G22.2210

Programming Languages: PL

B. Mishra
New York University.

Lecture # 14

—Slide 1—

Global and Local Variables

- **Global Variables**
  - Global Variables may be referenced in any function
  - They must be declared using the special function `DEFVAR`

```
(DEFVAR *COUNT* 0)
(DEFUN COUNT-CONS (X Y)
   (PROGN (SETQ *COUNT* (+ 1 *COUNT*))
          (CONS X Y)))
(DEFUN COUNT APPEND (X Y)
   (IF (NULL X)
       Y
       (COUNT-CONS (CAR X)
                   (COUNT-APPEND (CDR X) Y))))
```

- **NOTE:** `PROGN`: Explicitly sequences LISP statements. Value of the last subform is returned as the value of the `PROGN`-form

```
(SETQ *COUNT* 0)
(COUNT-APPEND ´(A B C) ´(D))  =>  (A B C D)
*COUNT*                       =>  3
```


—Slide 2—

LOCAL VARIABLES

- Local variables may only be referenced in the function in which they are defined.
- They can be declared by appearing as function's formal arguments, Or they can be declared explicitly by the control structure `LET` & `LET*`

```
(LET ((<var-1> <value-1>)
        ...
       (<var-n> <value-n>)
      <body> ))
```

1. Each of the S-expression `<value-1>`, ..., `<value-n>` is evaluated in turn.
2. The variables `<var-1>`, ..., `<var-n>` are given their respective values.
3. Evaluate `<body>`
4. In this evaluation `<value-j>` cannot refer to `<var-i>` even if $1 \leq i, j \leq n$.

—Slide 3—

`LET` *and* `LET*`

- **Example**

```
(DEFUN DISTANCE (P1 P2)
   (LET ((XDIFF (- (CAR P1) (CAR P2)))
         (YDIFF (- (CAR (CDR P1)) (CAR (CDR P2))))
      (SQRT (+ (* XDIFF XDIFF) (* YDIFF YDIFF))))
   ))
```

- `LET*`
  - ◦ Sequentially binds each new variable as its value is computed
  - ◦ Avoids the "*right crawl*" problem

```
(DEFUN PAINT-COST (COLOR)
  (LET ((PAIR (ASSOC COLOR
        ´((BLUE . 8.00) (RED . 5.50) (YELLOW . 13.25)))))
    (LET ((PRICE (IF (NULL PAIR)
                     *DEAFULT-PAINT-PRICE*
                     (CDR PAIR))))
       (+ PRICE (* *TAX-RATE* PRICE)))))
```

—Slide 4—

LET*: *(contd)*

- **Old Example**

```
   COLOR = BLUE
=> PAIR = (BLUE . 8.00)
=> PRICE = 8.00
=> PRICE = PRICE + *TAX-RATE* PRICE
```

- Old example with LET*

```
(DEFUN PAINT-COST (COLOR)
  (LET* ((PAIR (ASSOC COLOR
         ´((BLUE . 8.00) (RED . 5.50) (YELLOW . 13.25))))
         (PRICE (IF (NULL PAIR)
                    *DEAFULT-PAINT-PRICE*
                    (CDR PAIR))))
     (+ PRICE (* *TAX-RATE* PRICE))))
```


—Slide 5—

Lisp as Data Bases

- Lists can associate keys to values: ASSOC
- **Association List**

```
((<key-1>.<val-1>) (<key-2>.<val-2>)...(<key-n>.<val-n>))
```

- ASSOC searches the list linearly until
  1. It drops off the list and returns NIL, or
  2. It finds the key (EQL) and returns the cons-cell containing the key

```
(ASSOC 3 ´((1 PARTRIDGE)
           (2 TURTLE DOVES)
           (3 FRENCH HENS)
           (4 TURTLE DOVES)
           (5 GOLD RINGS)))
=>
 (3 FRENCH HENS)
```


—Slide 6—

Functional Programming Style

- FUNCALL

  ◦ It is possible to pass functions as values (i.e., data) and apply them to arbitrary sets of arguments.

```
(SYMBOL-FUNCTION <symbol>)     or #<symbol>
```

⇒ Returns *functional object* associated with <symbol>

```
(FUNCALL <functional-object> <arg-1> ... <arg-n>)
```

⇒ Calls the <functional-object> with the arguments it received.

—Slide 7—

Examples

```
(SETQ *RELATIONSHIP-FUNCTIONS*
      ´((FATHER . FATHER-OF?)
        (MOTHER . MOTHER-OF?)))

(DEFUN FIND-RELATIVE (RELATION PERSON)
   (LET ((FUN-NAME (CDR (ASSOC RELATION
                      *RELATIONSHIP-FUNCTION*))))
        (IF (NULL FUN-NAME)
            (ERROR "Unknown relationship")
            (FUNCALL (SYMBOL-FUNCTION FUN-NAME)
                     PERSON))))

(FIND-RELATIVE (FATHER TOM))
=>
    (FUNCALL (SYMBOL-FUNCTION ´FATHER-OF?) ´TOM)
=   (FUNCALL #´FATHER-OF? ´TOM)
=   (FATHER-OF? ´TOM)
```

- **Note**: In interpreted LISP, you may omit SYMBOL-FUNCTION (i.e., #)

```
(FUNCALL ´(LAMBDA (N) (+ 1 N)) 3) => 4
```

—Slide 8—

APPLY & LAMBDA

- `APPLY`

```
(APPLY <fun-obj> <arg-1> ...<arg-n> <arg-more>)
```

⇒ Calls the functional object with *variable* number of arguments and they may be in a list.

```
(APPLY #'+ '(1 2 3 4 5 6))          =>  21
(APPLY #'* 2 3 '(4 5 6))            =>  120
```

- `LAMBDA`

```
(LAMBDA <arg-list> ...<body>)
```

⇒ Lambda Expression (λ. `<arg-list>`) `<body>`. It is like `DEFUN`...except that it makes an *anonymous functional object*

```
  (APPLY
        #'(LAMBDA (A B C) (* A (+ B C)))
        '(4 3 5))
=> 4 * (3 + 5) = 32
```


—Slide 9—

Mapping Functions: MAPCAR

- Under mapping, a function is successively applied to applied to one more lists

```
(MAPCAR <function> <arglist-1>...<arglist-n>)
```

1. `<function>` must take n arguments.
2. First, it is applied to the `CAR`'s of each `<arglist-i>`
3. Next, it is applied to the `CADR`'s, etc., until the end of the shortest list is reached
4. Results of each application are collected into a list and returned as the value of the `MAPCAR`

—Slide 10—

Examples

```
    (MAPCAR #'NUMBERP '(A 3 B 2 4 C 7))
=>  (NIL T NIL T T NIL T)


    (MAPCAR #'(LAMBDA (N) (+ 1 N))
             '(5 3 6 7 2))
=>  (6 4 7 8 3)


    (MAPCAR #'(LAMBDA (X Y) (CONS X Y))
             '(MIAMI DENVER OAKLAND LOS-ANGELES)
             '(DOLPHINS BRONCOS RAIDERS RAMS))
=> ((MIAMI.DOLPHINS) (DENVER.BRONCOS)
    (OAKLAND.RAIDERS) (LOS-ANGELES.RAMS))
```


—Slide 11—

How LISP Works

- Lisp can be used as an *interpretive* or a *compiled* language
- **Interpreter**

  When a lisp program is interpreted, the lists that represent the functions are examined by `APPLY` and `EVAL`

  <u>`APPLY` and `EVAL`</u>

  1. Cooperate to walk recursively down the list structure
  2. Associate symbols and values to handle argument lists when functions are entered and to handle `LAMBDA` argument lists in internal `LAMBDA` application
  3. (Other variable introduction mechanisms, e.g., `LET` and `DO`, are implemented via `LAMBDA`

—Slide 12—
LISP Compiler

- Similar to the Interpreter
- **Compiler**

  1. An assembly language program is produced
  2. *Semantically equivalent* to what would be achieved by interpreting the functions
  3. Resulting code is highly efficient
     Speed up of factor of 20 or higher

—Slide 13—
LISP Storage Layout

Address space is divided into <u>three</u> parts:

- **Stack**
  Temporary variables and return addresses from function calls
- **Binary Program Space**
  Actual machine instructions
- **Heap**
  Lisp data objects: Cons cells, symbols, numbers, strings, etc.
- Initially, a Lisp starts off with pieces of itself in all three parts

  User program provides additional objects into all three parts

—Slide 14—
LISP Internals

Internally, user objects are indistinguishable from objects in the internal Lisp system itself

- The stack and heap are made of **pointers**. A pointer typically consists of an address and few more bits of additional informations.
- Cons cell ≡ Pair of pointers
- A symbol ≡ Collection of Pointers pointing to
  a) Its value
  b) Its print name (a string)
  c) Its property list
  d) Its functional definition

—Slide 15—
Type of a LISP Object

A LISP object can be stored as:

- **BIBOP (BIg Bag Of Pages)**
  Location of the objects within the address space provides sufficient information. Each page of memory is dedicated to holding objects of a certain type.
- **Encoding in the Pointer (Tag bits)**
  Each pointer contains the address of a Lisp object and few tag bits specifying the type of the object pointed to
- **Encoding in the Object**
  The type of the object is encoded in the storage representation of the object itself. Thus type check has to access the object.

  (Usually this scheme is used in combination with the preceding tag bits encoding.)

—Slide 16—

OBARRAY *or* OBLIST

A vector of list of symbols
(Think of a Hash Table)

- Given the name of a symbol, first the string is "hashed" to compute its "hash number."
- The hash number is used to index into the OBARRAY to retrieve one of the lists of symbols.
- The list is searched to find the symbol and construct its cons cell representation.

—Slide 17—

Garbage Collection

During Lisp computation, periodically the heap becomes full and it is necessary to *free space* by identifying those cons cells that are *no longer accessible* (thus, usable).

A cons cell not accessible,

1. **if either there is no pointer to it**
2. **or all objects that point to it are themselves inaccessible.**

- Each cons cell has an extra bit ("**mark bit**") associated with it.
- At the start of the garbage collection, all mark bits are assumed to be "**off**."

—Slide 18—

Garbage Collection (contd)

- All the symbols and the stack are checked for pointers to cons cell. starting with these pointers, during a **mark phase**, every cons cell that is still accessible by the user's program has its mark bit set.

```
(DEFUN MARK-CELL (CONS-CELL)
   (COND ((NOT (MARKEDP CONS-CELL))
          (SETMARK CONS-CELL)
          (IF (CONSP (CAR CONS-CELL))
              (MARK-CELL (CAR CONS-CELL)))
          (IF (CONSP (CDR CONS-CELL))
              (MARK-CELL (CDR CONS-CELL))))))
```


—Slide 19—

Garbage Collection (contd)

- The heap is "**swept**." Each cons cell is examined with a linear sweep through the heap. If it is marked, the mark bit is turned **off** for the next round of garbage collection; otherwise, it is added to a free-list, *FREE-LIST*.

```
(DEFUN CHECK-CELL (CONS-CELL)
   (COND ((MARKEDP CONS-CELL)
          (UNSETMARK CONS-CELL))
         (T (RPLACD CONS-CELL *FREE-LIST*)
            (SETQ *FREE-LIST* CONS-CELL))))
```

—Slide 20—

Lisp Function with Garbage Collection

Now, we can write the LISP `CONS` function as follows:

```
(DEFUN CONS (X Y)
   (IF (NULL *FREE-LIST*)
       (GARBAGE-COLLECT))
   (LET ((CONS-CELL *FREE-LIST*))
        (SETQ *FREE-LIST* (CDR CONS-CELL))
        (RPLACA CONS-CELL X)
        (RPLACD CONS-CELL Y)
        CONS-CELL))
```

[End of Lecture #14]