V22.0490.001
Special Topics: Programming Languages

B. Mishra
New York University.

Lecture # 13

—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`

—Last Slide—

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))
```

[End of Lecture #13]