LISP Programming Tutorial Notes
CSC4510

Sau-Ming LAU

Department of Computer Science and Engineering
The Chinese University of Hong Kong

September 5, 1996

1 Introduction

- Like most traditional programming languages, LISP is procedural.
  - LISP programs describe HOW to perform an algorithm.
  - This contrasts with declarative languages like PROLOG, whose programs are DECLARATIVE assertions about a problem domain.
- LISP is a FUNCTIONAL language. Its syntax and semantics are derived from the mathematical theory of recursive functions.
- Everything in LISP is a function!

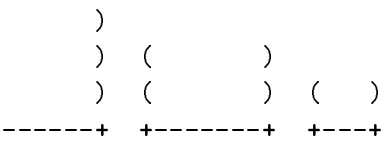
A function is a procedure that takes in one or more input arguments and returns only one value, with no side effect.

```
g(a, b) = a + b
```

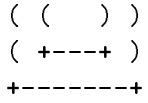
LISP is NOT a 'pure' functional language —
some of its functions produce side effects, e.g. SETF

1.1 The bowls analogy

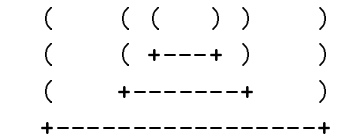
- We now talk about things called LISTS and ATOMS.
- The easiest way to understand lists and atoms is to think about an analogy with bowls, like those used to store things in a refrigerator.
- Often refrigerator bowls are made to fit inside one another so that they are easy to store. Here, for example, are three such bowls:



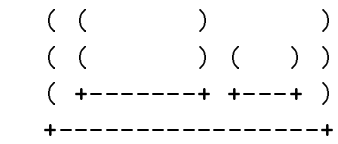
We can put the small bowl inside the medium one:



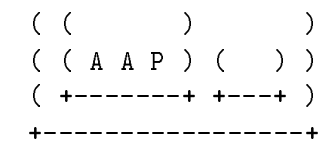
Then we can put the combination of the small and medium bowls inside the big bowl:



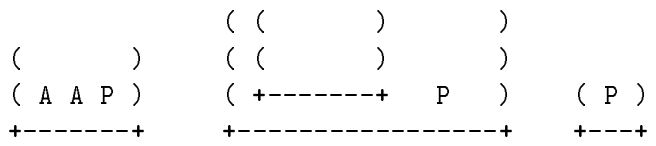
Here we can say that the small bowl is empty, the medium bowl contains the small bowl, and the big bowl contains the medium bowl.
Alternatively, we can put both the small bowl and the medium bowl inside the big bowl this way:



Here we say that both the small bowl and the medium bowl are empty and that the big bowl contains both the small bowl and the medium bowl.
Of course, we can also put some fruit in the bowls, represented here by As for the Apples and a P for the Pear:



The small bowl is still empty; the medium bowl now contains two apples and one pear; and the big bowl still contains the small and medium bowls. We could say that the big bowl also contains two apples and a pear, but let's agree to say that the big bowl contains a bowl that contains the two apples and one pear.
Thus CONTAINS means DIRECTLY CONTAINS, with no other container intervening.
Here is another example:



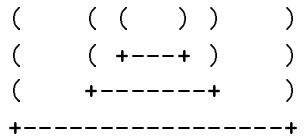
Now there are two medium bowls: one contains two apples and one pear, and the other is empty. The large bowl contains both a medium bowl and another pear. The small bowl contains a pear, too.

1.2 LISTS and ATOMS

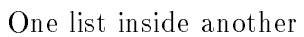
Now we are ready for the big leap to LISP:

- The basic data objects in LISP are LISTS and ATOMS.
- LISTS are like BOWLS. Like bowls, lists can be empty. Like bowls, lists can contain other lists. Like bowls, lists can contain things that are not containers.
- ATOMS are like the things people store in bowls.

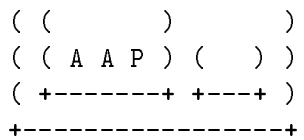
For example,
One bowl inside another inside another:



One list inside another inside another:

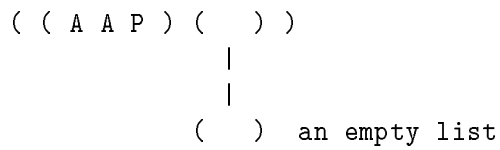


Here we can say that the inside list is empty; the empty list is inside another list; the combination is, in turn, inside still another list.
Here is a bowl set involving two apples and a pear:



And here are some lists:

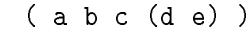
One list is empty, just as one bowl was. Another list contains three things, called atoms, just as one bowl contains three pieces of fruit. There is a list containing two lists, just as there is a bowl containing two bowls.
The second element of the following big list is an empty list:



1.3 Basic Data Types of LISP

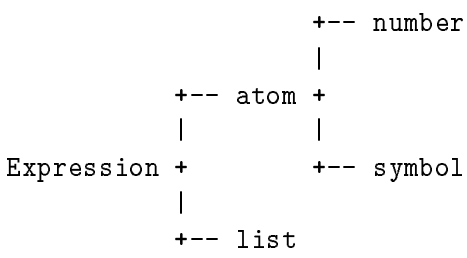
Every expression in LISP is either an atom or a list.

- Indivisible things like 27, +, FIRST, FOO, which have obvious meanings are called ATOMS.
- Atoms like 27, 3.14, etc. are called NUMERIC ATOMS.
- Atoms like FIRST, FOO, +, etc. are called SYMBOLIC ATOMS.
- A list consists of a left parenthesis, followed by zero or more elements, followed by a right parenthesis:

Elements of a list are separted by space. Lists themselves can be elements of another list.

- Both atoms and lists are called SYMBOLIC EXPRESSIONS.



1.4 Exercises:

```
1. ATOM                  atom
2. (THIS IS AN ATOM)     list
3. ((A B) (C D)) 3 (3)   neither!
4. (LIST 3)              list
5. (/ (+ 3 1) (- 3 1))   list
6. ()                    list (empty list or NIL)
7. ( () () )             list
                         (list containing 2 empty lists)
```

1.5 LISP as an Interpreter

- When LISP is resting, doing nothing, it displays a PROMPT symbol, normally an asterisk, to tell you that LISP is waiting for you to type something.

```
*   ;Indicates LISP is waiting for you to type.
```

- The words beyond the semicolon are COMMENTS.
- It is conventional to use the word FORM to refer to any symbol or list when it is obvious that the symbol or list has a value.
- The process of finding the value is called EVALUATION.

LISP can read procedure definitions and other information from the designated file:

```
* (load "c:\\lisp\\myprog.lsp")
```

To exit from LISP:

```
* (exit)
```

2 Introduction to LISP Primitives

Procedures that come with LISP are called PRIMITIVES. Thus the following are all LISP primitives.

```
Primitive name      Action
==============      ======
+                   Adds numbers.
-                   Subtracts numbers.
*                   Multiplies numbers.
/                   Divides numbers.
MAX                 Finds the largest number.
MIN                 Finds the smallest number.
```

For example,

```
* (+ 1 2 3)
6             ;LISP returns 6 because 1 + 2 + 3 = 6
```

Note that the plus sign comes before the numbers, not in between — prefix notation.

- In LISP, the symbol that names the thing to do is called the PROCEDURE NAME.
- Whenever a list is used to specify computation, a procedure name is always the first element of the list.
- The rest of the elements in the list are the things to work on. They are called ARGUMENTS.

In (+ 1 2 3) the PROCEDURE NAME is +.
The ARGUMENTS are 1, 2, and 3.

2.1 Some More Examples

```
* (MAX 3 6 9 1 6 9 5)
9
* (MIN 3 6 9 1 6 9 5)
1
* (+ (- 5 3) (/ 12 6))
4.0
* (* (MAX 3 4 5) (MIN 3 4 5))
15
*
```

Assume that we have define PI to be 3.14 and suppose we have a circle with a radius of 3.
Then its circumference is 2 x PI x 3. To get the result using LISP, type this:

```
* (* 2 PI 3)
18.84               ;2 x 3.14 x 3 = 18.84.
*
```

2.2 SETF Primitive

- Assign a value to a symbol (SET Field).
- Returns the 2nd argument as its value.
- Side Effect: Assigning the value of its 2nd argument to its 1st argument.

For example, suppose we want the value of E to be 2.72. We can do this as follows:

```
* (SETF E 2.72)
2.72
* E
2.72
*
```

Another example,

```
* (SETF ab-list '(a b))
(A B)
* ab-list
(A B)
* (setf ab-list '(c d e))
(C D E)
* ab-list
(C D E)
*
```

- SETF accepts multiple symbol-value pairs but only the value of the final argument is returned.

2.3 Some Notes on LISP Primitives

Note that:

- Binding is reserving a place in computer memory to store a value for a symbol. LISP binds symbols as they are encountered.
- Assignment is storing a value in that place (using SETF).
- Evaluation is recovering a value from that place.

It is important to know how to create new data, to get at old data, and to change old data. Consequently, for any kind of data, there are three groups of procedures:

```
Name of procedure       Purpose
=================       =======
Constructors            Create new data.
Selectors               Get at old data.
Mutators                Change old data.
```

3 List Selectors

These are the basic list selectors:

```
Selector  Purpose
========  =======
FIRST     Return 1st element of the list.
REST      Return list after removing the first element.
LAST      Return list after removing all but
            one element.
BUTLAST   Return list after removing the last n
            elements of a list, with the 2nd argument
            determing the exact number.
NTHCDR    Return list after removing the first
            n elements of a list, with the 1st argument
            determining the exact number.
LENGTH    Return the number of top-level elements
            in a list.
```

For example,

```
* (setf mylist '(fast computers are nice))
(FAST COMPUTERS ARE NICE)
* (first mylist)
FAST
* (rest mylist)
(COMPUTERS ARE NICE)
* (last mylist)
(NICE)
* (butlast mylist 2)
(FAST COMPUTERS)
* (nthcdr 2 mylist)
(ARE NICE)
* (length mylist)
4
*
```

3.1 Notes on List Selectors

- In addition to FIRST, Common LISP has SECOND and THIRD; bigger LISPs have even more.
- FIRST, REST, and the other selectors can be combined, just as arithmetic primitives can be.

  For example,

  ```
  * (FIRST (REST '(A B C)))
  B
  *
  ```

- LISP evaluates a procedure's arguments before evaluating the procedure itself.

3.2 Quoting Stop Evaluation

How can LISP know which is a specification of function call and which is a data item?
For example,

```
* (rest (a b c ))
```

'a' may be some sort of user-defined function! Should LISP evaluate it or should it just regard it as data ?

- You can specify where to stop evaluation by adding a single-quote character '.
- Expression after the quote character, ', will be regarded as data and NOT be evaluated.

For example,

```
* (first (rest '(a b c)))
B
* (first '(rest (a b c)))
REST
* (setf L '(first (rest (a b c))))
(FIRST (REST (A B C)))
*
```

Another example,

```
* (SETF L (PHW KAP DCB))
Error
*
```

The problem is that PHW is not a procedure name, yet PHW is the symbol in the first, procedure-name position in the form to be evaluated.

3.3 Exercises:

```
1. (first '(p h w))                        P
2. (rest '(b k p h))                       (K P H)
3. (first '((a b) (c d)))                  (A B)
4. (rest '((a b) (c d)))                   ((C D))
5. (first (rest '((a b) (c d))))           (C D)
6. (rest (first '((a b) (c d))))           (B)
7. (rest (first (rest '((a b) (c d)))))    (D)
8. (first (rest (first '((a b) (c d)))))   B
9. (rest '(c))                             NIL
```

```
10. (first ())                          NIL
11. (rest ())                           NIL
12. (nthcdr 2 '(a b c))                 (C)
13. (butlast '(a b c))                  (A B)
14. (last '(a b c))                     (C)
15. (last '((a b) (c d)))               ((C D))
16. (nthcdr 50 '(a b c))                NIL
17. (butlast '(a b c) 50)               NIL
18. (last ())                           NIL

[How to get the last element of a list ?]

* (first (last '(a b c)))
C
```

4 List Constructors

To put lists together, primitives called CONSTRUCTORS are used.

4.1 List Constructors — CONStruct

Takes an expression and a list and returns a new list whose 1st element is the expression and whose remaining elements are those of the old list.
For example,

```
         +-----+
         |     |
         |     v
* (CONS 'A   '(  B C) )
(A B C)

Assuming L's value is (A B) and
M's value is ((L M) (X Y)), then:

                            +---------+
                            |         |
* (CONS L M)                |         v
((A B) (L M) (X Y))       (A B)    (      (L M) (X Y))

* (cons L L)
((A B) A B)
* (cons '(a) '(b c))
((A) B C)
```

You will often combine SETF and CONS this way:

```
(SETF <name of a list>
      (CONS <new element> <name of the list>))
```

For example,

```
Assuming L's value is (Y Z).

* (SETF L (CONS 'X L))
(X Y Z)
* L
(X Y Z)
```

Note that L's OLD value is found, then the CONS makes a NEW list, and then SETF attaches the NEW list to the same symbol, L:
Without SETF, L's value is not changed after CONS:

```
* (CONS 'X L)
(X Y Z)
* L
(Y Z)
```

4.2 List Constructors — APPEND

Combines the elements of all lists supplied as arguments.
For example,

```
* (APPEND '(A  B  C) '(X  Y  Z))
           |  |  |    |  |  |
           |  |  |    |  |  |
           v  v  v    v  v  v
          (A  B  C    X  Y  Z)
```

Compare the result with what you would get with CONS:

```
* (CONS '(A B C) '(X Y Z))
((A B C) X Y Z)
```

- APPEND makes a list out of all the elements in its arguments.
- CONS adds its first argument to its second argument, a list.

Another example,

```
* (cons '(a b c) '())
((A B C))
* (append '(a b c) '())
(A B C)
* (append '(a b c) '(()))
(A B C NIL)
```

4.3 List Constructors — LIST

Make a list out of its arguments. Each "argument value" becomes an element of the new list.
For example,

```
* (LIST 'A 'B 'C)
(A B C)
* (list 'a 'b '(c d))
(A B (C D))

ASSUMING PI's value is 3.14,

* (list 2 PI 3)
(2 3.14 3)
* (list 2 'PI 3)
(2 PI 3)
```

Another example,

```
* (setf obstacles (list 'one 'two))
(ONE TWO)
* obstacles
(ONE TWO)
```

4.4 List Constructors — REVERSE

Reverse the order of the top-level elements of a list.
For example,

```
* (REVERSE '(A B C))
(C B A)
```

Note, however, that REVERSE does not turn the individual elements of a list around when those elements are lists:
For example,

```
* (REVERSE '((A B) (L M) (X Y)))
((X Y) (L M) (A B))
```

4.5 Exercises:

```
1. (cons '(a b c)'())                  ((A B C))
2. (append '(a b c)'())                (A B C)
3. (list '(a b c)'())                  ((A B C) NIL)

Assume L's value is (A B C), M's value is (X Y Z)

4. (list L M)                          ((A B C)(X Y Z))
5. (list 'L M)                         (L (X Y Z))
6. (list L 'M)                         ((A B C) M)
7. (append L M)                        (A B C X Y Z)

8. * (setf M (cons 'this (setf L '(is a list))))
   (THIS IS A LIST)
   * L
   (IS A LIST)
   * M
   (THIS IS A LIST)
   * (cons L L)
   ((IS A LIST) IS A LIST)
   * (list L L)
   ((IS A LIST) (IS A LIST))
   * (append L L)
   (IS A LIST IS A LIST)

9. (reverse '(a b c))                   (C B A)
10. (reverse '((a) (b) (c)))            ((C) (B) (A))
11. (reverse '((a b c)))                ((A B C))
```

5 Procedure Definition and Binding

5.1 Procedure Definition and Binding — DEFUN

LISP encourages you to create small, easily debugged procedures.
DEFUN $\longrightarrow$ DEfine FUNction

```
(defun  <function name>
   (<parameters>)
    <forms>)
```

- The DEFUN function establishes a function definition as its side-effect.
- The value returned by DEFUN is the function name.

For example,
Suppose you want to make a new list out of the first and last elements of an old list.

```
* (DEFUN BOTH-ENDS    ;Procedure's name is BOTH-ENDS.
   (L)                ;The parameter is L.
   (CONS (FIRST L) (LAST L))  ;The form.
   )                  ;End the DEFUN primitive
BOTH-ENDS             ;LISP returns the function name
* (BOTH-ENDS '(A B C D E))
(A E)
* (BOTH-ENDS '(START L M N FINISH))
(START FINISH)
```

- When calling a function, the value returned by that function is the value of the last form in its body.

For example,

```
* (defun cons-tail (tail alist)  ;fcn name, parameters
     (setf count (+ count 1))    ;first form
     (append alist (list tail))) ;last form
CONS-TAIL
* (cons-tail 'pear '(apple orange))
(APPLE ORANGE PEAR)
```

Virtual 'fences' between variables inside and outside a function.

- PARAMETER VARIABLES bindings are established when a function is entered. They are isolated from other variable bindings outside the function.

For example,

```
* (setf L '(a b c))
(A B C)
* (defun both-ends (L)
   (cons (first L)(last L)))
BOTH-ENDS
* (both-ends '(d e f))
(D F)
* L
(A B C)
```

- NON-PARAMETER variables are not isolated from variables outside the function.

For example,

```
* (setf count 0)
0
* (defun cons-tail (tail alist)
   (setf count (+ count 1))
   (append alist (list tail)))
CONS-TAIL
* (cons-tail 'a '(b c))
(B C A)
* count
1
```

5.2 Procedure Definition and Binding — LET

```
(LET ((<parameter 1> <initial value 1>)
      (<parameter 2> <initial value 2>)
      ...
      (<parameter m> <initial value m>))
     <form 1>
     <form 2>
     ...
     <form n>)
```

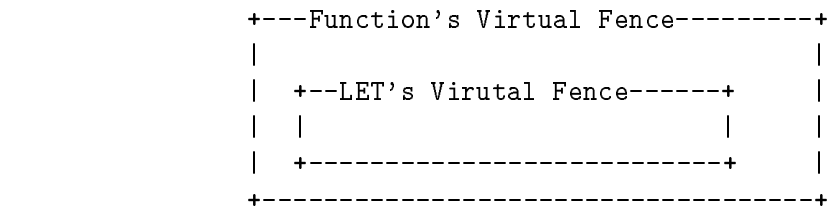
- The LET function binds parameters (and assigns values to them) and defines the scope in which these parameters and their corresponding values are valid.

For example,

```
* (LET ((X 3) (Y 5))
      (* X Y))
15
```

- The 'parameter' bindings within LET form are also isolated from the outside by a virtual 'fence', just like the local variables in a procedure of Pascal. Again, the non-parameter variables within a LET form is not isolated.

- Whenever a LET form appears in a function, the virtual fence of the LET form occurs within the virtual fence of the function.



For example,

```
* (defun test1 ()
     (setf acc 0)
     (let ((x 1) (y 2))
          (setf acc (+ x y))))
...
* (defun test2 ()
     (setf acc 0)
     (let ((x 1) (y 2) (acc 0))
          (setf acc (+ x y))))
...

* (test1)          * (test 2)
3                  3
* acc              * acc
3                  0
```

Note carefully that all initial-value forms are evaluated before the new parameter values are assigned.

```
                     These forms are ALL evaluated,
                         |
                         |
(LET ((<parameter 1> <value 1>)
       ...
      (<parameter n> <value n>))
            |
            +----- before ANY of these parameters are set.

     <form 1> ... <form n>)
```

We say that the LET parameter values are computed in PARALLEL.

```
* (setf x 'outside)
...
* (let ((x 'inside) (y x))
       (list x y))
(INSIDE OUTSIDE)
```

LET*

LET* binds parameter sequentially so that a parameter bound earlier can be used to evaluate the value of a parameter bound later.
For example,

```
* (setf x 'outside)
...
* (let* ((x 'inside) (y x))
        (list x y))
(INSIDE INSIDE)
```

6 Predicates and Conditionals

A predicate is a procedure that returns a value that is either true (T) or false (NIL).

6.1 Equality Predicates

```
EQUAL  Are the 2 arguments the same expr (list or atom)?
EQL    Are the 2 arguments the same symbol or number?
EQ     Are the 2 arguments the same symbol?
=      Are the 2 arguments the same number?
```

For example,

```
* (setf abc-list '(a b c)
        one 1 two 2 plus '+ minus '-)
-
* (equal abc-list '(a b c))
T
* (equal one 2)
NIL
* (equal abc-list one)
NIL
* (equal plus minus)
NIL
* (eql plus '+)
T
* (eq minus -1)
ERROR          <---- "EQ" expects symbols
* (= one 'a)
ERROR          <----  "=" expects numbers
* (eql 4 4.0)
NIL
* (eql 4 4)
T
* (= 4 4.0)
T
```

6.2 Membership Predicate

The MEMBER predicate test to see if its 1st argument is an (top level) element of its 2nd argument.

- MEMBER returns what is left in the list when the matching symbol is encountered.
- MEMBER returns NIL if the 1st argument is not a top level element of the 2nd element.

For example,

```
* (setf sentence '(tell me about your mother please))
(tell me about your mother please)
* (member 'mother sentence)
(MOTHER PLEASE)  <--- a more informative return value
                      instead of just a "T".
* (member 'father sentence)
NIL
* (MEMBER 'MOTHER '((FATHER SON) (MOTHER DAUGHTER)))
NIL
```

MEMBER normally tests things with a predicate that works with symbols only. If the first argument to MEMBER is not a symbol, you must modify slightly what you write; otherwise MEMBER won't work as you expect.

```
* (SETF PAIRS '((FATHER SON) (MOTHER DAUGHTER)))
((FATHER SON) (MOTHER DAUGHTER))
* (member '(father son) pairs)
NIL
* (MEMBER '(FATHER SON) PAIRS :TEST 'EQUAL)
T
```

6.3 Data Types Predicates

```
ATOM      Is it an atom?
NUMBERP   Is it a number?
SYMBOLP   Is it a symbol?
LISTP     Is it a list?
```

For example,

```
* (atom 'pi)
T
* (atom pi)
T
* (numberp 'pi)
NIL
* (numberp pi)
T
* (symbolp 'pi)
T
* (symbolp pi)
NIL
* (atom nil)
T
* (atom ())
T
* (symbolp nil)
```

```
T
* (symbolp ())
T
* (listp nil)
T
* (listp ())
T
* (numberp ())
NIL
```

6.4 Empty-List Predicates

```
NULL   Is the argument an empty list ?
ENDP   Is the argument, which must be a list,
       an empty list ?
```

For Example,

```
* (null ())
T
* (endp nil)
T
* (null '(a b c))
NIL
* (endp '(a b c))
NIL
* (null 'a-symbol)
NIL
* (endp 'a-symbol)
ERROR     <--- a-symbol is not a list
```

6.5 Number Predicates

```
NUMBERP    Is it a number?
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?
```

For example,

```
* (setf zero 0 one 1 two 2 three 3 four 4)
...
* (setf digits (list one two three four))
...
* (numberp 4)
```

```
T
* (numberp four)
T
* (numberp 'four)
NIL
* (numberp digits)
NIL
* (numberp 'digits)
NIL
* (zerop zero)
T
* (zerop 'zero)
ERROR            <------ zerop expects a number
* (zerop four)
NIL
```

Further examples,

```
* (plusp one)
T
* (plusp (- one))
NIL
* (plusp zero)
NIL
* (evenp (* 9 5 3 2 1))
T
* (evenp (* 10 8 6 4 2))
T
* (> four two)
T
* (> two four)
NIL
* (> three two one)
T
* (> three one two)
NIL
* (> 'four 'two)
ERROR         <------ not numbers
```

6.6 Combining Predicates — AND

- AND returns NIL if any of its argument evaluates to NIL.
- The arguments are evaluated from left-to-right, if any evaluates to NIL, *none of the remaining is evaluated*, and the value returned is NIL.
- If all arguments evaluate to non-NIL values, the value returned by the AND from is the value of the last argument.

- It takes any number of arguments.

For example,

```
* (setf pets '(dog cat))
...
* (and (member 'dog pets) (member 'tiger pets))
NIL
* (and (member 'dog pets) (member 'cat pets))
(CAT)
```

Things are not that simple, however, because AND may not evaluate all their arguments.

```
* (setf result NIL pi 3.14)
3.14
* (AND (LISTP PI) (SETF RESULT 'SET-IN-FIRST-AND))
NIL            ;Returns NIL because PI is not a list.
* RESULT
NIL
* (AND (NUMBERP PI) (SETF RESULT 'SET-IN-SECOND-AND))
SET-IN-SECOND-AND
* RESULT
SET-IN-SECOND-AND
```

6.7 Combining Predicates — OR

- OR returns NIL if all of its arguments evaluates to NIL.
- The arguments are evaluated from left to right, if any evaluates to non-NIL, *none of the remaining is evaluated*, and the value returned is that non-NIL value.
- If all arguments evaluates to NIL, the value returned by OR is NIL.
- It takes any number of arguments.

For example,

```
* (setf pets '(dog cat))
...
* (or (member 'dingo pets) (member 'tiger pets))
NIL
* (or (member 'dog pets) (member 'cat pets))
(DOG CAT)
```

Things are not that simple, however, because OR may not evaluate all their arguments.

```
* (setf result NIL pi 3.14)
3.14
* (OR (NUMBERP PI) (SETF RESULT 'SET-IN-FIRST-OR))
T
* RESULT
NIL
* (OR (LISTP PI) (SETF RESULT 'SET-IN-SECOND-OR))
SET-IN-SECOND-OR
* RESULT
SET-IN-SECOND-OR
```

6.8 Combining Predicates — NOT

- NOT just turns non-NIL to NIL and NIL to T

For example,

```
* (not 'dog)
NIL
* (setf pets '(dog cat))
(DOG CAT)
* (not (member 'dog pets))
NIL
* (not (member 'tiger pets))
T
```

7 Simple Branching Primitives

7.1 Simple Branching Primitives — IF

```
(IF <test>  <then form>  <else form>)
```

IF returns the value of its <then form> when <test> evaluates to T or ANYTHING other than NIL; otherwise, IF returns the value of its <else form>.
For example,

```
Assuming TRIGGER'S value is T.

* (IF TRIGGER 'ITS-TRUE 'ITS-FALSE)
ITS-TRUE

Now assume TRIGGER's value is NIL:

* (IF TRIGGER 'ITS-TRUE 'ITS-FALSE)
ITS-FALSE
```

Another example,

```
* (setf day-or-date 'monday)
...
* (if (symbolp day-or-date) 'day 'date)
DAY
* (setf day-or-date 9)
...
* (if (symbolp day-or-date) 'day 'date)
DATE
```

If there's no <else-form> and if <test> evaluates to NIL, NIL will be returned.

7.2 Simple Branching Primitives — WHEN

```
(WHEN  <test>  <then form>)
```

WHEN returns the value of <then form> when <test> evaluates to ANYTHING other than NIL. Said another way, WHEN triggers on non-NIL.
For example,

```
* (WHEN T 'ITS-TRUE)
ITS-TRUE
```

When WHEN's first argument evaluates to NIL, WHEN returns nil:

```
* (WHEN NIL 'ITS-TRUE)
NIL
```

Another Example,

```
* (setf high 98 temperature 102)
...
* (when (> temperature high) (setf high temperature))
102
* high
102
```

7.3 Simple Branching Primitives — UNLESS

```
(UNLESS  <test>  <else form>)
```

UNLESS returns the value of <else form> when <test> evaluates to NIL. Said another way, UNLESS triggers on seeing NIL.
For example,

```
* (UNLESS NIL 'ITS-FALSE)
ITS-FALSE
```

When UNLESS's first argument evaluates to non-NIL, UNLESS returns NIL:

```
* (UNLESS NIL 'ITS-FALSE)
* (UNLESS T 'ITS-FALSE)
NIL
```

Another example,

```
* (setf high 98 temperature 102)
...
* (unless (< temperature high) (setf high temperature))
102
* high
102
```

8 General Branching Primitives

8.1 General Branching Primitives — COND

```
(COND (<test 1> <consequent 1-1> <consequent 1-2> ...)
      (<test 2) <consequent 2-1> <consequent 2-2> ...)
      ...
      ...
      (<test m> <consequent m-1> <consequent m-2> ...))
```

- The symbol COND is followed by clauses. Each clause contains a test and zero or more forms called consequent.
- COND moves through the clauses, evaluating the test forms, until a test form evaluates to non-NIL. This clause is "triggered" and its consequent forms are evaluated.
- The value returned by COND is the value of the last consequent form in the triggered clause.
- If all test forms are NIL, the value returned by COND is also NIL.

For example,

```
* (cond ((= N 0) (setf answer '(value is zero)))
        ((> N 1) (setf answer '(value is positive)))
        (T       (setf answer '(value is negative))))
```

- The last clause in the above example is called a T-triggered clause.
- With a T-triggered clause, the final clause is used when none of the others are triggered.

8.2 General Branching Primitives — CASE

```
(CASE  <key form>
  (<key 1> <consequent 1-1> <consequent 1-2> ...)
  (<key 2> <consequent 2-1> <consequent 2-2> ...)
  ...
  ...
  (<key m> <consequent m-1> <consequent m-2) ...))
```

- CASE checks the <key form> against the <key> in each clause using "EQL" until a matching is found. The corresponding clause is then triggered and all its consequent are evaluated.
- The value returned by CASE is the value of the last consequent form in the triggered clause.
- If no matching occurs, the value returned by CASE is NIL.

For example,

```
* (setf thing 'point r 1)
...
* (case thing
      (circle (* pi r))
      (sphere (* 4 pi r r)
      (otherwise 0))
0
```

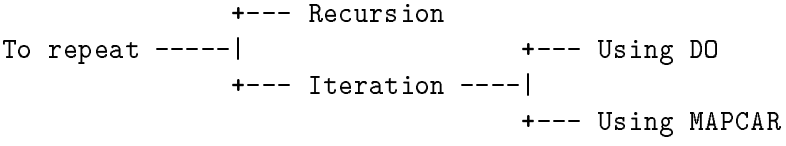
- The last clause in the above example is called a Catch-all clause.
- With a Catch-all clause, the final clause is used when none of the others are triggered.
- If the <key> in a clause is a list, not an atom, CASE checks the <key form> against the <key> list using MEMBER

For example,

```
* (setf thing 'ball r 1)
...
* (case thing
     ((circle wheel) (* pi r r))
     ((sphere ball)  (* 4 pi r r))
     (otherwise 0))
12.566
```

9 Repeating by Recursion

Many LISP procedures work by repeating a particular action over and over until a certain condition is met. There are several ways that this is arranged:



An example for recursion:

```
(DEFUN COUNT-ELEMENTS (L)
  (IF (ENDP L) 0
      (+ 1 (COUNT-ELEMENTS (REST L)))))
```

The definition of COUNT-ELEMENTS involves a form using COUNT-ELEMENTS itself. Such procedures are said to be RECURSIVE.
Many recursive procedures solve problems by breaking up a list, working on its pieces, and combining the results of working on those pieces. One particularly nice example is COUNT-ATOMS:
For example,

```
(DEFUN COUNT-ATOMS (E)
  (COND ((AND (LISTP E) (ENDP E)) 0)
        ((ATOM E) 1)
        (T (+ (COUNT-ATOMS (FIRST E))
              (COUNT-ATOMS (REST E))))))
```

10 Repeating by Iteration

You will often want to do a computation over and over until one of the two following criteria is satisfied:

1. A test has been satisfied.
2. All elements in a list have been worked on.

Let's start with DOs.

10.1 Repeating by Iteration — DOTIMES

```
(dotimes (<count parameter> <upper-bound form> <result form>)
         <body>)
```

- When DOTIMES is entered, the <upper-bound form> is evaluated, producing a number of n. The <count parameter> is then assigned from 0 to n-1, one after another. For each value, the body is executed once.
- On exit, the <count parameter>'s binding is eliminated and the <result form> is evaluated as the return value of DOTIMES.

```
* (setf result 1)
1
* (dotimes (count 5 result)
   (setf result (* (+ 1 count) result)))
120
*
```

10.2 Repeating by Iteration — DOLIST

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

- When DOLIST is entered, the <list form> is evaluated, producing a list of elements. The elements in the list are then assigned, one after another, to the element parameter. For each value, the body is executed once.
- On exit, the element parameter's binding is eliminated and the <result form> is evaluated as the return value of DOLIST.

For example,

```
* (LET ((foo '(a b (c d))) (bar '()))
       (DOLIST (ele foo bar)
       (SETF bar (CONS ele bar))))
((C D) B A)
*
```

Whenever a (RETURN <expression>) is encountered in DOTIMES or DOLIST, computation will be terminated immediatedly. The <expression> is evaluated as the return value of the terminated DOTIMES and DOLIST form.

10.3 Repeating by Iteration — DO

DO expressions have the following parts:

1. A list of parameter specifications, each of which creates, sets, and resets one of the DO form's parameters:

   ```
   (<parameter> <initial value> <reset form>)
   ```

2. A test and return clause that determines when to stop the iteration and specifies the DO form's value once stopped:

   ```
   (<trigger> <side effect forms, if any> <value form>)
   ```

3. A body that consists of DO's subforms, which are evaluated over and over until the DO is stopped:

   ```
   <body form>
   ```

The syntax of DO:

```
(DO ((<parameter 1> <initial value 1> <update form 1>)
     (<parameter 2> <initial value 2> <update form 2>)
     ...
     (<parameter n> <initial value n> <update form n>))
    (<test form>  <result form>)
    <body form>)
```

For example,

```
* (DO ((L '(THIS IS A LIST) (REST L))
       (RESULT NIL))
      ((NULL L) RESULT)
      (SETF RESULT (CONS (FIRST L) RESULT)))
(LIST A IS THIS)

* (defun do-exp (m n)
    (do  ((result 1)
          (exponent n (- exponent 1)))
         ((zerop exponent) result)
         (setf result (* m result))))
```

- On entering the DO, the list of parameters are all bound to its corresponding value (again, a virtual fence exists to isolate these parameters from the variables outside DO).

  The parameter specifications can include update forms. The parameters are updated accordingly in each pass.

- The 2nd part of DO is the termination test and the result form. The test is attempted before each pass, including the 1st one. The <result form> is evaluated as the return value of DO only when the test succeeds.

  There may be zero or more <result forms> after the <test form>. They are all evaluated when the test succeeds. However, only the last one gives the return value of the DO. If there is none, the return value is NIL.

- Whenever (RETURN <expression>) is encountered, DO is terminated immediately. <expresion> is evaluated as the return value of the terminated DO.

- All initializations and updates are done in parallel, i.e. all initial forms are evaluated before bindings and all update forms are evaluted before assignments.

For example,

```
* (defun do-exp (m n)
     (do ((result m (* m result))
          (exponent n (- exponent 1))
          (counter (- exponent 1) (- exponent 1)))
         ((zerop counter) result)))
```

Errors occurs because EXPONENT is not bound to any value yet when it is used to evaluate the initial form of COUNTER.
DO* like LET* binds values sequentially.

10.4 Repeating by Iteration — LOOP

`(loop <body>)`

The body is evaluated over and over until a (RETURN <expression>) is encountered. Again the <expression> is evaluated as the return value of the terminated LOOP.

11 Miscellaneous Primitives

11.1 Transformation Primitive — MAPCAR

```
(MAPCAR #'<procedure name> <list of things to work on>)
```

MAPCAR takes two arguments: a procedure name and a list of things to work on. The result is the list of values you would have if you worked on each element in the given list with the given procedure.
For example,

```
* (MAPCAR #'FIRST '((A B C) (X Y Z)))
(A X)
```

- FIRST's result, working on the first element, (A B C), is A.
- FIRST's result, working on the second element, (X Y Z), is X.
- All the results of FIRST, made into a list, is (A X).

If the transformation procedure requires more than 1 parameter, there must be a corresponding number of lists.
For example,

```
* (MAPCAR #'= '(1 2 3) '(3 2 1))
(NIL T NIL)
```

In other words, the transformation procedure is applied to the FIRST element of each list, then the SECOND element of each list, and so on until no more elements remain in one of the lists.

```
* (MAPCAR #'list '(a b c) '(1 2) '(x y z))
((A 1 X) (B 2 Y))
```

11.2 Filtering Primitive — REMOVE-IF, REMOVE-IF-NOT

```
(remove-if #'<filtering procedure name>
     <list of elements to be filtered>)

(remove-if-not #'<filetering procedure name>
         <list of elements to be filtered>)
```

- REMOVE-IF eliminates all elements that satisfy the filtering predicate.
- REMOVE-IF-NOT eliminates all elements that do not satisfy the filtering predicate.

For example,

```
* (REMOVE-IF #'evenp '(1 2 3 4 5))
(1 3 5)
* (REMOVE-IF-NOT #'evenp '(1 2 3 4 5))
(2 4)
```

11.3 Counting Prmitive - COUNT-IF

```
(count-if #'<filtering procedure>
           <list of elements to be counted>)
```

- COUNT-IF provides a way to count the number of elements in a list which satisfy a given condition.

For example,

```
* (count-if #'evenp '(1 2 3 4 5))
2
```

11.4 Finding Primitive — FIND-IF

```
(find-if #'<test procedure>
          <list of elements to be tested>)
```

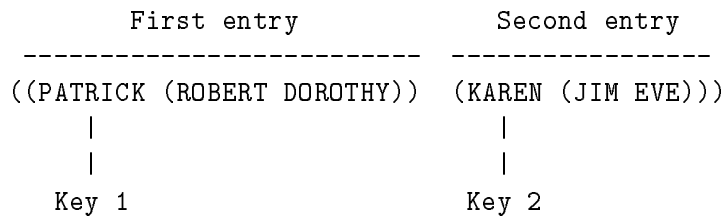
- FIND-IF finds the first element in the list which satisfy the testing predicate.

For example,

```
* (find-if #'evenp '(1 2 3 4 5 6))
2
```

12 Association List

An association list is a list of lists. The first element of each sublist is called a key.
Here is an example of an association list used to record the parents of PATRICK and KAREN:



12.1 ASSOC

```
(ASSOC <key> <association list>)
```

For example,

```
Assocation List:
Parents ==> ((PATRICK (ROBERT DOROTHY)) (KAREN (JIM EVE)))

 * (ASSOC 'PATRICK PARENTS)
 (PATRICK (ROBERT DOROTHY))

 * (ASSOC 'KAREN PARENTS)
 (KAREN (JIM EVE))
```

12.2 Exercise:

```
PARENT :
     ((PATRICK (ROBERT DOROTHY)) (KAREN (JIM EVE))
      (PATRICK (JOHN SARAH)) (SARAH (ISAAC MARY)))
```

How to retrieve the second instance of PATRICK in the association list PARENT ?

13 Property List

A symbol may have property values :

```
symbol   parent property  children property  spouse property
======   ===============  =================  ===============
PATRICK  (ROBERT DOROTHY) (SARAH)            (CANDY)
```

13.1 Retrieving a property value

```
(GET  <symbol>  <property name>)
```

For example,

```
* (get 'patrick 'parent)
(ROBERT DOROTHY)

* (get 'patrick 'grandparent)
NIL     ;;because no such property exists
```

13.2 Storing a property value

```
(SETF (GET <symbol> <property name>) <property value>)
```

For example,

```
* (setf (get 'patrick 'parent) '(albert dora))
(ALBERT DORA)

* (get 'patrick 'parent)
(ALBERT DORA)
```

13.3 Removing a property value

```
(REMPROP <symbol> <property name>)
```

For example,

```
* (remprop 'patrick 'spouse)
T

* (get 'patrick 'spouse)
NIL
```

N.B. If a symbol is a property list, only its properties have values. The symbol itself does not have any value:

```
* patrick
ERROR        ;; unbound variable
* (setf albert 'patrick)
PATRICK
* (setf albert patrick)
ERROR
```

□ *END.*