

LISP (List Processing and functional programming)

(1) What is LISP ? Explain the structural organization of Lisp with a suitable example.

Ans: Lisp is the second - oldest high-level programming language in the world which is invented by John McCarthy in the year 1958 at MIT.

→ Lisp has an overall style that is organized around expressions and functions.

→ Features of LISP :

- (1) It is a machine-independent language;
- (2) It supports all kind of data types like objects, structures, lists, vectors, sets, trees and hash-tables.
- (3) It is expression based language
- (4) It supports different decision making statements like if, when, case and cond, and iterating statements like do, loop, loopfor, dotimes and dolist.

Structural organization of LISP:

① Function Application is the central idea :

In LISP, function application is the central idea , that is, applying a function to its arguments .

function, ^{application} in Lisp is written as:

$(f\ a_1, a_2, \dots, a_n)$

where f is the function and a_1, a_2, \dots, a_n are the arguments.
This notation is called Cambridge Polish because it is a particular variety of Polish notation developed at M.I.T (in Cambridge).

- The distinctive characteristic of Polish notation is that it writes an operator before its operands. This is also sometimes called prefix notation.

Example: $(\text{plus}\ 2\ 3)$

Many constructs that have a special syntax in conventional languages are just function application in Lisp. For example, the conditional expression is written as an application of `cond` function. That is:

C. `cond`

$((\text{null}\ x)\ 0)$

$((\text{eq}\ x\ y)\ (\text{f}\ x))$

$(+ (\text{g}\ y))$

In Algol:

~~If `null(x)` then 0~~

~~Else if $x = y$ then `f(x)`~~

~~else `+ (g(y))`~~

Why is everything a function application in Lisp?

→ Simplicity Principle: - If there is only one basic mechanism in a language, the language is easier to learn, understand and implement.

(2) The list is the primary data structure constructor:

→ The one of Lisp's goal was to allow computation with symbolic data. This is accomplished by allowing the programmer to manipulate lists of data.

(set 'text' (to be or not to be))

The second argument to set is the list (to be or not to be)

The list above is composed of four distinct atoms: (to be or not to be).

List of list: ((to be or not to be) (that is the question))

The list is the only data structure constructor originally provided by Lisp, another example of simplicity.

If there is only one data structure, then there is only one about which to learn and which to choose when programming our application.

(3) Programs are represented as lists:

Function applications and lists look the same. That is, the S-expression

(make-table text nil)

could either be a three element list with elements are the atoms make-table, text and nil; or it could be an application of function make-table to arguments text and nil. Which is it?

- The answer is that it is both because a Lisp program itself is a list.
- Under most circumstances an S-expression is interpreted as a function application, which means the arguments are evaluated and the function is invoked.
- However if the list is quoted, then it is treated as data, that is, it is unevaluated. The function of prefixed quote mark in

(set 'text '(to be not be)) is to indicate that the

list is treated as data not as function application.

If it had been omitted as in:

(set 'text (to be or not to be)) then Lisp interpreter would have

attempted to call a function named 'to' with arguments 'be', 'or', 'not', 'to', 'be'. This would, of course, be an error.

The fact that Lisp represents both programs and data in the same way is of utmost importance (and almost unique among programming languages).

(4) Lisp is often interpreted:

- Most Lisp systems provide interactive interpreters.
- We interact with Lisp interpreter by typing in function applications.

(plus 2 3)

⇒ 5

(eq (plus 2 3) (difference 9 4))

⇒ t

- ① functional Application is the central idea
- ② Lists are primary data structure constructor
- ③ Programs are represented as lists
- ④ Lisp is interpreted.

(2) What are different searching techniques in LISP? Explain them with the help of walking down diagram.

OR

What are differ

How does CAR and CDR help in searching the data elements? Explain with the help of walking down diagram.

Ans: LISP provides selectors such as CAR and CDR for extracting the data from list or searching the list.

⇒ CAR selects the first element of list. for example:

`(car '(principles of programming languages))`

returns atom principles. The first element of a list can be either an atom or a list and car returns it whatever it is.

~~`car ((to 2) (three 3) (four 4))`~~

~~returns (to 2).~~

Argument to car is always non-null list (otherwise it can't have first element).

⇒ CDR provides access to the rest of the elements of list except the first.

Eg: (cdr '(one two three four))

returns the list (two three four)

Similarly: (cdr '(one 1) (two 2) (three 3))

returns ((two 2) (three 3))

- Like car, cdr also requires a ~~null~~ non-null list for its arguments (otherwise we cannot remove the first element).
- Unlike car, cdr always return a list. This could be a null list.

Eg: (cdr '(one)) ~~returns ()~~

Both cdr and car are pure functions and don't modify their arguments list.

- So, by using car and cdr, we can search any part of list no matter how complicated.

Suppose, we have following list:

(set! ((Don Smith) 45 80000 (August 2 1980)))

By applying following operations, we get:

Eg: (cdr '(one two three four))

returns the list (two three four)

Similarly: (cdr '(one 1) (two 2) (three 3))

returns ((two 2) (three 3))

- like car, cdr also requires a non-null list for its arguments (otherwise we cannot remove the first element).
- unlike car, cdr always return a list. This could be a null list.

Eg: (cdr '(one)) returns ()

Both cdr and car are pure functions and don't modify their arguments list.

- So, by using car and cdr, we can search any part of list no matter how complicated.

Suppose, we have following list:

(setDS' ((Don Smith) 45 80000 (August 2 1980)))

By applying following operations, we get:



(cdr DS) → (45 3000 (August 25 1980))

(cdr (cdr DS)) → (30000 (August 25 1980))

(cdr (cdr (cdr DS))) → ((August 25 1980))

(car (cdr (cdr (cdr DS)))) → (August 25 1980)

(car (car (cdr (cdr (cdr DS))))) → August.

We can see that the combination of car_x and cdr_y become large for complex selections. So, LISP provides abbreviation.

For example: an expression such as:

(car (cdr (cdr (cdr DS)))) becomes

(caddr DS)

The composition of car_x and cdr_y is represented by the sequence of a's and d's between initial 'c' and final 'r'.

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

To access the last name:

~~(car@x) → (Don Smith)~~

~~(cdar DS) → (Smith)~~

(Cadar DS) → Smith

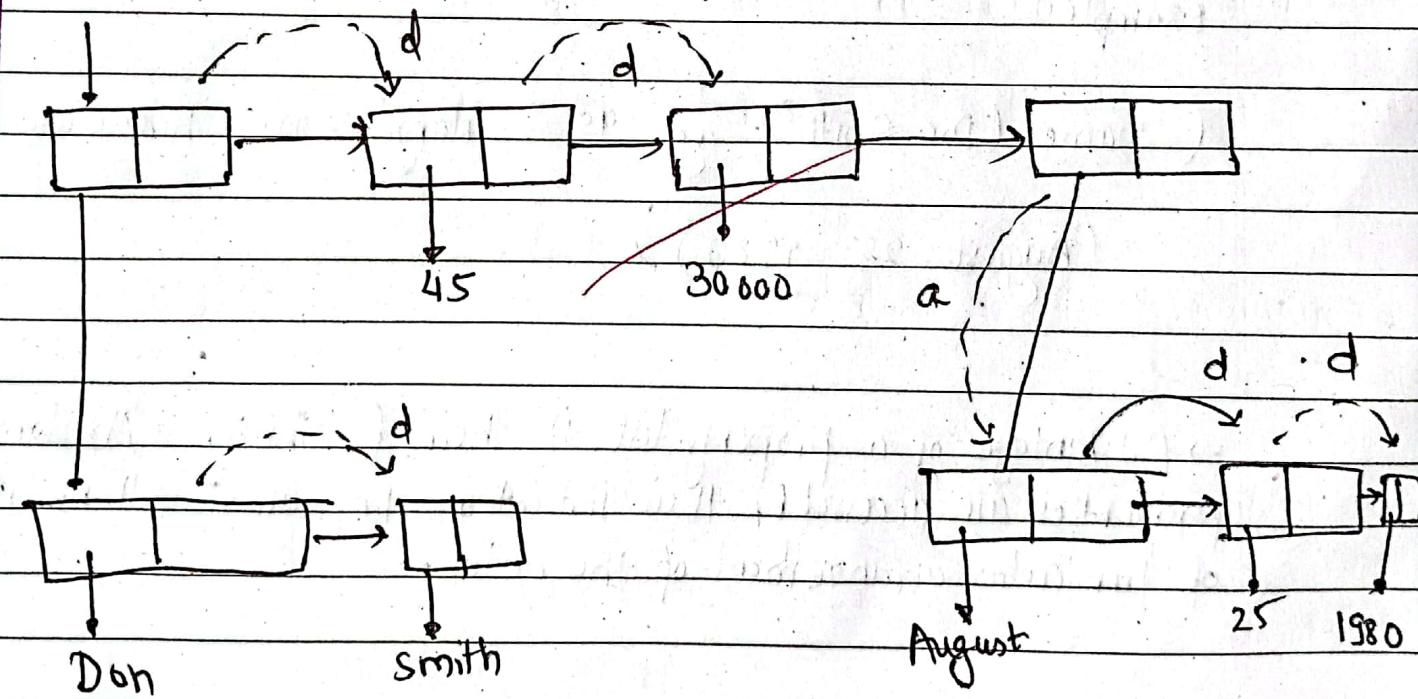
a → (Don Smith)

d → (Smith)

a → Smith

(caddr DS)

To access the year Don was hired, we use the linked list to visualize where 'd' moves to right and 'a' moves down.



~~fig:~~ Walking down a list structure

Date _____
Page _____

(3) Differentiate between Association list and property list with relevant examples.

Ans: Property list:

The method of representing information in the form $(p_1 v_1 p_2 v_2 \dots p_n v_n)$ where p_i is the indicator for a property and each v_i is the corresponding property value, is called a property list or p-list.

Example:

$(\text{name (Don Smith)} \text{ age } 45 \text{ salary } 30000 \text{ hire-date}$
 $\text{(August 25 1980)})$

→ Advantage of a property list is their flexibility. As long as properties are accessed by their indicators, programs will be independent of particular arrangement of the data.

How can the properties of p-list be accessed?

→ let's suppose we are searching for age in above p-list. We begin by looking at the first element of list; if it is age then the second element is the value of age so we return it and we are done.

If the first element of the list is not an age, we skip first two elements (first property and value) and repeat the process by checking the new first element.

Implementation of above searching algorithm.

Suppose, p is the property which we are searching and x is the object or p-list that we are searching.

We will get a getprop function such that $(\text{getprop } p \ x)$ is the value of p property in p-list x .

If the first element of x is p , we can do this by $(\text{eq } (\text{car } x) \ p)$. If first element of x is p , the value of $(\text{getprop } p \ x)$ is second element of x i.e. $(\text{cadr } x)$.

If first element is not p , we skip first two elements and continue further. $(\text{getprop } p \ (\text{caddr } x))$ will look for p beginning with third element of x .

The Algorithm would be :

$(\text{getprop } p \ x) =$

g: $(\text{name Ram age 25 salary 30000})$

If the first element of the list is not an age, we skip first two elements (first property and value) and repeat the process by checking the new first element.

Implementation of above searching algorithm.

Suppose, p is the property which we are searching and x is the object or p-list that we are searching.

We will get a getprop function such that $(\text{getprop } p \ x)$ is the value of p property in p-list x .

If the first element of x is p , we can do this by $(\text{eq } (\text{car } x) \ p)$. If first element of x is p , the value of $(\text{getprop } p \ x)$ is second element of x i.e. $(\text{cadr } x)$.

If first element is not p , we skip first two elements and continue further. $(\text{getprop } p \ x)$ will look for p beginning with third element of x .

The Algorithm would be:

$(\text{getprop } p \ x) =$

Eg: $(\text{name Ram} \ \text{age 25} \ \text{salary 30000})$

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

i.e.

```
fun getprop(p,x) {
  if (car x) == p return (cadr x)
  else return getprop(p, (cdr x))}
```

}

(2) Association list:

The property list data structure works best when exactly one value is to be associated with each property.

If property has several associated values, problems may arise with p-list.

These problems are solved by another USP data structure called associative list or a-list. An a-list is list of pairs with each pair associated two pieces of information.

The general form of an a-list is:

~~C(a₁, v₁) (a₂, v₂) (a₃, v₃) ... (a_n, v_n))~~

As for property lists, the ordering of information is an a-list is immaterial. Information is accessed associatively; that is, ~~given~~

The a-list representation of properties of Don Smith is:

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

(2) Association list:

The property list data structure works best when exactly one value is to be associated with each property.

If property has several associated values, problems may arise with p-list.

These problems are solved by another USP data structure called associative list or a-list. An a-list is list of pairs with each pair associated two pieces of information.

The general form of an a-list is:

~~$C(a_1, v_1) \ (a_2, v_2) \ (a_3, v_3) \ \dots \ (a_n, v_n)$~~

As for property lists, the ordering of information is on an a-list if immaterial. Information is accessed associatively; that is given

The a-list representation of properties of Don Smith is:

$$\begin{aligned} & (\text{Name} \ (\text{Don Smith})) \\ & (\text{Age} \ 45) \\ & (\text{Salary} \ 30000) \\ & (\text{Birth-date} \ (\text{August } 25 \ 1980)) \\ &) \end{aligned}$$

The function that does the forward association is called assoc.

for example:

(set 'DS' ((name (Don Smith))
 (age . 45)))

)

(assoc . 'age DS') gives 45

allowing in lisp

- (g) Explain the structures : conditional expression, The logical connectives and Mapcar and reduce functions.

① conditional expression:

LISP traditionally has cond statement for conditions. later versions also have if.

Basic syntax of cond is:

(cond ((predicate1) (then do something))

 C (predicate2) (then do something2))

 (t (else do this)))

)

Syntax of If:

(if (only one predicate)

 (then do something))

 (else do this))

)

With If there is only one predicate (or grouped predicate), one then and no optional else statement.

IF can only accept one statement for then and else clause

② mapcar:

- mapcar is a function that calls its first argument with each element of its second argument, in turn. The second argument must be a sequence.
- The 'map' part of the name comes from the mathematical phrase "mapping over a domain", meaning to apply a function to each of the elements in a domain.
- And 'car', of course, comes from the lisp notation of the first of a list.

for example: $(\text{mapcar } '1 + ' (2 4 6))$
 $\Rightarrow (3 5 7)$

The function $1 +$ which adds one to its argument, is executed on each element of the list, and new list is returned.

③ Reduce functions:

The reduce function combines all the elements of a sequence using a binary operation; for example, using + one can add up all the elements.

reduce function sequence & key: from-end :start :end :initial-value.

(reduce '+ '(1 2 3 4)) \Rightarrow 10

⑦ How hierarchical structures are processed in LISP?

Ans:

Small Talk (Object oriented Programming)

(2) Explain message passing and returning mechanism in Small Talk.

Ans: In smalltalk, the data elements (objects) are active; they respond to messages that cause them to act on themselves, perhaps modifying themselves, perhaps modifying or returning another object.

→ Messages are ways for communication between objects.

for example:

Scribe goto: loc which is equivalent to

Scribe goto(loc);

→ When a message is sent to an object, following steps take place:

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

(2) Identify the method being invoked by extracting the template from the message and then looking it up in the message dictionary for receiving object's class or superclasses.

That is, if it is not defined in the class, we must look ~~in~~ in the superclass, if it is not defined in the superclass, we must look in its superclass and so on.

- (3) Transmit the parameters to receiver's activation record.
- (4) Suspend the sender (caller) by saving its state in its activation record.
- (5) Establish a path (dynamic link) from the receiver back to the sender, and establish the receiver's activation record as the active one.

As one would expect, returning from a method must reverse this process.

- (1) Transmit the returned object (if any) from the receiver back to the sender.
- (2) Resume execution of the sender by restoring its state from its activation record.

→ In accordance to the Information Hiding Principle, the Storage Manager handles allocation and deallocation of all objects; this includes activation record objects. The effect of this is that activation records are created from free storage and reclaimed by reference counting, just like other objects.

→ This is the reason we do not explicitly deallocate activation records.

Instance

length
c.d.
instance vars

activation record

locals	parameters
non-locals	temporary vars
instruction part	intermediates
sender part	static link
	method
	instruction
	dynamic link

class

length
c.d.
superclass
class variables

superclass

length
c.d.
superclass
superclass variables

Fig: Parts of an Activation Record

② What are different forms of message template in Smalltalk? Explain them.

Ans: Messages are essentially procedure invocations, although the formats allowed for messages are a little more flexible. In most languages parameters are surrounded by parentheses and separated by commas; in Smalltalk parameters are separated by keyword.

For example:

`newBox setLoc: initialLocation tilt: 0 size: 100`

which is equivalent to Ada procedure call:

`NewBox. Set (initial-location, 0, 100);`

However, Smalltalk is not following labeling principle here since the parameters are required to be in right order even though they are labeled.

~~→~~ The Smalltalk has three formats for messages / message templates:

(1) Keywords for parameterless messages (e.g.: `Bl show`)

(2) Operators for one-parameter messages (e.g.: `x+y`)

(3) Keywords with colons for messages with one or more parameters

e.g.: `Scribe grow: 100`

The message format, Keywords followed by colon can be used if there are one or more parameters to method. If the method has no parameters, it would be confusing to both human and system if the keyword were followed by colon. This leads to format that we have seen for parameterless messages.

B1 show

omitting the colon

These two message formats are adequate for all purposes, they handle any number of parameters from zero on up.

Unfortunately, they would require writing arithmetic expressions in an uncommon way. for example: to compute $(x+t_2) \times y$ we would have to write:

$(x \text{ plus: } t_2) \text{ times: } y$

To avoid this unusual notation, smalltalk has made a special exception; the arithmetic operators (and other special symbols) can be followed by exactly one parameter even though there is no colon.

$x + t_2 * y$

The object named x first sent the message t_2 , and object resulting from this it sent the message $\times y$. Thus, this expression computes $(x+t_2)y$. The smalltalk does not obey usual precedence rules (a concession to Regularity Principle).

(4) How is Activation Record represented in SmallTalk?

Ans: Activation records are the primary vehicle for procedure implementation. Since, an activation record holds all of the information that pertains to one activation of a procedure, the process of procedure call and return can be understood as the manipulation of activation records. The activation records in Smalltalk are used to hold all of the information relevant to one activation of method.

Activation record of SmallTalk has three main parts:

- ① Environment part: Context to be used for execution of method
- ② Instruction part: The instruction to be executed when this method is resumed.
- ③ Sender part: The activation record of the method that sent the message invoking this method.

The sender part is a dynamic link, i.e., a pointer from the receiver's activation record back to sender's activation record. It is just an object reference since activation records like everything in Smalltalk are objects.

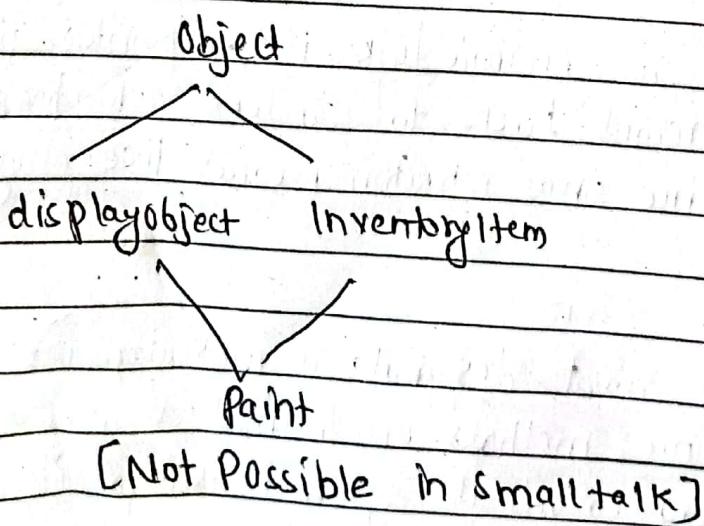
The instruction part must designate a particular instruction in a particular method. Since, methods are themselves objects (instances of class method), a two co-ordinate system is used for identifying instructions.

The environment part provides access to both local and non-local environments. The local environment includes space for the parameters to method and temporary variables. This part of activation record must provide space for hidden temporary variables such as intermediate expressions.

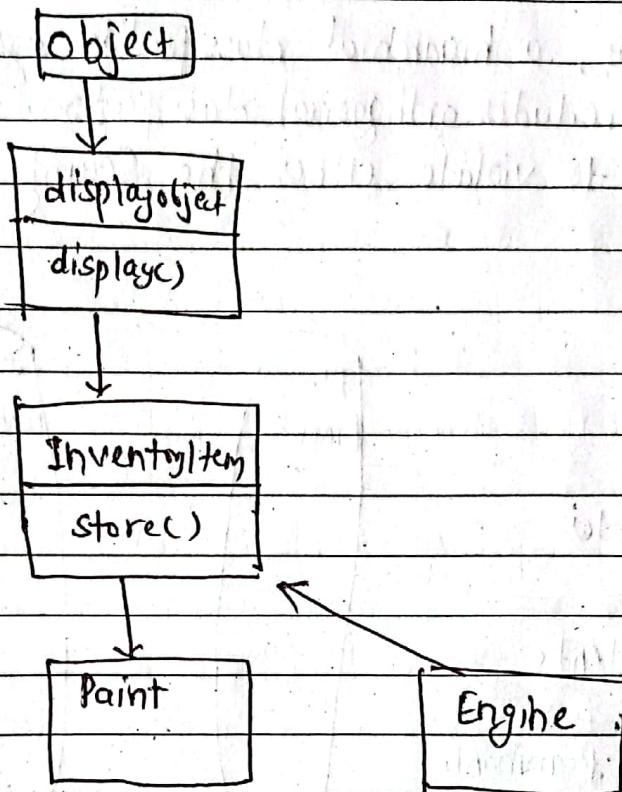
- (5) How do classes allow multiple representation of data types in Smalltalk? Explain with orthogonal classification.

Ans: Smalltalk organizes classes into a hierarchy; each class has exactly one immediate superclass.

But trying Suppose, we have a base class object. Two objects displayObject and InventoryItem inherits from class object. If we consider a class such as Paint, it has behaviours of both displayObject and InventoryItem. But this will be not possible in Smalltalk as one class has exactly one immediate superclass.



To overcome this, we can choose either `displayObject` or `InventoryItem` to be super class of other. Suppose, we choose `displayObject`.



This cause one problem; some classes like `Engine` are `InventoryItem` but not `displayObject`. They will have unwanted `display()` method. This is violation of Security principle.

Another solution is to repeat data/methods of `InventoryItem` in `Paint` and making it inherit from `displayObject` only. This will validate Abstraction principle.

In real life, we often find that the same objects must be classified in different ways. For example, a biologist might classify mammals as primates, rodents, ruminants, etc. A zoo might classify them as North American, South American, African, etc.

These are orthogonal classifications; each of the classes cuts across the others at right angles.

In summary, a hierarchical classification system, such as provided by Smalltalk, precludes orthogonal classification. This in turn forces a programmer to violate either the Security Principle or Abstraction Principle.

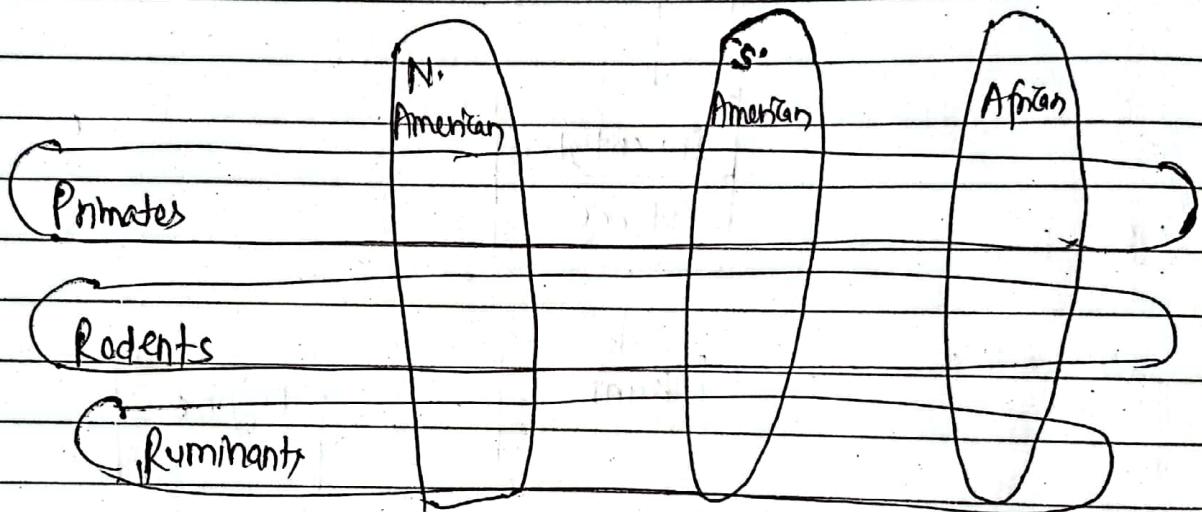


Fig: Example of Orthogonal Classification.

~~Seen~~

R.D.

Algol 60 : [Generality and Hierarchy)

Name structure:

- The primitives bind names to objects
- The constructor is the block
- Block defines nested scopes
- Dynamic and static scoping
- Block structured storage allocation in Algol.

Control structure

- Controls the flow of program [logical & looping]

① Branching statements:

- Algol provides same structure that of Fortran but in generalized and regularized form.

The if statement is also extended beyond Fortran to having an else-part, which is executed if condition is false.

The if statement can have any other statements including if - statement (not allowed in FORTRAN).

if $T[\text{middle}] = \text{Sought}$ then $\text{location} := \text{middle}$
else $\text{location} := \text{middle} + 1$;

If statements can have multiple statements as well:

If $X=Y$ then

begin

$Z := P$;

$A := B$;

end

else

begin

$Z := Q$;

$A := C$;

end

② Looping:

→ Algol for-loop is more general than Fortran Do-loop. It includes the function of a simple do-loop. For example,

for $i:=1$ step 2 until $N \times M$ do

$\text{inner}[i] := \text{outer}[N \times M - i]$;

(3) Nested statements:

compound statement \rightarrow statement that brackets any number of statements together and converts them to a single statement. A group of statements surrounded by begin and end is considered as a single statement and can be used anywhere where a single statement is allowed.

```
if X = Y then  
begin  
  if X > Z then  
    A := B  
  else  
    A := C  
end
```

(4) Procedures are recursive;

* Data Structure :

① Primitives Types → They are the mathematical scalars as Algol 60 was intended for scientific applications.

Therefore primitive data types are mathematical scalars: - integer, real and Boolean.

→ No double precision types because their use is necessarily machine dependent and one of the goals of Algol 60 was to be universal and hence machine independent language.

→ Algol takes a simpler approach to single and double precision problem i.e. a single type real.

Example:

```
begin  
  integer x;  
  x := 23  
end
```

② Arrays:

Arrays of integer and reals could be defined according to the pattern:

integer array A [start : end];

or

real array A [start : end];

or just:

array A [start : end];

in which case real was defaulted.

- The lower bounds can be any numbers unlike in fortran where lower bounds is 1.

e.g.: Integer array NumberofDays [-100 : 200]

We can also use negative indices for temperature below zero.

for example: NumberofDays [-25] represents the number of days with temperature less than 25.

Accessing elements in array can be done in modern ways:

A[4] or A[x]

③ Switches:

- Switch was an array data type containing jump labels. Such arrays could be used indexed as a label statement in a go to clause such as;

go to S[N]

jumping to position determined by contents of switch S at index position N.

This was later on declared evil by many programming language designers and programmers since the jump destination is not easily predictable by any program analyzer.

* Dynamic Arrays:

- size of arrays may not be known while declaring.
- In language with static arrays like FORTRAN, we dimension the array to 800 elements to solve this. But if number of elements are far fewer, it causes wastage of space. If elements are more than 800, it causes overflow.
- Algol permits dynamic array and permits expressions/variables in array declarations. This results in exactly right size of data.

Eg.: begin

integer N = 10;

integer array A[N];

end

» Algol has strong type system.

Procedure

integer procedure simple (param);

 value param;

 integer param;

 param := param + 1;

→ In Algol, procedures are recursive. for example, factorial:

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

integer procedure fac(n);

 value n;

 integer n;

 fac := if $n = 0$ then 1 else $n \times \text{fac}(n-1)$;

Pass By Value:

→ Value of the actual parameter is copied into a variable corresponding to formal parameter.

→ Pass by value is secure and reliable.

→ Since a local copy is made of actual, there is no possibility of an assignment to formal overwriting the actual; such assignment will only modify the copy.

for example,

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

is perfectly harmless. The assignment to n does not get reflected to actual parameter passed.

→ Pass by value is inefficient for arrays.

Consider the following skeleton for a procedure to compute average of an array:

```
real procedure avg(A, n);
```

```
value A, n;  
real array A; integer n;  
begin
```

```
end;
```

(A)

Since, both the array to be averaged and number of elements in the array (n) are input parameters, we have passed them both by value.

When the procedure Avg' is called, an activation record will be created containing space for values of actual parameters. This will include space for entire array A. Not only is this wasteful of storage, but it also wastes time since entire actual parameter corresponding to A must be copied into the activation record. For a large array, it could take as long to pass the array parameter as to compute the average. The conclusion we can draw is that pass by value is not satisfactory way of passing arrays.

Pass by Name:

- Rather than using pass-by-reference for input/output parameter, Algol used the more powerful mechanism of pass-by-name.
- In essence, we can pass in the symbolic "name" of a variable which allows it both to be accessed and updated.
- For example, to double the value of C[j], we can pass its name (not value) in the following procedure.

```
procedure double(x);
```

```
real x;
```

```
x := x * 2;
```

- In general, effect of pass-by-name is to textually substitute the argument expressions in a procedure call for corresponding parameters in body of procedure. e.g. double(c[j]) is interpreted

as $c[j] := c[j] * 2;$

Technically if any of the variables in the procedure clash with the caller's variables, they must be renamed uniquely before substitution.

→ Implications of pass-by-name mechanism:

- ① The argument expression is re-evaluated each time formal parameter is accessed.
- ② The procedure can change the values of variables used in the argument expression and hence change the expression's value.

Pass-by-Name Elegance - Jansen's Device:

→ Putting expressions into a procedure so they can be repeatedly evaluated has some valuable application.

→ Suppose we want to calculate $x = \sum_{i=1}^n v[i]$ as a procedure

$$x := \text{sum}(i, 1, n, v[i])$$

This is easy to do with name parameters, by altering the index variable i .

real procedure sum (k, l, u, ak);

value l, u;

integer k, l, u;

real ak; real s;

begin

s := 0

for k := l step 1 until u do

s := s + ak

sum := s

end;

Each time through loop, each time K (or i) is changed
 a_k ($v[i]$) is reevaluated.

Thunks:

- Thunks are parameterless procedures used to implement pass-by-name.
- A thunk returns a pointer to current location represented by a parameter.
-

Pass-by-Name can be dangerous

A sample program to swap two integers:

1. procedure swap (a, b);
 integer a, b, temp;
 begin
 temp := a ;
 a := b ;
 b := temp ;
 end .

2. Effect of call swap (x, y) :

temp := x ;
 $x := y$;
 $y := \text{temp}$;

with swap (5, 4) :

temp := 5 ;
 $x := 4$;
 $y := 5$;

3. Effect of call swap ($i, x[i]$)

temp := i ;
 $i := x[i]$;
 $x[i] := \text{temp}$;

let, $i = 2$ $x[2] = 5$

temp := 2 ;
 $i := 5$;
 $x[5] = 2$;

So After call;

$i = 5 \rightarrow x[5] = 2, x[2] = 5$
 So, swap (i, j), swap (j, i), swap ($A[i], j$) works but
 swap ($i, A[i]$) does not work!

Syntactic Structures:

(1) Machine Independence and Portability:

→ Algol focuses on making itself machine independent hence it adopted free format i.e. format independent of columns or other details of layout of program.

→ This convention is familiar from natural language such as English, in which the meaning of sentences does not depend on the columns or the number of lines on which they're written.

→ Almost all languages designed after Algol have used a free format.

→ Algol allowed continuous stream as in English:

```
procedure sum(a,b); value a,b; integer a,b; begin, sum = 0;
sum := a + b; end;
```

However it is not readable.

(2) 3 levels of representations used:

① Reference language:

→ It is used in all examples in Algol report.

Eg: $A[i+1] := A[i] + 1;$

④ Publication language:

→ Intended to be the language used for publishing algorithms in Algol. Allows various lexical and printing conventions (subscripts and Greek letters) that would aid readability.

$$\text{for } a_{i+1} \leftarrow (a_i + \pi * r) / 6.02 \times 10^{35};$$

⑤ Hardware representations:

→ Hardware representations of Algol that would be appropriate to character sets and input and output devices of their computer systems.

⑥ Algol solved some problems of fortran lexics:

→ In Algol, keywords are bold faced or underlined and have no relation with identifier.

→ This means that the confusion between keywords and identifiers that occurs in fortran is not possible in Algol.

Eg: if procedure then until := until + 1 else do := false;
underlined are keywords.

The above example is perfectly unambiguous (though confusing)

There are many hardware representations for above statements

'if' procedure 'then' until := until + 1 'else' do := 'false';

IF procedure THEN until := until + 1 ELSE do := FALSE;

#IF procedure #THEN until := until + 1 #ELSE do := #FALSE;

The words (keywords) used by Algol are marked in some unambiguous way (preceded with #, surrounded by "", capital, etc.)
This is Keywords approach.

Other Three lexical conventions:

(1) Reserved words: Cannot be used as identifier. Eg: Java

(2) Keywords: keywords are marked separately. Eg: Algol

(3) Keywords in context: words used by language are keywords only in those contexts in which they are expected, otherwise they are identifiers. Eg: fortran

(4) Arbitrary restrictions are eliminated:

Algol adheres to zero-one-infinity principle. for y: there is no limit on number of characters in identifier.

→ Reduces the number of special cases for programmer and they support the idea "anything you think you ought to be able to do, you will be able to do".

else
Dangling else problem:

Algol's syntactic conventions had some problems. e.g:-

If B then if C then S else T

Does the else go to first if or second? That is, is this statement equivalent to:

If B then begin if C then S else T end

or

If B then begin if C then S end else T

This is called dangling else problem.

Algol solved this by requiring that the consequent of an if-statement to be an unconditional statement.

① Descriptive Tools (BNF):

- Descriptive tools are used to formulate, record and evaluate various aspects of developing design. Although English and other languages can be used for this purpose, most designers have developed specialized languages, diagrams and notations for representing aspects of their work that otherwise could be difficult to express.
- Natural language prose is not sufficiently precise. They may be understood differently by different people.
- Natural language also takes up more space than symbols or notations while describing a programming language.
- Backus and Naur developed Backus-Naur form (BNF) for defining and describing Algo.
- BNF gives different structures for describing a language

① name ::= expansion

The symbol ::= means 'may expand into' or 'may get replaced with'.

- Every name in Backus-Naur form is surrounded by angle brackets <>, whether it appears on left or right hand side of rule.
- An expansion is an expression containing terminal symbols and non-terminal symbols, joined together by sequencing and selection.

A terminal symbol may be a literal like C "+" or "function") or a category of literals (like integer).

A vertical bar | indicates choice, (or operator)

BNF of decimal fraction:

$\langle \text{decimal fraction} \rangle ::= \begin{array}{l} \text{terminal} \\ \bullet \langle \text{unsigned integer} \rangle \end{array}$

→ BNF is a meta language as it is used to describe another lang.

$\langle \text{integer} \rangle ::= \begin{array}{l} + \langle \text{unsigned integer} \rangle \\ | - \langle \text{unsigned integer} \rangle \\ | \langle \text{unsigned integer} \rangle \end{array}$

Recursion is used for repetition:

→ $\langle \text{unsigned integer} \rangle ::= \begin{array}{l} \langle \text{digit} \rangle \\ | \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle \end{array}$

Extended BNF is more descriptive:

→ In BNF, we represent idea of ^{sequence} one or more digits through recursion.

→ It would be more convenient to have a notation that directly expresses the idea 'sequence of one or more of ...'. One such notation is Kleene cross C^+ , which means sequence of one or more strings from syntactic category C.

So, unsigned integer can be defined as:

$$\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle^+$$

A useful variant of this is Kleene star, C^* which stands for sequence of zero or more elements of class C.

for example, an Algol identifier (name) is a $\langle \text{letter} \rangle$ followed by any number of $\langle \text{letter} \rangle^*$ or $\langle \text{digit} \rangle^*$ i.e.

$$\langle \text{alphanumeric} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{digit} \rangle$$

$$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \langle \text{alphanumeric} \rangle^*$$

We do not need to separately define $\langle \text{identifier} \rangle$ if it is used only in one place and we do not have to give it.

$$\langle \text{identifiers} \rangle ::= \langle \text{letter} \rangle \{ \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \}^*$$

This can be more pictorial as:

$\langle \text{Identifier} \rangle ::= \langle \text{letter} \rangle \left[\begin{array}{l} \langle \text{letter} \rangle \\ \langle \text{digit} \rangle \end{array} \right]^*$

for $\langle \text{integer} \rangle$:

$\langle \text{integer} \rangle ::= + \langle \text{unsigned integer} \rangle$
| - $\langle \text{unsigned integer} \rangle$
| $\langle \text{unsigned integer} \rangle$

We can use alternative notation:

$\langle \text{integer} \rangle ::= \left\{ \begin{array}{l} + \\ - \end{array} \right\} \langle \text{unsigned integer} \rangle$
| $\langle \text{unsigned integer} \rangle$

This is better but we want to express the idea that $\langle \text{unsigned integer} \rangle$ is optionally preceded by a '+' or '-'. The square bracket is often used for this purpose.

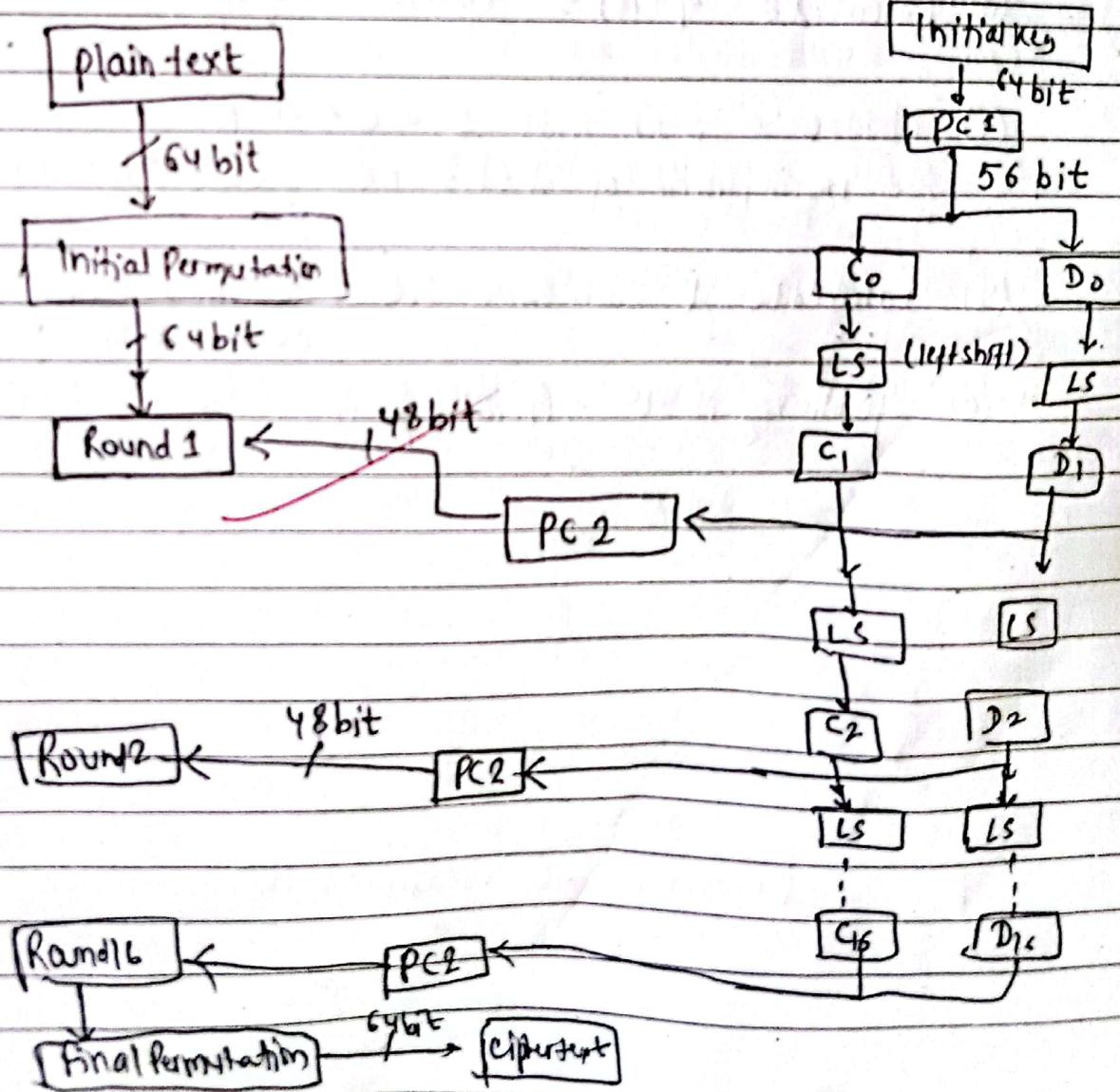
$\langle \text{integer} \rangle ::= \left[\begin{array}{l} + \\ - \end{array} \right] \langle \text{unsigned integer} \rangle$

Extended BNF adheres better to Structure Principle: that is forms of EBNF definitions are closer (visually) to strings they define than pure BNF notation.

These three formats fits ① Zero-one-infinity Principle since they're the only special cases are for zero parameters and one parameter. However, the fact that these cases are handled differently from the general case violates the Regularity Principle.

Encryption :

① Symmetric key : DES (Data Encryption Standard)



FORTRAN : Emphasis on efficiency:

Structural organization:

- ① Programs are divided into disjoint subprograms:

Main program

Subprogram 1

Subprogram 2

- ② Constructs are either declarative or imperative

- declarative → describes data areas, their length and their values
- imperative → contains commands to be executed during running of program.

In fortran, declarative constructs are called non-executable statements and imperative constructs are executable statements.

- ③ Declarations include bindings and initialized initializations:

- while declaring a variable,

- ① An area of memory of specified size is allocated.
- ② They effect A symbolic name is attached to that area of

memory. This is called binding a name to an area of memory.

③ Content of memory areas are initialized.

Eg: DIMENSION DTA(800)

→ allocates 800 words and bind the name 'DTA'.

Instructions are either computational, control flow or Input - Output

Computative statements can be of three types:

- (i) Computational statements → such as arithmetic and ~~logical~~ operations
- (ii) Control flow statements → comparisons and looping statements
- (iii) Input - output statements → READ and PRINT

The primary computational statement is assignment statement.

Eg: AVG = SUM / FLOAT(N)

A program goes through several stages to be run;

- ① compilation ④ Execution
- ② Linking
- ③ Loading

Control structure:

→ Governs the flow of control of a program.

- ① lower level control structure - IF, GOTO
- ② higher level control structure - Do-loop
- ③ subprogram control structure - CALL, RETURN

① IF statement:

IF (condition) statement

e.g.: If ($X \cdot EQ \cdot A(I)$) $K = I - 1$

If X equals to $A(I)$ then assign $(I - 1)$ to K .

(FORTRAN uses ' $\cdot EQ \cdot$ ' for equality relation)

Similarly, ' $<$ ' and ' $>$ ' are written as ' $.LT.$ ' and ' $.GT.$ ' respectively.

② GOTO statement:

→ Transfers or jumps to a particular instruction.

→ Goto is the workhorse of control flow.

→ Goto is the raw material from which control structures are built.

for example: A two-way branch is often implemented with a logical if-statement and a goto.

If (condition) goto 100
case for false condition

Goto 200

100 case for false condition
200 other statements

This is an example of if-then-else or conditional statement structure of newer programming languages.

Goto can be used for dividing into more than two cases:

(goto (10, 20, 30, 40), I
10 handle case 1
20 handle case 2
30 handle case 3
40 handle case 4
100 other statements

if I is 1 goto 10
2 20
3 30
4 40
positions

③ Loops can be implemented using Goto:

Example: → a trailing -decision loop

```
100    body of loop  
      IF (loop not done) Goto 100
```

→ A leading - decision loop

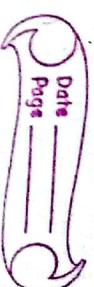
```
100  IF (loop done) Goto 200  
      body of loop  
      Goto 100
```

```
200  other statements
```

This is similar to while ~~do~~ and do-while loop of modern languages.

Since, goto can be used to implement all construct of control statement it is the workhorse of control flow.

Problems: → Using one statement (Goto) to build all control structures is that we never know what control structure is intended without going deep into the code. for example, if could be branching or looping depending upon use of Goto.



→ Goto's are hard to understand or read.

→ Computed and Assigned Goto's are easily confused:

~~Computed Goto:~~

Goto (L₁, L₂, L₃, ..., L_n), I

where L_i are statement numbers and I is any integer.
The computer Goto transfer to statement number L_I if I
contains K. The compiler stores address of each statement L_i
in an array and then compiler code to use I as an index to this
array.

Assigned Goto:

Goto N, (L₁, L₂, ..., L_n)

where N is also an integer variable. This statement transfer
to statement whose address is in the variable N.

Most compilers don't check whether the statement whose
address is in N has its label included in the list.

The only Goto is used in conjunction with ASSIGN statement
which assigns the address of a statement number into a variable.

ASSIGN 20 TO N

Assigns address of statement 20 to N.

The computed and assigned goto's are easily confused as they are almost identical. So, it is not uncommon for developers to write computed goto where an assigned goto is intended.

This violates 'Syntactic consistency principle' — things which look similar should be similar and things which look different should be different'.

Assigned and computed goto looks similar but are different.

⑥ Do-loop:

- Do-loop is more structured than goto.
- Fortran contains only one built-in higher level control structure, the Do-loop.
- Do-loop provides a method of constructing a counted loop.

Syntax: Do label I=1, N]
 body
 [continue — ends the Do-loop
 label continue

→ Do-loop can be nested.

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

- ↳ Fortran has rather a linear structure than a hierarchical structure
- ↳ The statements are listed one after another and numbered in a way reminiscent of memory addresses rather than being nested like the statements in more modern languages.

Other than Do-loop, only nesting permitted in Fortran is in logical if, although the statement controlled by If is only allowed to be a simple unconditional statement (e.g: assignment or Goto, not an IF or DO).

① Sub-Programs (CALL and RETURN)

In fortran, variables can be passed to a subroutine and subroutines can hold control flow of the program.

SUBROUTINE DIST (D, X, Y)

D = X - Y

IF (D < 0) D = -D

RETURN

END

SUBROUTINE Name (formals)

... body ...

RETURN

END

CALL DIST (DIST1, X, Y)

Parameters are usually passed by reference

→ efficient for arrays but has dangerous consequences.

→ output variables are passed to a subroutine whose values are updated.

→ input variables are input to program and are not for modification

→ But in pass by reference, subroutine can update input variable.
It is called side effect of sub program call.

Ex: SUBROUTINE SWITCH(N)

N = 3

RETURN

END

→ Switch writes 3 into its parameter.

→ CALL SWITCH(I) would result 3 being stored in I.

→ What if a constant/literal is passed?

→ CALL SWITCH(2) : The constant/literal are stored in the area of memory called literal table. This subroutine call passes the address of location in literal table where 2 is stored. In that location, now 3 is written by SWITCH.

→ This corrupts the literal table.

→ It is because compiler would compile a program so that it loads content of this location whenever it requires a 2, even though it now contains 3. So, $I=2+2$ would result in 6.

→ There is no way to distinguish between input/output parameters.

→ 'The compiler should pass by value when a constant or expression is used as an actual parameter to solve this problem'.

Activation Records :

- Subprograms are implemented using activation records.
- Callee (subprogram being called) should know the caller so that it can return control flow to Caller when it is done.
- Following tasks must be completed to perform a subprogram invocation:
 1. Place the parameters in the callee's activation record.
 2. Save the state of the caller in caller's activation record (include the point at which the caller is to resume execution).
 3. Place a pointer to the caller's activation record in the callee's activation record.
 4. Enter the callee at its first instruction.

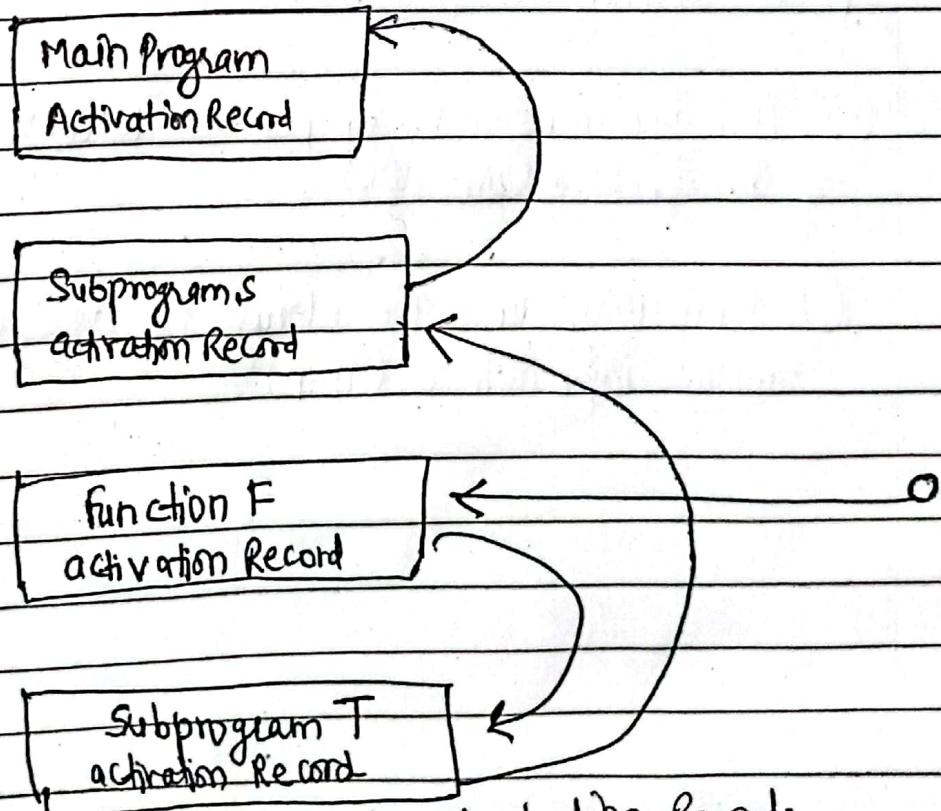


Fig: Dynamic chain of activation Records

The steps required to return from the callee to the 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 the state of its execution (registers, etc.) from the activation record.

The activation record must contain following information:

- ① The parameters passed to this subprogram when it was last called (PAR).
- ② The IP, or resumption address, of this subprogram when it is not exiting.
- ③ The dynamic link or pointer to activation record of the caller of this subprogram (DL).
- ④ Temporary areas for storing register contents and other volatile information (Tmp).

- Name structure of Algol & Fortran
- Data structure of Fortran
- Context Free and Regular Grammar Algol
- Programs of Fortran

LISP :

Cons:

- Cons constructs lists.
- Cons is the constructor and inverse of selectors do what
- Cons add a new element to beginning of a list.

for example:

(cons 'one '(two three)) returns '(one two three)

given by car given by cdr

- Second argument must be a list. The first argument can be either atom or a list.

(cons '(a b) '(c d)) $\Rightarrow ((a\ b)\ c\ d)$

given by car given by cdr

car $\rightarrow (a\ b)$
cdr $\rightarrow (c\ d)$

LISP :

cons:

→ cons constructs lists.

→ cons is the constructor and inverse ~~of~~ what selectors do

→ cons add a new element to beginning of a list.

for example. given by car given by cdr

(cons 'one '(two three)) returns '(one two three)

→ Second argument must be a list. The first argument can be either atom or a list.

→ (cons '(a b) '(c d)) \Rightarrow ((a b) c d)

given by car given by cdr

car \rightarrow (a b)
cdr \rightarrow (c d)

logical connectives:

→ logical connectives can be defined in terms of conditional expressions

- ① → for example, the function of $(\text{or } x \ y)$ has the value 't' (true) if either or both of x and y have value 't'; in any other case 'or' has the value false 'nil'. Another way is to say that if x has value 't' then 'or' has value 't' otherwise the 'or' has same value as y . So can be defined as:

$$(\text{or } x \ y) = (\text{cond } (x \ t) \ (t \ y))$$

- ② Similarly ; $(\text{and } x \ y)$ if x is false , and gives false and if x is true, result is same value as y .

$$(\text{and } x \ y) = (\text{cond } (x \ \text{nil}) \ (\text{nil } y))$$

→ Build a truth table.

- ③ NOT → $(\text{not } x)$ inverts the value of x .

$$(\text{not } x) = (\text{cond } (x \ t) \ (\text{nil } t))$$

Hierarchical structures are processed recursively!

→ Hierarchical structures are processed recursively which would be difficult to handle iteratively.

→ As an example, we try to implement the 'equal' function which determines two arbitrary values are same or not. It is different from primitive 'eq' as 'eq' requires values to be atomic.

→ The 'equal' function will look like:

(equal '(1 2 3) (1 2 3)) \Rightarrow t

(equal '(1 (2 3)) '(1 2 3)) \Rightarrow f

(equal 'One '(Two)) \Rightarrow f

→ Base cases: ① If both x and y are atoms then
 $(\text{equal } x \text{ } y) = (\text{eq } x \text{ } y)$

② If x is an atom and y isn't, or y is an atom and x isn't

then $(\text{equal } x \text{ } y) = \text{nil}$

→ Recursive case: If both are non-atom or list, we recursively check if they are equal or not.

→ First we check if first element is equal or not. If it is not equal, lists x and y are not equal. If first element is equal, we repeat this process with remaining of elements.

If x and y are both lists then

(and (equal (car x) (car y))
(equal (cdr x) (cdr y))
)

So, 'equal' function would be :

(defun equal (x y))

(. or (and (atom x) (atom y) (eq x y))

(and (not (atom x)) (not (atom y))

(equal (car x) (car y))

(equal (cdr x) (cdr y)))

))

Recursive Interpreter:

- The first LISP interpreter resulted from writing a universal function for LISP.
- A universal function is a function that can interpret any other function.
- Developing a universal function is an example of recursive interpreter.
- Recursive interpreter is written in LISP, although it could be written in any language with recursive procedures and the ability to implement linked lists.
- Since recursive interpreter is written in LISP, it makes use of the facilities of LISP, such as the list processing operations 'car', 'cdr' and 'cons'. In particular, these operations are used to interpret 'car', 'cdr' and 'cons' operations in program that we are interpreting. That might seem circular and problematic. In fact it is exactly analogous to way floating-point operations were implemented in pseudo-code.
- The LISP universal function is conventionally called 'eval' since it evaluates a LISP expression. In addition to expression to be evaluated, 'eval' must have a second parameter, which is a data structure (a list of some sort) representing the context in which the evaluation is to be done.

If E is any LISP expression (written in S-expression notation) and A is a list representing a context then:

$$(\text{eval } ' E A) = V$$

where V is value of E in that context.

Eg:

$$(\text{eval } ' (\text{cons } 'A ' (B C D)) \text{ nil}) = (A B C D)$$

Types of LISP expressions:

Type	Example
Numeric atom	2
Nonnumeric atom	val
Quotation	(quote (B C D))
Conditional	(cond (x t) (t y))
Primitive	(cons x y)
User-defined	(make-table text nil)

→ ① Value of a numeric atom if that atom $(\text{eval } 2 \text{ a}) = 2$

② Quote suppresses evaluation:

$$(\text{eval } ' (\text{quote } x) A) = x \quad (\text{No matter what } x \text{ might be})$$

The 'quote' suppresses / prevent evaluation of its argument

③ Arguments are recursively interpreted - evaluated.

Storage Reclamation :

→ 'Cons' operation calls allocate a new storage area. Since 'cons' are so frequent in LISP, it is obvious that the free storage area will be exhausted by most programs. Unless there is some way of reclining allocated area to free area, the program will have to be aborted.

Methods of freeing space:

① Explicit erasure:

When a program no longer needs a cell or an allocated memory, the programmer must explicitly return it to the free storage area.

→ It is the job of programmer to manage the allocation and freeing of memory.

→ Explicit storage erasure has several problems:

① Programmers need to work harder and chances of memory leakage if not managed efficiently.

② It may lead to dangling pointers, pointers that do not point to an allocated cell or pointer still pointing to deallocated area.

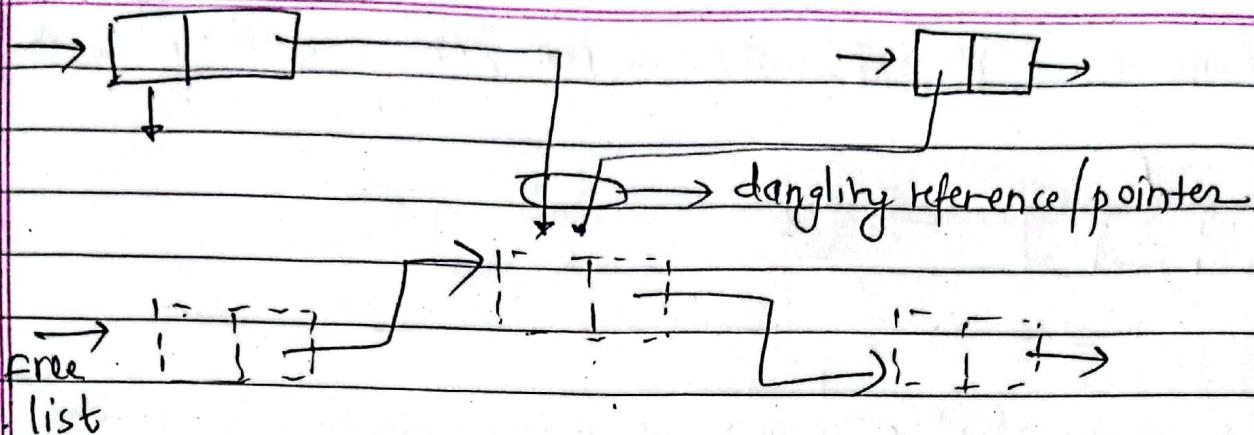


Fig: Dangling pointers .

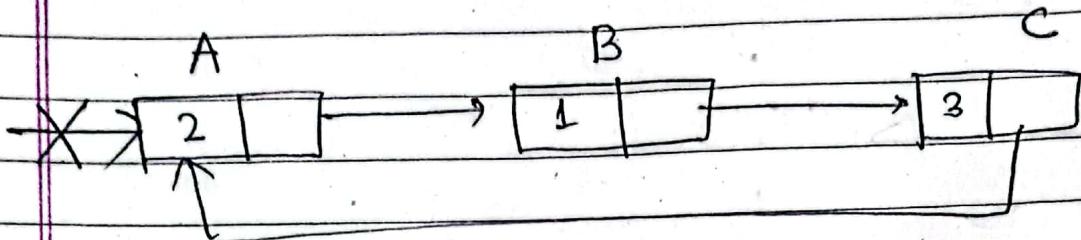
② Automatic Erasure:

- When a storage is no longer needed, it is automatically returned to free space. When a cell is no longer accessible from program, it is returned to free storage.
- An automatic erasure system must keep track of accessibility of each cell. It can be done by maintaining the reference count field in each cell.

③ Garbage Collection:

- Cyclic structures are not reclaimed by ~~an~~ automatic exposure.
- In the given figure, A, B and C form a cyclic structure and entry point is pointer to A. Suppose, pointer to A is destroyed, the structure is inaccessible and should be erased. But, the

Reference count of A, B, and C is not zero due to cyclic structure.



So, cyclic structures cannot be automatically erased or returned to free storage even though they are inaccessible.

- One solution is to disallow the cyclic structures altogether.
- There is an alternative solution/approach to automatic storage reclamation that can handle cyclic structures, it is called garbage collection.
- With reference counts, cells are released as soon as they become inaccessible (reference count 0). Garbage collection adopts a different philosophy. Inaccessible cells are simply abandoned; no attempt is made to return them to free storage. When the supply of free cells is exhausted, the system enters a garbage collection phase in which it identifies all of the unused cells and returns them to free storage. One way to view garbage collection is that it ignores the storage reclamation problem until a crisis develops and then it deals with crisis.
- There are many techniques for garbage collection. One of them is mark-sweep garbage collector, a simplest kind of garbage collector.

→ Mark-Sweep garbage collector operates in two phases. In the mark phase, garbage collector identifies all of those cells that are accessible, that is, that are not garbage. In the sweep phase the garbage collector places all of the inaccessible cells in the free storage area, often by placing them on a free list.

Mark Phase:

→ The garbage collector starts at the root and follows the pointer, marking each cell it reaches as accessible. For this purpose, each cell requires an associated bit that indicates whether or not it is accessible.

At the beginning of garbage collection process, all cells are unmarked. When marking process encounters a cell that is already marked, it knows it need not continue down the path (since already marked). So, this ensures garbage collector will not enter into infinite loop.

mark phase :

for each node R, mark (R)

mark (R) :

if R is not marked then :

 set mark bit of R ;

 mark (R.left);

 mark (R.right) ;

endif

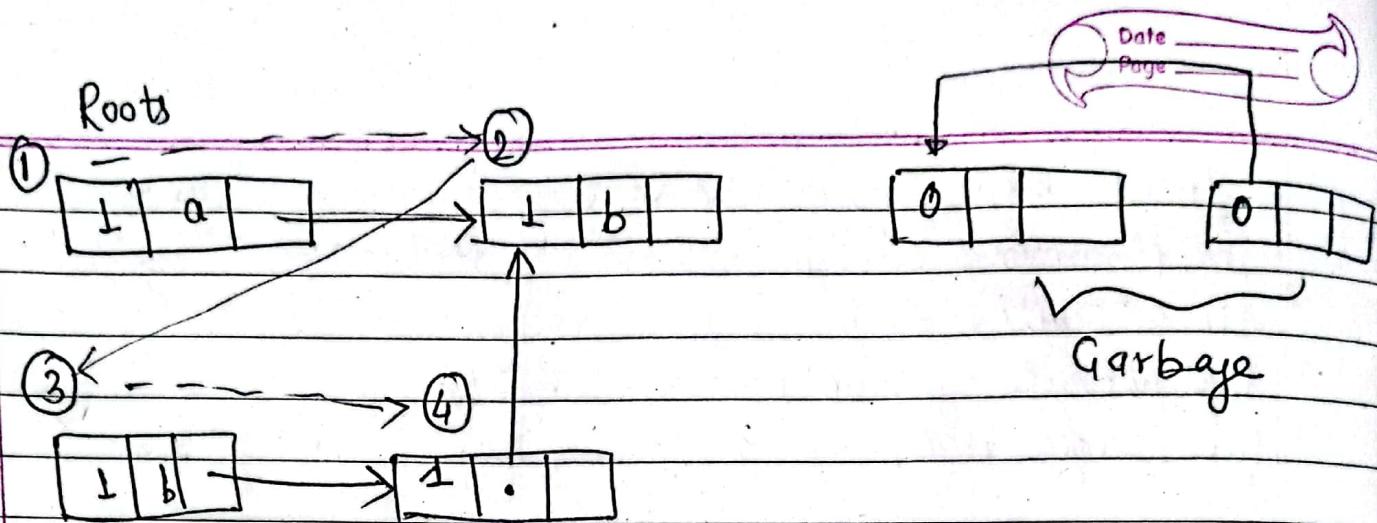


Fig: Example of Mark phase of Garbage Collection

Sweep Phase:

- In this phase, all of the inaccessible cells are returned to free storage.
- By end of mark phase, all accessible cells have been marked. Therefore, in sweep phase we can visit each cell in order (that is sweeping through memory). If a cell is unmarked, that it's inaccessible and can be linked to free list. If it's marked, it's accessible; we reset its mark bit (in preparation for next garbage collection) and go on to next cell.

Sweep phase:

for each cell C :

If C is marked then reset C's mark bit,
else link C into free list.

Name structure of Fortran:

→ Structure/organization of names in program.

① The primitives bind names to objects :

→ The primitive name structure (binding constructs) give meanings to the name.

→ The bindings have three functions; allocating a region of memory, initializing the region and binding the identifier to the address of that region. These functions are performed by FORTRAN declarations.

Eg: INTEGER I, J, K

② Declarations are non-executable:

→ Fortran declarations are often called non-executable to differentiate from executable statements such as assignments, GOTOs, IF-statements and subroutine calls.

→ Nonexecutable statements provide information for the compiler and the other preexecution processors of the program (e.g.: loader, linker).

→ Type in declaration (Integer) is used to determine amount of storage it must be allocated.

③ Optional variable declarations are dangerous:

- FORTRAN has an unusual convention that has been abandoned in almost all newer programming languages: automatic declaration of variables.
- In other words, if a programmer uses a variable but never declares it, the declaration would be automatically supplied by compiler.
- Suppose a programmer intended to type $COUNT = COUNT + 1$ but accidentally typed $COUNT = COUMT + 1$, this would NOT result in error. Instead, COUMT would automatically declared by compiler and would contain a garbage value.

④ Environments determine Meanings!

- The meaning depends upon the context in which it is stated.
- e.g.: $X = COUNT(1)$ may have many meanings, depending on definitions of its constituent name.
- X might be real or integer.
- $COUNT$ might be an array or function.
- The context of a statement is set of definitions that are visible to that statement. It is also called an environment.

⑤ Variable Names are local in scope:

- fortran programs are divided into number of disjoint subprograms that can be developed independently.
- Due to this they have disjoint environment containing the variables declared in that subprogram making variables local in scope.

⑥ Subprogram names are global in scope:

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

Eg: C Main Program

INTEGER X

— (*)

CALL R(2)

CALL S(X)

END

SUBROUTINE R(N)

REAL X, Y

;

CALL S(Y)

;

END

— X*

SUBROUTINE S(N)

INTEGER X

;

END

— X**

→ The subprograms have a global scope and local variables inside subprogram have local scope visible only inside that subprogram.

→ Here, the scope of X marked (*) is the main program and scope of binding X (**) is subroutine R. Similarly, scope of binding Y (*) is R and scope of binding Y (**) is S.

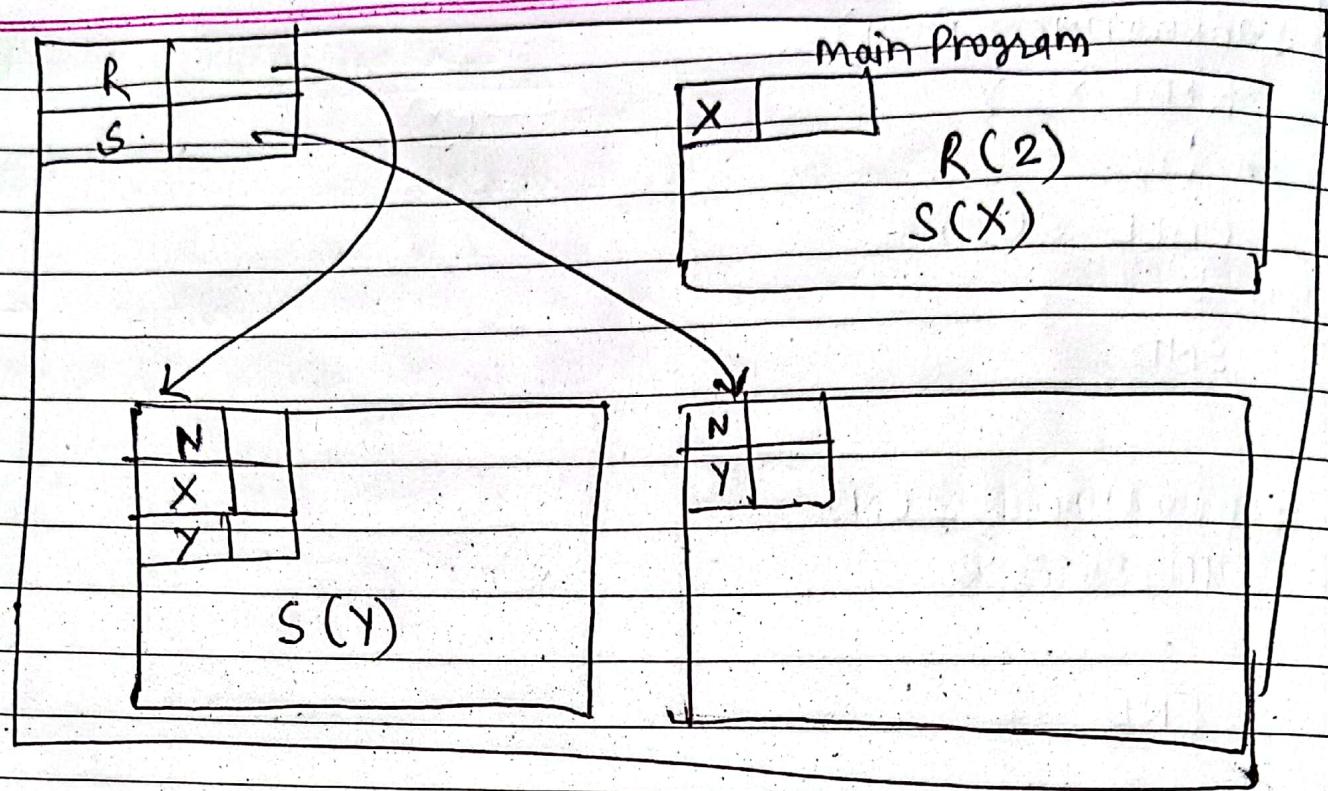


Fig: Contour Diagram showing scope of variables

Contour diagrams are valuable aid to visualizing scopes, particularly in languages with more complicated scope rules than Fortran.

Fortran Common Blocks:

- Allows sharing data between subprograms.
- COMMON blocks permit subprograms to have common variables, that is, to share data areas.

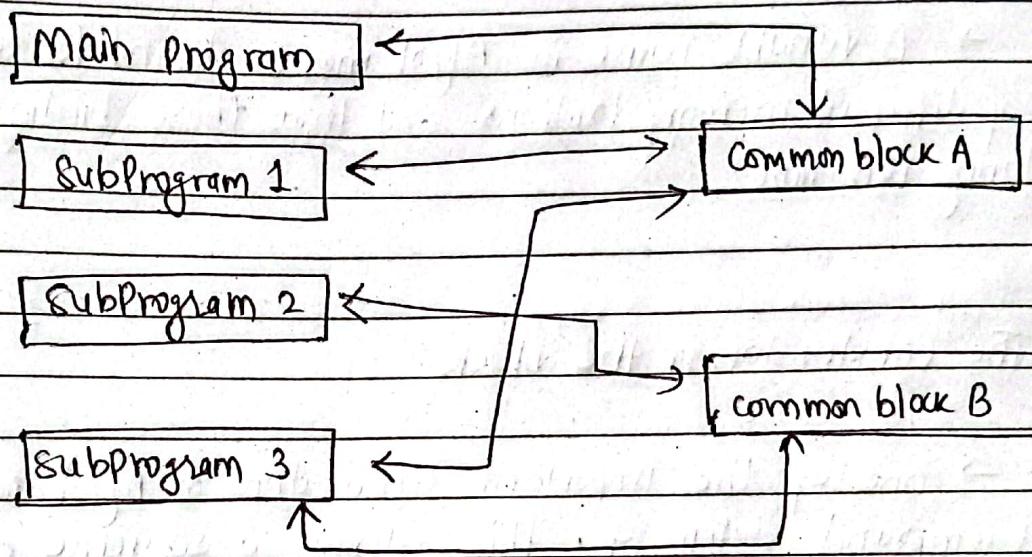


fig: Subprogram communication through COMMON block.

- COMMON permits Aliasing which is dangerous.

Algol: Name structure:

① The primitives bind names to objects :

→ Primitive name structures are the declarations that define names by binding them to objects .

→ A variable name in Algol may be bound to a number of different memory locations and that these bindings can change during run-time .

② The constructor of the block :

→ one of the important contributions of Algol was the idea of a compound statement . This allows a sequence of statements to be used wherever one was permitted . For instance , although one statement would normally form the body of a for-loop such as :

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

Several statements can form the body if they were surrounded by begin-end brackets :

for $i := 1$ step 1 until N do

begin

end

③ Blocks define nested scopes:

→ Programs can define any number of scopes nested to any depth; this is accomplished with a block.

begin

real x, y ;

real procedure cash(x);

real x ;

$cash := x$

procedure $f(y, z)$

integer y, z ;

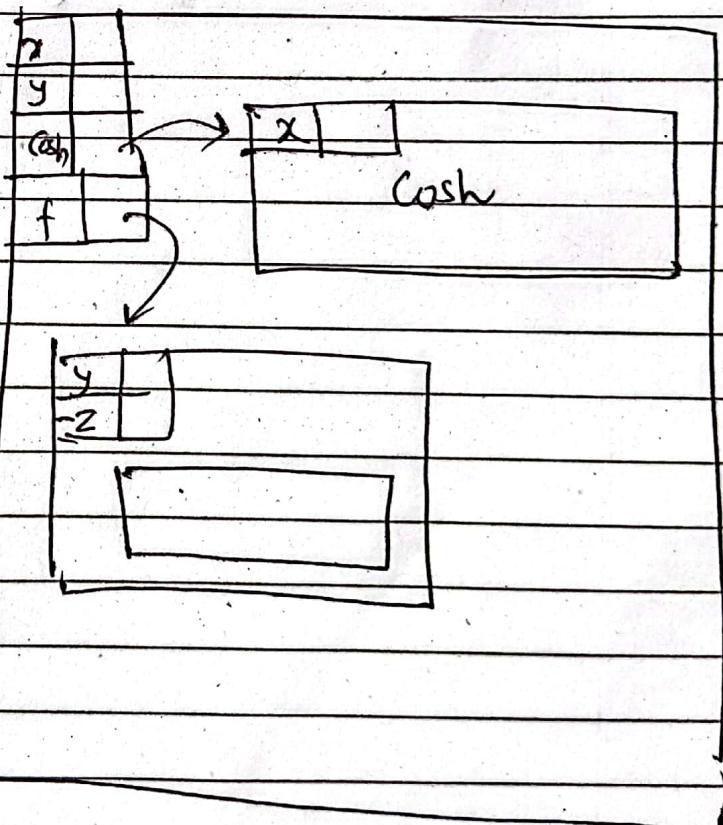
begin

...

...

end

end



(4) Static and Dynamic Scoping:

- There are two scoping rules that can be used in block-structured languages - static scoping and dynamic scoping.
- In static scoping, a procedure is called in the environment of its definition.
- In dynamic scoping, a procedure is called in the environment of its caller.
- Algol follows static scoping.

Chapter-1

Pseudocode :

→ Instruction code that is different than that provided by a real machine. It offered floating point and indexing which was actually not supported by earlier generation of computer. It provided entirely new instruction set not offered by real hardware.

Design of Pseudo Code

- ① Instruction format
- ② orthogonal Design
- ③ Comparisons
- ④ Moving
- ⑤ Indexing:

+1	150	000	280	→ <i>read from</i>
+6	xxx	iii	zzz	$(x_i \leftarrow z)$
-6	xx	yy	ii	$(x \rightarrow y_i)$

- ⑥ Input output

+8 → read data
 -8 → print data

→ *Chable to*

Phenomenology of Programming language:

- ① Tools are ampliative and reductive
- ② Fascination and fear are common reactions to new tool
- ③ With mastery, objectification becomes embodiment
- ④ Programming languages influence focus and Action

Small Talk

$p \rightarrow \text{key}$ $\ell \rightarrow \text{list}$

```
(defun amoc ( p l )
  (cond ( eq car (caar l) p ) car (cdar l)
        ( t (amoc p (cdr l)) ) ) )
```

This amoc finds value of key p in a list l.

(Q) Evaluate lisp expression:

@ $\frac{1}{2} \sqrt{\pi r^2 - l^2}$

$$= (\text{times} (\text{quotient} l 2)$$

$$(\text{sqrt} (\text{difference} (\text{times} (\text{pi} (\text{expt} r 2))) \\ (\text{exp} l 2))))$$

$$\textcircled{2} \quad -b - \frac{\sqrt{b^2 - 4ac}}{2a}$$

~~E quotient (times 2 0)~~

~~C quotient (minus difference (minus b) (sqrt (difference
(expt b 2) (times 4 a c)))) (times 2 a)~~

Small Talk

① objects can be instantiated:

anotherScribe ← pen newAt: 200 @ 800

This sends message 'newAt: 200 @ 800' to class 'pen' which creates and returns a new pen located at co-ordinates (200, 800) and pointing to right. The name 'anotherScribe' is bound to this new object.

② classes can be defined:

→ Class definition has a two-dimensional arrangement;

class name	box
instance variable names	loc tilt size scribe
instance messages and methods	
shape //	
scribe penup ; goto : loc ; turnTo : tilt ; pendn . 4 timesRepeat : [scribe go : size ; turn : go]	
show //	
scribe color ink .	
self shape	
erase //	
scribe color background	
self shape	
grow : amount //	
self erase	
size ← size + amount	
self show	

Fig: Definition of class box

③ Classes can also respond to messages:

→ class methods (static) Eg: newAt

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

newAt: initiallocation | newBox |

newBox ← self new.

newBox setloc: initiallocation size:100 tilt:0 scribe: pennew
newBox

instance messages and methods }

len	6
c.d.	-
loc	-
tilt	-
size	-
scribe	-

box

len	6
c.d.	-
loc	-
tilt	-
size	-
scribe	-

len	4
c.d.	-
x	-
y	-

Point

500@600

500

200

fig: object representation of box.

Name structure of Fortran:

→ Structure/organization of names in program.

① The primitives bind names to objects :

→ The primitive name structure (binding constructs) give meanings to the name.

→ The bindings have three functions; allocating a region of memory, initializing the region and binding the identifier to the address of that region. These functions are performed by FORTRAN declarations.

Eg.: INTEGER I, J, K

② Declarations are non-executable:

→ fortran declarations are often called non-executable to differentiate from executable statements such as assignments, GOTO, If-statements and subroutine calls.

→ Nonexecutable statements provide information for the compiler and the other preexecution processors of the program (e.g: loader, linker).

→ Type in declaration (Integer) is used to determine amount of storage it must be allocated.

③ Optional variable declarations are dangerous:

- FORTRAN has an unusual convention that has been abandoned in almost all newer programming languages: automatic declaration of variables.
- In other words, if a programmer uses a variable but never declares it, the declaration would be automatically supplied by compiler.
- Suppose a programmer intended to type $\text{COUNT} = \text{COUNT} + 1$ but accidentally typed $\text{COUNT} = \text{COUMT} + 1$, this would NOT result in error. Instead, COUMT would automatically declared by compiler and would contain a garbage value.

④ Environments determine Meanings!

- The meaning \Rightarrow depends upon the context in which it is stated.
- E.g.: $X = \text{COUNT}(1)$ may have many meanings, depending on definitions of its constituent name.
- X might be real or integer.
- COUNT might be an array or function.
- The context of a statement is set of definitions that are visible to that statement. It is also called an environment.

⑤ Variable Names are local in scope:

- fortran programs are divided into number of disjoint subprograms that can be developed independently.
- Due to this they have disjoint environment containing the variables declared in that subprogram making variables local in scope.

⑥ Subprogram names are global in scope:

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

Eg: C Main Program

INTEGER X

— ①

CALL R(2)

CALL S(X)

END

SUBROUTINE R(N)

REAL X, Y

:

CALL S(Y)

:

END

X*

SUBROUTINE S(N)

INTEGER X

:

END

X**

→ The subprograms have a global scope and local variables inside subprogram have local scope visible only inside that subprogram.

→ Here, the scope of X marked (*) is the main program and scope of binding X (**) is subroutine R. Similarly, scope of binding Y (**) is R and scope of binding Y (***) is S.

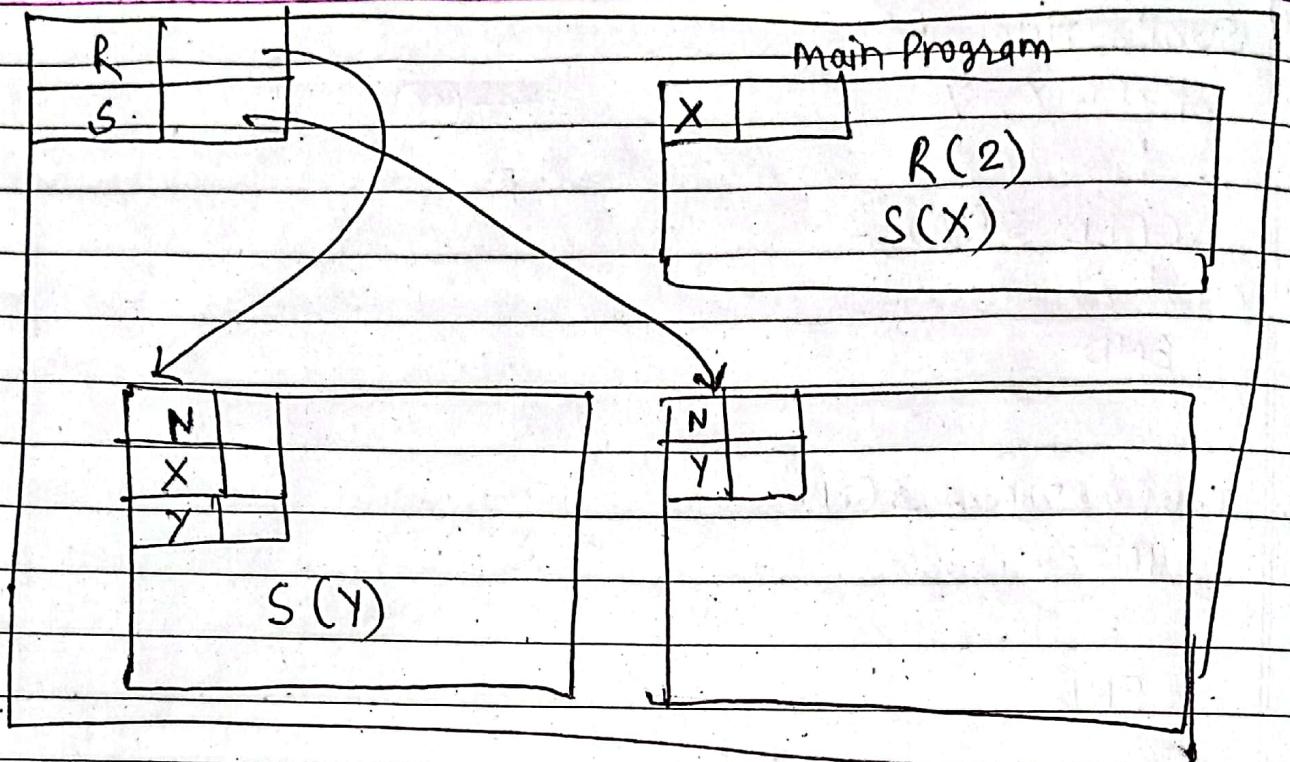


Fig: Contour Diagram showing scope of variables

Contour diagrams are valuable aid to visualizing scopes, particularly in languages with more complicated scope rules than Fortran.

Fortran Common Blocks:

- Allows sharing data between subprograms.
- COMMON blocks permit subprograms to have common variables, that is, to share data areas.

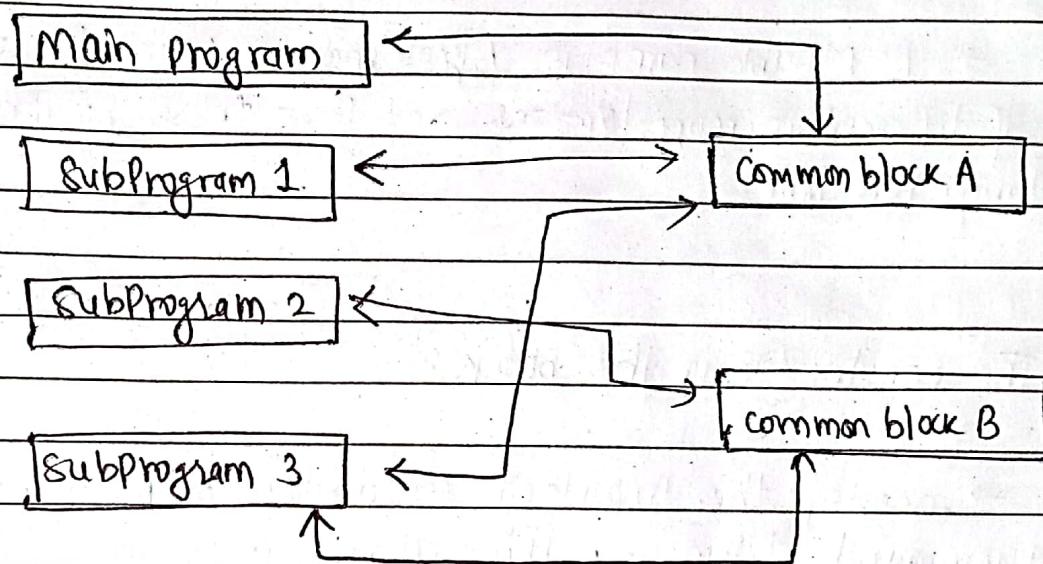


Fig: Subprograms communication through COMMON block

- COMMON permits Aliasing which is dangerous;

Algol: Name structure:

① The primitives bind names to objects:

→ Primitive name structures are the declarations that define names by binding them to objects.

→ A variable name in Algol may be bound to a number of different memory locations and that these bindings can change during run-time.

② The constructor is the block:

→ one of the important contributions of Algol was the idea of a compound statement. This allows a sequence of statements to be used wherever one was permitted. For instance, although one statement would normally form the body of a for-loop such as:

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

Several statements can form the body if they were surrounded by begin-end brackets:

```

for i := 1 step 1 until N do
begin
    - - -
    - - -
end

```

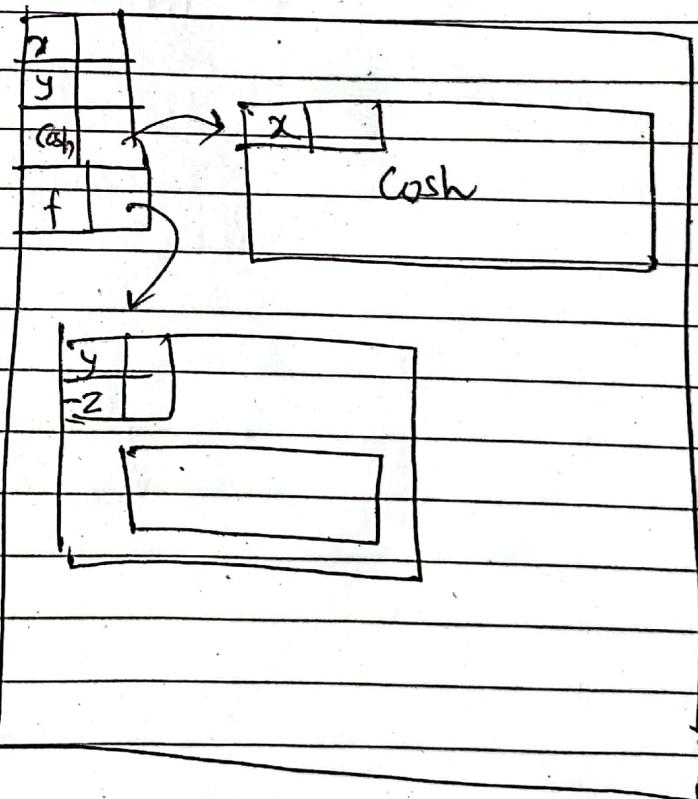
(3) Blocks define nested scopes:

→ Programs can define any number of scopes nested to any depth; this is accomplished with a block.

```

begin
    real x, y;
    real procedure cash(x);
        real x;
        cash := x
    end
procedure f(y, z)
    integer y, z;
    begin
        ...
    end
end

```



④ Static and Dynamic Scoping:

- There are two scoping rules that can be used in block-structured languages - static scoping and dynamic scoping.
- In static scoping, a procedure is called in the environment of its definition.
- In dynamic scoping, a procedure is called in the environment of its caller.
- Algol follows static scoping.

Chapter-1

Pseudocode :

→ Instruction code that is different than that provided by a real machine. It offered floating point and indexing which was actually not supported by earlier generation of computer. It provided entirely new instruction set not offered by real hardware.

Design of Pseudo Code

- ① Instruction format
- ② orthogonal Design
- ③ Comparisons
- ④ Moving
- ⑤ Indexing:

+1	150	000	280	→ <i>assignment</i>
+6	xxx	iii	zzz	$(x_i \leftarrow z)$
-6	xxx	yyg	iii	$(x \rightarrow y_i)$

- ⑥ Inputoutput

+8 → read data
-8 → print data

Phenomenology of Programming language:

- ① Tools are ampliative and reductive
- ② Fascination and fear are common reactions to new tool
- ③ With mastery, objectification becomes embodiment
- ④ Programming languages influence focus and Action

① Write a program to find out sum & average of first 10 odd natural numbers.

INTEGER i, sum

REAL avg

DO 1000 i = 1, 20

sum += i

sum = sum + i

i

INTEGER i=1, sum=0

REAL avg

100 If i.LT.20 GOTO 200

300 Print * "Sum is", sum

Print * "Avg", sum / 10.0

GOTO 300

200 sum = sum + i

i = i + 2

GOTO 100

300 Print * "Successfully ended"