LISP

Everything in a computer is a string of binary digits, ones and zeros, which everyone calls bits. From one perspective, sequences of bits can be interpreted as a code for ordinary decimal digits, but from another perspective, sequences of bits can be interpreted as a code for real world like objects and sentences like objects.

- In LISP , the fundamental things formed from bits are world like objects called atoms.
- Groups of atoms form sentence like objects called lists. Lists themselves can be grouped together to form higher-level lists.
- Atoms and lists collectively are called symbolic expressions, or expressions. Working with symbolic expressions is what symbol manipulation using LISP about.

Basic LISP Primitives.

In LISP, the prompt is an asterisk: ***.** The words beyond the semicolon are comments inserted for explanation-a semicolon makes the remainder of the words on a line invisible to LISP.

Suppose for example, that we want some help adding numbers. The proper way of doing is, once LISP displayed the prompt**,** is as follows

```
*(+3.14 2.71)
```

LISP responds as follows:

```
5.85
```

This is a simple example of LISP's ability to handle arithmetic. When left and right parentheses surround something, we call the result a list and speak of its elements. In our first example, the list (+3.14 2.71) has three elements, +, 3.14, and 2.71, separated by spaces. + is standing before the two things to be added, rather than between them, as in ordinary arithmetic notation . In LISP the name of the thing to do, the procedure, is always specified first, followed then by the things that the procedure is to work with, the arguments. FIRST and REST take lists apart. For example

```
*(first ‘(fast computers are nice))
FAST
*(rest ‘(fast computers are nice))
COMPUTERS ARE NICE
```

Also, when REST is applied to a list with only one element, it returns the empty list, which we can denote by either () or NIL.

```
*(rest ‘(nice))
NIL
```

To assign values to symbols is to use SETF. This primitive causes the value of its second argument to be assigned to its first argument. Consider this:

```
*(setf ab-list ‘(a b))
```

When we type :

```
*ab-list
(A B)
```

While FIRST and REST take things apart, CONS, APPEND, and LIST. CONS takes expression and a list and returns a new list whose first element is the expression and whose remaining elements are those of the old list:

```
*(cons ‘a’(b c))
```

This will display : A B C . The template for cons is (cons <new first element> <a list>). APPEND combines the elements of all lists supplied as arguments. For example:

```
*(setf ab-list ‘(a b)
        xy-list ‘(x y))
*(append ab-list xy-list)
(A B X Y)
```

A LIST makes a list out of its arguments. Each argument becomes an element of the new list:

```
*(setf front ‘a middle ‘b back’c)
*(list front middle back)
(A B C)
```

NTHCDR trims off n elements, not just one, with the first argument determining the exact number:

```
*(setf abc-list ‘(a b d))
*(nthcdr 2 abc-list)
(D)
```

BUTLAST is similar to NTHCDR, but trims off the last n elements, rather than the first.

```
*(setf abc-list ‘(a b d))
```

```
*(butlast abc-list 2)
(A)
```

LAST returns a list with all but the last element of a list trimmed off:

```
*(setf abc-list ‘(a b d) ab-cd-list ‘((a b) (d e)))
*(last abc-list)
(D)
```

The LENGTH primitive counts the number of top-level elements in a list. REVERSE reverses the order of the top-level elements of a list.

```
*(setf ab-list ‘(a b) )
*(length ab-list)
2
*(reverse ab-list)
(B A)
```

The ASSOC primitive is especially tailored to work with a particular kind of association-recording expression called an association list. The first element of each sublist is used as a key for recovering the entire sublist.

```
*(setf sarah ‘((height .54) (weight 4.4)))
*(assoc ‘weight sarah)
(WEIGHT 4.4)
```

LISP also has other very important mathematical functions. Round function rounds the number to the nearest integer.

```
*(round (/22 7))
3                  :The nearest integer
1/7                : The remainder
```

MAX and MIN find maxima and minima. EXPT calculates powers; it raises its first argument to its second. SQRT takes the square root, and finally ABS computes the absolute value.

```
*(MAX 2 4 3)
4
*(MIN 2 4 3)
2
*(expt 2 3)
8
*(sqrt 9)
3
*(abs -5)
5
```

Procedure Definition and Binding

DEFUN is LISP's Procedure-Definition Primitive For example:

```
*(setf whole-list ‘(breakfast lunch tea dinner))

(defun both-ends

   (whole -list)

   (cons (first whole-list)

           (last whole-list)
```

LET forms bind parameters to initial value.

```
*(setf whole-list ‘(breakfast lunch tea dinner))

*(let ((element (first whole-list))              ‘Bind ELEMENT to initial value.

      (trailer (last whole-list))                ‘Bind TRAILER to initial value.

      (cons element trailer))                    ‘Combine ELEMENT and TRAILER.
```

This will display BREAKFAST DINNER.

Predicates and Conditionals

A predicate is a procedure that returns true or false. For example :

```
*(equal (+2  2) 4)

T

*(equal(+2 3) 3)

NIL
```

Several predicates determine if their arguments are equal. They range from EQUAL, which is a general purpose predicate, but computationally expensive, to EQL, EQ and =.

MEMBER Tests for List Membership. The MEMBER predicate tests to see if its first argument is an element of its second argument. MEMBER returns what is left of the list when the matching symbol is encountered.

```
*(setf sentence ‘(tell me more about your mother please))
*(member ‘mother sentence)
(MOTHER PLEASE)
```

LISP has many predicates that test objects to see whether they belong to a particular data type.

Name	**Purpose**
atom	Is it an atom?
numberp	Is it a number?
symbolp	Is it a symbol?
listp	Is it a list?
null	Is the argument an empty list?
endp	Is the argument, which must be a list, an empty list?
zerop	Is it zero?
plusp	Is it positive?
minusp	Is it negative?

evenp	Is it even?
oddp	Is it odd?
>	Are they in descending order?
<	Are they in ascending order?

Predicates help IF, WHEN, and UNLESS Choose among Alternatives.

```
*(if (sumbolp day-or-date) ‘day ‘date)
```

The result is the symbol DAY if the value of day-or-date is a symbol, and DATE otherwise. A WHEN can be used instead of an IF whenever the false consequent in an equivalent IF form would be NIL. An UNLESS can be used instead of an IF whenever the then form in an equivalent IF form would be NIL. Predicates also help COND choose among alternatives. COND is particularly useful when there are more than two cases to consider. In the following example, a probability number between 0 and 1 is converted into one of four symbols:

```
*(setf p .6)
*(cond ((>p .75) ‘very-likely)
        ((>p .5) ‘likely)
        ((>p .25) ‘unlikely)
        (t ‘very-unlikely))
LIKELY
```

CASE checks the evaluated key form against the unevaluated keys using EQL. If the key is found, the corresponding clause is triggered and all of the clause’s corresponding clause is triggered and all of the clause’s consequents are evaluated.

```
*(setf thing ‘ball r1)
*(case thing
    ((circle wheel) (*pi r r))
    ((sphere ball) (*4 pi r r))
    (Otherwise 0))
12.56637
```

Procedure Abstraction and Recursion

Procedure Abstraction Hides Details Behind Abstraction Boundaries. For Example:

```
*(setf l ‘(breakfast lunch tea dinner))
*(both-ends l)
(BREAKFAST DINNER)
```

To do its job, both-ends must do three things:

- BOTH-ENDS must extract the first element.
- It must extract the last element.
- It must combine the two extracted elements.

Recursion allows procedures to use themselves. To calculate the *n*th power of some number, m,it is sufficient to do the job for the (n-1)th power, because this result, multiplied by m, is the desired result for the *n*th power:

```
(defun higher-level-expt (m n)
     (if (zerop n)
    1                           ‘If n=0, return 1.
    (* m                        ‘Otherwise, use a higher-level abstraction
        (higher-level-expt m(-n 1)))))
```

Data Abstraction

We can hide the details of how our data is represented by isolating that data behind a set of constructor procedures that create data, reader procedures that retrieve data, and writer procedures that change data. For example BOOK-AUTHOR is the reader procedure for retrieving a book’s author. When we start out, BOOK-AUTHOR is defined as follows

```
(defun book-author (book)     ;Data is in a simple list.
     (second book))
```

And here is an example of BOOK-AUTHOR in use

```
(setf book-example1
   ‘((Artificial Intelligence)        ;Title
     (Patrick Henry Winston)          ;Author
     (Technical AI)))                 ;Index terms.
*(book-author book-example-1)
(PATRICK HENRY WINSTON)
```

And this is how Constructor works

```
(defun make-book (title author classification)
   (list (list ‘title title)
        (list ‘author author)
        (list ‘classification classification)))
```

The constructor creates a new book, given appropriate arguments.

```
* (setf book-example-4
        (make-book ‘(Common Lisp)
                   ‘(Guy Steele)
                   ‘(Technical Lisp)))
(TITLE (COMMON LISP))
(AUTHOR (GUY STEELE))
(CLASSIFICATION (TECHNICAL LISP)))
```

Loops

The DOTIMES primitive is popular because it provides a way to write simple counting-oriented iteration procedures.

```
(defun dotimes-expt (m n)
  (let ((result 1))                    ;Initialize result parameter.
    (dotimes (count n result)          ;Evaluate body n times
      (setf result (*m result)))))     ;Multiply by m again
```

The DOLIST primitive is similar to DOTIMES except that the elements of a list are assigned, one after another, to the parameter. This is the general template.

```
(dolist (<element parameter> <list form> <result form>)
      <body>)
```

The DO primitive can be used for iteration when DOLIST and DOTIMES are just not flexible enough. Otherwise, we should avoid DO because it is harder to use.

```
(defun do-expt (m n)
   (do ((result 1)                              ;Bind parameters
        (exponent n))                           ;Bind parameters
        ((zerop exponent) result)               ;Test and return
        (setf result (* m result))              ;Body
         (setf exponent (- exponent 1))))       ;Body
```

Printing and Reading

For example

```
*(setf temperature 100)
*(print temperature)
100
```

When the primitive READ is encountered, LISP stops and waits for us to type an expression.

FORMAT formats the line while it is printing. For example

```
*(format t "~%Hello!")
Hello!                  ;FORMAT prints Hello!
NIL
```

READ does not Evaluate Expressions, but EVAL evaluates twice

String manipulation is done by special primitives:

length	gives length of the string
reverse	reverses the string
elt	If the first argument in an elt form is an list, then elt returns the element specified by the second argument, where 0 specifies the first element.
string=	To determine if two strings are the same(case sensitive)
string-equal	To determine if two strings are the same (is not case sensitive)
char=	To determine if two characters are the same (case sensitive)
char=equal	To determine if two characters are the same (is not case sensitive)
search	To determine if one string is contained in another.
read-line, read-char	Useful for reading strings and characters from a terminal or file

Properties, Methods, Structures,Classes

Get and Setf are the primitives that we use when we want to retrieve information of a property, or assign to a value to the property. Methods are procedures selected from generic functions by argument types.

```
(defmethod area ((figure rectangle)
    (*(rectangle-width figure)
       (Rectangle-height figure)))          ;Method for rectangles
```

User defined structure types facilitate data abstraction. Here is how they are used:

```
(defstruct person
      (sex nil)                        ;Default value is NIL
      (personality 'nice))             ;Default value is NICE
```

Here is what is done by DEFSTRUCT forms:

- Creates a constructor procedure
- Creates reader procedures for getting things out of an instance's fields.
- Generalizes setf, to serve a writer when used in combination with the readers
- Creates a predicate for testing objects to see if they are instances of the DEFSTRUCT defined data type

Classes are usually arranged in hierarchies. When we can reach one class by walking up a series of connections, the lower class is called a subclass of the upper one, and the upper one is called a superclass of the lower one. The connections are called inheritance links because each class inherits slots from its superclasses.

Encapsulation

Lambda is used when mapping is done. Lambda defines anonymous procedures. Encapsulation enables the creation of sophisticated generators. The sophisticated way to create generators is to encapsulate free variables in lambda definitions by embedding the lambda definition in a parameter-containing form.

Reference: "LISP 3rd Edition" Patrick Henry Winston and Berthold Klaus Paul Horn