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, ---an)

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

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

Example: (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:

```
(cond
  ((null x) 0)
  ((eq x y) (f x))
  (t (g y)))
```

In Algol:

```
if null(x) then 0
elseif x=y then f(x)
else t(g(y))
```

Why is everything a function application in LISP?

→ Simplicity Principle. — If there is only one basic mechanism in a language, the langague 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 orginally 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 USP represents both programs and data in the same way is of utmost importance (and almost unique among programming languages).

(4) USP is often interpreted:

→ Most USP systems provide interactive interpreters.
→ We interact with USP 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 USP 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 whichever 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 donot 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 30000 (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 ~~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 donot 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 30000 (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 cars and cdrs become large for complex selections. So, USP provides abbreviation.

For example: an expression such as:

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

(caddr DS)

The composition of cars and cdrs 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 accen the last name:

(car DS) → (Don Smith)

(cdar DS) → (Smith)

(cadar DS) → Smith

a → (Don Smith)
d → (Smith)
a → Smith

(caddar DS)

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

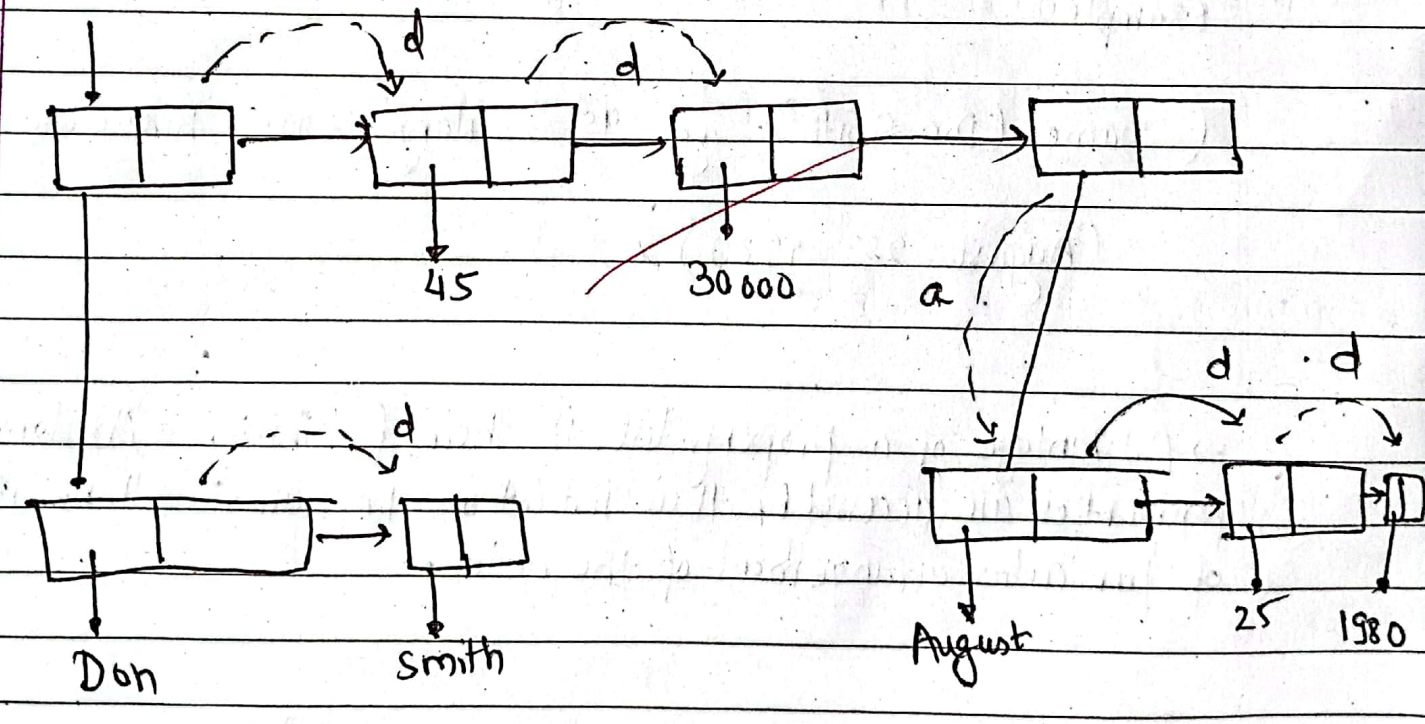


fig: Walking down a list structure.

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

Ans: **Property list:**

The method of representing information in the form $(p_1\ v_1\ p_2\ v_2 \ldots p_n v_n)$ 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:

(name (Don Smith) age 45 salary 30000 hire-date (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 (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 (eq (car x) p). If first element of x is p, the value of (getprop p x) is second element of x i.e. (cadr x).

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

The function in Lisp would be:

~~(getprop p x) =~~

Eg: (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 (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 (eq (car x) p). If first element of x is p, the value of (getprop p x) is second element of x i.e. (cadr x).

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

The function in LISP would be:

~~(getprop p x) =~~

Eg: (name Ram age 25 salary 30000)

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

ie.

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

(2) Association lists:

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 LISP 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:

$$((a_1, v_1)\ (a_2, v_2)\ (a_3, v_3) \ldots (a_n, v_n))$$

As for property lists, the ordering of information in 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 lists:

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 LISP 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:

$((a_1, v_1)\ (a_2, v_2)\ (a_3, v_3) \ldots (a_n, v_n))$

As for property lists, the ordering of information in 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))
)
```

The function that does the forward anociation is called assoc-

for example:

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

(assoc 'age DS) gives 45
```

(8) Explain the flowing in lisp 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) )
       ( (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 an optional else statement. ~~then~~

If can only accept one statement for then and else clau

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: `( mapcar '1+ '(2 4 6) )`

⇒ (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)) ⇒ 10

⑦ How hierarchial structures are processed in LISP?

Ans:

Small Talk (Object oriented Programming)

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

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

→ 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.

(3) That is, if it is not defined in the class, we must look ~~into~~ 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 donot explicitly deallocate activation records.

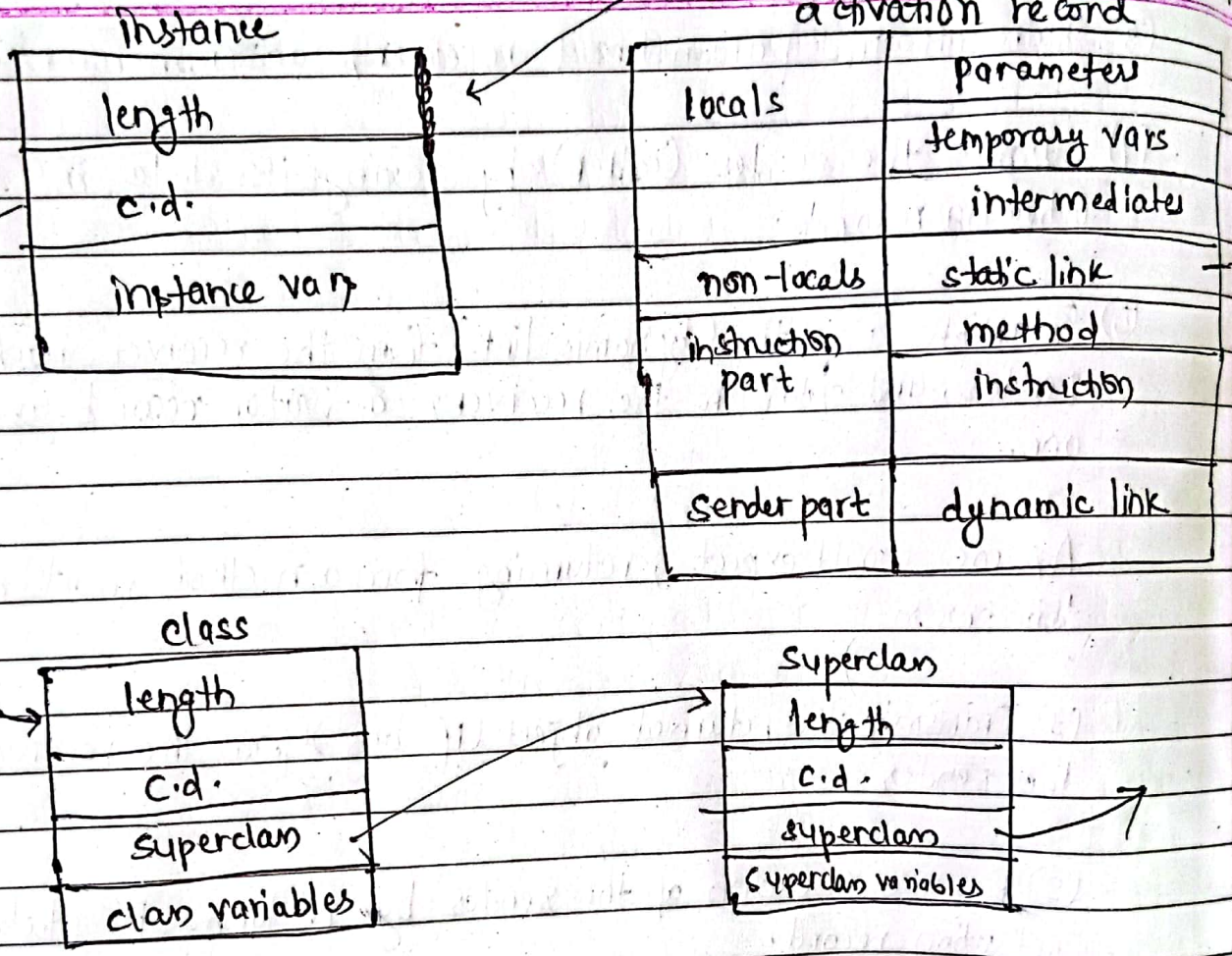


Fig: Parts of an Activation Record

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

Ans: Menages are enentally procedure invocations, although the formats allowed for menages are a little more flexible. In most languages parameters are surrounded by paranthesies 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.

first → The smalltalk has three formats for menages: / menage templates:

(1) Keywords for parameterless menages (eg: B1 show)

(2) Operators for one-parameter meneges (eg: x + y)

(3) Keywords with colons for menages with one or more parameters
eg: Scribe grow: 100

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

B1 show

~~omitting the color~~

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

Unfortunately, they could require writing arithmetic expressions in an uncommon way. for example: to compute $(x+2) \times y$ we would have to write:

(x plus: 2) 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 + 2 * y

The object named x is sent the menage +2, and object resulting from this is sent the menage * y. Thus, this expression computes (x+2)*y. The smalltalk doesnot obey usual precedence rules (a concession to Regularity principles).

(4) How is Activation Record represented in Small Talk?

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 menage invoking this method.

The sender part is a dynamic link, ie., 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.

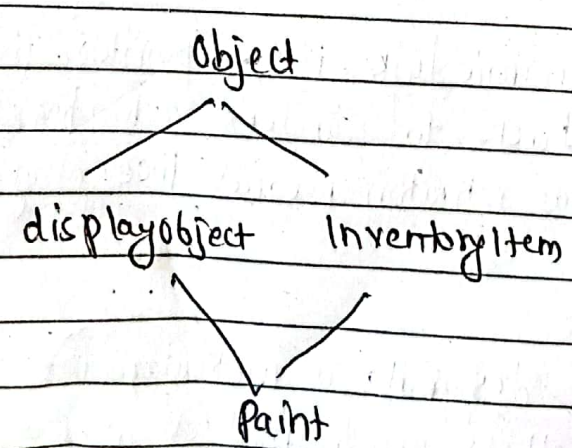
The instruction part must designate a particular instruction in a particular method. Since, methods are themselves objects (instances of class method), a two co-ordinate system is used for identifying instructions.

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.

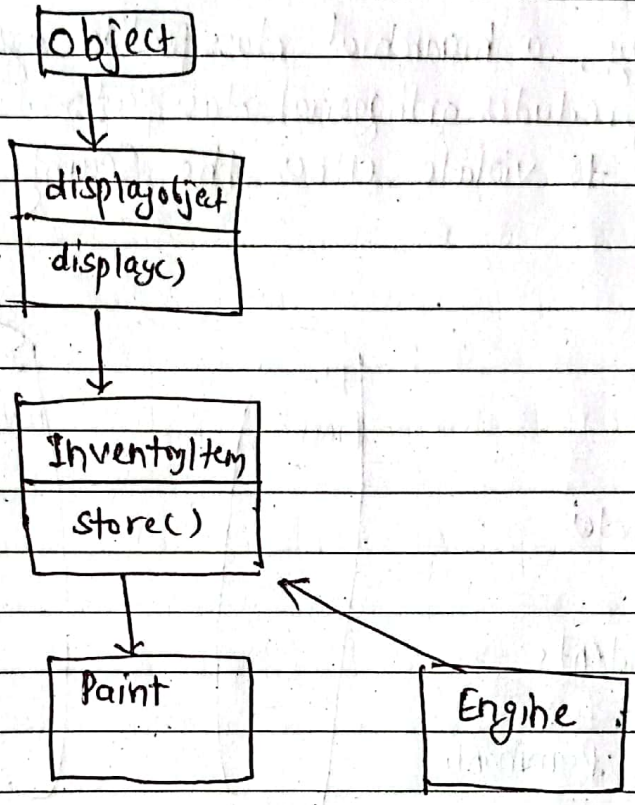
Ans: Smalltalk organizes classes into a hierarchy; each class has exactly one immediate superclass.

~~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.



[Not Possible in Smalltalk]

To overcome this, we can choose either displayobject or InventoyItem to be superclass of other. Suppose, we choose displayobject.



This cause one problem; some classes like Engine are InventoyItem but not displayobject. They will have unwanted display() method. This is violation of Security Principle.

Another solution is to repeat data/methods of InventoyItem in Paint and making it inherit from displayobject only. This will vidate 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.

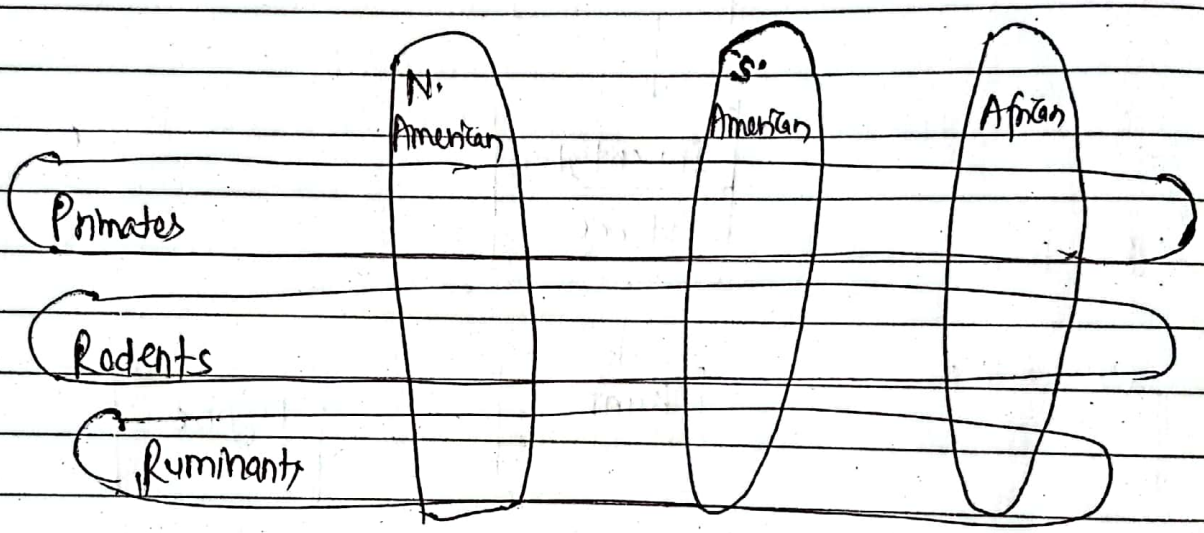


Fig: Example of orthogonal classification.

Seen

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 fortan 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[middle] = sought then location := middle
   else location := 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 step2 unil N×M do
    inner[i] := outer[N×M - i];
```

③ Nested statements:

compound statement → 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 ie: 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.

Eg: Integer array NumberofDays [-100 : 200]

We can the 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 analyster.

* Dynamic Arrays:

→ size of arrays may not be known while declaring.
→ In language with static arrays like FORTRAN, we dimension the array to 900 elements to solve this. But if number of elements are far fewer, it causes wastages of space. If elements are more than 900, 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.

control structure :

Procedure

```
integer procedure simple (param.):
    value param;
    integer param;
    param := param + 1;
```

→ In Algols, 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 × fac(n-1);
```

Pass By Value:

→ Value of the actual parameter it coped 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 anignment to n doesnot 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;
```

Since, both the array to be averaged (A) 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. Eg: 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:

→ Passing 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 := 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 := 1 step 1 until u do
            s := s + aK
        sum := s
    end;
```

Each time through loop, each time K (or i) is changed aK (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.

→

Pan-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 := 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] := temp;
```

let; i = 2 x[2] = 5

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

So After call;
i=5, x[5] = 2, x[2]=5

So, swap(i,j), swap(j,i), swap(A[i], j) works but swap(i, A[i]) doesnot work.

Syntactic STructures:

(1) Machine Independence and Portability:

→ Algol focuses on making itself machine independent hence it adopted free format ie. 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 are 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;

(6) 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.

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

(7) Hardware representations:

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

(8) Algol solved some problems of fortran lexics:

→ In Algol, keywords are boldfaced 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 (preceddy 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 Arbitary restrictions are eliminated:

Algol adheres to zero-one-infinity principle. for eg: 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".

Dangling else problem solution:

Algol's syntactic conventions had some problems. Eg:

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.

(4) 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 Algol.

→ 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 ("+" or "function") or a category of literals (like integer).

A vertical bar | indicates choice. (or operator)

BNF of decimal fraction:

terminal → "." ← non-terminal

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

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

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

Recursion is used for repetition:

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

Extended BNF is more descriptive:

⇒ In BNF, we represent idea of sequence one or more digits through recursion.

⇒ It would be more convient 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 <letter> followed by any number of <letter>s or <digit>s 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 donot need to separately define <alphanumeric identifier> if it is used only in one place and we donot have to give it.

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

This can be more pictoral as:

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

for <integer> :

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

We can use alternative notation :

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

| <unsigned integer>

This is better but we want to express the idea that <unsigned integer> is optionally preceeded by a '+' or '-'. The square bracket is often used for this purpose.

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

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

These three formats fits Zero-one-infinity Principle since they : 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)