F.G.H.
6/12/85

LispCourse #36: More on Variable Binding; Control Structures in Lisp

NLAMBDA Functions

Recall that NLAMBDA functions are treated specially by the Lisp interpreter.

In particular, when evaluating a function call containing an NLAMBDA function, the interpreter does ***not*** evaluate the arguments before the function is applied.

To define an NLAMBDA function, just use the keyword NLAMBDA instead of LAMBDA in the function definition.

For example:

```
1_ (DEFINEQ
        (E  (NLAMBDA (FileName)
                    (* * Call TEdit on the named file)
                    (TEDIT FileName))))
(E)
2_ (E <LISPCOURSE>OUTLINE01.TED)
{PROCESS}#32,12345  (Starts up a TEdit)
3_ (SETQQ File  <LISPCOURSE>OUTLINE01.TED)
<LISPCOURSE>OUTLINE01.TED
4_ (E File)
{DSK}File not found
```

Contrast the preceeding NLAMBDA with an analogous LAMBDA function:

```
5_ (DEFINEQ
        (EX  (LAMBDA (FileName)
                    (* * Call TEdit on the named file)
                    (TEDIT FileName))))
(EX)
6_ (EX '<LISPCOURSE>OUTLINE01.TED)
{PROCESS}#32,12337  (Starts up a TEdit)
7_ (SETQQ File <LISPCOURSE>OUTLINE01.TED)
<LISPCOURSE>OUTLINE01.TED
```

8_ (EX File)
{PROCESS}#32,12678 (Starts up a TEdit)

The NLAMBDA version of this function has the *advantage* that user/programmer does not have to QUOTE the file name argument when calling the function since the argument is not evaluated.

I.e., compare events 2 and 6 in the preceding examples.

However, the NLAMBDA version has the *disadvantage* that it is less easily integrated into the workings of the Lisp Exec.

For example, you cannot set a variable to the name of the desired file and then use this variable to refer to the file when calling the NLAMBDA function. This is shown in the comparison between events 4 and 8 above.

The bottom line is that NLAMBDA function have their uses, especially when writing functions that interface with the user, but they also have their drawbacks in terms of flexibility.

For example: LISTFILES is an NLAMBDA function.

It is nice to be able to type: (LISTFILES {DSK}HOMEWORK34) without worrying about the QUOTE.

On the other hand, you can do: *(FOR File in ListOfFiles DO (LISTFILES File))* to print out each file in a list of files.

You would have to type *(FOR File in ListOfFiles DO (APPLY 'LISTFILES (LIST File)))*, which is more than a bit clumsy.

The other place that NLAMBDA function are very handy is when you want to write functions that evaluate a bunch of SExpressions in a non-standard order.

For example, the following function evaluates the first of two SExpressions only if the first evaluates to non-NIL:

```
(DEFINEQ
    (FirstOnlyIfSecond
        (NLAMBDA (SExpr1 SExpr2)
```

(COND ((EVAL SExpr2) (EVAL SExpr1))))))

Note that COND, AND, OR, PROG, etc. are all standard Interlisp functions that use this trick to change the order in which a set of SExpressions are evaluated.

Spread and NoSpread Functions

The ***Spread/NoSpread*** distinction is a second classification of functions in Interlisp that is orthogonal to the LAMBDA/NLAMBDA distinction.

So far, we have been looking only at ***spread*** functions.

A ***spread*** function is one whose parameter list is in fact a LISTP.

Thus, a spread function definition has the form **(LAMBDA *List SExpr1 ... SExprN*)** or the form **(NLAMBDA *List SExpr1 ... SExprN*)**.

When APPLYing spread functions, the Interlisp interpreter matches each parameter in this parameter list with the corresponding argument in the function call.

Note that the effect of this scheme is that each spread function has a fixed number of parameters. If there are more arguments then parameters, the arguments are just ignored.

In contrast, a ***nospread*** function has a parameter "list" consisting of a single LITATOM.

Thus, a nospread function definition has the form **(LAMBDA *Litatom SExpr1 ... SExprN*)** or the form **(NLAMBDA *Litatom SExpr1 ... SExprN*)**.

When APPLYing a nospread function, Interlisp sets up this LITATOM to point to the list of all arguments in the function call. Using the LITATOM as a reference (as described below), you can access this list from within the function.

This scheme allows a single parameter to refer to an arbitrarily long list of arguments. Thus, nospread functions have no fixed number of parameters.

The manner in which one can reference the arguments of a nospread function differs for LAMBDA and NLAMBDA functions.

NLAMBDA nospread functions

Interlisp simply sets the LITATOM that is the parameter "list" to the list of arguments (unevaluated). You can then get to any element of the this list using CAR, CDR, LAST, etc.

For example, the following function calls TEdit for each of an arbitrary number of files:

```
10_(DEFINEQ
        (TEDs
                (NLAMBDA Files
                        (FOR File in Files DO (TEDIT File)))))
(TEDs)
11_(TEDs {DSK}FOO)
NIL (Starts TEdit on a single file {DSK}FOO)
12_ (TEDs {DSK}BAR {ERIS}<HALASZ>BAZ {DSK}ARG)
NIL (Starts 3 TEdits on three files as specified)
13_ (TEDs A B C D E F G H J K)
NIL (Starts 10 TEdits on ten files as specified)
```

LAMBDA nospread functions

LAMBDA nospread functions are a bit more complicated.

The LITATOM that is the parameter "list" is bound to the number of arguments being passed in the current function call.

The arguments can be accessed individually using the functions ***ARG*** and ***SETARG***.

(ARG *Litatom M*) ž In a LAMBDA nospread function whose parameter specification is the LITATOM *Litatom*, ***ARG*** returns the *M*th argument of the current function call. ARG is itself an NLAMBDA function that doesn't evaluate its argument but DOES evaluate its second argument.

(SETARG *Litatom M Value*) ž In a LAMBDA nospread function whose parameter specification is the LITATOM *Litatom*, ***SETARG*** sets the *M*th argument of the current function call to *Value*. SETARG is itself an NLAMBDA function that doesn't evaluate its argument but DOES evaluate its second and third arguments.

For example, the following (nonsense) function collects the results of evaluating an arbitrary number of SExprs, where each result is made into a string:

```
14_(DEFINEQ
      (ResultsAsStrings
            (LAMBDA SExprs
                  (FOR Index FROM 1 to SExprs
                        COLLECT
                              (MKSTRING (ARG SExprs
                              Index))))))
```

(ResultsAsStrings)

```
15_ (ResultsAsStrings (PLUS 1 2) 'ABC (REVERSE '(A B C)))
```

("3" "ABC" "(C B A)")

16_ (ResultsAsStrings (PLUS 2 3) '(PLUS 2 3) (EVAL (PLUS 2 3)))

("5" "(PLUS 2 3)" "5")

What are nospread functions good for?

Nospread functions are handy whenever you want to write a function that handles an arbitrary number of arguments.

COND, PROG, AND, OR, and PLUS are all implemented as nospread functions since they all handle an arbitrary number of arguments.

Another example might be the print function, LC.Print, from Homework #35.

LC.Print was defined as a spread function as follows:

```
(DEFINEQ (LC.Print
    (LAMBDA (#Items Thing1 Thing2 Thing3 Thing4
    Thing5)
        (FOR Thing
            IN (LIST Thing1 Thing2 Thing3 Thing4
            Thing5)
            AS Ctr FROM 1 TO #Items
            DO (PRIN1 Thing))

        (TERPRI))))
```

This version of LC.Print could handle at most 5 things to be printed and required a sixth argument specifying how many actual arguments there were if there were less than 5.

But LC.Print should have been defined as a nospread function as follows:

```
(DEFINEQ
    (LC.Print (LAMBDA Things
        (FOR Index FROM 1 TO Things
```

```
                                    DO (PRIN1 (ARG Things Index)))
                            (TERPRI))))
```

Note that this nospread version of LC.Print can print an arbitrary number of things and does not require an explicit parameter specifying how many arguments there are.

Summary of 4 function types

Since Spread/Nospread and LAMBDA/NLAMBDA are orthogonol distinctions, there are a total of 4 different kinds of functions in Interlisp:

LAMBDA spread

LAMBDA nospread

NLAMBDA spread

NLAMBDA nospread

You can find out the type of an arbitrary function in Interlisp using the *?=* facility in the Lisp Exec.

In the Lisp Exec, just type a *"(",* the *function name,* a *space,* the characters *"?="* and a *carriage return.*

Once the return is typed, the ?= will be erased and on the next line there will be a print out the parameter "list" of the named function.

Following the parameter list will be a indication of the type of the function as follows:

LAMBDA spread -- Blank

LAMBDA nospread -- *{L*}*

NLAMBDA spread -- *{NL}*

NLAMBDA nospread -- *{NL*}*

Examples:

LAMBDA spread

LAMBDA nospread

NLAMBDA spread

NLAMBDA nospread

Functions as Arguments in Function Calls

In Lisp, using a function as an argument in a function call is no big deal. Functional arguments follow the same rules as any other argument.

Example:

```
1_(DEFINEQ
        (DoItToFiveAndFour
                (LAMBDA (ItFn)
                        (APPLY ItFn (LIST 5 4)))))
```

(DoItToFiveAndFour)

2_(DoItToFiveAndFour ’PLUS)

9

3_(DoItToFiveAndFour ’DIFFERENCE)

1

4_(DoItToFiveAndFour (MKATOM (CONCAT "TI" ’MES)))

20

*5_* (DoItToFiveAndFour (LIST ’PLUS))

UDEFINED FUNCTION

(PLUS)

Note: you can generally use a function definition (i.e., a list beginning with a LAMBDA) where a function name is required. This is because most Lisp functions that require a function name, will accept a function definition instead.

Example:

6_ (DoItToFiveAndFour ’(LAMBDA (X Y) (SUB1 (PLUS X Y))))

8

7_ (DoItToFiveAndFour ’(LAMBDA (X Y) (TIMES (PLUS X Y) Y)))

*36 [=(5+4)*4]*

When passing functions as arguments, it is important to pass down function with the appropriate characteristics, e.g., the required number of parameters.

Example:

8_ (DoItToFiveAndFour ’(LAMBDA (X Y Z) (PLUS X Z Y))))

NON-NUMERIC ARG

NIL

[Because Z is bound to NIL when (PLUS X Y Z) is evaluated.]

FUNCTION ž in the preceeding examples, I used QUOTE to prevent evaluation of function names or function definitions used as arguments in a function call.

I actually should have used the function FUNCTION in place of QUOTE in these cases.

FUNCTION behaves much like QUOTE, but tells Lisp that it is dealing with a function rather than a data object. In some cases (e.g., in the compiler) this will allow Lisp to behave much more efficiently.

Example:

```
9_ (DoItToFiveAndFour (FUNCTION (LAMBDA (X Y) (SUB1 (PLUS
X Y)))))
8
10_(DoItToFiveAndFour (FUNCTION PLUS))
9
```

A more realistic example

```
1_ (DEFINEQ
    (WindowOp (LAMBDA (Op Window)
        (OR (WINDOWP Window)(SETQ Window (GetWindow)))
        (APPLY* Op Window)))
(WindowOp)
2_ (WindowOp (FUNCTION CLOSEW) W)
CLOSED [Closes the window that is the value of W]
3_ (WindowOp (FUNCTION MOVEW))
(100 . 100) [Gets a window from the user, then moves that window]
```

Control Structures in Lisp

Procedural Abstraction

If there's one important concept in programming, its ***abstraction***.

Abstraction starts with the process of breaking down a large task into a set of sub-tasks, breaking down each sub-task into a set of sub-sub-tasks, and so on until you get to atomic tasks that can't be further decomposed.

You then write the primitive functions to carry out the lowest-level, most detailed sub-sub-... tasks. Once these functions are written, you can write the functions that carry out the next higher level in terms of these more primitive, lower level functions. And so on, until the function that accomplishes the top-level task can be written in terms of the functions that carry out its compoenent sub-tasks.

The important concept here is that the functions at each level should be as independent as possible from both the lower level functions it uses and the higher level function that use it.

> The only assumptions that should be made are about these higher or lower level functions concern the syntax and semantics of their arguments and returned values, and some information about their side-effects on the environment.
>
> In particular, NO assumptions should be made about the details of how they are implemented.

The goal of abstraction is to be able to write programs in which bugs, required changes, etc. will be isolated to only those few functions that are directly relevent to the bug or change or whatever.

> For example:
>
> > Consider a program that does some complex computations, printing out intermediate results in the Exec window along the way.
> >
> > The task of printing out the intermediate results on the screen should be isolated into a separate function (or functions) and not be part of the functions that do the calculations.
> >
> > Then, if you want to change the printout to use special fonts or to go to a file rather than the screen, you have to change only the printing function(s). No changes will have to made to all of the functions that do the computational work.

The advantages of properly abstracted programs are too many to enumerate!!! Ease of programming, ease of debugging, ease of modification, and so on.

We covered ***data abstraction,*** i.e. abstraction when dealing with data structures, in glorious detail earlier (see LispCourse #s 24 thru 26).

Procedural abstraction is the analogous concept in the realm of writing procedures to carry out arbitrary tasks.

Procedural abstraction is the art of writing functions by *combining* calls to other, more detailed functions.

There are various schemes for doing this *combination* of function calls to make interesting and useful functions in Lisp.

The various schemes are called *control structures* and are the topic of the next section.

Interlisp Control Structures

A *control structure* is a scheme for controlling the order of a set of function calls being made in order to accomplish some task.

In Lisp, there are a number of basic control structures available for use in writing your functions.

These control structures are generally implemented as functions that take an arbitrary number of SExpressions (i.e., function calls) and evaluate them in some specific order.

Review: Control Structures Already Covered

The following are basic control structures that we've already covered in previous LispCourses.

Sequential evaluation

As a default rule, when you can specify an arbitrary number of SExprerssions for evaluation, these

SExpressions are evaluated in sequential order and the value of the last SExpression is returned as the value of the whole set of SExpressions.

The most salient instance of this is in the body of a function definition.

Another instance is after the DO or COLLECT in a FOR loop.

Embedding

Perhaps, the most prevelant control structure in Lisp is embedding function calls as arguments to other (LAMBDA) function calls.

Example:

```
(SQRT
    (PLUS
        (SQUARE (DIFFERENCE
        X1 X2))
        (SQUARE (DIFFERENCE
        Y1 Y2))))
```

For LAMBDA function calls, the Lisp evaluator insures that the arguments will be evaluated in sequential order (from left to right) before the function is applied.

In the precedding example this means that before the SQRT function is applied, the PLUS is evaluated. But before the PLUS is applied, the two SQUARE calls are evaluated in order. And so on.

Note: NLAMBDA functions to not have this property.

If the arguments to an NLAMBDA function are evaluated at all, they are evaluated by the function itself and not by the Lisp evaluator.

Since the function can evaluate its arguments in any order, the order of evaluation of an NLAMBDA function's arguments is impossible to predict, *a priori.*

However, if you happen to know how an NLAMBDA function evaluates it arguments, you can use embedding.

Example:

(SETQ FOO (SETQ BAR (PLUS A B))) Sets both FOO and BAR to the result of (PLUS A B).

COND

COND implements a conditional (IF-THEN-ELSE) control structure in Lisp. (See LispCourse #4, page 5)

COND has the form:

(COND (*Test1 Consequent1*)(*Test2 Consequent2*) ...)

COND evaluates each *TestI* in turn until the first one that evaluates to non-NIL.

It then evaluates each SExpression in the *Consequent* after this first non-NIL *Test* and returns the value of the last of these SExpressions. (Or the value of the *Test* if there are no SExpressions in the *Consequent*.)

If there is no *TestI* that evaluates to non-NIL, COND returns NIL.

Example:

```
(COND
    ((LITATOM SExpr)(LookupValue SExpr Stack))
```

((LISTP SExpr)(LC.Eval SExpr Stack))
(T SExpr))

In English:

> ***If*** SExpr is a LITATOM, Lookup its value;
>
> **Else if** SExpr is a list, the eval that list,
>
> **Otherwise**, return SExpr.

AND, OR

AND and OR are two functions that are meant to be logical functions, but often serve to control the order of evaluation of SExpressions.

Both AND and OR take an arbitrary number of SExpressions as arguments.

AND ž evaluates the SExpressions in sequential order but stops at the first SExpression that evaluates to NIL and returns NIL. If no such SExpression is found, AND returns the value of the last SExpression.

OR ž evaluates the SExpressions in sequential order but stops at the first SExpression that evaluates to non-NIL and returns its value. If no such SExpression is found, OR returns NIL.

AND is often used to replace single clause COND expressions:

1. *(AND YesFlg (SETQQ Blat Fumble))*

   is equivalent to

   (COND (YesFlg (SETQQ Blat Fumble))).

2. *(AND (SETQ Baz (CAR Fu)) (PLUS 3 Baz))*

   is equivalent to

   (COND ((SETQ Baz (CAR Fu)) (PLUS 3 Baz))).

3. *(AND Window TextStream ID*
   (LC.SetID ID Window TextStream))

   is equivalent to

   (COND
   ((AND ID Window TextStream)

```
                    (LC.SetID ID Window
                        TextStream))))
```

OR is often used to replace COND statements with a number of clauses, each having a test and no consequents:

1. *(OR*
   (WINDOWP Window)
   (SETQ Window (CREATEW)))

   is equivalent to

   ```
   (COND
         ((WINDOWP Window))
         ((SETQ Window (CREATEW))))
   ```

2. *(OR List Array*

   (ERROR "Neither list nor array present"))

   is equivalent to

   ```
   (COND
         (List)
         (Array)
         ((ERROR ".."))
   ```

3. *(SETQ Foo*
   (OR Arg1 Arg2 (SETQ Arg1 (SETQ Arg2 Arg3))))

   is equivalent to

   ```
   (SETQ Foo
         (COND
               (Arg1)
               (Arg2)
               ((SETQ Arg1 (SETQ Arg2
                     Arg3))))
   ```

Iteration: FOR, WHILE, and UNTIL

FOR, WHILE, and UNTIL implement iterative loops in Interlisp that allow you to repeat a set of operations for each element in a sequence or list of elements.

The basic notion of an iterative loop is that there is an *iteration sequence* (e.g., a list or a sequence of integers), an *iterative variable*, and a *body of SExpressions* that (may) use the iterative variable.

> For each element in the iteration sequence in turn, the iteration loop binds the iterative variable to this element and then evaluates the SExpressions in the body.
>
> The clauses in a basic FOR, WHILE, or UNTIL statement specify the iteration sequence, the iteration variable, and the body.
>
> > See LispCourse #5, page 1 for a description of the basic FOR, WHILE and UNTIL loops.
>
> Additional clauses, can specify what to do with the results of evaluating the body (e.g., the COLLECT statement), under what conditions to terminate the iteration before the iteration sequence is exhausted, etc.
>
> In particular,
>
> > For conditional execution of the body of an iterative loop, see LispCourse #26, page 16 for a description of the WHEN clause in FOR/WHILE loops.
> >
> > For multiple iteration variables, see LispCourse Homework #28, page 6 for a description of the AS clause in FOR/WHILE loops.

Loaded example:

```
(FOR Item IN List
     AS Index FROM 1 TO (LENGTH List) BY
     1
     WHILE (KEYDOWNP 'A)
     WHEN (NUMBERP (ELT Array Index))
     COLLECT (ADD1 (SETA Array Index
     Item)))
```

This example has 2 iterative sequences (*List* & *FROM 1 TO (LENGTH List) BY 1*) and, correspondingly, two iterative variables (*Item* & *Index*).

It has a standard body thgat uses both iteration variables -- *(ADD1 ...)*

It has an early termination condition -- *WHILE* ...

It has conditional evaluation of its body -- *WHEN* ...

And it returns the list of evaluation results from each iteration -- *COLLECT* ...

LET

LET is an implementation of the basic sequential evaluation of SExpressions. Thus as a control structure its not very interesting. Its basic use is for binding variables.

(SEE LispCourse #34, page 19)

PROG with RETURN

Like LET, PROG is used basically for binding variables.

(SEE LispCourse #34, page 23)

Basically, PROG implements a sequential evaluation of SExpressions.

However, with an embedded RETURN statement, you can terminate this sequential evaluation at any point and force PROG to return an arbitrary value.

PROG with RETURN together with COND, AND, OR, etc. can be used to built tailor-made control structures.

Example:

```
(PROG NIL
    (AND (WINDOWP Window)(RETURN
    Window))
    (AND
        (WINDOWP
            (SETQ Window
                (WindowOfText
                TextStream))
        (RETURN Window))
    (SETQ Window (CREATEW))
    (AND
        (WindowOffScreenP Window)
        (RETURN Window))
```

```
    ... Do lots of work with the window ...
    (RETURN Window))
```

This is an example of using PROG with no binding list just to take advantage of the RETURN statments.

The example is actually equivalent to the following COND statement. But the PROG version is much easier to follow that the COND version (at least for an old FORTRAN programmer!).

```
(COND
    ((WINDOWP Window) Window)
    ((WINDOWP
        (SETQ Window
            (WindowOfText
            TextStream)))
      Window)
    ((WindowOffScreenP
        (SETQ Window
        (CREATEW)))
      Window)
    (T  ... Do lots of work with the
    window ...
      Window))
```

PROG with GO

Note that PROG also supports a GO clause that can be used to construct tailor-made iterative loops.

The GO clause is like the FORTRAN GOTO.

Since GOTOs are considered BAD programming style, we won't cover GO here.

See pages 4.3 & 4.4 of the IRM for more information.

Note, however, that FOR/WHILE/UNTIL loops are actually constructed by CLISP from PROG/GO/RETURN!!!

Other Control Structures

SELECTQ

SELECTQ is a control structure that will select a sequence of SExpressions to evaluate based on the value of its first argument.

SELECTQ has the following format:

(SELECTQ *Selector Clause1 Clause2 ... DefaultSExpr*)

SELECTQ is an NLAMBDA function.

Selector is an arbitrary SExpression that evaluates to an atomic value.

DefaultSExpr is an arbitrary SExpression. It MUST be present. (Its a common mistake to forget it!!)

Each *ClauseI* is an SExpression of the format:

(*Key SExpr1 SExpr2 ...*)

Key is either a single atom or a list of atoms.

The *SExprI* are arbitrary SExpressions.

SELECTQ works as follows:

Selector is evaluated. And then compared against the *Key* (which is unevaluated) of each *Clause* in turn until a match is found.

A match is defined as follows:

If *Key* is an atom, then *Key* must be **EQ** to the value of *Selector*.

If *Key* is a list, then the value of *Selector* must be a MEMEBER of *Key*.

For the first *Clause* whose *Key* matches, each of the SExprs in the *Clause* are evaluated in turn, and the value of the SELECTQ is the result of the last evaluation.

If no matching clause is found, then *DefaultSExpr* is evaluated and the result is returned as the value of the SELECTQ.

Example:

```
1_(DEFINEQ
  (ProcessCommand
    (LAMBDA (Command)
      (SELECTQ Command
        (Create
          (CREATEW))
        (Close
          (SETQ Window
          (GetWindow))
          (CLOSEW Window)
```

```
            Window)
        (Move
            (SETQ Window
            (GetWindow))
            (MOVEW Window)
            Window)
        ((Shape Reshape)
            (SETQ Window
            (GetWindow))
            (SHAPEW Window)
            Window)
        (ERROR "Unknown Command"
        Command))))
```

(ProcessCommand)

3_ (ProcessCommand ’Create)

[Prompt for a region and create a window in it.]

{WINDOW}#43,12557

4_ (ProcessCommand ’Shape)

[Prompt for a window and reshape it.]

{WINDOW}#43,12345

4_ (ProcessCommand ’Display)

Unknown Command

Display

EFS: What is the format of the COND statement equivalent to SELECTQ.

PROG1, PROG2, PROGN

PROG1, PROG2, and PROGN are three variations on the theme of sequential evaluation.

All of these functions sequential evaluate an arbitrary number of SExpressions.

They differ in the value that they return. In particular:

PROG1 returns the value the **first** SExpression.

PROG2 returns the value of the **second** SExpression.

PROGN returns the value of the **last** SExpression.

Note: These are NOT variations of the PROG, they do NOT bind any variables.

PROG1 and PROG2 are used when need to evaluated a sequence of SExpressions in a particular order, but want to return the value of the first or second SExpression rather than the last as is usually the case.

Example from LC.Apply from Homework#35:

```
( LAMBDA
        ...
        (PROG1
                (LC.Eval (CAR (LAST SExprs))
                Stack)
                (LC.Unbind Stack)))
```

The problem here is that LC.Apply has to evaluate the last SExpr BEFORE unbinding the stack, but needs to return the value of the evaluatiuon AFTER unbinding the stack.

PROG1 solves the problem efficiently. In the Homework #35 solutions, we needed to bind an extra variable (i.e., Result) using LET in order to accomplish the same thing:

```
( LAMBDA
        ...
```

```
(LET (Result)
        ...
        (SETQ Result
                (LC.Eval (CAR (LAST
                SExprs)) Stack)
        (LC.Unbind Stack)
        Result))
```

PROGN is used where only a single SExpr is allowed, but you need to evaluate several SExpressions in order. In this case, you just wrap the SExpressions in a PROGN, which counts as a single SExpression.

Example:

```
(SELECTQ
        Command
                (Create (CREATEW) 'Created)
                (Close
                        (SETQ Window
                        (GetWindow))
                        (CLOSEW Window)
                        'Closed)
                (PROGN
                        (SETQ Window
                        (GetWindow))
                        (MOVEW Window)
                        'Moved))
```

The problem here is that the default clause of a SELECTQ must be a single SExpression. But we want to do three things in the default case: get the window, move it, and return the atom Moved. Solution is to use a PROGN to wrap the three function calls into a single unit.

Simple Repetition: RPT and RPTQ

RPT and RPTQ implement a simple repetition control structure.

(RPT *N SExpr*) ž Evaluates *SExpr N* times and returns the value of the last evaluation. Before each evaluation the free variable **RPTN** is set to the number of repetitions still to take place. RPTN can be used inside of *SExpr*.

RPT is a LAMBDA-spread function. Therefore, it is actually the value of the argument that is being repeatedly evaluated.

Example:

```
1_ (RPT 5  (QUOTE (PRINT RPTN)))
5
4
3
2
1

2_ (RPT 3 (QUOTE (CREATEW)))
[Creates 3 windows]
{WINDOW}#34,00123
```

(RPTQ *N SExpr1 SExpr2 ...*) ž Evaluates each *SExprI N* times and returns the value of the last evaluation. Order of evaluation is to do each *SExprI* once, then repeat. Before each evaluation the free variable **RPTN** is set to the number of repetitions still to take place. RPTN can be used inside of *SExpr*.

RPT is a NLAMBDA-nospread function.

Example:

```
3_ (SETQ A 6)
6
```

4_(RPTQ 6 (SETQ A (ADD1 A)) (SETQ A (SUB1 (SUB1 A))))

0

5_ A

0

6_(RPTQ 2 (PRINT "A")(PRINT "B"))

"A"

"B"

"A"

"B"

"B"

Mapping functions: MAP, MAPC, MAPLIST, MAPCAR

Mapping functions are a common Lisp control structure.

A mapping function iterates over some data structure (usually a list) and applies some other function to each element of the data structure in turn.

Interlisp has several built in mapping functions for lists. We will cover only MAPC and MAPCAR.

(MAPC *List WorkFn NextFn*) ž Does (*APPLY* WorkFn* (CAR *List*)), then does (*APPLY* WorkFn* (CAR (*NextFn List*))), then does (*APPLY* WorkFn* (CAR (*NextFn* (*NextFn List*)))) and so on until (*NextFn* (... (*NextFn List*))) returns NIL. MAPC always returns NIL.

If *NextFn* is NIL, CDR is used.

Examples:

(MAPC (OPENWINDOWS) 'CLOSEW) will CDR down the list of OPENWINDOWS and CLOSEW each window in turn.

(MAPC (OPENWINDOWS) 'CLOSEW 'CDDR) will map down every other window in the OPENWINDOWS list (due to the use of CDDR rather than CDR) and CLOSEW each of these windows in turn.

(MAPCAR *List WorkFn NextFn*) ž Essentially the same as MAPC, but returns a list of the values returned by all the evaluations of *WorkFn*.

Examples:

(MAPC (LIST 1 2 3 4 5 6 7) 'ADD1) will CDR down the list (1 2 3 4 5 6 7) and ADD1 each item in turn, returning the result of all the ADD1s, i.e., *(2 3 4 5 6 7 8)*.

(MAPC (LIST 1 2 3 4 5 6 7) 'ADD1 'CDDDR) will map down every third element the list (1 2 3 4 5 6 7) and ADD1 each of these elements in turn, returning the result of all the ADD1s, i.e., *(2 5 8)*.

Note: Section 5.3 of the IRM describes several relatives of MAPC and MAPCAR.

Note: MAPC and MAPCAR are nearly identical in functionality to various FOR/COLLECT constructions. (In fact, CLISP actually implements many FOR loops as MAPC/MAPCAR type functions) Since FOR is much easier to use, I seldom use MAPC and MAPCAR and their relatives.

But, the notion of a mapping function appears other places in Lisp, where functions like FOR are not available.

For example, TEdit has a function called TEDIT.MAPPIECES that allows you to apply an arbitrary function to every "piece" of text in a TEdit text.

Recursion -- The ultimate control structure in Lisp

To be completed

References

The LAMBDA/NLAMBDA and Spread/Nospread distinctions are covered in 5.1 of the IRM. Sections 5.1.0 thru 5.1.4 are most relevent. Also look at Section 5.1.7.

Functional arguments and FUNCTION are covered in Section 5.4 of the IRM.

Most control structures (e.g., COND, AND, OR, SELECTQ, PROG, PROG1, PROGN, etc.) are covered in Chapter 4 of the IRM.

RPT, RPTQ, MAPC and MAPCAR are covered in Section 5.3 of the IRM.