**Program Control in LISP**

🡪 Predicates are one way to make tests in programs and to take different actions based on outcome of the test. However, to make the use of predicates, we need some constructs to permit branching. Lisp branching is also based on function evaluation: control functions evaluation tests, and depending on the results, selectively evaluate alternative forms.

Syntax: (cond ( <condition 1> <action 1> )

( cond ( <condition 2> <action 2> )

…………………………………..

( cond ( <condition n> <action n> )

Conditions and actions may be arbitrary S-expression; each pair is enclosed in the parenthesis. Like defun, cond does not evaluate all of its arguments. Instead, it evaluates the condition in order until one of them returns a non-nil value. When this occurs, it evaluates the associated actions and returns this result as the value of cond expression. None of the other actions and none of the subsequent conditions are evaluated. If all the conditions evaluated to nil, cond returns nil.

Each ( <condition i> <action i> ), I = 1,2,3,….n, is called clause.

Each clause consists of a test proportion and an action or result portion. The first clause following the cond is executed by evaluating <test 1>. If this evaluates to non-nil, the <action 1> portion is evaluated, its value is returned and the remaining clause are skipped over. If <test 1> evaluates to nil, control passes to the second clause without evaluation <action 1> and the process is repeated.

If all test evaluates to nil, cond returns nil.

Eg: (defun result(a)) (cond((<ao)’(less than zero))(>a0)’(greater than zero)))

An alternative definition of result is:

(defun result(a) (cond ((<a0)’(less than zero))(t a )))

The version note that the second condition (>x0) is always true if the first is false. The ‘t’ atom in the final condition of the cond statement is a list atom that roughly corresponds to true. By convention, ‘t’ always evaluates to itself; this cause the last action to be evaluated if all preceding condition return nil. This construct is extremely useful, as it provides a way of giving the cond statement a default action that is evaluated if and only if all the preceding condition fails.

LISP also provides a form of if……then……else condition.

Syntax:

(if test <then\_action> <else\_action>)

For this form, test is first evaluated. If it evaluates to non-nil, the <then\_action> is evaluated and the result is returned; otherwise <else\_action> is evaluated and its value returned. The <else\_action> is optional. If omitted, then where test evaluates to nil, the if function returns nil.

Eg: (if(<5 0) ‘ (five) ‘ (zero))

Eg: (defun sum(b) (if(>b1) (prong (setq a (+ b 1 )) (setq c ( + a 2 ))) ( setq b 4 )))

1. Define a function using cond operator that take two inputs as argument and return greater among two numbers.
2. Define a function using if clause that take two inputs as arguments and return difference among them.