F.G.H.
4/30/85

LispCourse #23: What is Programming; Basic Lisp Revisited

What is Programming?

Procedures and Data: The Semantics of Programming

A program is basically a description of a sequence of ***actions*** (i.e., a ***procedure***) to be carried out on a set of ***objects***. In the computer world, the objects are different types of "information" and are thus called ***data***.

For example: (PLUS 3 4) is a trivial Lisp program. The procedure PLUS takes the action of adding two numbers. The objects or data are the numbers, in this case 3 and 4.

Writing a program involves specifying two things:

1) a description of the procedure to be followed
2) a description of the data to be used.

The abstract description of the procedural component of a program is called its **control structure**. The abstract descriptions of the data components of a program are called its **data structures**.

Programming is all about control structures and data structures. Learning to program means learning to build combinations of control and data structures that accomplish target tasks effectively and efficiently.

For example:

Assume that the goal is to program up a simple database mapping people in ISL to serial numbers of their machines.

The first task is to determine a data structure for the database, e.g., a list of lists where each sublist begins with a name the second and subsequent items are serial numbers for that person’s machine(s). Note that I would have to further specify the data structure for the name and serial numbers (e.g., they could be atoms or they could be lists,

depending on what operations I would want to perorm on them).

The second task is to write the Lisp procedures to create, access, and modify the chosen data structure. This may feed back to modify the data structure ÿ for example, you might discover that the procedures would be much simpler if the second element of each person's list were the number of machines that the person had.

Lisp is a language for specifying the control and data structures of a program. Fortran, Basic, C, Pascal, Mesa, etc. are alternative languages for doing the same thing.

The same basic concepts of control structure and data structure appear in all of these languages. They differ only in how they express various control and data structures. Control/data structures that are easy to express in Lisp may be painfully hard to express in Fortran, and vice versa.

Lisp is unusual among these languages in that it blurs the distinction between procedures and data, both syntactically and semantically. A list [e.g., (PLUS 2 3)] that is a piece of data to one Lisp program, may be a description of a procedure that is executed by another Lisp program. Languages like Fortran and Pascal make a much cleaner distinction between procedure and data.

Blurring the distinction between procedure and data can be good or bad, depending on the task you are trying to program.

In this course, we'll start out making clear distinctions between procedure and data in our Lisp programming style. Later, we may look at some of the advantages of bluring over this distinction a bit.

Procedure versus Process: Running a Program

A program is a ***procedure***, i.e., a description of a sequence of actions to be carried out.

For the program to be useful, this description must be transformed into the actual sequence of actions being described. This transformation is commonly called "running the program".

The sequence of actual actions that take place when a program runs is called a ***process***.

The entity that transforms the procedure description (i.e., program) into a running process is an ***interpreter***.

> At the bottom level, the ultimate interpreter of any program is the hardware of the computer on which the program is running. But in most languages (as in Lisp) , there is another program (an interpreter) that is responsible for transforming the program from the higher level-language into the running process. This interpreter is the entity responsible for "understanding" the language and carrying out the requested actions.
>
> > Note that the interpreter is itself a program being interpreted (directly or indirectly) by the machine hardware.

Understanding Lisp requires understanding the Lisp interpreter, the program that transforms Lisp code into a running process. The nice thing about Lisp is that the interpreter is itself a Lisp program, making it easy to understand (and modify) if you know Lisp.

> Seems a bit circular ÿ but its really a process of decomposition where you break Lisp down into simpler and simpler actions until you get to those actions the machine hardware can execute directly.

So, theoretically, there are three components to understand about programming: control structures, data structures, and interpreters. Unfortunately, this analysis skips all of the pragmatic aspects of programming in the real world.

Correctness, Efficiency, Maintainability, & Adaptability Quickly and with Minimal Effort: The Pragmatics of Programming

A good program is:

> **Correct** ÿ it correctly accomplishes the task it is intended to

Efficient ÿ it make efficient use of the available resources; in particular, it works as quickly as possible using as few of the computer's resources (memory, disk space) as possible.

Maintainable ÿ it should be easy to debug and make minor changes to; in particular, it should be easy enough to understand that someone other than the original programmer can do the maintenance

Adaptable ÿ it should be easy to make large changes to the original program

Good programs must also be written **quickly and with minimal effort**: i.e., using as little programming time and as little programming effort as possible. A program that takes 1000 person-years to write will never get written.

Programming is a constant trade-off between these five goals since it is nearly impossible to satisfy all five simultaneously.

Interlisp takes a definite stand on which of these are important. In particular, correctness and efficiency are discarded in favor rapid-prototyping, i.e., programming quickly and with minimal effort.

Efficiency and correctness are achievable in Interlisp, but only extremely careful programming and close attention to issues of efficiency and exact correctness.

Maintainablilty and adaptablility in Interlisp (and in most modern programing languages) are a matter of good programming style. If you follow the rules of good programming, then your programs will be maintainable and adaptable.

In this course we will focus in particular on these "rules of good programming", covering in detail the notions of abstraction and modularity. In the later part of the course, we will briefly consider the tools for writing efficient and correct programs in Interlisp.

Reviewing the Basics of Lisp Prgramming

The READ-EVAL-PRINT Loop

Lisp is in a continual loop in the Exec window

Read user input
Evaluate user input
Print result of evaluation

All work in Lisp is done in the Evaluation phase of this loop!

Basic Syntactic Building Blocks: Atoms and Lists

One defining feature of Lisp is that its syntax is trivial.

There are two basic building blocks of Lisp: ***Atoms*** and ***Lists***.

Together Atoms and Lists are known as ***S-expressions*** ÿ (almost) everything in Lisp is an S-expression.

An *atom* is a symbol represented by one or more alphnumeric characters. Atoms are the "words" of Lisp; they provide a way in which to reference the actions and objects in Lisp world.

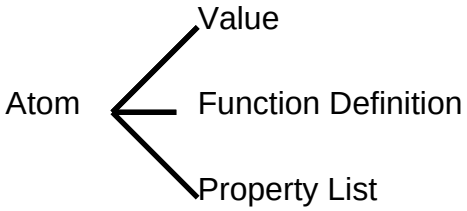
Examples: Sam, FOO, 123, A2233, VeryLongAtomName, Dashed-Atom.

A *list* is a "(", followed by zero or more atoms or lists, followed by a ")". Lists are the "sentences" of Lisp; they provide the structure to glue atoms together to make statements or to represent things.

Examples: (A B C); (A (FOO BAR) C); (SETQ A 5); (CAR (CAR A)).

Facts about atoms:

Every atom can reference three things: a *value*, a *function definition*, and a *property list*.



An atom's *value* is some other Lisp object; e.g., another atom, a list, a window, a process, etc.

Function definitions are discussed below.

An atom's *property list* is simply a list with a special format: *(Prop1 Value1 Prop2 Value2 ... PropN ValueN)*, where each *PropI* is an atom and each *ValueI* is some other Lisp object; e.g., another atom, a list, a window, a process, etc.

When evaluated by the Lisp evaluator, an atom evaluates to its value.

Numeric atoms are atoms whose name consists of digits only. The value of a numeric atom is the atom itself.

Non-numeric atoms are called *litatoms*.

T and *NIL* are special atoms. The value of T is T and the value of NIL is NIL. NIL is also a list, in particular, the empty list ( ). NIL is the only thing that is both a list and an atom.

Facts about lists:

The first item in a list is called its *CAR*.

The rest of the list, after the CAR is removed, is called its *CDR*.

The operation of the Lisp evaluator w.r.t. lists is discussed in the next section.

Thats all there is of Lisp syntax: all Lisp programs are build up from combinations of atoms and lists.

Forms and the Lisp Interpreter

The basic building block for procedures in Lisp is the ***form*** or ***function call***. A form is simply a list structure, the CAR of which is the name of a function (i.e., an atom that references a function definition) and the CDR of which is a list of the arguments passed to the function.

Example: (IDIFFERENCE 5 2), IDIFFERENCE is the function name and (5 2) is a list of the arguments to be passed to the IDIFFERENCE function.

The Lisp evaluator evaluates forms in the following manner:

The CAR of the list is assumed to be the name of a function.

First, each element in the CDR of the list is evaluated, resulting in a list of evaluated arguments.

Then, the function named by the CAR is ***applied*** to the resulting evaluated argument list.

What it means to ***apply*** a function to an argument list is discussed in the following section.

Special Forms & QUOTE

There are some functions which are special in the sense that their arguments are NOT evaluated before they are executed. They are executed using their unevaluated arguments. [Called NLambda functions]

QUOTE is such a special form. QUOTE simply returns its unevaluated argument. QUOTE can be abbreviated by a ’. (QUOTE A) is equivalent to ’A.

QUOTE is used to prevent evaluation where it is not required.

Example:

```
31_ (SETQ A 4)
4
32_ (SETQ B 5)
5
33_ (COPYFILE A B)
4: File not found
5: File not found
NIL
34_ (COPYFILE ’A ’B)
{PHYLUM}<HALASZ>B;5
```

EVAL

EVAL is a function which evaluates its arguments. Note that means that the arguments of EVAL are evaluated TWICE!

33_ (SETQ Harp 'Viola)
Viola
34_ (SETQ Viola 'Tuba)
Tuba
35_ Harp
Viola
36_ Viola
Tuba
37 _ (EVAL Harp) [means evaluate the value of Harp]
Tuba

Lisp Control Structures

Basically, a Lisp procedure is a list of forms to be evaluated one after the other in sequence.

Example, the following might be considered a Lisp procedure:

(SETQ A 55)
(SETQ B 66)
(IPLUS A B)

Note a sequence of actions is a very limited control structure. Using this control structure, it is impossible to specify an action to be done only under certain circumstances or to be repeated a variable number of times.

Interlisp has many special forms that provide more interesting control structures. We covered two: COND and Iterative Loops.

COND

COND is a special form that implements a conditional control structure. COND has the following form:

(COND
(*Test1 Consequents1*)

(*Test2 Consequents2*)
(*Test2 Consequents2*)
...
(*TestN ConsequentsN*))

Each *Testi* is an S-expression (usually a predicate) that evaluates to NIL or non-NIL. Each *Consequentsi* consists of 0 or more S-expressions.

COND works as follows:

Test1 is evaluated.

If it returns a non-NIL value, each S-expression in Consequents1 is evaluated in turn, and then the COND is exitted. The value of the COND is the value of the last S-expression in Consequents1.

If it returns a NIL value, go on to Test2

Test2 is evaluated.

If it returns a non-NIL value, each S-expression in Consequents2 is evaluated in turn, and then the COND is exitted. The value of the COND is the value of the last S-expression in Consequents2.

If it returns a NIL value, go on to Test3.

...

TestN is evaluated.

If it returns a non-NIL value, each S-expression in ConsequentsN is evaluated in turn, and then the COND is exitted. The value of the COND is the value of the last S-expression in ConsequentsN.

If it returns a NIL value, the COND is exitted with a value of NIL.

Example:

Return X if X is an atom, NIL otherwise.

```
(COND
    ((LITATOM X)  X)
    ((NUMBERP X) X)))))
```

Iterative control structures: The FOR Loop and its cousins

Iteration is a control structure that makes it possible to repeat the same operation on each element in a sequence (list) of things.

"To iterate" means "to repeat".

The FOR Loop

The major iterative construct in Interlisp is the FOR loop.

[Footnote*:* The FOR loop construct is not a standard part of most Lisps. The iterative control structure is available in all Lisps, it just has a different syntax than the Interlisp FOR loop.]

The FOR loop has the following form:

(FOR *variable* IN *list* DO *operation*)

Note: FOR is a special form; the elements of the CDR are not evaluated automatically.

IN and DO are keywords.

Variable is unevaluated. It is the name of an atom to be used as the local variable in the iteration.

List is evaluated. It is a S-expression whose value is a list.

Operation consists of 0 or more S-expressions to be evaluated. Ordinarily, these S-expression make some use of the value of *the* atom in the *variable* role.

FOR works as follows:

The *variable* is bound (i.e., temporarily SETQed) to the CAR of the *list*. The S-expressions in *operation* are then evaluated.

Then the *variable* is bound to the second item in the *list* and the the S-expressions in *operation* are again evaluated.

Then the *variable* is bound to the third item in the *list* and the S-expressions in *operation* are evaluated.

...

The *variable* is bound to the last item in the *list* and the S-expressions in *operation* are evaluated.

The FOR loop returns NIL.

Example:

(FOR Window in (OPENWINDOWS) DO (CLOSEW Window))

Alternative FOR loops

1. DO versus COLLECT

The DO keyword can be replaced by the COLLECT keyword in the FOR loop. In this case, on every iteration the value of the last S-expression in operation will be saved. The FOR loop will then return a list of these values (in order) instead of returning NIL.

Example:

4_ (FOR Item IN '((A B)(C D)(E F)) COLLECT (CAR Item)(CDR Item))
((B)(D)(F))

2. "IN *list*" versus "FROM *n* TO *m* BY *k*"

FOR also allows for iteration over a sequence of numbers. To do this, replace the "IN *list*" construction with the construction "FROM *n* TO *m* BY *k*", where n, m, and k evaluate to numbers.

Note: The "BY *k*" is optional and defaults to "BY 1" if it is not specified.

Example:

5_ (FOR N FROM 1 TO 10 COLLECT (PLUS N 6))
(7 8 9 10 11 12 13 14 15 16)

WHILE and UNTIL loops: Alternatives to FOR

WHILE and UNTIL loops allow repetative operations without an explicit sequence.

WHILE has the form:

(WHILE *predicate* {DO, COLLECT} *operations*)

Note: WHILE is a special form.

Predicate is evaluated. It is an S-expression that evaluates to NIL or non-NIL.

Operations is 0 or more S-expression to be evaluated.

The WHILE loop repeatedly evaluates *operations* as long as *predicate* evaluates to non-NIL.

DO versus COLLECT works exactly as in FOR. If DO is used, then the WHILE loop always returns NIL. If COLLECT is used, the WHILE loop returns a list containing, in order, the value of the last S-expression in *operations* from each iteration.

Example:

```
11_ (WHILE (MouseButtonDown) COLLECT
                (WHICHW))
```

UNTIL is similar to while, but it iterates until its *predicate* becomes non-NIL, i.e., as long as its *predicate* is NIL.

"UNTIL *predicate*" is equivalent to "WHILE (NULL *predicate*)".

Defining and Applying Functions

All of the work in Lisp is done when the Lisp evaluator evaluates a form or function call.

Recall that evaluating a form involves applying a ***function*** to a list of evaluated arguments.

A function is simply a sequence of forms that have been packaged into a unit and named.

Once packaged into a function, the sequence of forms can be manipulated as a single entity. The process of treating a sequence of forms as a single entity that carries out a single (but compound) action is known as *procedural abstraction*. We'll have more to say about procedural abstraction later in the course.

In Lisp packaging and naming a sequence of forms is known as *defining a function*. Programming in Lisp consists of defining functions that call other functions you have already defined (or will define before you run the program).

Defining functions

DEFINEQ is a special form that allows you to define functions in Lisp. It has the form:

(DEFINEQ *Defn1 Defn2 ...*)

Each *Defni* is the name and definition of a function and has the form: **(*FunctionName FunctionDefn*).**

FunctionName is any atom.

FunctionDefn has the form:

(LAMBDA *ParameterList FunctionBody*)

LAMBDA is a keyword indicating that the list is function a definition. Just put it there. It is an historical remnent from Church's Lambda calculus, on which Lisp was originally built.

ParameterList is a list of the parameters (arguments) for the function.

FunctionBody is 1 or more Lisp forms.

Example:

```
(DEFINEQ
    (SumOfSquares                [FunctionName]
        (LAMBDA    [Keyword]
            (X Y)  [ParameterList]
                (PLUS (TIMES X X)(TIMES Y
                Y)  [FunctionBody]
```

Applying functions

Recall that the final step in evaluating a form involves ***applying*** the function named by the CAR to the list of evaluated arguments.

APPLY works as follows:

1. The value of each parameter in the function definition is (temporarily) set to the corresponding element of the evaluated argument list.
2. All of the forms of the function body are evaluated in turn using the normal rules of Lisp evaluation.
3. The value returned by the function is the value of the last (only) form in the function body.
4. All parameters (i.e., L) are set to back to their original value.

The process of setting the values of the parameters to the values of the arguments is known as *binding* the local variables. Note that binding affects the parameters only locally within the function. The value of the same atom outside of the function is unaffected.

Lisp Interpreter: A combination of EVAL and APPLY

EVAL and APPLY combine to produce the Lisp interpreter, as the following example illustrates:

Evaluating the form (SumOfSquares (TIMES 2 3) 2)

1. Each item in the CDR of the form is **evaluated** in turn:

1.1) *(TIMES 2 1)* evaluates to 6 (by recursive call to the Lisp evaluation of a form).

1.2) *2* evaluates to itself.

2. The SumOfSquares function is **applied** to the list (6 2).

2.1) X is bound to 6 and Y is bound to 2.

2.2) (PLUS (TIMES X X)(TIMES Y Y)) is evaluated by recursive call to the Lisp evaluation of forms, resulting in the value 40.

2.3) X and Y are rebound to their previous values (if any).

2.4) SumOfSquares is exited returning, 40.

3. The value of the form (40) is printed.

Lists: the primary Lisp data strucuture

The primary data structure in Lisp is the list. When you want to erepresent an object or piece of data in Lisp, the first thing you think of is a list!!!

There are other data structures in Lisp, but lists are the only ones we covered in the earlier part of the course.

Lists can be used to represent almost any data:

Examples:

A person's name might be a list of three atoms: *(First Middle Last)* as in *(Frank Geza Halasz)* or *(John Seely Brown)*.

ISL-People might be represented by a list of names (which are themselves lists): *((John Seely Brown) (Dana ?? Bloomberg)(Thomas P. Moran)(Frank Geza Halasz))*

PARC-Personnel might be a futher aggregation of lists representing each lab. Each lab list would consist of two items, a lab name and a list of lab people: *(PARC (ISL ((John Seely Brown) (Dana ?? Bloomberg)(Thomas P. Moran)(Frank Geza Halasz)))(CSL (Robert ?? Ritchie)(Robert ?? Haggman) ...)(SCL (Adele ?? Goldberg) ...)))*

List Maniupulation Functions

Interlisp has lots and lots of fucntions for creating, maintaining and decomposing these list data structures. Some of those we covered earlier (See LispCourses #2 & #3) were:

List Decomposition Functions

CAR ž returns the first element of the list

CDR ž returns the list minus the first element

CxxR (where x=A or D) ž compisitions of CAR and CDR.

NTH ž returns the list *tail* starting at the Nth element

LAST ž returns the list tail containing only the last element

List Composition Functions

APPEND ž returns the concatenation of two or more lists

LIST ž creates a list consisting of its arguments

CONS ž (CONS Arg1 Arg2) returns a list with Arg1 as its CAR and Arg2 as its CDR.

List Manipulation Functions

REVERSE ž returns list with reverse order of top-level elements

LENGTH ž returns number of top-level elements in list

List Formats

Each of the lists given in as examples above has a format that determines the meaning of the elements of the list. Programs that might use these lists would have to have built into them the necessary procedures for making use of these formats.

For example, the name list has three elements representing the first, middle and last names. If a program needs the first name of a person, the programmer would have to know enough about the formatto take the CAR of the person's name list.

The examples given above were ad-hoc list formats. Much of Lisp programming involves creating (and documenting) such ad-hoc list formats. But, there are some special list formats that are commonly used as data structures in Lisp:

An **ASSOC list** has the following form: ((*key1 data1*)(*key2 data2*)(*key3 data3*)(*key4 data4*) ... (*keyn datan*)), where each *keyi* is an atom and each *datai* consists of 0 or more S-expressions.

The function **ASSOC** is used to retrieve items from an ASSOC list. ASSOC takes two arguments, a key and an ASSOC list. It returns the first *key-data* pair in the list whose key is equal to the key argument.

A **Prop list** has the form: (*prop1 value1 prop2 value2 prop3 value3 ... propn valuen*), where each *propi* is an atom and each *valuei* is exactly one S-expression.

The function **LISTGET** retrieves a prop values from a PROP list. LISTGET takes 2 arguments, a prop list and a prop. It returns the value corresponding to prop on the prop list.

LISTPUT adds a prop-value pair to an already existing prop list. Its three arguments are a prop list, a prop, and a value. If the prop is already on the list, it simply updates its value. It returns the value.

Exercise: Learning Data Abstraction

Write a program that manipulates a database of people in ISL, their office numbers and their phone numbers.

First, determine a data structure for your database.

Then write functions for:

creating the database

adding people to the database

getting a list of all the first names in ISL

getting a list of all the last names in ISL

getting a list of all the phone numers in ISL.

(Hint: you will find FOR-COLLECT loops very handy here.)

References

LispCourse Notes #2 thru #6.