Blocks, let, let* blocks

- Sometimes we want to run a set of functions in sequence (e.g. prompt then read)
- can be done at function level (using sequence of statements as body of function)
- can also be done anywhere a lisp function call is valid, using either block, let, or let*
- block permits a list of statements, while let and let* also allow declaration and use of local variables
- return value from block/let/let* is the return value of the last statement run

block

- Using block: first argument is a symbol that identifies/names the block (for use with goto’s later), the remaining arguments are statements to run, e.g.

```
; a block to prompt the user
;    then read and return their response
(block
   PromptNRead ; our symbol/identifier for the block
   (format t “enter something: ”)
   (read)) ; returns the result of the last action
```

Blocks fit anywhere

- Can be placed anywhere a lisp statement works:

```
(if ; a block returning the condition to check
    (block ; to return t iff they enter a number
        GetAndCheckNum
        (format t “enter a number”)
        (numberp (read)))
     ; now the t/f parts of the if
     (format t “Well done!”)
     (format t “That was not a number”))
```

Let blocks: local variables

- Act like blocks, but no name/identifier needed, instead declare a list of pairs specifying local variables/values

```
(let
    ((x 5) (y “foo”))  ; list of local vars/values
    (format t “Enter two items”)
    (setf x (read)) ; set local x to first thing read
    (setf y (read)) ; set local y to second thing read
    (list x y))     ; returns a list of the two values
```

Possible complication

- Local vars can be initialized by function calls, but no guarantee what order they’ll run in, e.g.

```
(let  ((a (read)) (b (read)))
       (format t “~A ~A~%” a b))
```

- Suppose user enters 10 and 20,
- there is no way of predicting which winds up in a and which winds up in b

let* guarantees order of evaluation

- let* acts like let, except initializes local vars in order given

```
(let* ((a (read)) (b (read)))
       (format t “~A ~A~%” a b))
```

- Guaranteed to read into a first, b second
- Useful when you want to initialize one local variable from another, e.g.

```
(let* ((x (read)) (b (if (numberp x) (* x x) 0)))
....
```