Multiple Value Functions

Wllensky Chapter 16.4

Multiple value functions

- In other languages one can pass multiple parameters to return multiple values (though not function values) on one call
  - **In Pascal, Turing use var - to change the parameter directly**
  - **In ADA declare a parameter as output**
- In Lisp all parameters are passed by value – they cannot be changed
- To return multiple values you need to construct a list of the results you want the function to return and the caller must extract, through car and cdr, the values of interest
- This occurs frequently enough that Lisp permits multiple values to be returned by a function.

Catching multiple values

- By default, if a function returns multiple values, only one is passed back – the rest are discarded – unless you specifically ask for the other values
- For example **(round aNumber)** returns two values – the rounded value and the value needed to add to the rounded result to get the original number

(round aNumber) ==> roundedValue difference

> where difference = aNumber - roundedValue

(round 7.6) ==> 8 -0.4

> Not a list! (car (round 7.6)) fails

> (print (round 7.6)) ==> 8

use first value by default

Catching multiple values – 2

- Use the following **macro** to create a list of multiple value returns

  (multiple-value-list (round aNumber))

  ==> (roundedValue restoreNumber)

  (multiple-value-list (round 7.6)) ==> (8 -0.4)

- Can assign the values to symbols using the following **macro**

  (multiple-value-setq (val diff) (round 7.6))

  > **8 ==> val and -0.4 ==> diff**

  > **Note setq implies global symbol**

Catching multiple values – 3

- Can create a local context for variables instead of using globals

```
(let ((val  nil) (diff nil))
     (multiple-value-setq (val diff) (round 7.6))
    ;; ... use val and diff in list of forms
     (print val)
     (print diff)
     (print (+ val diff))
)
```

  > setq actually uses the closest symbol in the environment
    - The first one found

Catching multiple values – 3a

- The following shows that let is syntactic sugar for a lambda function

```
( (lambda ( val  diff )
     (multiple-value-setq (val diff) (round 7.6))
    ;; ... use val and diff in list of forms
     (print val)
     (print diff)
     (print (+ val diff))
   )
   nil nil                    ; initial values for val & diff
)
```

Examine multiple-value-setq

- Use macroexpand-1

```
(macroexpand-1
        '( multiple-value-setq (val diff) (round 7.6))
)

(let* ((#:g6 (multiple-value-list (round 7.6)))
       (#:g7 (car #:g6)))
  (setq val (nth 0 #:g6))
  (setq diff (nth 1 #:g6))
  #:g7
)
```

- **#:g6** and **#:g7** are symbols generated by the macro. They are local to the **let*** form

Catching multiple values – 4

- Instead of using **let** which needs initial values for its parameters, can use the following

```
( multiple-value-bind (val diff ) (round 7.6  )
   ;; ... list of forms  using val and diff ...
  ( print val )
  ( print diff )
  ( print  ( +  val  diff ) )
)
```

Catching multiple values – 5

- Can use the following to pass the return values to a function

  > **Its arity equals the number of returned values**

```
( multiple-value-call  #'functionName  (round 7.6 ) )

(defun functionName (val diff)
   (print val) (print diff) (print (+ val diff))
)
```

Throwing multiple values

- The last form in a function is a call to **values**

```
(values 1 2 3)  ==>  1  2  3
```

- Here is a function to tear a list into its first and rest parts

```
(defun  unCons  ( theList )
  (values  ( car  theList )  ( cdr  theList ) )
)

(uncons  '( a  b  c ) )  ==>  a   ( b  c )
```

- What about unconsing an entire list? Use apply to strip the outer level of parenthesis

```
(apply 'values '(a b c d e))  ==>  a  b c d e
```

- Why would one want uncons?