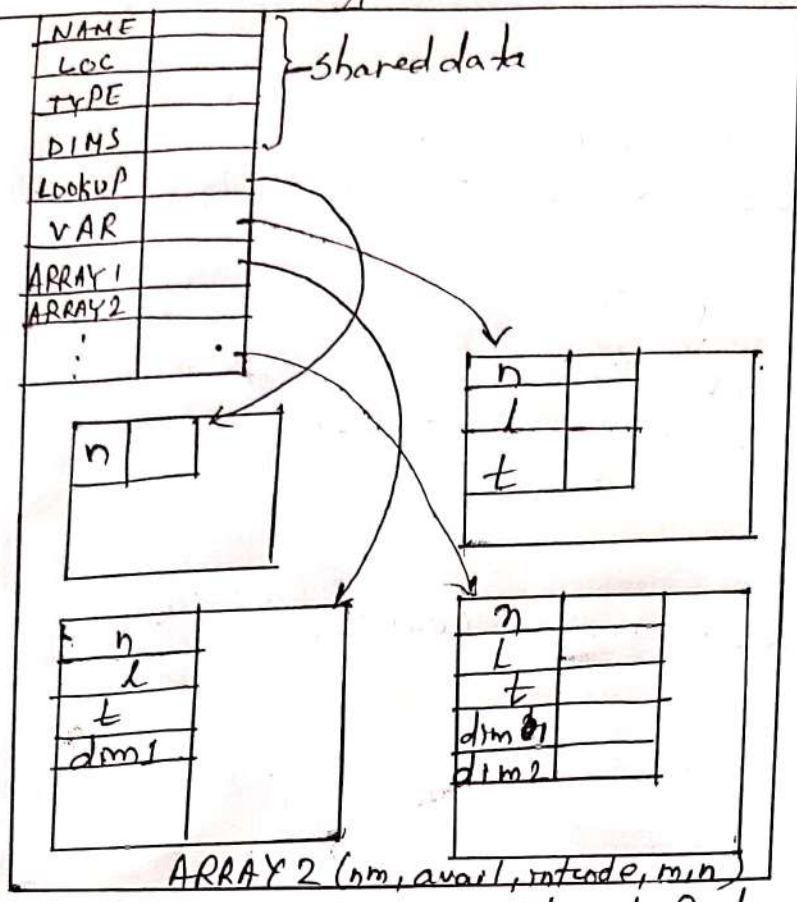


Fig: Contours showing Shared Data.

⇒ Dynamic Scoping allows the Context to Vary

→ There are two scoping rules that can be used in block structured languages static & dynamic scoping.

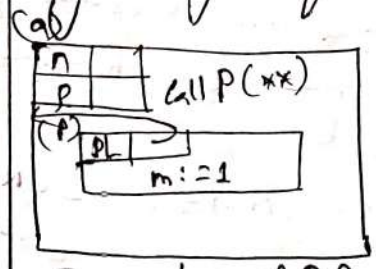
→ Dynamic scooping: A procedure is called in the environments of its caller.

Eg:
```
a: begin integer m;
     procedure P;
       m := 1;
     b: begin integer m;
          P ..... (*)
        end
        ..... (**)
     P
   end
```

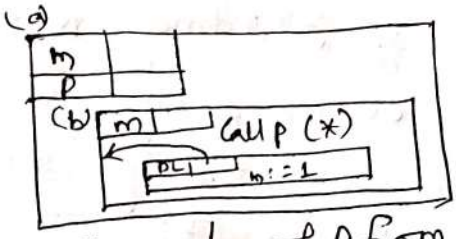
→ With dynamic scoping, the assignment m := 1 refers to the outer declaration of m when P is called from the outer block (**) and the inner declaration of m when P is called from the inner block (*).

→ The invocation from block b, which is nested in block a, we can see that the identifier m in m:=1 refers to the variable declared in block b. i.e P is called in the environment of caller (i.e, dynamic scoping).

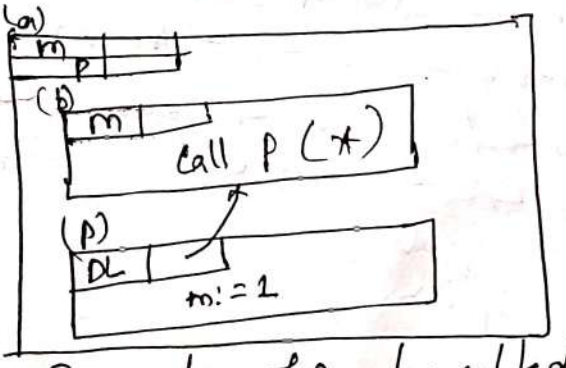
→ With static scoping, the assignment m:=1 always refers to the variable m in the outer block, i.e, P is called (always) in the environment of definition ie, the context in which P is executed is always the context in which it was originally defined.



∴ Invocation of P from Outer Block (a)



∴ Invocation of P from Inner Block (b)



∴ Invocation of P when called is Environment of Definition.

[Book page No: 108/109 for figure]

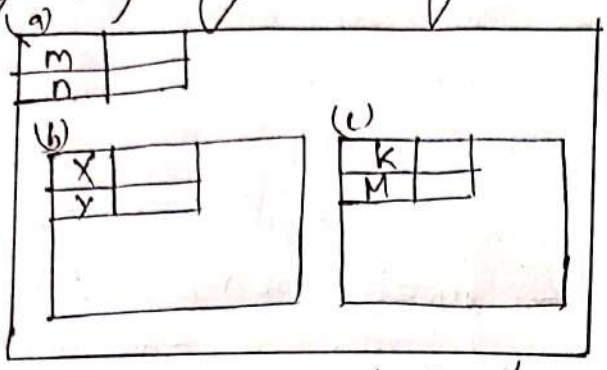
5) BLOCKS permit Efficient Storage management on a stack.

→ Conserving memory for using it for multiple purposes is important part.

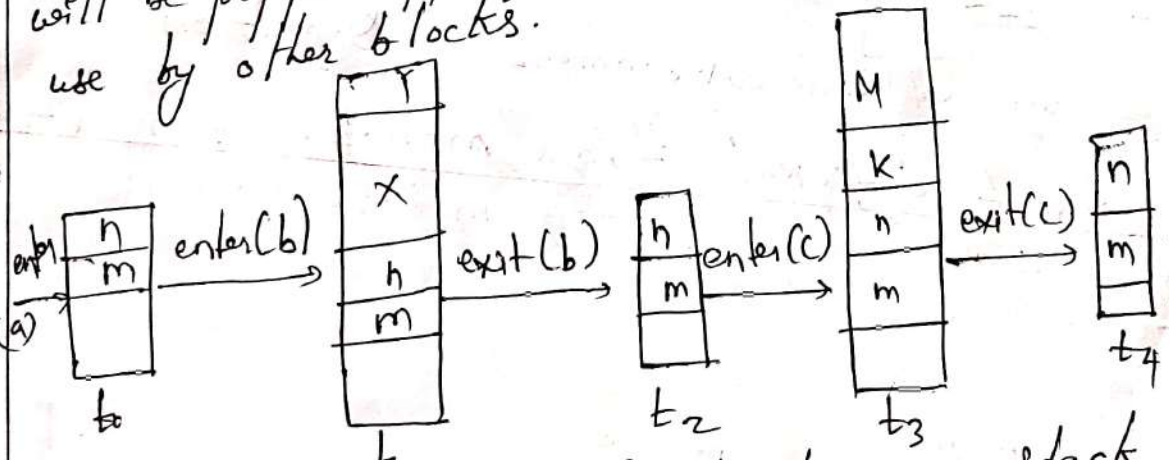
→ Example:

```
a: begin integer m, n;
   real array x(1:100); real y;
   [b: begin
       :
    end
    c: begin integer k; integer array M(0:50);
       :
    end
end
```

→ Contour diagram for given disjoint scopes.



→ Two blocks a & b are disjoint i.e. not nested.
→ Whenever the program is executing in (b), the local variables of c' namely K & M are not visible and vice-versa.
→ ALGOL prevents arrays from being corrupted and ensure the security of the system by sharing storage only between disjoint environments.
→ ALGOL Blocks obey a last in, first out discipline (STACK).
→ Whenever a block is entered, an activation record for that block is pushed onto the top of runtime stack. This activation record contains space for all variables local to that block.
→ Conversely, whenever the block is exited, its activation record will be popped off of the stack thus freeing its storage for use by other blocks.



∴ Fig: Storage Reallocation on a stack

7) Responsible Design Entails understanding user's Problems!
→ "The Responsible Design Principle"
[Do not ask users what they want, find out what they need.]

3.4. DATA STRUCTURES: ALGOL-60.

1. The Primitive are mathematical Scalars.
→ data types - integer, real and Boolean.
→ no double precision (because machine dependency)
→ No complex datatypes, i.e need to write procedure for handling complex numbers.
→ follows "Portability Principle"

2 ALGOL follows the Zero-One-Infinity Principle

Q. Algol follows zero-one-infinity Principle. Explain and verify it comparing with FORTRAN.

→ Zero-One-Infinity Principle: "The only reasonable numbers in a programming language design are zero, one, and infinity."

→ In FORTRAN, the design is filled with numbers other than zero, one or infinity.
→ Identifiers are limited to 6 characters.
→ There were at most 19 continuation cards.
→ Arrays can have at most 3 dimensions.
→ So, these 6, 19 and 3 are all numbers the programmer must remember.

→ But in ALGOL arrays are generalized. Array can be defined any number of dimension. It also allows the programmer to define upper and lower bounds. In FORTRAN. arrays are always addressed as A(1) through A(n), where n is the dimension of array. In ALGOL, array can be declared as A[m:n], it addresses as A[m] through A[n].

→ Stack Allocation permits Dynamic Arrays: FORTRAN was totally static. For example DIMENSION A[900]. If only 100 input values are to be processed, then the remaining 800 input values get wasted and memory is occupied. If more than 900 input values required, the error will be generated. In ALGOL dynamic arrays are permitted so that we can declare input values during run time.

Eg:
```
begin
integer N;
Read Int(N)
begin
  real array A[1:N]
  ...
end
end.
```

→ ALGOL has strong typing:
→ A strong type system prevents the programmer from performing meaningless operation on data, i.e, it doesnot have typing "loopholes".
– for example: A programmer cannot do an integer addition on floating point numbers or do a floating point multiply on Boolean values. In FORTRAN if didnot had such facility. FORTRAN violates the security principle while using COMMON and EQUIVALENCE
→ ALGOL doesnot support any stack or acting as though integer were a boolean or anything else

⋆⋆⋆. Please go through Data Structures section of ALGOL once and try to write answer to this question. You may add or omit some point by yourself. (Page No. 115–120).

Q. ALGOL was a major milestone in programming language? Justify.

→ It is remarkable that although ALGOL never achieved widespread use, it is one of the major milestones in programming language.
→ It added programming language vocabulary.
→ This vocabulary included type, formal & actual parameters, Block, call by value, call by name, dynamic arrays, global and local variables.
→ It was a universal tool with which to view, study and proffer solutions to almost every kind of problem in computation.

→ Eventually, all programming languages became "ALGOLike" i.e, block structured, nested, recursive and free form.

→ ALGOL has also formed the basis for several direct extensions, one of the most important of which is "Simula". which new feature called the "class" that allows grouping of related procedures & data declarations.

→ ALGOL had an direct influence on the computer architecture. (effectiveness of computer).

→ BNF in ALGOL report led to the development of mathematical theory of formal languages and to automatic means for generating programming language parsers.

→ ALGOL also was able to automate the semantic parts of compiler writing to the same extent as the syntactic parts.

→ Thus ALGOL although was a commercial failure, it was a "Scientific Triumph".

Characteristics of Second generation Programming Languages.

→ Make short note by yourself (पढ़ लो खुद से)
(See Page No. 163 of your text book).

Context-Free & Regular Grammars are the most useful.

→ Are 2 most important classes of languages and grammars.

→ A context-free grammar is any grammar that can be expressed in extended BNF, possibly including recursively defined non-terminals.

→ The context-free ~~grammar~~ language is a language that can be described by a context-free grammar.

→ A regular grammar is one that is written in extended BNF without the use of any recursive rules. A regular language is a language that can be described by regular grammar.

For example, the language defined by number, which includes the strings '-273', '$6.02_{10}23$', '3.14159', '$+3_{0}8$', is a regular language.

The main distinction is the use of recursion in these context free & regular grammar.

Recursion allows the definition of nested syntactic structures.

Consider this simplified form of ALGOL's definition of a <statement>:

<unconditional statement> ::= { <assignment statement> | for <for List> do <statement> }

<statement> ::= { <unconditional statement> | if <expression> then <unconditional statement> | if <expression> then <unconditional statement> else <statement> }

We can see that <statement> is defined recursively in terms of itself since part of an if statement is itself a <statement>. Also, <statement> is defined in terms of <unconditional statement> which is in turn defined in terms of <statement>; this is example of indirect recursion.

They allowed nesting of statements.

$x := x+1$ is an <assignment statement> and also ~~a~~ <unconditional statement>. So it can be made part of if statement like

if $x<0$ then $x := x+1$.

Since if statement is itself a <statement> it can be made part of for-loop like

[for $i := 1$ step 1 until n do if $x<0$ then $x := x+1$]

The process of embedding statements into larger statements can be continued indefinitely. //

But regular grammar doesnot permit this facilities. //

DESCRIPTIVE TOOLS: BNF.

Q. Differentiate BNF and EBNF with the help of syntactic structure of ALGOL-60.

→ * BNF is short form of "Backus Naur Form".

→ Backus developed a formal syntactic notation which was thought to be inadequate for the programming by Peter Naur. So, Naur adapted the Backus notation which was later renowned as BNF or Backus Naur Form.

→ Initally decimal number were represented by '.02', '.14159' etc and integer represented strings like '3', '+3', '-273', etc. BNF notation represents classes of strings (syntactic categories) by words or phrase in angle brackets such as <decimal fraction> and <unsigned integer>.

→ Decimal fraction is a decimal point ('.') followed by an unsigned integer. BNF represented it as

<decimal fraction> ::= <unsigned integer>

→ The ::= symbol can be read as "is defined as".

→ BNF is called a metalanguage because it is used to describe another language, the object language (ALGOL).

→ BNF describes Alternates: The <integer> has three different possible forms.

+ <unsigned integer>
- <unsigned integer>
<unsigned integer>

→ Alternation such as this is very common & BNF provides means to express it. The symbol '|' is read as "or", using it <integer> can be defined as follows:

<integer> ::= + <unsigned integer>
| - <unsigned integer>
| <unsigned integer>

→ BNF is a free form.

→ In BNF, recursion is used for repetition. The definition of number is not simple as decimal fraction is defined, most of them have alternative form.

→ Eg: Integer is
<integer> ::= + <unsigned integer>
| - <unsigned integer>
| <unsigned integer>

<unsigned integer> ::= <digit>
| <unsigned integer> <digit>.

i.e, we can use recursion as in above example. <unsigned integer> is defined in terms of itself. It is not a circular definition because the single <digit> provides a base for starting point for the recursion.

<decimal fraction> ::= . <unsigned integer>

<exponent part> ::= 10 <integer>

<decimal number> ::= <unsigned integer>
| <decimal fraction>
| <unsigned integer> <decimal fraction>

∴ BNF Definition of Numeric Denotations.

* EBNF: short form of extended Backus Naur form.
→ BNF used recursion to express one or more sequence of digits. EBNF generated a notation which directly express such sequence using syntactic category known as Kleene Cross. Eg.

<unsigned integer> ::= <digit>$^+$

The useful variant of this notation is the kleene star, * which stands for a sequence of zero or more elements of class
→ C. Eg: ALGOL identifier <name> is a <letter> followed by any number of <letter>s or <digit>s (ie sequence of zero or more <alphanumeric>s). This can be written as

<alphanumeric> ::= <letter> | <digit>

<identifier> ::= <letter> <alphanumeric>*

→ BNF permit this by allowing us to substitute '{<letter>|<digit>}' for <alphanumeric>.

→ Final EBNF <identifier> ::= <letter>{<letter>|<digit>}*

This can be read as, "an identifier is a letter followed by a sequence of zero or more letters or digits."

→ This can be made even more pictorial by stacking the alternatives.

$$\langle identifier \rangle ::= \langle letter \rangle \left\{ \begin{matrix} \langle letter \rangle \\ \langle digit \rangle \end{matrix} \right\}^*$$

→
$$\langle integer \rangle ::= + \langle unsigned\ integer \rangle \mid - \langle unsigned\ integer \rangle \mid \langle unsigned\ integer \rangle$$

Alternatively, $\langle integer \rangle ::= \left\{ \begin{matrix} + \\ - \end{matrix} \right\} \langle unsigned\ integer \rangle \mid \langle unsigned\ integer \rangle$ is preferred when we want to

→ But the use of square brackets express optional preceded by a '+' or '-'.

$$\langle integer \rangle ::= \left[\begin{matrix} + \\ - \end{matrix} \right] \langle unsigned\ integer \rangle$$

$$\langle number \rangle ::= \left[\begin{matrix} + \\ - \end{matrix} \right] \left\{ \begin{matrix} \langle decimal\ number \rangle \\ [\langle decimal\ number \rangle]_{10}\ \langle integer \rangle \end{matrix} \right\}$$

$$\langle unsigned\ integer \rangle ::= \langle digit \rangle^+$$

∴ EBNF Definition of Numeric Denotations.

4. LIST PROCESSING AND FUNCTIONAL PROGRAMMING: LISP

4.1. History and Motivation: The fifth generation programming languages which comprises 3 overlapping programming paradigms - functional programming, object oriented programming and logic programming. LISP is the functional programming.

→ LISP developed in Late 1950's.

→ Included concept of artificial intelligence programming.

→ Main idea developed was of the Linked representation of List structures & use of a stack to implement recursion.

→ Gerlenter and Gerberich of IBM developed FLPL, the FORTRAN List-Processing Language by writing a set of list processing subprograms for use with FORTRAN programs.

→ McCarthy developed central idea of LISP. He came up with the concept of conditional expression: Later in 1958 he began using recursion in conjunction with conditional expressions in his definition of List-processing functions.

→ Later List handling routines were developed. McCarthy became convinced that recursive list processing functions with conditional expressions formed an easier to understand basis for the theory of computation. He defined a universal LISP function that could interpret any other LISP function.

→ Once the representation was designed & the universal function was written, one of the members realized that the group had, in effect, an interpreter. Therefore, he translated the universal function into assembly language & linked it with the list-handling subroutines.

→ LISP is widely used in Artificial intelligence and symbolic applications.

→ LISP was standardized after a period of divergent evolution.

9.4.2. DESIGN: STRUCTURAL ORGANISATION.

3. What is LISP? Define the structural organisation of LISP program with example.

→ The Name LISP derives from "LIST Processor." LISP is the second oldest high level programming language still in use. It is also the programming which became favoured programming language for artificial intelligence. (AI). All program code is written as S-expressions or parenthesized lists. For example, a function f that takes 3 arguments would be called as (f arg1 arg2 arg3).

→ Example of LISP Program.

```
(defun make-table (text table)
    (if (null (text)
        table
        (make-table (cdr text)
            (update-entry table (car text)))))

(defun update-entry (table word)
    (cond ((null table) (list (list word 1)))
        ((eq word (caar table))
        (cons (list word (add1 (cadar table)))
            (cdr table)))
        (t (cons (car table)
            (update-entry (cdr table) word)))))

(defun lookup (table word)
    (cond ((null table) 0)
        ((eq word (caar table)) (cadar table))
        (t (lookup (cdr table) word))))

(set 'text' (to be or not to be))
(set 'freq (make-table text nil)).
```

*[Do not expect to be able to read this program unless you have previous experience. It is just general appearance and a example of LISP program.]

→ Function application is the central idea. Programming languages are often divided into 2 class.
1) Imperative - which include all languages.
2) Applicative - central idea is functional application.
→ Function application in LISP is written as
(f a1 a2 .. an) where f = function & a1, a2 .. an = arguments.
→ List is the primary data structure constructor provided by LISP. No need to learn about others.
→ This notation is called Cambridge Polish because it is a particular variety of Polish notation developed in MIT (in Cambridge, MA). Polish notation is named after the Polish logician Jan Lukasiewicz.
→ The distinctive characteristics of polish notation is that it writes an operator before its operands. This is also sometimes called prefix notation because the operation is written before the operands. For example, to compute 2+3 we would type
(plus 2 3) → to an interactive LISP system, it would respond 5.
→ LISP notation is little more flexible than the usual infix notation since one plus can sum more than 2 numbers. We can write (plus 10 8 5 64) for the sum 10+8+5+64.
→ LISP is fully parenthesized.
→ The List is the primary data structure constructor and programs are represented as lists.
→ Eg: (to be or not to be) - This is composed of 4 different atoms. to be or not.
The list is the only data structure provided by LISP. This is example of Simplicity Principle.
→ LISP also allows 2 sublists. (lists to be constructed from other lists). Eg
→ ((to be or not to be) (that is the question)) → has 2 sublists.
first second.

Q: What is LISP? Explain about car and cdr operations with example. (searching techniques in LISP)

→ (We have already written about what is LISP. So lets move to second part of question).

- Under the (Design: Data Structure portion of the LISP, we came to the two term name as "car" and "cdr"

— Both the car and cdr are the parts of Lists. Operation that builds a structure is called constructor (cons) and those that extract their parts are called selector (car, cdr).

→ "Car" function selects the first element of the list. Eg:

(car '(to be or not to be)) returns the atom 'to'.

The first element of the list can be either an atom or a list and car returns it. For example, since 'freq' is the list.

((to 2) (be 2) (or 1) (not 1))

the application (car freq) returns the list (to 2)

→ 'cdr' function returns all of a list except its first element. So, (cdr '(to be or not to be) returns the list (be or not to be). Similarly, (cdr freq) returns.

((be 2 (or 1) (not 1)).

Remember both car and cdr requires nonnull list for the argument.

→ cdr always returns a list.

→ car and cdr are both pure functions, i.e, they do not modify their argument list.

— Both make new copy of the list.

→

P.T.O. (47)

→ Car and cdr can be used in combination to access the components of a list. Suppose DS is a list representing a personnel record for Don Smith:

(Set 'DS' ((Don Smith) 45 30000 (August 25 1980)))

→ The list DS contains Don Smith's name, age, salary, hire date.

→ To extract the first component of this list, his name, we can write (car DS) which returns (DON smith).

→ Now how to select his age?

→ Notice that cdr operation deletes the first element of the list, so that second element of the original list is the first element of the result of cdr. That is, (cdr DS) returns, (45, 30000 (August 25 1980)) so that (car (cdr DS)) is 45.

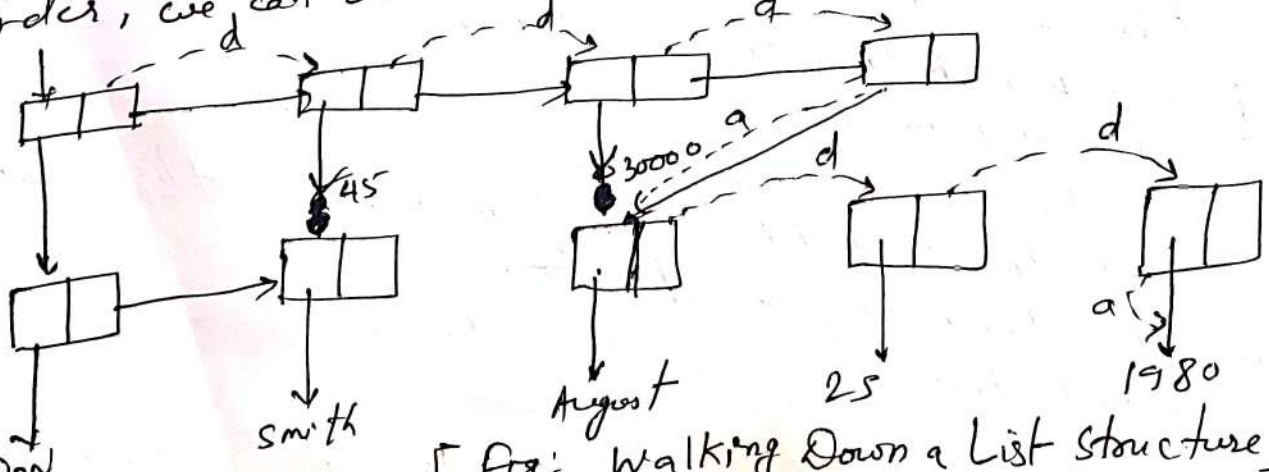
For DON smith salary, (car (cdr (cdr DS)))

→ For hire Date, (car (cdr (cdr (cdr DS))))

→ This can be seen clearly if the list is written in Lisked data structure then 'd' moves to the right and 'a' moves down. The composition of cars and cdrs is represented by the sequence of 'a's & 'd's between the initial 'c' and the final 'r'. For example, (car (cdr (cdr (cdr DS)))) can be written shortly as (caddr DS).

initial final

→ By reading the sequence of 'a's and 'd's in reverse order, we can use them to 'walk' through data structure.



[Fig: Walking Down a List Structure] 48

Q: What is property list?

→ Information can be represented by property list.
→ The personnel record would probably not be represented in LISP in the way we have discussed in car and cdr.
→ Each property of Don Smith is assigned a specific location in the list, so it becomes difficult to change the properties associated with a person.
→ Better arrangement is to precede each piece of information with an indicator identifying the property. For example,

(name (Don Smith) age 45 salary 30000 hire-date (August 25 1980)).

→ This method of representing information is called a property list or p-list. Its general form is

$(p_1\ v_1\ p_2\ v_2 \cdots p_n\ v_n)$

where p_i = indicator of property
v_i = corresponding property value.

→ Advantage: ① Flexibility - As long as property is accessed by their indicators, program will be independent of the particular arrangement of the data, i.e. same information can be written as

(age 45 salary 30000 name (Don Smith) hire-date (Aug, 25, 1980)).

→ We can use LISP's 'if' for conditional expression.
→ The 'getprop' function is

```
(defun getprop (p x)
  (if (eq (car x) p)
      (cadr x)
      (getprop p (cddr x))))
```

(49)

* Please go through property list explanation in your book once. (Page No: 324 & 325).

(Association Lists)

Q. Write assoc. function in LISP to access the value of a list. How would you handle the case where the requested attribute is not associated by a list?

→ Sometimes it will be inconvenient to use property List, because some properties are flags that have no associated value.

→ It will be necessary to associate some value with the indicator even though it has no meaning (property indicator and value must alternate in property list).

→ An analogous problem arises if a property has several associated values. It will be necessary to group the names together into a subsidiary list i.e. association list or a-list.

→ An a-list is a list of pairs with each pair associating two pieces of information. The a-list representation of the properties of Don Smith is

```
((name (Don Smith))
 (age 45)
 (salary 30000)
 (hire-date (August 25, 1980)))
```

- The general form of an 'a-list' is a list of attribute value pairs:

((a1 v1) (a2 v2) ... (an vn))

→ The function that does the forward association is normally called assoc. For example:

```
(set 'DS '(name (Don Smith) (age 45) ....))
((name (Don Smith)) (age 45) ...)
(assoc 'hire-date DS)
(August 25 1980)
(assoc 'salary DS)
30000
```

→ For the second part of question....
→ we came to know about car and cdr. car selects first element cdr removes first element of a list.

→ LISP's only constructor, 'cons' adds a new element to the beginning of a list. for example
(cons 'to '(be or not to be)) returns the list (to be or not to be).

→ cons is the inverse of car and cdr.
(car '(to be or not to be) = to
(cdr '(to be or not to be) = (be or not to be)
(cons 'to '(be or not to be) = (to be or not to be).

→ Lists are also constructed Recursively: When we want to concatenate 2 lists, that (a b c d) is required result from (a b) & (c d). How can this be accomplished? Here the concept of recursion to construct lists is implemented. Our goal is to develop function 'append' such that
(append '(a b) '(c d) = (a b c d).
In general (append L M) will return the concatenation of the lists L & M.

★ More examples & explanation (if you want), see book, page no. <327-329>

★ Mapcar function: mapcar is a function that calls its first argument with each element of its second argument in turn. The second argument must be a sequence.
. mapcar function that applies a given function to each element of a list and returns a list of the results.
. The 'map' part of the name comes from the mathematical phrase "mapping over a domain" meaning to apply a function to each of element in the list.

Eg: (mapcar '1+ '(2, 4, 6)
⇒ (3, 5, 7)

Eg: (mapcar 'add1 '(1, 9, 8, 4))
(2 10 9 5)

Reduce function:

→ The reduce function combines all the elements of sequence using a binary operation. for eg: using '+' can add up all the elements.

(reduce '+' (1, 2, 3, 4)) ⇒ (10) //.

★ mapcar function allow abstraction in functional arguments.

★ reduce function allow programs to be combined in functional arguments.

(functional argument is part of control structure of LISP - under functional programming).

★ **The Conditional Expression:** The logical connectives are evaluated conditionally.

eg: $sg(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{if } x < 0 \end{cases}$

```
(defun sg (x)
  (cond ((plusp x) 1)
        ((zerop x) 0)
        ((minusp x) -1)))
```

★ **Logical connectives:** The function (or x y) has the value t (true) if either or both of x & y have the value 't', in any other case or has the value nil (false). Another way to say this is if x has the value t, than or has the value t, otherwise the 'or' has the same value as y:

```
(or x y) = (if x t y)
(not x) = (if x nil t)
(and x y)
```

Here operands are evaluated sequentially.

Interpretation of the logical connectives is known as the conditional or sequential interpretation, which evaluates both the arguments.

LISP allows both 'and' and 'or' to have more than 2 arguments.

(and p_1 p_2 ... p_n) - any false आएगी तब false

(or p_1 p_2 ... p_n) → true तभी सभी evaluate होंगे।

Iteration is done by Recursion:

- LISP does not require conventional programming control structures. All forms of iteration are performed by recursion. In previous example of append and getprop functions, we saw

```
(defun getprop (p x)
  (if (eq (car x) p)
      (cadr x)
      (getprop p (cddr x)) ))
```

- This is analogous to while loop in a Pascal language. The function continues to call itself recursively until a termination condition is satisfied.
- In LISP, it is more convenient to write a recursive procedure that performs some operation on every element of a list. We can write a recursive function 'plus_red' that does the 'plus reduction' of a list.

```
(defun plus-red (a)
  (if (null a)
      0
      (plus (car a) (plus-red (cdr a)) )))
```

- Example of 'plus-red' are

```
(plus-red '(1 2 3 4 5))
  15
(plus-red '(3 3 3 3))
  12.
```

Hierarchical structures are processed Recursively:

- As an example we will design the 'equal' function, which determines whether two arbitary values are the same. It is different from 'eq' primitive (works only on atom) since 'equal' function tells us if two arbitrary list structure are same or not.

Example: (equal '1 2 3) '(3 2 1))

```
nil
(equal '1 2 3) '(1 2 3))
  t
```

- 'equal' function is applicable to any 2 arguments.
- 'equal' function can be designed as the same as other recursive functions and it helps is reducing the complicated case to easy cases.
- Atoms can be handled immediately since they can be compared with the 'eq' function.
- If both x and y are atoms, then (equal x y) reduces to (eq x y).
- If either x or y is an atom & the other is not, then we know they can not be equal. This can be summarized as follows:

if x and y are both atoms, then
(equal x y) = (eq x y),
if x is an atom & y isn't
or y is an atom & x isn't
then (equal x y) = nil.

- This two can be combined using LISP's sequential 'and':
(and (atom x) (atom y) (eq x y)).
- If neither x nor y is an atom, we have to compare (car x) and (car y) and if they are equal, we have to call function equal recursively.
(equal (cdr x) (cdr y)).
- Here we will call 'equal' recursively of cars and x and y. This solves the problem when x & y are both lists.

if x and y are both lists then,
(and (equal (car x) (car y))
(equal (cdr x) (cdr y))

- Moreover, we can combine all of above discussed function for two mutually disjoint cases.

```
(defun equal (x y)
  (or (and (atom x) (atom y) (eq x y))
      (and (not (atom x)) (not (atom y))
           (and (equal (car x) (car y))
                (equal (cdr x) (cdr y))))))
```

Lambda Expressions are Anonymous Functions:

- LISP allows us to write anonymous functions that are evaluated only when they are encountered in the program. These functions are called Lambda functions.

→ Syntax for Lambdas' expressions:

```
(lambda (parameters) body)
(write ((lambda (a b c x)
   (+ (* a (** x) (* b x) c))
      4 2 9 3)
   )
```

→ We have already discussed mapcar function. Eg:
(mapcar '(cons val x) L).

→ The trouble with this is that if is ambiguous, we don't know which names are the parameters to the 'cons' & which are globals. In such case we use Lambda's expressions.

→ Lambda function is mathematical theory called as the 'lambda calculus' provides a solution to above problems.

- It provides the notation for anonymous functions, that is, for functions that have not been bound to names. LISP uses the notation.

(Lambda (x) (cons val x)).

→ Lambda expressions are ~~valuable~~ valuable and can be manipulated like any other LISP Lists; ie, can be passed as parameters.

Now, (mapcar '(cons val x) L) can be solve by writing Lambda expression as.

(mapcar '(Lambda (x) (cons val x)) L).

→ To double up all the elements of L, we write,

(mapcar ((lambda (n) (times n 2)) L).

* Please go through Functional Programming of LISP & Design of control structures where you'll see all the terms we discussed above (like conditional expression, lambda, logical, iteration & hierarchical structures).

User defined function in LISP (short note) Imp..

→ In LISP we use assoc function.

```
(defun my-assoc (A L)
    (cond
        ((null L) nil)
        ((eq A (car (car L))) (cadr L))
        (t (my-assoc A (cdr L)))))
```

Where A is the property list and L is the Association list.

→ If list is null or empty it returns nil, and if property is equivalent to first property of the list then choose first associative value otherwise it goes for other sublist to check whether property is met or not.

---

3. DESIGN : NAME STRUCTURES.

① The primitive bind Atoms to their values:

→ In LISP bindings are established in two ways - through property lists & through actual-formal correspondence. The formal is established as by pseudo functions such as 'set' & 'defun'. Example

(set 'text' (to be or not to be)) binds text to the list (to be or not to be) by placing 'apval' property with the value (to be or not to be) on property list of text. Similarly a 'defun' binds a name to a function by placing the expr property on an atom's property list.

→ Binding established through property list are global.

② Application binds formal to Actuals: The other primitive

binding operation is actual-formal correspondence. Eg:

```
(defun getprop (p x)
    (if (eq (car x) p)
        (cadr x)
        (getprop p (cddr))))
```

and then evaluate the application

```
(getprop 'name DS)
```

- the formal p will bound to the atom 'name' & formal x will be bound to value of actual DS.

③ Temporary Bindings are a Simple Syntactic Extension:

→ A number of LISP dialects (including common LISP) provided method of introducing local names analogous to our use. The let function allows a number of local names to be bound to values at one time, the second argument to 'let' is evaluated in resulting environment.

Eg: (let ((n1 e1) ... (nm en)) E)

evaluates each of expressions ei & binds name ni to the corresponding value. The expression E is evaluated in the resulting environment & is returned as the value of 'let'. example:

```
(defun roots (a b c)
  (let ((d (sqrt (difference (expt b 2)
                             (times 4 a c)))))
    (list (quotient (plus (minus b) d)
                    (times 2 a))
          (quotient (difference (minus b) d)
                    (times 2 a)))))
```

- 'let' function is like Algol block which introduces a new scope & declares a set of names in that scope, each with an initial value.

④ Dynamic Scoping & the Constructor:

→ The function is called in the environment of caller, In LISP, 'let' is equivalent to a function application, so 'let' must be invoked in the environment of the caller.

③ Dynamic Scoping complicates functional Arguments:

→ The collision of variables takes place while using dynamic scooping. It violates information hiding principle. The problem is known as functional argument problem. So special form called 'function' was defined that binds a lambda expression to its environment of definition.

Eg: (twice (function (lambda (x) (times x a b x))) 3)

4.6: DESIGN: SYNTACTIC STRUCTURE:

1. Much of LISP's syntax has evolved:

→ Over the years, there have been several improvements in LISP's syntax. The initial use of 'define' function was replaced by 'defun' function.

Eg: (define ((n1 e1) ... (nk ek))) would define each of ni to be ei: There were lot of parenthesis used in 'define' function which some people claim that LISP stands for "Lots of Idiotic Single Parenthesis!"

→ So the improved version of 'define' function was 'defun' function.

```
(defun (getprop p x)
       (if .... ))
```

→ Another example of evolution of LISP was the use of 'set' function.

```
((set (quote val)) 2)   older way
        ↓
(setq val 2)            newer way
```

② List Representation facilitates program manipulation.

→ The list representation makes it easier to get the important parts of LISP programs. Eg: If L is lambda expression then (cadr L) is its list of bound variables & (caddr L) is its body:

```
(set L '(Lambda (x y) (f x y)))
        (Lambda (x y) (f y x))
(cadr L)
  (x y)
(caddr L
  (f y x)
```

→ It balances the reduction of readability.

4.7: RECURSIVE LIST-PROCESSOR: LISP.

Implementation of List Recursive Processor

1. **A LISP Interpreter can be written in LISP:** The recursive interpreter is written in LISP, although it could be written in any language with recursive procedures & the ability to implement linked lists. Since it is written in LISP, it makes the use of facilities of LISP, such as the list-processing operations like car, cdr and cons operations.
   - The LISP universal, is called 'eval' since it evaluates a LISP expression and are also called manipulation operators for doing manipulation operations. Form of eval

     (eval 'E A) = V

     where V is the value of E in that context.
     A is a list of representing a context
     E is any LISP expression (S-expression)

2. **The interpreter is arranged by Cases.**
   - The 'eval' function will break its input down into cases as shown is as below:

Type	Example
Numeric atom	2
Non-numeric atom	val.
Quotation	(quote (B C D))
Conditional	(if (null x) nil 1)
Primitive	(cons x y)
User-defined	(make-table text nil)

   - The LISP mechanism for handling case is 'if', so eval will take the form of a larger 'if':

```
(defun eval (e a)
   ( if
        (atom e)
        Handle atoms
        Handle lists))
```

3. **The value of a numeric atom is that atom:** The value of 2 is the atom 2. so, (eval 2 a) = 2. and the case for handling numeric atoms is

```
((and (atom e) (numberp e)) e)
```

4. Non numeric atoms must be looked up in the environment:

→ The value to which an atom is bound is determined by the environment in which the atom is evaluated.

→ (eval E A) → evaluation of expression of E in environment represented by list A.

→ For example, if the environment A binds 'val' to 2 & 'text' to (to be or not to be), then

```
(eval 'val A)
  2
(eval 'text A)
  (to be or not to be)
```

→ Environment can be represented by association list. The association list which bind 'val' and 'text' as specified above can be written as

((val 2) (text (to be or not to be)))

→ The evaluation of val is this environment is

(eval 'val '((val 2) (text (to be or not to be))))

⇒ 2

(5) Conditional Delays evaluation of its Arguments:

→ Consider a conditional. (if P T F) → Its evaluation is that; evaluate P. If it is ~~true~~ (t), then evaluate T, if it is nil, then evaluate F. The evaluations are accomplished by a recursive call of eval, which ensures that any legal LISP expression can be used as the condition, consequent or alternate of an 'if'.

→ We can write the checking process in LISP as

```
(if (eval (cadr e) a)
    (eval (caddr e) a)
    (eval (cadddr e) a))
```

where (cadr e) is P.
(caddr e) is T
(cadddr e) is F

→ We are using 'if' to interpret 'if'.

(6) Arguments are recursively evaluated:

→ Arguments are recursively evaluated by a function called 'evargs'.

→ If e is the application (f x_1 x_2 -- x_n) then (car e) is f, the function to be applied & (cdr e) is the list of arguments. Therefore, (evargs (cdr e) a) will be the list of arguments values.

7) Primitive operations are performed directly:
- Since there are number of primitive operations, the natural structure for the apply function is a cond that handles the different primitive operations. That is,

```
(defun apply (f x a)
  (cond
    ((eq f 'plus)  Handle a plus)
    ((eq f 'car)   Handle a car)
    ((eq f 'cdr)   Handle a cdr)
    ((eq f 'cons)  Handle a cons)
    .....))
```

8) User defined function applications require four steps:
→ ① Evaluate actual parameters.
② Bind the formal parameters to the actual parameters.
③ Add these new bindings to the environment of the evaluation
④ Evaluate the body of the function in this environment.

4.8. STORAGE RECLAMATIONS:

① Explicit Erasure Complicates Programming

→ Most of the older programming uses explicit erasure for returning the program to the free storage area after its completion of the task. But that is very bad idea and also it violates the security principle.
→ Programmer needs to work harder and also keep track of the each cell & all of its lists.
→ Automatic erasure is advantageous.
→ It also violates the security of programming system. Suppose a cell is returned to free storage but is still referenced by several other list. These other lists will now have dangling pointers; that is, pointer do not point to allocated cell. Program error arises.
→ Garbage collection represents an application of Responsible design principle since it provides better solution.

② References counts identify inaccessible Lists:

→ We have seen explist erasure is low level & error prone. So automatic erasure system is required since it keep track of the accessibility of each cell. This observation leads to a technique of storage reclamation called reference counts. Reference counts must be maintained correctly. We can include reference count field to each cell.

③ Cyclic Structures are not reclaimed

→ Non of the field will have free storage when there is the cylic structure.

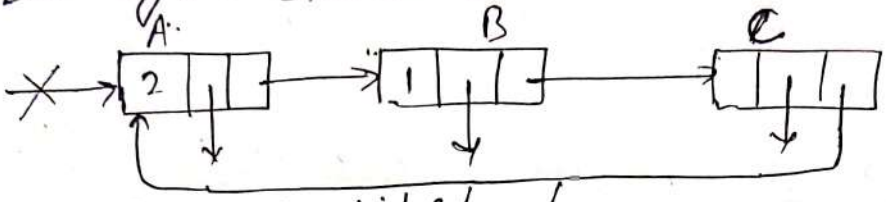


Fig: Cyclic List structure

If the pointer is the A is destroyed, making the structure inaccessible, this will cause A's reference count to be decreme nted by to 1. Since neither A, B & C has a zero reference count, no cells be returned to free storage even though none of this cells is accesible.

④ GARBAGE COLLECTION RECLAIMS CYLIC STRUCTURES

There is an alternative approach to automatic storage reclamation that can handle cyclic structure and it is called garbage collection.

When the supply of free cells is exhausted, the system enters a garbage collection phase in which it identifies unused cells & returns them to free storage. It deals only after crisis arises.

Here we deal with simplest kind of mark-sweep garbage collector which operates in 2 phases.

In markphase, the garbage collector identifies all of those cells that are accesible i.e. they are not garbage.

In sweep phase, the garbage collector places all of the inaccessible cells in the free storage area, often by placing them on the free list.

The algorithm for mark phase

```
mark phase:
    for each root R, mark (R)
Mark (R):
    if R is not marked then:
        set mark bit of R;
        mark (R↑. left);
        mark (R↑. right);
    endif.
```

- In mark phase, the garbage collector starts at the roots and follows pointers, marking each cell it reaches as accessible.
- The second phase sweep phase of garbage collection is the sweep, when all of the inaccessible cells are returned to free storage. We can visit each cell in order (that is, sweeping through memory). If a cell is unmarked, then it is inaccessible & it can be linked onto the free list.

```
Sweep phase:
    for each cell c:
        if c is marked then reset c's mark bit.
        else link c onto the free-list.
```

(5) Garbage Collection Results in Non Uniform Response Time.

- The program runs smoothly until it runs out of storage then it grinds to a halt for garbage collection.
- If the garbage collection happens at wrong time, then the system lead to disester.
- If there are many cells to be reclaimed for the garbage collection, the system may be interrupted for several seconds.

Look at page No: 391 & 392 for figure for Mark phase garbage collector.

Characteristics of Function Oriented Programming Languages:

- LISP illustrates many of the characteristics of applicative (function-oriented) programming.
- We have seen there is an emphasis on the use of pure functions & minimal use of assignment operations which leads to the use of recursion as a method of iteration and the Polish notation as the basic syntactic structure.
- Use of lists is supported by dynamic and automatic storage management. Applicative languages usually have list as principal data structure.
- The basic control structures of applicative programming are the conditional expression & recursion.
- They have many desirable properties such as
  1. abstraction - suppresses many of the details
  2. Can be evaluated in many different orders which allows for parallel programming in computers.
  3. The absence of assignment operations makes function oriented programs much more amenable to mathematical proof and analysis than are imperative programs.
  4. Since functional programming takes place in the timeless realm of mathematics, it avoids the difficulties of temporal reasoning.

5. OBJECT ORIENTED PROGRAMMIN - SMALLTALK.

5.1. History and Motivation;

- Smalltalk was developed in the learning research group (LRG) at Xerox's Palo Alto Research Center in early 70's. The group was led by Alan Kay who worked on the vision he called as "Dynabook" the computer could be used creatively like a dynabook even by children. Their exploration led them to develop not only the personal vision of notebook computing (the Dynabook) but also Smalltalk object oriented programming language.
- The first Smalltalk programming system called Smalltalk-72 was designed and it supported Alan kay's new programming paradigm called object oriented programming.
- Before going to this new paradigm, Kay investigated simulation and graphics oriented languages which could make computers accessible to non specialists. He had helped design called FLEX which had idea about classes and objects.

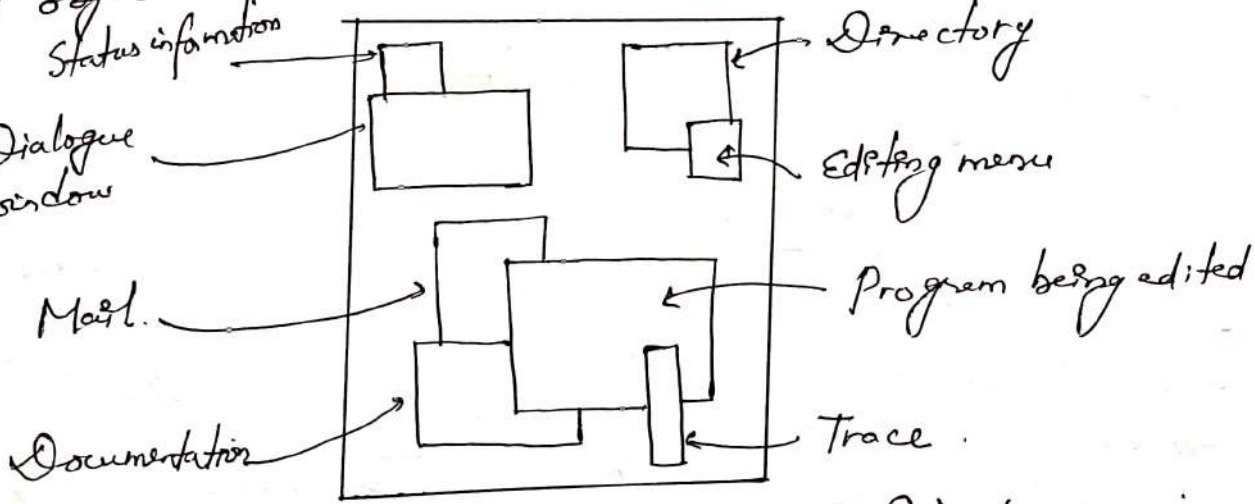


Fig: Example of a Dynabook Display.

5.2. STRUCTURE ORGANISATION:

(1.) Smalltalk is interactive & interpreted:

There are 2 primary way to define things in Smalltalk. The first binds a name to an object. For example:

x ← 3 → binds 'x' to object 3 [Similar to LISP]

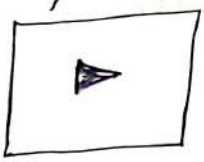
y ← x+1 → y to the object 4.

- The other way is by class definition.
- We can write user defined expression as $x * 2 \Rightarrow 6$.

(2) Objects reacts to message.

→ Since the idea of Smalltalk is borrowed from LOGO & it is also graphics oriented language, it uses "turtle graphics" of LOGO, that is based on object (called turtles) that draw as they move around the screen. Smalltalk provides a similar class of objects called "pens"

→ Eg: We will investigate the behaviour of a pen named Scribe and see how it can draw on the screen. The display shows us the position of Scribe.

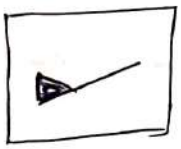


→ Let us assume scribe is at position (500, 500) center of screen which is written as 500 @ 500.

→ To draw a line from 500 @ 500 to (200, 400), we have to write/enter

Scribe goto: 200 @ 400

(this message tells us to ~~scribe~~ draw line to (200, 400).

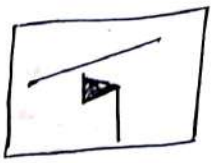


→ If we want to move Scribe without drawing a line, we use "Scribe penup" (which tells us to stop drawing)

→ For example, to draw a vertical line from (500, 500) to (500, 400) we can enter

Scribe penup.
scribe goto: 500 @ 100
scribe pendn.
Scribe goto: 500 @ 400.

The result is



Summary

1) The objects have behaviour.
2) Objects can be made to do things by sending them message.
3) Repetitive operation can be simplified by using control structure.

③ Objects can be instantiated:

→ The process of creating objects is called instantiation. In our example, we have created scribe objects from pen class. If we want to instantiate another object of pen class we will write the command as

anotherScribe ← pen newAt: 200@800.

→ This sends the message newAt: 200@800 to the class pen, which creates & returns a new pen located at coordinates (200, 800) & pointing to right. The name anotherScribe is bound to this new object, hence we can direct messages to new object by using this name.

anotherScribe penDn.
5 timesRepeat: [anotherScribe go: 50; turn: 72]

→ Eg: To create equilateral triangle.

Writer ← pen newAt: 800@800
Writer penDn.
3 timesRepeat [Write go: 50; turn: 60]

④ Classes can be Defined:

→ Smalltalk allows us to define a class box that can be instantiated any number of times. (Supports Abstraction & Regularity Principle).

class name	box
instance variable name	loc tilt size scribe
instance messages and methods	

```
shapeIl
  scribe penUp; goto: loc; turnTo: tilt; penDn.
  4 timesRepeat: [scribe go: size; turn: 90]

showIl
  scribe color ink.
  self shape

eraseIl
  scribe color background.
  self shape

grow: amountIl
  self erase.
  size ← size + amount
  self show
```

Fig: Definition of class box.

- First block represent name of class.
- Second block represent/contains instance variables. These are local to each instance of class (i.e, to each box) & instantiated for each box. i.e, each box has its own loc, tilt, size & scribe.

loc = location of upper left corner
tilt = angle
size = length of box side
scribe: It is the pen used by box to draw its shape.

→ Suppose B1 is a box, then sending it the message shape
B1 shape causes it to
i) lift its pen
ii) go to the specified location
iii) turn to the specified angle
iv) draw a box of the specified size.

5) Classes can also Respond to Messages:

B2 ← box newAt: 300 @ 200

. the message newAt: 300 @ 200 will be sent to class Box. This class method begins by sending itself (i.e, the class Box) the message new. new is common for all classes.

5.3 DESIGN : CLASSES AND SUBCLASSES:

• Smalltalk objects model Real-World Objects.

→ This allows many programs to be viewed as a model or simulation of some aspects of the real world. The dominant ~~paradigm~~ paradigm (or model) of programming is simulation.

• Class group Related Objects.

→ In real world, all of the objects that we observe are individuals; every object differs from every other in a number of ways. It follows abstraction principle.

• Subclasses permit Hierarchical classification.

→ Class will have the subclasses. (Concept of inheritance can be explained here). Methods defined in the superclass are also inherited.

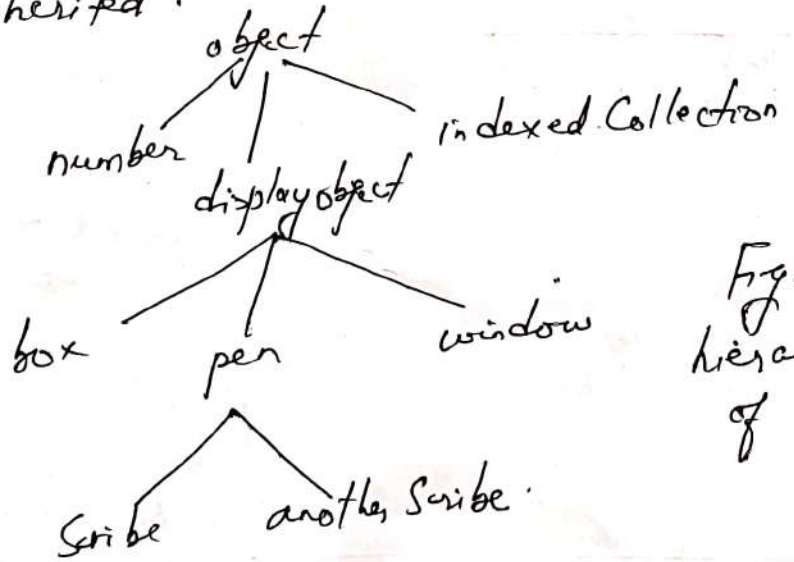


Fig: Example of hierarchical classification of objects.

- Behaviour can be extended or modified.

→ Subclasses can build upon the behaviour of their superclasses, they can also modify it. The definition of a method in a subclass overrides any definitions of that method that may exist in its superclasses.

- Overloading is implicit and inexpensive.

→ Explain concept of overloading.

- Class allows multiple Representation of Data types.
- Objects can be self displaying
- Methods accept any object with the proper protocol.
- Operators are handled asymmetrically.
- Hierarchical subclasses preclude orthogonal classifications.
- Multiple inheritance raises difficult problems.

Look at your book for more explanations
page: 411 – 421.

Q. What are the different form of message template in SMALLTALK? Explain them.

Sol. There are 3 forms of message template.

(1) Keywords followed by colons

(2) One parameters message

(3) Multiple parameter (one or more parameter) message.

(1) Keywords: In smalltalk parameters are seperated by keywords. Example: newBox setLOC: initialLocation tilt: 0 size: 100 scribe: pen new

is equivalent to ADA procedure call

NewBox.Set (Initial_Location, 0, 100, Pen.New);

we can use colons if there are more parameters to the method, but what if it has only one parameter. we can use parameterless message as

131 show

→ One parameter message: To avoid the unusual notation, Smalltalk has made a special exception. The arithmetic operators (and other symbols) can be followed by exactly one parameter even though there is no colon. For example. In x + 2 * y, the object named x is sent the message +2 & the object resulting from this is sent the message *y.

→ Multiple parameter (one or more parameter) message -
- It uses keywords with colons for messages.

```
(
    (scribe grow : 100)
)
```

- This format convention fits the zero-one-infinity principle since the only special cases are zero parameters and one parameters.

- In summary, there are 3 formats for messages:
  1. Keywords for parameterless messages (eg: 8t show)
  2. Operators for one parameter message (eg: x + y)
  3. Keywords with colons for messages with one or more parameters (eg: Scribe grow : 100)

5.5. IMPLEMENTATION : CLASSES AND OBJECTS

Q. Explain an object and class specification in SMALLTALK. How does class & object are represented diagrammatically in SMALLTALK?

→ Object Representation ⇒ Smalltalk implementation can be derived by application of the abstraction and information hiding principles.

→ The representation of an object must contain first that information that varies from object to object; information that is the same over a class of objects is stored in the representation of that class; which is known as instance variables. We will not be able to access the information stored with the class of an object unless we know what class of that object is. (10)

→ The representation of the object must contain some indication of the class to which the object belongs.

+ Among the various ways of implementing objects, here we deal with representation of the object a pointer to the data structure representing the class.

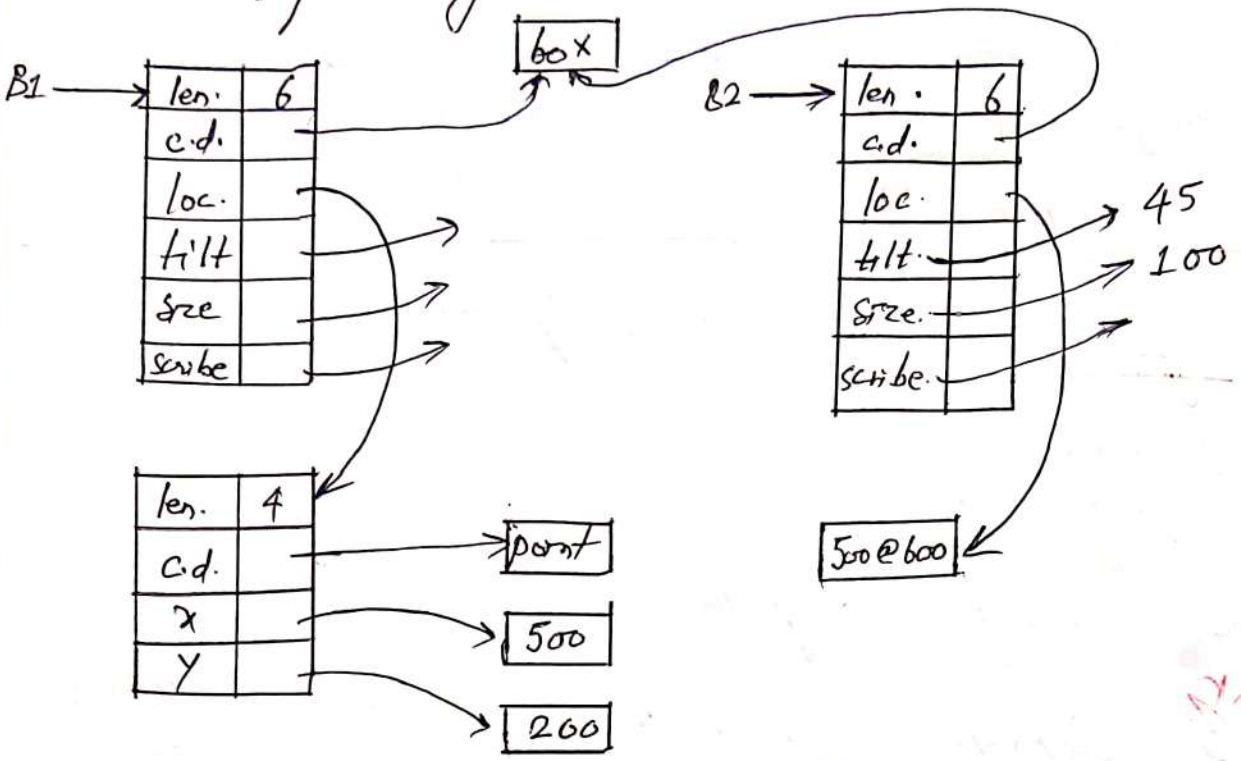


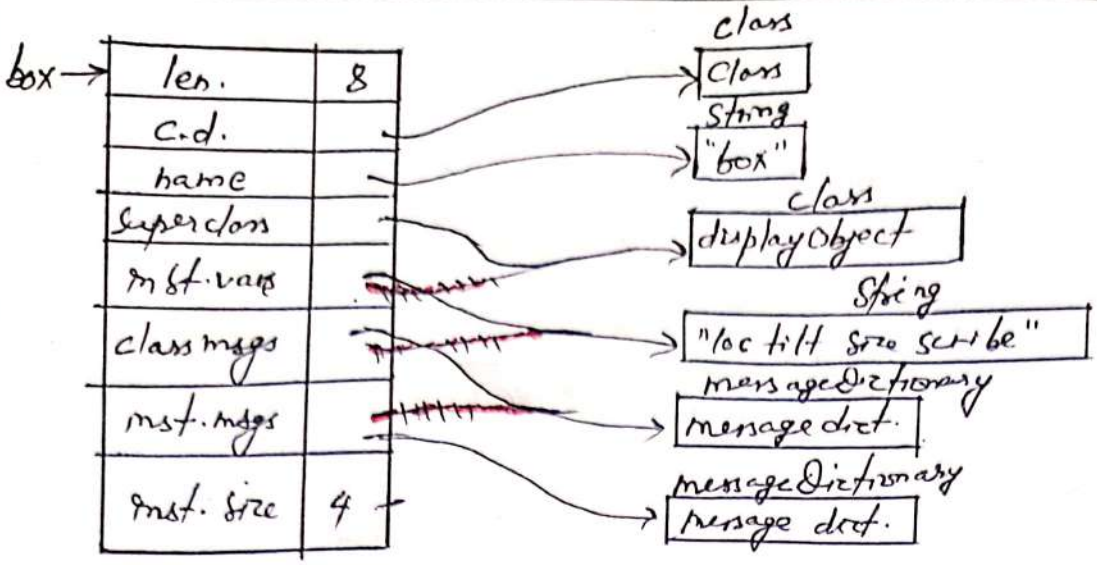
Fig: Representation of objects.

→ Notice that in addition to the instance variables and class description (c.d), each object has a length field (len). This is required by storage manager for following tasks such as allocating, deallocating & moving objects in storage.

→ Class Representation: The classes are represented like the objects just discussed above, with length and class description field (the latter pointing to the class called Class). The instance variable of a class object contain pointers to objects representing the information that is the same for all instances of the class.

→ Information includes following

i) The class name
ii) The superclass
iii) The instance variable name
iv) The class message dictionary
v) The instance message dictionary.



** यहाँ Red arrow यानि Mistake हो सकता है. Don't give importance to it.
• Page no: 431.

Fig: Representation of class Object.

→ • Field inst. size (instance size), which indicates the number of instance variables.

→ Message dictionary: Message identified by the keywords that appear in message invoking the method.

Q. How is activation record representation in SMALLTALK?

• Activation Record Representation: The process of procedure call and return can be understood as the manipulation of activation records. We will use activation records to hold all of the information relevant to one activation of a method.

• It has 3 main parts

i) Environment Part: The context to be used for execution of the method.

ii) Instruction Part: The instruction to be executed when this method is resumed.

iii) Sender Part: The activation record of the method that sent the message invoking this method.

Consider in reverse order:

Sender Part: It is a dynamic link. i.e, a pointer from the receiver's activation record back to the sender's activation record.

Instruction Part: Must designate a particular instruction in a particular method. Two coordinate system is used for identifying instructions:

① object pointer ② relative offset

Environment Part: It deals with local variables (variable defined inside the method) global variable (class variable or Instance variable).

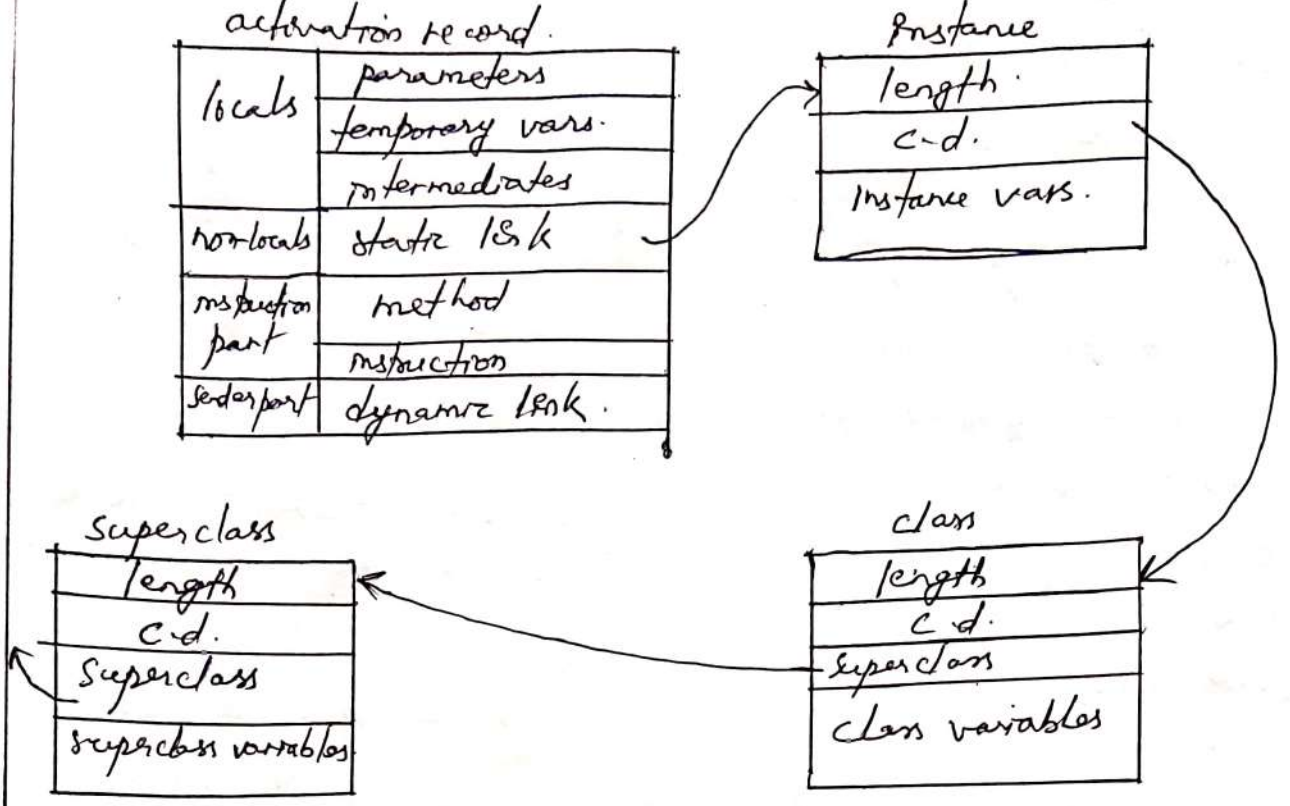


Fig: Parts of an Activation Record.

→ The local environment include space for the parameters to the method & the temporary variables along with hidden temporary variables.

→ The non local ~~variables~~ environment includes all other visible variables, namely, instance variables and class variables.

Note: Under implementation: classes & Objects of Smalltalk, we have to discuss about following (all of which we have discussed above).

i) Object Representation.
ii) Class Representation
iii) Activation Record Representation.
iv) Message Sending & Returning.

[In exam, you may be asked to describe these individual implementations].

Imp.

Q: Explain Message Passing & Returning mechanism in SMALLTALK.

While messaging to an object (Message passing) we will analyse following steps.

i) Create an activation record for the receiver (callee).

ii) Identify the method being invoked by extracting the template from the message & then looking it up in the message dictionary for the receiving objects' class or superclasses.

iii). Transmit the parameters to the receiver's activation record.

iv) Suspend the sender (caller) by saving its state in its activation record.

v). Establish a path (dynamic link) from the receiver back to the sender & establish the receiver's activation record as the active one.

While Returning from a method, this process must be reversed as

i) Transmit the returned object (if any) from the receiver back to the sender.

ii) Resume execution of the sender by restoring its state from its activation record.

→ We do not explicitly deallocate activation records since activation records are created from free storage & reclaimed by reference counting, just like other objects.

→ Smalltalk also provide interrupt facility that automatically interrupts the executing task after it has run for a certain amount of time which support concurrency.

Q. Translate the following expression into LISP.

① $\frac{1}{2}\sqrt{4r^2 - l^2}$

⇒ (* (/ 1 2) (sqrt (- (* 4 (expt r 2)) (expt l 2))

② $\frac{n!}{r!\,n-r!}$

⇒ (/ (fac n) (* (fac r) (fac (- n r))))

③ $\frac{\pi R^2 E}{180}$

⇒ (/ (* (pi (expt R 2) E)) 180)

④ $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$

⇒ (quotient (plus (minus b)
(sqrt (difference (expt b 2)
(times 4 a c))))
(times 2 a))

Don't write minus here. In class I got confused here, so this is actual expression.

other Method:

⇒ (/ (+ (-b) (sqrt (- (expt b 2) (* 4 a c))))
(* 2 a))

⑤ $r_1, r_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. We want to define a function roots that returns a list containing the 2 roots.

(defun roots (a b c)
(list (quotient (plus (minus b)
(sqrt (difference (expt b 2)
(times 4 a c))))
(times 2 a))
(quotient (difference (minus b)
(sqrt (difference (expt b 2)
(times 4 a c))))
(times 2 a))))

FORTRAN PRACTICAL (The program may be asked in Board Exam) so, Practise more.

1) This is our first program

```
program hello
  implicit none
  print *, "Hello World!"
end program
```

1. Description (optional)
2. Write "program program-name"
3. Write "implicit none"
4. Executable statements
5. ..... Declaration section.
6. End program.

Rules

1. Each program must declare its name in the very first line.
2. All programs must be enclosed between "program" and "end program".
3. Anything written after "!" will be considered as a comment.
4. Literal strings must be enclosed within double quotes.
5. Fortran is case insensitive
6. Multiple assignments in one line is allowed, but they should be separated by semicolons ";"
7. All the variables must be declared for there type at the top of the program before they can be used.
8. It is possible to continue a line to the next line if its gets too long - just put an ampersand (&)
9. A variable name must start with a letter, then you are allowed to put numbers OR "_"
   ABC123 (allowed) 123ABC (not allowed)
10. A variable name can be no longer than 31.

** These are few important rules which you have to remember while programming in FORTRAN (Not so important for exams).

** Let us write some FORTRAN which are useful (for your examination).

Note: Implicit none declaration is always used. It specifies a type & size for all cases defined names that begin with any letter, either a single letter or in a range of letters. No need to declare variables, they are automatically provided by framework.

Examples

① WAP to assign 1 2 3 4 5 to array

```
program array
  implicit none
  dimension j(5)
   k=1
   do 100 i=1,5
    j(i)=k
    write(*,*) "a=",j(i)
    k=k+1
  100 continue
end program array
```

② WAP to display from 1 to 10.

```
program hello
 implicit none
Do 100 I=1,10
  // Write(*,10,advance='no')
   I
Sum=sum+I**2
 // 10 format(i2)
  100 Continue
end program hello
```

③ Program to add 2 complex number.

```
program complex
 implicit none
  complex :: cx1, cx2
   read (*,*) cx1
   read (*,*) cx2
  write(*,*) cx1+cx2
end program complex
```

④ Ways to display squareroot of number from 1 to 5.

```
program square
 implicit none
   do 100 i=1,5
    write(*,900,advance='no') i**0.5
     900 format(f 8.3)
  100 continue
end program square
```

(5) Display
```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```
in FORTRAN.

→
```
program name
implicit none
  do 10 i=1,5
    do 20 j=1,i
      write(*,500,advance='no')j
      500 format(i3)
    20 continue
    write(*,*)
  10 continue
end program name
```

(6) Program to sort in ascending order.
```
program ascend
implicit none
  dimension num(5)
  do 5 i=1,5
    read(*,*)num(i)
  5 continue
  do 10 i=1,5
    do 20 j=i+1,5
      if(num(i).gt.num(j))then
        k=num(i)
        num(i)=num(j)
        num(j)=k
    end if
    20 continue
    10 continue
  do 30 i=1,5
    write(*,*)num(i)
  30 Continue
end program ascend
```

(7) program to add & sub.

```
program addsub
implicit none
integer :: a, b, c, d
print *, "type value of a and b"
read *, a, b
  c = a + b
  d = a - b
print *, "addition is, "c, " Subtraction is "; d
end program addsub.
```

(8). Program string concatenation.

```
program string
implicit none
  Character (len = 4) :: first_name
  character (len = 5) :: last_name
   character (10)  :: fullname
   first_name = 'John'
   Last_name = 'Smith'
! string concatenation
fullname = first_name // ' ' // last_name
print *, fullname.
end program string
```

9 WAP to display the sum of first n natural number.

```
program sum
implicit none
integer :: n, i, s = 0
read *, n
  do i = 1, n
   s = s + i
end do
print *, "the sum of first n natural number is", s
end program sum.
```

10. Write a fortran program which find the average of number N numbers.

```
program average
implicit none
Real prod, avg
read *, n
prod = 1
do i = 1, n
prod = prod * float(2*n-1) / float(n**2)
end do
avg = prod / float(n)
write (*, 7) avg
7 Format (3x, f10.5)
end program average.
```

11. Calculating the square root and cube roots of set of value

```
program square_root_and_cube_roots
implicit none
integer, parameter :: max_size = 10.
integer :: j
real, Dimension (max_size) :: value ! Array of numbers
real, Dimension (max_size) :: square_root ! array of sqare roots
real, Dimension (max_size) :: cube_root ! Array of cube roots.
! Calculate the square root & cube root now.
do j = 1, max_size
value (j) = real (j)
square_root (j) = sqrt (value (j))
cube_root (j) = value (j) ** (1./3.)
end do
write (*, 100)
Format (20x, 'Table of square and cube roots', /,
        4x, 'Number  Square root  cube root',
        3x, 'Number  Square root  cube root', /,
write (*, 110) (value (j), square_root (j), cube_root (j), j = 1, max_size)
Format (2 (4x, f6.0, 6x, f6.4, 6x, f6.4))
end program .....
```

if Else program example:

```
program ifelseProg
implicit none
  ! local variable declaration
   integer :: a = 100
  ! check the logical condition using if statement
   if (a < 20) then
  ! if condition is true then print the following
   print*, "a is less than 20"
   else
   print *, "a is not less than 20"
   end if
   print *, "value of a is", a
end program ifelseProg.
```

program do loop:

```
program printNum
implicit none
! define variables
integer :: n
do n = 12, 20, 2
! printing the value of n
print *, n
end do
end program printNum.
```

array program example:

```
program arrayProg
real :: numbers(5) ! one dimensional array
! assigning some values to the array numbers
do i = 1, 5
 numbers(i) = i * 2.0
 end do
! display the values
 do i = 1, 5
 print *, numbers(i)
 end do
! short hand assignment
 numbers = (/1.5, 3.2, 4.5, 0.9, 7.2/)
 ! display the values
 do i = 1, 5
 print *, numbers(i)
 end do
end program arrayProg.
```

☆ program on function call in fortran example

```
program program_name
implicit none
  ! start of the specifration part
  ! the integer variable
INTEGER :: the_integer, the_output
! The character variable
CHARACTER (LEN=6) :: the_character
! End of the specifration part
! start of the execution part
! putting the variables to use
the_character = "World!"
the_integer = 1
the_integer = 1 + the_integer
print *, "Hello!", the_character
print *, the_integer
! calling the function Addition (a, b)
print *, "addition:", Addition (the_integer, the_integer)
! calling the subroutine Subtraction (a, b, c)
call Subtraction (the_integer, the_integer, the_output)
print *, "Subtraction: ", the_output
  ! End of the execution part
! Start of subprogram part
contains
    INTEGER FUNCTION Addition (a, b)
     implicit none
     integer :: a, b
     addition = a + b
   end function Addition
   SUBROUTINE Subtraction (a, b, c)
     implicit none
     integer, intent(in) :: a, b
     integer, intent(out) :: c
     c = a - b
end of subroutine Subtraction. ! End of subprogram part.
```