UNIT 1 A.I. LANGUAGES-1: LISP

1.0 INTRODUCTION

The task of solving problems using computer as a tool, in general, is a quite a comprehensive task. Ever since the use of computers in solving problems, it has been found that the solving of problems can be facilitated by using appropriate style/paradigm for a given type of problem and using a language designed and developed according to the basic principles of the style.

LISP has jokingly been called '*the most intelligent way to misuse a computer*'. I think that description is a great compliment, because it transmits the full flavour of liberation; it has assisted a number of most gifted fellow humans in thinking previously impossible thoughts'

Edsger W. Dijkstra

Some of the well-known programming languages like C support **imperative style** of programming for solving problems with the help of a computer. The major feature of imperative style is that the proposed solution is expressed in terms of *variables, declarations, expressions and commands. Declarations* assign names to locations in the memory and associate types with the values. *Commands* may be thought of as names for actions, that are required to be executed by a computing system, mainly to change values stored in the memory locations. Commands are generally executed in the order, from top to bottom, as these appear in the program, though through conditional and unconditional jumps, flow of execution can be changed. One of very important concept in imperative style of programming is that of *the state (of memory),* i.e., the set of values assigned to various locations in the memory at a particular point of time.

In this style of programming, the programmer is required to think and express proposed solution (to a problem under consideration) in terms of the basic actions that a machine is to carry out. For complex problems, the task of the programmer becomes more and more difficult with increase in the complexities of the problems to be solved. In view of this difficulty, *alternative paradigms or styles* for solving problems have evolved since almost the beginning of problem solving with computers. The language LISP, we are going to discuss in this unit, supports an alterative paradigm, namely, **functional paradigm**, for solving problems with the help of a computer. In

functional paradigm, the solution is primarily considered as an exercise in *defining functions* and *applying functions* either recursively or through composition. Common LISP is not a pure functional style programming language. *It has some features of imperative style also*. For example, the symbol SETQ allows assigning names to memory locations and storing values in the memory locations. But, Lisp incorporates features dominantly of functional programming style.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- explain what is *functional paradigm* of problem solving using a computing system;
- explain how functional paradigm is different from the *normally used imperative paradigm;*
- discuss LISP as a language having features of functional paradigm;
- enumerate and discuss characteristic features of LISP;
- enumerate and discuss data types and data structures of LISP;
- use these types and structures in defining data required to solve a problem under consideration ;
- explain the role of the inbuilt function EVAL;
- explain and use various built-in general functions, list manipulation functions, predicates, and logical operators;
- explain the role of the special forms viz. Defun, COND, DO, LET and use these in writing complex programs in LISP;
- explain and use the various input/output primitives available in LISP;
- explain the significance of the concept of *recursion* in solving problem, and how recursion is achieved in LISP;
- explain the role of *association lists* and *property lists* in developing database and use these in defining objects and their attributes/properties;
- explain the role of a *symbol* as a *variable* and further should be able to explain the concepts of *bound variable* and *free variable;*
- explain the concept of dotted pairs and be able to use the concept in representing lists in LISP, and
- Write complex LISP programs.

1.2 BASICS OF LISP

The **programming language LISP** takes its name from **Lis**t **P**rocessing. LISP* was developed by **John McCarthy, during 1956-58** and was implemented during 1959-62. LISP has a number of dialects. However, with the development of COMMON LISP in the 1980s and its acceptance by a large number of system implementers and manufactures, it has become almost standard for LISP users. We also shall be using and discussing COMMON LISP only.

LISP has been one of the most popular languages for AI applications. **LISP was specifically designed for A.I. applications**, and we have already mentioned that AI applications involve *symbolic processing*, instead of mere *numeric processing*. Also, AI systems are generally large and complex. Their development requires that the implementation language and support environment provide *flexibility, rapid prototyping and good debugging tools*. On all these counts, LISP wins over all other

*LISP is based on a formal system called λ-calculus (Lambda-calculus) originally proposed by Alonzo church and who developed it later alongwith Stephen Kleene as a foundation for Mathematics.

programming languages. Hence, the language was used in its earliest applications for writing programs which performed symbolic differentiation, integration and mathematical theorem proving. Later applications mainly written in LISP, include expert systems and programs for *common sense reasoning, natural language interfaces, education and intelligent support systems, learning, speech and vision.* LISP has been found quite useful for the purpose of systems programming to the extent that LISP machines have been developed in which whole of the programming from top to bottom is in LISP. In **LISP machines**, which are personal computers, the operating system, the user utility programs, the compilers, and the interpreters are all written in LISP.

We summarise below the main characteristics of LISP.

(i) It is an **applicative/functional language**. *In a functional language the primary effect is achieved by applying functions either recursively or through composition.* For example, in order to evaluate the arithmetic expression (x*y) + (z – u) where the variable x, y, z and u have values respectively 9, 2, 8 and 6, the following LISP expression

(+ (* x y) (– z u))

evaluates to 20. For evaluating the expression, first the function '*' is applied to the values of x and y, next the function '–' is applied to the values of z and u and then recursively the function '+' is applied to the results of the applications.

(ii) The above example also shows that **prefix notation is used in LISP**, i.e., the operator '*' comes before operands x and y. The prefix notation has the advantage that the operators (like +, –, * etc.) can be easily located in an expression.

In contrast, the programming languages introduced to us earlier like, FORTRAN and C are all **imperative languages**. A language is said to be *imperative which achieves its primary effect by changing the state of variables by assignment*. For example, to compute the arithmetic expression (x * y) + (z – u), the programme code in *Pascal* would include a sequence of instruction like,

*Product = x * y;*
Difference = z – u
Result = product + sum

It is easily seen that **the emphasis is on assigning values** to the variables, viz, product, difference and result.

(iii) *The language LISP allows programs to be used as data and vice-versa*.

LISP has mainly one data structures viz list, in addition to the elementary data types: number and symbol. All expressions, whether data or programs, mainly are expressed in terms of lists.

The main advantage of this property of LISP is that the *declarative knowledge, i.e.*, information about properties of an object *can be easily integrated* with *procedural knowledge*, i.e., information about what actions to be performed. The facility of uniform representation **in LISP is also useful in the sense that it allows us to write**

- LISP programs which can modify other programs (written in any language) including themselves.
- LISP program that can write entirely new LISP programs.
- AI programs that learn new tasks.

(iv) LISP is a highly *modular* language and hence suitable for development of large software.

(v) The LISP environment provides a facility called *Trace* by using which programs written in LISP can easily keep track of the various instructions that have been executed, the number of times each has been executed and the order in which the instructions have been executed.

(vi) LISP, being based on the mathematical discipline of λ-calculus, is **the most well-defined of all the programming languages**. Hence, programs in LISP are more reliable. The well-definedness of a language is a very important issue as can be seen from the fact that due to a small error in FORTRAN program of the type quoted below and **FORTRAN environment's incapability to detect it, lead to the loss a spaceship.**
In FORTRAN, the statement

DO 12 I = 1,5

denotes the beginning of a loop, whereas the statement

DO 12 I = 1.5

is an assignment statement. Through the execution of the second statement, the value 1.5 is assigned to the variable DO12I, because blanks are ignored at all places in FORTRAN.

(vii) **Comments in LISP** are given by using the character for semicolon i.e. ';' as the first symbol on the line which is to be treated as a comment. Sequence of characters on a given line after semi-colon, is treated as a comment.

We included this feature here because while explaining various features of LISP, we would be required to provide comments in LISP environment.

(viii) **The types of the variables are not required to be declared** in the beginning as is done in imperative languages like, FORTRAN or C. Also a variable name say x, we can assume any type for the values of the variable within the same program or procedure.

(ix) LISP is format-free. Any valid LISP expression, say

(x (yz) u)

can be written in any one of the following (or even other) formats:

```
( x (          ( x ( y
y z )          z
u              )
)              u )
```

or

(x) We should note that the **two parentheses** viz the left parenthesis denoted by '(' and the right parenthesis denoted by ')' **are two most important characters in LISP and must be used very carefully**. They are used to denote lists and for each left parenthesis, there is a right parenthesis for any valid LISP expression. In the light of the above fact, the expressions

xyx) or (fg

are illegal symbols or expressions.

Next most important character in LISP is quote, the role of which is explained after some time under (ii) of evaluation of S-expr.

(xi) **Separator**. Blanks are used to separate S-exprs, i.e., any valid Lisp entity or object, specially numbers and symbols. Lists are always separated automatically from other S-exprs. **Comma is used for special purposes and not as a separator**.

(xii) **List is defined recursively**. A list is a sequence of atoms and/or other lists enclosed within parentheses. Each of the following three expressions is an example of a valid list.

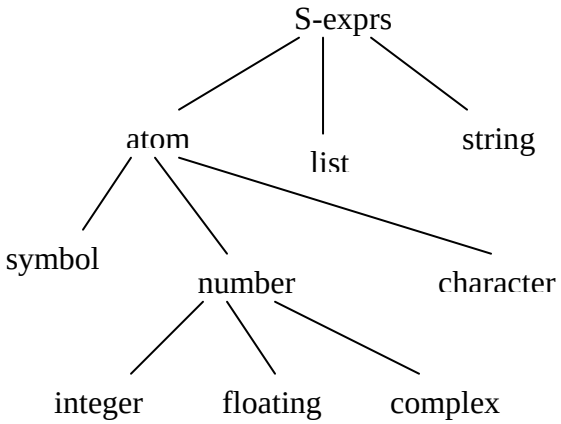
```
( 3 a ( c d ) )
( this is a list of only symbols)
(  )
```

But the following expressions, one on each of the next three lines is *not* a valid list.

```
( b a 3
this – is – not – a – list
)   a   b   c   (
```

1.3 DATA STRUCTURES AND DATA VALUES

Notation for a *valid* object in LISP is called **S-expressions** or just *S-exprs* (**shorthand for Symbolic Expression**). Even *objects themselves* are also referred to sometime as S-exprs. **The main data types of LISP objects** and their interrelationships are shown in the following diagram:



As shown above, **an atom** is a number, a symbol or a character. **A number** is a sequence of digits, possibly, involving a dot and the letter E appropriately so that the value is either an integer or a real (decimal) number. The following are **examples of numbers:**

11.5E–3 –14.E3

A **symbol** is any string of characters which does not represent a number and does not include parentheses and quotes. **However, each of the expressions, '( )' and 'nil', represents an empty list, and, is both an atom as well as a list**. **Each of the following is a symbol**

3 + 11 *Name* BLOCK#8

Note that 3 + 11 is not a number but a symbol. The reason for this is that any arithmetic expression involving at least one operator, when represented in LISP, has to be a list with first and the last characters as '(' and ')' respectively, and first element within the list being necessarily an operator. If we intend to represent the arithmetic expression 3 + 11 which is equivalent to 14, then it is represented as (+ 3 11).

String: A sequence of characters enclosed within double quotes is a string, e.g.,

"abc 2 @ string "
is a string

However, the following is *not* a string

"square of two is four

because right double quote is missing.

1.4 THE EVAL FUNCTION

Initially LISP was designed as an *interpreted* language, though later compiler based versions also became available. In the interpreted mode, the prompt visible on the screen is the symbol '→' , which, of course, may be changed.

LISP environment provides an inbuilt function called *eval* **or commonly known as** *read–eval–print loop*. Any legal LISP object, i.e., **an S-expr** typed after the prompt is considered both as an S-expr and as an input to the read–eval–print loop. The S–expr *is evaluated* according to the rules to be explained and then printed. The value, so obtained, is also an S–expr, provided that the input is an S–expr. Else, error will be printed.

Next, we explain, in some detail, how LISP expressions are evaluated.

(i) A **quoted expression** is evaluated to the expression obtained by removing the quote. For example,

The following expression
→ '(+ x (* y z))
evaluates to the expression

→ (+ x (* y z))

→ 'Colour
; *where colour is a symbol we get respectively the*
; following situations after printing
→ colour

(ii) **For evaluating a symbol** say *colour*, first of all, some value or binding must have been associated at some stage before the current evaluation. And then evaluation of the symbol returns the associated value or binding. Suppose, earlier at some stage, the symbol *colour* is given the value RED, then, if give the input colour, i.e., if we have
→ colour
;then RED is returned, i.e., we get after evaluation
→ RED

(iii) **Evaluating a number:** A number evaluates to itself
For example if the input is given as
→ 41
;then after evaluation the number 41 is returned.

(iv) **Evaluating a function application**

A list of the form

(< *function–name* > < *param1* > < *param2* > …. < *paramk* >)
with < *function–name* > being an atom and the name of a function, and < parami > being an S–expr for each i, is called a **function application**. For evaluating a function application, first < param1 > is evaluated to get arg1. Similarly < parami > is evaluated to get argi, first for i =2, then for i = 3, and so on. Once all the arguments are evaluated then the function is applied to the arguments. Finally the value so obtained is printed. For example, if the following expression is given as input
→ (* 4 11)

as 4 evaluates to 4 and 11 evaluates to 11, therefore 44 is returned and printed. Next suppose, the symbol *x is bound to 2 and y to 5* then for the input

→ (+ (* x 3) (– y 1))

first *(* x 3)* is evaluated, which in turn requires evaluation of symbol x. The symbol x being bound to 2 is evaluated as 2. Thus, the expression (* x 3) evaluates to 6. Similarly, the expression (– y 1) evaluates to 4 and finally the whole of the above expression evaluates to 10.

Next, assume *sum-sq* is a function defined in a program which returns the sum of the squares of its arguments and again suppose *x is bound to 2 and y is bound to 5*, then if we have the following expression is given as input

→ (sum-sq (* x 3) (– y 1))

the expression is evaluated and printed as
→ 52.
on the monitor

(v) **Evaluating a special form**
A list of the form
(< *special-word* > < *param 1* > < *param 2* > …… < *param k* >)

with < *special-word* > being a special word in LISP (*to be discussed*) and < *parami* > an S-expr, is called a special form. The evaluation of special form depends upon the special word. The parameters < *parami* > may or may not be evaluated depending upon the
< *special-word* >.

Some of the well-known special words are: **defun, cond, do, quote.** We shall discuss evaluation of these special forms at appropriate place. The special form *(quote x) is just equivalent to* 'x and hence evaluates to x.

→ (quote (* 3 7));
evaluates to
→ (* 3 7)

Note that '(* 3 7) or (quote (* 3 7)) evaluates to (* 3 7) and not to 21

(vi) **Evaluating a list of the form**

(< *non-atom* > < *param1* ….. < *paramk* >),
where < *non-atom* > is an S-expr is evaluated as follows :
Each of the parameter < *parami* > is evaluated yielding arguments and then the S-expression < *non-atom* > is applied to the evaluated arguments. We shall discuss examples of such evaluations later.

(vii) Certain atoms have preassigned meaning or evaluations as follows in LISP, therefore, should not be used for other purposes.

Atom	**Associated meaning**
t	logical value *true*
nil	denotes either logical value *false* or the empty list ( ) depending on the context.

The special symbol Setq

Before coming to evaluation of functions in LISP we consider evaluation of special-form for the special word SETQ. It binds symbols to values and will be useful in explaining evaluation of other functional expressions. The special word *setq* takes two parameters, the first a *symbol* and the second an *S-expr*. The first parameter is not evaluated. *Actually SETQ may be taken as shorthand for SET QUOTE. And, as we have explained earlier, QUOTE EXPRESSION evaluates to EXPRESSION and not to the value of EXPRESSION. e.g. (QUOTE 3 + 5) evaluates to 3 + 5 and not to 8).* The second parameter is evaluated and the value so obtained is bound to the symbol represented by first parameter, e.g., the S-expr

(*setq x 27*) binds 27 to x and also the value 27 is returned
(*setq x (* 3 5*)) binds 15 to x and returns the value 15,
(*setq x'(* 3 5*)) binds the list (* 35) to x and also returns the list (*3 5) and not 15.

We have already mentioned that the action of *return* includes printing of the returned value. Also the action of *return* includes the fact that returned value can be utillised in further processing, e.g., the S-expr

(+ (setq x 7) (setq y 3))

not only binds x to 7 and y to 3 but also the values 7 and 3 respectively are used in further evaluation and

(+ (setq x 7) (setq y 3))

returns 10, in addition to binding x to 7 and y to 3.

Sometimes the pairs of arguments to several occurrences of SETQ are run together and given to a single SETQ. In such a situation, odd-numbered arguments are not evaluated and even-numbered arguments are evaluated. Further each even-numbered value is associated or bound to the immediately preceding (or bound to the immediately preceding) odd numbered argument, e.g., the S-expr

(setq x '(1 2) y 7 z 11)

binds the list (1 2) to x, 7 to y and 11 to z.
associates or binds (12) to x, 7 to y and 11 to z. It may be noted that the list (12) and the number 12 is associated with x.

Ex 1: Explain the effect of execution of the following statements:
(i) (* (+ (setq x 5) x) (+ (setq y 7) y))
(ii) '(– (+ setq p 9) (setq s 3)) (* p s))

1.5 EVALUATION OF PRIMITIVE FUNCTIONS

LISP has a number of basic functions. In this section, we discuss how S-exprs involving these primitive functions are evaluated.

We have already mentioned that LISP uses prefix notation for representing functional expressions.

We explain of evaluation of LISP objects through examples, preceded by some explanatory remarks, if required.

(i) Some Primitive Numeric Functions: As numeric evaluation is self-evident, therefore, the following table is sufficient for the purpose

Function call	**Value**	**Comments**
(+ 3 4 8 11 –2)	24	+ or plus can take any finite
(plus 3 4 8 11 –2)	24	number of appropriate arguments
(– 13 55)	–42	differences or – takes exactly two arguments
(difference 13 53)	–42	
(* 2 3 4)	24	times or * may take any finite
(times 2 3 4)	24	number of appropriate arguments
(/ 9 3)	3	quotient or '/ 'takes exactly two arguments
(quotient 9 3)	3	
(abs –5.7)	5.7	only one argument
(abs 5.7)	5.7	
(expt 3 2)	9	exponential function
(sqrt 4.0)	2.0	positive square-root
(max 8 11 9 7)	11	any finite number of appropriate arguments
(min 8 11 9 7)	7	
(turncate 14 4)	3	turncate returns the quotient in integer division neglecting the remainder;
(rem 14 4)	2	rem returns the remainder on division
(round 14.3)	14	round returns the integer nearest to its
(round 14.6)	15	argument.
(float 14)	14.0	integer argument; returns real.

Ex 2: Evaluate the following:
(i) (+ (* 2 3 4) (– 8 9) (truncate (15 7)))
(ii) (* (rem 17 6) (truncate 8 9) (max 5 9 11))

Ex 3: Write a function *division* which divides a number X by Y such that if Y = 0 then the function returns the symbol 'infinity' else it returns the quotient X/Y.

1.6 PRIMITIVE LIST MANIPULATION FUNCTIONS

We have already mentioned that the programming language LISP is mainly designed for symbolic processing though it may be used for numeric purposes also. Symbolic processing in LISP is mainly about manipulating lists. **Here we consider main list processing operations.**

(i) **car:** It takes a list as an argument and returns the first element of the list. (car ’(d b c)) returns the element d, (car ’((d b) c)) returns (d b). We should note that argument has to be a quoted list. This is in consonance with our earlier statement that for all functions denoted by an atom, the parameters are evaluated to return arguments for the function.

Thus for evaluating (car ’((d b) c)), first of all its parameter viz. ’((d b) c) is evaluated. Also we mentioned earlier any quoted expression is evaluated to its unquoted part i.e., ’((d b) c) evaluates to the list ((d b) c). And the first element ((d b) c) is (d b) . Hence (d b) is returned.

Further

(car (car ’((d b) c))) is evaluated to the atom d.
(car ’(LISP IS AN AI LANGUAGE))

returns the symbol LISP.

If the statement (setq x ’(a b c)) is followed by the statement (car x) then the symbol a is returned.

(ii) **cdr** (*pronounced as ‘KUDDR’*) also takes a list as its argument and returns a list obtained from the given list by deleting its first element e.g.

(cdr ’(d b c)) returns the list (b c)
(cdr ’((d b) c)) returns the list (c)
(cdr ’(LISP IS AN AI LANGUAGE)) returns the list

(IS AN AI LANGUAGE).

Also if the statement (setq x ’(a b c)) is followed by the statement (cdr x) then the value returned is (b c).

But note (cdr ’x) returns error, because, ’x evaluates to x and not to the list (a b c) and the cdr of a symbol, in this case x, is not defined.

Sequence of CARs and CDRs

```
( car ( cdr ( cdr ( cdr ’( person ( Name Raj )
                          ( Residence K–76 HauzKhas )
                          ( City New Delhi )
          ) ) ) ) )
returns  K – 76
```

LISP provides facilities to simplify notation for a sequence of cars and cdrs e.g. the S-expr

(*car* (*car* (*cdr* (*car* (*cdr* (*cdr given-list*)))))).

where given-list is bound to some valid list, can be simplified to
(caadaddr given-list)
or to any other S-expr like
(caar (cdadr (cdr given-list))).

Remark: The functions *car* and *cdr* take things apart. Next we describe three functions *cons*, *list* and *append* which put things together.

(iii) **Cons**: takes two arguments, the first may be any (valid) S-expr but the second must be a list. Then *cons* returns a new list in which the first argument is the first element in the returned list followed by the elements of the list given as second argument, preserving the earlier order of occurrence in the second argument.

(Cons ' * ' (4 7)) returns the list (* 4 7)
(Cons ' (a b) ' (b c)) returns the list ((a b) b c)
(Cons 'a nil) returns the list (a)

Also, if S-exprs (setq x ' (a b)) and (setq y ' (c d)) are followed by the S-expr (Cons y x) then the list ((c d) a b) is returned but (Cons 'y x) returns the list (y a b).

(iv) **list**: may take any number of parameters, each of which is an S-expr, it evaluates the parameters and then the arguments so obtained are grouped into a list in the same order as the corresponding parameters are given initially, e.g., if the inputs (setq x ' (a b)) and (setq y 'intelligence) are followed by the input s-expr (list 'x y x 'knowledge 'y) then on evaluation of the last expr, we get the list (x intelligence (a b) knowledge y)

(v) **Append**: the parameters of the function append can not be arbitrary S-exprs but must evaluate to lists. It removes the parenthesis from the arguments obtained by evaluating the parameters and puts all the lisp objects so obtained into a list. For example, if the S-exprs (setq x ' (a b)) and (setq y ' (c d)) are followed by (append x y ' ((a b)))
then the last S-expr on evaluation returns (a b c d (a b). If we try to evaluate (append ' x y) then an error is returned, because value of 'x is a symbol x and a symbol can not be an argument of append.

Next, we define some more built-in list processing functions, which are quite useful in writing LISP programs for solving problems requiring symbolic processing.

(vi) **Reverse**: takes a list as its argument and reverses the top level elements of the argument e.g.

(reverse ' (a b (a b) (c d)))
returns
((c d) (a b) b a)

(vii) **Length:** again takes a list as its argument and returns the number of the top level elements, e.g.,
(length ' (a (a b) (c d) e)) returns the number 4.

(viii) **Last:** again takes a list as argument and returns the last top-level element of the list, e.g.,

(last ' (a b (c d))) returns the list (c d).

(ix) **Subst (***stands for substitution***):** takes three arguments, such that each occurrence of second argument in the third argument are replaced by the first argument. Second argument must be an atom, e.g.,

(subst ’ A ’B ’ (D B A))
returns
(D A A)
Also
(subst ’ (A B) ’C ’ (D B A C))
returns
(D B A (A B))

(x) **eval:** In some situations, we may need another evaluation, in addition to the evaluation provided by *read –eval–print loop*. The function eval is explained with following examples:

→ (setq x ’y)
→ (setq y z) ; then x evaluates to y but (eval x) evaluates to z.

Ex 4: Explain the sequence of steps of evaluation of the following LISP expression:
(length (append (setq x ’(a b)) ’(c d) (reverse (sublist’x ’(s t) ’(u v x)))))

1.7 BUILT-IN PREDICATES

Predicates are functions which return nil or t depending upon the values of their arguments. The evaluation by some important predicates is explained in the following table:

Function call	**Value Returned**	**Comments**
(> 7 (+ 2 3))	t	normal ‘greater than’ relation
(< 7 3)	nil	normal ‘less than’ relation
(= (* 3 7) (+ 16 5)	t	‘ = ’ tests equality of only numbers
(= equal ’(one two) ’(one two))	t	‘equal’ tests
(equal ’(two one) ’ (one two))	nil	equality of any two S-exprs,
(equal ’ (a b) (car ((a b) c)))	t	
(evenp (+ 4 7))	nil	returns t if arguments is an integer
(evenp (* 2 5))	t	and is even, else returns nil,
(numberp 2.1416)	t	returns t if parameter evaluates to a
(numberp ’x)	nil	number else evaluates to nil.

Further if we have (setq fifty 50) before the next S-expr then

(numberp fifty)	t	

But

(number ’fifty) nil

(zerop 0) t

if we have (setq x 0) then

(zerop x) t

But

(zerop ’x) nil

(zerop .00001) nil

The predicate ‘null’: null returns t if its argument is nil else returns nil, e.g.,

(null ( )) returns t
(null ’man) returns nil
(null ’ (a b)) returns nil

The predicate ‘member’: *The predicate ‘member’ has a little different behaviour. It may not return t and/or nil*. The predicate *member* tests an atom for the membership of a list. If an atom is not a member of the list then it returns nil, else it returns the portion of the list starting with the atom in the list up to the last element of the list. For example, if we define

(setq last-alphabet ’(u v w x y z))
; then
(member ’a last-alphabet) returns nil
(member ’w last-alphabet) returns (w x y z)

However, the predicate *member* tests the atom only for *top membership* of the argument list. Hence,

(member ’w ’ (u v (w x y) z) returns nil,

because w is not a member of the list given by the second argument, but w is a member of a member, viz of (w x y) of the list given by the second argument.

The predicate eql: We considered two forms of predicates for equality viz. ‘*equal*’ and ‘=’. We consider another predicate *eql* for equality. The predicate *eql* checks the equality of the internal structure of its arguments. If the *structures* of arguments are identical, it returns t else nil. In order to explain the behaviour of eql, we need the following additional information:

Each time we use the *function list* even with the same elements, it takes new memory cells and creates the list. Thus the two s-exprs,

(setq list1 (list ’x ’y ’z))
and
(setq list2 (list ’x ’y ’z))

creates two lists viz list1 and list2 which *are internally different* though each of them consists of the same three elements x, y and z. However, further we have

(setq list3 list1)

then list3 and list1 point to the same memory locations and hence

```
( eql   list1   list2 )   returns  nil
( eql   list1   list3 )   returns  t
( eql   list2   list3 )   returns nil
```

1.8 LOGICAL OPERATORS: AND, OR and NOT

The main differences in the behaviour of the logical operators in LISP from their behaviour in Boolean Algebra or in some other/programming languages are:

i) The operators AND and OR may take one, two or more than two arguments

ii) The values operated by logical operators in LISP are not exactly *true* or *false*

but the values are *nil* and *non-nil*, i.e., in LISP any S-expr which is not nil has the same logical status as that of *true* in Boolean Algebra. Hence, modified definitions of the three logical operators are:

NOT: Not of nil is t, and, Not of non-nil is nil or ( )
For example, (not ' (a b)) is nil
(not ( )) is t

AND and OR are treated as special forms as described below:

AND: The arguments of AND are evaluated from left to right until some S-expr evaluates to nil then other arguments are not evaluated. If, at any stage, an argument evaluates to nil, then nil is returned. However, if none of the arguments evaluates to nil then the value of the last argument is returned.

OR: The arguments of OR are evaluated from left to right, until either some value returned is non- nil, then the value is returned as the value of application of OR and the rest of the arguments are not evaluated. However, every argument evaluates to nil, then nil is returned.

Examples:

```
( not   nil )                  returns t
( not t )                      returns nil
( not     'dog )               returns nil
( and   t     'dog )           returns dog
( and   t     nil       'dog ) returns nil
( or      t     nil         'dog )  returns t ; first argument that is
                                    non-nil, if any
( or      'dog     nil      t )    returns dog;
( or      'dog )                  returns  dog
( and     'dog )                  returns  dog
( or nil ( ) )                    returns nil.
```

1.9 EVALUATION OF SPECIAL FORMS INVOLVING DEFUN and COND

So far we have discussed only built-in functions including predicates and relations. The special word DEFUN allows us to write our own functions and build our own programs. **To build up highly complex programs, we need**

(i) **iteration**, i.e., repeated execution of a sequence of statements, and

(ii) **Selection.**
In LISP, capability for iteration is provided by the *Do construct*. On the other hand, the selection capability in LISP is provided by COND.

(i) **Functions are defined using the syntax:**

(*defun* < *function-name* > < *parameter-list* >
< *function-body* >)

where < *function-name* > is a symbol which names the function being defined, < *parameter-list* > is a list of distinct symbols, which forms a list of parameters to the function and < *function-body* > is the sequence of S-exprs which describes (or denotes) the desired computation.

Simple examples: The LISP function, corresponding to the mathematical function

$f(x, y) = x^3 + y^3$ for all x, y, is given by the function definition

(*defun sumcube* (*x y*)
(+ (* x x x) (* y y y)))

Note 1: We have already mentioned that in interpreter mode, every S-expr, which appears after the LISP prompt '→' is read, evaluated and printed. The execution of an S-expr that defines a function, **returns the name of the function**. The name, acting as a symbol, has a value obtained through execution, associated with it and the associated value can be used in further processing. For example,

→ (*defun sumcube* (*x y*) (+ (* *x x*) (* *y y*)); *returns*
; *sumcube*
Next S-expr

→ (*length* (*list* (*defun sumcube* (*x y*) (+ (* *x x*) (* *y y*))
; returns
;*the integer 1* because (defun ….) returns symbol sumcube then (list …) returns the
;list (sumcube) and finally (length …….) returns the integer 1.

Note 2: Applying a function to its arguments is termed as making a **function call**. For the above definition of the function sumcube, the following sequence

→ (setq x 3) ; returns 3
→ (sumcube 2 x) ; returns 2 * 2 * 2 + 3 * 3 * 3, i.e. 35

The capability of conditional or selective evaluation is provided by the special symbol COND.

The syntax (or legal form) for COND is:

(*cond*
(< *test-1* > < *S-expr* > < *S-expr* > … < *S-expr* >)
(< *test-2* > < *S-expr* > < *S-expr* > … < *S-expr* >)
(< *test-n* > < *S-expr* > < *S-expr* > … < *S-expr* >)
)

Each list of the form (< test-i > < S-expr > . . . < S-expr >) in the above is called a **clause**.
The COND form is evaluated according to the following rule :

Evaluate the first test viz. < test-1 > in clause 1. If it evaluates to non-nil then the remaining < S-expr > 's in the clause are evaluated in the order from left to right and the value of the whole COND is the same as the **value of the last S-expr in clause1**. If < test-1 > evaluates to nil, the same sequence of steps is repeated for second clause, and so on. Until either some < test-i > evaluates to non-nil, for which earlier described sequence of steps for the non-nil case, is followed. However, if all < test-i > evaluate to nil then COND form evaluates to nil.

A special case of COND form is the one in which < test-i > is the first test which invariably evaluates to non-nil and the corresponding ith clause has no other S-expr, i.e., ith clause is of the form (< test-i >).

In this case, the value of < test-i > which is assumed to be non-nil is returned.

Examples of COND special form

```
(   defun       our-max-3          ( x y z )
        (    cond
                (   ( > x y )
                        ( cond
                        (    ( > x z ) x )
                        ( t         z)
                    )
                )
                        (     ( > y z ) y )
                          ( t         z )
                        )
        )
)
```

Comment (a): The example given above graphically demonstrates a style of placing corresponding parentheses in the code. *This is allowed as LISP code is format free*.

Comment (b): We also know that t evaluates to itself and hence is non-nil. Therefore, the two occurrences of the clause (t z) state that in case (> x y) is true but (> x z) is false then z is returned. Similarly, if (> x y) is false then the next clause as given below is executed

```
        (
                (         ( > y     z ) y )
                (         t              z )
        )
```

In this clause, first of all, condition (> y z) is evaluted which, if the value happens to be nil then the condition t in the next sub-clause (t z) is tested. As t always evaluates to non-nil, hence z is returned.

Comment (c): As z is a number and a number always evaluates to non-nil, therefore, each of the two occurrences of (t z) in the definition of our-max-3 can be replaced by just (z).

1.10 THE SPECIAL FORMS *DO* AND *LET*

The special form Do provides the power of iteration to LISP. We may recall from our earlier studies that iterative constructs are very useful, specially for denoting long sequences of actions by shorter code.

Also, LET is special form useful in LISP because, it facilitates the creation of local variables and often yields code which is both compact and efficient. *Let us first discuss the special form Do in detail.*

The do construct has the following form:

```
(
( do
      (   ( var1   init1     step1 )
          ( var2   init2     step2 )
          ( vari   initi     stepi )
          ( varm   initm     stepi )
      )   ; this part initiates various variables
          ; also specifies the possible modifications to the value of var-i through step-i
( end-test                   ; test to see if Do loop

end-form1                    ; is to be terminated


end-form-n return-value
)
body1                        ; body of Do-loop
body2


body-n ; where each body i is an S-exp.
) )
```

The above Do form is evaluated through the following sequence of steps:

Step 1: Variables var-i are initially bound to init-i for all i in parallel

Step 2: If the S-expr viz *end-test* is present, it is examined. If end-test evaluates to nil then the following sub-steps are followed:

(i) Each of body-j is evaluated, and if in body-j any S-expr of the form (*return value*) is encountered, *do* is exited and its value is the *value* in return value.

(ii) We should note that only utility of the body is for exiting or for side-effects.

(iii) Next iteration starts (only if end-test evaluated as nil) with binding of each of the var-i to the value of step-i. If step-i is omitted, the var-i is left unchanged.

(iv) Repeat whole of step2 again if in step2, end-test evaluates to nil else go to step3

The next two steps are executed when end-test evaluates to non-nil.

Step 3: Each of end-form-i is evaluated, the utility of which is for exiting or side-effects.

Step 4: Return-value is evaluated and is returned as value of DO loop.

To illustrate the applications of Do construct, we solve two problems using the construct.

Example 1: to print the first n natural numbers, where n is a parameter to the function, The required function in LISP is:

```
→       ( defun print-beginning-integers (n)
```

```
                ( do
                        (
                        ( count 1 ( + 1 count ) )
                        )
                        ( ( equal count n ) 'done )
                        ( print count )
                )
        )

→       ( print-beginnning-integers 12 )

        123456789101112 done
```

; here *done* is used to indicate the successful

; completion of the loop. The I/O function

; print shall be explained later.

Next we explain the LET construct:

As we have already mentioned that LET is used for creating local variables. The purpose and use of LET may be explained through the simple example:

```
( defun  explain–let ( x y )
        ( let
                (
                   ( x     1 )
                   ( y     2 )
                )
                ( print 'x = x )
                ( terpri  )
                ( print ' y = y)
                ( terpi  )
        )
        ( print 'x = x )
        ( print 'y = y )
```

The printing command (terpri) asks the printer to leave the line and start printing on the next line.

Let us call *explain–let* with x and y respectively as 8 and 9
→ (explain–let 8 9) ; we get

x = 1
y = 2

(∵ through let, we define a local loop in which x is 1 and y is 2. But, once execution exits the loop, then x and y assume the assigned values.

x = 8
y = 9

Remarks: As in Do, the values of the variables within LET structure are bound in parallel. If we wish to bind values in sequential order then we use LET *

Ex 5: Write a LISP program *expo* to compute i raise to power j where i and j are natural numbers.

1.11 INPUT / OUTPUT PRIMITIVES

In this section, some frequently used I/O primitives are introduced example:

(i) The *read* statement for input is explained through the following:
→ (+ 7 (read));
→ 8 ; the value given by the user;
→ 15 ; the value returned by the system.
(ii) The primitive TERPRI directs the printer to start on a new line.
(iii) The primitive PRINT takes one argument. It prints its argument in the same form in which it is received and also returns the argument as value of the print statement.

Example :

→ (print '(x y z u)); returns two lines of (X Y Z U) as shown below.
(X Y Z U)
(X Y Z U)

One of the occurrences of (X Y Z U) is for the value returned by the function *print* and the other occurrence is printed by the *print statement.*

Another Example:

→ (print 'good morning'); the statement an execution returns the following two
;lines
'good morning'
'good morning'

(iv) The primitive prin1 is the same as print except that the new-line characters and space are not provided

Example

(print 'left) (print 'right)
left
right
→ (print 'left) (prin1 'right)
left right
→

(v) ***The primitive princ*** **:** is the same as *prin1* except it does not print the double-quotation marks at of the beginning and end of its argument, if given, e.g.,

→ (princ "good morning")
good morning "good morning"
→

princ statement has eliminated the double quotes, but returned value is still having double quotes.

(vi) **Formatted Printing through the primitive FORMAT**. LISP allows us to have cleaner output through the primitive FORMAT which has the following syntax.

(*format* < *destination* > < *string* > *arg1 arg2* ……)

Here < *destination* > specifies where the output is to be directed, e.g., to the printer or to the monitor or some other external file. Default value is generally the monitor. The word < string > in the format clause indicates the desired output string which is mixed up with format directives. The format directives specify how each argument is to be represented. The order of occurrence of the directives is the same as the order in which the arguments are to be printed. The character ~ is used before each directive to identify the directives. **Most common directives are:**

~ % : indicates that new line is to be printed.
~ A : is a placeholder for a value which will be printed as if print were used,
~ S : is a placeholder for a value which will be printed as if prin1 is used,
~ D : is a place holder for a value which must be an integer and which will be printed as a decimal number,
~ F : is a place holder for a value which must be a floating point number and will be printed as a decimal floating point number,
~ C : is a place holder for a value which will be printed as character output,

Next, field widths for appropriate argument values are specified by an integer between tilde (i.e., ~) and the directive, e.g., ~ 3D indicates the integer field width is 3.

1.12 RECURSION IN LISP

Recursion: LISP expresses recursive computation in a very natural way.
Let us first recall below what is a recursive function:

Recursive function: is a function which calls itself repeatedly, but each call with simpler arguments than the arguments used by the preceding call.

Definition of a recursive function requires
(i) a recursive step.
(ii) stopping condition for stopping the processing.

One of the most well-known example of recursive definition in Mathematics is that of factorial, which in the following definition is named as ***fact*** *and is defined as follows:*

fact (1) = 1 (*the stopping/termination condition*)
fact (n) = n*fact (n-1) (*the recursive step*)

In LISP, the above function is expressed as

```
→ (defun fact (n)
        ( cond
                (
                        ( ( = n 1) 1 )
                ( t ( * n fact ( –   n 1)
                    )
                )
                )
        )
)
```

FACT

→ (fact 5)

Another simple example is given below to explain recursion in LISP:

We define our own function LEN that returns the number of top-most elements in a given list say L:

```
( defun LEN ( L )
        ( Cond
                (
                        (  ( null L ) 0 )
                          ( t ( + 1 ( LEN ( Cdr L ) ) ) )
                        )
                )
        )
)
```

Ex 6: Write a function *deep-length* that counts the number of a atoms (not necessarily distinct) in a given list. The atoms may be in a list which is a member of another list which at some level occurs as an element of the given list. For example, for the list

L = (1 (2 (3 4)) (5))

length L is three. However, deep-length of L is five.

1.13 ASSOCIATION LIST AND PROPERTY LIST

Association lists are useful tools to associate attributes and their values with objects. For example, to describe a particular book *viz. LISP by Winston & Horn published by Addison-Wesley Publishing Company in 1984*, we may use the representation:

```
( setq book ' ( ( title ( LISP second edition ) )
                    ( author ( Winston & Horn ) )
                    (year 1984 )
                    ( publisher addison-wesley)
                )
)
```

The value of the attribute *title* is a list viz. (*LISP second edition*) whereas the value of the attribute year is *1984*. The values of the attributes may be any S-expr, e.g., number, symbol or list. *Formally we define.*

Association List is a list of embedded sublists, in which first element of each sublist is a key. In the example of book given above, the symbols *title, author, year* and *publisher* are keys.

The procedure ASSOC: to retrieve values of keys or attributes, the procedure ASSOC takes two arguments viz *the key* and *the object-name* and returns the list of two element, the first element is the key and the second element is the associated value of the key, e.g.

```
→ (ASSOC 'year book ); returns
( year 1984 )
```

For a given object, ASSOC looks down the sublist (*each sublist representing key* s *associated value of the key*) starting from the first sublist in the list, and matches the car of each sublist with the key given as first argument of ASSOC. If the two do not

match, ASSOC goes further down to next sublist. However, if the key and the car of the sublist match then whole of the sublist is returned.

Property List, and the primitives GET, PUTPROP and SETF

Another way of associating properties and their associated values to objects in LISP, is through property lists. Considering again the earlier example of book with title as *LISP*, author as *Winston & Horn*, Publisher as *Addison-Wesley*, year as *1984*, we can put this information in the database using the above-mentioned primitives as follows:

→ (putprop ’book ’LISP ’Title) ; putprop returns the attribute values LISP

→ (putprop ’book ’ (Winston & Horn) ’Author)

’ (Winston & Horn)

→ (putprop ’book ’AddisonWesley ’Publisher)

Addison-wesley

→ (putprop ’book 1984 ‘year)
1984

The general form of putprop statement in LISP is

(putprop < object – name – symbol > < attribute – value > < attribute – name >)

where < object – name – symbol > and < attribute – value > must be symbols and < attribute –value > may be any S – expr.

The newer version of COMMON LISP avoid PUTPROP and instead use SETF.

The primitive SETF : SETF is like SETQ. However, SETF is more general than SETQ. The primitive SETF also takes two arguments, but the first argument is allowed to be an **access function** in addition to being an atom. An **access function** includes car, cdr and get. Second argument to SETF is the value, as is in the case of SETQ. The above-mentioned LISP statements using purtprop can equivalently be replaced by the following statements using SETF and GET :

→ (SETF (GET ’book ’Title) ’LISP)
LISP
→ (SETF (GET, book ’ Author) ’ (Winston & Horn))
(Winston & Horn)
→ (SETF (GET ’book ’Publisher) ’Addison-Wesley)
Addison-Wesley
→ (SETF (GET ’book ’year) 1984)
1984

The general format for associating values to attributes of an object using SETF and GET is

(SETF (GET < object – name –symbol > < attribute – name >)
< attribute – value >)

SETF can also be used to replace values of car or cdr of a list as follows:

→ (SETQ L ’ (x y z))

```
        ( x y z )
→ ( SETF (Car L ) 'a )
        ( a y z )
→ ( SETF  (Cdr L ) ' ( u v w ) )
        ( a u v w )
```

In general (*SETF* (*car* < *list* >) < *expr* >) *replaces the car of* < *list* > *by* < *expr* >

and (*SETF* (*cdr* < *list* > < *expr* >))

replaces the cdr of < *list* > *by* < *expr* >

This usage of SETF allows us to change the values of attributes, whenever required.

In order to retrieve values of attributes or properties, the primitive GET is used. Assuming, we have already put the information about the book: LISP by Winston & Horn in the database. Then the relevant information pieces may be retrieved as follows:

```
→ ( GET 'book 'year )
        1984
→ ( GET 'book 'Title )
        LISP
→ ( GET 'book 'Author )
        ( Winston & Horn )
```

etc.

The general format of GET, in order to find the value of the attribute having name as < attribute – name > of the object having name as < object – name >, **is** :

(GET < object – name > < attribute >)

If there is no value in the data-base for < attribute –name > and < object – name >, the value nil is returned.

Note : When more than one SETF or PUTPROP are used to give *different values to the same attribute* of a given object then the effect of *only the latest* remains. Earlier values are overwritten. In order to change values of attributes, we write another statement using SETF or PUTPROP. Thus, in continuation of our example about the book entitled *LISP by Winston & Horn*, if in addition to the earlier statement, we give the following statement:

```
→ ( SETF ( GET 'book 'year ) 1987 ) ; returns
```

the year of publication as 1987, replacing the year 1984.

Optional parameters: So far we have restricted to the definition of those functions which have *fixed number of arguments* that are always evaluated. However, there are situations, in which it may be desirable to define functions with *variable number* of parameters. For example, we want to define a function *exponentiate* such that either the value of *base b* is given or else it is assumed as 10. For this purpose, we use the inbuilt function *expt* which returns m^n for (expt m, n). Thus, *exponentiate* may require one argument or two arguments. For such situations LISP provides a key word *&OPTIONAL*, which when used in parameter-list indicates that the parameters listed

after *&OPTIONAL* may or may not have arguments corresponding to them. If the arguments corresponding to some parameters after *&optional* are present, the function uses these in determining the output result, else there would be no error because of their absence. For example, the function *exponentiate* as defined above can be described in LISP as follows:

```
→ ( defun exponentiate ( n & optinal m )
                ( Cond
                        ( ( null m ) ( expt 10 n ) )
                        ( t ( expt m n) )
                )
  )
        exponentiate
;exponentiate, in the previous line, is the value returned by the definition
```

It may be noted that if *&optinal* keyword had not been available and/or had we defined exponentiate by replacing the parameter-list (n & optional m) by (n m), then there would have been an error if m is not supplied assuming it to be 10.

The key word &rest

In order to explain the use of *&rest*, we consider the following example:

```
→ ( defun our-sum-3 ( n1 n2 & optinal n3)
                (Cond
                        ( ( null n3 ) ( + n1 n2 ) )
                        ( t       ( + n1 n2 n3 ) )
                )
  )
```

Our-sum-3 ; returned by the definition.
Let us make the following calls:

```
→ ( our-sum-3  5  6 )
→  (our-sum-3 )     5 6 7 )
          18
→  (our-sum-3       5 6 8 9 )
   ERROR
```

The above ERROR occurred, because for correct response by our-sum-3, minimum number of arguments in this case must be 2 (i.e., number of parameters before *&optimal* and *maximum number* in this case, must be 3 (i.e. number of all parameter before or after &optimal), but in the last call to our-sum-3, we supplied four arguments viz. 5, 6, 7, and 8.

Now, it is not always possible to remember the exact number of optional parameters and hence not always possible to check erroneous function calls. To remedy this situation, LISP provides for the keyword &REST which is followed by exactly one argument say ‘remaining’. Then if m denotes number of parameters before &optional and n the number of parameters after &optional but before &rest and whenever k arguments are supplied and k > m + n, then all the *remaining (k – (m + n)) arguments are grouped into a list and bound to &rest*. In order to explain the ideas explained above, let us define a function say specialsum-3 as follows.

```
→ (defun specialsum-3       ( n1  n2 &optimal  n3  &rest n4 )
        ( Cond ( ( and ( null n3 ) ( null n4 ) )  (  + n1 n2 ) )
                ( ( null n4 ) ( + n1  n2  n3) )
```

```
                ( t         ( ( + n1 n2 n3 ) ( print n4) ) )
    )
)
; the definition returns.
Special-sum-3

→ (special-sum-3        5 7)
        12
→ ( special-sum-3 5 7 9 )
          21
→ ( special-sum-3 5 7 9 11)
              21 ( 11 )

→ ( special-sum-3 5 7 9 11 12 13 )
          21   ( 11 12 13 )
```

1.14 LAMBDA EXPRESSION, APPLY, FUNCALL AND MAPCAR

When a function is to be called *only once* in a program then we may not like to give a name to the function in the definition of the function. In such a situation, instead of the keyword DEFUN we use the keyword LAMBDA. Rest of the definition of the function remains the same as it would have been under DEFUN. Suppose we need to compute $(x^2 – y^2)^2$, the following LAMBDA expression will accomplish the task:

```
    →  (LAMBDA  (  X  Y  )
            ( * ( – ( * X X  ) ( * Y Y ) )
                ( – ( * X X ) ( * Y Y ) )
           )
    )
```

Application of a LAMBDA expression is similar to that of application of a function under normal definition through DEFUN, e.g.,

```
→        ( ( LAMBDA         ( X Y )
            ( *  ( – (  * X X ) ( * Y Y )
                    ( * X X )  ( * Y Y)
               )
            )
) (3   4); returns
49
```

The functions APPLY and FUNCALL

APPLY takes two arguments, each of which is evaluated. The first argument, which is either a function-name or LAMBDA expression, is applied to second argument which is a list. **FUNCALL** is similar to the function APPLY *with the difference* that arguments are supplied without boundary parentheses of a list. Function-names are preferably quoted with #’ in stead of just quote.

Examples:

```
    ( APPLY #'* ( 2 3 ) )
      6
→ (FUNCALL #'* 2 3 )
      6
```

For the earlier defined function *our-sum-3* which returns sum of 2 or 3 arguments, what ever number of arguments out of 2 or 3, are supplied. Let us consider

→ (APPLY #' our-sum-3 (4 5))
9
→ (FUNCALL #' our-sum-3 4 5 6)
15
→ (APPLY #' (LAMBDA (X Y) (* (+ x x) (+ y y))) (3 4))
48
→ (funcall #' (LAMBDA (X Y) (* (+ X X) (+ Y Y))) 3 4)
48

The Backquote facility : The backquote is just like quoted expression and evaluates to itself except the following difference : Those subexpressions of the expression that are preceded by a comma or by the comma followed by the symbol @ are evaluated and substituted appropriately before returning the result. Let A be bound to
' (3 x 4) then

→ ' (A B C) ; evaluates to
(A B C)

→ ' (,A B C) ; evalutes to
((3 x 4) B C)

→ ' (A ,A B ,@ A C) ; evaluates to
(A (3 x 4) B 3 x 4 c)

Explanation of evaluation of backquoted expressions: Any subexpression *which is preceded by a comma is evaluated and substituted by the value so obtained. Further, if a subexpression is preceded by , @ then it is evaluated and splice substituted. i.e., substitution after removing bounding parentheses, where a the result of evaluation of the argument must be a list.*

The function MAPCAR: MAPCAR is a useful function which is to be repeatedly applied to a set of lists where each list constitutes one argument list for the function. Let f be a function of arity k, i.e., the function f requires k arguments in one application.

The application of mapcar is explained through the following examples.
→ (mapcar # ' + ' (1 2 3 4) ' (3 4))
10 7
(i.e., elements of each of the two lists are added separately)
→ (mapcar #' (lambda (x) (+ x 7)) ' (2 3 4)
; returns
9 10 11
→ (mapcar #' list' (x y z) (a b c))
;returns
(x a) (y b) (z c)

1.15 SYMBOL, OBJECT, VARIABLE AND REPRESENTATION

In the contexts in which a symbol has or is expected to have some object or S – expr associated with it, it is called a **variable**. The symbol book1 becomes a variable, when associated with the object which represents a book entitled *LISP*, authored by

Winston & Horn in the year *1984*. The association between the symbol and the object may be achieved through the LISP statement:

```
→ ( setq book1' ( ( title LISP )  ( author ( Winoston & Horn ) )
                ( year 1984 ) ( printing first 1984 )
    )  ; returns
book1
```

The associated object may be referred to as the value of the symbol. The variable may be considered as the ordered pair: (symbol, value).

Also, a symbol used in the parameter list of a function definition, though does not have any associated value or object at the time of definition, yet is a variable because it is expected to be associated with some object at the time of application of the function.

Bound & Free Variables: A symbol that appears in the parameter list of a procedure, is called a *bound* variable w.r.t the procedure. A symbol, that does not appear in the parameter list of a procedure, is called a *free* variable w.r.t the procedure.

Representation of Symbols: In LISP environment, the link between a symbol and the associated object is unique and is achieved through the following mechanism.

LISP system maintains a Symbol Table (in some part of the memory) in which each symbol, when encountered for the first time, is entered alongwith some starting address, say 3000, of some location in the memory where the associated object is stored. We may note that some of the components of the object may be changed over time, e.g., if the copies of book1 are again printed in the in the year 1988. In such a case, the component ' (printing first 1984)' of the object is changed by ' (printing second 1988)'. However, the entry (book1 3000) remains unchanged. Next time when book1 occurs in a program, the LISP system searches through its possible occurrences in the symbol table and on finding it there, does not attempt to associate with it another address or location in memory. Further, the statements like,

```
( setq book2 ' book1 ) and
( setq book3 ' book1 )
```

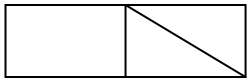
will associate address 3000 (i.e. the address associated with book1) with symbols book2 and book3 as their address parts of (symbol, address) pairs. Thus, we may also say that a variable is an ordered pair (symbol, pointer).

The Predicate EQ : EQ returns t if and only if *the internal structures* of its arguments are identical. Hence, continuing with the earlier discussion, the value t is returned in all the following three cases:

```
( EQ ' book1 ' book1 )
( EQ ' book1 ' book2 )
( EQ ' book2 ' book3 )
```

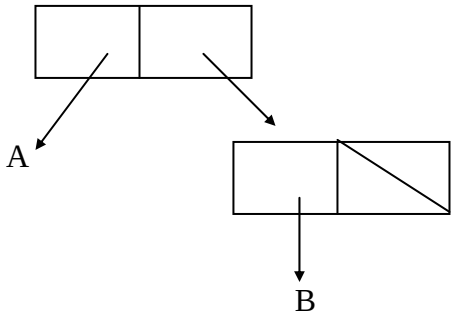
Internal Representation of Lists, cons-cells in memory and the data structure Dotted Pair: We pointed out that **in order to store symbols**, LISP system marks a section of memory, called symbol table and which is considered as composed of cells to store (symbol, pointer) pairs. The first component of each cell is interpreted as a symbol and the second component as an address. Similarly, **in order to store lists**, LISP system marks out a segment of memory constituting of what is known as **cons-cell**. Each cons-cell is a pair (*pointer, pointer*), i.e., each cons-cell contents are interpreted as pair of pointers or addresses. The first pointer, in the cons cell to a list

say L, points to the location in memory where information about CAR of L can be found and second component of a cons cell points to a location where information about CDR of L can be found. The cons-cell may be diagrammatically represented as in the diagrams below, where the left arrow points to CAR and right arrow points to CDR of the list to be represented by the cons-cell. The CDR of a single element list is nil and is represented by the cons-box, where we may call the boxes of the form as cons boxes.

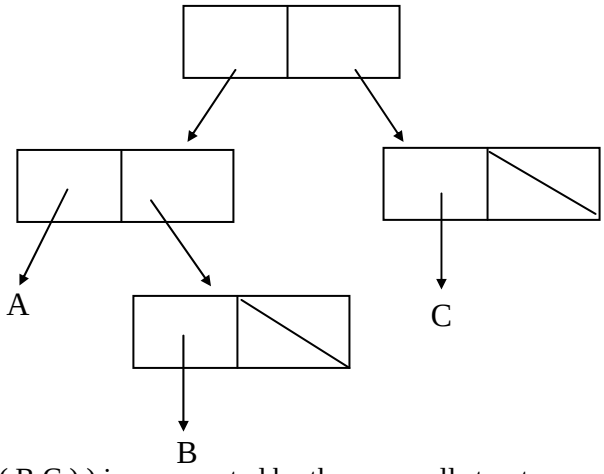


Example of representation of lists by cons cells

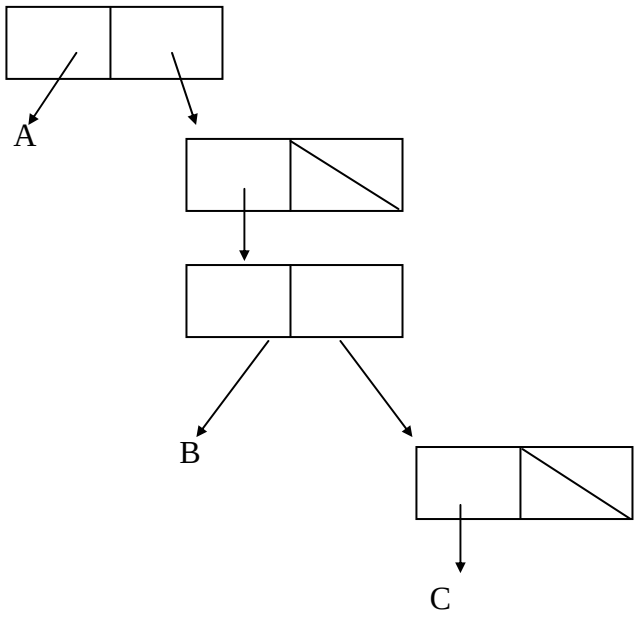
1. The list (A B) is represented by the cons cell structure:



2. The list ((A B) C) is represented by the cons cell structure:

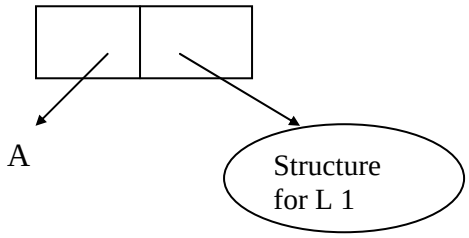


3. The list (A (B C)) is represented by the cons cell structure:

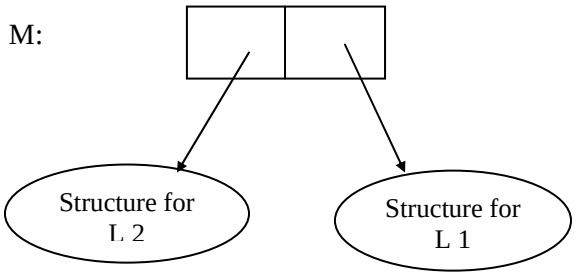


Remarks: The name cons in the cons-cell structures, is justified on the following grounds:

Let L1 be a list and A be an atom and we have LISP statement (*setq L (Cons ' A L1*)) then L is represented by adding one cons cell in the memory as shown below:



Also if L2 is another list then, on the command, (*setq M (Cons L2 L1*)) resultant list M is obtained by adding one cons cell in memory as shown below:



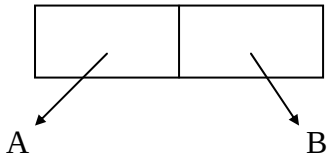
It can be easily seen that using the Cons cells representation for lists, operation like CAR, CDR, ATOM etc can be efficiently implemented.

Ex 7: Draw cons-cell structure for the list ((A B) (C D)).

The Data Structure: Dotted Pair

The cons-cell structure suggests that a cons-cell in memory may represent a LISP object in which the CDR need not be a list but may be an atom or a symbol.

Dotted Pair: A LISP data structure pair is a structure like list with the difference that CDR of a dotted pair may be an atom also. **A dotted pair** with CAR as symbol A and **CDR as symbol B** is **denoted by (A . B)** with spaces around dot on both sides. Thus, cons-box representation for dotted pair (A . B) is



and (CAR ' (A . B)) is the symbol A
and **(CDR ' (A . B)) is the symbol B.**

1.16 DESTRUCTIVE UPDATES, RPLACA, RPLACD & SETF

We have earlier discussed updates of attribute values in property lists through **PUTPROP** and **SETF**. But, so far, we have not discussed primitives, in context of general lists, which achieve *destructive updates* in parts of given lists. SETF may be used for the purpose. However, first we introduce two mnemonics for **replace CAR**

and for **replace CDR**. The mnemonic *RPLACA* is used for *replace CAR* and the mnemonic *RPLACD* is used for *replace CDR*.

Now we explain the two primitives. Both RPLACA and RPLACD take two arguments. For RPLACA, first argument is a non-empty *list* say bound to a symbol, say, X and second is an arbitrary LISP *object* say bound to a symbol, say, Y. Then (*RPLACA X Y) replaces (Car X) by Y in the given list and the resulting list is still bound to X*. For RPLACD, the first argument is again a non-empty list bound to the symbol, say, X and the second argument *also must be a list*, say, bound to Y then (RPLACD X Y) replaces (CDR X) by Y and the resulting list is still bound to X.
Examples:

→ (SETQ X) '((a b) c d) ; returns
((a b) c d)
→ (SETQ Y 3) ; returns
3
→ (REPLACA X Y) ; returns
(3 c d); Further if we give

→ (SETQ Z ' (3 7 9)) ; returns
(3 7 9)
→ (SETQ U '(c e f) g)) ; returns
((c e f))
→ (REPLACA Z U); returns;
(((c e f) g) 7 9); this list is bound to Z

→ (SETQ V'((a b) c)) ; returns
((a b) c)

→ (REPLACD Z V) ; returns
(((e e f) g) ((a b) c))

The above two primitives viz RPLACA and RPLACD can be obtained from SETF as follows:
(RPLACA L S) is same as (SETF (CAR L) S)
and (RPLACD L S) is same as (SETF (CDR L) S)

In general we can make changes to lists in arbitrary positions in stead of just to the (CAR L) and (CDR L), as follows:

(RPLACA) (Cxxxxr L) S) or (SETF (Caxxxxr L)) S)
and (RPLACD (Cxxxxr L) S) or (SETF (Cdxxxxr L)) S)
where xxxx is a sequence made out of A's and D's.

Example :
→ (SETQ X ' ((a b) (c (d e) f))) ; returns
((a b) (c (d e) f))

→ (RPLACA (Cdadr X) 'p)
((a b) (c p f)) ; still bound to X

→ (SETQ Z '((a b) (c d e)))
((a b) (c d e))

→ (REPLACD (Cdadr Z) '(F G)) ; returns
((a b) (c d F G))

Ex 8: Find
(i) (RPLACA (Cdadr '((u v w) (s (t u) m))) '(a b c))
(ii) (RPLACD (Cdadr '((u v w (s (t u) m))) ' (a b c))

1.17 ARRAYS, STRINGS AND STRUCTURES

I. ARRAYS: LISP provides the primitive MAKE-ARRAY to create array structures. To create an array structure of *dimensionality n* and *dimensions* <*dim–1*>, …, <*dim–n*> the following syntax using MAKE-ARRAY is used:

(MAKE–ARRAY '(<dim–1> <dim–2> … <dim–n>))

For example to create an array structure named *Matrix–3* with dimensionality 3 and <dim–1>as 2, <dim–2> as 2 and <dim–3> as 3 of integers, the following LISP statement is used:

→ (SETQ Matrix–3 (MAKE–ARRAY '(2 2 3) & KEY : integer))

; returns the name Matrix–3

The above statement creates an array Matrix–3 of 12 elements. The slots in Matrix–3 are empty and we will discuss how to fill values in the slots. The 12 slots in Matrix-3 are referred to as

Matrix–3 (0, 0, 0), Matrix–3 (0, 0, 1) Matrix–3 (0, 0, 2)
Matrix–3 (0, 1, 0), Matrix–3 (0, 1, 1), Matrix–3 (0, 1, 2)
Matrix–3 (1, 0, 0), Matrix–3 (1, 0, 1), Matrix–3 (1, 0, 2)
Matrix–3 (1, 1, 0), Matrix–3 (1, 1, 1), Matrix–3 (1, 1, 2)

The primitive AREF is used to refer to a particular slot in the array, e.g., .
(AREF Matrix–3 1 0 2) refers to the slot Matrix–3 (1, 0, 2)

In order to assign a value to Matrix–3 (i, j, k), the following statement is used:

→ (SETF (AREF Matrix–3 1 0 2) 10); assigns value 10 to the slot (1 , 0 , 2) of Matrix–3

The above value-assigning statement can be easily used to give value say integer g to (i, j, k)th element of Matrix–3 as

→ (SETF (AREF Matrix–3 i j k) g)
Further, it can be generalised for any array in stead of Matrix–3.

In order to retrieve values from any slot, say (i , j , k) of Matrix–3 we use the following LISP statement:

→ (AREF Matrix–3 i j k)
; the value g is returned
; if g is the value stored at
; Matrix–3 (i, j, k) then g is returned

Next, we consider defining of strings in LISP:
II. A string is a one-dimensional array, whose elements are *characters*. MAKE-ARRAY or MAKE-STRING, a new primitive may be used for the purpose, e.g.,

→ (SETQ A (MAKE-ARRAY '(4) & KEY : character))
A ; creates a string-structure A capable of storing
; characters
or equivalently we can write the statement
→ (SETQ A (MAKE-STRING 4))

In order to store say 'WORD' in the string structure A

we can use either

(SETF (AREF A 0) #\W)
(SETF (AREF A 1) #\0)
(SETF (AREF A 2) #\ R)
(SETF (AREF A 3) # \ D)

or

(SETF (AREF A "WORD")

Note that the two-character sequence viz. # \ is used preceding a character to indicate that the following is to be interpreted as character.

III. Structure: Next, we describe commands in LISP for

- defining Pascal's record-like structures in LISP environment,
- assigning values to components of such a structure and
- retrieving values from the components of such a structure.

The primitive DEFSTRUCT is used to define structures. The syntax to **create a structure** (without values assigned to components) is

(DEFSTRUCT < structure-name > < slot-1 > … < slot-k>).

The structure created by the above type of LISP statement will be named < structure-name > and will have <slot-i>'s as slots. We may recall that '<*entity*>' enclosed between angular brackets indicate place-holder for *entity* to be suitable *replaced*. The above type of LISP statement automatically generates the keyword constructor called MAKE – <structure-name> and also automatically creates the selector functions as <structure–name> – <slot–i> for each i.

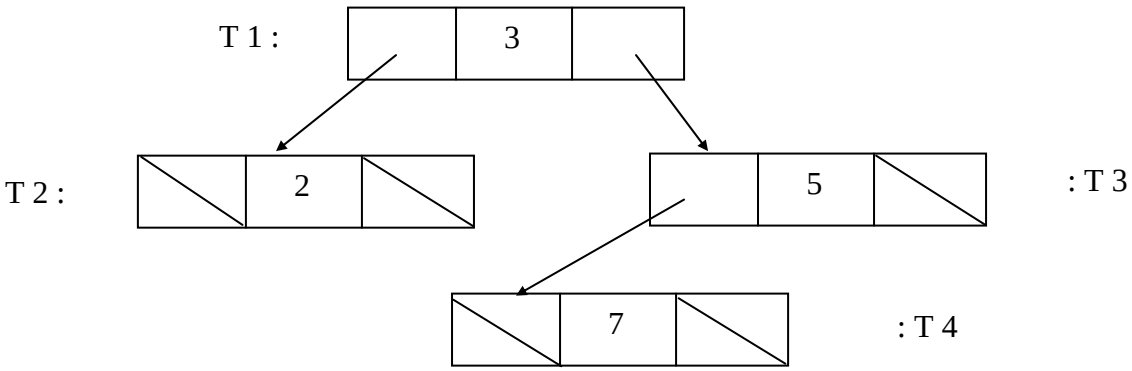
We explain the somewhat terse description above through an example of defining a binary-tree structure. For this purpose, let us recall the definition of a **binary tree**: A binary tree, is either empty or it consists of a node called the root together with two binary trees called the *left subtree* and the *right subtree* of the root.

For this purpose, a node of binary tree will have three slots: left-tree, value, right-tree. The left-tree and right-tree are pointers.

Using DEFSTRUCT, we create the structure, which we name as *bin-tree* through LISP statement:
(DEFSTRUCT bin-tree left-tree value right-tree)
The in-built mechanisms automatically create the following:

i) Constructor function *MAKE-bin-tree*
ii) Selector function *bin-tree-left-tree*
iii) Selector function *bin-tree-value* and
iv) Selector function *bin-tree-right-tree.*

Using the above description let us create the following binary tree to be called T1. The children and grand-children nodes are named as T2, T3 and T4. And a diagonally crossed cell indicates nil pointer.



The following sequence of LISP statements create the tree shown above:

→ (SETQ T4 (MAKE-bin-tree: left-tree nil : value 7 :right-tree nil))
;# S (nil 7 nil) is the value returned
; Note we can state : left-tree, : value, : right-tree in any order, e.g.

→ (SETQ T3 (MAKE-bin tree : value 5 : left-tree T4 : right-tree nil))
;#S (#S (nil 7 nil) 5 nil) is the value returned.

→ (SETQ T2 (MAKE-bin-tree : right-tree nil : left-tree nil: value 2))
;#S (nil 2 nil) is the value returned

→ (SETQ T1 (MAKE-bin-tree: left-tree T2 : right-tree T3 : value 3))
; the value returned is the following
S (# S (nil 2 nil) 3 # S (# S (nil 7 nil) 5 nil))

In the above the symbol #S indicates the fact that the part following # is a structure. In order to access values of: left-tree, : right-tree or : value the selectors *bin-tree-left-tree, bin-tree-right-tree* and *bin-tree-value* respectively are used, for example

→ (bin-tree-value T1) ; returns
3
→ (bin-tree-left-tree T1) ; returns
#S (nil 2 nil)

→ (bin-tree-left-tree T1); returns
#S (nil 2 nil)

→ (bin-tree-right-tree T1); returns
#S(#S (nil 7 nil) 5 nil)

Also if we may give the name of a structure then the whole of the structure would be available, e.g.

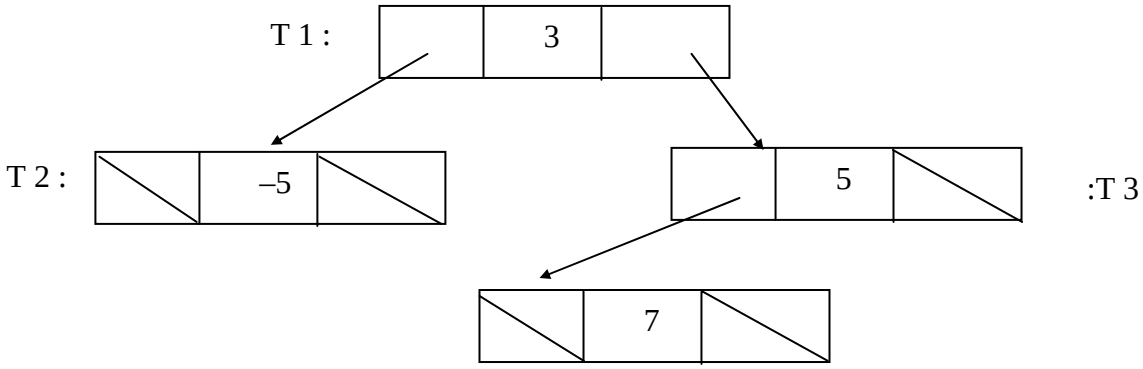
→ T3; the following value is returned
#S (#S (nil 7 nil) 5 nil)

→ T1; the following value is returned
S (#S (nil 2 nil) 3 # S(# S(nil 7 nil) 5 nil))

Further, in order to change or even create value of any component, we use the primitive SETF. For example, if we wish to change : *value* component of T2 to – 5 we can use

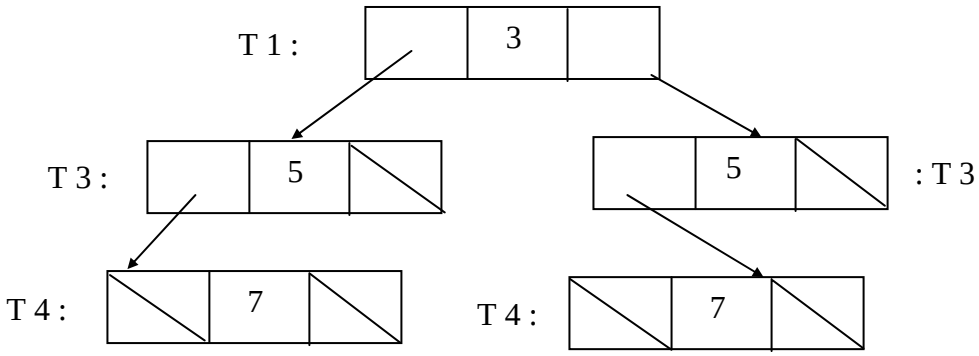
→ (SETF (bin-tree-value T2) – 5); returns

;the following tree as



If further, we want T1 to be more symmetrical having T3 as both *: left-tree* as well as *: right-tree*, then we can use
→ (SETF (bin-tree-left-tree T1) T3); returns T3 as value
#S (#S (nil 7 nil) 5 nil)

The new shape of tree T1 after the sequence of two changes mentioned above is like:



Further if give the command
→ T1; the new structure returned is

#S (#S (#S (nil 7 nil) 5 nil) 3 #S (#S (nil 7 nil) 5 nil))

1.18 SUMMARY

The programming language LISP is based on the *functional paradigm* of solving problems using a computer system. The concepts of '*paradigm of solving problem*', '*functional paradigm*' and '*imperative paradigm*' are briefly discussed in Section 1.0, i.e., in the *Introduction* section. Elements of the syntax of the language LISP are introduced in Section 1.2. The issues relating to structuring of data and representing data in LISP are discussed in Section 1.3. How a LISP system evaluates a valid LISP object is explained in the next section. Section 1.5 explains how primitive LISP functions are evaluated. Primitive List manipulation functions are discussed in Section 1.6. The next section discusses built-in predicates of LISP. The various logical operators are discussed in Section 1.8. Some facilities to write complex programs in LISP are provided in the form of special forms which use the special words DEFUN,

COND, DO and LET etc. These special forms are explained in Sections 1.9 and 1.10. The input/output functions and facilities are discussed in Section 1.11. The concept/mechanism of *recursion* plays a very important role in *programming* in general, and *symbolic programming* in particular. Recursion in LISP is discussed in Section 1.12.

In order to define objects in terms of their attributes and attribute values, *association lists* and *property lists* are used in LISP. These concepts are discussed in Section 1.13. Some more general and robust facilities in LISP defining and applying functions in the form of **Lambda Expression, Apply, Funcall and Mapcar** are discussed in Section 1.14. The representation of *symbols* and associated/represented objects in LISP environment and representation of operations on such representations are discussed in the next three sections.

1.19 SOLUTIONS/ANSWERS

Ex 1: (i) The variable x is bound to 5 and the variable y is bound to 7. Further the value (5 + 5) ∗ (7 + 7) is evaluated to 140

(ii) The expression having a quote in the beginning itself is evaluated to
(– (+ (setq p 9) (setq s 3)) (p ∗ s)).
No binding of p to 9 and s to 3 takes place.

Ex 2: (i) The expression in the first round reduces to (+ 24 (– 1) 2) which reduces to 25
(ii) The expression in the first round reduces to (∗ 5 0 11) which reduces to zero (0).

Ex 3:
```
(Defun ( X  Y)
   ( cond  ( = Y  0)   'infinity)
   ( t     ( /   X   Y   ) ) )
```

Ex 4:
Stepwise explanation

1. x is associated with (a b) and (a b) is returned.
2. '(c d) evaluates to (c d)
3. through subst x is replace by list (s t) and we get a new list
(u v (s t))
4. through reverse, from the last list, we get ((s t) v u)
5. the append of three lists viz (a b), (c d) and ((s t (v u) returns the list
(a b c d (s t) v u)
6. Finally 7, the length as number of topmost elements, is returned.

Ex 5:
```
( defun expo ( i j )
  ( do
      ( answer i ( * i answer ) )
                ; initially answer is i and is
                ; multiplied in each iteration by i
       (power j ( – power 1 ) )
       (counter ( – j 1) (– counter 1 ) )
                ; initially power is j and in each iteration power is reduced by 1.
                ; counter is an auxiliary variable
                )
```

```
                ( ( zerop counter ) answer )
        )
)
```

→ (expo 2 3)
8

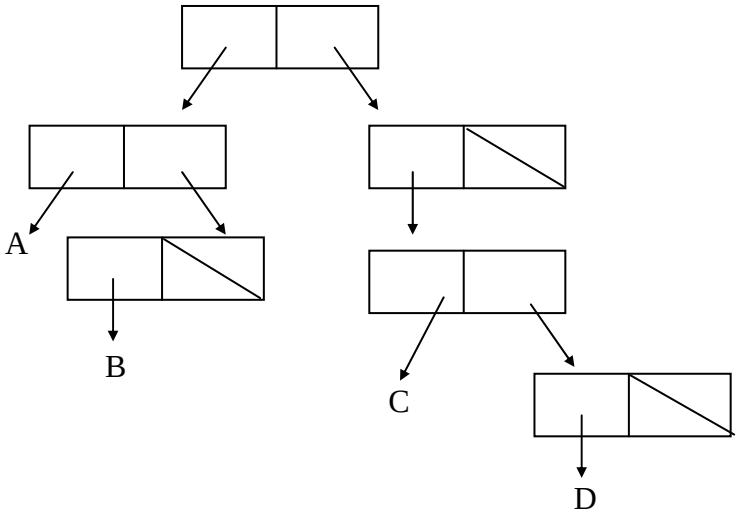
Remarks: The clause (power j (– power 1)) is actually not required. However, it is introduced to explain. It can be deleted without affecting the overall (final) result. But it has been introduced to explain an important point about do-loop. We *may be tempted to write the above function expo* by replacing the clause (counter (– j 1) (– counter 1)) by (counter (– j 1) (– power 1)) i.e. replacing last occurrence of counter by power; because it is also being computed in the earlier clause. But this *replacement will be wrong*, leading to incorrect result because of the fact that in Do loop all the variables, viz *answer, power, and counter* in the above example are computed *in **parallel***, using values from the previous iteration/ initialization. *Current* values are available only in the same clause. Therefore, if 'power' replaces 'counter' then *previous* value of power would be available for processing whereas we require the current value.

*In many situations, we need **sequential** computation of the variables in the loop. For this purpose LISP Provides do*. Now, the function of Example 2 above may be rewritten using the earlier computed values of power as is given below:*

```
( defun expo ( i j )
   ( do*   (
                ( answer  i       ( * i answer ) )
                ( power   j       (– power 1 ) )
                (counter          (– power 1 )
                                  (– power 1 )
                )
              )
                ( ( zerop counter ) answer )
    )
)
```

Ex 6: (defun deep-length (L)
```
        ( cond
                (( null L ) 0)
                (( list p ( car L )) ( + deep-length (car L )) (deep-length ( cdr  L ))
                ( t ( + 1 ( cdr L )))
        )
     )
```

Ex 7: The list ((A B) (C D)) is represented by the cons cell structure



Ex 8:
(i) ((u v w) (s (a b c) m))
(ii) ((u v w) (s (t u) a b c))

1.20 FURTHER READINGS

1. Graham P. : *ANSI Common Lisp* Prentice Hall (1996).
2. Sangal R. : *Prgramming Paradigms in LISP* McGraw-Hill, Inc. (1990).
3. Patterson D.W.: ***Chapter 3 of*** *Introduction to Artificial Intelligence and Expert Systems*; Prentice-Hall of India (2001).