1.47

DATATYPE PREDICATES

Since data types need not be declared, Lisp provides predicates for dynamically querying the system.

atom		
numberp	----------	floatp
symbolp		integerp
listp		ratiop
		zerop
		plusp
		minusp
		evenp
		oddp
		>
		<

(< datatype > x) returns T if x is of that type, else NIL.

1.48

Numeric Functions

Basic Numeric Datatypes

-- integer	**1237**
-- ratio	**5/8**
-- floating point	**3.14159**

(+ argl …. argn)	**argl + ….. + argn**
(* argl ….. argn)	**argl * ….. * argn**
(/ argl arg2)	**argl / arg2**
(- argl arg2)	**argl - arg2**
(expt argl arg2)	**argl $^{arg\ 2}$**

(max argl ….. argn)
(min argl ….. argn)
(abs argl)
(sqrt argl)
(- argl)
(1+ argl)
(1- argl)

(< argl arg2)	**(numberp argl)**
(> argl arg2)	**(floatp argl)**
(zerop argl)	**(null argl)**

1.49

Setf

Setf provides a facility for explicit binding (assignment) in Lisp.

```
(setf   ID1   VAL1
        ID2   VAL2
        IDn   VALn )
```

quotes the odd - numbered arugments (the IDs) and assigns the evaluation of each even - numbered argument to the preceding odd - numbered one.

(setf	**x 'A**	**x ← A**
	y '(A B C))	**y ← (A B C)**
(setf	**EL (CAR '(A B)))**	**EL ← A**

- ★ **Since setf performs global assignment, we will generally use it**
- ★ **for initializations at the top level of the Lisp interpreter.**

1.50

More About Function Definitions

A Common Lisp function may have more than one form in its body.

The value returned by the function is the value of the LAST form.

The other forms produce side effects.

```
( defun  function_with_side_effects  ( L )
        ( setf   begin   ( first   L ))
        ( setf   end     ( car  ( last   L ) ) )
        ( list   begin   end ) )
```

L ((8) 71 9 65 (2))
begin: 8, end: 2

result (8 2)

1.51

Let and Let*

The let and let* forms implement local variables.

```
( let      ( ( parm1          init 1 )
             ( parm 2         init 2 )
                  :
             ( parm m         init m ) )
   form1    …..   form n )
```

initializes variables parm1, parm2, …. , parmm to the values init1, init2, …. , initm and these variables are bound to the values in the scope of the let, which includes form1 …. formn .

```
( let  ( ( x   10 ) )    ( print   ( 1+   x ) ) )      11

( setf    x    5 )
( let  ( ( x  10 )   ( y   ( *  x  x ) ) )             25
( print   y ) )

( let*  ( ( x  10 )  ( y   ( *  x  x ) ) )
( print  y ) )                                          100
```

1.52

Using LET in Functions

```
( defun    function_with_let   ( L )
        ( let     ( ( begin    ( first   L ) )
                    ( end      ( car   ( last    L ) ) ) ) )
        ( list     begin     end  ) ) )

( defun    roots    ( a  b  c )
    ( let  ( (disc  (sqrt (- (* b b ) (* 4 a c ) ) ) ) )
          ( denom   ( *  2  a ) )
          ( minusb   ( -  b ) ) )

    ( list  ( /     ( +   minusb  disc )   denom )
           ( /     ( -   minusb   disc )    denom ) ) ) )
```

Functions as Arguments to Functions

-- Sometimes you want to pass the name of a function to a second function and have the second function use the first.

-- The 'functions applied to functions' are called **functionals**.

-- **mapcar** is probably the most commonly used functional in Lisp.

(mapcar (function < name >) < list >)

or

(mapcar # '< name > < list >)

applies the function of one argument to each element of the list, returning a newly created list.

(mapcar #'1+ '(2 4 6 8 10))

returns (3 5 7 9 11)

For mapping functions of 2 (or more) arguments, mapcar requires 2 (or more) lists.

(mapcar # '+ '(1 7 3 9) '(8 6 2 3))
returns (9 13 5 12)

More examples

(defun twice (n) (+ n n))

(defun double (L) (mapcar #'twice L))

(assoc < key > < association list >)

searches for and returns the first (2 element) sublist whose car matches the key.

(setf words '((one un) (two deux) (three trois)))

(defun trans-word (w) (cadr (assoc w words)))

(defun trans-list (L) (mapcar #'trans-word L))

(trans-list '(two three one two))

1.55

LAMBDA EXPRESSIONS

In the lambda calculus, you define functions using expressions such as

$\lambda x \quad f(x)$

Lisp borrowed the λ, changed it to ‘lambda’ for utility functions that don’t have names.

(lambda (<args>) <body>)

is the definitions of a nameless function to be used by another function

((lambda (x) (+ 2 x)) 3)
returns 5

((lambda (x y) (+ x y)) 2 3)
returns 5

(mapcar #’(lambda (x) (+ x x)) ’(4 1 3))
returns (8 2 6)

1.56

EVAL, APPLY, AND FUNCALL

are forms for evaluating Lisp expressions.

(eval < unevaluated expression >)
evaluates the expression and
returns the result

(eval '(+ 2 3)) returns 5

(apply < function name > < arg list >)
applies the function to the list of args.

(apply #'+ '(2 3)) returns 5
(apply (function (lambda (x) (* x x)) '(2))
returns 4

(funcall < function name > < separate args >)
does the same

(funcall #'+ 2 3) returns 5

1.57

It's easy to use EVAL to write an interpreter for some other language.

Just convert expressions to Lisp forms and EVAL them!

1.58

OPTIONAL ARGUMENTS

The Keyword &optional means all following parameters are optional.

```
( defun  mplus  ( x  y  &optional  ( m 1 ) )
         ( *   m   ( +   x   y ) )  )
```

```
( mplus   2   3 )             5
( mplus   2   3   6 )         30
```

ARBITRARY NUMBERS OF ARGUMENTS

The Keyword &rest says the parameter following it will name a list that an arbitrary number of corresponding arguments will be put into.

```
( defun   mp2   ( v   &rest   vals )
   ( mapcar   #'( lambda  (x)  (+  v x ) )   vals) )
```

```
( mp2    6  9  0  1  3 )       returns  ( 15  6  7  9 )
```

1.59

PROPERTY LISTS

A PLIST (property list) is associated with each Lisp symbol.

-- (get < symbol > < property name >)
retrieves values for that property.

-- (setf (get < symbol > < property name >) < value >)
associates the value with the property.

(setf (get ’ bob ’ height) 60)

(get ’ bob ’ height)

-- (remprop < symbol > < property name >)
removes this property from this symbol

(remprop ’ bob ’ height)

ARRAYS

1.60

-- (make - array < list of dimensions >

[**:**initial-contents < list >])

constructs a new array

(setq a (make-array ’(2 2)

:initial-contents ’((1 2) (3 4))))

produces:

	0	**1**
0	**1**	**2**
1	**3**	**4**

-- (aref < name > < subscripts >)
retrieves an element

-- (setf (aref < name > < subscripts >) val)
stores a value in an element

(setf (aref a 1 1) 27)

(aref a 0 0) returns 1

SOME Lisp Programming Techniques

1.61

1. “currying” -- use lambda expressions to create functions with fewer arguments

```
( defun incr ( x L )
      ( mapcar # ’(lambda ( y ) (+ x y ) ) L ) )

( incr 2 ’( 4 1 3 ) )   returns ( 6 3 5 )
```

2. Use of auxiliary functions to make programs more readable and sometimes more efficient.

3. “tail recursion” -- recursion where all the work is done on the way down, returns just pass control back up.

```
( defun tr-fact ( n ) ( f-aux n 1 ) )
( defun f-aux ( n res )
      ( if ( zerop n ) res
            ( f-aux ( 1- n ) ( * n res ) ) ) ) )
```

1.62

DESTRUCTIVE OPERATIONS

(rplaca < name > val)

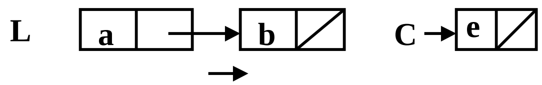
replaces the car field by val

(rplacd < name > val)

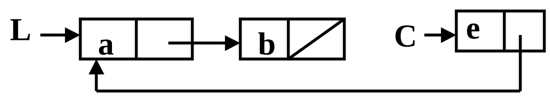
replaces the cdr field by val

(setf L (cons ’a (cons ’b NIL)))

(setf C (cons ’e NIL))



(rplacd C L)



Tower of Hanoi Program Example

```
(defun TowerOfHanoi   nil
            (print  '(Enter number of disks)  )
            (setf disks  (read)  )
            (Transfer  '1  '3 '2  disks)  )


(defun MoveDisk  (Frompin Topin)
            (print  (list Frompin  '--> Topin) )
            (terpri) )


(defun Transfer  (Frompin Topin Usingpin Height)
        (cond
              ( (equal Height 1)  (MoveDisk Frompin Topin ) )
              (  T  (Transfer Frompin Usingpin Topin (1- Height) )
                    (MoveDisk Frompin Topin )
                    (Transfer Usingpin Topin Frompin (1- Height) )
              )
        )
)
```