Page No.
Date: / /

Emphasis on Efficiency - FORTRAN

2.1 History And Motivation

- Backus & 3 associates
- 1954 - preliminary external specification for "the IBM Mathematical FORmula TRAnslating system, FORTAN".

- ANSI developed FORTAN IV (ANS FORTAN) in 1662 and completed 1966.

- In 1977 a new ANS FORTAN was developed a.k.a FORTAN 77

- & then FORTAN 90 (after 12 yrs of F77)

Page No.
Date: / /

Q. Program in FORTRAN I to compute the average of the absolute value of an array.

```
      DIMENSION DTA(900)
      SUM = 0.0
      READ 10,N
10    FORMAT (I3)
      DO 20 I = 1, N
      READ 30, DTA(I)
30    FORMAT (F10.6)
      IF (DTA(I)) 25, 20, 20
25    DTA(I) = -DTA(I)
20    CONTINUE
      DO 40 I=1, N
      SUM = SUM + DTA(I)
40    CONTINUE
      AVG = SUM / FLOAT(N)
      PRINT 50, AVG
50    FORMAT (1H, F10.6)
      STOP
```

Page No.
Date: / /

2.2 Structural Organization

→ Program are divided into Disjoint sub programs.

⇒ Over all structure of FORTRAN program is main program and zero or more subprograms :

main program

Sub program$_1$

⋮

Sub program n

→ Constructs are either declarative or Imperative:-

programs consist two part

declarative
⇒ describes data areas, their length, and initial values

A.K.A
Non Executable ~~& Executable~~ statements in FORTRAN

Imperative
⇒ contains the command ~~to be executed~~ during the running of program.

A.K.A
Executable statement in FORTRAN

Page No.
Date: / /

⇒ Declarations include Bindings and Initialization

⇒ Usually declaration performs three functions:
1) They allocate area of memory of a specified size.
2) They attach symbolic name to that area of memory. A.k.a binding a name to an area of memory.
3) They Initialize content of that memory.

e.g

DIMENSION DTA (900)

causes loader to allocate 900 words and to bind the name DTA to this area.

DATA DTA, SUM / 900*0.0, 0.0

will initialize variables

→ Imperative are either computational, control Flow or I/O

⇒ imperative statements are of 3 types
1) Computational statement - arithmetic, move
2) Control-flow - comparison and looping
3) Input-output - namely Read and Print

Page No.
Date: / /

⇒ Program Goes Through Several Stages to Be Run:

⇒ Program progress through a series of stages.

1) Compilation — translate FORTRAN program into relocatable object code.
2) Linking - addresses the need for incorporating libraries
3) Loading - placed in computer memory
4) Execution - turning control of computer is turned over to program in memory.

⇒ Compilation Involves Three phases.

1) Syntatic analysis :-
↳ compiler must classify the statements and constructs of FORTRAN and extract their par

2) Optimization:
↳ to produce code as efficient as could be produced by an experienced programmer

3) Code Synthesis:
↳ put together the parts of the object code instructions in relocatable format.

Page No.
Date: / /

2.3 Control Structure

→ Control Structures Govern Primitive statements.

Control structs ⇒ those structs had govern the flow of control of the program.

- assignment statement is most important statement in FORTRAN

→ Control structure were based on IBM 704

⇒ As basically designed for IBM 704, FORTRAN control structure were based on IBM 704.

So, FORTRAN has sometimes been called as "assembly language for the IBM 704"

Page No.
Date: / /

	Fortran II Statement	Explanation
	GOTO (10, 20, 30, 40) I	→ Computed GOTO
10	..handle case 1...	⇒ value of I is checked
	GOTO 100	if 1 then goes to 10
20	..handle case 2...	2 " " " 20
	GOTO 100	3 " " " 30
30	...handle case 3...	4 " " " 40
	GOTO 100	
40	..handle case 4	
100	.....	

→ The GOTO is workhorse of Control Flow

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

In FORTRAN, the GOTO-statement is the raw material from which control structure are build.

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

like do while

fig:- sample loop constructed from If & Goto

A.K.A TRAILING-DECISION LOOP

OR,

```
IF (loop done) GOTO 200
  ...body of loop...
  GOTO 100
  ...
```

like repeat until

A.K.A LEADING DECISION ROLE

$100

$'100

$200

$200

$'100

Page No.

Date: / /

→ It is difficult to co-relate static & dynamic structures.

```
100   -- first half of loop....
      IF (loop done) GOTO 200
      ... Second half of loop ...
      GOTO 100
200
```

fig:- mid-decision loop.

THE STRUCTURE PRINCIPLE

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

i.e it must be easy to visualize the behaviour of the program easily from it's written form.

Page No.
Date: / /

→ The computed and assigned GOTOs are easily confused

computed GOTO

GOTO (L1, L2, ... Ln), I
⇒ where Li are statement number and I is integer variable. The computed GOTO transfers to statement number Lk if I contains K

assigned GOTO

GOTO N, (L1, L2, L3 ... Ln)
⇒ where N is also integer variable. This statement transfers to the statement whose address is in the variable N; or it is indirect GOTO i.e 'N' will contains address, the list provides documentation for reader to know where statement goes.

So, confusion b assigned and computed GOTO is more.

SYNTACTIC CONSISTENCY PRINCIPLE
Things that look similar should be similar and things that look different should be different

Page No.
Date: / /

The DO-LOOP IS More Structured than the GOTO

Controlled variable

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

extend / body of loop

- do loop provides a simple method of constructing a counted.

I → is called Controlled variable

→ Do loop is Higher level

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

Higher level constructs such as this are safer to use as they remove from programmer the opportunity to make certain errors. This DO-loop illustrate

The Impossible Error Principle
Making errors impossible to commit is preferable to detecting them after their commission

Page No.
Date: / /

→ The DO-LOOP can be Nested

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

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

(Bad approach)

→ Do loop is highly optimized

Page No.
Date: / /

→ Subprogram were important, late addition

Despite the importance of subprogram, it was only added on FORTRAN II using SUBROUTINE & FUNCTION

→ Subroutine Define Procedural Abstraction

```
SUBROUTINE name (formals)
... body of subroutine ...
RETURN
END
```

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

while invoking

```
:
CALL DIST (DIST1, X, Y)
:
CALL DIST (DIFFER, POSX, POSY)
```

Page No.
Date: / /

→ Sub-programs allow large programs to be modularized.

⇒ By defining a program as a set of relatively small subprograms, the total difficulty of the design process can be decreased since each subprogram can be written, debugged, and read as an independent unit.
A.k.A <u>modularization</u>

So, modularization causes <u>Abstraction</u>

i.e. $ax - bx$
$\Rightarrow (a - b)x$
↖ abstraction

The Abstraction principle
Avoid requiring something to be stated more than once: factor out the recurring pattern

Page No.
Date: / /

→ Subprogram Encourage libraries.

→ Sub program support seperate compilation. Hence, can enchance development of libraries.

→ Parameters are usually passed by reference

FORTRAN ~~has~~ uses Pass by Reference.

→ Pass by Reference is always efficient

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

while invoking

```
DIMENSION VALUES (100)
AVGVAL = AVG (VALUES, 100)
```

Page No.
Date: / /

→ Pass by Reference has Dangerous Consequence

⇒ Main problem of pass by reference is that all parameters can potentially be used for both input and output.

⇒ In pass by reference, input variable may be inadvertently updated by a sub program. This is called <u>side effect</u> of ~~sub~~ sub program call.

Suppose,

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

mem
3

The above subroutine simply writes 3 into it's parameter.

e.g CALL SWITCH (I)
results I = 3

But, suppose we wrote
CALL SWITCH(2)

$2 + 2 = 6$
$\quad\;\; \shortparallel \quad\;\; \shortparallel$
$\quad 3 \quad\;\; 3$

Page No.
Date: / /

Then, the consequence is different As, 2 which is literal is stored in <u>LITERAL TABLE</u>. It's address will be passed. Hence, it will override that value in LITERAL TABLE to 3.
Now, when we call statement like

$$I = 2+2$$

we will obtain $I = 6$. which is major bug.

Hence, this example shows lack of Security in language implementation, since the compiler allows its own runtime data structure (e.g literal table) to be corrupted. So, it violates the security principle

The Security Principle
No program that violates the definition of the language or its own intended structure, should escape detection

Page No.
Date: / /

⇒ Pass by Value-Result is preferable.

Pass-by-Value-Result (copy-restore)
⇒ Value of the actual parameter is copied into the formal parameter at sub-program entry, and the result, or final value of the formal, is copied into the actual parameter at sub program exit.

⇒ Sub program are implemented using activation ~~Results~~ Records.

⇒ Activation Area:
This is the private data area where all the information must be stored in a Caller-private data area before a callee is entered.

There is one activation record for each subprogram.

So, task's to be completed to perform a sub program invocation.

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 callee at it's first instruction.

The steps required to return from the callee to caller are as follows:

1. Get the address at which the caller is to resume execution and transfer to that location.
2. When the caller regains control, it will have to restore the rest of it's execution (registers etc) from its activation record.

Page No.
Date: / /

Data Structures

* Data structures were suggested by Mathematics

- FORTRAN is scientific programming language.
- Data structuring methods provided by all programming languages were also largely determined by the computer architectures and programming techniques of that time.

* The primitive are scalars

- Numeric scalars are the primary data structure primitives provided in FORTRAN

integers — for indexing & counting
floating-point — evaluating mathematical & physical formula

double-precision — floating-point arithmetic with numbers twice as large as the normal floating point numbers

FORTRAN II language included a type DOUBLE PRECISION and arithmetic operations on numbers of this type.

Page No.
Date: / /

Scalar Types are represented in different ways.

- integers were represented as a binary number with a sign bit

S	b30	b29	b28	....	b2	b1	b0

So, number represented by bit pattern is

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

1500
0.15×10^4

Floating point are represented in scientific notation, that is, to the convention of representing a number by a coefficient and a power of 10, for e.g -1.5×10^3

A typical representation for a floating-point number is

s_m	s_c	c_7	..	c_0	m_{21}	m_{20}	....	m_1	m_0

So, value represented by this number is $m \times 2^c$

where 'm' is mantissa

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

note: $m < 1$

2+4.7

Page No.
Date: / /

and c is the characteristics.

$$c = (-1)^{sc} \sum_{i=0}^{7} c_i 2^i$$

A set of data values, together with set of operations on those values that is defined without reference to the representation of data values is called abstract data types

* The arithmetic operators are overloaded

For e.g - same '+' operator is used for different purposes

✓ Integer + integer ⇒ integer addition
✓ Real + real ⇒ floating-point addition
✓ Double precision + double precision ⇒ double floating addition
✓ Complex + complex ⇒ complex addition

if we want to perform ~~at~~ operation

I = X+I

Page No.
Date: / /

where I = integer
X = Floating point

The first integer must be converted to Floating point. This is done by

X + FLOAT(I)

As, the result is saved in integer variable and the sum is floating point. so, it must be converted to integer.
hence, final solution

$$\boxed{I = \text{IFIX}(X + \text{FLOAT}(I))}$$

Similarly

X ** (1/3) to compute cub root of 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 zeroth power.
So, we must write X ** (0.333333) or X ** (1.0/3)

On other hand if we write X ** 2.0 to square X, we will get correct answer, but less efficiently

than if we had written the mixed-mode expression X**2. This is because most compilers compute X**2, the same as X*X, but implement X**2.0 by subroutine calls on the log and antilog functions

* The integer type is overworked.

In FORTRAN the integer type ~~is requ~~ represents both integers and character strings. This results the fact that a Hollerith constant is considered type integer. Hollerith constant was an early form of character string.

E.g 6HCARMEL Represent 'CARMEL'

so, Syntax of Hollerith constant is
1) a literal integer ('6')
2) followed by letter 'H' (for "Hollerith")
3) followed by number of characters specified by the literal integer

Page No.
Date: / /

It causes big problem line

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

so, ~~fort~~ FORTRAN permits us to write
N = ISUCC (6HCARMEL)

so, it is an example of weak typing since FORTRAN allow something of one type (eg Hollerith) to be used where another type (eg integer) is expected solely because they are represented the same way (a one-word bit-string). It is security loophole. It is violation of the Security Principle

* The data Constructor is the Array

In FORTRAN the only data structure constructor is the array.

Page No.
Date: / /

* Arrays are Static and Limited to Three Dimen

Keyword DIMENSION is used to declare array

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

so, DTA is 100-element array (from 1 to 100) and COORD to be a 10×10 array

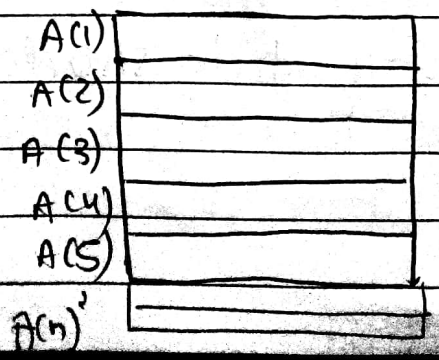
* Array implementation is simple and Efficient

Array are implemented as continuous regions of memory; this permits indexing to be accomplished by adding the index value to the base address of the array.

e.g DIMENSION A(n)

n → represents an integer denotation

The layout of A in memory is.



Page No.
Date: / /

if $\alpha(A(1))$ = Address ~~of~~ in memory of $A(1)$

$\Rightarrow \alpha(A(2)) = \alpha\{A(1)\} + 1$

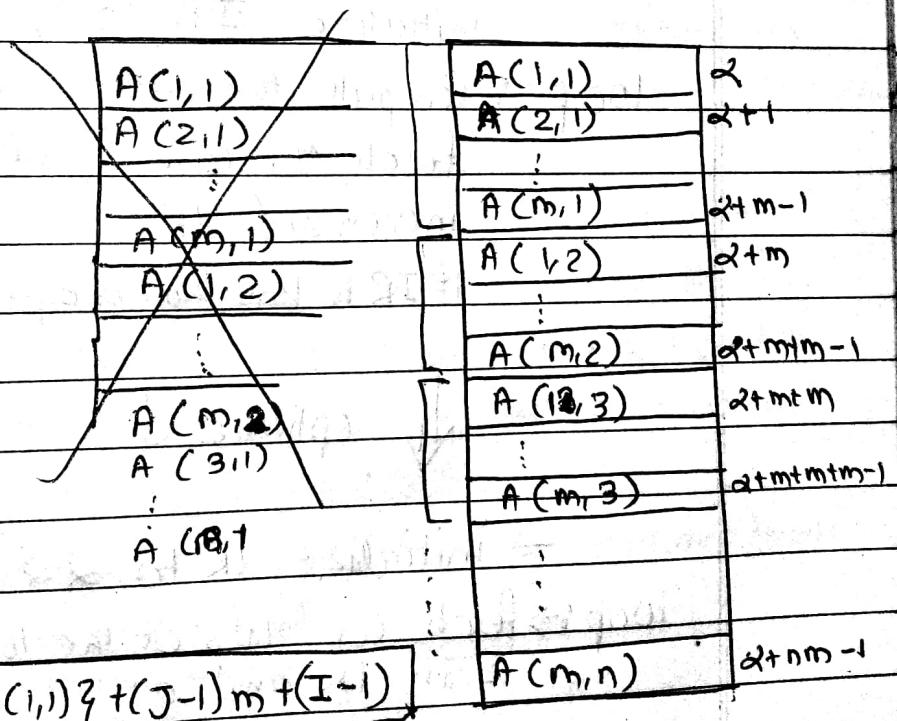
$\boxed{\Rightarrow \alpha(A(n)) = \alpha\{A(n)\} + n - 1}$

address equation for the array A.

~~Dimension~~

DIMENSION A(m,n)

so, FORTRAN arranges arrays in memory in <u>Column-major</u> order i.e columns occupying adjacent memory location.



so

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

address equation where $\alpha = \alpha\{A(1,1)\}$

* FORTRAN Arrays allow many optimizations.

for e.g

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

here, IR (here it is I) is firstly initialized to 1 and then increments.

⇓ this is

Initialize IR to 1;
loop : Compute the address α{A(1)} - 1 + IR;
fetch the contents of this and add to sum;
increment IR
If IR is less than or equal to 100, goto loop

⇓ optimized

→ initialize IR to α{A(1)};
loop :→ fetch 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

Page No.
Date: / /

Name Structures

* The Primitives Bind Names to Object

Name structure – organize the names that appear in a program.

* Declarations are Nonexecutable

- FORTRAN declarations are often called non-executable statements to differentiate them from executable statements like assignment statements, GOTOS, IF-statem and sub-routine calls.
- Non executable statements provide information for the compiler and other preexecution processors of the program (i.e loader & linker). like allocating the amount of storage ~~that must be all~~ for variable Hence called static allocation.
- location of variables are placed in Symbol table

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

Page No.
Date: / /

* Optional Variable Declarations are dangerous

- FORTRAN has automatic declaration of variable which is abondoned in almost all new programming languages.

automatic declaration of variable:-
If a programmer uses a variable but never declares it, the declaration will be automatically supplied by the compiler.

Advantage
- Convinence for programmers since it saved from the effort of declaring all of their variables

Disadvantage
- false economy since it undermines security.

In automatic declaration, the data type (INTEGER, REAL, COMPLEX) are declared with rule "I Through N Rule"

It states that any variables whose name begins with I through N are declared integers and all others are declared reals

so, i, j, k, l, m, n — used for indexes and counts

x, y, z, a, b, c... — are used for other types

Page No.
Date: / /

Side-effect of automatic declaration

what programmer tried to write: COUNT = COUNT + 1
" " wrote : COUNT = COUMT+1

Hence, FORTRAN behaves COUMT as new variable so. and assigns new address. It's value is the junk value stored ~~pre~~ at that location. hence, outcome is unpredictable.
It avoids (SECURITY PRINCIPLE)

* Environment Determine meanings

Suppose, we have statement

X = COUNT (I)

when X can be INTEGER or REAL, similarly COUNT can be array or a function. The context of this statement is set of definitions that is visible to that statement or construct. Since the context of a construct can be though of as surrounding that construct, a context is often called an environment.

gist of this page

variable — local visibility/scope

funx^n name — global visibility/scope

Page No.
Date: / /

* Variables name are locals in scope

- Information hiding principle is used
- Caller of a subprogram should not be able to see the names of any variables declared in that subprogram i.e these names are not visible to the caller, that is they are not in caller's environment.

* Subprogram Names are Global in scope

- Subprogram names are visible throughout the program; thus are said to be global.

```
C       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
```

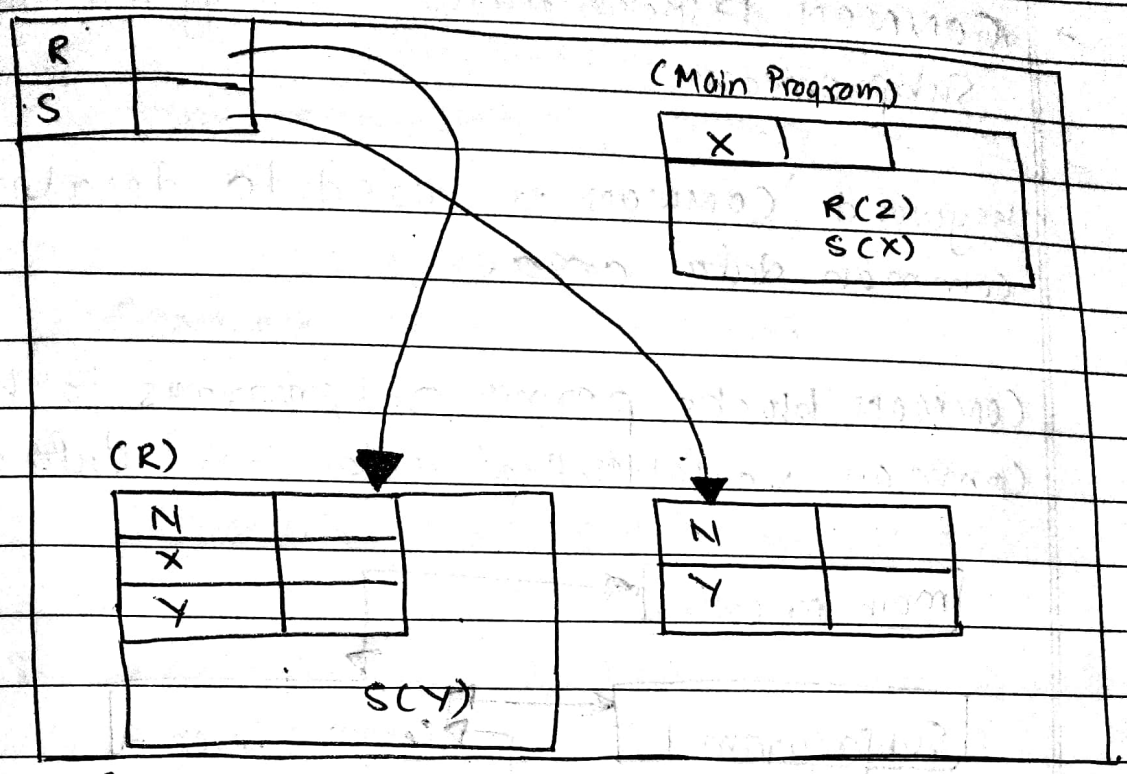


fig:- Contour Diagram for showing Scope of Variables

* It is difficult to Share Data with just Parameter

e.g CALL ARRAY2 (NM, AVAIL, INTCOD, M, N, NAMES, LOC, ...)

↳ it is unreadble

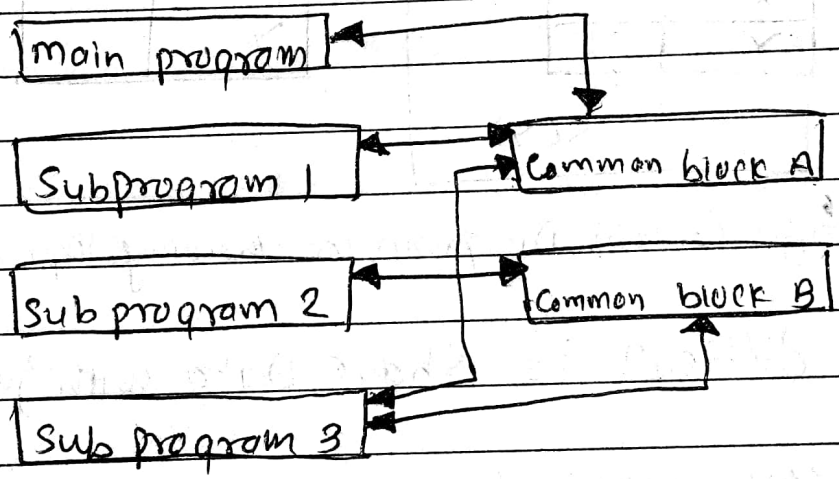
↳ hard to maintain and unrealistic

Page No.
Date: / /

* COMMON Blocks allow sharing between Subprogram

keyword 'COMMON' is used to declare common data area.

COMMON blocks permit subprograms to have common variables, that is to share data areas.




```
MAIN PROGRAM
  :
CALL ARRAY2 (NM, AVAIL, INTCOD, M, N)
  :
END
```

```
SUBROUTINE ARRAY2 (N, L, C, D1, D2)
COMMON /SYMTAB/NAMES(100), LOC(100), TYPE(100), DIMS(100)
  :
END
```

```
SUBROUTINE VAR (N, L, C)
COMMON /SYMTAB/NAMES(100), LOC(100), TYPE(100), DIMS(100)
  :
END
```

Page No.
Date: / /

COMMON permits Aliasing which is dangerous.

Consider example

in 1st sub-program
COMMON /B/ M, A(100)

in 2nd sub-program
COMMON /B/ X, K, C(50), D(50)

so, from above we can see correspondance as

in 1st sub-program		in 2nd sub-program
M	→	X
A(1)	→	K
A(2)	→	C(1)
⋮		
A(51)	→	C(50)
A(52)	→	D(1)
⋮		
A(100)	→	D(49)
		D(50) ≡ new memory only declared in 2nd sub-program

Hence, problem here is M corresponds X, but their data types don't match. User can put any data.

Page No.
Date: / /

Similarly, A(2) → C(1) & A(51) → D(1)
so, there is breaking of Array A.
Finally D(50) is not even accessible from the first subprogram.

Hence, same memory area is assigned different names a.k.a aliasing. This ~~can cause~~ makes programs difficult to understand and difficult to maintain.

* EQUIVALENCE allows sharing within subprogram

During the time when FORTRAN was developed memory was big issue. So, memory conservation was big deal.

If a program is finished with the first array before it needs to begin using the second array, then memory will be better utilized if the two. This can be done in FORTRAN with EQUIVALENCE declaration such as

```
DIMENSION INDATA(10000), RESULT(8000)
EQUIVALENCE (INDATA(1), RESULT(1))
```

Page No.
Date: / /

This states that the first element of INDATA is to occupy the same location as first element of RESULT. i.e we have explicitely aliased the two arrays so that they will share storage.

The use of 'EQUIVALENCE' keyword has outlived as

i) Computer memories are much larger
ii) Modern Computers have virtual memory System that automatically and safely manage the sharing of real memory.
iii) Modern Computers support dynamic storage management that also automatically and safely manage the allocation of memory.

Page No.
Date: / /

SYNTACTIC STRUCTURES

* Languages are defined by Lexics and Syntax

[The syntax of a language is the way that words and symbols are combined to form the statements and expressions]

[The lexics of a language is the way in which characters (i.e letters, digits and other signs) are combined to form words and symbols.]

In FORTRAN an identifier must begin with a letter and be followed by no more than five letters and digits.

The syntatic analysis phase of a compiler is broken down into two parts.

- lexical analyzer a.k.a scanner
- syntactic analyzer or parser a.k.a parser

Page No.
Date: / /

A fixed format lexics was inherited from the Pseudo-Codes

FORTRAN's one statement per card and columns dedicated to particular purposes:

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

* Ignoringing Blanks Everywhere is a mistake

In Fortran
D I M E N S I O N DATA (10, 10)
is <u>valid</u> which is
DIMENSION DATA(10,10)

This can cause problem (sometimes), like

DO 20 I = 1. 100 (actually it is DO20I = 1.100)
is mistaken for DO 20 I = 1, 100

Page No.
Date: / /

* The Lack of Reserved Word is mistake

FORTRAN permits programmer to have an array called IF.

so, IF(I-3) = 1 2 3 — assignment statement
is confused with
IF (I-1) 1, 2, 3 - if statement

Similarly
DO 20 I = A(I,J) — assignment statement
is confused with
DO 20 I = I, J — Do loop

* Algebraic Notation was an important contribution

FORTRAN ~~made~~ allowed programmers to use a conventional algebraic notation.

e.g $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$

$\Downarrow$

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

Page No.
Date: / /

Arithmetic Operators have Precedence

as, we use BODMAS rule for operator priority.

FORTRAN follows following precedences among arithmetic operators:

1. Exponentiation
2. Multiplication & Division
3. Addition & Subtraction.

* Linear Syntactic Organization is Used.

Statements are strung together one after another; just like the instructions in memory; they are addressed with numeric labels. This is called <u>linear syntactic organization</u>.

i.e

ALGOL-60

Generality & hierarchy

1950s — the need of single, universal, machine-independent programming language was felt; because
- it was difficult to transport programs from one machine to another. as each machine has its own instruction sets, assemblers & pseudo-code interpreters.

So, in 1955, GAMM (a European Association for applied Mathematic and Mechanics) set up committe to study the development of such language

The language known as Algol-58 was designed by eight representative who met in Zurich (four each members from ACM (The association of Computing Machinery) in May & June in 1958).
John Backus was also member of the ACM.

So, ~~ae~~ IAL (International Algebraic Language) was was released. Later it was called Algol-58 (Algorithmic Language)

Page No.
Date: / /

The definition ~~of~~ of Algol-58 made by John Backus and Peter Naur ~~did not~~ was not same. Hence. So, more precise method of describing syntax was required and prepared some samples of a variant of the Backus notation. As a result, this notation was adopted for the Algol-60 report and is now known as BNF (Backus-Naur-Form).

So, in May 1960 a different language from Algol-58 was described, by taking suggestions that had been received. Thirteen members of ACM and GAMM met in Paris for 6 days in January 1960 to for Algol-60

- A few remaining errors and ambiguities were corrected at a meeting in Rome 1962.

- ~~Repo~~ Revised Report of Algol-60 was published in 1963 - Jan. (15 page long repat - shortest report at that time)

Page No.
Date: / /

DESIGN

* Algol programs are hierarchically structured

- Main contribution of Algol is its use of hierarchical structure, or nesting, throughout it's design.

```
begin
    integer N;
    Read Int(N);
    begin
        real array Data[1:N];
        real sum, avg;
        integer i
        sum:=0;
        for i:=1 step 1 until N do
        begin real val;
            Read Real (val);
            Data [i]:= if val<0 then -val else va
        end;
        for i:=1 step 1 until N do
        sum:= sum + Data[i];
        avg:= sum/N;
        Print Real (avg).
    end
end
```

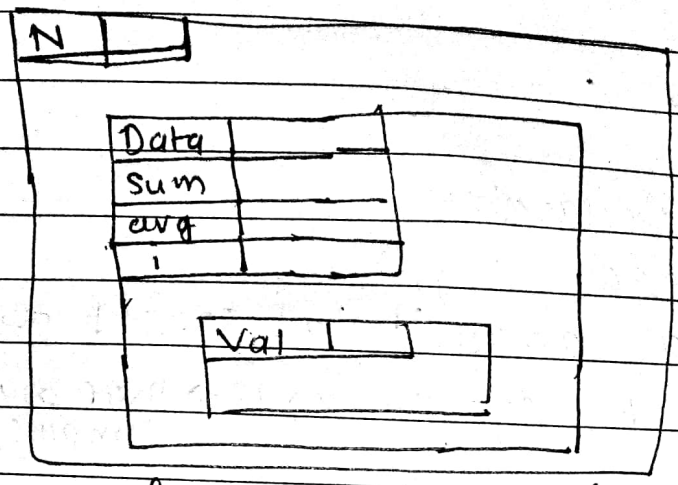


fig:- Contour Diagram for above program

* Constructs are either Declorative or Imperative

As in FORTRAN, the construct of Algol-60 can be divided in two categories

declarative
↳ binds name to objects
↳ are of 3 types

Imperative
↳ perform computation

Variable declaration
↳ Same as FORTRAN except the only data type allowed are integer, real and Boolean

procedure declaration
typed procedure → function
untyped procedure → sub-routines

Switch-declaration
↓
same as FORTRAN's computed GOTO

Only data type allowed < integer, real, boolean

Page No.
Date: / /

* ~~Imperatives~~ .

- Variable declarations
  e.g integer i, j, k
  real array Data[-50:50] ⇒ array declaration
- (Algol has dynamic arrays → their bounds can be computed at run time)

- Procedure declarations
  e.g
  real procedure dist(x1, y1, x2, y2);
  real x1, x2, y1, y2;
  dist := sqrt((x1-x2)↑2 + (y1-y2)↑2)

* Imperatives are computational and control-flow

imperative
↓
computational — control flow

↳ assignment is only computational statement
↳ used symbol ':='

Page No.
Date: / /

Algol provides arithmetic operators (+, - etc) Relational operators which returns boolean (=, >, ≤ etc.) and Boolean operators (∧, ∨, ¬, etc.)

- I/O construct was not taken as ~~Com~~ it was intended that I/O be handled by library.

* Algol has Familiar Control Structures

Algol uses well known control structure, like goto-statement ; if-then-else ., for-loop (elaboration of FORTRAN'S DO-loop)

* The Compile-Time, Run-Time Distinction is important-

- Algol program has to go through compilation process.
- Algol data structures have a later binding time than FORTRAN data structures.

Due to dynamic arrays and recursive procedures

* The stack is the central run-time data structure.

- For organizing the dynamic allocation and deallocation of memory Algol uses stack. They use it for holding activation records for procedures and blocks. Dynamic allocation and deallocation are achieved by pushing and popping these activation records on the stack.

NAME STRUCTURES

* The Primitives Bind Names to Objects

In FORTRAN a variable name is statically bound to a memory location, where as in Algol we will find a single variable may be bound to a number of different memory locations and these bindings can change during run-time.

Page No.:
Date: / /

* The constructor is the block

Algol-58 introduced idea of a compound statement. This allows a sequence of statements to be used wherever one statement is permitted.

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

'begin-end' keywords can be used to surround statements.

for loop में एक से अधिक statement होने पर उन्हें multiple statement block statement कहते हैं

```
for i := 1 step 1 until N do
begin
    if Data[i] > 10000 then Data[i] := 10000;
    Sum := Sum + Data[i];
    Print Real (sum).
end
```

block

The body of procedure is taken to be a single statement.

```
real procedure cosh (x); real x;
        cosh := (exp(x) + exp(-x))/2;
```

Page No.
Date: / /

* Blocks define Nested Scopes

- The defect of COMMON block in FORTRAN was that, they must be redeclared in each subprogram. So, Algol-60 avoids this redeclaration by allowing the programmer to define any number of scopes nested to any depth; This is accomplished with a block

begin declarations; statements end

block	Compound Statement
→ contains declarations	→ no declaration.

* Blocks simplify constructing Large programs.

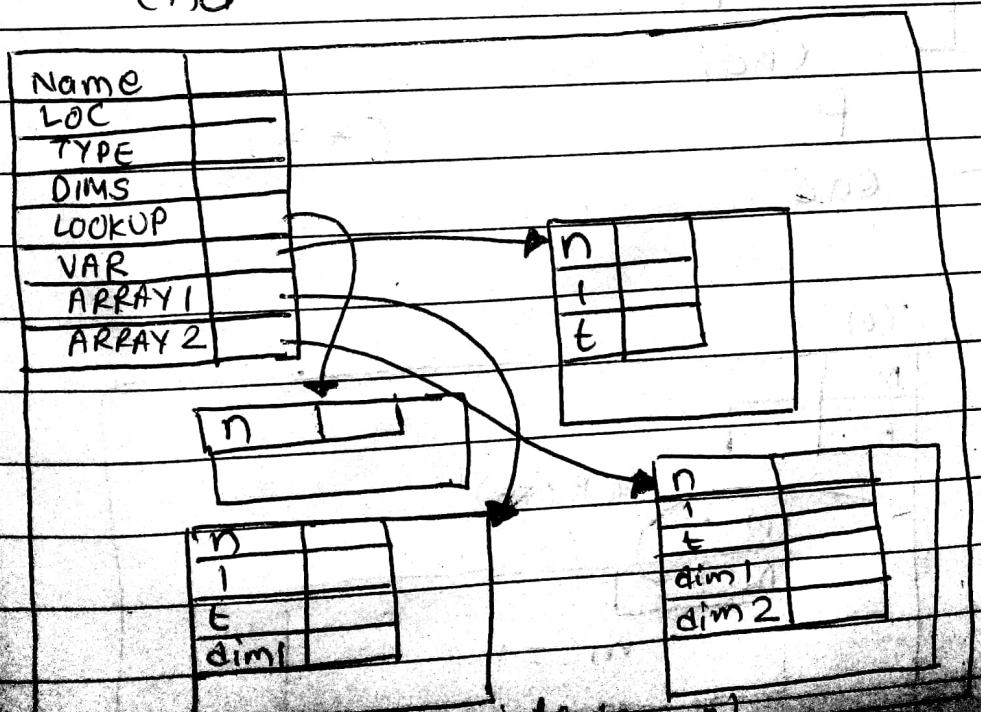
The mechanism used in FORTRAN to share variable was COMMON declaration. So, Algol found a better way to solve this problem by breaking program into blocks. So, a common factor can be factor

Out as a block:

```
begin
    integer array Name, Loc, Type, Dims[1:100];
    procedure Lookup(n);
        ... Lookup procedure....
    procedure Var(n, l, t);
        .... Enter variable procedure ....
    procedure Array1(n, l, t, dim1);
        ... Enter 1-dimensional Array procedure....
        . . . .

        Array2 (nm, avail, intcode, m, n);
        . . . .

end
```



~~ARRAY2(nm,avail,intcode,m,n)~~

Page No.
Date: / /

* Dynamic Scoping allows the context to Vary

2 Scoping rules
- Static scoping
  - a procedure is called in environment of its definition
  e.g ~~LISP~~ Algol
- dynamic scoping
  - a procedure is called in the environment of its caller
  e.g LISP

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

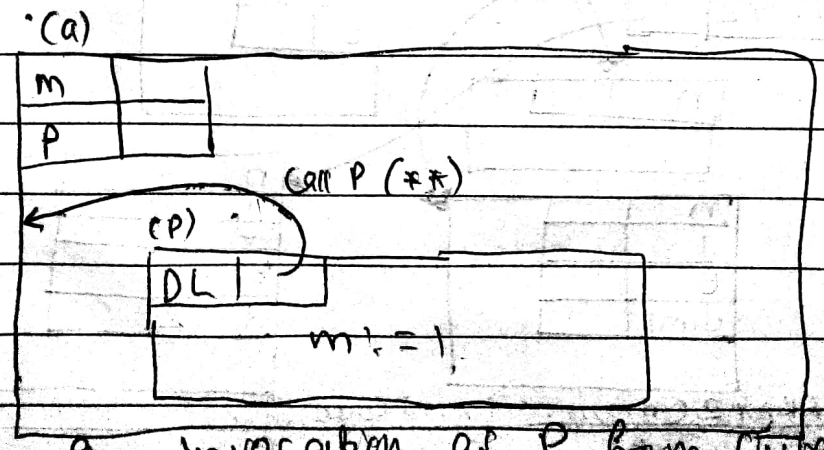


fig:- Invocation of P from Outer block

with <u>dynamic scoping</u> the assignment m:=1 refers to the outer declaration of m when P is called from outer block (**) and the inner declaration of m when P is called from the inner block (*)

DL → Dynamic Link
= a pointer from the <u>callee</u> to the <u>caller</u>.

Since P is called in the environment of its caller, block (a) (*,*) the contour for P is nested in the contour of block (a). Hence, m:=1 refers to the m declared in block (a)

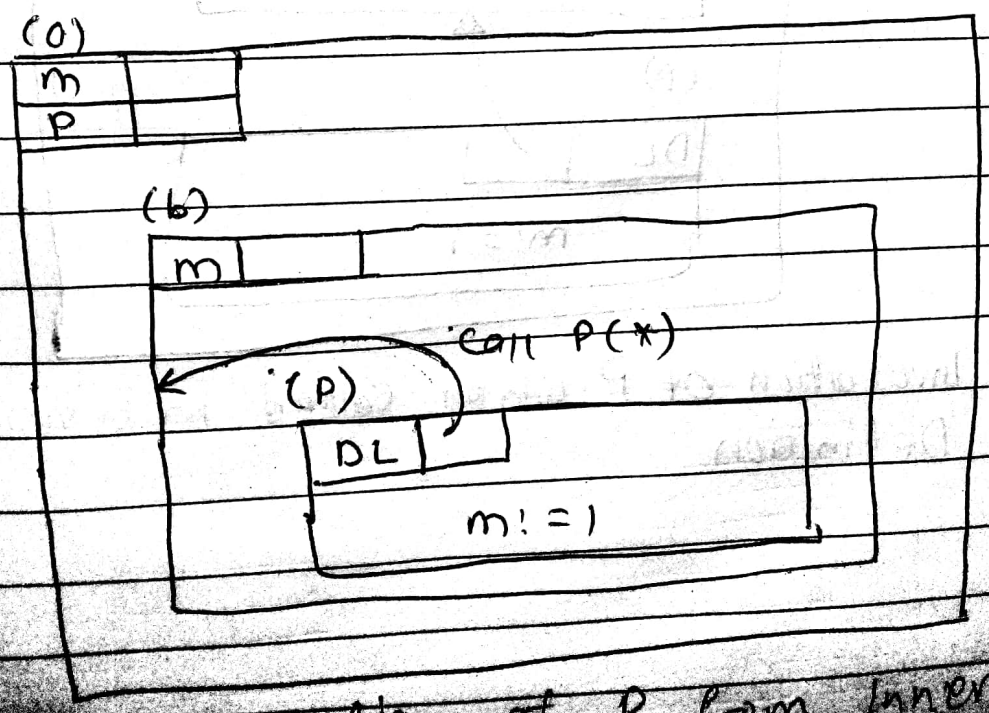


Fig:- Invocation of P from Inner Block

~~When the call was assumed~~

If we assume the call P(*) was static scoping the assignment m:=1 always refers to the variable m in outer block. This is so because P is called in the environment of its definition i.e the context in which P is executed is always in which it was originally defined

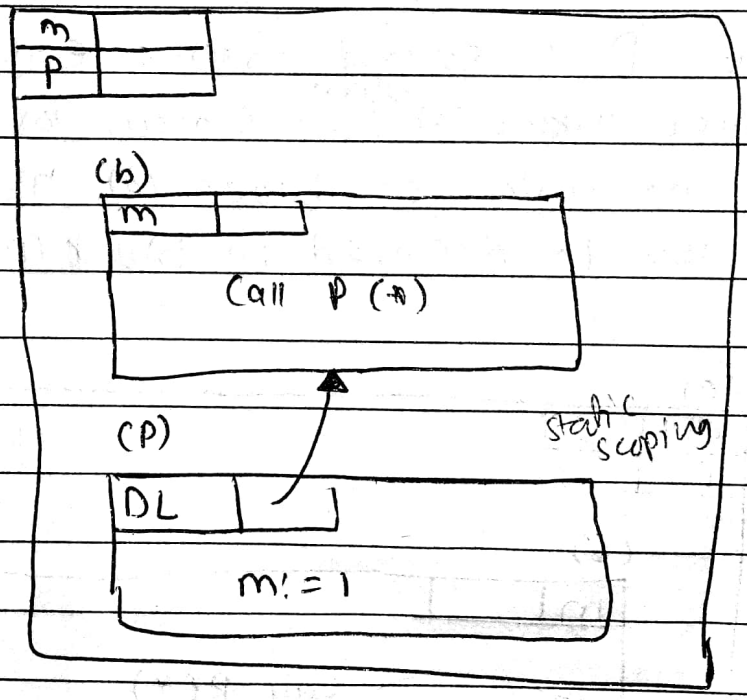


fig:- Invocation of P when called in Environment of Definition

Page No.
Date: / /

so, in Conclusion:-

Dynamic Scoping:-
Scopes of names are determined dynamically that is, at run-time. So in such language the meanings of statements and expressions may vary at run-time.

Static scoping
scopes of names are determined statically by the structure of the program so meanings of statements and expressions are fixed. i.e meanings of all statements and expressions can be determined by inspecting the static structure of the program without having to understand its dynamic behaviour.

Page No.
Date: / /

Static scoping Aids Reliable Programming

Static scoping causes the static structure of a program to agree more closely with its dynamic behaviour than does dynamic scoping. i.e Structural Principle.

Structural Principle:
~~Static scope cause the static structure of a program to agree.~~
Dynamic behaviour of a program should correspond in simple way to its static structure.

Block permits Efficient Storage management on a stack

Consider following example

```
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
```

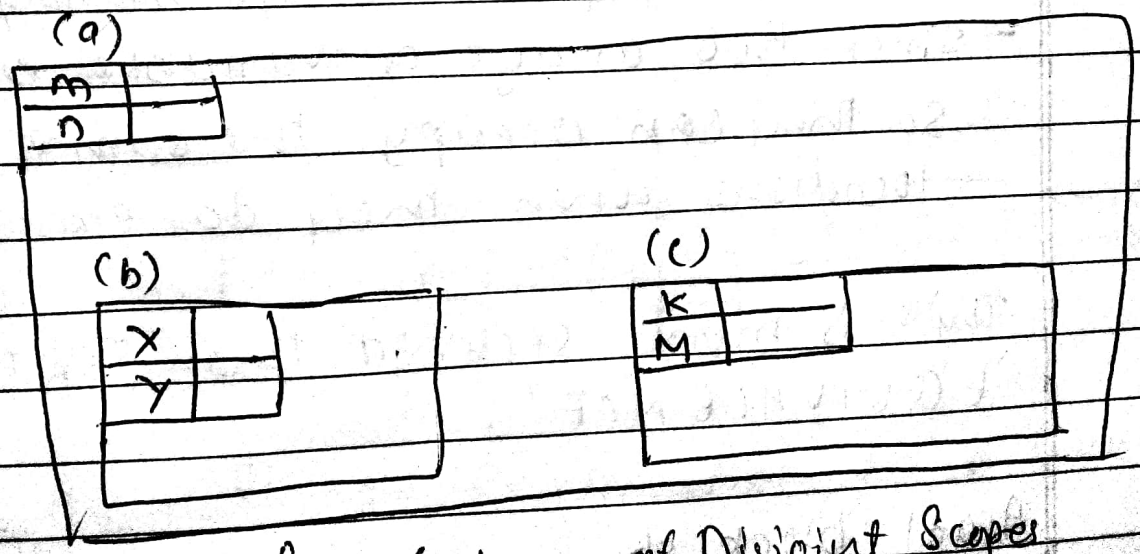


fig:- Contours of Disjoint Scopes

Date: / /

Two blocks (b) & (c) are disjoint that is neither is nested in the other. That means whenever program is executing in (b) local variable of (c), are not visible and vice versa.

So, a value of variable is retained only so long as the program is executing in the scope of the variable or in a block or a procedure that will eventually return to the scope of that variable i.e if flow of control leaves the scope of a variable, then the contents of variable is discarded.

This can help in storage management
- Since, two arrays or variable coexist, so they can occupy the same storage locations when they do exist.

This is more secured than FORTRAN's 'EQUIVALENCE'.

Algol block structure permits memory to be used for multiple purposes. Algol blocks obey a last-in, first-out discipline. i.e bl

that was last entered is the first to be exited, and block that was first entered is last to be exited. This enhance structured hierarchy

Stacks are allocated in following way.

- When a block is entered, an activation record for that block is pushed onto the top of run-time stack. This activation record contains space for all the variables local to the block.

- Conversely, when a block is exited, its activation record will be popped off of the stack, thus freeing its storage for use by other blocks.

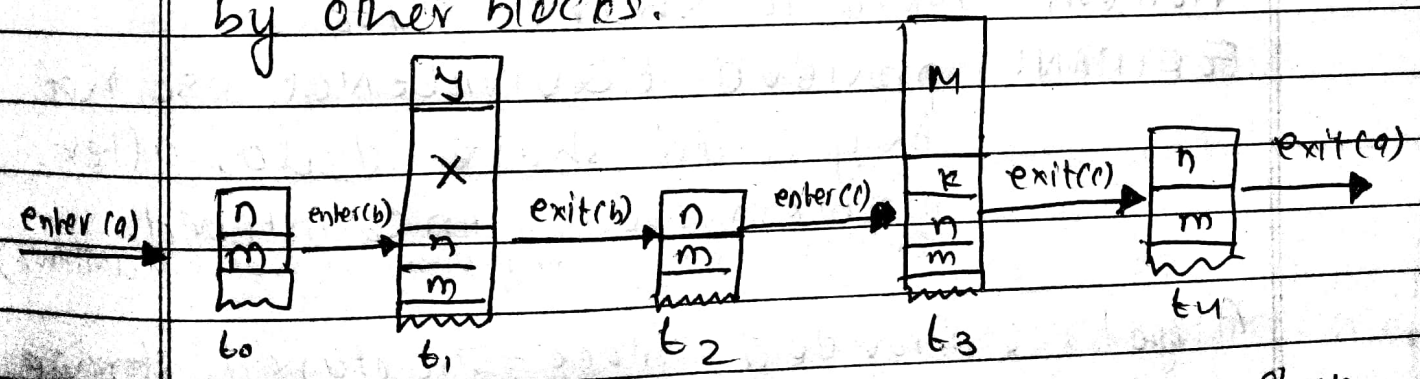


Fig: - Storage Reallocation on a Stack

[Array X & M share the same memory locations.]

* Responsible Design Entails Understanding User's problems.

"The language designer must not ask 'what do you want?', but rather 'how does your problem arise?'"

- Niklaus Wirth
  - designer of Pascal

Responsible Design Principle
Do not ask user what they want; find out what they need.

Problem: Need to share data
FORTRAN: provided EQUIVALENCE so that they could share data, after problem was encountered.
(initiation done after problem was encountered)

Algol : Provided block-structured storage allocation so solve this problem
(provided solution before problem encountered)

Page No.
Date: / /

Data Structures

* The primitives are mathematical scalars

primitive data types — integer, real, boolean } mathematical scalars

double-precision — was not included as their use is necessarily machine dependent.

Q) why is double-precision machine dependent?
⇒ Suppose if we want 10 digits of accuracy in a computation. To decide whether to use single or double precision, we would have to know the word size and floating point representation used on our computer

So, this supports Portability Principle

Portability Principles
Avoids features or facilities that are dependent on a particular computer or a small class of computer

* Complex number - not included in Algol as they are not primitive, but was included in FORTRAN

treated as second class citizen in ...

- Strings are allowed in only a few limited contexts.
  - String denotations are only allowed as actual parameters, and the string type is only allowed in the specification of formal parameters. *
  - There are no operators, relations, coecions, or variable declarations for strings.
  - String can be passed as arguments to procedures

V.V. Imp

* Algol follows the zero-one-Infinity Principle

In FORTRAN IV

max character in an identifier name = 6
" continuation cards = 19
" dimension of array = 3

hence, FORTRAN programmer must remember the number 6, 19, 3 and so on. (difficult to remem...

And, why only 6 chars. in identifer ?
=> No one know actual answer (may be due to impact of IBM 704).

but longer is the no. of char available, it is more easy to remember them.

The <u>zero-One-Infinity</u> Principle says that the number of characters allowed in a name should be zero, one, or infinity. It means that there must not be a limit on the length of an identifier.
[It is better to have limit due to memory capacity]

So, Zero-One-Infinity Principle ~~say~~ means that any limit in a programming language design should be none, one or infinite.

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

Date: / /

Arrays are Generalized

- Algol array can be of any dimension. To follow Zero-One-Infinity Principle.
  [In FORTRAN IV ~~only~~ array could have 3 dim

Algol also generalizes FORTRAN array by allowing lower bounds to be numbers other than 1.

Comparing arrays in FORTRAN & Algol.

Language	Starts from index	ends in index	Total size of array
FORTRAN	1	any number(n)	n (A[1] ... A[n])
Algol	any number(m)	" "	m-n+1 (A[m] ... A[n]

e.g.
integer array Number of days [-100:200]

Page No.
Date: / /

Stack Allocation Permits Dynamic Arrays

Algol permits dynamic array i.e the size of array is determined at run time. This can help to save memory.

e.g real array A[1:N];
— a.k.a flexible arrays

Every time a block is entered, an activation record allocation in a block-structured language,

Algol arrays are dynamic in a limited fashion; the array's dimensions can be computed each time the block to which it is entered, but once the array is allocated, its bounds remain fixed until its scope is exited.

- Every time a block is entered in stack, an activation record for that block, containing all of the block's local storage, is pushed onto the run-time stack, when the block is exited, this activation record is deleted. Hence, it's size is known at block entry time, an appropriate size activation record can be allocated.

Page No.
Date: / /

* Algol has Strong Typing

Strong Typing:-
Type abstractions of the language are enforced.

Advantage of Strong Typing:-
* A strong type system prevents the programmer from performing meaningless operations on data.
eg Programmer can't do an integer addition on floating-point numbers, or a floating point multiply on Boolean value.

FORTRAN did not support strong typing.

Disadvantage of weak typing:-
* In weak typing variables of different types are aliased i.e share same memory location. This is security problem when done accidently and maintenance problem when done intentionally.

But Algol supports coercions (type conversion) from integers to real, and v.v with an operat

Page No.
Date: / /

CONTROL STRUCTURES

- directs & manage the flow of control from one ~~statement~~ assignment statement to another.

~~operation's~~

* Primitive Computational statements have changed little

Algol has changed some computational statements, like

Operation's	in FORTRAN	in Algol
→ Assignment	=	:=
→ I/O operation	READ WRITE	by library procedures

I/O operation was performed by library procedures as Algol belives I/O is not assignment, so not a control structure.

Page No.
Date: / /

* Control structure's are generalization's of FORTRAN's

Algol has provided essentially the same structures in a generalized and regularized form. For eg -

IF (logical expression) simple statement

- but Algol added else part also, so,

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

→ if... else statement

Algol for loop is more general than FORTRAN's DO loop. It includes function of simple DO

```
for i := 1 step 2 until N×M do
    inner[i] := outer[N×N-i];
```

variation of FORTRAN's DO. loop

Page No.
Date: / /

It also provides variants that is similar to while-loops

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

variation of while loops

and also

```
NewGuess := Improve(OldGuess);
while abs(NewGuess - OldGuess) > 0.0001 do
    begin
        OldGuess := NewGuess;
        NewGuess := Improve(OldGuess)
    end;
```

Hence, Algol control structure are more regular, more symmetric, more powerful and more general than FORTRAN's.

Page No.
Date: / /

* Nested statements are very important

Suppose in FORTRAN

```
      IF (X. LE. Y)  GOTO 100
        X = X/2
        Y = Y + DELTA
100   .....
```

So, when we want to execute multiple statements when a IF condition is met we have to use 'GOTO' to locate our code. Here code doesn't flow throughout as as expected. So, programmer must think his logic in otherwise form.

Hence, Algol begin .... end help to provide blocks. It acts as brackets to distinguish nested scopes and group statements into compound statements.

Compound statements are hierarchial structures.

Algol's compound statement is a good example of hierarchial structure. Simple statements such as assignment, procedure invocation, and the goto-statement, complex structures are built up by hierarchically combining these statements into larger and larger compound statements. Hierarchial structure is important principles of language and program design.

* Nesting led to structured programming

```
      IF (.NOT. (condition)) GOTO 100
        statement 1
        :
        statement m
        GOTO 200
100     statement 1
        :
200     statement n
```

FORTRAN'S way
→ must use otherwise logic

⇒

```
If condition then
begin
    statement 1;
    :
    statement m
end
else
begin
    statement 1;
    :
    statement n
end
```

Algol's way
→ use simple logic

Page No.
Date: / /

- Algol helped to reduce no. of GOTO statements (Edsger Dijkstra proposed GOTO statement must be eliminated in High level language)
- Algol programs were easier to read

Hence, GOTO-less programming was popular. Ultimately this led to loose body of programming methods and techniques called Structured Programming.

* Procedures are Recursive

Algol permits recursive procedures
For e.g (for calculating factorial)

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

$\Downarrow$

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

Conditional expression

Locals may have several instantiations

Consider above procedure to calculate factorial.
suppose we called fac(3), then in n we will have 3, now fac(3) invokes fac(2), which will store 2 in n. hence, overriding n:=3 to n:=2 in this case will cause massive damge.
So, In Algol

Each invocation of 'fac' must have its own <u>instance</u> of the formal parameter n, which will contain actual parameter to that record for 'fac' each time it is called.

So, in general, each invocation of procedure results in the <u>instantiation</u> (i.e creation) of a new <u>activation</u> record for the procedure.

Page No.
Date: / /

~~value of actual parameters~~
value copied into a variable corresponding to the formal parameter

* Parameters can be Passed by Value

```
procedure switch (n);
    value n; integer n;
    n := 3;
```

This above program is harmless in Algol (unlike FORTRAN). An invocation switch (2) will cause the value 2 to be copied into the local storage for the formal n. The assignment n := 3 merely alters the value of this location; it has no effect on the actual, 2, or does not endanger the literal table. This means invoking switch (k) will have no effect on value 'k'. so, value parameters are only used for input purpose.

So, Advantage of Pass by Value
- It is more secured

Page No.
Date: / /

* Pass by Value is inefficient for Arrays:

Consider following sample code to average values of arrays

```
real procedure avg (A, n);
    value A, n;
    real array A;
    integer n;
    begin
      :
    end;
```

So, pass by value is inefficient for Array as

⇒ So, when procedure avg is called, an activation record will be created containing space for values of the actual parameter, but only 'n' space was required. So, waste of memory

⇒ As entire array has to be copied into activation record. It is waste of time

So, pass by value is not satisfactory way to pass array.

Page No.
Date: / /

* Pass by Name is based on Substitution

suppose we have code

```
procedure Inc (n);
  Integer n;
  n := n+1;
```

This is "pass by name" in Algol, as this is mode we get if we do not specify pass by value. (i.e 'value' keyword is absent)

Hence, if ~~w~~ above 'Inc' procedure was pass by value invoking Inc(i) will cause 'n' to copy value of 'i', increment it and finally, no impact is seen in i. i.e Only local copy gets changed.

But as Inc is 'pass by Name', invoking Inc(i) will actually alter i, because the body of Inc will be modified as

i := i+1;

where 'i' is global variable. Hence it's effect will be seen when procedure exits also.

Page No.
Date: / /

Ex

~~Inc [A~~ Inc(A[i]) ⇒ A[i] := A[i]+1 *

So, pass by name satisfies Algol's need for output parameters. So, in Pass by name parameter acts as though they were substituted for the corresponding formals.

* Pass by Name is Powerful

Consider,
```
SUBROUTINE S(EL,K)
K=2
EL=0
RETURN
END
```
and, program segment
```
A(1) = A(2) = 1
I = 1
CALL S (A(I), I)
```
Above is a segment of FORTRAN's code. So, calling sub-routine, we get pass by reference since I has the value 1 at the time of call, the reference passed for A(I) will be address of A(1). The subroutine S has two effects. First it assigns K=2, which assigns 2 to I since the formal K is bound to the address of I. Second, it assigns EL=0, which assigns 0 to A(1). Since

Page No.
Date: / /

The formal EL is bound to the address A(1). The fact that in the mean time I has been changed to 2 has no effect on EL since the reference of A(I) was computed at call time. When S exits I will have the value 2 and A will have the value (0,1).

Consider, Algol ~~p~~ code

```
procedure S (el, k);
  integer el, k;
  begin
      k := 2;
      el := 0
    end;
A[1] := A[2] := 1;
i := 1;
S (A[i], i);
```

Since parameter are passed by name the effect of S will be as though the actuals were substituted for the formals. That is, the effect of S(A[i], i) will be

Page No.
Date: / /

i := 2;
A[i] := 0;

Thus S will assign 2 to i and 0 to A[2], so when S exits i will have the value 2 and A will have the value (1,0). Hence, Algol results is different from FORTRAN

Page No.
Date: / /

* Pass by name is dangerous & expensive

Consider following code

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

Now, let's perform ~~A[i], i) and A[~~ swap (A[i], i) and ~~A~~ swap (i, A[i]). As, we intend to change value of i and A[i]

for swap(A[i], i)	for swap(i, A[i])
begin integer t;	begin integer t;
t := A[i];	t = i;
A[i] := i;	i = A[i];
i := t.	A[i] = t. — ⊛
end;	end;
- output is as expected	- output is not as expected as at ⊛ value of i has already change we store at different

So, if i has 1 & A[i] has 27 then

~~Swap (~~ swap (i, A[i])

Then effect will be to assign 27 to i and ~~A[27] to~~ 1 to A[27], but we expected A[i] to be 1 & i to be 27.

[Note:- There is no way to define a Swap procedure in Algol-60 that works for all actual parameters.]

This happens because, in Algol, parameters are reevaluated every time they are used and, hence may refer to different locations on each use.

* Pass by Value:- At time of call, the formal is bound to the value of the actual. Since value parameter takes a snapshot of the actual parameter, later change in actuals value will not be seen by callee.

* Pass by Reference:- At time of call, the formal is bound to a reference to the actual. The reference cannot vary thereafter, although the value stored at reference can

* Pass by name - At the time of call, the formal is bound to a thunk for the actual. Although this cannot vary, each time it is evaluated it may return a different reference and consequently a different value

Page No.
Date: / /

Good Conceptual models help users.

A Conceptual model is a mental representation of the way something works, it helps us to predict the effects of our actions when dealing with that thing. A good conceptual model allow us to predict the effects of our actions.

Norman distinguishes three conceptual models

1) Designer's model
- it includes system's construction, which for a programming language feature includes its implementation, ~~formal syntax etc.~~

2) System image
- Created by designer and is the basis on which user's image is formed. It may present various displays and controls or even expose its mechanism to the user's view. It includes manuals, diagrams, explanations, instruction etc provided with system.

3) User's ~~sima~~ model
- Users form model on basis of the systems image.

TOS - top of stack

Page No.
Date: / /

Out-of-Block gotos can be expensive

Algol has one scope rule for all identifiers. Consider following program skeleton:

```
a: begin array X[1:100];
      :
   b: begin array Y[1:100];
         :
         goto exit;
         :
         end;
exit:    :
      end
```

Suppose block 'b' is executed and finished without executing 'goto' statement, then from stack activation record of block 'b' is deleted and block 'a' becomes top of stack. But if 'goto' in block b is executed, same thing will occour (deleting activation record of block 'b' & block 'a' as TOS.) but it is much more complicated than simple machine jump as it involves exiting through several levels of block structure.

```
begin
procedure P(n);
  value n; integer n;
  if n=0 then goto out
  else P(n-1);
P(25);
out:
end:
```

So, above program causes P to be called recursively 25 times at which point it will jump to label out. Hence, number of recursion and hence number of activation records to be discarded, are data dependent. The compiler cannot know beforehand how many activation records to delete. So, it is expensive process.

* Feature Interaction Is a Difficult Design Problem.

Feature Interaction is one of the most difficult problems in language design. If new feature is added, its almost all interaction must be considered, which is almost difficult.

* The for loop is ~~ge~~ very general

Algol for-loop is generalization of FORTRAN'S DO-loop.

e.g

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

* for loop generate sequence of values to be assigned successively to the controlled variable.

e.g 1 step 1 until 5
⇒ will generate sequence 1, 2, 3, 4, 5

also, i/2 while i ≥ 1
⇒ will generate 8, 4, 2, 1 if i = 16

* values to be used in for-list can be listed explicitely

e.g for days := 31, 28, 30, 30, 31, 30,
31, 31, 30, 31, 30, 31, do .....

* Algol permits conditional expression to be used

for days := 31,
if mod(year, 4) = 0 then 29 else 28,
31, 30, 31, 30, 31, 31, 30, 31, 30, 31
do ....

* The for loop is Baroque

Baroque
- features of extreme generality and doubtful utility

Algol's for loop is highly generalized,
- It provides surplus of features, i.e unwanted features of questionable usefulness.

* The switch is for handeling cases

Algol generalized FORTRAN's computed GO
. The construct is called a switch-declaration and amounts to an array of statement labels
E.g

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