F.G.H.
6/18/85

LispCourse #37: Recursion; Organizing Large Programs

Rercursion ÿ The ultimate Lisp control structure

Recursion is THE control structure in Lisp.

Reasons:

Recursion is the right way to organize many procedures!

The structure of Lisp encourages you to think recursively.

The structure of Lisp makes it very convenient to write recursive functions.

To review: Recursion a control structure in which the definition of a function includes a function call to itself.

The basic idea: many problems can be solved by combining the solutions to two or more smaller problems, each of the same nature as the larger problem.

For example: to count the number of atoms in a list, you can combine by addition the number of atoms in the CAR of the list and the number of atoms in the CDR of the list.

This works fine as long as the CAR and CDR are both non-empty lists, because you can then (recursively) apply the same procedure to count their atoms.

But you must have an alternative procedure when the CAR and/or CDR is a non-list or an empty list because the count atoms in list procedure won't work in these cases.

All recursive functions have the same basic underlying structure:

```
(DEFINEQ
    (RecursiveFunction (LAMBDA (Args...)
        (COND
            ((Is Args... a terminating case?) (Process terminating case))
            (T (Call combining function with args:
                (Call RecursiveFunction using first "part" of Args...)
                (Call RercursiveFunction using second "part" of Args...)
                ...
                (Call RercursiveFunction using last "part" of Args...)))))
```

The CountAtoms procedure from LispCourse #5, page 12 implements the count-atoms-in-a-list procedure outlined in the example above and follows exactly this standard recursive function format:

```
(DEFINEQ
  (CountAtoms (LAMBDA (List)
    (COND
        ((NULL List) 0)                          [Terminating case]
        ((LITATOM List) 1)                       [Terminating case]
        (T (PLUS                                 [Combining Function]
            (CountAtoms (CAR List))              [Recursion: 1st part of List]
            (CountAtoms (CDR List)))))))         [Recursion: 2nd part of List]
```

As another example, consider the function EVAL (LispCourse #34, page 5):

```
    (DEFINEQ
      (EVAL  (LAMBDA (SExpr)
        (COND
            ((LITATOM SExpr)
                (LookupValue SExpr )             [Terminating case]
            ((NLISTP SExpr) SExpr))              [Terminating case]
            (T (APPLY (CAR SExpr)                [Combining Function]
                (FOR Item in (CDR SExpr)
                    DO (EVAL Item))))))
                [Recursive call on each part of SExpr]
```

EFS: Trace EVAL and APPLY during the evaluation of *(CountAtoms '((A B) D E))*. You will see that recursion works in Lisp because each time CountAtoms is APPLYed to another sub-list, a new stack frame is created and *List* is rebound to the value of that sub-list.

Recursion works in Lisp because

1) the stack is a expandable data structure that holds the results of incomplete calculations while (recursive) sub-calculations are going on

2) the process of rebinding variables on the stack allows you to apply a given function to a new set of arguments while the processing of a prior call to that **same** function is still incomplete

Recursion can be ***indirect***, e.g., a FunctionA calls FunctionB which in turn calls FunctionA again.

The Lisp evaluator (LispCourse #34, pages 5 & 6) is an execellent example:

EVAL recursively calls EVAL, but it also calls APPLY.

APPLY in turn calls EVAL.

So EVAL calls APPLY which calls EVAL.

For example on a list:

EVAL calls EVAL on each item in the CDR of the list, then calls APPLY using the CAR and the evaluated arguments.

APPLY in turn calls EVAL on each list in the function definition of the CAR of the original list.

Types of Recursion

Single Recursion (and tail recursion)

Singly recursive functions are recursive functions that call themselves only once during each application of the function.

Note the a call to itself may appear several times in the function defintion (e.g., in several clauses of a COND statement), but only one of these should be evaluated during each call to APPLY.

Examples:

```
(DEFINEQ
    (MEMBER (Thing List)
        (* * Is Thing EQUAL to any item in List?)
        (COND
            ((NULL List) NIL)
            ((EQUAL Thing (CAR List)) T)
            (T (MEMBER Thing (CDR List))))))

(DEFINEQ
    (LENGTH (List)
        (* * Return the number of items in List)
        (COND
            ((NULL List) 0)
            (T (ADD1 (LENGTH (CDR List)))))))

(DEFINEQ
    (REVERSE (List)
        (* * Make a copy of List with the items in reverse
        order)
        (COND
            ((NULL List) NIL)
            (T (APPEND (REVERSE (CDR List))
            (CAR List))))))
```

Note that these three function are all singly recursive, but differ in what they do to the result returned by the recursive function call.

Each call to MEMBER just returns the value of the recursive function call (or NIL if List is NULL).

Each call to LENGTH returns 1 plus the result returned by the recursive function call.

Each call to REVERSE returns a computation based on the value returned by the recursive function call and a computation based on the value of the main argument.

Because of these differences, an all-knowing Lisp evaluator would have to maintain different amount of state about each of these recursive functions during evaluation. In particular,

For MEMBER, the evaluator could thow out all information about a function call (i.e., its stack frame) once it made its recursive call.

This is because there is no information about the state of the computation to be maintained once the recursive call is made. The value of the highest level call is the value of the lowest-level function call, with no modifications.

For LENGTH and REVERSE, the evaluator needs to maintain information about each recursive call until the recursion is complete.

This is because when the recursive call is made, the computation at the current level is incomplete. In LENGTH, for example, the recursive computation is being done in the middle of an ADD1 evaluation. Information about the status of ADD1 needs to be maintained until the recursive call to LENGTH has returned a value.

MEMBER is a ***tail recursive*** function, LENGTH and REVERSE are not.

A ***tail recursive*** function is recursive function whose value depends only on the value returned by a recursive call or some value that is computed directly without a recursive call.

Tail recursive function are important because a good Lisp evaluator can eliminate the unnecessary stack frames during the computation. making these computations very efficient. (See comparison of iteration and recursion below).

Recursive functions that are not tail recursive need to maintain their intermediate state on the stack ,and therefore can be relatively expensive to compute.

Double (or more) Recursion

Doubly recursive functions are functions that during each application call themselves two (or more) times.

CountAtoms (above) is a perfect example: for each invocation (where List is a LISTP) it recurses once on its CAR *and* once on its CDR.

As a second example, the following function is like MEMBER but rather than just looking at the top-level elements in a list, it descends into each sub-list looking for a sub-item that might be EQUAL to its first argument:

```
(DEFINEQ
    (MEMBER*
        (LAMBDA (Thing List)
            (COND
                ((NULL List) NIL)
                ((NLISTP List) (EQUAL Thing List))
                (T (OR
                    (MEMBER* Thing (CAR List))
                    (MEMBER* Thing (CDR List))))))))
```

In general, doubly recursive functions cannot be tail recursive because the evaluator always needs to maintain state (i.e., the result of) about the result of the first recursive call while evaluating the second recursive call.

The exception to this rule is doubly recursive functions that are evaluated for side-effect only and do not return a useful value. In this case, computation can be done on the returned value of a recursive call, so no state has to be maintained.

Double recursion is often called tree recursion because it is used to traverse tree structures. See Homework for examples.

Iteration versus Recursion

Iteration and recursion are in many ways similar control structures.

In fact, any iterative procedure can be *automatically* converted into an equivalent recursive procedure.

For example:

```
(LET ((Sum 0))
        (FOR Item IN List DO (SETQ Sum (PLUS Sum Item)))
        Sum)
```

can be converted to

```
(DEFINEQ
        (Sum (LAMBDA (List)
                (COND
                        ((NULL List) 0)
                        (T (PLUS (CAR List) (Sum (CDR List)))))))
```

In contrast, not every recursive procedure can be converted into an equivalent iterative procedure:

CountAtoms, for example, has no iterative equivalent.

Tail recursive procedures are important, however, because they can always be rewritten in an equivalent iterative form.

For example, MEMBER can be written as:

```
(FOR Item IN List WHEN (EQUAL Thing Item) DO
(RETURN T))
```

It is interesting that the Sum function just above is NOT tail recursive, but *is* equivalent to an iterative procedure. The reason is that the Sum function can be rewritten as the following equivalent tail recursive procedure, which in turn can be rewritten as an iterative procedure:

```
(DEFINEQ
    (Sum2 (LAMBDA (List Total)
        (COND
            ((NULL List) Total)
            (T  (Sum2 (CDR List) (PLUS (CAR List)
            Total))))))
```

The ability to rewrite recursive procedures as iterative procedure is important because iterative computations are in general much more efficient than recursive computations.

This is because recursive function calls require the evaluator to maintain a set of stack frames containing information about partially completed computations. This stack grows as the recursion gets deeper.

In contrast, iterative procedures require only a fixed set of state variables to be maintained (e.g., the iterative variable). The number of these state variables does not increase with the number of iterations.

It is important to not that while recursive ***procedures*** cannot always be rewritten as iterative ***procedures***, it is often the case that a given ***problem*** can be solved using a recursive procedure *or* an iterative procedure that is not strictly equivalent to the recursive procedure.

For example:

The REVERSE function above cannot be written iteratively because it is not tail recursive.

But we can write two slightly different procdures that reverse the order of a list:

```
(DEFINEQ (ReverseTR (LAMBDA (List ResultSoFar)
    (COND
        ((NULL List) ResultSoFar)
        (T (ReverseTR
            (CDR List)
            (CONS (CAR List) ResultSoFar))))))

(DEFINEQ (ReverseIt (LAMBDA (List)
    (LET (Result)
        (FOR Item IN List
            DO (SETQ Result (CONS Item
            Result)))
        Result))))
```

Untimately, the choice between iteration and recursion is one of programming style and programming ease.

Some problems are best thought about recursively. In this case, it easiest and clearest to write recursive functions.

Other problem are best thought of iteratively. In this case, iterative functions are probably best.

The only exception is when efficiency considerations are important. In this case, iterative solutions, if possible, are probably called for.

Organizing Large Programs by Object and Operation

Many large programming projects have the structure that there are a number of different types of objects in the world and a number of operations that can be applied to any of these types of objects.

For example, in arithmetic programming the objects are integers, real numbers, complex numbers, etc. There are also a few standard operations, addition, subtraction, multiplication and division.

In a text editor, the objects are characters, words, lines, and paragraphs. The operations might be insert, delete, replace, transpose, etc.

It is helpful to illustrate the structure of these programming projects using a table of objects and operations.

For example:

Operations \ Objects	Integer	Real	Complex
Addition	AddInts	AddReals	...
Subtraction	SubInts	...	
Multiplication	...		
Division			

Each cell of this table defines the need for a function to carry out the designated operation on the designated type of object.

For example, the cell in the first row and first column of the table above indicates the need for a function to add integers.

Similarly, the cell in the first row, second column defines the need for a function that adds real numbers.

And so on.

When organizing the program to handle the given objects and operations, you have several choices:

1) You can organize your functions by the rows in the table, i.e., by operation.

2) You can organize your functions by the columns in the table, i.e., by object type.

3) You can organize your functions using the whole table.

Organization by Operation

If you organize by operation, you would write a single operation for each operation. This function would then determine the type of its arguments and then call the appropriate function to do the work.

For example:

The addition function might look like:

```
(DEFINEQ
  (ADD
    (LAMBDA (N1 N2)
      (COND
        ((AND (FIXP N1)(FIXP N2))
          (IntegerAdd N1 N2))
        ((AND (FLOATP N1)(FLOATP
        N2))
          (RealAdd N1 N2))
        ((AND (ComplexP N1)(ComplexP
        N2))
          (ComplexAdd N1 N2))
        (T (ERROR "Unknown argument
        types or argument types different"
        (LIST N1 N2)))))))
```

The subtraction. multiplication, and division functions would have similar structures.

Once the generic operations functions were written, you could write all further functions in terms of the generic operators.

For example, the following function would add the items in a list of integers or a list of reals or a list of complex numbers:

```
(DEFINEQ (Sum  (LAMBDA (List)
    ((NULL List) 0)
    (T (ADD (CAR List)(Sum (CDR List)))))))
```

Organization by Type of Object

Alternatively, you could organize your program by object type.

In this case, you might create a RECORD or DATATYPE for each object type containing each a function for each operation to be carried out on that object type.

For example, the following RECORD would be used for defining the arithmetic object types:

```
(RECORD ArithObjectType (PredicateFn AddFn SubFn MultFn
DivFn))
```

Integers would then be defined by:

```
(create ArithObjectType
        PredicateFn _ (FUNCTION FIXP)
        AddFn _ (FUNCTION IntegerAdd)
        SubFn _ (FUNCTION IntegerSubtraction)
        MultFn _ (FUNCTION IntegerMultiplication)
        DivFn _ (FUNCTION IntegerDivision))
```

You would then maintain a list of all of the arithmetic object types in the system.

Once this list was created, you could write a generic ADD function as follows:

```
(DEFINEQ  (ADD (LAMBDA (N1 N2)
    (LET (TypeRecord)
        (FOR Type IN ListOfSystemTypes
            WHEN (AND (APPLY* (fetch PredicateFn of
            Type) N1)
                    (APPLY* (fetch PredicateFn of
                    Type) N2))
                DO (RETURN Type)))
        (COND
            (TypeRecord
                (APPLY* (fetch AddFn of TypeRecord) N1
                N2))
            (T
                (ERROR "Unknown argument types or
                argument types different" (LIST N1 N2))
```

Organization using the table: Data-directed programming

Alternatively, you could organize your program by operation and object type.

In this case, you would create a table of functions indexed by object type and operations.

For example:

```
(SETQ Table
     ((Integer Add IntegerAdd)
      (Integer Subtract IntegerSubtract)
      ...
      (Real Add RealAdd)
      ...
      (Complex Divide ComplexDivide)))
```

You could then create a generic function called *Operate* that carried out a given operation on a given set of arguments.

Operate would look like:

```
(DEFINEQ (Operate (LAMBDA (Operation N1 N2)
     (LET (TypeN1 TypeN2 Function)
          (* * Get the type of the arguments)
          (SETQ TypeN1 (GetType N1))
          (SETQ TypeN2 (GetType N2))
          (COND
               ((OR
                    (NEQ TypeN1 TypeN2)
                    (NULL TypeN1)
                    (NULL TypeN2))
                (ERROR ...))
          (* * Lookup the function in the table)
          (SETQ Function (FOR Item IN Table
                              WHEN (AND (EQ (CAR
                              Item) TypeN1)
```

```
                                        (EQ (CADR
                                        Item)
                                        Operation))
                                    DO (RETURN (CADDR
                                    Item))))
                        (* * Apply the function)
                        (COND
                            (Function (APPLY* Function N1 N2))
                            (T (ERROR ...)))))))
```

The Sum function could then be written as:

```
(DEFINEQ (Sum  (LAMBDA (List)
    ((NULL List) 0)
    (T (Operate 'Add (CAR List)(Sum (CDR List)))))))
```

Organization by Object Instance: Object-oriented programming

All of the preceding organizations looked at the type of an object in order to determine what functions to use on that object.

An alternative method would be to allow each data object to carry around with it the functions that are necessary to operate on that object.

In this case, each data object would be a RECORD or DATATYPE of the following form:

```
(RECORD ArithObject (TypeName Value AddFn SubFn MultFn DivFn))
```

You could then write an ADD function as follows:

```
(DEFINEQ (ADD (LAMBDA (N1 N2)
    (COND
        ((NEQ (fetch (ArithObject TypeName) of N1)
            (fetch (ArithObject TypeName) of N2))
         (ERROR ...)))
```

```
(APPLY* (fetch (ArithObject AddFn) of N1)
        (fetch (ArithObject Value) of N1)
        (fetch (ArithObject Value) of N2)))))
```

You would still need one function indexed by object type, the function that creates a new object of that type. This would create a new object with its own attached functions. The attached functions could be supplied specially for each individual object or could be the same for all objects of a given object type.

For example, the following might be a function that creates the integer object:

```
(DEFINEQ (MakeInt (LAMBDA (Value AddFn SubFn MultFn
DivFn)
    (create ArithObject
            TypeName _ 'Integer
            Value _ Value
            AddFn _ (OR AddFn
                        (FUNCTION DefaultIntegerAddFn))
            SubFn _ (OR SubFn
                        (FUNCTION DefaultIntegerSubFn))
            MultFn _ (OR MultFn
                        (FUNCTION
                        DefaultIntegerMultFn))
            DivFn _ (OR DivFn
                        (FUNCTION
                        DefaultIntegerDivFn))))))
```

The integer of value 5, with default functions would then be created using the function call: *(MakeInt 5)*.

Similarly, an integer of value 7 with a special AddFn would be created using: *(MakeInt 7 (FUNCTION MyAddFn))*

Much of the Interlisp system is programmed in this style.

For example, each window in the system is a DATATYPE of the following form:

```
(DATATYPE WINDOW (... CLOSEFN SHRINKFN MOVEFN ...))
```

Each window in the system has attached to it the functions that specify how to close it, how to shrink it, how to move it, etc.

When the window is created, default functions are placed in these fields in the WINDOW datatype unless otherwise specified by the user.

The function CLOSEW (SHRINKW, MOVEW etc) in Interlisp is implemented as follows:

```
(DEFINEQ (CLOSEW (LAMBDA (Window)
        (APPLY* (fetch (WINDOW CLOSEFN) of
        Window) Window))))
```

At any time, the user can change the behavior of a window by changing its CLOSEFN, MOVEFN, SHRINKFN or whatever.

Choosing from among these organizations

Choosing among these organizations is largely a matter of programming style and the type of problem you are dealing with.

If the program has a fixed number of objects, but may increase in the number of operations, then an operation-based organization may be best since each added operation means adding a single new function.

If the program has a fixed number of operations but an increasing number of object types, then an object type-based organization might be best since each added object type would involve only a new object-type record or datatype.

If there is a variability in both object types and operations, table-based data-directed programming might be best.

Finally, if there are many specialized individual objects that need to be treated differently from the rest of the objects of their basic type (as is the case for windows), then an object-oriented organization would be the most efficient.

References

Recursion

Winston & Horn, Chapter 4

Touretzky, Chapter 8

Program Organization

Abelson and Sussman, Section 2.3