LISP NOTES #2

Links

Free Microsoft Software -
http://www.microsoft.com/education/facultyconnection/software/softwarelist.aspx?c1=en-us&c2=0

Defining Functions

Syntax: (**DEFUN** *function_name list_of_parameters body_block1 ... body_block_last*)
DEFUN is a special function, it associates the function definition with the symbol *function_name*
The defined function is used just like any other LISP function, the actual parameters are plugged into the formal parameters listed in list_of_parameters, then the blocks *body_block1 ... body_block_last* are evaluated and the result of *body_block_last* is returned from the function
Examples:

```
(defun addthree (x) (+ x 3))
ADDTHREE

(addthree 5)
8
(addthree (* 4 (1- 7)))
27
```

Function wrappers – user defined functions that modify the appearance of other existing functions, ex. accepting a different order of parameters

```
(defun xcons (ls hd) (cons hd ls))
```

New functions can be used inside (composed) other functions

```
(defun listoflists (x) (list (list (car x)) (list (card x))))
```

Variables declared in the list of function parameters are “bound” and any changes to their value are effective within the function body i.e. they have a locals scope.
Variables not declared in the list of function parameters are “free” and any changes to their value are effective even outside the function body, under normal circumstances (no apply or funcall function invocations in effect) they are treated as global variables, i.e. they have a global scope.
NOTE: Free variables may have a dynamic scope if apply or funcall function invocations are in effect. In dynamic scoping the free variable refers to the most recent variable created with the same name. This is discouraged and more about this later.

TIP: Free variables should be done with big care – avoid or know what is going on. Use free variables to communicate between functions ex. for counters and timers etc. It is a good practice to put * before and after free symbols that are meant to be global, ex. *my_global_var*

Logical values in LISP

Nil represents false, any non-nil value represents TRUE. For convienience atom t is done as a special symbol – its value cannot be changed, it is used to clearly represent TRUE and it evaluates to t (self). Likewise atom nil is special its value cannot be changed, it is used to clearly represent FALSE and it evaluates to nil (self).

LISP Predicates

Functions that test for some properties of objects and answer FALSE (i.e. nil) or TRUE (i.e. non-nil, usually t). Predicates have a letter p at the end of their name.

(atom x) = test if x is an atom
(listp x) = test if x is a list
(null x) = test if x is nil
(consp x) = test if x is a cons cell
(equal x y) = test if x and y are equivalent
(eq x y) = test if x and y point to the same object
(eql x y) = test if x and y point to the same object or equivalent atoms
(number x) = test if x is a number
(zerop x) = test if x is 0
(oddp x) = test if x is an odd number
(evenp x) = test if x is an even number
(typep x ‘number) = test if x is of type ‘number, the other types are ‘list, ‘atom, ‘symbol etc.
(member el ls) = test if element el is in the list ls, return the sublist of ls starting at the occurrence of el
(< x y) = test if x < y
(> x y) = test if x > y

(member ‘b ‘(a b c))
(b c)
(member ‘b ‘(a (b) c))
Nil

Custom-defined predicate to test id the first element of a list is an atom:

(defun car_atomp (x) (atom (car x)))
CAR_ATOMP
(car_atomp ‘(a b c))
T
(car_atomp ’a)
ERROR, we will protect against this later

Conditional Execution

IF – simple if … then … else …
(if test_block block_if_true block_if_false)

Ex. define a function to add an element to a list if it is not there:

(defun addjoin (el ls) (if (member el ls) ls (cons el ls))
ADDJOIN

COND – any required (one and up) conditional blocks, the processing exits when the first positive block is found, otherwise returns nil

```
(cond
        (cond1 block11 … block1n)
        …
        (condm blockm1 … blockmk)
)
```

Ex. define ADDJOIN using COND, if the second parameter is an empty list then suppress appending

```
(defun addjoin (el ls)
        (cond
                ((member el ls) l)
                ((null ls) ls)
                (t (cons el ls))
        )
)
ADDJOIN
```

Logical operators

Functions that perform logical operations on their arguments.

(not x) = negate x

(and c1 … cn) = test if all (c1 and … and cn) conditions are TRUE, the evaluation is "lazy" i.e. the if a FALSE condition is encountered then the operator stops (no point going further) and answers FALSE (i.e. nil), else return what cn evaluates to.

(or c1 … cn) = test if any (c1 or … or cn) conditions are TRUE, the evaluation is "lazy" i.e. the if a TRUE condition is encountered then the operator stops (no point going further) and answers TRUE (i.e. t) , else return what cn evaluates to.

Ex. Define a predicate for "nice number" that are between 4 and 11

```
(defun nicep (x) (and (< 4 x) (< x 11)))
NICEP
```

Ex. Define a predicate for testing for a number or nil

```
(defun numnilp (x) (or (null x) (numberp x)))
NUMNILP
```

TRICK: the following are equivalend due to the "lazy" nature or AND

```
(cond ((listp lls) (number (car ls))))   =   (and (listp lls) (number (car ls)))
```

Iteration and Recursion

Iteration should not be used for functional programming and thus is not permitted in this course. You must use recursion perform processing equivalent to iteration.

Recursion occurs when a function calls (directly or indirectly) itself from within its body.

Three elements of successful recursive solution:

1) Recursive Relation i.e. explain why and how the function needs to invoke itself
2) Grounding Condition i.e. when the function should stop invoking itself
3) Progress towards the goal i.e. make sure that if the function invokes itself then the parameter passed to the recursive invocation is closer to the grounding condition i.e. the function invokes itself with a smaller and smaller argument if the grounding condition is an empty list.

Example:

```
(defun recursive-length (ls)
  (if (null ls)
    0
    (1+ (recursive-length (cdr ls)))
  )
)

(recursive-length '(a b c))
```

➔ 3

Example:

```
(defun recursive-member (el ls)
  (cond
    ((null ls) nil)
    ((equal el (car ls)) ls)
    (t (recursive-member el (cdr ls)))
  )
)

(recursive-member 'a '(b c a d))
```

➔ (a d)

Example:

```
(defun recursive-member (el ls)
  (and
    ls
    (or
      (equal el (car ls))
      (recursive-member el (cdr ls))
    )
  )
)

(recursive-member 'a '(b c a d))
```

➔ T

Example:

```
(defun factorial (n)
  (cond
    ((zerop n) 1)
    (t (* n (factorial (1- n))))
  )
)
```

➔ (factorial 500)

```
1220136825991110068701238785423046926253574342803192842192413588385845373153881
9760549644750220328186301361647714820358416337872207817720048078520515932928547
7907571939330603772960859086270429174547882424912726344305670173270769461062802
3104526442188787894657547771498634943677810376442740338273653974713864778784954
3848959553753799042324106127132698432774571554630997720278101456108118837370953
1016356324432987029563896628911658974769572087926928871281780070265174507768410
7196243903943225364226052349458501299185715012487069615681416253590566934238130
0885624924689156412677565448188650659384795177536089400574523894033579847636394
4905313062323749066445048824665075946735862074637925184200459369692981022263971
9525971945217823331756934581508552332820762820023402626907898342451712006207714
6409794561161276291459512372299133401695523638509428855920187274337951730145863
5757082835578015873543276888868012039988238470215146760544540766353598417443048
0128938313896881639487469658817504506926365338175055478128640000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000
```

Note: Be careful to always catch the grounding condition. Does (factorial 10.5) work?

Types of Recursion

- **Single Recursion** – one effective occurrence of self-calling within a function body; can be disguised i.e. self-calling may appear in multiple places within the function body but only one recursive call can happen along any branch of the code; at most linear complexity for functions that make progress towards the grounding condition with regards to the size of the argument
- **Tail Recursion** – a special case of single recursion; the self-call is the last statement within the function body; compilers are often equipped to recognize it and it is converted to a loop at compile time hence very efficient
- **Multiple Recursion** – the function may call itself several times within its body; can be very expensive computationally ex. exponential if the function calls itself twice

NOTE: recursion adds some overhead to “save” local variables since new set is needed in the recursive call.

Example (disguised single recursion)

```
(defun recursive-substitution (in out struct)
  (cond
    ((equal out struct) in)
    ((atom struct) struct)
    (t (cons
      (recursive-substitution in out (car struct))
      (recursive-substitution in out (cdr struct)))
    )
  )
)
```

(recursive-substitution '(a b) '(c d) '(a b (c d (c d))))
➔ (a b (c d (a b)))

Example (exponential complexity multiple recursion)

```
(defun fibonacci (n)
  (cond
    ((or (equal n 1) (equal n 2)) 1)
    (t (+ (fibonacci (- n 1)) (fibonacci (- n 2))))
  )
)
```

NOTE: simple and elegant but expensive, try (fibonacci 100) !!!

Accumulators and Auxiliary Functions

Sometimes it is possible to keep some partial results in a variable and pass it to the recursive call only to amend it, then output these partial results at the grounding case level since at this point the entire argument is considered processed; think of it as an **accumulator** for your result that can be output once there is nothing more to do.

Example

```
(defun reverse-in-out (todo done)
  (cond
    ((null todo) done)   ; nothing more to do – just output what we have so far
    (t (reverse-in-out (cdr todo) (cons (car todo) done)))   ; add to partial results and reverse a smaller list
  )
)

(reverse-in-out '(a b c d) '())
```
➔ (d c b a)

Now, do we need to concern the end-user of the function that there is some empty initial argument that needs to be passed? No, we can hide this fact – we will define a "proper" reverse function that will call reverse-in-out. The reverse-in-out function is an **auxiliary** function for the reverse function.

```
(defun reverse-easy (todo)
  (reverse-in-out todo nil))

(reverse-easy '(a b c d))
```
➔ (d c b a)