Principles of Programming Language

1. Define programming language? write its importance or studying languages for software manager, language designers and implementers.

→ Programming languages are set of rules that converts strings, syntax and its semantics, into numeric values, instructions into machine code output.

Programming languages are used by programmers to create softwares, implement algorithms, create simulation, write scripts or manage a network.

possess common features like, abstractions, Expressive power, Functional programming.

It is important to study programming languages for software manager, language designers and implementers as:

1) programming languages are tools

that can be used in all fields of sciences such as physics to simulate atomic particles, in chemistry to simulate chemical reactions. or in medicine to ~~monitor~~ monitor a patient's vitals ~~signals~~ by mapping these analog signals to digital

2> Programming languages can be used to script minial tasks like web scraping, data entry or inventory management

3> Programming languages can be used in research in all fields where they can go through all combinations of possible dna in biotechnology whereas in neuroscience they can ~~easily~~ help understand neurons in the brain.

4> Programming languages and Probability can be used together to write prediction algorithms to predict stock markets weather. They can also be used in AI.

5> Programming can be used in scientific calculations as well as quantum computing

2. Phenomenology of programming languages?

→ since programming languages are tools, below are its phenomenology:

phenomenology : philosophy of experience.

1. Tools are : Appliative and Reductive

→ high level programming languages such as Java have automatic memory management allocating and de-allocating objects, by the use of garbage collector. Thus we are not required to implement memory management logic in our application.

But due to this certain memory problems occur such as memory leaks, as well as inefficient memory usage since all the memory management has been handed over to a tool.

hence the memory management features of Java make it Appliative while its issues make it reductive.

2. Fascination and Fear are Common reactions to new tools.

-Scripting capabilities of programming languages allows for scripting of any repetive task on the computers. This feature of programming is fascinating but with this feature, data entry jobs and inventory management jobs become useless and once written script can run forever with no pay, rest days etc. This is a common fear.

Just like this machine learning are used in predicting what we will buy which is a fascination. but machine learning can be used to predict things for us giving us an imaginary free will to choose from what has already been selected for us by AI.

3. With Mastery, Objectification Becomes Embodiment!

when introduction to programming language as a tool. a person is unable to use it without prior experience or guidance.

he experiences this new tool as or "alienated otherness" something he doesn't understand and cant use. he treats it like a foregin object.

But with experience on it and its mastery he learns to make it part of his arsenal and slowly embodiment of himself by using it to solve his problems.

4. programming languages Influences Focus and Action:

- design patterns used in a programming language promotes certain techniques, algorithms etc that may not have been used in other languages

→ programming languages do influence coding practises and semantics. which may intern drive technological development towards certain goals and away from some deemed bad techniques by the language.

→ Characteristics of a good programming language!

1. A programming language must be simple, easy to learn and use, have good readability, and human recognizable.

2. Abstraction is a must-have characteristics for a programming language in which the ability to define the complex structure and then its degree of usability comes.

3. A portable programming language is always prefe

4. Programming language's efficiency must be high so that it can be easily converted into machine code and executed, consuming as little memory as possible.

5. Programming language must be well structured and documented to be suitable for application development.

6. Necessary tools for development, debugging, testing maintenance of a program must be provided by a programming language.

7. programming language must be consistent in terms of syntax and semantics.

<4>

major programming paradigms includes:

1. Imperative
2. logical
3. Functional
4. Object-oriented.

Imperative paradigm assumes that computers can maintain through environments of variables any changes in computational process.
Computations are performed through a guided sequence of steps, in which these variables are referred to or changed.
here order of ~~changes~~ steps ~~also determine~~ is crucial as given step will have different consequences.

depending on the current values of variables when step is executed.

Logical paradigm:

→ Declarative approach is taken to problem solving where various logical assertions about a situation are made establishing all known facts. Then queries are made. The role of computers becomes maintaining data and logical deduction.

Functional paradigm:

Functional paradigm views all subprograms as functions in mathematical sense - informally, they take in arguments and return single solution.

The solution returned is based entirely on the input, and the time at which an function is called has no relevance.

Object Oriented Paradigm:

here real world objects are each viewed as separate entities having their own state which is modified only by built in procedures, called methods.

Because objects operate independently, they are encapsulated into modules which contain both local environments and methods.

Communication with an object is done by message passing.

Objects are organized into classes, from which they inherit methods and equivalent variables reusable and code extensibility

* any thing solvable using one of these paradigms can be solved using the others: however certain types of problems lend themselves more naturally to specific ~~language~~ paradigms.

explain application of pseudocode in programming?

→ ① pseudo code uses interpreter that saves storage as pseudocode is much more compact than a machine code.

② pseudo code's interpreter allow for pseudo code's own set of data type (floating points) and operators (eg. indexing). this made pseudo code like a virtual computer. ; regardless of data type imposed by hardware making ~~progra~~.

③. The virtual computer concept also made pseudocode programming easier to understand and study as they ~~looked~~ lacked special cases, and followed Regularity principle.

④. Concepts like algorithms could be tested and implemented in pseudocodes as well as be ~~concepts~~ all mathematical concepts could be implemented in pseudocode.

dont write all in caps

5. pseudocode has ~~fo~~ implementations and techniques that allowed, Indexing and loops, comparisions and Control flow, Operators and Operands, Input and Output as well as moving data around which allowed it to behave ~~as an~~ just like any other programming language, giving instructions to computer.

how can pseudocode interpreters Simplify programming?

→ In early computers Floating point operations and Indexing were missing.

In Case of Floating point operations, early computers were used in mathemdtical / numerical computaton due to lack of built in floating point operations; manual scaling had to be done to keep them in range of integer arithmetic facilities of computer & floating point subroutines had to be written for floating point additions, subtractions, look etc. which slowed the program.

In case of Indexing, the ability to add variable index quantity to a fixed address in order to ~~express~~ address an element of an array was also missing, so had to be done by address modification which was error prone.

So this led to the creation of pseudo code where instruction code can be different from machine code.

~~which allowed~~ pseudo codes interpreters allow pseudo code to implement of virtual computer with its own set of data types (eg. floating point) and operations (eg indexing). ~~in terms~~ of which would be interpreted to machine code with its own data types and operation's.

here pseudo code would be a higher-level so in this way pseudo code simplified programming.

7.

what are programming domains? how does logical programming differ from functional programming?

→ programming and programming languages can be used in different ~~respects of life human life~~ fields of education, science, health, bussiness and research. These fields constitute programming domains, eg. AI, scripting, Web Applications, Enterprise Resource planning etc.

Logical Programming:

This paradigm is based of first order predicate calculus. This programming style emphasizes the declarative description of a problem rather than the decomposition of the problem into an algorithmic implementation.

A logic program is a collection of logical declarations describing the problem to be solved. As such, logic programs are close to specifications. The problem description is used by inference engine to find solution.

Logic program consits of

1> axioms - defining facts about objects
2> rules - defining ways for inferencing new facts.
3> a goal statement - defining a theorem potentially provable by given axios and rule

Functional programming

It is based on the theory of mathematical functions, more precisely on the lambda - calculus.

It allows programmer to think about a problem at a higher level of abstraction.

Functional language has 3 main sets of components

a> data objects - such as list, array
b> built in functions - for manipulating basic data objects.
c> functional forms - also called high order functions for building new functions.

8>

"The complexity of programming led to the development of program design notations". explain with reference to pseudo code.

→ program design notations where precursors to programming languages.

→ Von Neumann and goldstein's flow diagrams where developed into flow charts still used today for program design.

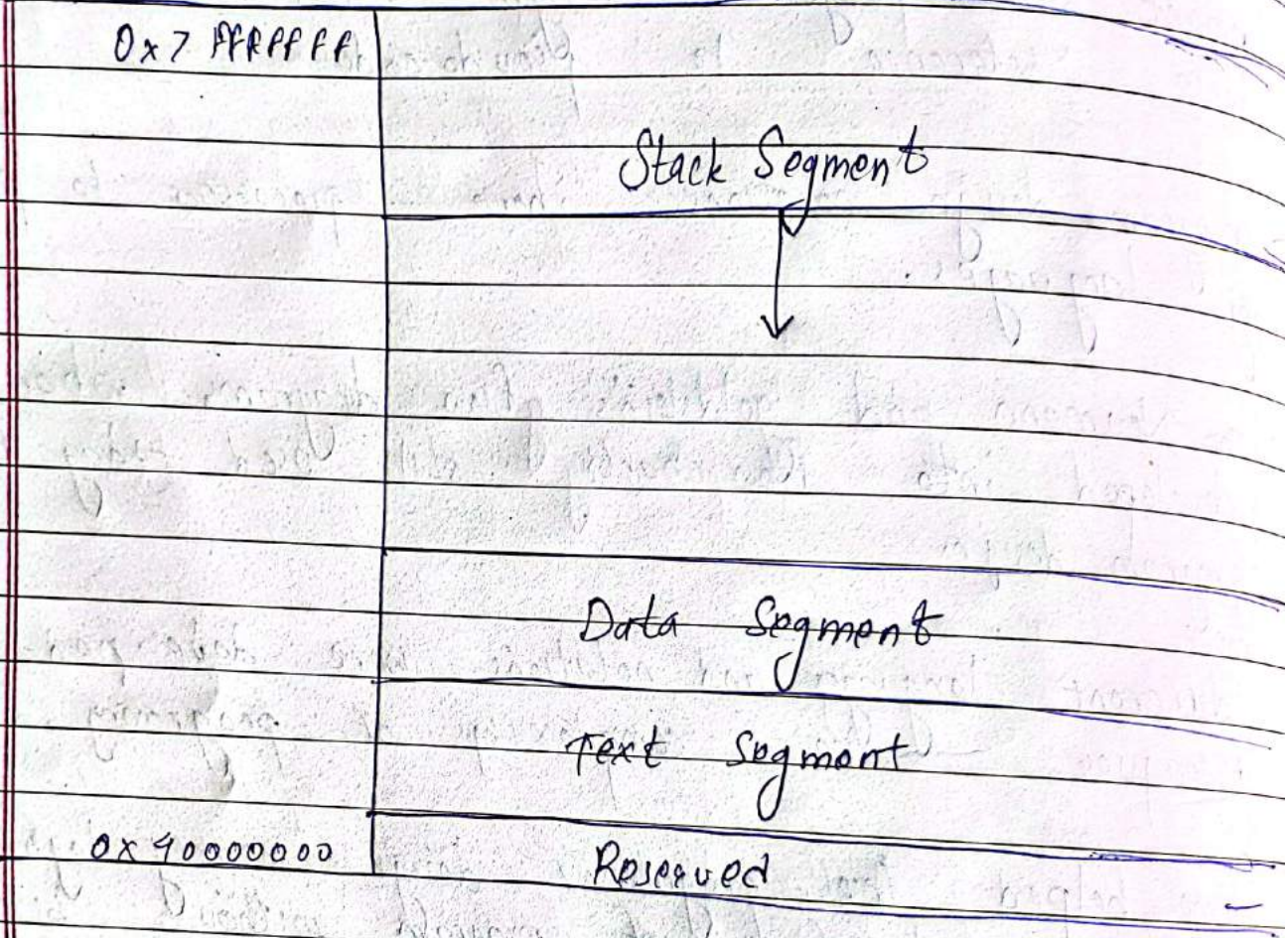
→ different languages and notations were developed to conquer the complexity of programming.

→ Some helped the programmer design memory layout and control flow of program without being concerned with details.

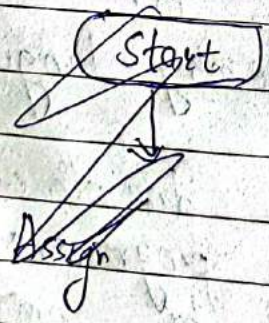
Others provided mnemonics for the machine operations, much like assembly language.

programming languages with data structures where were created to be used by programmers and not direct processing by computers.

eg memory layout :

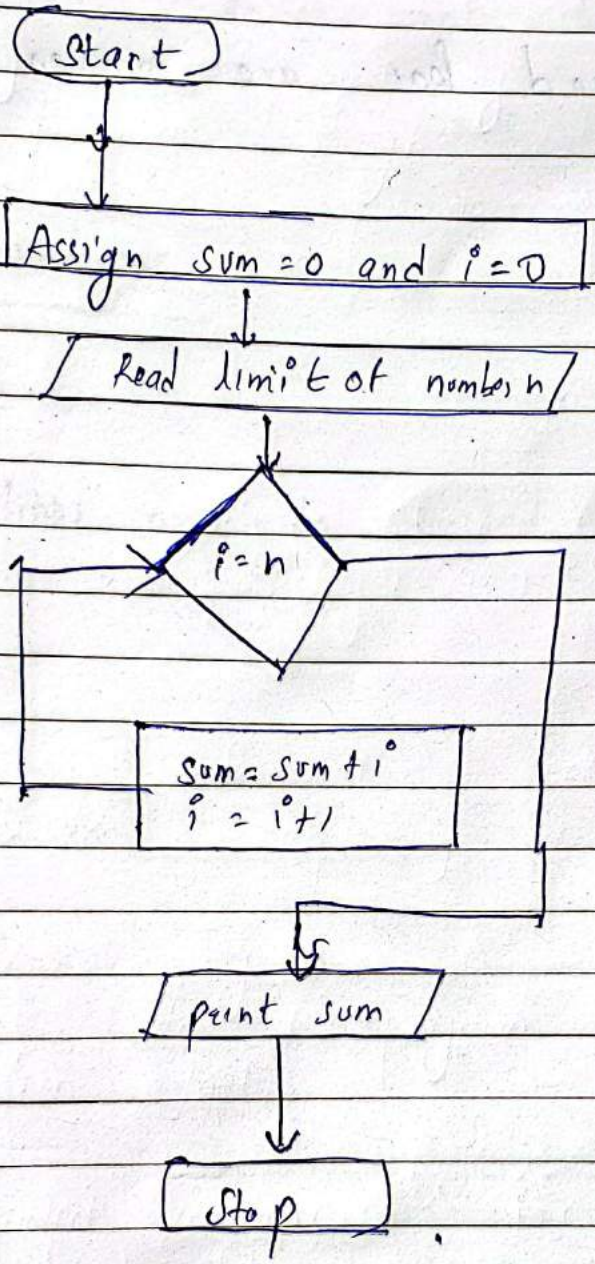


eg ~~Flow chart~~

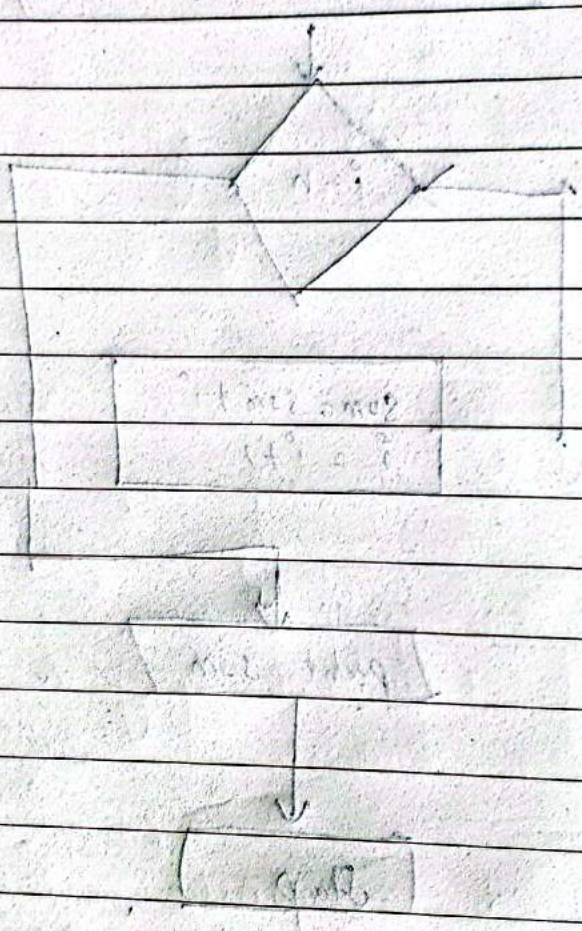


classmate
Date
Page

Eg: Flow chart to find sum of N natural numbers.



"Fascination and fear are common to new tools"



10. Explain design of pseudocode and its implementation highlight functional enhancements brought by the pseudocode.

Design of Pseudocode:

1. Basic capabilities must be decided:

FORTRAN

Describe the name structure of Fortran language?

→ name structure: structure / organize names in programs

1. The primitive bind names to objects.

In FORTRAN: syntax;

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

INTEGER primitive declares variable I, J, K.
meaning storage of 1 word must be allocated for these variables and names I, J, K must be bound to the addresses of those locations.

here use of INTEGER also specifies its type.

2. declarations are non executable.

declarations are non executable statements unlike initializations, if statements, Goto's or calls

non executable statement provide information to compiler and preexecuting procedures.

~~declarations provide~~ Compilers keeps tracks of all locations that have been allocated and allocation done once before execution of program (Static allocation).

Compilers keep track by ~~placing~~ maintaing a symbol table.

Name	Type	Location
I	INTEGER	0.256
J	INTEGER	2.257
K	INTEGER	2.258.
.		

3. Optional variable declarations are dangerous.

Older versions of fortan allowed automatic declarations of variables.

This was done by ~~making~~. ~~variab~~ using 'I through N' rule ie i, j, k, l, m, n are all integers while rest are declared reals.

this led to ~~ob~~ obscure names declared as variables like ISUM, KOUNT, XLENGTH. and not just

that errors that would be found during compile time ~~without~~ in cases without automatic declarations would cause undebuggable problems.

eg:

COUNT = COUMT + 1

undeclared, should've been count

here COUMT would be declared and since uninitialized would use previous value left at the memory location assigned to it.

hence dangerous.

9. Environments determine Meanings:

Eg: Consider a statement,

X = COUNT(I)

here COUNT can be an function as well as an array.

meanwhile I can be a real, or an integer

here. The context of this statement are the set of definitions visible to that statement or a construct.

A context of a construct is its environment.

5 Variables names are local in scope:

Fortran programs are divided into a number of disjoint subprograms that can be developed independently and stored in libraries. due to this they have disjoint environment containing just the variables (formal parameters) declared in that subprogram. making variables names local in scope.

6. Subprogram names are global in scope:

Subprograms have global visibility. for the program which they are declared.

Eg:

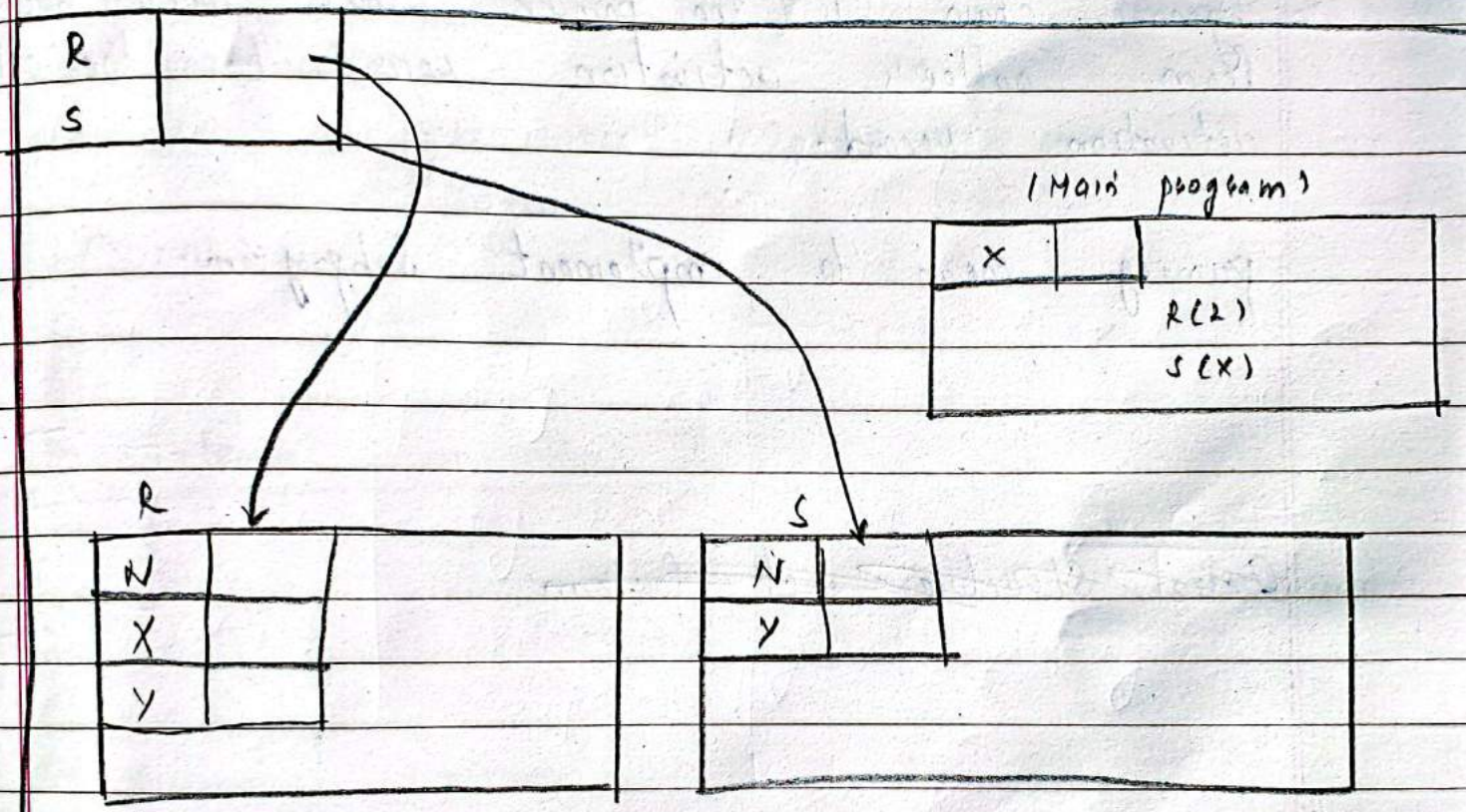
Eg:

```
C     MAIN PROGRAM
      INTEGER X
      .
      .
      .
      CALL R(2)
      .
      .
      CALL S(X)
      .
      .
      END

      SUBROUTINE R(N)
      REAL X, Y
      CALL SLY)
      .
      .
      .
      CALL SLY)
      .
      .
      END

      SUBROUTINE S(N)
      INTEGER Y
      .
      .
      END
```

Contour diagram showing scopes of variables



Dynamic chain of Activation Record:

Dynamic chain is the pointer that reach back from callee's activation record to the caller activation record.

primary mean to implement subprograms.

~~Control Structure of fortran~~

Control structure of Fortran:

those constructs in the language that govern the ~~control~~ flow of control in a program.

1. lower level Control Structure: IF-Statement, GOTO statement
2. higher level Control structure: Do-Loop
3. Subprogram control structures (CALL and RETURN).

IF-Statement in Fortran II:

Eg:

IF (a) n1, n2, n3

the above statement evaluates to,

branch to n1 when a is -ve.
branch to n2 when a is 0.
branch to n3 when a is +ve.

above equivalent to CAS instruction in IBM 704x.

IF Statement in Fortran IV:

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

EQ = Equality relation.

Passport size photo dp:

GOTO statement:

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

the above is equivalent to if-then-else.

DO LOOP

used for definite Iterations.

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

the above instructions is a command to execute the statements between the DO and the corresponding CONTINUE with I taking values 1 through N

the variable that changes its value is I ie control variable

and statements repeated is body of loop

CALL AND RETURN:

In Fortran, variables can be passed to a subroutine and subroutines can ~~return the variable as well~~ hold control flow of the program.

```
SUBROUTINE    DIST  (D, X, Y)
D = X - Y
IF (D.LT. 0) D = -D
RETURN
END
.
.
.
CALL  DIST (diff1, X1, Y1).
```

here, X1, Y1 are input variables and Diff1 is used as output variables.

Content of diff1 is modified.

<3>

Illustrate looping in FORTRAN by writing a program to find square root of the first 20 natural numbers.

```
C   The PROGRAM FINDS ~~CUBE~~ SQURE ROOT OF FIRST
C   20 NATURAL NUMBERS.
C
      DO 100 I = 1, 20
        PRINT *, i**1.0/2
100   CONTINUE
      END DO
100   CONTINUE
      END
```

<9>

Write a program to find the sum and the average of first 10 odd natural numbers

"Pass by reference is a dangerous proposition in FORTRAN". Justify with eg:

⇒ passing by reference is dangerous as it assumes all parameters passed to can be used both as in put and output

meaning an input variable may be inadvertently updated by the subprogram.

Consider the following definition

```
SUBROUTINE    SWITCH (N)
N = 3
RETURN
END
```

The above subroutine simply writes 3 in its parameter.
Therefore an invocation of CALL SWITCH (I) would write 3 in its parameter.

now consider call switch (2) invocation.

Then the literal 2 would be assigned a location in the literal table and its address of this location is passed on to the subroutine.

and hence when subroutine recieves the address of literal 2 it overides its value held in literal table as 3.

And whenever compiler will compile a program so that whe it encounters a constant 2 its value would be 3.

ie I = 2 + 2, then I would be 6.

due to this debugging would be a nightmare and "Pass by ~~prop~~ reference is a dangerous proposition in FORTRAN".

6.

"Fortran has been revised several times", Explain this statement with ~~suitable example~~. successive history

→ With the experience gained with First FORTRAN system, led Backus and his colleques to propose FORTRAN II in September 1957. While compilor made available in spring of 1958. and language still in use to this day.

A dialect called FORTRAN III was designed and implemented in late 1958., but it never achieved widespread use because of its many dependencies on the IBM 704.

In 1962 the Fortran IV programming language was designed; ~~which became very popular~~. And By 1963 virtually all manufacturers had either delivered or committed themselves to producing some version of FORTRAN. reflecting FORTRAN's popularity.

Reflecting Fortrans popularity American national Standard Institute (ANSI) began development of standard FORTRAN IV ↓ (ANS FORTRAN) in 1962 which was completed in 1966.

Even after this dialects of fortran were very common and few compilers implemented ANSI standard.

In 1977 an new ANS FORTRAN was developed that is sometimes known as FORTRAN 77. which incoperated number of ideas from later languages it was still a 1950s language in its capabilities.

After a 12-year ~~period~~ development process, a new standard, commonly known as FORTRAN 90 was produced.
although it incoperporates modern features from newer languages. It is still a minor revision.
as
FORTRAN 2000 was already on drawing board.

(5)

Give specific examples how FORTRAN IV violates principles of programming languages.

7>

how is data represented in FORTRAN? differentiate roles of arrays from those of scalar data types of FORTRAN data structure.

→ FORTRAN is a scientific programming language. Therefore data structuring methods included in Fortran were those most familiar to scientific and engineering applications of mathematics:

scalars and arrays.

1. Scalar data types (Primitives)
2. Arrays data types

Scalar

The primary data structure were primitives. which included integer

INTEGER

In Fortran, integer, floating point, and logical values all normally occupied one word

Even characters were normally manipulated in groups that corresponded to the number of characters that could fit in one word.

Each of the primitive data types is represented in a manner appropriate to the operations defined on that data type.

For example : integers are usually represented as a binary number with a sign bit.

Floating point numbers can also be represented in FORTRAN.

The convention of representing a number by a coefficient and a power 10.

Eg - 1.5×10^3.

A typical representation of floating point number is

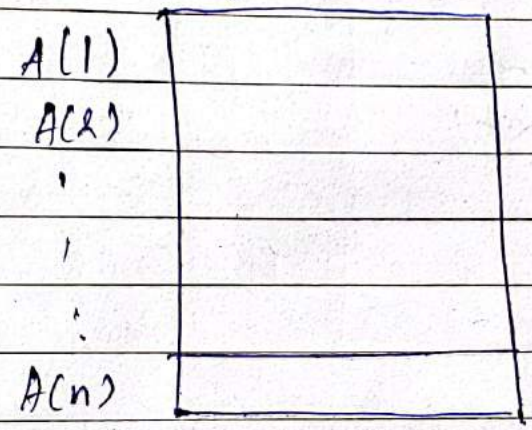
The value represented by this number is $m \times 2^c$ where m is the mantissa.

Array implementation in Fortran:

Consider a one dimensional array declared by

DIMENSION A(n)

where n represents an ~~integration~~ ~~de~~ integer denotation. The layout of A memory is



If $\alpha\{A(1)\}$ is address in memory of A(1)

then in general addressing equation for array A is

$$\alpha\{A(i)\} = \alpha\{A(1)\} - 1 + i$$

Now consider a 2 dimensional Array. it can be represented by instruction.

DIMENSION A(m,n)

Fortran arranges arrays in memory in column major order

i.e with columns occupying adjacent memory locations. as shown in figure.

addressing eqn for 2 dim array

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

A(1,1)	α
A(2,1)	$\alpha+1$
.	
.	
.	
A(m,1)	$\alpha+m-1$
A(1,2)	$\alpha+m$
.	
.	
.	
A(m,2)	$\alpha+m+m-1$
A(1,3)	$\alpha+m+m = \alpha+2m$
.	
.	
.	
A(m,n)	$\alpha+nm-1$

Column-Major layout of 2d array.

Do 100 ?

(DIST, 2, 3) ?

modes of passing parameters in FORTRAN. an example

~~1. pass by Reference~~

~~2. pass by Value~~

Fortran allows subprogram's parameter to be used as both input and output. which makes Fortran pass parameters only by reference.

Eg:

LT = less than

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

```
X1 = 10
X2 = 6
CALL DIST (DIST1, X1, X2)
```

"Subprograms are implemented using activation record in FORTRAN"

To invoke a subprogram we must transmit parameter to the subprogram.

since a subprogram can be called from many different callers and from many different places within one caller.
there is no unique place to which the callee (ie subprogram called) should return when it has finished.

hence it is necessary to tell callee who is caller is.

Also when a subprogram is called, the state of the caller must be preserved before the callee is entered and must be restored after callee ~~gets control back~~ returns.

3. a)

state of a caller includes the contents of all variables, the contents of any registers in use and its IP (Instruction pointer Register) data.

all these information must be stored in a callee private data area before the callee is entered which is called activation record.

In a non-recursive language like FORTRAN there is only one activation record for each subprogram.

hence, ~~when a call~~ before a callee is entered in its running state it is passed a reference to callee ~~by~~ passing an pointer to the caller's activation record.
which is done by storing the pointer in callee's activation record.

The pointer from callee's activation record to its caller is activation record is called dynamic link.

The dynamic chain is a sequence of dynamic links that reach back from each callee to its caller.

dynamic chain begins at currently active subprogram and terminates at the main program.

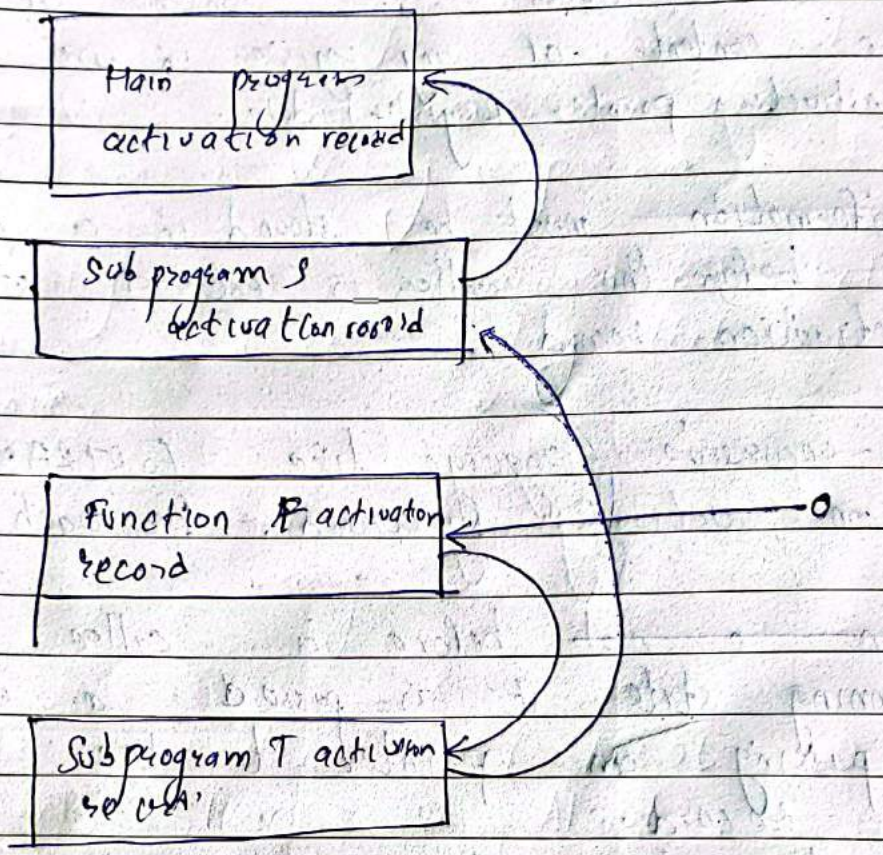


fig: dynamic chain of Activation records.

7. By introducing 2 forms of for loops:

→ Q no 3 (ans).

<4>

history of Algol and its failure factors:

Algol
<1>

Algol follows the "zero to ∞" principle. Verify it by comparing it with Fortran:

→ Algol 60 was focused on regularity meaning IE focused on fewer special cases. making it easier to learn, remember and master

Zero - One - Infinity is based on such regularity principle such that it considers only regular numbers in a programming language design to be zero, one, and ∞.

In Fortran IV design, it is filled with numbers other than zero, one or infinity.

eg identifiers are limited to six characters while Arrays can be of maximum 3 dimensions. and at most 19 continuation cards.

Six, Nineteen, three are all numbers a fortran developer must remember. which will be dificut since the number are arbitary.

According to Zero - One - Infinity the number of characters allowed in name should be zero, one or ∞.

zero, character length doesn't make sense.
one, character length in name is limiting.
whereas using ∞ means the identifier name could be as long as possible.

In reality ∞ is not allowed but identifier name's limit could be so large that is effectively is infinite.

<2>

Data Structures of ALGOL-60. describe the different forms of for-loop ALGOL:

Data Structures of ALGOL-60:

Algol-60 was intended for mathematical operations and scientific applications therefore all primitive data types used are mathematical scalars.

— Integers
real
Boolean
no double precision types.

```
procedure S(K,L)
    integer K, L;
    begin
        K := 2;
```

end;

Arrays were generalized allowing as many dimension as possible following zero - one - two rule.

They were dynamic as well

eg

integer array Number of days [-100 : 200]

Forms of for - loop in Algol - 60:

In Algol For - loops are of 2 forms, which are generalization of FORTRAN's DO - loop:

for var := exp step exp' until exp'' do stat.

for var := exp while exp' do stat

(I) In First if 1 step 1 until 5

would generate sequence 1, 2, 3, 4, 5

(II) If i := 16.

i/2 while i ≥ 1 would generate

8, 4, 2, 1. sequence.

(9)

Why "Pass-by-name" in Algol-60 considered as a dangerous and expensive method? Explain with an example.

Consider a swap program in Algol:

```
procedure swap (x, y);
    integer x; y;
    begin integer t;
    t := x;
    x := y;
    y := t;
end.
```

The above code works

```
swap (i, j);
swap (j, i)
swap (A[i], i);
```

But when in the case of swap (i, A[i])
(i, A[i])
it completely fails.

this is due to early change in values of i

ie.

~~t := A[i]~~
~~A[i] = j~~
~~i = A[i]~~

```
t = i
i = A[i]
A[i] = t;
```

the i value changed but we needed the original i value in 3rd line.

This is a sign of design mistake
also in Algol-60 scientist have also found no procedure possible to swap that works for all characters.
also additional IF's needed for those.

So pass-by-name dangerous and expasive.

BNF and EBNF. what are ~~significant use of context free and regular grammar~~. Enhanced features of EBNF that make it more efficient then BNF with example?

→ BNF is a descriptive tool for programming languages design, to formulate, record and evaluate various aspects of the developing design.

Naur adopte the Backus Notation to the Algol-60 report by making number of improvement hence, called Backus-Naur Form.

Example:

For example, integers are represented in strings like '3', '+3' and '-273' and decimal fraction represented in strings like '.08' and '.14159'.

What we are doing is defining certain syntactic categories of strings like decimal numbers in terms of other syntactic categories like unsigned integers.

So in its BNF would. it represents classes of strings by words or phrases in angle brackets such as <decimal fraction> and <unsigned integer>.

So BNF for integer would be

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

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

::= means is defined as

'|' = OR.

sequence of one or more <digit>
is an unsigned integers.

EBNF :

extensions were made to BNF to ~~make it~~ improve its readability.

eg: In BNF.

Unsigned integer was expressed by recursive definition.

<unsigned integers> ::= <digit> | <unsigned integers> <digit>

a better notation would be with the use of Kleene Cross, C^+ which means sequence of one or more strings of category C.

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

Also Kleene star, C^* which means sequence of zero or more elements of class C.

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

{ } dialect allows meaning substitute.

<identifier> ::= <letter> { <digit> | <letter> }*

$\Rightarrow$ { <digit> { <letter> }* }

an improvement would be:

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

→ by BNF

Now By EBNF

<integer> = [+ | -] <unsigned integer> | <unsigned integer>

even better

<integer> = [+ | -] <unsigned integer>

↳ either +, or -, or none

Significant uses of context free grammars and regular grammar?

grammar is a tool for describing programming features. eg: BNF and EBNF notation

context free grammar and regular grammar are classes of grammar.

A regular grammar is the one written in EBNF without any recursive rules.

Eg.

<number> ::= [+ | -] { <decimal number> | <decimal number>]$_{10}$ <integer> }

Therefore a regular language is the one described by regular grammars.

A context free grammar is any grammar that can be expressed in extended BNF, possibly including recursively defined nonterminals.

Therefore a ~~recursive~~ context free ~~grammar~~ language is the language described by context free grammar.

And all regular grammar are also context free grammer.

So all regular languages are also context free languages.

Eg. Context free grammer : describing Algol's definition of

<Statement> ::= { <unconditional statement>
if <expression> then <unconditional statement>
if <expression> then <unconditional statement>
else <Statem

where

<unconditional statement> ::= { <assignment statement>
for <for list> do <Statemen

A major significant use of recursion is it allows nested syntactic structure.

hence.

In regular grammer indefinetly deep nesting not possible,

Control Structure, data structure, name structure and syntactic structure in ALGOL.

Control Structure in Algol

→ Control structure direct and manage the flow of control from one assignment statement to another.

→ Control structure are generalizations of fortran's in Algol:

In Algol 'IF' done be removing constraint of fortran having IF required to be a single, simple, unconditional statement (GOTO or call) and by introducing else that allowed consequent of if to be a group of statements.

```
IF T[middle] = sought then location := middle
else lower := middle + 1 ;
```

In Algol for -loop also generalization of fortran's do loop.

for (i) = 1 step 2 until N x M do
inner [i] := outer [N x M - i]

It also has a variant that is similar to while loops found in Pascal.

for NewGuess := Improve (OldGuess)
while abs (NewGuess - OldGuess) > 0.0001
do OldGuess := NewGuess;

nested statements:

need of → reduced use of goto but still syntax correct

Algol eliminated goto by using compound statement

if condition then
begin
statement 1;
:
statement m
end
else
begin
statement 1;
:
Statement n
end

~~Data Structures for Algol~~

Switch cases in Algol:

```
begin
  switch marital status = single, married, divorced,
  widowed;

begin
  switch marital status := single, married, divorced,
  widowed
  ;
  goto marital status [i];
single :
        goto done
married :
        goto done;
divorced : ... handle divorced case ...
        goto done;
widowed : ... handle widowed case ...
done : ...
end
```

Data Structures in Algol:

← On 3.

Name Structure in Algol:

1) The primitives bind names to objects similarly to Fortran

eg real x;

but unlike Fortran where a variable name is statically bound to a memory location.

In Algol a single variable may be bound to number of memory locations and these bindings can change in run-time.

2) ~~Several statements~~ compound statements:

compound statements in algol allowed : Be sequence of statements wherever only one statement was permitted.

eg Be i:=1 step 1 until N do
sum := sum + Data[i]

could be :

```
for i = 1 step 1 until N do
    begin
    if Data[i] > 100000 then Data[i] := 100;
        sum := sum + Data[i]
        Print Real (sum)
    end
```

3> blocks define Nested scope as well as simplify large programs

in Fortran scopes were nested in 2 levels, but Algol allows as many depths as possible with blocks. also simplifying the program in

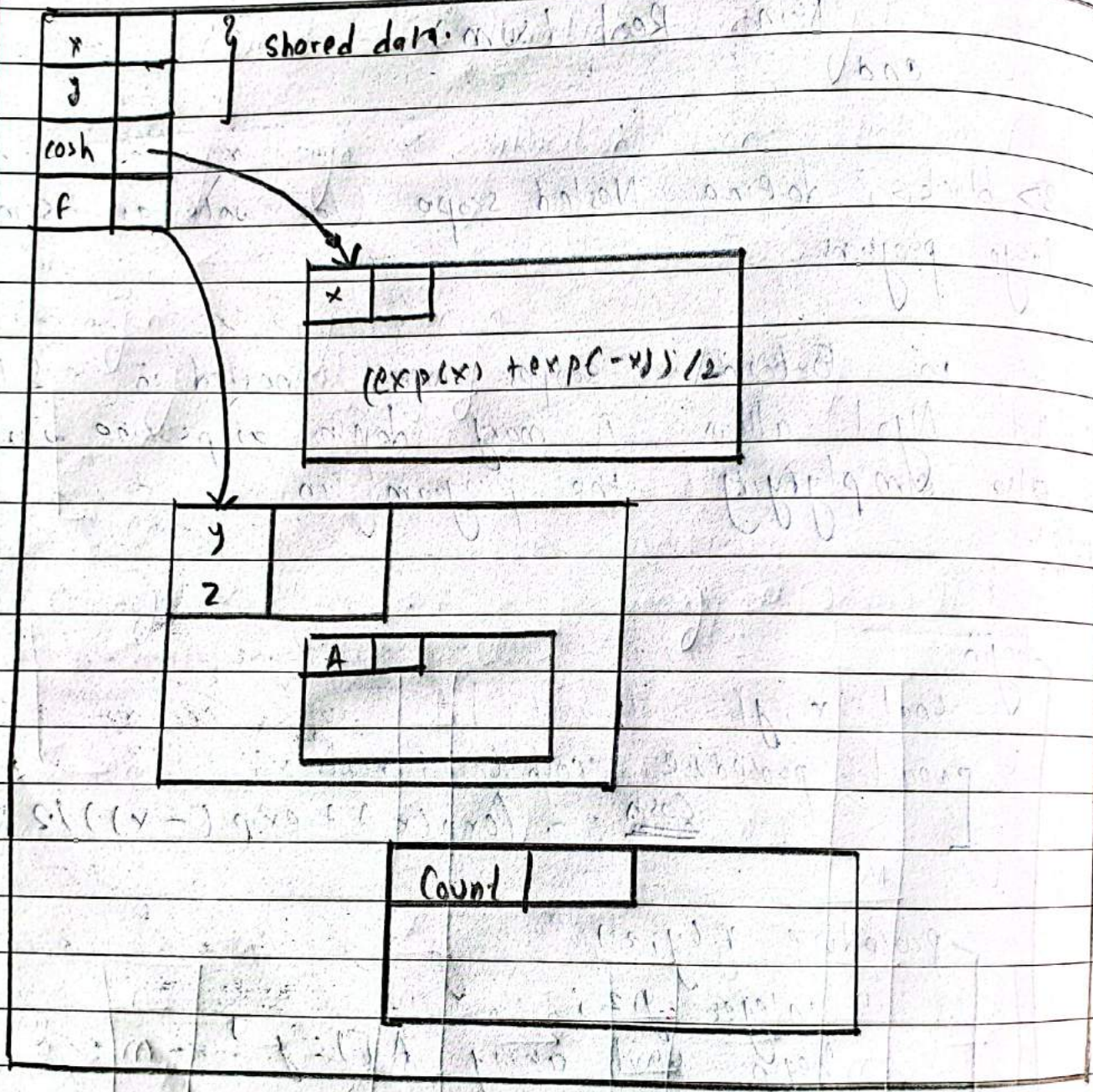
```
begin
    real x, y
    real procedure cosh(x); real x;
        cosh := (exp(x-1) + exp(-x))/2;

    procedure F(y,z)
        integer y,z;
        begin real array A[1:y]
        :
        end

    begin integer array Count[0:99];
        :
    end
end
```

in algol done declaration of parameters before

Contour diagram of Nested Scopes



Dynamic scoping and static scoping both allowed in Algol.

In dynamic scoping the meaning of statements and expressions are determined by the dynamic structure or the computations evolving in time.

In static scoping of the meanings of statements and expressions are determined by the static structure of the program.

Eg:

```
a: begin integer m;
   [procedure P;
      m := 1;
   [b: begin integer m;
      P
   end;
   P
   integer K;
   K := 2
   end
end
```

here, when procedure P is called with block b but outside b the value of m of block a becomes m = 1.

while being called from within block b value of m of block b becomes 1.

This is dynamic scoping.

meanwhile, K = 2 is defined statically and doesn't change in run time.

Block structured ~~Storage~~ allocation in Algol:

→ done by storage Reallocation on Stack in Algol

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

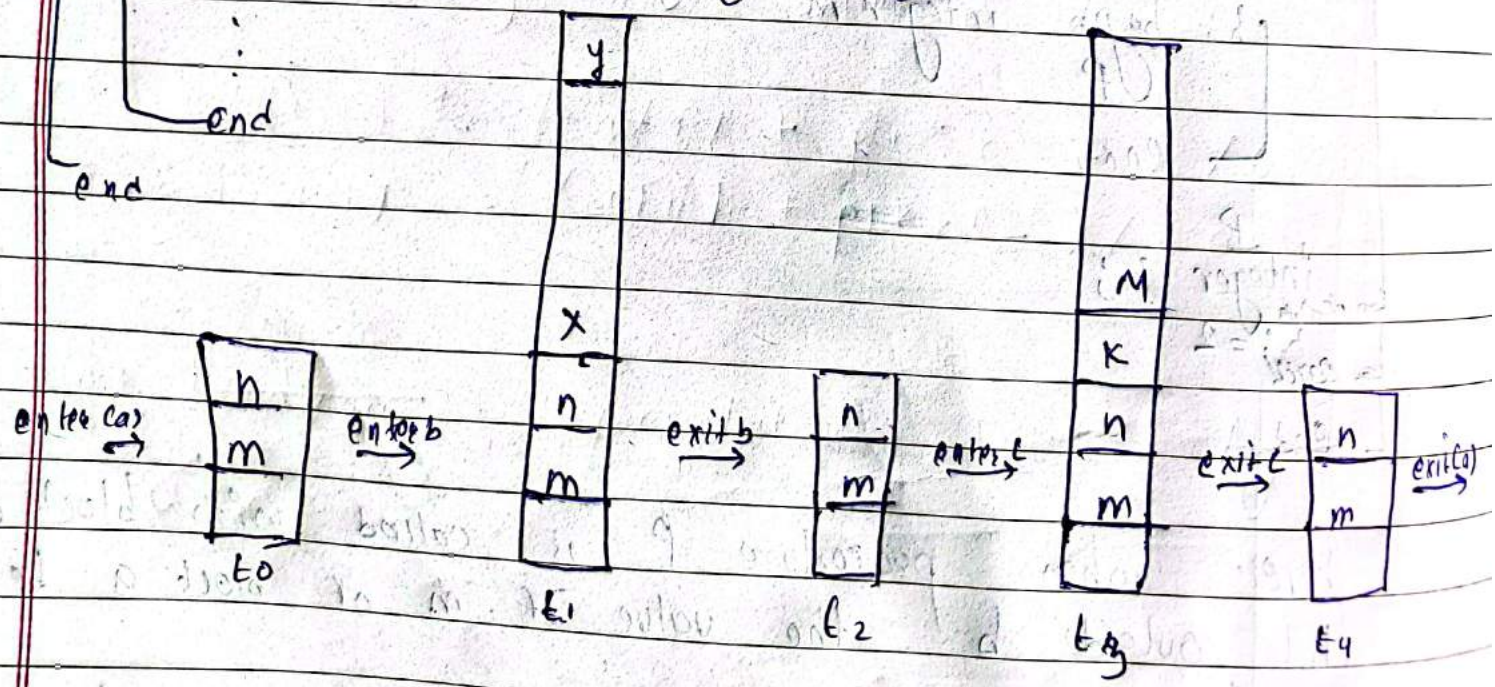


Fig Storage reallocation on Stack.

Syntactic Structures of Algol:

1> Machine Independence and Portability:

Algol focus on making itself machine Independent, hence it adopted free format ie format independent of columns or other details or the layout of program. neither did it force any layout on programmers. unlike fortran.

Eg Algol allowed continuous stream ie English style.

sum := 0; for i := 1 step 1 until N do begin read val; read val (val); x := 5; y := 7; sum := (x+y);

2> 3 levels of Representations used:

In Algol 3 levels of languages used.

(1) Reference language

It is used in all examples in the Algol report.

Eg:

a[i+1] := (a[i] + pi x i↑2) / 6.02₁₀23;

2. publication language

This level of language was intended for publishing algorithms allowed for ~~sup~~ subscripts and greek letters as well as other lexicals and printing conventions.

$$a_{i+1} \leftarrow \{ a_i + \pi \times t^2 \} / 6.02 \times 10^{23};$$

3. hardware representation;

The hardware representation was up to the Algol implementors standard.

Eg in europe the standard for decimal was ',' comma while in USA a period (.)

```
a[i+1] := (a[i] + pi*t**2)/6.02E23
A[i+1] := (a[i] + pi*t^2)/6.02E23;
```

There three hardware representations make Algol portable

37 Algol allowed ~~tde~~ keywords to be used as identifiers there are 3 lexical conventions ~~in Algol~~.

① Reserved words:
words like 'if', 'procedure' etc reserved by language.

② keywords:
here when an reserved word used as an identifier name it is marked by a symbol.

Eg .XIF , or .#do.

Algol approach

③ keywords in context:
This is Fortran's approach.
where

```
IF   IF THEN
       THEN = 0;
     ELSE
       ELSE = 0;
```

context matters of ~~variable~~ keywords. using = to indicate it is identifier.

9. Algol doesn't allow consequent of IF statement to be a unconditional statement.

eg IF B then IF C then S else T

here, does else T go with first IF or 2nd.

→ this is dangling else problem.

Solved in Algol by preventing IF statement's consequent to be an unconditional statements.

<2>

How Algol changed programming in efficient way?

Rosyr

machine independent
lexical probe
py 121 — else which allowed goo
recursion
py 112 programming
Algol has many
block simplify

Algol changed programming in efficient way by:

By.

a> Being machine Independent and introducing portability

— from 8n 5. cars,

+ 3 levels of representation

b> By introducing else to an IF

In Boolean simple logical IF statement was used:

IF (logical expression) simple statement

where consequent or then part, of the IF is required to be single, simple and unconditional statement (such as go to call)

In Algol this restriction is removed and consequent is allowed to be ~~any~~ group of statements

and also ~~an~~ extension by adding alternate to this if with else was done

Eg. IF T[middle] = sought then location := middle
else lower := middle + 1

If condition then
begin
statement 1;
.
.
statement m;
else
begin
statement 1;
.
.
statement n;
end

need of
Eliminating goto made programms much easier and ~~readable~~. computer scientist ~~began~~ like Edsger djikstra, Peter Naur, Peter Landin experimented without goto statement.

6. By allowing recursion:

integer procedure fac(n):
value n; integer n;
fac := if n=0 then 1 else
n × fac(n-1);

→ code for factorial.