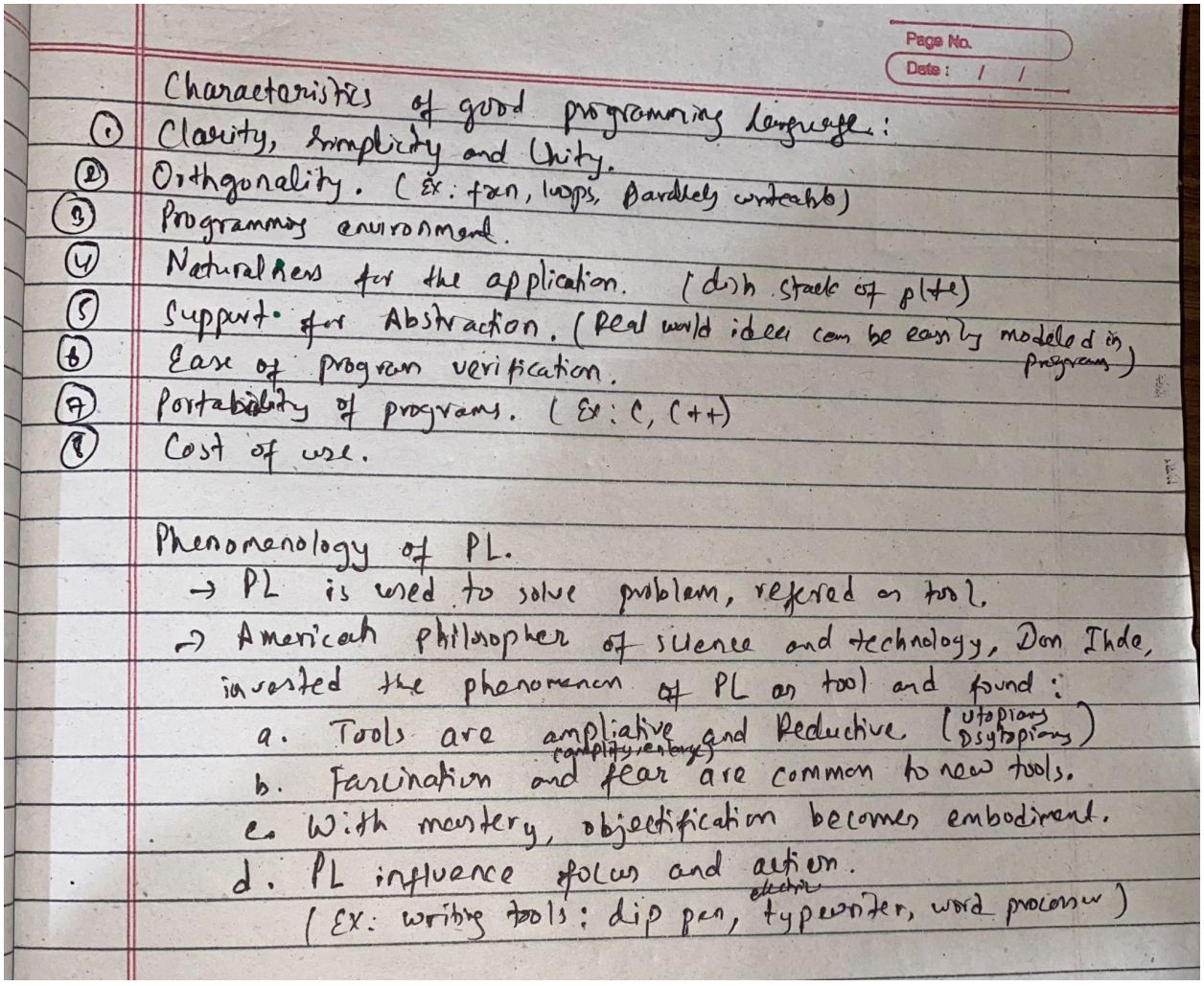
Reason for studying Principle of programming languages :

- Increase capacity to express ideas.
- Improve background for choosing appropriate programming language.
- Helps us to Better understand the significance of implementation.
- And soon.



Page No.
Date : / /

Characteristics of good programming language :
① Clarity, simplicity and Unity.
② Orthogonality. (Ex: for, loops, Parallely writeable)
③ Programming environment.
④ Naturalness for the application. (dish stack of plate)
⑤ Support for Abstraction. (Real world ideas can be easily modeled in programs)
⑥ Ease of program verification.
⑦ Portability of programs. (Ex: C, C++)
⑧ Cost of use.

Phenomenology of PL.
→ PL is used to solve problem, refered as tool.
→ American philosopher of science and technology, Don Ihde, invested the phenomenon of PL as tool and found :
a. Tools are ampliative (amplify, enlarge) and Reductive (utopians dystopians)
b. Fascination and fear are common to new tools.
c. With mastery, objectification becomes embodiment.
d. PL influence focus and action.
(Ex: writing tools : dip pen, typewriter (electric), word processor)

Programming Paradigms :

1. Imperative programming :
   Imperative programming focuses on describing a sequence of steps to achieve a goal. It emphasizes changing program state through statements.
   Ex: Fortran,C, Python.

   # Python example
   x = 5
   y = 10

result = x + y

2. Functional programming:
   Functional programming treats computation as the evaluation of mathematical functions. It emphasizes immutability and avoids changing state or mutable data.
   Ex: JavaScript, Clojure, Haskell.
   Code Ex: Higher Order Fxn in JS.

3. Object-oriented programming.
   Ex: C++, Java, Python.

4. Procedural Programming Language:
   Procedural programming organizes code into procedures or functions, which manipulate data and perform actions. It emphasizes modularity and reusability.
   Ex: C, Pascal.
   Code Ex : C

5. Event-Driven Programming Language :
   Ex : Javascript , Java

6. Logical (Declarative) Language :
   Logical programming focuses on specifying what needs to be done rather than how to do it. It uses formal logic to solve problems.
   Ex: Prolog

```
% Prolog example
parent(john, mary).
parent(jane, mary).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
```

Pseudo-code

- Instruction code
- Provided support for floating point and indexing. (which was not available previously).
- Pseudo-code interpeteres were used to convert the pseudo code instruction to machine level codes. As pseudo code has its own data and instruction set.
- Before Pseudo-code, programmer have to program individually for different computer by understanding their hardware specification; which was basically tough and tedious.

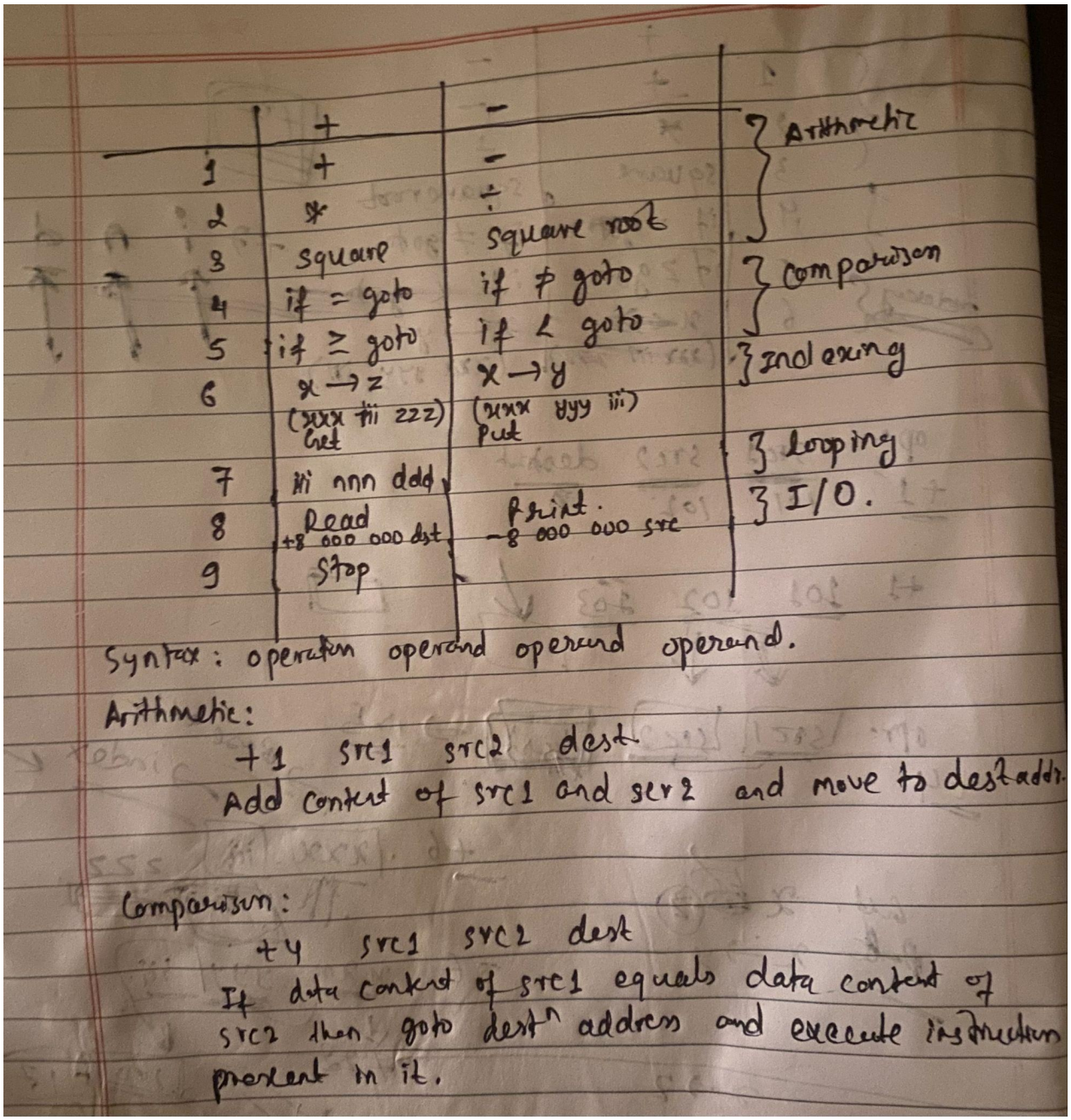
PseudoCode follows two basic Principles of Programming

- The Automation Principle

Automate mechanical, tedious, or error prone activities.

- The Regularity Principle
  Regular rules, without exceptions, are easier to learn, use, describe, and implement.

Symbolic Language Table



Indexing :

Need => Base address, index value.
So, 2 operands are occupied by them. Whereas third operand is the place where we stored the indexed element so it can be lateron used for other operations.

Two types of indexing :

1. Get x -> z
   +6 xxx iii zzz
2. Put x-> y
   -6 xxx yyyy iii

Here xxx = Base Address
Iiii = Index Address.

Ex: If there are 100 array elements beginning at location 250 in data memory. And location 050 contains 17.
Then +6 250 050 800
Means, The content of 267 (250 + 17) is move to location 800.

Similarly, -6 722 250 050
Means, the content of 722 is moved to location 267 (250 + 17)

Looping :

Looping through the element of array is frequently used.
What's needed ?

- Iterator variable (array index i)
- Upper bound (n)
- Address of beginning of loop (d)

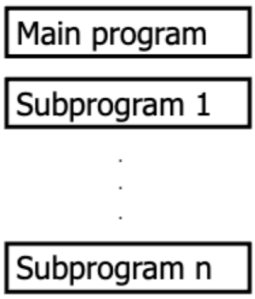
+7 iii nnn ddd

Chapter 2 : Fortran

Goal : (as compared to pseudo-code)

- To reduce programming cost.
- To increase efficiency

Fortan has the concept of subprograms. (ie. one main programs and zero or n sub-programs)



Name structure in fortran

As data structure organizes the Data, control structure organizes the control flow of the program, name structure organizes the names that appear in a program.

When we declare a new variable in a program:
INTEGER I
It bound the identifier I to a location in a memory. Here INTEGER specify the type of the variable, and I is the identifier.

Declarative constructs are the primitive name structure of FORTRAN.

In our pseudo-code we saw that the declarations performed three functions:

1. They *allocated* an area of memory of a specified size.
2. They attached a symbolic name to that area of memory. This is called *binding* a name to an area of memory.
3. They *initialized* the contents of that memory area.

These are three important functions of declarations in most languages, including FORTRAN. For example, the declaration

```
DIMENSION DTA (900)
```

causes the loader to allocate 900 words and to bind the name "DTA" to this area. A separate kind of declaration, called a DATA declaration, can be used for initialization. For example,

```
DATA DTA, SUM / 900*0.0, 0.0
```

would initialize the array DTA to 900 zeroes and the variable SUM to zero. FORTRAN does not require the programmer to initialize storage; this lack of initialization is a frequent cause of errors. Declarations and initializations are discussed in Section 2.3.

Multiple variables at once :
INTEGER I , J, K.

It will make entries in the symbol table for I, J, K. These entries will consist of location of these variables and their type.

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

Optional Variable declarations are dangerous. Fortan has this unusual convention which is avoided in newer programming languages.

Control Structure :

1. Low level Control Structure : IF Statement, GOTO Statement (if then else in newer programming language)
2. High Level Control Structure : DO-LOOP
3. Sub program control structure : CALL,RETURN

IF Statement :

Arithmetic IF-statement

- Example of machine dependence
  - `IF (a) n1, n2, n3`
  - Evaluate a: branch to
    - n1: if -,
    - n2: if 0,
    - n3: if +
  - CAS instruction in IBM 704
- More conventional IF-statement was later introduced
  - `IF (X .EQ. A(I)) K = I - 1`

Goto Statement :

GOTO

- Workhorse of control flow in FORTRAN
- 2-way branch:

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

- Equivalent to *if-then-else* in newer languages

Two types :
Computed Goto and Assigned Goto

GOTO (L1, L2, L3 … LN), I

The **computed** GO TO statement selects one statement label from a list, depending on the value of an integer or real expression, and transfers control to the selected one.

Example

Example: Computed GO TO

```
      ...
      GO TO ( 10, 20, 30, 40 ), N
      ...
10    CONTINUE
      ...
20    CONTINUE
      ...
40    CONTINUE
```

:

In the above example:

- If N equals one, then go to 10.
- If N equals two, then go to 20.
- If N equals three, then go to 30.

Assigned Goto:

ASSIGN 20 TO N

GOTO (20, 30, 40, 50), N

Do loop Syntax :

```
DO <label> VARIABLE= INTIAL_VALUE,FINAL_VALUE
        //
    <label> CONTINUE
END DO
```

Example : To print Cube root of first 10 natural numbers.

```
DO 100 I=1,10
        PRINT *,I**1.0/3
        100 CONTINUE
END DO
```

Sub-program Example:

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

.
.
CALL DIST(diff1, X1, Y1)
```

Pass by reference is a dangerous proposition in FORTAN.

In Fortran, subroutine arguments are typically passed by reference, meaning that when you pass an array or variable as an argument to a subroutine, you are passing a reference or a memory address to the data rather than making a copy of the data. This is more memory-efficient than copying large arrays but can lead to side effects if not used carefully. Here's why it can be potentially dangerous:

Unintended Modification of Data: If you pass an array to a subroutine and that subroutine modifies the array's elements, it can change the original data in the calling program. This can lead to unexpected changes in the data, making code harder to understand and debug.

Lack of Encapsulation: Fortran does not provide strong encapsulation for subroutine arguments. In other words, subroutines have access to the original data and can modify it directly, potentially leading to errors that are difficult to trace.

Concurrency Issues: In parallel or multithreaded programs, when multiple threads or processes access the same data through subroutine calls, it can result in data races and unpredictable behavior.

Data Structure :

1. Scalar Data Types (INTEGER, REAL,)
2. Arrays Data Types (DIMENSION; arrays are declared using DIMENSION).

Things violated by Fortan IV :

1. IF Statement.
2. Security Principle (COMMON, EQUIVALENCE).
3. Not portability of programs. (limited to particular machine only).
4. Automatic creation of variables (if not defined aslo, fortran would create it).

COMMON:

COMMON /block-name/ var1, var2
Used for sub-routines and variables.

EQUIVALENCE var1, var2

Chapter 3 : ALGOL (Algorithmic Language)

History and Motivation :

- There was a need for a universal language . ie. a language that can run on various machines independent of hardware (i.e the portability of programs; and to make general language).
- Emphasized on using mathematical notations and increased readability of the program using various context free grammar and regular grammar expressions.
- In the Paris conference, John Backus introduced some notations for the language, which was denied by Peter Naur. Later on they worked together and created Backus Naur Form Notation and was implemented from ALGOL 58 and hence to ALGOL 60.
- The BNF was so efficient that it was reported to make notation for the entire language within 15 pages; whereas other languages had taken 100 to thousands of pages for notation of programming languages. It was possible because BNF are concise, easy to read-understand and easy to implement the ALGOL expressions from it.
- ALGOL 60 was designed; and it was known for its generality and elegance.
- Follows the “portability principle”.

Structural Organization :

- ALGOL is hierarchically structured;and programs are composed of a nested environment decreasing the need of GOTO statements.
- Uses Stack as the fundamental block for the data structure
- Concept of “Data hiding and Data sharing” came from ALGOL.

Algol follows zero-one-infinity principle:

- Fortran case : 3 (dimension array), 6 (characters identifier), 19 (They were at most 19 continuation cards).
- So programmer have to remember 3, 6, and 19.
-
- But in algol, arrays are generalized. One can define the bound for the array itself ie. A[m:n] from A[m] to A[n].
- Stacks allowed dynamic array. In fortran, there will be always wastage of memory. For ex: Dimension (90), if only given for 10 ; 80 will be wasted; sometimes overflow which will only be found on run-time error.
- ALGOL has strong typing. Cannot mix float with integer and integer with float and soon.
- And doesnot voliate security principle as like FORTAN-IV (COMMON and EQUIVALENCE).

Algol as major milestone :

- Although it didn't get widespread as like FORTAN, but has given major milestone in programming history.
- Eventually, all programming language become ALGOL Like i.e. block structured, nested, recursive,
- Commercial failure but Scientific triumph.

BNF (Backus Naur Form) and EBNF (Extended Backus Naur Form)

They are general notation formats used for describing the syntax of programming languages and other formal grammars. However, they have been used in the context of ALGOL and many other programming languages for specifying their syntax. Let's briefly discuss BNF and EBNF:

Backus-Naur Form (BNF):

BNF is a notation used for formally describing the syntax of programming languages and other formal grammars.
It was developed by John Backus and Peter Naur for the definition of the ALGOL 58 programming language.
BNF uses a set of production rules to describe the structure of valid sentences or programs in a language.
A BNF rule consists of a non-terminal symbol (representing a syntactic category) followed by the "::=" symbol and a sequence of terminal and non-terminal symbols, often enclosed in angle brackets or quotes.
Example BNF rule for a simple arithmetic expression in ALGOL-like notation:

```
<expression> ::= <term> | <expression> '+' <term> | <expression> '-' <term>
```

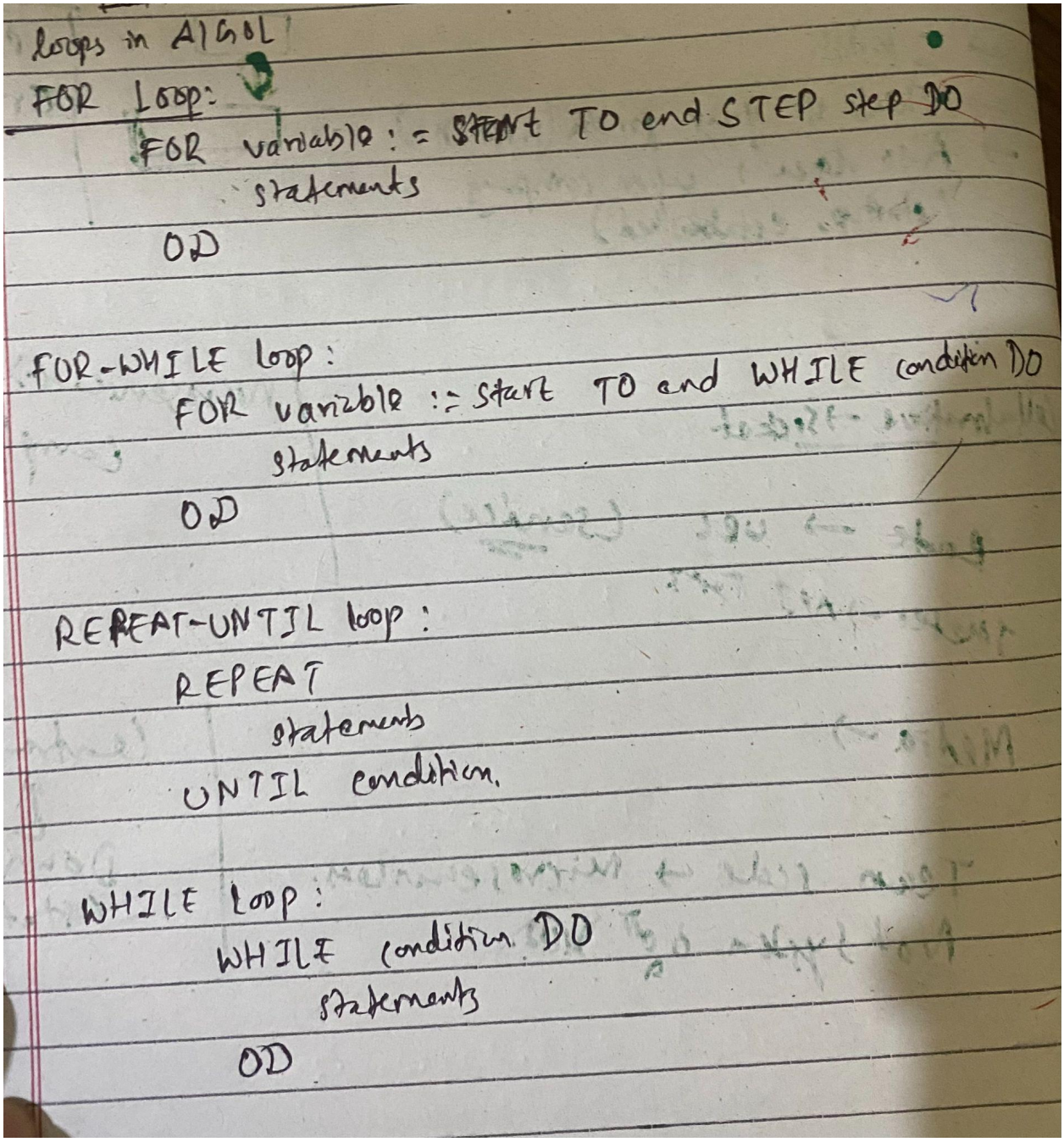
BNF are used to describe Context free Grammars .

Extended Backus-Naur Form (EBNF):

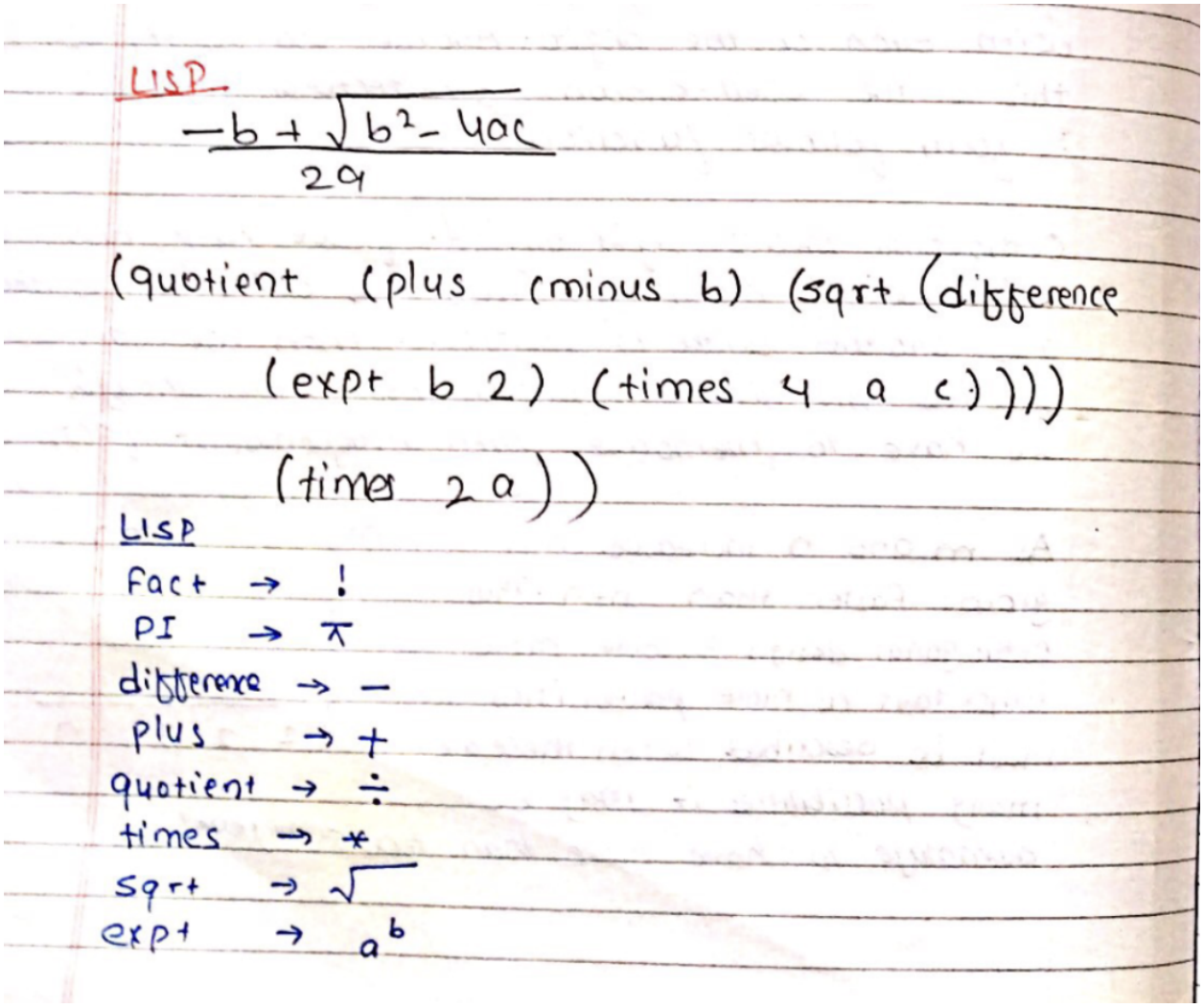
EBNF is an extension of BNF that includes more expressive notation for describing syntax. It was introduced to make it easier to specify complex grammars.
EBNF uses symbols like [ ], { }, and | to denote optional elements, repetition, and alternatives.
EBNF is widely used in defining the syntax of programming languages and is more readable and concise than traditional BNF.
Example EBNF rule for the same arithmetic expression:

```
<expression> ::= <term> { ('+' | '-') <term> }
```

EBNF can be used to describe regular grammars as well as context-free grammars.Regular grammars are a simpler type of formal grammar that can be described using EBNF, especially when it comes to regular expressions used in lexical analysis.



Chapter 4 : LISP



In Lisp, a property list (often abbreviated as plist) is a data structure used to associate properties or attributes with symbols.

What is LISP & Explain CAR and CDR :

LISP (short for "LISt Processing") is a programming language that was developed in the late 1950s and is known for its unique and influential features. It is one of the oldest programming languages still in use today and is particularly well-suited for tasks that involve symbolic processing and artificial intelligence.

One of the distinctive features of LISP is its use of linked lists as a fundamental data structure. In LISP, a list is represented as a series of nested pairs of parentheses, with elements

separated by spaces. The two most important operators in LISP for working with lists are CAR and CDR.

CAR: The CAR operator extracts the first element (the "head") of a list. It stands for "Contents of the Address part of Register."
CDR: The CDR operator extracts the rest of the list (everything except the first element). It stands for "Contents of the Decrement part of Register."
Here's a simple example in LISP to illustrate how CAR and CDR work:

lisp Copy code

```
(setq myList '(1 2 3 4 5))
```

In this example, we define a list called myList containing the integers 1 through 5.

Now, let's use CAR and CDR to extract elements from myList:

lisp Copy code

```
(car myList)
```

This will return 1 because it extracts the first element of the list.

lisp Copy code

```
(cdr myList)
```

This will return (2 3 4 5) because it extracts everything except the first element of the list.

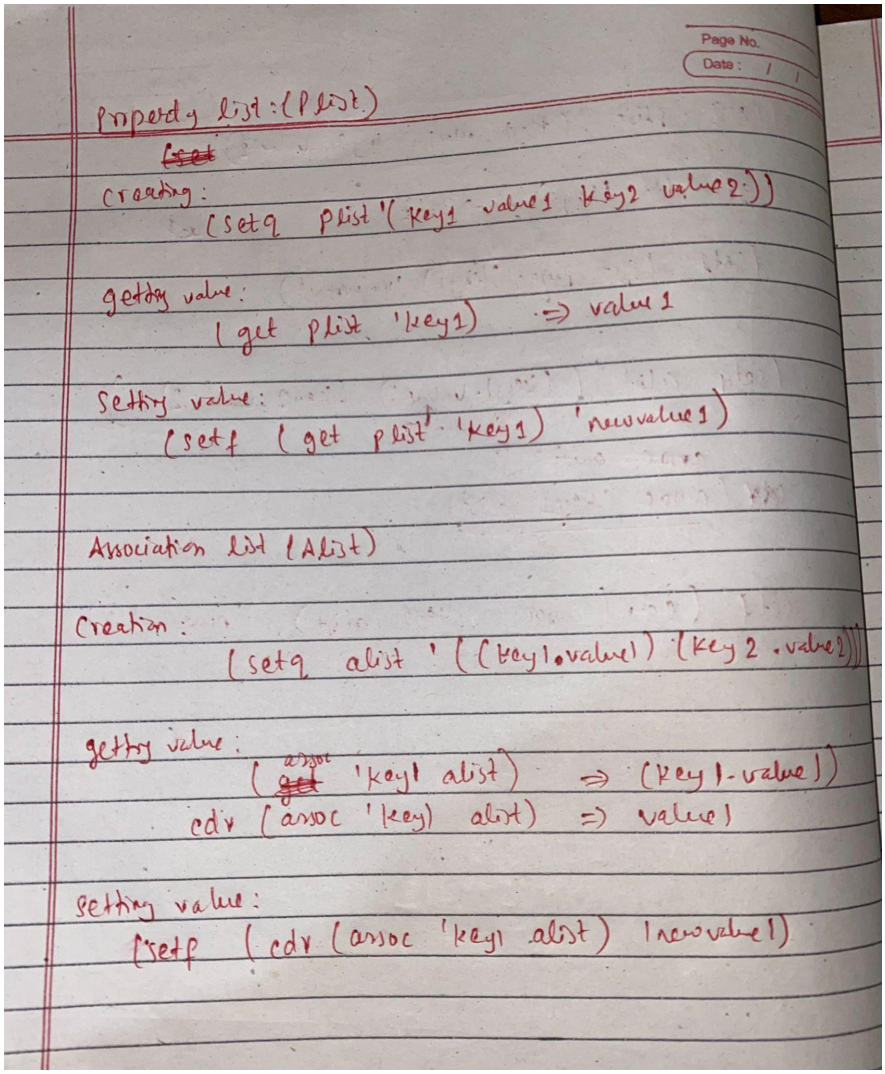
You can also nest CAR and CDR operations to access deeper elements of nested lists. For example:

```lisp
(setq nestedList '((1 2) (3 4) (5 6)))
(car (car nestedList)) ; Extracts the first element of the first sublist, wh
(cdr (car nestedList)) ; Extracts the rest of the first sublist, which is (2
```

LISP's use of linked lists, CAR, and CDR forms the basis for its powerful and flexible list processing capabilities, making it well-suited for symbolic processing and recursive algorithms.

- CDR and CAR always requires not-null argument for the processing.
- CAR always returns single value whereas CDR always returns list.
- CAR and CDR both are pure function ie. they donot modify the argument list.
- Both make the new copy of the list.

Plist and Alist :



Storage Reclamation:

Storage reclamation, often referred to as "garbage collection," is the process of automatically identifying and reclaiming memory that is no longer in use by a program. In Lisp, garbage collection is a fundamental part of memory management due to the dynamic nature of Lisp programs, where new objects are created and discarded frequently.

Here's how garbage collection works in Lisp:

Lisp uses a system of reference counting or reachability analysis to determine which objects are still in use.
Objects that are no longer reachable from the program's roots (typically global variables and function call stacks) are marked as garbage.

A garbage collector periodically scans the memory, identifies garbage objects, and frees the memory occupied by them.
Garbage collection helps prevent memory leaks and ensures that memory is efficiently used, allowing Lisp programs to run without excessive memory consumption.

Recursive Interpreters

A recursive interpreter is an interpreter for a programming language that employs recursion as a fundamental mechanism for interpreting and executing programs written in that language. Lisp is well-suited for recursive interpreters because of its natural support for recursion through its list-based syntax and function calls.

Here's a simplified example of a recursive interpreter in Lisp:

```
(defun evaluate (expr)
  (cond
    ((numberp expr) expr)                ; Numbers evaluate to themselves.
    ((eq (car expr) 'add) (+ (evaluate (cadr expr)) (evaluate (caddr expr))))
    ((eq (car expr) 'sub) (- (evaluate (cadr expr)) (evaluate (caddr expr))))
    ((eq (car expr) 'mul) (* (evaluate (cadr expr)) (evaluate (caddr expr))))
    ((eq (car expr) 'div) (/ (evaluate (cadr expr)) (evaluate (caddr expr))))
    (t (error "Unknown expression"))))
```

In this simple example, the evaluate function interprets arithmetic expressions represented as lists. It recursively evaluates sub-expressions until it reaches base cases (numbers) or known operations (add, subtract, multiply, divide).

Recursive interpreters are valuable for understanding the principles of language interpretation and execution. In practice, Lisp implementations use more efficient mechanisms for interpretation, but the concept of recursion remains fundamental to Lisp's design.

Chapter 5 : Object Oriented Programming (Small Talk)

Activation Record

An activation record, often referred to as a "stack frame" or "activation frame," is a data structure used by a computer's runtime environment (usually associated with a programming language or system) to manage the execution of a function or procedure. Activation records are stored on the call stack, a region of memory used for function call and return management.

Each time a function or procedure is called, an activation record is created and pushed onto the call stack. This record contains information necessary for the function's execution, including:

a. Environment part .
   - It is the context in which the method is executed.
   - The environment part of an activation record contains references to the variables, parameters, and temporary variables used within the method.
   - It allows the method to access and manipulate its local data. The environment part holds bindings between variable names and their corresponding values. As like in dictionary form.

b. Instruction part:
   - The instruction part of an activation record contains a reference to the method's bytecode or compiled code. (Also contains the information about the method itself).
   - It specifies the sequence of instructions that should be executed to carry out the method's logic.
   - Two coordinate systems are used for identifying the instructions.
     a. Object pointer: identifies the method-object containing all of the instructions of the method.
     b. Relative offset: Identifies the particular instruction within the method-object

c. Sender part:
   - contains information about the sender or caller of the current method. (caller and callee)
   - It is used for call stack management, allowing the Smalltalk interpreter to know where to return after the current method finishes execution.
   - A pointer is used (that establishes the dynamic link between the caller and callee).

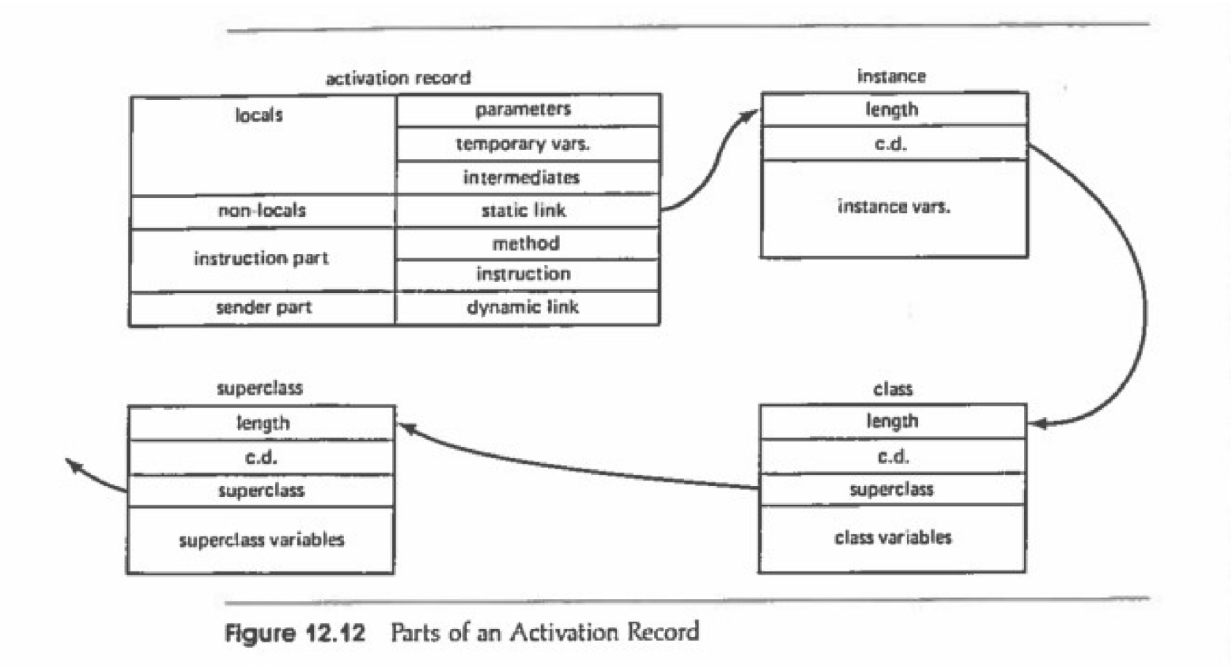


Figure 12.12 Parts of an Activation Record

Message Template

The term "message template" can be used informally to describe a pattern or structure that represents a message that will be sent to an object.

In Smalltalk, messages are a fundamental concept. They are sent to objects to request that the object performs a specific action or returns a result. Messages in Smalltalk typically consist of a message selector (the name of the message) and zero or more arguments. The combination of the message selector and its arguments is what constitutes a message.

3 types of message templates :

1. One parameter / Unary Message Template:

   Syntax: object selector
   Description: A unary message template consists of a single selector with no arguments. It instructs the receiver object to perform a method associated with the selector without any additional arguments.
   Example: receiver print sends the "print" message to the receiver object.

   2. Multiple parameter / Binary Message Template:

Syntax: object selector: argument
Description: A binary message template includes a single selector followed by a colon and an argument. It instructs the receiver to perform a method associated with the selector, passing the specified argument.
Example: receiver at: 2 sends the "at:" message to the receiver object with the argument

3. Keyword Message Template:

Syntax: object selector: arg1 and: arg2
Description: A keyword message template includes a selector followed by one or more keyword-argument pairs. It instructs the receiver to perform a method associated with the selector, passing the specified arguments.
Example: receiver from: 'Alice' to: 'Bob' sends the "from:to:" message to the receiver object with the arguments 'Alice' and 'Bob'.

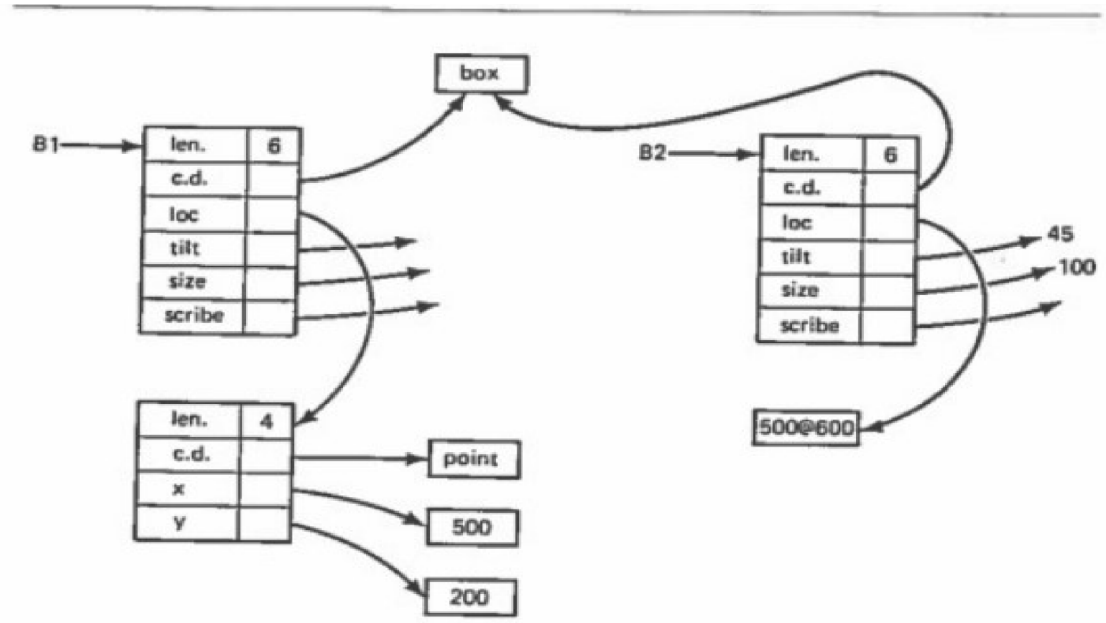
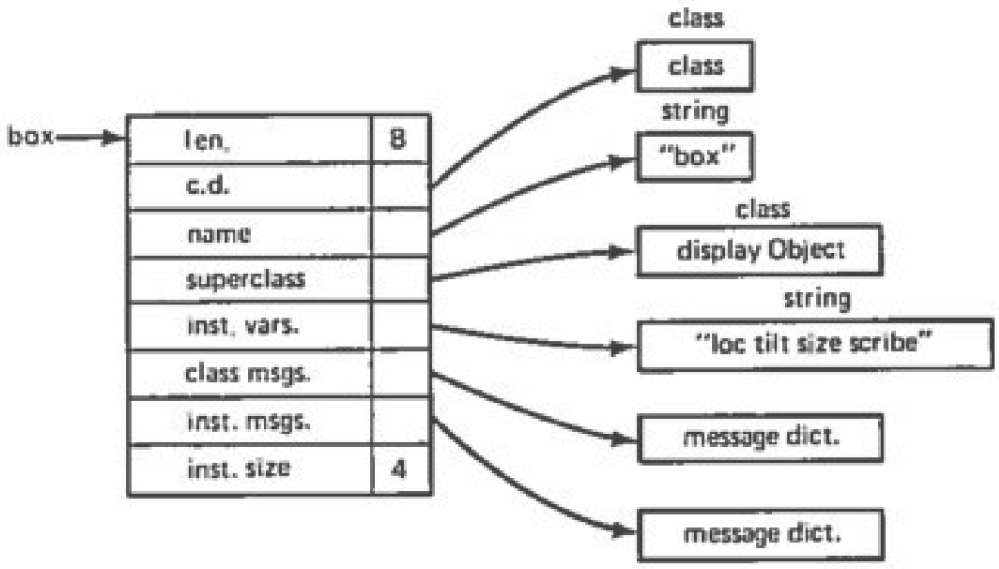


Figure 12.9 Representation of Objects



ure 12.10 Representation of Class Object

https://www.geeksforgeeks.org/similarities-and-difference-between-java-and-c/