DATATYPE PREDICATES

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

atom
numberp      floatp
symbolp      integerp
listp
zerop
plusp
minusp
evenp
oddp

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

Numeric Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>argl + ..... + argn</td>
</tr>
<tr>
<td>*</td>
<td>argl * ..... * argn</td>
</tr>
<tr>
<td>/</td>
<td>argl / arg2</td>
</tr>
<tr>
<td>-</td>
<td>argl - arg2</td>
</tr>
<tr>
<td>expt</td>
<td>argl^arg2</td>
</tr>
<tr>
<td>max</td>
<td>argl ..... argn</td>
</tr>
<tr>
<td>min</td>
<td>argl ..... argn</td>
</tr>
<tr>
<td>abs</td>
<td>argl</td>
</tr>
<tr>
<td>sqrt</td>
<td>argl</td>
</tr>
<tr>
<td>-</td>
<td>argl</td>
</tr>
<tr>
<td>1+</td>
<td>argl</td>
</tr>
<tr>
<td>1-</td>
<td>argl</td>
</tr>
<tr>
<td>&lt;</td>
<td>argl arg2</td>
</tr>
<tr>
<td>&gt;</td>
<td>argl arg2</td>
</tr>
<tr>
<td>zerop</td>
<td>argl</td>
</tr>
<tr>
<td>nilp</td>
<td>null argl</td>
</tr>
</tbody>
</table>
Setf provides a facility for explicit binding (assignment) in Lisp. 

\[(\text{setf } \text{ID1 VAL1} \text{ ID2 VAL2} \text{ IDn VALn})\]

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

\[(\text{setf } x \ 'A \ y \ '(A \ B \ C) \ x \leftarrow \ A \ y \leftarrow \ (A \ B \ C))\]

\[(\text{setf } \text{EL (CAR '(A B))}) \ \text{EL} \leftarrow \ A\]

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

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.

\[(\text{defun } \text{function\_with\_side\_effects}( L ))\]

\%(setf \begin{array}{l}
\text{begin} \ (\text{first} \ L)) \\
\text{end} \ (\text{car} \ (\text{last} \ L))
\end{array})\]

\%(list \begin{array}{l}
\text{begin} \ 9 \\
\text{end} \ 2
\end{array})\]

result \( (8 \ 2) \)
**Let and Let**

The let and let* forms implement local variables.

\[
\text{(let ( ( parm1 init1 )
        ( parm2 init2 )
        \ldots
        ( parm m init m )
    )
    form1 \ldots form n )}
\]

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

\[
\text{(let ((x 10)) (print (1+ x))) 11}
\]

\[
\text{(setf x 5)}
\]

\[
\text{(let ((x 10) (y (* x x))) 25}
\]

\[
\text{(print y)}
\]

\[
\text{(let* ((x 10) (y (* x x))) 100}
\]

\[
\text{(print y)}
\]

**Using LET in Functions**

\[
\text{(defun function_with_let ( L )}
\]

\[
\text{(let ( ( begin ( first L )}
        ( end ( car ( last L ) ) )
    )
    ( list begin end ) )}
\]

\[
\text{(defun roots ( a b c )}
\]

\[
\text{(let ((disc (sqrt (- (* b b ) (* 4 a c ) ) ) )}
        ( denom (* 2 a ) )
        ( minusb (- b ) )
    )
    ( list (/ (+ minusb disc ) denom )
        (/ (- minusb disc ) denom ) ) )
\]
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 ) )
LAMBDA EXPRESSIONS

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

\( \lambda x \ f(x) \)

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

\[
( \text{lambda} \ ( <\text{args}> \ ) \ <\text{body}> \ )
\]

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

\[
( \text{lambda} ( x ) \ ( + \ 2 \ x ) )
\]

returns 5

\[
( \text{lambda} ( x \ y ) \ ( + \ x \ y ) )
\]

returns 5

\[
( \text{mapcar} \ #'( \text{lambda} ( x ) \ ( + \ x \ x ) ) \ '(4 \ 1 \ 3 ))
\]

returns \( (8 \ 2 \ 6) \)

1.55

EVAL, APPLY, AND FUNCALL

are forms for evaluating Lisp expressions.

\[
( \text{eval} \ <\text{unevaluated expression} > )
\]
evaluates the expression and returns the result

\[
( \text{eval} \ '(+ \ 2 \ 3) )
\]
returns 5

\[
( \text{apply} \ <\text{function name} > \ <\text{arg list} > )
\]
applies the function to the list of args.

\[
( \text{apply} \ #'+ \ '(2 \ 3))
\]
returns 5

\[
( \text{apply} \ (\text{function} \ (\text{lambda} \ (x) \ (* \ x \ x)) \ '2))
\]
returns 4

\[
( \text{funcall} \ <\text{function name} > \ <\text{separate args} > )
\]
does the same

\[
( \text{funcall} \ #'+ \ 2 \ 3 )
\]
returns 5

1.56
It’s easy to use EVAL to write an interpreter for some other language.

Just convert expressions to Lisp forms and EVAL them!
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

-- (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:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

```lisp
(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.

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

### DESTRUCTIVE OPERATIONS

- `rplaca <name> <val>`
  replaces the car field by `val`

- `rplacd <name> <val>`
  replaces the cdr field by `val`

- `setf ( cons `a ( cons `b NIL ) )`
- `setf ( cons `e NIL )`

![Diagram showing destructive operations](image-url)

- `rplacd C L`

![Diagram showing destructive operations](image-url)
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) )
    )
  )
)