LISP NOTES #4

NOTE: Please refer to the slides posted on the course Web site regarding: **Symbolic Computing, Symbolic Programming, Operational Programming, Denotational Programming, Denotation, Prescription, Description**

Optional Parameters

Last parameters to a function can be defined as optional using &optional keyword and if they are not provided during invocation then they get NIL, for example

```
(defun opt-par-v1 (par1 &optional par2) (list par1 par2))
    ➔ OPT-PAR-V1
(opt-par-v1 3 4)
    ➔ (3 4)
(opt-par-v1 3)
    ➔ (3 nil)
```

Last parameters to a function can be defined as optional using &optional keyword and given a default value, if they are not provided during invocation then they get the default value, for example

```
(defun opt-par (par1 &optional (par2 55)) (list par1 par2))
    ➔ OPT-PAR-V2
(opt-par-v2 3 4)
    ➔ (3 4)
(opt-par-v2 3)
    ➔ (3 55)
```

Flexible Number of Parameters
A function can be defined to take any number of parameters after some required parameters (although these could be none) using &rest keyword; all parameters that fall after the &rest keyword during invocation are collected as a list and given to the function as one parameter that is a list.

Example

```
(defun sth (par1 &rest other-args) (* par1 (apply '+ other-args)))
    ➔ STH
(sth 3)
```

➔ 0

(sth 3 4)

➔ 12

(sth 3 4 5 6)

➔ 45

Macros

Functions that produce the code text

Macro evaluation:

- From command line - macro is expanded (executed) and the output is evaluated
- In DEFUN - macro is expanded (executed) and the output remains as text inside the function definition

Advantages:

Efficiency – it is not what you type that executes but the text that the macro expands to

Parameters are nor evaluated upon expansion so if the produce code does not evaluate them then they are not evaluate, this way you can make special functions like SETQ

Example

(defmacro get-radius (ls) (list ‘caddr ls))

➔ GET-RADIUS

Now when you type (get-radius ‘(43 51 12)) **you really type what the macro expands to** (the text that it produces (caddr ‘(43 51 12))

Compiling

If you compile your function, you still use it the same way it just runs much faster from that point, although debugging becomes harder.

(compile ‘myfunction) = compiles function myfunction

Matrix Representation & Manipulation

A matrix can be represented as a list of its rows

A B C
D E F
G H J

Or in LISP

((A B C) (D E F) (G H J))

Now to transpose a matrix we need to make the columns to be the rows

A D G
B E H
C F J

Or in LISP

((A D G) (B E H) (C F J))

Here is a recursive function to do the matrix transposition

```
(defun trans ( theMatrix )
        (cond ( ( null ( car theMatrix ) ) nil )
        ( t ( cons ( firstOfEach theMatrix )
        ( trans (restOfEach theMatrix ))))
))

(defun firstOfEach ( theMatrix ) ; Extract first of each row
        (cond ( ( null theMatrix ) nil )
        ( t (cons ( caar theMatrix )
        ( firstOfEach ( cdr theMatrix ))))
        ))

(defun restOfEach ( theMatrix ) ; remove first of each row
        (cond ( ( null theMatrix ) nil )
        ( t ( cons ( cdar theMatrix )
        ( restOfEach ( cdr theMatrix ))))
))
```

Here is a functional program that does the same

```
(defun trans ( theMatrix )
        (apply 'mapcar 'list theMatrix ))
```

Functional Programming

What is functional programming? What are the prime attributes of functional programs?

Functional programming consists of writing functions that have functions as input and frequently as output. That is writing functions that themselves create new functions. Use of generalized functions that abstract control flow patterns -- e.g. mapcar and reduce.

Functional programs have no explicit loops (recursion), have no sequencing at a low level, have no local variables. Frequently input is a single list of parameters.

Characteristics of functional programming:

- Meaningful Units of Work – computations are divided into meaningful units of work

  Work with operations is meaningful to the application, and not to the underlying hardware & software:

  » Analogy with word processing is not to work with characters and arrays or lists of characters

  » But work with words, paragraphs, sections, chapters and even books at a time, as appropriate.

- Requires Abstraction – requires to think using concepts and about what needs to be done and not how it is done

  Abstract out the control flow patterns and give them names to easily reuse the control pattern

  » For example in most languages we explicitly write a loop every time we want to process an array of data

  » If we abstract out the control pattern, we can think of processing the entire array as a single operation

  » The abstracted operations do not have: explicit (loops or recursion), no sequencing at low level, no local variables

  » Some flow of control abstractions are:

  (α op) = apply operation op to all elements concurrently ;
  a list (a b c) would become ((op a) (op b) (op c))
  (/ op) = reduce a list using operation op to a single element;
  a list (a b c d) would become (op a (op b (op c d)))
  op1 ° op2 = compose two unary functions;
  an argument x becomes (op1 (op2 x))
  bu op const = bind the first argument of a function op to const; this reduces the arity of a function so a binary function (i.e. that takes 2 arguments) becomes unary (i.e. that takes 1 parameter); an argument x becomes (op const x); for example a binary function cons can be mapped on a single lists (otherwise it can be mapped only on a pair of lists) i.e.
  (mapcar (bu 'cons 7) ' (1 2 3)) → ((7.1) (7.2) (7.3))

  » The abstracted functions are usually collected into functional libraries and the libraries are assembled to solve some application specific computations

NOTE: Please look at the functionals.lsp and the document "Functional Programming" (provided on the course Web Site) to see some major functionals, how they are defined, and what they do, especially: **bu, rev, comp, compl, trans, distr, distl, range, filter, sigma, genlist**

Two Vector Inner-Product Example

Having 2 vectors ((A1 A2 … An) (B1 B2 … Bn))
produce the inner product (A1*B1+A2*B2+ … + An*Bn)

Here is a recursive function to do the inner product:

```
(defun innerProduct ( a-b-pair )
  ( cond ( ( null ( car a-b-pair ) ) 0 )
    ( t ( + ( * ( caar a-b-pair ) ( caadr a-b-pair ) )
    ( innerProduct ( list ( cdar a-b-pair)
    ( cdadr a-b-pair) ) ) ) )
))
```

Here is a functional program that does the same:

```
(defun innerProduct ( a-b-pair )
   (reduce ‘+ (mapcar ‘* (car a-b-pair) (cadr a-b-pair))))
```