classmate
Date
Page

x

set ('n)

≈ means binding a variable name n with value of n.

LISP

</>

What is Lisp? Explain the structural organization of Lisp with a suitable example.

Lisp is a programming language developed in late 1950s out of need for Artificial Intelligence programming.

Where the pointer and linked list structures are its natural data-structuring methods.

Lisp was first implemented in IBM 704 and allowed for 3 overlapping programming paradigms.

Structural Organization of Lisp.

1. Lisp is extremely applicative language:

Lisp is an extremely applicative language. In Lisp almost everything is a function same way. It also represents program and data in In lisp function written as:

(F a1 a2 ... an) ← operator before operands.

{Cambridge polish notation}

where F is the function and a1, a2, ... an

'text means passing - "text" rather than value held by variable text.

treating program as data

are the arguments.

This notation is cambridge polish notation where operators before operands

ie (plus 5 2)

lisp would respond 5.

Also (set 'text (make to do)) is equivalent to Algol set (text, make (to, do))

2. In lisp list is the primary data structure Constructor.

→ In lisp list is the primary data structure means lisp treats both program and data in the same way. in the form of lists.

This means lisp's focus is on allowing programmers to manipulate list of data. which includes comparing them, passing them to functions, putting them together and taking them apart.

Eg.

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

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

This means variable text is set the list composed of values/atoms; to, be, or, not.

3) programs are represented as list:

in lisp, functional applications and lists look the same.

Eg in S-expression:

(make-table text nil)

could either be a 3 element list or it could be an application of the function make_table to which arguments named text and nil are passed. So which is it?

In lisp its both since lisp program itself is a list.

Under most circumstances an S expression is interpreted as function application which means that arguments are evaluated and function is invoked. however, if the list is quoted, then it is treated as data. that is it is unevaluated.

Eg

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

here set function is passed a "text" as data followed by a list of data's.

If the above statement was written as

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

~~means the set funct~~ is equivalent to algol.

→ set (text, to (be, or, not, to, be))

4) Lisp is Interpreted.

Lisp system has an interactive interpreter.

ie lisp interpreter a statement and prints out results.

Eg (plus 2 3)

gives 5

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

the system responds with t

which means true as 2+3 = 9+4.

eq and plus are pure functions while set is a pseudo function.

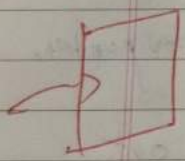
Eg (set 'text '(to do or to be))

defun is a pseudo function.

(defun f (n1 n2 ... nm) B)

that defines a function with name f. and formal parameters $n_1, n_2 \cdots n_m$;

5.> An lisp program example:



```
(defun make-table (text table)
    (if (null text)
        table (make-table (cdr text)
                (update-entry
                table (car text)))))

(defun update-entry (table word)
    (cond ((null table) (list (list word)))
        ((eq word (caar table))
        (cons (list word (add1 (cadar table)))
            (cdr table)))
        (t (cons (car table)))
        (t (cons (car table)
                (update-entry (cdr table)
                word)))))

(defun lookup (table word)
    (cond ((null table) 0)
        ((eq word (caar table)) (cadar
                table))
```

(t (lookup (cdr table) word))))

(set 'text '(to be or not to be))
(set 'Freq (make-table text nil)).

how lisp maintained the simplicity principle? Explain different searching techniques in lisp.?

<4>

differentiate cdr and car.

- cdr and car are used to access parts of list in lisp.

→ A list requires operations for building the structures and operations for taking them apart.

hence lisp has operations that build structure such called constructors.
while those that extracts their parts are called selectors.
List has one constructor as - cons - and two selectors — car and cdr.

Car and cdr are pure functions as they dont modify argument's list.

car

car is used to select first element of a list.

Eg (car '(to be or not to be))

returns atom to.

The first element can either be an atom or a list, and car returns it. whichever it is. For example, let Frog be the list

((to 2) (be 2) (or 1) (not 1))

the application

(car Frog)

returns the list

(to 2)

The argument of car is always an non null list. while it may return an atom or a list depending on what its argument's first element is.

Cdr

Car can access only first element (either atom or list).
So cdr is a selector that is used to access rest of the elements. except the first.

eg:

(cdr '(to do not be)

returns list (do not be)

Cdr also like car requires non-null argument but unlike car, cdr ~~car~~ always returns list. In case of only one argument it returns a null list.

eg (cdr '(1))
returns ().

About Cdr and Car:

Both dont modify their argument list (hence pure functions) ie they work on a copy of list provided to them as argument

Eg: cdr doesnt delete first element of its argument rather, it returns a new list exactly like its

argument except its first element.

In Lisp Car and cdr can be used in combination to access the components of a list.

Eg.

(set 'DS '((Don Smith) 45 3000 (August 25 1980)))

The list DS contains Don Smith's name, age, salary and hire date.

to access his name (car DS) can be used which returns (Don Smith).

While to access Don Smith's hire date the below statement can be used.

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

4 3 2 1

here,

① cdr returns (45 30000 (August 25 1980))

② cdr returns ((30000 (August 25 1980)))

③ cdr returns ((August 25 1980)).

④ Car gives us (August 25 1980).

<2>

how lisp maintained its simplicity principle. also explain different searching techniques in lisp.

→ In lisp two selectors car and cdr ~~fune~~; pure functions are adequate for accessing the components of any list structure.
due to this lisp maintained its simplicity principle.

Along with this they provided abbreviation to prevent long and large compositions of cars and cdrs.
Eg:

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

could be abbreviated to [DS is the same list as On 4]

(cadddr DS).

Also it is observed that these abbreviations write can also be error prone.

So in lisp the solutions for this is to write special-purpose functions for accessing the parts of a record.

Eg:

(define hire-date (x) (caddr x))

the function (hire-date DS) returns Don Smith's hire date. or date of any list passed of any other personnel records with same structure.

This makes lisp program maintainable while lisp maintains its simplicity.

lisp thinks of personnel records as an abstract data type that can only be accessed through the provided function.

(accessing)

→ Searching techniques in lisp are cdr, car. [Note here explain those]

and just like above; abbreviations of lisp are presented in form of linked data structure.

Eg. (caddaddr DS) where DS is

((Don Smith) 45 30000 (August 25 1980))

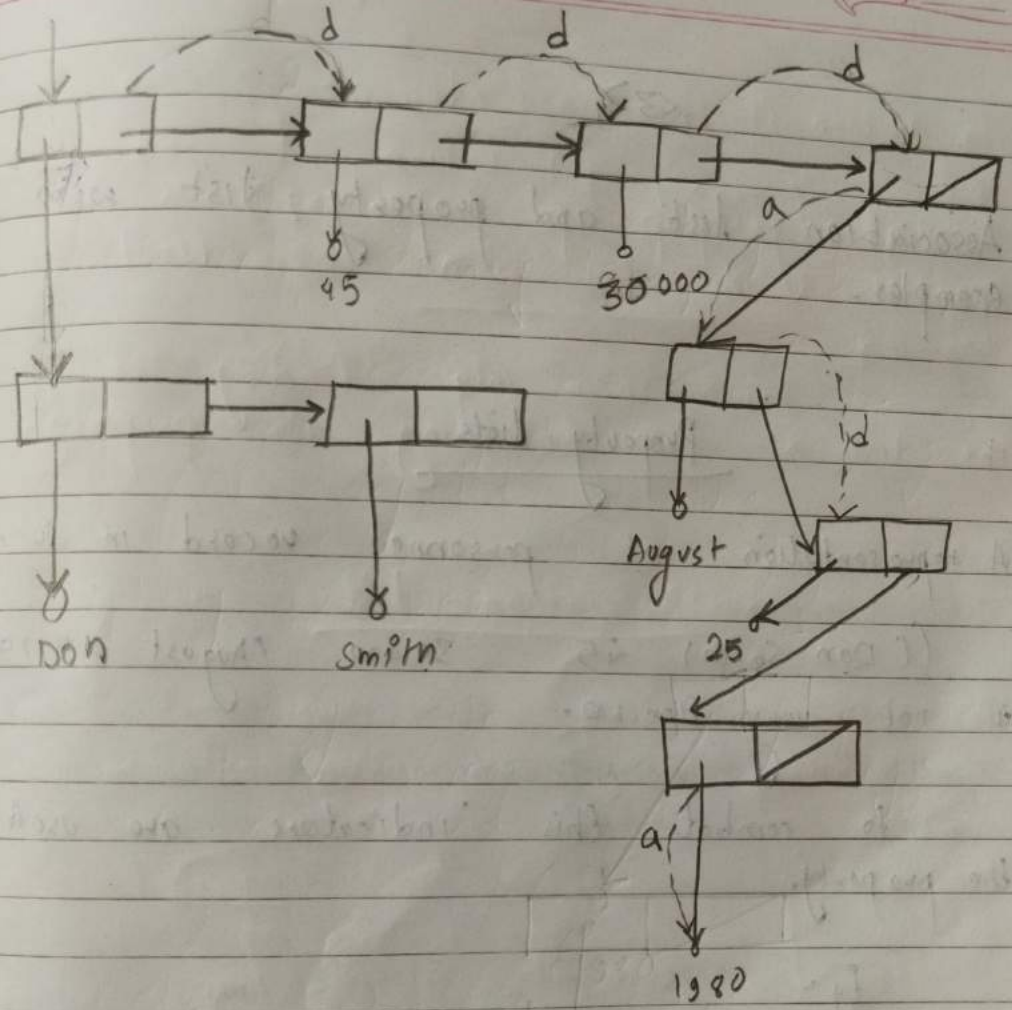


Fig Walking down a list structure.

<3>

Association list and property list with relevant examples.

Property lists

A representation of personnel record in Bom.

((Don Smith) 45 30000 (August 15 1980))

is not very flexible.

to combat this indicators are used to identify the property.

Eg:

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

This method of representing information are called Association list.

general form

(p1 v1 p2 v2 . . . pi vi)

p = ~~property~~ indicator

v = ~~ind~~ property.

Advantages of property is flexibing.

In lisp these property list can be accessed with the following code or something familiar.

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

the above lisp code uses recursion to get values based on indicators.

where p is the indicator or the property we need from a property list x

(getprop 'name DS)

returns → (Don Smith)

getprop 'hire-date DS)

returns → (August 25 1980).

Explaination of how the get prop works:

If the first element of x is the is indicator p then (eq (car x) p) returns a 't' and IF statement gets validated

to return the first element ; else the (eq (car x) p) returns f and else called on return (get prop p (cddr x)) is the list x but with the first element removed.

hence recursively finding the indicator in the list x and returning its property.

Association list

A property list data structure is where one atom or list value is associated with a property.

eg (p.v1 ... pi.vi)

This is inconvenient at times where the data structure has flags.

In case of flags its presence itself is the value or information and requires not to be in p.v indicator - property form.

Eg retired atom in the list DS indicates the personnel being retired

DS => (name (Don Smith) age 45 salary 30000 hire-date (August 25 1980) retired).

here retired flag breaks the p-list property and could be fixed by ~~not~~ making the flag in indicator-property form, which makes the p-list inflexible to flags.

These inflexibility can be solved by an association list where:
it could be in form.

((name (Don Smith)) (age 45) (salary 30 000) (hire-date (August 25 1980)) (retired))

The Alist is in form.

((a1 v1) (a2 v2) . . . (an vn))

ie attribute value pairs.

Statement: (assoc 'hire-date DS) → (August 25 1980)

An assoc function is used in forward association.

<5>

how does cdr and car help in searching data elements? explain with walking down diagram.

Ans 4 (about car and cdr)
Searching techniques in lisp part of Qn 2 write about abbreviation and its walking down diagram equivalent.

nil is an atom in LISP

<7>

polish notation

→ ans in (3n<1)

how hierarchial structures processed in LISP?

→ In Lisp hierarchial structures difficult to be handled by iterations are handled Recursively.

An example would be an equal function implemented by using pred function eq that returns 't' or 'nil'

Example:

```
(defun equal (x y))
  (or (and (atom x) (atom y) (eq x y))
      (and (not (atom x)) (not (atom y))
           (equal (car x) (car y))
           (equal (cdr x) (cdr y)))))
```

here
(atom x)
returns t if x is an atom if a list returns nil

defun defines a function equal with two parameters x y.

~~for~~

for means if any of ~~its~~ elements in the list it has is ~~then~~ 't' it returns t ~~and stops computing~~. else returns nil. or returns as soon as it encounters 't' in itself.

(1) (and (atom x) (atom y) (eq x y))

here and means if ~~its dist~~ all elements besides itself is 't' it returns a 't'.

So the statement (1) means. If the arguments passed to the equal function ~~are~~ ~~then~~ atoms, compare them and if their equal return t else. nil.

Statement

(11). the (and (not (atom x)) (not (atom y)))
means when both x and y arguments are not atoms return 't' else nil.

<8>

Explain conditional Expression, the logical connectives, and Mapcar and reduce functions.

Conditional Expression

→ lisp is first language to contain a conditional expression.

→ in previous languages 'Fortran' or newer 'Pascal' required the user to drop from the expression level to the statement level in order to make a choice.

→ this was not efficient when representing Mathematical expressions and equations.

→ hence cond was introduced in lisp to allow for conditional expression

Eg $$sg(x) = \begin{cases} 1 & , \text{if } x > 0 \\ 0 & ; \text{if } x = 0 \\ -1 & , \text{if } x < 0 \end{cases}$$

In lisp is defined as:

```
(defun sg (x)
    (cond ((plusp x) 1)
          ((zerop x) 0)
          ((minusp x) -1)))
```

Format (cond $p_1 e_1$ ... $p_n e_n$).

Pgsuy

(null L)
= checks if L is null
if L null => t
else nil.

classmate
Date
Page

Logical connectives

MapCar

It is a function that applies a given function to each element of a list and returns a list of results.

implementation:

```
(defun mapcar (f x)
  (if (null x)
      nil
      (cons (f (car x)) (mapcar f
                              (cdr x)
      )))
```

Reduction Function :

the implementation of plus-red is an example of reduction function.

~~that it is~~

plus-red is a binary function to reduce a list to a single value.

```
(defun plus-red (a)
    if (null a)
       0
       (plus (car a) (plus-red (cdr a))
    ))
```

Example

```
(plus-red '(1 2 3 4 5))
```

→ 15

Short notes:

1) User defined function in Lisp:

User defined function in Lisp are done by using pseudo function defun which defines a function.

(defun f (n1 n2 ... nm) b)

this statement defines a function f, with formal parameters (n1 n2 ... nm) and a body b.

Example:

~~(defun sum (x) &~~

(defun add1-map (a)
(if (null a)
nil
~~(add1~~
(cons (add1 (cdr a))
(add1-map (cdr a)))))

~~the add1-map~~

(add1-map '(2 4 6 8))
3 5 7 8

→ it adds 1 to all elements of list.

ii) Car 'cdr Function
← On 4.

~~iii) Storage Reclamation~~

iv) Recursive Interpreters:

A lisp Interpreter can be written in lisp. this is done by a universal Function A universal Function is an function that can interpret any other function

Since this universal function is written in lisp it can make use of the facilities of Lisp, such as cdr, car, and cons.

which in turn are used to interpret cdr, car, and cons of lisp itself.

If a recursive Interpreter is being made in any other language other than lisp it should have recursive procedures and ability to implement Linked list.

The lisp universal function is called eval.

and in addition to the expression to be evaluated eval must have second parameter. which is a data structure representing the context in which the evaluation to take place.

eg.

(eval 'E A) = A.

Eg:

(eval '(cons (quote A) (quote (B C D))) nil)

→ (A B C D) where nil is empty context.

this is equivalent to

(cons ('A) ('(B C D))

→ (A B C D)

here recursive interpretes has implementations to convert quote to i before execution.

(10) Evaluate the lisp expressions:

a) $\frac{1}{2}\sqrt{\pi r^2 - 1^2}$

= (times (quotient 1 2) (sqrt (difference (times (pi (expt 4 2))) (expt 1 2))))

b) $\frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$

=> (quotient (difference (minus b) (difference (expt b 2) (times 4 a c))))

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

c) $(-1)^k \; k^{(1/k)}$

→ ~~(times (mins~~

(times (expt (minus 1) k) (expt k (quotient 1 k)))

d) $\frac{n!}{4!(n-4)!}$

→ ~~(quotient (fact n) (times (fact 4) (difference n 4)))~~

= (quotient (fact n) (times (fact 4) (fact (difference n 4))))

e) $\frac{-b - \sqrt{b^2 - 4ac}}{1/2\sqrt{4a^2 - b^2}}$

opening ( ) closing of same function.

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