PPL

Chap-1

- Programming language collection of
  - syntatic rules
  - keywords
  - naming structures
  - data structures
  - control structures

  that is intended for expression of computer programs.

Importance of programming language

Paragraph.
- supports to better understanding of basic logic of PL.
- supports to understand the language paradigm.

Benefits for eng students:

a) Increased ability to express ideas.
- depth at which a programmer thinks, influenced by the expressive power of language.
- difficult to conceptualize structures, if they can't describe verbally.

b) Improved background for choosing appropriate language:
- studying PPL, students can choose best suitable.
- students tend to write with languages that are familiar even if poorly studied.
- if familiar with other lang, better choice.

Characterichics of PL

Programming domain is a set of PL or Environments, that were written specifically for particular domain.

c. Provides greater ability to learn new languages.
- programmers who understand OO can learn java easily.
- once through understanding of fundaments, easier to see concepts being incorporated.

d. Understand significance of implementation
- Leads to an understanding of why languages are designed the way they are (for better implementation).

e. Ability to design new language.
- more languages learnt, better the understanding of PL.

f. Overall advancement of computing

Characteristics of a good programming language

- Clarity, simplicity and Unity.
  - must be simple, easy to learn and use.
  - minimum no of concepts, so that combining them is not complex.
  - Complex syntax may be easy to write but difficult to debug

eg: APL programs are difficult even for developer. However, power shouldn't be sacrificed for simplicity

- orthogonality.
  - being able to combine various features of languages, with every combination being meaningful.
  - helps to develop new algorithm.

eg: arithmetic calculation & conditional statement.

Characteristics

- Naturalness
  - natural for application areas for which it is designed
  - provide appropriate operators, data types, structures and syntax. to facilitate programmers.

- support for abstraction
  - ability to use structures or operations, ignoring many other details.
  - Many language fail to implement real life problems.

Eg: C++ provides this facility

- Ease of program verification.
  - verification should be provided by languages to minimize errors.
  - testing program with random inputs and obtaining corresponding output.

- Programming environment
  - vital role in success of PL.
  - IDE, which is weak might get a bad response of programmer.

- Portability of programs
  - transportability from one comp or sys to another.
  - Language which is widely available and doesn't support features on different systems are very limiting.
  - should be hardware idependent.

Difference betwn logical and functional programming.

- Cost of use
  Here, cost is time resources:
  - cost of program execution
  - cost of program translation creation, testing, use maintainence.
- Compactness
- Extensibility.

Functional P	Logical P
- totally based on functions.	- totally based on formal logic.
- programs are constructed applying and composing functions	- program statements represent facts and rules related to problems with formal logic.
- designed to handle symbolic computation and list processing apps.	- designed for diagnosis, NLP, planning and machine learning.
- aim is to not isolate from the code and reuse them.	- aim is to allows machines to reason.
∴ reduces redundancy, solves complese problems, increase maintainability.	- it is data driven, array oriented and used to express knowledge
- testing is much easier compared to logical.	- testing is more difficult.
- Simply uses functions.	- Simply uses predicates.

Pseudo code and how it simplified programming.

Pseudo - code.

- first generation programming language
- intended to overcome earlier issues like floating point and indexing.
- An instruction code that is different than that provided by the machine.
- Has own data types and operations.

Need for Pseudo - code

- During first generation of computer, programming was difficult as programmer needed to know the hardware specification of every machine that program was for. — no programming language ∴ Less memory - stored program and data in rotating drums

Pseudo code interpreter.

- is an interpretive subroutine.
  ↳ instructions that usually perform one task, are stored in the memory and are reusable.
- developed to run pseudo code.
- saves memory, since pseudocode is more compact than machines real program code.
- implements its own virtual computer which allows functionality with its own data types and operations not provided by the actual hardware
- has its own data type and operations.

Functional enhancements brought by Pseudocode

- Used to perform floating point operations and indexing.

Pseudo code implements:

- A virtual computer
  - New instruction set.
  - New data structures.

→ Higher level than actual hardware.
Abstracts away hardware details.
provides facility more suitable to applications.

Principles:

The automation principle (Automate mechanical, tedious, error prone activities)
The regularity principle
↳ regular rules, without exceptions.
easier to learn, use, describe and implement

Functional enhancements brought by Pseudo code

- floating-point arithmetic (+, −, ×, ÷, √)
- floating-point comparision (=, ≠, <, >, ≤, ≥)
- Indexing
  - to perform indexing,
    address of array & address of index variable
    which consumes 2 out of the 3 address fields.
  - so, only operation is moves them.
  - can use +6 and −6 to move.
    eg: +6 xxx iii zzz (from x ← z)
    −6 xxx yyy iii (to x → y)
- Transfer of control

- Input/Output -
  - program not useful if it can't read data or print result
  - ·t6 operation to read a 10-digit number into a memory location
  - ·t6 to print content of memory location.

FORTRAN (in 1950s, by IBM)

- Formula Translating System.
- general-purpose, imperative programming language
- used for numeric and scientific computing.
- ruled programming area for a long time.

Name structures of FORTRAN languages.

- The purpose of name structures are to organize the names that appear in a program.
- Name structures structure names.
- Primitives bind name to objects -
  eg: - INTEGER I, J, K.
  - Allocate integers I, J, & and bind them to memory locations.

Declaration of Name structures.

- Declarations are non-executable
  (Type of variable → amount of storage)
  (static or dynamic allocation can be used.
- variable names are local in scope
  (Data hiding concept)
- sub programs are global in scope.

- Allocation is static (cannot be deallocated)
  (FORTRAN - no dynamic allocation)
- FORTRAN doesn't require variables to be declared

Naming convention
- variables starting with, i, j, k, l, m, n are integers.
- others floating points.
- Gives rise to funny names: (KOUNT, ISUM, XLENGTH).

-Environment determine meanings

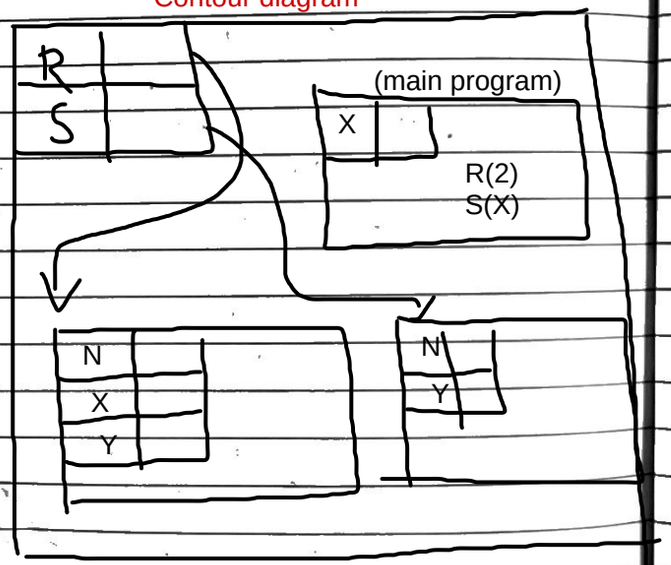
eg: consider a statement,
x=COUNT(I)

-COUNT can be a function as well as an array.
-Meanwhile 'I' can be real or integer
-Here,
context of the statement are the set of definition visible to that statement.
-Context of a statement is its environment

Contour diagram

eg.

```
MAIN PROGRAM
INTEGER X
.
CALL R(2)
.
CALL S(X)
END
SUBROUTINE R(N)
REAL X,Y
.
CALL S(Y)
.
END
SUBROUTINE S(N)
.
INTEGER Y
.
END
```



Modes of Passing Parameters (FORTRAN)

Modes of Passing Parameters in FORTRAN

- allows parameters to be used for input, output or both.

Pass by reference
- pass the address of the variable, not the value
- In fortran, parameters are usually passed by reference.

Advantages:
- faster for larger data constructs.
- Allows output parameters.

Dis :
- too much space, - too much time
- Address has to be de-referenced.
- values can be modified

eg.

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

Not call by reference only an example of parameter passing

Here, X1 & Y1 are used are input variables.
Diff1 is used as output variable. content of diff1 is modified.

Pass by value

Pass by value

- Also called copy-restre.
- copy value of actual parameter into formal parameters.
- Upon return, copy new values back to actual
- Callee never has access to caller's variables.

Eg:

```
SUBROUTINE TEST(X, Y, Z)
X = 1
Z = X + Y
RETURN
END

M = 2
CALL TEST(M, M, M)
```

- If it is passed by reference, value of M?

If it is passed by reference,
X = 2, Y = 2 when subroutine is called, value of X is set to 1 & X also to one since they share the same address.
Result = 4

- If passed by value,
X = 2, Y = 2 when subroutine is called, X is set to 1 but the value of Y remains 2
So, result = 3

CONTROL STRUCTURE OF FORTRAN

Control structure of FORTRAN

- Control structures are those constructs in the language that govern the flow of control of program

Two types:
- Lower Level control structure: IF-statement, GOTO statement
- Higher Level control structure: DO-Loop

1. Arithmetic IF statement

- If (e) n1, n2, n3

Here, evaluates expression e and branches to n1, n2, n3 depending on value of e. (-ve or +ve)

- not very satisfactory
- Later, a more conventional If was added.
- Arithmetic IFs are violations of portability features.

2. GOTO statements

- is the raw material from which control structures are built in FORTRAN.
- two way branch: combination of IF and GOTO

```
If (condition) GOTO 100
    .... case for condition
    If (condition) GOTO 200
    ... case
```

- If we like if-then-else or conditional statement
- to put true and false value, negate the condition

eg: IF (not condition) GOTO 200

left & -computed and Assigned GOTOs
- Dynamic chain of activation records

- for more than two cases computed GOTO is provided.

```
        GOTO (10, 20, 30, 40), I
10      ... case I
        GOTO 100
20      .. case II
        GO TO 100
30      .. case III
        GO TO 100
40      ... case IV.
100
```

Transfers to statement 10, 20, 30, 40 if I is 1, 2, 3, 4 respectively

3. Loops
- Loops can be implemented, combination of IFs and GOTO
- trailing decision loop

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

- leading decision loop

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

4. Computed GOTOs and Assigned GOTOs.
- computed & GOTO (L1, L2 ... Lk), I
  - Li is statement number
  - I is any integer variable
  - computer GOTO transfers to Lk if I = K.
  - usually implemented by a jump table.
- Assigned GOTOs.

```
        ASSIGN 20 TO N
        GOTO N, (L1, L2 ... Ln)
```

  - N integer variable
  - transfers to the statement whose address is in N.
  - List is not actually needed.
  - ASSIGN 20 TO N puts the address of statement 20, say 347 in N.

5. Do - loop
- provides a simple method of constructing a counted loop, also called definite iteration.

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

- statements between DO & continue is executed
- I - controlled variable.
- DO-loop is higher level.
- Loop can be nested.
- Do-Loop is highly optimized.

6. Subroutine (subprogram)

- has libraries of I/O, mathematical like sin, cos as Pseudocode did.
- FORTRAN II added: so called late addition
- they define procedural abstraction
- instead of repeating some sequence of codes define once and call.

```
SUBROUTINE   name(formal para)
    --- body ---
RETURN
END
```

name = function name

formal parameters -> dummy variable.

Eg:

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

This subroutine can be invoked by

```
CALL  DIST ( DIST1, X1, X2)
:
CALL DIST ( DIFFER, POSX, POSY)
```

ALGOL-60 Data structures

Data structures in ALGOL-60.

- Like FORTRAN, Algol-60 is used in scientific applications.
- So, primitive data types are integer, real and Boolean.

Data structures.

- Primitive
  - mathematical scalars like in FORTRAN
  - integer, real, Boolean
  - No complex and double data types.
- Double (Platform dependent)
  - Not portable
    <-> size of word needs to be known

why no complex data type is used?

- can be constructed using other types (2 reals)

Real
- Easy to use, but inconvinient.
- Needs supporting operations.

Strings
- Data structure that needs full supporting operations
- Strings as second class citizens (by Algol designers)
  - only allowed for formal parameters.
  - literals can be actuals.
  - can only be passed around.
  - cannot do anything with them.

- will gets forced to output procedure written in machine language.

- Gives rise to zero-one-Infinity principle
  - programmers not require to remember arbitrary constants.
  - In fortran,
    - identifiers have max 6 characters
    - 19 continuation cards
    - Arrays can have 3 dimensions
  - eg: identifiers name should be either 0 or 1 or unlimited length.

Arrays.
- Arrays are generalized.
- can have any no of dimension.
- lower bound can be other than 1. (less error prone)
- Array bounds can given as expression
- Array size is set until block is exited.

FORTRAN - has fixed array sizes.

Parameter Passing in ALGOL - 60

Parameter Passing

Parameters passing modes are:-
- Pass by value
- Pass by name.

Pass by value.

```
integer procedure fac(n);
    value n; integer n;
```

- Actual copied to formal para
- local variable will not override.
- No output parg
- Ineffecient for arrays (copy must be made)

Pass by name.

- Based on substitution.

```
eg: integer procedure fac(n);
        integer n;
        n := n+1;
    - call fac(i)
```

- we need output parameter tha affects i, not just n;
  - &, i is substituted for n

  Like, i := i+1;

- Copy Rule.
  - copying the procedure and replacing formals with actuals.

- Powerful rule.
  - Evaluation using pass by value, reference and name.

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

Value : A[1] = 1, A[2] = 1 , i = 1
Reference: A[1] = 0, A[2] = 1 , i = 2
Name : A[1] = 1, A[2] = 0 , i = 2

- Implementation of pass by name (using Thunks)
  - to pass the text, would need to compile at runtime.
    ↳ not possible.
- copying code would increase size.
- Use Thunk
  - Pass address to compile code.
  - Address of mem location is returned to callee.

Name structure of Algol-60

Name structure in Algol

- As data structure organize data, control the control flow, name structures organizes the name that appear in program.

eg: when we declare

INTEGER I

- bounds the identifier I to mem location
- INTEGER = type of variable
  I = identifier.

- Declarative constructs are the primitive name struct of Algol
- Array declaration similar to FORTRAN (but allows the index to be other than 1)

-Compound statements

-compound statements are allowed in Algol-60.
-Solves the problem in FORTRAN of allowing only one statement.(Using

eg

```
For i=1 stop | until N do
    sum:=sum+Data[i]
```

Can be written as:

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

-blocks define Nested scope as well s simplify programs:

-In FORTRAN scopes were nested at only 2 levels.
-Algol allows as many as possible with block.
-Simplifies the program as well.

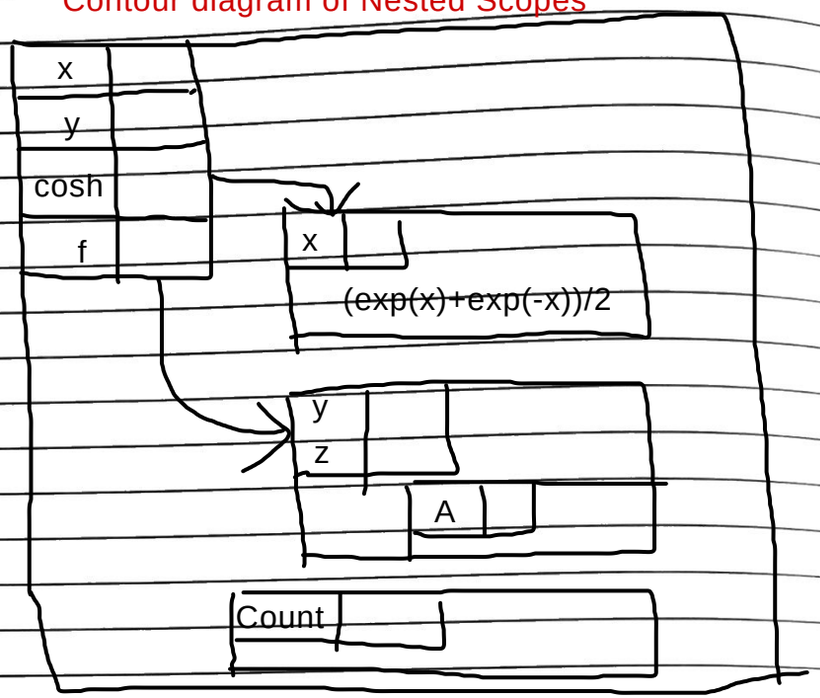
eg:

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

Procedure f(y,z);
        integer y,z;
    begin real array A[1:y];
        .
        .
    end

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

Contour diagram of Nested Scopes



Algol was a major milestone in programming language.

- Though competed with FORTRAN, but never widely used

- Terms it added in programming
  - Type
  - formal parameter
  - actual parameter
  - block
  - call by value
  - call by name
  - scope
  - dynamic array
  - global and local variable.

- concepts it added.
  - Activation record
  - Thunks
  - displays
  - Static & dynamic chains.

- Influence on succedding programming languages
- Helped adaptation of BNF in math theory.
- Machine independence (tend to) → free format
- permits dynamic array.
- Nesting → structured programming.

explain these points.

BNF & EBNF

BNF and EBNF. What features of EBNF that makes it more efficient than BNF

- BNF is a descriptive tool
  - for PL design, to formulate and evaluate aspects of developing design.

- Backus-Naur-Form is Naur's adaptation of Backus Notation to Algol-60. because it
- BNF is → descriptive metalanguage → describes another language

~~Eg: examples~~

- define certain categories of strings like a decimal no. in terms of other syntatic categories like unassigned integers.
  - → integers represented in strings,
    '3', '+3'
    decimal
    '0.1' & '.1415'
- we define in angular brackets using words.
  - → classes of strings
  like <decimal fraction>, <unassigned integer>

-lo, BNF for integer,

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

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

::= means defined as
| = OR.

EBNF

- extensions were made in BNF to improve readability.

eg: In BNF
unassigned integers were defined as recursive defn.
<unassigned integer> ::= <digit> | <unassigned integer> <digit>

- A better notation would be with the use of Kleene Crocs; C+ → sequence of one or more strings of category C.
<unassigned integer> ::= <digit>+
- Also, Kleene Star, C* → sequence of zero or more elements of class C

sequence of zero or more alphanumeric: <alphanumeric> ::= <letter> | <digit>
<identifier> ::= <letter> <alphanumeric>*

- { } dialect allows substitution.

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

≈

<letter> { digit | letter }

example of improvement.
BNF
<integer> ::= <unassigned integer> | -<unassigned integer> | +<unassigned integer>
EBNF
<integer> = { + | - } <unsigned integer> | <unsigned integer> OR,
<integer> = [+ | -] <unassigned integer>
→ + or - or none.

History of Algol-

History of Algol

- need for computer scientists for a single universal, machine-independent PL
- Each machine had its own,
  - instruct set
  - assemblers

  which made portability difficult.
- Hence, May - June 1958, Algol 58 was designed and completed in 8 days.

- Document associated state the following objective:
  - New language should be
    - close as possible to standard mathematical notations and readable with little explanation.
    - possible to use it for description of computing process in publication
    - mechanically translateable to machine lang.
- In January 1960, Algol 60 was created
  - using suggestions (collected)
  - renowned per its elegance.
- Report only 15 pages (whereas other PL 100s or 1000s)
  - ↳ By the help of BNF notations.

  which provided, simple, concise & easy to read, per describing Algol's syntax.

Context free & Regular Grammar.

Significant uses of Context free & Regular Grammar.

- Grammar is a tool used for describing programming features
  eg: BNF & EBNF
- context free and regular are classes of g.
- A regular grammar is one written in EBNF without recursive rules.

eg:

$$\langle number \rangle ::= \left[\begin{matrix} + \\ - \end{matrix}\right] \left\{ \begin{matrix} \langle decimal\ number \rangle \\ [\langle decimal\ number \rangle]_{10} \langle integer \rangle \end{matrix} \right\}$$

(OR previous eqs)

- regular grammar → regular language
  describes
- A context free grammar, that can be ~~extent~~ expressed in extented BNF (possibly with recursive rules)
- context free language described by context free grammar.
- All regular grammar → context free.
- so regular is also context free.

Eg: Context free grammar are to Algol's defn of statement

Rules

$$\langle unconditional\ state \rangle ::= \left\{ \begin{matrix} \langle assigned\ state \rangle \\ For\ \langle for\ list \rangle\ do\ \langle state \rangle \end{matrix} \right\}$$

$$\langle statement \rangle ::= \left\{ \begin{matrix} \langle unconditional\ statement \rangle \\ If\ \langle expression \rangle\ then\ \langle un\ state \rangle \\ If\ \langle exp \rangle\ then\ \langle un\ state \rangle else \langle state \rangle \end{matrix} \right\}$$

- Use of recursion allows nested syntatic structure
- regular grammar ↳ not possible.

LISP Structural organization

LISP. Explain the structural organization of LISP with a suitable example.

- Programming language developed in 1950s out of need for AI programming.

- natural data structuring methods: pointer
  linked list

Structural organization of LISP.

1) Extremely applicative language:
   - almost every thing is a fcnc in LISP.
   - function is written as:

(cambridge polish notation) (f a1, a2 ... an)
↳ operator before operand — function — arguments.

i.e (plus 5 2)

eg: (set 'text (make to do))
≡ set (text, make (to, do)) in Algol.

2) List is the primary data structure constructor:
   - Focus is allow programmer to manipulate lists of data.
   - comparing them, passing to func, putting together & taking apart

eg: (set 'text '((to be or not to be)))
↳ variable 'text' is set the list composed of values 'to, be, or, not

3) Programs are represented as list:
- functional apps and lists look the same.

Eg: in S-expression
(make-table text nil)

↳ can be both an application or function
- In lisp it both.
- most cases,
  S-expression is interpreted as fun application with evaluation of arguments
- if list is quoted, then it is treated as data.

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

4) Lisp is interpreted
- has interactive interpreter
- interprets a statement and prints out result.

Eg: (plus 1 2 3)
interprets 5
(eq (plus 2 3) (difference 9 4))
interprets t as 2+3=9-4.

- eq & plus pure function
- set pseudofunction
- defun → (defun f (n1 n2 --nm)
  ↳ defines a func with name f

5) Give example.

CAR & CDR

CAR and CDR and their difference.

- are used to access parts of lists in lisp.
- List requires operation for building and breaking them apart
  - Building - constructors as cons
  - selectors as - car and cdr
    pure functions

CAR
- used to select first element of the list.

eg: (car '(to be or not to be))
returns 'to'.

- first element can be an atom or list.

eg
Let prog be a list,
((to 2) (be 2) (or 1) (not 1))
the application (car prog)
returns (to 2)

- argument always non-null list.

CDR
- car can only access first element
- CDR used to access rest of the elements except first.

eg:
(cdr '(to do not be)
returns list (do not be)

- also requires non-null arguments.
- in case of one argument, returns null.

eg (cdr '(1))
returns ();

LISP searching

How LISP maintained its simplicity principle? Explain different searching techniques in LISP.

- Two selectors in lisp car and cdr which are adequate for accessing elements of list this maintaine simplicity.
- Also provide abbreviation to prevent long and large composition of car and cad.

eg:

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

can be abbreviated to

(caddr DS)

DS

= (set 'DS '((Dan smith) 45 3000 (August 25 1980)))

name, age, salary and hire date

- abbreviations can be error prone.

↳ solution is to write special-purpose functions for accessing parts of the record.

Eg:

(define hire-date r (caddr r))

The function (hire-date DS) returns Dan Smith's hire date or any date of any list passed of any other p-records with same structure

- this makes lisp maintainable & simple
- lisp thinks personel records as abstract data type
- can only be accessed through function.

Searching techniques are cdr and car

Example of searching:

(caddaddr DS) where DS is.

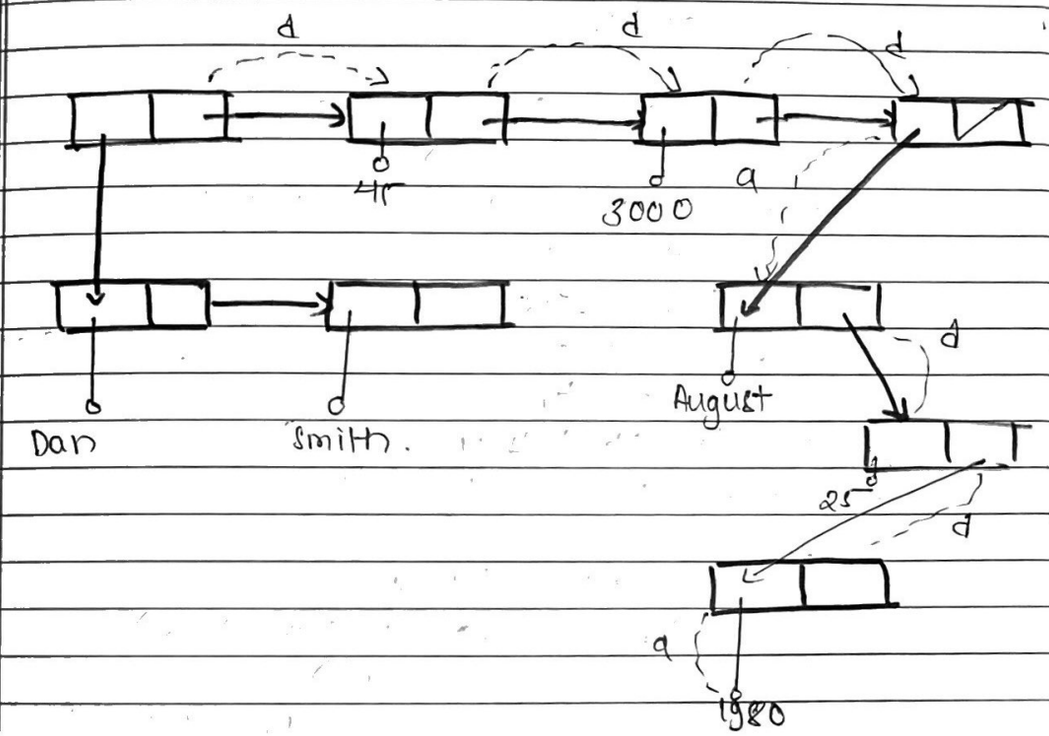


fig: walking down a list structure.

Association and Property list

Association list and property list.

Property List

- representation of personnel record in form,
  (Dan smith) 45 3000 (August 15 1980))
  not flexible.
- So, to combat this indicators are used to identify the property.
  Eg:
  (name(Dan smith) age 45 salary 8000 hire-date (August 25 1980))
- This method is called property list
  general form,
  (p1 v1 p2 v2 ... pn vn)
  p = indicator
  v = property
- provides flexibility.
- code to access property list

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

- In above code, we use recursion to get values.
- p = indicator, x = list.

```
(getprop 'name DS)
    → Dan smith.
```

Why property list is not flexible for flags.

etruct = structure.

Association list

- in p list one atom or list value is associated with a property. (P, V, ... Pn Vn)
- can be inconvinient when the data struct has flags
- In case of flags, does not require to be in a indicator-property form.
  Eg: retired atom in DS indicates the person being retired.

DS ⇒ (name (Dan smith) age 45 salary 3000 hire-date (August 25 1980) retired)

- Here, retired flag breaks the p-list properties
  - could be fixed by making it in indicator-property form. So,
    - inflexible to flags.

↳ Solved by association list

((name (Dan smith)) (age 45) (salary 3000) (hire-date (August 25 1980)) (retired))

- General form
  ((a1 v1) (a2 v2) ... (an vn))
- An, assoc function is used in forword association statement:

```
(assoc 'hire-date DS)
```

returns → (August 25 1980)

left
- Control structure in Lisp
- 2018 (Spring) Q9
- Recursive interpreters and storage Reclamation in LISP

Small talk

Questions.

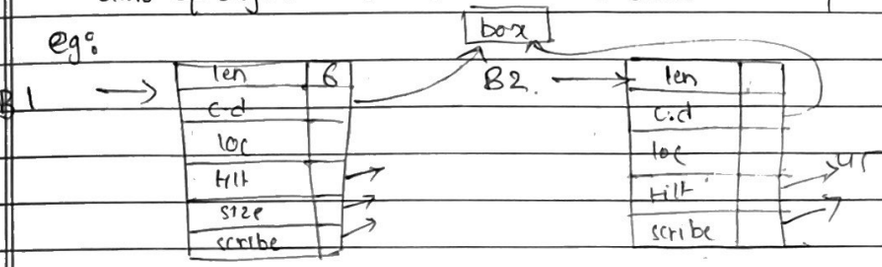
1. Explain how class and objects are represented in SMALLTALK. Diagram also.
2. How do classes allow multiple representation of data types in the Smalltalk? Explain with the help of orthogonal classification
3. Three forms of message template in smalltalk.
4. Explain how Small Talk represents the object oriented paradigm with suitable example.
5. Explain message passing and returning mechanism in Smalltalk.
6. How is activation record represented in SMALLTALK.

Object and Class Representation.

Object Representation

- Everything is an object in Smalltalk
- even classes are objects in smalltalk.
- Representation of objects follows the information hiding and abstraction principle.
- Objects must contain varying info.
- Similar info are stored in definition of class.
- class of object must be known to access the info.

eg:



Class representation

- classes are just instances of the class named 'class'.
- Therefore ~~obje~~ classes are represented just as objects.
- described with length and class description field.
- instance variables of class object contain pointers to objects
  ↳ that represents info. That is same for all instances
- information includes: class name, superclass instance variable names, class message dictionary, instance message dictionary.
- inst. size indicates no of instance variable

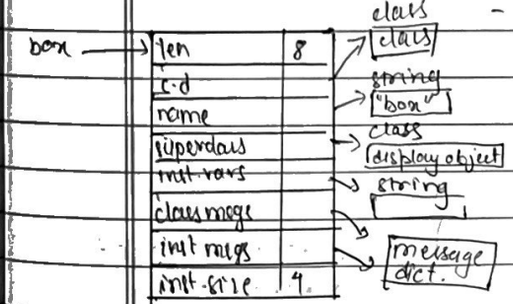


fig: Representation of class.

Activation record representation

Activation record representation

- Sending message in smalltalk similar to procedure call in other
- Activation record holds,
  info related to activation of a procedure
- process of calling procedure and returning is called manipulation of activation records.
- some case in smalltalk as well.

Three major parts:
- > Environment part (content for execution)
- > Instruction part (instruction for execution)
- > Sender part (AR that sent message to invoke method)

Sender part:
- dynamic link
- a pointer from the receiver's activation record is sent back to the sender's AR (just an object reference)

Instruction part:
- particular instruction is designated to particular method
- since methods are object themselves, two-coordinate system used to identify instructions:
  - > object pointer > relative offset

Environment part
- must provide access to both local & non-local environments.

Local environment: includes space for parameters of method.
- temporary variables & hidden temp variables

eg: newAt: initialLocation |newBox|
newBox ← box new
- must contain space for this parameter
- must contain space for this temp variable

Message sending & Receiving

non-local environment: includes all visible variables (instance and class variables)

Message sending
1. Create an activation record for receiver
2. Identify the method being invoked,
   by extracting the tempolate from message and looking it up in message dictionary for receiving the object class or superclasses
3. Transmit the parameters to receiver's activation record
4. Suspend sender,
   by saving its state in its AR.
5. Establish a path from receiver back to sender and establish receiver's activation record as active

Returning messages

1. Transmit the returned object from receiver back to the sender
2. Resume execution of sender by restoring its state

Forms of message tempelate

Forms of message tempelate in Smalltalk.

eg
>B1 show(keywords for parameterless messages)
- adequate for all purposes
- handle any parameters from zero on up.
- Uncommon way of arithmetic expression

eg: (x+2)xy' written as:
(x plus:2) times:y

> Operators for one-parameter messages(eg 'x+y')

exception - arithmetic operators can be followed by one parameter exactly even if no colon.
eg: x + 2* y
- object 'x' is sent message '+2',
result of this is sent message '*y'.
- So, expression computes as (x+2)y.
- Doesn't follow precedence rule.

> Keyword with colons for one or more parameter messages
eg scribe grow:100

How does Small talk represent OO paradigm

- Don't have primitives and control structures like procedural languages
- perhaps the purest example of object-oriented programming
- everything in smalltalk is an object.
- Objects are independent codes that manage a piece of data.
- Other objects act upon the data by passing messages to its objects.
- objects do not have access to each other by other means (encapsulation).
- Advantage,
code is reusable and easy to test.
↳ this reduces the work of programmer.
- has influenced other languages such as C, Python, Java which also follow OO paradigm.

- polymorphism, encapsulation(abstraction), inheritance sabai cha.

Example:
Adding two numbers
x ← add(y)
different add if x is integer, complex etc.

But in conventional programming,
add(x, y)
function add has fixed meaning.